PROGRAMMER TO PROGRAMMER™

ASP 3.0

Programmer's Reference

Richard Anderson
Matthew Gibbs, Marco Gr
Simon Robinson, John Schenken, Kevin Williams

wrox
PROGRAMMER TO PROGRAMMER™

ASP 3.0 Programmer's Reference

Richard Anderson, Dan Denault, Brian Francis,
Matthew Gibbs, Marco Gregorini, Alex Homer, Craig McQueen,
Simon Robinson, John Schenken, Kevin Williams

Wrox Press Ltd. ®

ASP 3.0 Programmer's Reference

Reprinted June 2000

Published by Wrox Press Ltd
Arden House, 1102 Warwick Road, Acock's Green, Birmingham B27 6BH, UK
Printed in USA
ISBN 1-861003-23-4

Table of Contents

Extending ASP 615

Performance and Security 1007

1

What is ASP?

Since its introduction, the use of Microsoft's **Active Server Pages**, or ASP, has grown rapidly. Many programmers consider it *the* tool for dynamic, easily maintainable web content. The real power of ASP derives firstly from the fact that the HTML for the page is only generated when the specific page is requested by the user, and secondly from the fact that it is browser-independent, since what is sent to the browser is usually purely HTML (although it can also include client-side code), rather than relying on the browser to support a particular language or application.

ASP enables us to tailor our web pages to the specific requirements of our users and their browser type, as well as our own needs. It allows us to interact with the user, which helps to keep our site interesting and up-to-date. Although it is not the first technology to offer dynamic page creation, it is one of the fastest and most powerful. It is indicative of the impact that ASP has made that it has now got its own imitators.

This book is for primarily intended for developers who need a reference to the many ways in which ASP contributes to the running of web sites and the resources we can utilize to achieve that goal. In order to get the maximum benefit from this book you will need to have some understanding of both ASP and the Web in general. Some knowledge of a scripting language such as VBScript or JScript is assumed.

This is the book for you if you need easy access to the methods and properties which the different components and objects expose, you are looking to expand the scope and functionality of your site, or you are fairly new to ASP and need an overview of what it has to offer. We hope that you benefit from and enjoy this book.

Firstly in this chapter we explore where ASP is derived from, look at some of the essential building blocks of ASP and briefly explore what is new to ASP 3.0. We look at:

- The origins of ASP
- ASP, HTTP, HTML and IIS
- Managing state on the Web
- The role of `global.asa`

- ASP directives
- Virtual applications
- What's new in ASP version 3.0
- What's new in JScript 5.0
- What's new in VBScript 5.0

The Origins of ASP

At the same time that the huge business potential of the Web was being realised, the limitations of HTML and HTTP were also becoming very apparent to developers. The static, stateless nature of HTML pages means that, although they are great for 'on-line brochure' web sites, they do not meet the specific needs and requirements of fast-moving business, building customer loyalty and selling goods and services. Various technologies, including ASP, grew out of the need to create pages with content specific to an individual user.

A simple type of customer interaction is processing information entered by a user on an HTML form. This normally involves either a user searching for information, or alternatively the customer entering personal information that needs to be stored by a business for further processing. In either case, we probably want to communicate with a database, which cannot be done purely using HTML. The initial solution to these, and other applications that are equally problematic with HTML and HTTP, involved reading the user input and programmatically creating a response. The interface which the server exposed to connect HTML and other applications became known as the Common Gateway Interface (CGI), and can be implemented in any language (the most popular being Perl). This approach, however, requires developers to have extensive programming knowledge and is restricted by the need to compile the code. Although Perl and CGI are still valuable tools, several alternatives are now also available, including ASP. This enables sections of script to be embedded in HTML pages. ASP-embedded code can contain logic which inserts content, formats data and carries out actions depending on decisions relating to how a page is requested.

ASP, HTTP, HTML and IIS

HyperText Transfer Protocol (HTTP) is the protocol that handles requests and responses sent between a web server and browser. The HTTP Request is the format of any message sent from the client to a server. It includes the URL of the required resource and information about the client and the platform they are using. The HTTP Response can contain a resource, a redirection to another page or site, an error message, etc.

ASP provides its own `Request` and `Response` objects, which enable us to access the information stored in the HTTP Request message and Response headers respectively. Using these objects we can check for certificates, read and write cookies, and get access to browser information and forms data. We can insert data into the body of the page to be sent to the client, redirect the browser, check if the client is connected, and manage the sending of content so that the client does not wait for too long for long sections of content.

The relationship between ASP and HTML can be described as follows:

Active Server Pages is a technology that allows for the programmatic construction of HTML pages for delivery to the browser.

In other words, with ASP we can write a set of instructions that can be used to generate HTML and other content just before it is delivered. This makes it a good tool for HTML developers, because of its power and flexibility to generate fresher HTML, and ultimately produce more spectacular, interactive, personalized and up-to-date web sites.

But what actually *is* ASP? It's not a conventional programming language in the sense that Pascal and C++ are, although it does make use of existing scripting languages such as VBScript or JScript. It's also not an application in the sense that FrontPage and Word are. The best way to think of ASP is as a technology for building dynamic and interactive web pages.

An alternative way to create dynamic pages is to use client-side scripting. This must be written in a language interpreted by the client browser and hence code generally consists of sections of Javascript embedded in an HTML page. This can programmatically control the layout of the page, how the page reacts to user actions and what is shown on the page. This is all useful but has its limitations. Typical uses for client-side code are, for example, to respond to user actions like clicking their mouse on the page, passing it over certain hotspots, and also checking forms prior to sending them.

Client-side code depends on the browser supporting the scripting language, and can fall over if the language is not supported, or includes code which differs between implementations and language versions. A second limitation is that the code is accessible to the user, which makes it unsuitable for passing, for example, passwords and connection strings.

The alternative then, and this is what ASP relies on, is to include scripts which are processed by the server. These server-side scripts do not depend on the browser or the user's platform executing, as the result returned to the browser is typically in plain HTML (or text, XML etc). However, server-side script is often used in conjunction with client-side code – there's no reason why an ASP page can't contain `<SCRIPT>` sections.

IIS (previously called Internet Information Server, but renamed with IIS5 to **Internet Information Services**) was Microsoft's answer to dynamic page creation by servers. Originally IIS 1.0 consisted of a fairly standard setup with CGI support and an interface to allow more efficient execution of compiled applications written in languages like C and C++. It provided additional features to access the input and output streams. This interface is called the **Internet Server Application Programming Interface**, or **ISAPI**.

The ASP scripting engine still uses ISAPI to connect to IIS5. It runs in-process with the server. This means that it shares the same memory space with the server and can get direct access to values in that memory. This does mean that if the application fails it can cause the server to fail, but makes for a very efficient and fast process and generally gives ASP the edge over other technologies.

What does ASP Code Look Like?

When a web author writes an ASP page, it is likely to be composed of a combination of three types of syntax – some ASP, some HTML tags, and some pure text. The following table summarizes these ingredients, their purpose, and their appearance:

Type	Purpose	Interpreter	Hallmarks
Text	This is hard-coded information to be shown to the user	The viewer's browser shows the text	Simple ASCII text.
HTML tags	This consists of instructions to the browser about how to format text and display images	The viewer's browser interprets the tags to format the text	Each tag within < > delimiters. Most HTML tags come in pairs (an open tag and a close tag), e.g. `<TABLE>`, `</TABLE>`.
ASP statements	This consists of instructions to the web server running ASP about how to create portions of the page to be sent out	The web server's DLL `asp.dll` performs the ASP commands	Each ASP section contained within `<%` `%>` delimiters. ASP statements support features such as variables, decision trees, cyclical repetitions etc.

The file containing these constituent parts of the ASP page is saved with an `.asp` extension.

It's not too hard to distinguish the different elements of the ASP page. Anything that falls between the `<%` and `%>` markers is **ASP script**, and will be processed on the web server by the ASP script engine.

Lets take a look at an example and at the same time demonstrate one of the keys to ASP's success – how easy it is to get started. For example, consider the few lines of code below:

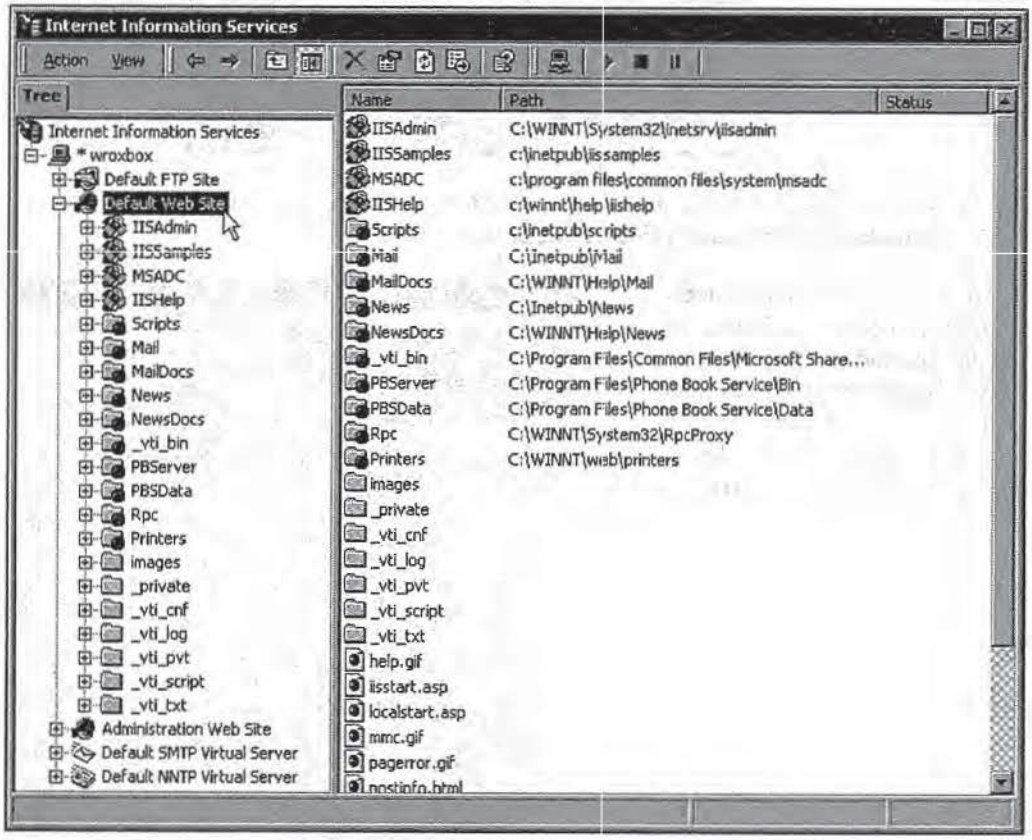
```
<HTML>
<P>This date/time is now : <%= Now() %> </P>
</HTML>
```

The content of the resulting web page depends on the HTML that is generated by the ASP code. In this particular example, the effect of the script code is to generate HTML for the time and date that the page is requested, and then to make a decision (based on the situation) on what text will be sent to the browser as part of the HTML stream.

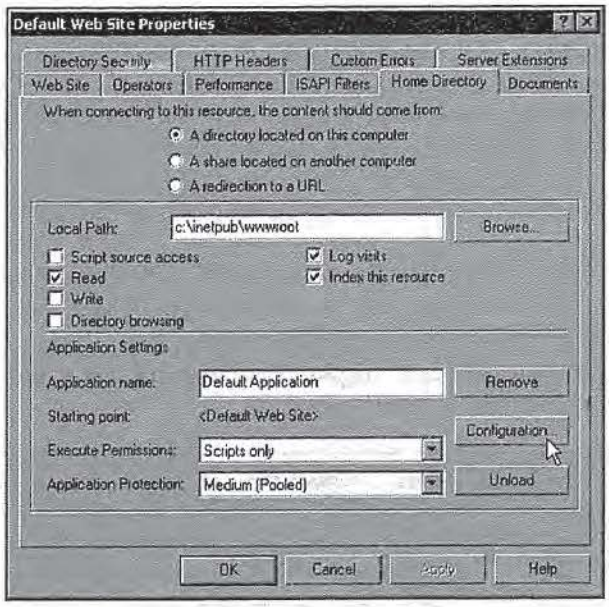
How does ASP Work

ASP works with a single DLL called `asp.dll` (or alternatively the ASP scripting engine). This is installed by default into your `WinNT\System32\Inetsrv` directory. This DLL is responsible for taking an ASP page (indicated by the `.asp` file extension) and parsing it for any server-side script content. The script is passed to the appropriate scripting engine, to interpret for example the VBScript or JScript. The results of executing the script are combined with any text and HTML in the ASP page and the completed page is then sent back the client browser via the web server.

To see this, open the Internet Services Manager from the Administrative Tools section of your Start menu (for Windows 2000 Server version – or go to it via Administrative Tools in your Control Panel with Windows 2000 Professional). This runs the Microsoft Management Console (MMC) to display the entire Internet Information Services tree for IIS, which looks something like this:

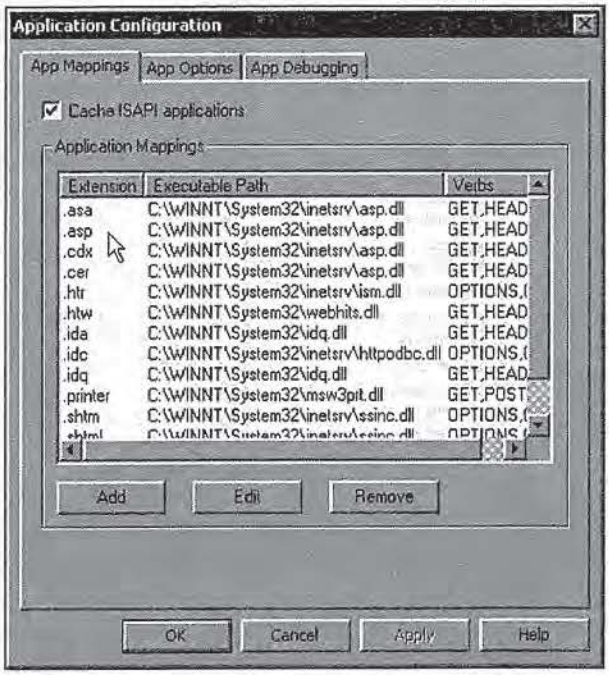


Right-click on the Default Web Site entry and select Properties, then the Home Directory page:



In the lower half of the page there is a name for the application and the Execute Permissions and Application Protection settings.

Then, click the Configuration button to open the Application Configuration dialog. In the App Mappings tab, you can see the way that IIS links each type of file (using the file extension) with a specific DLL:



Any pages that have the `.asp` file extension are sent to the `asp.dll` for processing; you can see our `global.asa` page (which we discuss later in this chapter) is also mapped to the asp DLL, as well as several others. Pages with file extensions that are *not* mapped to a DLL, for example `.html` and `.htm` for HTML pages, and `.xml` for XML files, are simply loaded from disk and sent directly to the client.

You might like to have a look at the other file types in this page. Pages with `.ida`, `.idc`, *and* `.idq` *file extensions are sent to the DLL* `httpodbc.dll` *for processing. As you can guess from its name, it uses ODBC (discussed later in this book) to execute a SQL statement that returns a set of records for inclusion in the page. Likewise, the* `.shtm`, `.shtml` *and* `.stm` *file extensions are mapped to a DLL named* `ssinc.dll`*. These file types are traditionally used for files that require* ***server-side include (SSI)*** *processing.*

While you have the Application Configuration and Properties dialogs open, you might want to briefly explore it (just don't change any settings for the moment, unless you're sure that you know what you're doing, the defaults usually suffice!).

Processing an ASP File

When `asp.dll` receives an ASP page, it converts it to an output suitable for the server to send to the client. It deals with any script marked for its attention, evaluating it, and sending the result on to the server. It does this by first checking the page to see if contains any ASP code. If it does not find any, it informs IIS to send the page to the client. A new feature of Windows 2000 means that this is done with no marked performance penalty.

When ASP receives a page that does contain server-side scripting, it parses it line by line and each section of script is passed to the approproate scripting engine for compiling and execution. The result of this is inserted into any content which does not require server intervention by ASP and the whole is passed on to the client.

To make this more efficient, ASP caches the compiled code so that it does not need to be compiled again unless the source is changed. The result of this is that subsequent requests for the specific page are returned more quickly as the compilation stage is bypassed.

Including Separate Script Files

An additional feature is the ability to include separate files that contain script code. This allows us to write generic functions and both encourages and greatly simplifies code re-use. It also allows us to encapsulate processes which depend on the setup of our system and our server, and so may change.

Scripting Performance Issues

Web servers generally have plenty of spare processor cycles available (except on the busiest of sites) because the main task they have is loading pages from disk and sending them to the client. Therefore, each page request results in the processor waiting for the disk to respond. These spare cycles mean that ASP scripts can usually be executed with very little overall hit on performance. To add to this, as most page requests will be for pages where a compiled version of the script code is available, only the execution of this script needs to take place.

Of course, as the number of requests, and hence the server load, increases, the effect of having to parse and execute each ASP page takes its toll. It's wise, therefore, to squeeze as much performance as possible from the ASP interpreter. Here are some useful tips:

Avoid Mixing Scripting Languages on the Same Page

Using both scripting languages on any one page has an effect on the execution order of the code. This will mean both scripting engines will be loaded by the ASP DLL, increasing memory usage and the time taken to process the request.

Avoid Excessive Context Switching Between Script and Other Content

Having sections of ASP interspersed within other content can have a significant effect on the time it takes to process the page request. Every time a section of script ends, control is passed back to IIS (and vice versa) and this will have an impact on performance. An alternative to this is to use the `Response.Write` method (rather than using `<% = ... %>`) and this is recommended for any but small sections of code.

Build a Separate Component

For any complex processes, consider building a separate componant to install on the server. This will be far more efficient than instantiating and interpreting ASP script code. This will become even more important with the next version of ASP (currently called ASP+) in which almost everything is done using COM+.

Managing State on the Web

In a normal single-user program, such as when we build an executable application (an `.exe` file for example) using C++, Delphi etc., we take for granted the fact that we can declare a global (or `Public`) variable and then access it from anywhere in our code. All the time the application is running, the value remains valid and accessible.

The ability to hold values in memory, and relate specific values to specific users, provides **state**. You can think of it as representing the values and context of the applications' and users' internal variables throughout the life of the application.

When we create web-based **applications**, we often need be able to provide an individual *state* for each user. This might be as simple as remembering their name, or as complex as storing object references or recordsets that are different for each user. If we can't do that, we can't reasonably expect to do anything that requires more than *one* ASP page, as the variables and other references in that page are all destroyed when the page is finished executing. When the user requests the next page, we've lost all the information that they've already provided.

It is also useful to be able to store values that are global to *all* users. An obvious example is a web-style visitor counter. There's not much point in giving each user their own counter, because they usually want to see the total number of visitors, not just the number of times that they have visited. The number of visitors needs to be stored with **application-level state**, rather than **user-level state**.

With ASP we need a way of storing state information, otherwise variables and references within a page are destroyed when that page has finished execution. One of the ways of providing state between page requests and site visits is through **cookies**, which are sent along with each page request to the domain for which the cookie is valid. ASP uses a cookie to provide the concept of a user **session**, which we interact with through the ASP `Session` object.

A new and separate `Session` object is created for each individual visitor when they first access an ASP page on the server. A session identifier number is allocated to the session, and a cookie containing a specially encrypted version of the session identifier is sent to the client. The `Path` of the cookie (see the previous chapter for a description of cookie properties) is set to the path of the root of the ASP application running on our server. This will be the root of the Default Web site (i.e. `"/"`) or the root directory of the ASP application containing the page they request. No `Expires` value is provided in the cookie, so it will expire when the browser is closed.

Every time that this user accesses an ASP page, ASP looks for this cookie, named `ASPSESSIONID`*xxxxxxxx*, where each *x* is an alphabetic character. If found, it can be used to connect the visitor with their current `Session`, which is held in memory on the server.

This cookie doesn't appear in the `Request.Cookies` or `Response.Cookies` collections. ASP hides it from us, but it's still there on the browser and ASP looks for it with each ASP page request.

If the client browser doesn't accept or support cookies, each new page request forces a new session to be created, thus state cannot be maintained without cookies.

As a warning you might note that if the browser does not support, or is set to reject, cookies this function will not be available. What do we do then? An alternative is to have server-side cookies. The information is stored on the server and matched to each user. We can then persistently store information on that user for a length of time determined by ourselves. This just leaves matching the user to their information and this can be done with the use of logins. Each time that a user enters the domain of our server they can be asked to log in with their name and password. We can then make our site available to them according to their specific needs and preferences.

In addition we can restrict access to resources so that we can expose varying levels of access to the general public, clients of the company or subscribers, staff or site administrators. Now we are starting to have a level of control over our site which in any other application we might take for granted.

The Role of global.asa

All ASP applications can contain a file named `global.asa`, placed in the root directory of the application and which applies to all sub-directories of this. The `global.asa` file in the root directory of the entire web site (`Inetpub\wwwroot`) defines the whole site as being part of the **Default ASP application**.

The `global.asa` file can contain code that instantiates objects and creates and sets the values of variables that will be available in either **Application-level** or **Session-level** scope. Object instances can be created using the `Server.CreateObject` method or an `<OBJECT>` element.

Creating Object Instances

If an `<OBJECT>` element is used, the `SCOPE` attribute can be set to `"Application"` or `"Session"`, and the object is then created in the appropriate context:

```
<!-- Declare ASPCounter component with application-level scope -->
<OBJECT ID="ASPCounter" RUNAT="Server" SCOPE="Application"
        PROGID="MSWC.Counters">
</OBJECT>

<!-- Declare ASPContentLink component with session-level scope -->
<OBJECT ID="ASPContentLink" RUNAT="Server" SCOPE="Session"
        PROGID="MSWC.NextLink">
</OBJECT>
...
```

The remainder of the `global.asa` file can contain ASP script that defines event handlers that run when the application or a user session starts or ends. Using VBScript this looks like:

```
...
<SCRIPT LANGUAGE="VBScript" RUNAT="Server">

Sub Application_OnStart()
   'Code here is executed when the application starts
End Sub

Sub Application_OnEnd()
   'Code here is executed when the application ends
End Sub

Sub Session_OnStart()
   'Code here is executed when a user session starts
End Sub

Sub Session_OnEnd()
   'Code here is executed when a user session ends
End Sub

</SCRIPT>
```

Or using JScript:

```
...
<SCRIPT LANGUAGE=JScript RUNAT=Server>

function Application_OnStart() {
   // Code here is executed when the application starts
}

function Application_OnEnd() {
   // Code here is executed when the application ends
}

function Session_OnStart() {
   // Code here is executed when a user session starts
}

function Session_OnEnd() {
   // Code here is executed when a user session ends
}

</SCRIPT>
```

Within the `OnStart` event handlers, script code can be used to instantiate objects with the `Server.CreateObject` method. This code creates an instance of the Ad Rotator component (see Chapter 15):

```
Set Session("ASPAdRotator") = Server.CreateObject("MSWC.AdRotator")
```

If placed in the `Application_OnStart` event handler, the object will have application-level scope. If placed in the `Session_OnStart` event handler, the object will have session-level scope, i.e. each visitor will have a separate instance of the object. The `Server.CreateObject` method is covered in detail in Chapter 7.

Referencing Object Type Libraries

Many objects and components provide enumerated and other constants which the various methods take as their paramters. This allows us to use the constant's name in place of its value. These constants can be referenced through the `METADATA` directive. Using the `METADATA` directive we can specify a type-library which ASP will then load when the page is executed. The syntax for this is shown below:

```
<!-- METADATA TYPE="TypeLib"
              FILE="path_and_name_of_file" | UUID="type_library_uuid"
              [VERSION="major_version_number.minor_version_number"]
              LCID="locale_id" -->
```

where:

- `path_and_name_of_file` is the absolute physical path to a type library file (`.tlb`) or ActiveX DLL. If this is not provided the `type_library_uuid` must be specified.
- `type_library_uuid` is the unique identifier for the type library. Either this or the `path_and_name_of_file` parameter must be provided.
- `major_version_number.minor_version_number` (optional) defines the version of the component required. If this version is not found the most recent version is used.
- `locale_id` (optional) is the locale identifier to be used. If a type library with this locale is not found the default locale for the machine (defined during setup) will be used.

For example, this code makes the intrinsic ADO pre-defined constants available in an ASP page:

```
<!-- METADATA TYPE="TypeLib"
              FILE="c:\Program Files\Common Files\System\ado\msado15.dll"
-->
```

In order to maintain backward compatibility the file name `msado15.dll` is used for later (i.e. ADO 2.5) versions of the ADO component.

If ASP is unable to load the type library, it will return an error and halt execution of the page. The possible error values are:

Error	Description
ASP 0222	'Invalid type library specification'.
ASP 0223	'Type library not found'.
ASP 0224	'Type library cannot be loaded'.
ASP 0225	'Type library cannot be wrapped' (i.e. ASP cannot create a type library wrapper object from the type library specified).

Web Applications

We have used the term web application a number of times rather loosely, to indicate something that isn't really a web site, but isn't a 'traditional' application (an `.exe` file for example) either. We can think of a web application as a set of web pages and other resources, such as COM+ objects, that are designed to carry out some task.

> *A COM object is an instance of a COM component, which should be thought of as a compiled piece of code that can provide a service to the system, not just a single application. (COM objects are discussed further in Chapter 2, ASP, Windows 2000 and Windows DNA.)*

When IIS and ASP are installed in Windows 2000, a Default Web Site is created. This is configured as an ASP application, which involves several settings in the Properties dialog, that we looked at earlier. The `global.asa` is used to determine the way that this default application behaves.

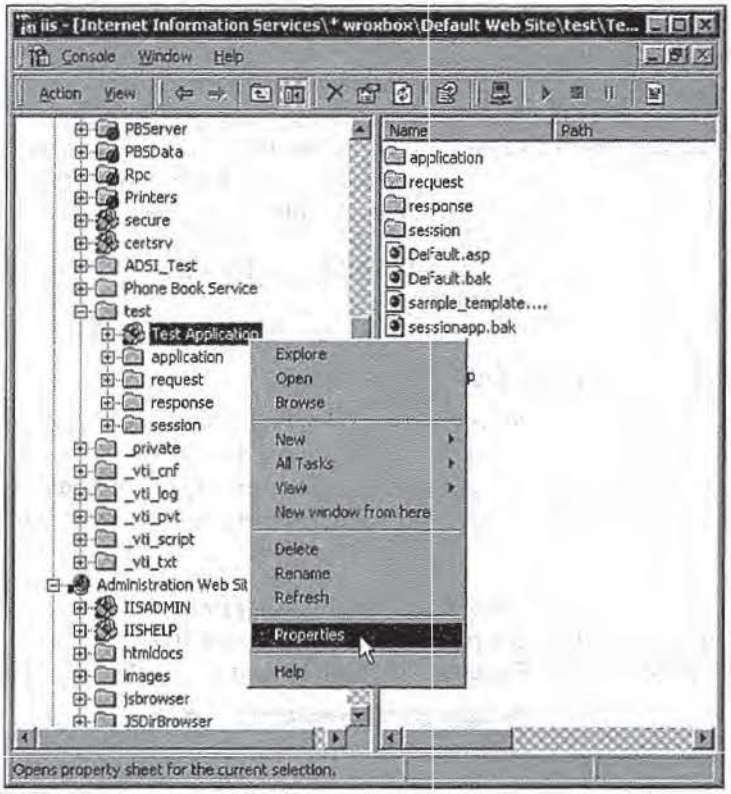
Virtual Applications

In addition to the default web application, ASP virtual applications can be created in any subdirectory of the web site. All sub-directories will then be part of this virtual application. Now, because the directory is itself within the default application for the site, this means that it will share the global space created by the default `Application` object. Any variables stored in the default application are available within the application; however, if an ASP page in the virtual application overwites a global value, the original value is maintained for the root application. This offers some protection to the server and other applications running alongside.

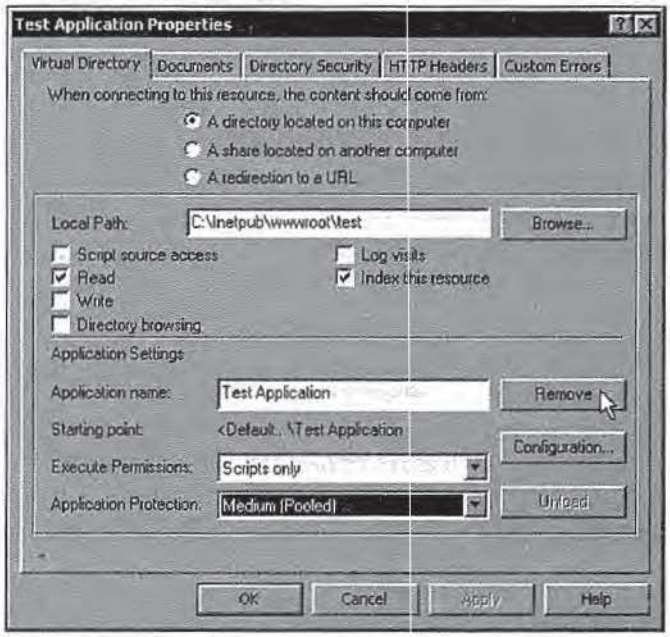
Creating ASP Virtual Applications

In Internet Services Manager, right-click on the directory in which you wish to create the new virtual application, select New, then Virtual Directory. This starts the New Virtual Directory Wizard, which steps through the settings required. This includes the name (or **alias**) for the new virtual application. Combined with the path of the directory selected in Internet Services Manager, this will become the URL of the application. To convert an existing directory into an application with the same name as the directory, select the directory containing the one you want to convert and use the directory name in the Virtual Directory Alias page of the wizard.

The wizard also allows you to specify the path that contains the content (pages) for the application. You can click Browse to select an existing directory. This is the directory that the new virtual application will point to. The final step allows you to select the access permissions, with the default being Read and Run Scripts. These settings can be changed later if required. The wizard then creates the new application, and marks it in Internet Services Manager with an 'open box' icon:



Right-click the new application and select **Properties** to see the settings that the wizard has chosen. The **Local Path**, access permissions, and **Application Settings** can be changed here if required. You'll also see a **Remove** button, which we can use to remove the virtual application:



Clicking the Remove button doesn't actually remove the entry in Internet Services Manager. Instead, it converts the existing virtual application into a **virtual directory**. It will have a 'folder' icon with a 'globe' on it, indicating that this is a *redirection* to another folder on disk. It is accessed in the same way as the virtual application from which it was created (i.e. using the same URL), but does not act as an application. In other words, it doesn't support its own `Application` object but inherits the one for the default web site, or for another application within this directory's parent directories.

To delete a virtual application, select Delete from the right-click shortcut menu for the application in Internet Services Manager.

Virtual Application Configuration

Virtual applications provide control and management of objects and components that are instantiated in pages within that application's directories. The settings for a virtual application provide control over whether objects are created in the memory space of the web server, or separately in shared or individual out-of-process instances of `DLLHost.dll`.

The Properties dialog in Internet Services Manager provides these settings. At the bottom of the Home Directory page of the Properties dialog for a virtual application are two combo boxes marked Execute Permissions and Application Protection:

Application Protection and Execution Settings

The Execute Permissions options are:

Execute Permission	Description
None	No scripts or executables can be run in this virtual application. In effect, this provides a quick and easy way to disable an application if required.
Scripts Only	Allows only script files, such as ASP, IDC or others to run in this virtual application. Executables cannot be run.
Scripts and Executables	Allows any script or executable to run within this virtual application.

While the Execute Permissions options control the type of execution that can take place in the virtual application, the Application Protection options affect the way that executables and components are run. The available options are:

Application Protection	Description
Low (IIS Process)	All application executables and components for ASP virtual applications with this setting are run in the process (i.e. the memory space) of the web server executable (`Inetinfo.exe`). Hence the web server is at risk if any one of the executables or components should fail. This provides the fastest and least resource-intensive application execution option.
Medium (Pooled)	(Default) All application executables and components from all ASP virtual applications with this setting are run in the process (i.e. the memory space) of a single shared instance of `DLLHost.exe`. This protects the web server executable (`Inetinfo.exe`) from the risk of any one of the executables or components failing. However, one failed executable or component can cause the `DLLHost.exe` process to fail, and with it all the other hosted executables and components.
High (Isolated)	All application executables and components for an ASP virtual application with this setting are run in the process (i.e. the memory space) of a single instance of `DLLHost.exe`, but each ASP application has its own instance of `DLLHost.exe` that is exclusive to that application. This protects the web server executable (`Inetinfo.exe`) from the risk of any one of the executables or components failing, and protects the virtual application from risk if an executable or component from another virtual application should fail. Microsoft suggests that a maximum of ten isolated virtual applications should be hosted on any one web server.

Microsoft recommends a configuration where mission-critical applications run in their own processes, i.e. **High (Isolated)**, and all remaining applications in a shared, pooled process, i.e. **Medium (Pooled)**.

Threading Issues and Object Scope

One other factor that affects the performance of instantiated objects and components is the threading model that they use. This also controls the scope in which they will perform successfully. There are five different threading models:

- **Single-threaded** components allow only one process to access the component at a time, and so each must wait in turn for the component to become available. Single-threaded components should **never** be used in ASP.

- **Apartment-threaded** components allow multiple instances of an object to be created, with each user getting their own instance. Object instances cannot be shared amongst processes, however. Apartment-threaded components are suitable for use in ASP with certain limitations as described in the table below.
- **Free-threaded** components allow multiple processes to access them concurrently, so a single instance can service more than one process. However, access is slower than with apartment-threaded objects as each access has to cross a process boundary. Free-threaded objects are suitable for use in ASP pages.
- **Both-threaded** components can act as though they are either apartment-threaded or free-threaded, depending on the context of the calling application. Both-threaded objects are suitable for use in ASP pages.
- **Neutral-threaded** components (new in Windows 2000 with COM+) allow multiple instances of the object to be created like apartment-threaded components. However, they do not limit each instance to always working in the same process, and so can be shared amongst requests. Neutral-threaded components are the best choice for ASP applications, although few tools are currently able to create this type of component.

> **This is only an overview of the different component types. Threading issues are covered further in Chapter 41, Optimizing ASP Performance. For an exhaustive technical discussion of threading issues see Beginning ASP Components from Wrox (ISBN 1-861002-88-2).**

Object instances can be created in three different levels of scope:

- **Application-level** scope means that the object will be available to all pages within that virtual application (or the default web site). One instance of the object will service all requests from all users. For this reason, only both-threaded or neutral-threaded components should be used in application-level scope. However, if possible, avoid using any components at application-level scope at all as this always risks becoming a performance limitation.
- **Session-level** scope means that one object instance will service all requests from a single user within their ASP session. Both-threaded or neutral-threaded components work well at session-level scope, because they do not tie the session to a single process thread, as do apartment-threaded objects. Again, if possible, avoid using any components at session-level scope unless it is absolutely necessary.
- **Page-level** scope means that the object is created and destroyed within a single ASP page. While this seems to be inefficient, the COM+ Component Services within Windows 2000 are specially designed to make this fast and provide minimum use of resources. Objects can be pooled and/or recreated very quickly. With the exception of single-threaded components, any threading model is acceptable at page-level scope. However, apartment-threaded objects generally provide the best performance here.

ASP Directives

For each page that we put together we have several options which we can set which affect the way that the server processes it. A processing **directive** is always the first line of the ASP Page and is delimited by `<%@...%>`. The outside section you may recognize as the standard way of informing `asp.dll` that inside it there is content pertinent to it. The additional `@` sign denotes that it is the processing directive. This may contain all or any of the following keywords; if none are required this line can be omitted:

Processing Directives	Description
`CODEPAGE-"code-page"`	This defines the character set for this page. The code page is the numeric value of the character set. This value may differ between locales and languages to support both additional and alternative characters.
`ENABLESESSIONSTATE="True |False"`	This value can be set to `False` in which case no session cookie is set to the browser, this effectively disables sessions. The default is `True`. The main reason for doing this is to improve efficiency for pages that do not require state information.
`LANGUAGE="language-name"`	This sets the default language for the page. This does not preclude use of other languages within that page. The default language if this is not specified is VBScript, unless the default for the entire application has been changed.
`LCID="locale-identifier"`	An integer value which uniquely identifies the locale from which the page has been sent. This may affect such things as the currency symbol used.
`TRANSACTION="transaction _type"`	This directive specifies that the page will run under a transaction context. See Chapter 34, Transactions and Message Queuing.

What's New in ASP Version 3.0

If you're already familiar with ASP 2.0, and are looking for a concise list of what has actually changed in version 3.0, you'll find the information below:

Summary of New Features in ASP 3.0

These are the new, or substantially changed and improved, features which have been added to ASP in version 3.0. (Also see Chapter 2, ASP, Windows 2000 and Windows DNA for details of how Windows 2000 improves ASP 3.0.)

Scriptless ASP

ASP is now much faster at processing `.asp` pages that don't contain any script. If you are creating a site or Web application where the files may eventually use ASP, you can assign these files the `.asp` file extensions, regardless of whether they contain server-side script or only static (HTML and text) content.

New Flow Control Capabilities

A new feature to ASP 3.0 is an alternative to the `Response.Redirect` statement. Effectively this sent an instruction to the client browser to load an alternative page. Unfortunately, this is both error-prone and a slow process. In ASP 3.0, two new methods to the server object allow page transfers without browser intervention.

`Server.Transfer` transfers execution to another page, while `Server.Execute` will execute another page then return control to the original one. Inside the new page you can access the original page's context, including all the ASP objects like `Response` and `Request`, but you lose access to page scope variables. If the original page indicates that it is a transaction type in the processor directive (the opening `<%@...%>` element), the transaction context is passed to the new page. If this happens and the second ASP file's transaction flag indicates that transactions are supported or required, then an existing transaction will be used and a new transaction will not be started.

Error Handling and the New ASPError Object

Configurable error handling is now available, by providing a single custom ASP page that is automatically called if an error occurs with the `Server.Transfer` method. In that page, `Server.GetLastError` can be used to return an instance of the new `ASPError` object, which contains more details about the error including the error description and the relevant line number.

Encoded ASP Scripts

ASP script and client-side script can now be encoded using Base64 encryption, and higher levels of encryption are planned for future releases of ASP. (Note that this feature is implemented by the VBScript 5.0 and JScript 5.0 scripting engines, and so requires them.) Encoded scripts are decoded at run time by the script engine, so there's no need for a separate utility. Although not a secure encryption method, it does prevent casual users from browsing or copying scripts.

A New Way to Include Script Files

Rather than using the `<!-- #include ... -->` element to force IIS to server-side include a file containing script code, ASP 3.0 can do the 'including' itself. The `<SCRIPT>` element can be used with `RUNAT="SERVER"` and `SRC="file_path_and_name"` attributes to include files containing script code. The full and relative physical path or virtual path of the file can be used in the SRC attribute:

```
<SCRIPT LANGUAGE="language" RUNAT="SERVER" SRC="path_and_filename">
</SCRIPT>
```

Server Scriptlets

ASP 3.0 supports a powerful new scripting technology called **server scriptlets**. These are XML-format text files that are hosted on the server and become available to ASP as normal COM objects (i.e. Active Server Components). This makes it much easier to implement (or just prototype) your web application's business logic script procedures as reusable components, as well as using them in other COM-compliant programs.

Performance-Enhanced Active Server Components

Many of the Active Server Components that come with ASP have been improved to provide better performance or extra functionality. One example is the new Browser Capabilities component. In addition, there are some new components, such as the XML Parser that allows applications to handle XML formatted data on the server. Closer integration between ADO and XML is also provided (through the new version 2.5 of ADO that ships with Windows 2000), which opens up new opportunities for storing and retrieving data from a data store in XML format.

Performance

A great deal of work has been done to improve performance and scalability of ASP and IIS. This includes self-tuning features in ASP, which detect blocking situations and automatically increase the number of available process threads. ASP now senses when requests that are currently executing are blocked by external resources, and automatically provides more threads to simultaneously execute additional requests and to continue normal processing. If the CPU becomes overloaded, however, ASP reduces the number of available threads, to minimize the thread switching that occurs when too many non-blocking requests are executing simultaneously.

Changes from ASP Version 2.0

These are the features that have been changed or updated from version 2.0.

Buffering is On by Default

ASP has offered optional output buffering for some time. Since IIS 4.0, this has provided much faster script execution, as well as the ability to control the output that is streamed to the browser. In ASP 3.0, this improved performance has been reflected by changing the default setting of the `Response.Buffer` property to `True`, so that buffering is on by default. This means that the final output will be sent to the client only at the completion of processing, or when the script calls the `Response.Flush` or `Response.End` method.

> *Note that you should turn buffering off by setting the `Response.Buffer` property to `False` when sending XML-formatted output to the client to allow the XML parser to start work on it as it is received. You may also want to use `Response.Flush` to send sections of very large pages, so that the user sees some output arrive quickly.*

Changes to Response.IsClientConnected

The `Response.IsClientConnected` property can now be read before any content is sent to the client. In ASP 2.0, this only returned accurate information after at least some content had been sent. This resolves the problem of IIS responding to every client request, even though the client might have moved to another page or site. Also, if the client is no longer connected after three seconds, the complete output that has been created on the server is dumped.

Query Strings with Default Documents

When a user accesses a site without providing the name of the page they require, the default document is sent back to them. However, if a query string is appended to that URL this is now passed to the default page. In previous versions this information was lost. For example, if the default page in a directory that has the URL `http://www.wrox.com/store/` is `default.asp`, then both the following will send the name/value pair `code=1274` to the `default.asp` page:

```
http://www.wrox.com/store/?code=1274
http://www.wrox.com/store/default.asp?code=1274
```

Server-side Include File Security

Server-side include files are often used for sensitive information, such as database connection strings or other access details. In ASP 2.0 specifying the virtual path of a file in a server side include to specify a file bypassed the security checking for the file. In other words the authenticated or anonymous account was not compared with the access control list entries for the file.

In ASP 3.0 on IIS 5.0, these credentials are now checked, and can be used to prevent unauthorized access.

Configurable Entries Moved to the Metabase

In IIS 5.0, the registry entries for `ProcessorThreadMax` and `ErrorsToNTLog` have been moved into the metabase. All configurable parameters for ASP can be modified in the metabase via Active Directory and the Active Directory Service Interface (ADSI).

Behavior of Both-Threaded Objects in Applications

For best performance in ASP, where there are often multiple concurrent requests, components should be **Both-Threaded** (Single Threaded Apartment (STA) and Multi-Threaded Apartment (MTA)) *and* support the COM Free-Threaded Marshaller (FTM). Both-Threaded COM objects that do not support the Free-Threaded Marshaller will fail if stored in the ASP `Application` state object.

Earlier Release of COM Objects

In IIS 5.0, instantiated objects or components are now released earlier. In IIS 4.0, COM objects were only released when ASP finished processing a page. In IIS 5.0, if a COM object does not use the `OnEndPage` method, and the reference count for the object reaches zero, then the object is released before processing completes.

COM Object Security

IIS uses the new **cloaking** feature provided by COM+ so that local server applications instantiated from ASP can run in the security context of the originating client. In previous versions, the security context assigned to the local server COM object depended on the identity of the user who created the instance.

Components Run Out-of-Process By Default

In earlier versions of ASP, all components created within the context of an ASP page ran **in-process** by default, i.e. within the memory space of the web server. In IIS 4.0, the ability to create a virtual application allowed components to be run **out-of-process**. In IIS 5.0 and ASP 3.0, components are now instantiated **out-of-process** by default. This is controlled by the metabase property `AspAllowOutOfProcComponents`, which now has a default value of `1`. Setting it to zero changes the default back to that of IIS 4.0.

To better fine-tune the component performance to Web server protection trade-off, you can now choose from the three options for Application Protection in the Properties dialog for a virtual application; see earlier in this chapter. The recommended configuration is to run mission-critical applications in their own processes – i.e. **High (Isolated)** – and all remaining applications in a shared, pooled process – i.e. **Medium (Pooled)**. It is also possible to set the **Execute Permissions** for the scripts and components that make up each virtual application. The three options are: None, Scripts only, or Scripts and Executables.

What's New in JScript 5.0

The only change to JScript is the long-awaited introduction of proper error handling.

Exception Handling

The Java-style `try` and `catch` constructs are now supported in JScript 5.0. For example:

```
function GetSomeKindOfIndexThingy() {

  try {
    // If an exception occurs during the execution of this
    // block of code, processing of this entire block will
    // be aborted and will resume with the first statement in its
    // associated catch block.
    var objSomething = Server.CreateObject("SomeComponent");
    var intIndex = objSomething.getSomeIndex();
    return intIndex;
  }

  catch (exception) {
    // This code will execute when *any* exception occurs during
    // the execution of this function
    alert('Oh dear, the object didn't expect you to do that');
  }

}
```

The built-in JScript `Error` object has three properties that define the last run-time error. We can use these in a `catch` block to get more information about the error:

```
alert(Error.number);  // Gives the numeric value of the error number
// AND the result with 0xFFFF to get a 'normal' error number in ASP

alert(Error.description);  // Gives an error desciption as a string
```

If you want to throw your own errors, you can raise an error (or **exception**) with a custom **exception object**. However there is no built-in exception object, so you have to define a constructor for one yourself:

```
// Define our own Exception object
function MyException(intNumber, strDescription, strInfo) {
  this.Number = intNumber;              // Set the Number property
  this.Description = strDescription;    // Set the Description property
  this.CustomInfo = strInfo;            // Set some 'information' property
}
```

An object like this can then be used to raise custom exceptions within our pages, by using the `throw` keyword and then examining the type of exception in the `catch` block:

```
function GetSomeKindOfIndexThingy() {
  try {
    var objSomething = Server.CreateObject("SomeComponent");
    var intIndex = objSomething.getSomeIndex();
    if (intIndex == 0) {
      // Create a new MyException object
      theException = new MyException(0x6F1, "Zero index not " +
                                     "permitted", "Index_Err");
      throw theException;
    }
    return intIndex;
  }

  catch (objException) {
    if (objException instanceof MyException) {
      // This is one of our custom exception objects
      if (objException.Category == "Index_Err") {
        alert('Index Error: ' + objException.Description);
      else
        alert('Undefined custom error:' + objException.Description);
      }
      else
        // Not "our" exception, display & raise to next high routine
        alert(Error.Description + ' (' + Error.Number + ')');
        throw exception;
    }
  }
}
```

What's New in VBScript 5.0

The features that are available in ASP include those provided by the scripting engines, which means that improvements there are also available in ASP. The changes to VBScript are as follows.

Using Classes in Script

The full Visual Basic `Class` model is implemented, with the obvious exception of events in ASP server-side scripting. You can create classes within your script, which make their properties and methods available to the remainder of the code in your page. For example:

```
Class MyClass

  'local variable to hold value of HalfValue
  Private m_HalfValue

  'executed to set the HalfValue property
  Public Property Let HalfValue(vData)
    If vData > 0 Then m_HalfValue = vData
  End Property

  'executed to return the HalfValue property
  Public Property Get HalfValue()
    HalfValue = m_HalfValue
  End Property

  'implements the GetResult method
  Public Function GetResult()
    GetResult = m_HalfValue * 2
  End Function

End Class

Set objThis = New MyClass

objThis.HalfValue = 21

Response.Write "Value of HalfValue property is " & _
               objThis.HalfValue & "<BR>"
Response.Write "Result of GetResult method is " & _
               objThis.GetResult & "<BR>"
...
```

This produces the result:

```
Value of HalfValue property is 21
Result of GetResult method is 42
```

The With Construct

The `With` construct is now supported, allowing more compact scripts to be written where the code accesses several properties or methods of one object:

```
...
Set objThis = Server.CreateObject("This.Object")

With objThis
  .Property1 = "This value"
  .Property2 = "Another value"
  TheResult = .SomeMethod
End With
...
```

String Evaluation

The `Eval` function (long available in JavaScript and JScript) is now supported in VBScript 5.0. This allows you to build a string containing script code that evaluates to `True` or `False`, and then execute it to obtain a result:

```
...
datYourBirthday = Request.Form("Birthday")
strScript = "datYourBirthday = Date()"

If Eval(strScript) Then
   Response.Write "Happy Birthday!"
Else
   Response.Write "Have a nice day!"
End If
...
```

Statement Execution

The new `Execute` function allows script code in a string to be executed in much the same way as the `Eval` function, but without returning a result as is usually the case with the `Eval` statement. It can be used to dynamically create procedures that are executed later in the code. For example:

```
...
strCheckBirthday = "Sub CheckBirthday(datYourBirthday)" & vbCrlf _
                 & "  If Eval(datYourBirthday = Date()) Then" & vbCrlf _
                 & "    Response.Write ""Happy Birthday!""" & vbCrlf _
                 & "  Else" & vbCrlf _
                 & "    Response.Write ""Have a nice day!""" & vbCrlf _
                 & "  End If" & vbCrlf _
                 & "End Sub" & vbCrlf
Execute strCheckBirthday
CheckBirthday(Date())
...
```

Either a carriage return (as shown) or a colon character ':' can be used to delimit the individual statements within the string.

Setting Locales

The new `SetLocale` method can be used to change the current locale of the script engine. This enables it to properly display special locale-specific characters, such as those with accents or from a different character set:

```
strCurrentLocale = GetLocale
SetLocale("en-gb")
```

Regular Expressions

VBScript 5.0 now supports regular expressions (again, long available in JavaScript, JScript and other languages). The `RegExp` object is used to create and execute a regular expression. For example:

```
strTarget = "test testing tested attest late start"
Set objRegExp = New RegExp        'create a regular expression

objRegExp.Pattern = "test*"       'set the search pattern
objRegExp.IgnoreCase = False      'set the case sensitivity
objRegExp.Global = True           'set the scope

Set colMatches = objRegExp.Execute(strTarget)   'execute the search

For Each Match in colMatches      'iterate the colMatches collection
  Response.Write "Match found at position " & Match.FirstIndex & "."
  Response.Write "Matched value is '" & Match.Value & "'.<BR>"
Next
```

This produces the result:

```
Match found at position 0. Matched value is 'test'.
Match found at position 5. Matched value is 'test'.
Match found at position 13. Matched value is 'test'.
Match found at position 22. Matched value is 'test'.
```

Setting Event Handlers in Client-side VBScript

While not applying directly to ASP scripting techniques, this new feature is useful when writing client-side VBScript. You can now assign a reference to a function or subroutine to an event dynamically. For example, given a function named `MyFunction()`, you can assign it to a button's `ONCLICK` event using:

```
Function MyFunction()
  ...
  'Function implementation code here
  ...
End Function
...
Set objCmdButton = document.all("cmdButton")
Set objCmdButton.onClick = GetRef("MyFunction")
```

This provides similar functionality to that existing in JavaScript and JScript, where functions can be assigned as properties of an object dynamically.

On Error Goto 0 in VBScript

Although this technique was not documented previously, it does in fact work in existing versions of VBScript (as those of you with a VB background and an inquisitive mind will have already discovered). It is now documented, and can be used to 'turn off' custom error handling in a page after an `On Error Resume Next` has been executed. The result is that any subsequent errors will raise a browser-level or server-level error and the appropriate dialog/response.

Other New Features

A couple of other features have been made available in IIS 5.0.

Distributed Authoring and Versioning (DAV)

This standard, created by the Internet Engineering Task Force (IETF) and now in version 1.0, allows authors in several locations to concurrently build and maintain Web pages and other documents. It is designed to provide upload and download access, and control versions so that the process can be properly managed. Internet Explorer contains features that integrate with DAV in IIS 5.0. However, in the IETF standard, and in the current release of IIS 5.0, the versioning capabilities are not yet implemented.

Referencing Type Libraries

In the past, it has been common practice to use a server-side include file to add constants from a type library (such as scripting objects, ADO, or MSMQ) to an ASP page. This is necessary, as ASP does not create a reference to the type library or component DLL as does, for example, Visual Basic. In IIS 5.0, you no longer need to use include files for constants. Instead, you can access the type library of a component directly using a new HTML comment-style element, placed in the <HEAD> section of the page:

```
<!-- METADATA TYPE="typelib" FILE="c:\WinNT\System32\scrrun.dll" -->
```

This makes all the constants in the specified file available within the current ASP page. (Although this is slated as being new in IIS 5.0, it was working but undocumented in IIS 4.)

FTP Download Restarts

The FTP service now (at last, some would say) provides a restart facility for downloads. If a file download stops part way through – perhaps because of a dropped connection at the user end – it can be resumed from that point. This means that failed file downloads do not require the client to download the entire file all over again.

HTTP Compression

IIS now automatically implements compression of the HTTP data stream for static and dynamically generated files, and caches compressed static files as well. This gives faster response and reduces network loading when communicating with suitably equipped clients.

> *New versions of the scripting engines, JScript 5.5 and VBScript 5.5 are currently available in beta. A description of the key new features of these can be downloaded from our web site.*

Summary

In this chapter, we've very briefly looked at many of the major topics that you need to be aware of when working with ASP 3.0. We've purposely taken the point of view of the experienced web developer, assuming that either you have previous experience of ASP through using an earlier version, or at least you know how the Web works when clients and servers interact.

By now, you should have a good overall view of what ASP 3.0 offers – both in terms of the existing features in earlier versions, and the new features that are available in version 3.0. If you feel that you don't fully understand the concepts of the ASP object model, or the way that context is used to allow access to this object model from other sources, don't worry. Providing you have a broad understanding of the topics we've covered here, you will easily be able to follow the next chapter and the following section, which cover it all in more detail.

2

ASP, Windows 2000 and Windows DNA

In this chapter we focus on Windows 2000, and Windows DNA, and look at how they affect the way we use ASP.

First let's take a look at Windows 2000. This latest release provides us with many exciting new technologies such as Active Directory, along with enhancements to products like IIS and ASP, substantial improvements to core technologies such as COM+, and enhancements to supporting techologies like ADO, where a common theme is improved XML support.

A major change over former Windows operating systems is that all these technologies are more tightly integrated into the Windows 2000 operating system. Some are installable components, but many are fundamentall to the Windows 2000 operating system and are ready for us also to use on any Windows 2000 installation. Many of these features have the potential to *dramatically* enhance the functionality, stability, integration, and performance of our web applications.

For the ASP developer, two obvious changes to Windows 2000 are that it ships with ASP 3.0 and IIS 5.0 as standard. These provide some great new functionality, especially in the area of performance. The new versions represent an evolution in each of these technologies, with no new programming paradigms to learn and many of the improvements being internal. This makes the task of upgrading while continuing to get the best out of our applications, much less onerous.

There are no major new changes to the ASP object model (a new `ASPError` object and a few methods and attributes here and there), so our existing IIS4 applications should continue to work fine in IIS5. You'll also find that the base functionality of IIS isn't radically different from before, although a few names have been changed and some of the dialogs have been streamlined. The fact that, at least on the surface, IIS5 doesn't appear much different from IIS4 is deceptive, however. It is much faster and has many

new features such as server-side page redirection, nested ASP page execution and many new and improved add-on components and user friendly dialogs. These are listed in Chapter 1, What is ASP?

Windows 2000 itself, however, is *significantly* different to Windows NT, and ASP and IIS are able to take advantage of many major changes and improvements in the underlying operating system. Whilst these benefits may not be immediately apparent, the net result is that the web server is now a fundamental service of the OS rather than just an add-on. This is reflected by a new name – **Internet Information Services** (it was previously called Internet Information Server).

IIS 5.0 takes advantage of all the native services provided by Windows 2000 such as transaction services, object pooling, queued components, role-based security, cloaking etc., which now work in a more tightly integrated way. While some of these services were available with Windows NT and with add-ons like MTS, they were not integrated to the OS or COM and so incurred an overhead in terms of performance and programmer sanity. (If you've ever used MTS and forgotten to call `SafeRef` you know what I'm talking about.)

> *MTS is still part of Windows 2000: Now it is an integral part of component services, rather than a separate add-on, and the name MTS has been dropped (although we still refer to it this way in Chapter 34 for convenience).*

I guess it's an obvious statement, but ASP *is* now a pivotal technology used by hundreds of thousands of web sites worldwide today. Based upon the technique of combining HTML with server-side scripting to create web pages dynamically, IIS and ASP provide a significant amount of infrastructure and functionality that is needed to create both simple and advanced intranet/internet/extranet applications. In conjuction with ADO and other proximate techologies, rich user interfaces and functional web applications can easily be created.

We can create a dynamic page with only a few lines of code. Since ASP script has the ability to create and access COM objects, our ASP pages can have staggering power and flexibility to scale, adapt, and develop.

COM is *the* most important feature of Windows 2000 for ASP programmers (now called COM+ to reflect enhancements such as the addition of MTS). COM enables ASP to use ADO and therefore access databases. It also enables our ASP pages to access numerous other components such as the Ad Rotator, and to use features such as transaction support. Even ASP's own object model is a set of COM objects. Moreover, not only is it fundamental to Windows 2000, IIS5 and ASP, but also the forthcoming ASP+ uses COM for just about everything.

Windows 2000

Windows 2000 is a functionally rich operating system that comes with numerous ready-built services and applications. This frees us, as developers, to focus on our application rather than devoting time to the core infrastructures needed to make our applications work.

Windows 2000 allows us to use IIS5 as our web server, Active Directory to share enterprise information, Message Queueing Services and Transaction support. Microsoft's intention is clearly to provide an attractive, all-encompasing operating system, hosting and supporting all types of applications, ranging from those suited to small businesses, up to enterprise-level applications.

OK, by using Microsoft's built-in services and applications, we may be giving up a degree of control (like waiting until the first service pack if we find a serious bug), but typically the tradeoff with the amount of time saved not having to write, develop, and maintain these components ourselves is worthwhile.

So far, Windows 2000 has proved to be a very stable operating system. I would recommend installing it sooner rather than later given my experience of it. (Of course, don't put it on your production servers before testing your required configuration on a test server first!) Microsoft is also already well underway with major updates to IIS and ASP which should be released within 12-18 months. These will extend Windows 2000 with more operating systems services in the shape of COM+, and some very exciting features in ASP 4.0 (ASP+).

Windows DNA

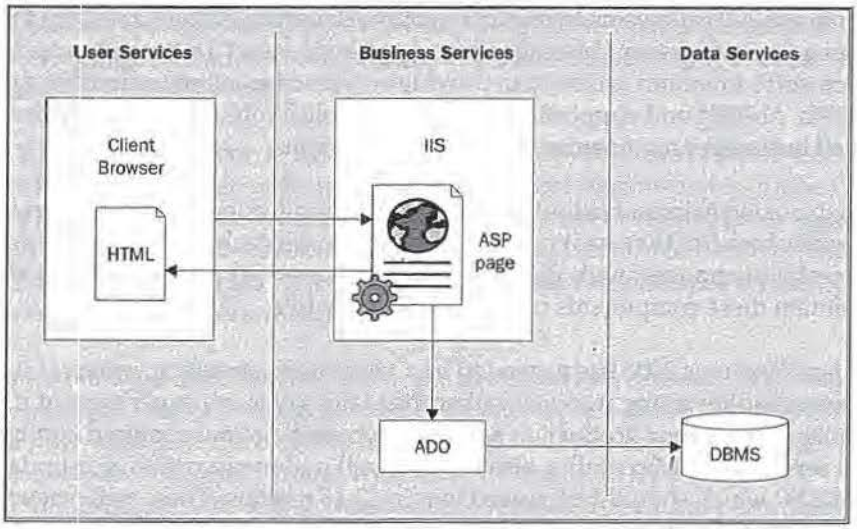
In 1997 Microsoft announced **Microsoft Windows Distributed interNet Architecture (Windows DNA)**, its framework for designing and implementating web-enabled n-tier applications that utilize the power and capabilities of the Windows platform. Windows DNA is a programming model or blueprint for designing and developing distributed component-based applications that use a broad set of products and services.

Microsoft Windows Distributed interNet Architecture 2000 (Windows DNA 2000) is a revised version of Windows DNA which uses the functionality of Windows 2000 and related Microsoft products such as BizTalk Server 2000 (an e-commerce server primarily aimed at business-to-business XML document exchange). Another key extension to Windows DNA 2000 is its use of non-Microsoft-specific technologies such as XML and SOAP (Simple Object Access Protocol). The adoption of such technologies demonstrates Microsoft's acknowledgment of the importance of having Windows applications achieve a good level of interoperability with other platforms.

> **SOAP is an open standard that defines how simple remote procedure calls can be made using XML as the payload, and HTTP as the transport mechanisms. For more information about SOAP see** `http://www.develop.com/soap`.

The Structure of a DNA 2000 Application

The basic premise of Windows DNA 2000 is that applications consist of a number of logical tiers. Focussing on each tier of this model helps us factor the overall architecture of our applications, making it easier to extend and understand it, as shown here:



The three tiers shown in this diagram are:

- **User Services** – This tier is responsible for user interfaces and interactions. As ASP programmers, our user interfaces are typically HTML or DHTML (XML, XSLT, and XPath are covered in Chapter 35 and 36) pages and communicate with the business services tier using HTTP. Alternatively, the user services tier could consist of regular WIN32 applications using DCOM to connect to the business services tier. Windows DNA defines four types of application that exist in this tier depending how they interact with the Internet. These are:
  - **internet-enhanced** applications (WIN32 applications, such as Microsoft Money, that can be used with or without the Internet).
  - **internet-reliant** applications (traditional WIN32 applications that require an internet connection to function).
  - **browser-enhanced** applications (HTML/DHTML-based but typically requiring a specific browser such as IE4/5).
  - **browser-neutral** applications (HTML 3.2 based and therefore suitable for most Netscape or IE users).
- **Business Services** – This tier contains components for carrying out the core processing in our applications, abiding by any business rules or constraints. Typically the business logic is either written into ASP pages for simple applications, or preferably held in COM components that are instantiated and used by our ASP pages (and/or a regular WIN32 application). This use of COM components for our business logic is also good preparation for the forthcoming ASP+, which uses this model.
- **Data Services** – The data services tier contains the data for an application, residing in one or more data sources such as SQL Server or Oracle.

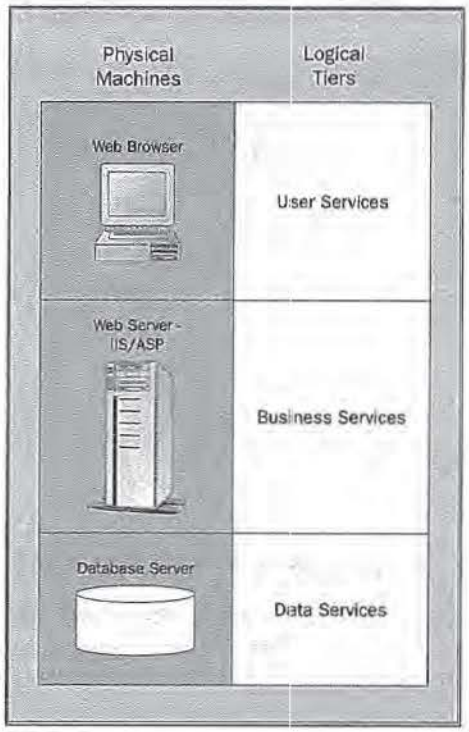
Each logical tier interacts with its neighbouring tiers. It isn't good practice to bypass tiers, for example accessing a data source directly from the user interface, since this sacrifices benefits of n-tier design such as encapsulation and scalability. In this

example our HTML pages would need to have intimate knowledge of the data source structure (and probably the data store type) and hence we'd have to update every single HTML page if we wanted to change our database structure or store. Alternatively if our HTML pages made use of a COM object to interact with the database, we'd only need to update that single point of knowledge, and all our HTML pages referencing it would continue to work. The same rationale applies for scalability: if an additional layer of indirection exists between the user interface and the database, that middle layer (the mediator) could potentially reside on many different machines and possibly cache the database state, therefore reducing hits upon a database server if that's a hot spot in a busy system.

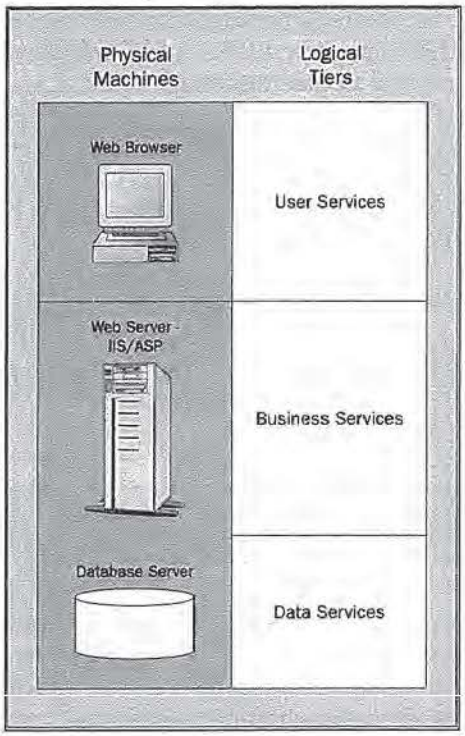
DNA applications can consist of more than just three tiers. We can add additional tiers, maybe sub-dividing tiers into several sub-tiers. For example, the business services tier could be sub-divided into three tiers: one containing ASP pages, one containing COM components that hold our business logic, and one for COM components that encapsulate data access.

Each tier in an n-tier web application provides a layer of indirection that affords us flexibility at the potential cost of some performance.

In addition to the logical tiers which are conceptually important, we also need to think about the **physical tiers** across which our application is deployed in practice. A physical tier is a machine on which the elements defined in a logical tier execute. We can either have a different physical tier (machine) for each of our various logical tiers, or they can co-exist on the same machine. For a large system each tier typically is on a different machine, for load and performance reasons:



For medium sized applications several logical tiers execute on the same physical tier:



Using Windows 2000 DNA for an N-tier Infrastructure

Windows DNA 2000 presumes you are going to use the technologies, tools, and products provided with Windows 2000. The key elements of Windows DNA 2000 are:

- Components Services – COM/COM+
- IIS & Active Server Pages
- Data access (ADO, ADSI)
- Transactions
- Messaging
- XML/Web Services

Component Services – COM/COM+

Component Services are the foundations upon which Windows DNA 2000, Windows 2000, IIS and ASP are built. Microsoft has made it clear that our web applications should be using components if we want them to be usable in the future and in order to

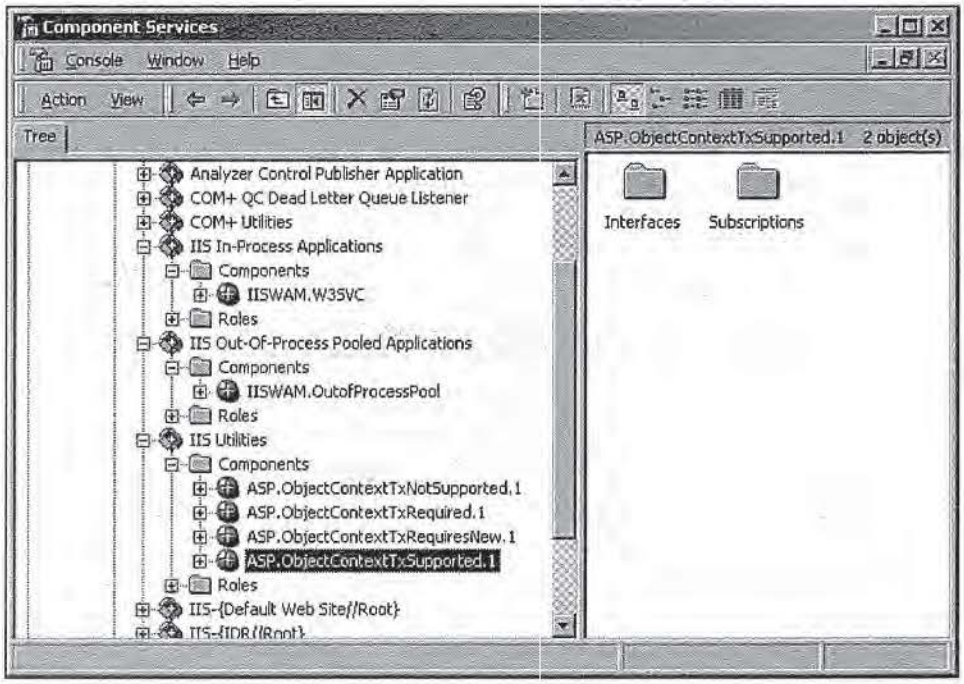
make the most ASP+, the next generation of ASP. With ASP+ you'll be able to write your ASP pages using VB or C/C++. To use these new features without major rewriting of your pages you will need to use components.

> **If you're unfamiliar with using COM and MTS we recommend reading an ASP-centric introduction to COM, such as *'Beginning ASP Components'* published by Wrox Press (ISBN 1-861002-88-2).**

Put simply, COM is a mechanism by which software components (such as ADO) can be written in one programming language, packaged up into a DLL or EXE file, and then used by any another programming language or environment that supports COM (such as ASP). COM is based on a binary specification which means that it is language neutral, as long as the component compiles into an agreed format, it can interact with and be used by, any COM-aware language or application. It's specification defines how a client can consume the functionality of the object, and on the flip side, how an object exposes it's functionality to a client. It is also object-based and builds upon the three object-oriented principals of identity, state and behavior.

COM+ extends COM by integrating the functionality of MTS into the COM runtime. To be more precise, it uses interception and a runtime environment to provide services to our COM+ components dynamically at creation time. MTS did this to provide transaction support, Just-In-Time activation and As-Soon-As-Possible deactivation. COM+ uses the same technique in a far more generic fashion, and provides a far greater number of services.

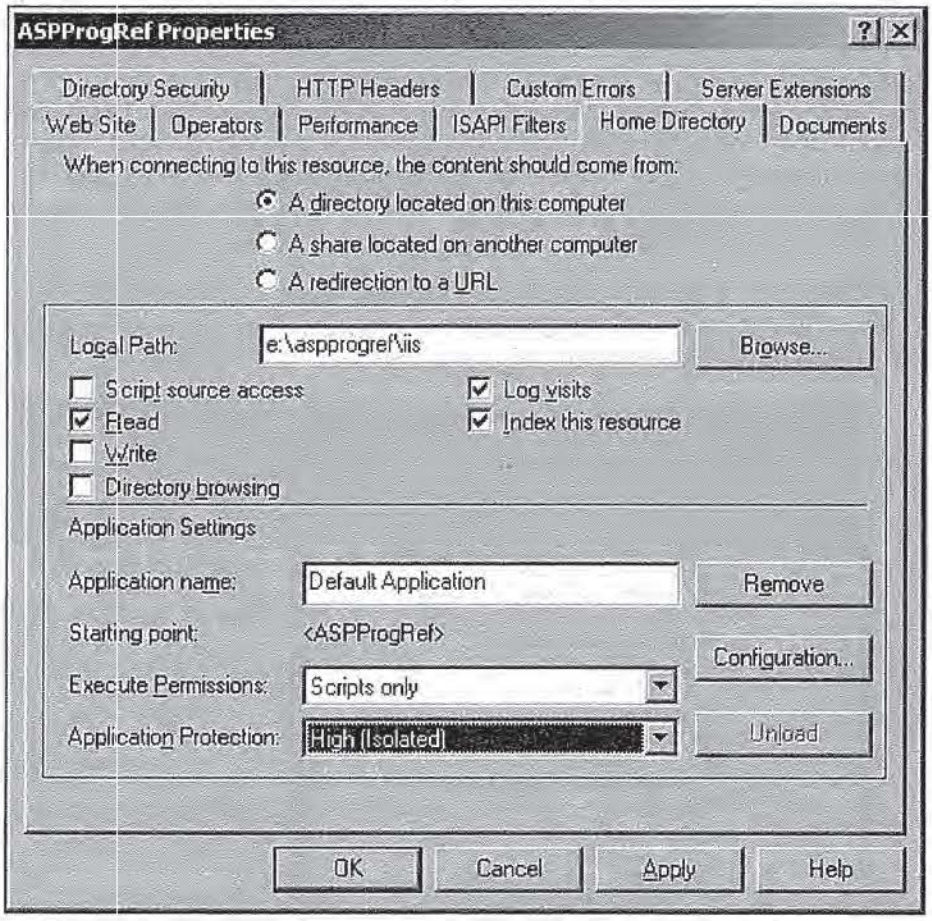
If we look at the Microsoft Management Console (MMC) snap-in for component services (the administration tool for COM+ which can be found under comexp.msc) we can see the integration between COM+ and IIS5:



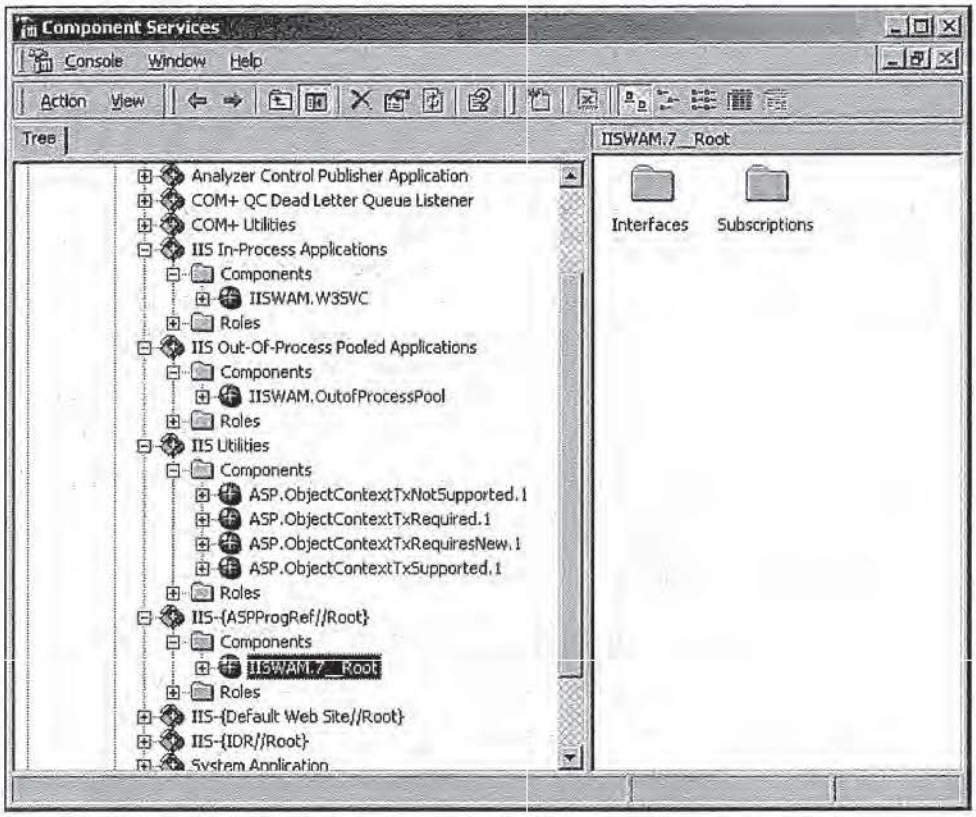
Under the COM+ Applications folder we can see the sub-folders IIS In-Process Applications, IIS Out-Of-Process Pooled Applications, and IIS Utilities. IIS has created these folders so it can utilitize transaction support, process isolation and other features such as fail-safe restart from COM+, rather than implementing them itself. Given these features are also probably going to be needed in non-IIS applications too, it makes sense that Component Services works this way.

Each COM+ application uses **declarative attributes** to specify the services it uses. In English, declarative attributes mean check boxes in the GUI, that when clicked define additional runtime behavior and functionality the component should inherit when used. When an instance of any component within a COM+ application is created, COM+ uses interception to ensure that services you specified using declarative attributes are correctly injected and used at runtime.

To create a COM+ application, we can change the Application Protection attribute of a web site to be High (Isolated):



We'll see that IIS5 creates a COM+ application by means of which the process isolation for that web site is provided:



At runtime you'll see an instance of `dllhost.exe` for each highly isolated web site and one instance for pooled web sites (medium isolation).

If you've used IIS4 and have a good understanding of IIS/COM/MTS relationships, you'll see that this isn't very different. The name of MTS has changed due to being a COM+ application. However, the key difference is that transactions are now an integrated part of the operating system. This integration applies all the way down to the context object (see more later), which is the recommended way for ASP intrinsic objects to be accessed.

Contexts and Interception

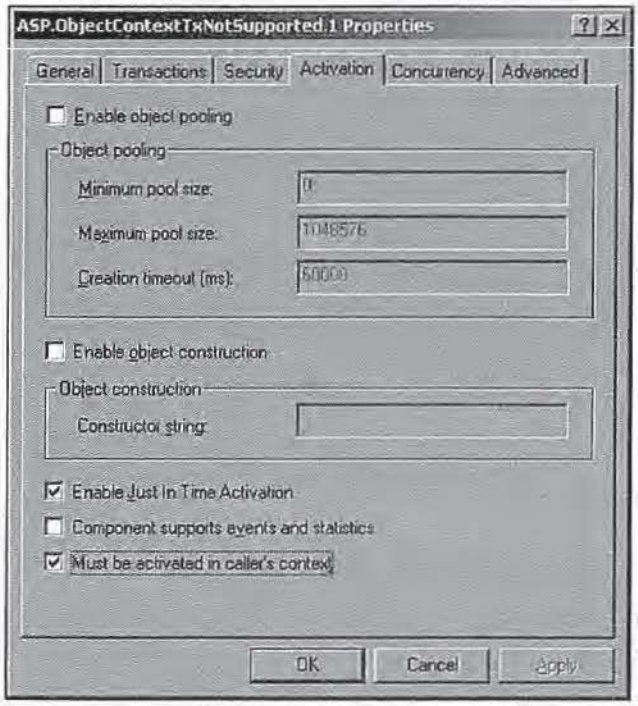
The most important feature that MTS brings to COM is the notion of the **context**. This information container enables MTS to make a stand-alone object form part of a distributed application, with synchronized concurrency, distributed transactions, and role-based security. You've probably seen this when accessing the ASP intrinsic objects with code like this:

```
Set objContext = GetObjectContext
objContext.Response.Write "ASP Rules OK!"
```

At first sight this looks like the object context is an IIS intrisic object, but it's not; the object context is transparently created by COM+ and is basically used to track and service objects. COM+ makes it possible to add services to a component after it is compiled, using **declarative attributes**. We don't have to write additional code to use transactions, queuing, security etc., it is done by the COM+ runtime automatically. It does this by inserting code before and after each method call pertaining to each requested service. This technique is referred to as **interception**.

Every COM+ object is associated with a single context when it is first created, or pulled from an object pool. For example, when IIS processes an ASP file, it creates a COM+ object to interpret the contents of the page and execute any ASP script statements. Once activated, the context associated with an object such as the ASP interpreter remains static throughout the object's lifetime, until the object is returned to a pool, or destroyed. The context is being used by COM+ to associate **out-of-band** information with an object throughout its lifetime, data needed to apply services to a component. The term 'out-of-band' simply means that the data is managed by COM+ behind the scenes. Without the context you would have to maintain such information yourself.

Several objects can share a context if their runtime requirements are compatible. The algorithm COM+ uses for its context compatibly test is not currently documented. If you do need to ensure objects are activated within the same context, you can use the 'Must be activated in caller's context' check box:



If two objects cannot be created in the same context, the creation will fail with the error `CO_E_ATTEMPT_TO_CREATE_OUTSIDE_CLIENT_CONTEXT`.

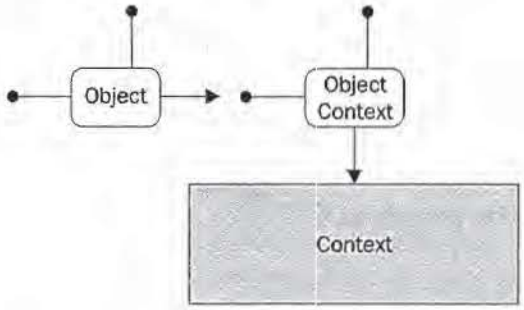
Within an object, you can obtain a reference to the context object by calling the `GetObjectContext` API. This returns a COM object called the **ObjectContext**. The default interface returned and used on this object is `IObjectContext`.

Activation

The process of getting an object into a state in which a client can use it is known as **activation**. The object creating the object is called the **activator**, and the activation process occurs when you call `CreateObject` in an ASP script. A context is associated with an object during its activation.

Objects and Contexts

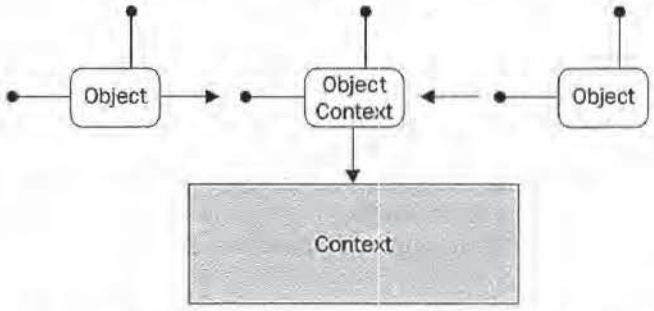
The relationship between an object, the object context, and the context is shown here, and remains static once an object has been activated:



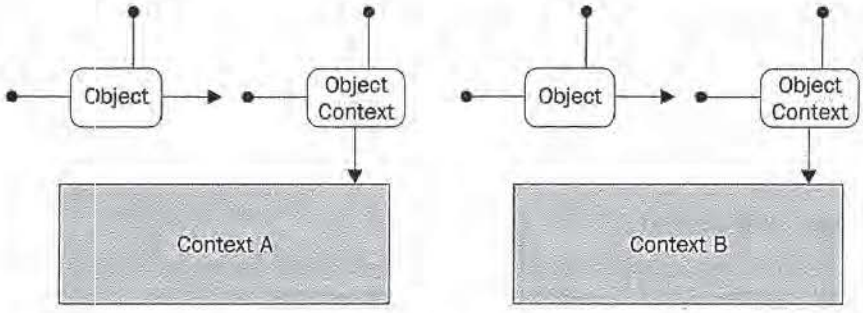
The name 'object context' is somewhat confusing, and don't be surprised if you find people using the terms 'object context' and 'context object' interchangeably.

Context Negotiation

When a COM+ object is created, the COM+ catalog is used to determine the services a component uses. If an existing context matches the newly-created object's requirements it will be used, if not, a new context is created:



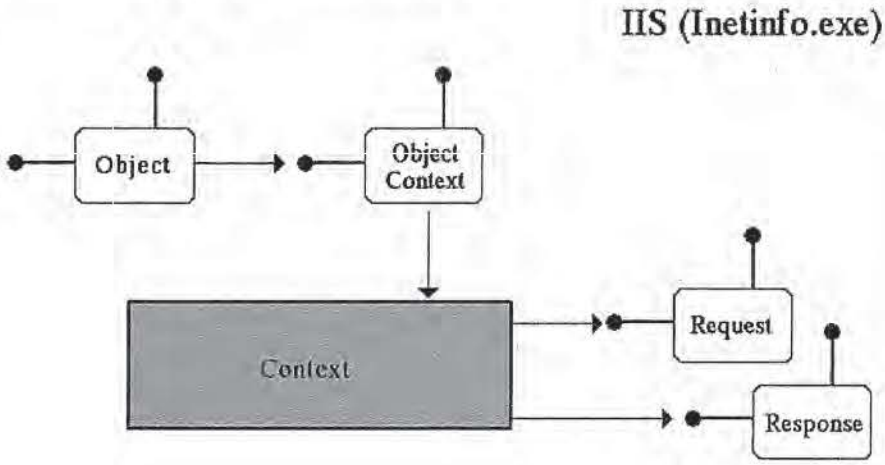
If the services required by the component mean that the context of the activator is incompatible with what it needs, a different context (and therefore object context) is used for the object.



COM+ uses interception (sometimes also referred to as a lightweight proxy) to ensure that the differences between the contexts of the activator and the object it creates are transparently managed at runtime by the COM+ runtime, so we don't have to worry about that in our code – client or component.

IIS and the Context

IIS uses the context to provide a COM object with access to the ASP intrinsic objects:



References to all of the ASP intrinsic objects (such as `Request` and `Response` objects) are held with the context as properties, put there by IIS. Any COM object that is associated with that context can access those properties through the object context associated with the context. Remembering the object context is a COM object that provides an interface into the context from within an object, we can access the IIS objects and use them:

```
Set objContext = GetObjectContext
objContext.Response.Write "ASP Really Does Rule OK!"
```

What this means is that a COM object can access the functionality of IIS (or any other application that uses the context to expose functionality) even though there is no **explicit** link or association between IIS and the object.

Using COM+, we can build **COM+ applications** out of components, leveraging foundation **services** and code from the operating system. These services give us a

jump-start when developing n-tier applications based upon the Windows DNA framework. The applications we create can take advantage of the services and infrastructure that COM+ provides, which a typical enterprise application needs, such as transaction processing, component management, security and object pooling. A lot of Windows DNA 2000 is provided by COM+.

Each COM+ service is non-trivial to implement and would typically require many years of development and testing, but as they are provided as part of Windows 2000, all we have to do to make use of them is to use the **Component Services Explorer** to define our applications – a tool to manage and administer our COM+ applications and associated components in a very easy-to-use environment.

> *Component Services is the umbrella name Microsoft uses to encompass COM, COM+, and the related technologies like Microsoft Message Queue (MSMQ).*

IIS and Active Server Pages

The role of IIS and ASP within Windows DNA should be fairly clear: they connect the user services tier with the business services tier. Using the HTTP protocol, web pages and any others type of document can be sent between the two tiers. One of the enhancements in IIS5 is that the ASP `Request` and `Response` objects now support streaming of data in-memory. This means that a file (such as an XML document) can be saved directly to the `Response` object, or loaded directly from the `Request` object without having to go via file or some other medium. For example, the XML object model in IE5 can be used to create an XML document and save it to the `Response` object. In VBScript this looks like:

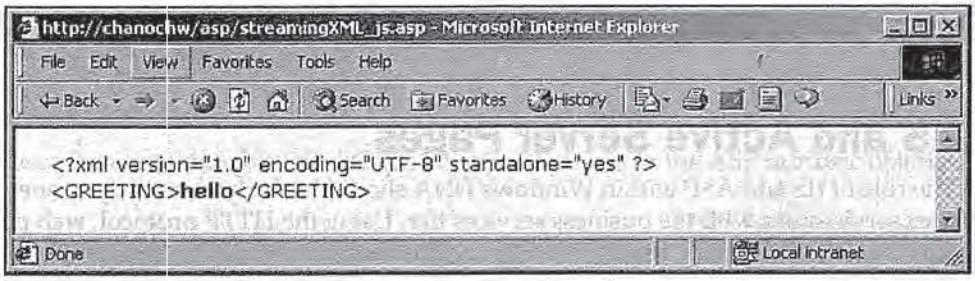
```
<%
Response.ContentType = "text/xml"
Set objResponse = Server.CreateObject("Microsoft.XMLDOM")

Dim objDocElement
Set objPI = objResponse.createProcessingInstruction("xml", _
                "version='1.0' encoding='UTF-8' standalone='yes'")

Set objDocElement = objResponse.createElement("GREETING")
objDocElement.text = "hello"

objResponse.appendChild objPI
objResponse.appendChild objDocElement

objResponse.save response
%>
```

Or in JScript:

```
<%@ LANGUAGE=JScript %>

<%
Response.ContentType = 'text/xml';
var objResponse = Server.CreateObject('Microsoft.XMLDOM');
```

```
var objPI = objResponse.createProcessingInstruction('xml',
            "version='1.0' encoding='UTF-8' standalone='yes' ");

var objDocElement = objResponse.createElement('GREETING');
objDocElement.text = 'hello';

objResponse.appendChild (objPI);
objResponse.appendChild (objDocElement);

objResponse.save (Response);
%>
```

If we view this page under a Windows 2000 client (and therefore IE5) the browser will render the XML:



Even if you don't know much about the XML object model (which is covered in Chapter 35) you can appreciate the benefits of in-memory streaming as the document didn't have to be saved to disk. As it turns out, this code is also the only way in which we can send XML documents in UTF-8 format via ASP. If we tried writing code like this:

```
objResponse.save objResponse.xml
```

you'd find the XML document sent back to the client would be invalid if it contained characters that were outside of the range 0-126. Characters > 126 (such as Umlauts – Ü) require special encoding.

That is because the XML property returns a Unicode string which doesn't match the UTF-8 encoding we specified in the XML declaration.

Transactions

A transaction is any set of operations which must be performed as a single unit and any changes by which must be undone if any one operation fails. There is no intermediate state or partial completion; either they all succeed in which case the changes are saved or no changes occur. Transactions have the following properties, known as ACID:

- Atomicity – the set of operations must be completed as a whole or all will fail,
- Consistency – the end result is that which we were trying to achieve,
- Isolation – from other transactions; any one transaction can complete its tasks without needing to know about other such transactions
- Durability – which means that only when the transaction has completed should it report succesful completion.

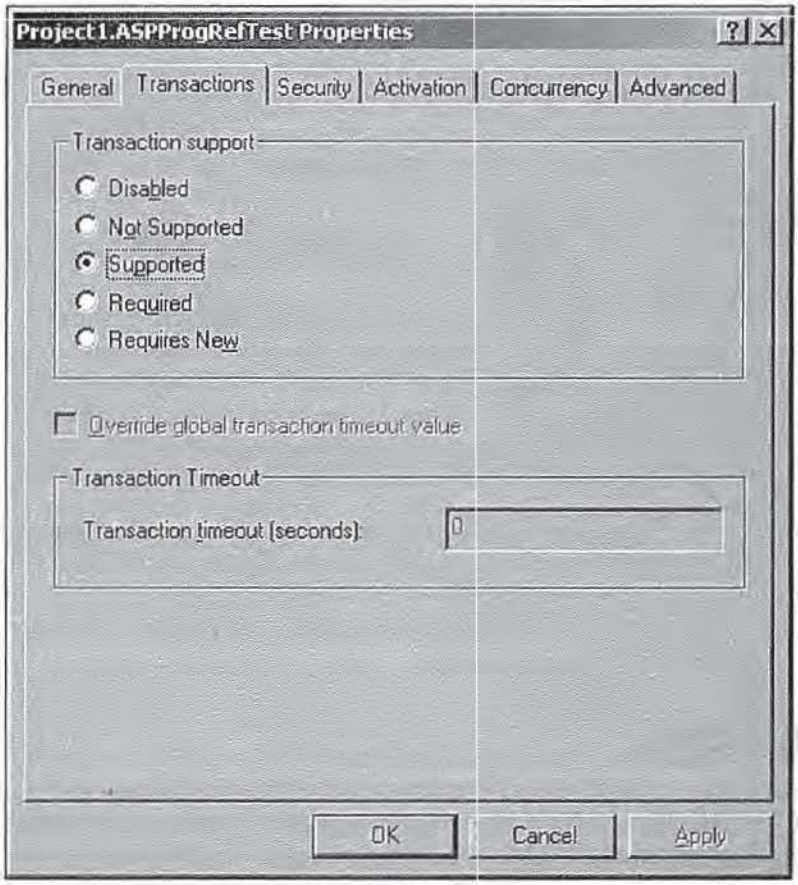
For more information see Chapter 34, Transactions and Message Queueing.

Transaction support is available to an ASP script in two ways. The first way uses the `@TRANSACTION` directive inside of an ASP page to cause a transaction to be started when the page is first processed, which then completes when the page has been completely processed. This looks like:

```
<%@ TRANSACTION=Required %>

<%
    ' Some work here
    ' and here ...
%>
```

The second way of using transactions is within a page using one or more components. These need to be installed into Component Services using the MMC (since with Windows 2000, MTS is now just another part of Component Services (COM+) known as **Transaction Services**) and marked as transactional. In this scenario the transaction starts when an instance of the component is first created, and completes when all references to that component are released.

If we select the properties of a component from within the MMC and select the Transactions tab, we see:

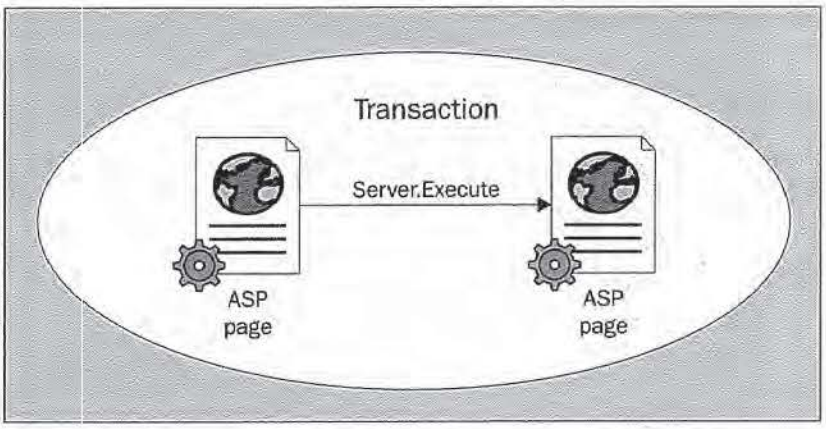


When MTS was a layer on top of COM in NT 4.0, COM didn't actually know about it, so there was a degree of trickery and overhead in terms of performance and programming. With MTS you couldn't pass a reference to a COM object installed inside of an MTS package directly. Instead you had to call `SafeRef` to return a special context wrapper (not context object!) that you could pass safely. The context wrapper would ensure that the runtime environment for an MTS component was setup before a method was invoked. Because transaction services are now an integral part of COM+, both of these overheads have been removed.

A new feature of IIS5 in transactions is that ASP pages can now invoke other ASP pages much like subroutines:

```
<%@ TRANSACTION=Required %>

<%
   ' Some work here
   Server.Execute "somepage.asp"
   ' Some more work here
%>
```

If the second page has a compatible transaction attribute, the same transaction is used to encompass the work of *both* pages and so actions can be rolled back across several pages; that is, any changes made will not be saved unless all the operations within each page have processed successfully, and otherwise the system will be returned to the state it was in when we began:



Although not as efficient as using components, this technique of being able to nest pages is an interesting way of reusing ASP scripts. Common code that has to be transactional can easily be included within other files. This could not be achieved using the include directive, as page attributes such as `@TRANSACTION` are not valid for them.

Messaging

Messaging is an integral part of Windows 2000. It enables asynchronous messages to be sent between two parties. If the two systems are disconnected, messages are queued until the recipient is accessible.

Allowing disconnected operations is important for modern applications as more and more people have laptops and need to work on the move. In this situation they need to be able to use their applications whether or not they are connected to their company's network. They therefore may or may not have access to the data sources that an application typically depends upon. The application must be willing to take an order for a salesman even if they are in a plane over the Atlantic and may not be connected to a network.

Another example: Imagine a Wrox salesman who is at a customer site. He sells 20,000 copies of "Professional XML" and 10,000 copies of "Beginning ASP Components". He enters the order into his laptop (which is not connected to any network), thanks the customer for the order, and then heads off back to the office thinking about his bonus. The operation was performed in a disconnected mode. The order can't be entered into the main order processing database back at Wrox, until he gets back to base, so the application has created a **message** and placed it in a **queue** for later processing. Assuming the journey back to the office goes well, when the laptop is connected to the Wrox network, the queue can be processed and the message forwarded to the ordering processing system.

Messaging is also important from a web perspective to allow our applications to work in a disconnected mode and may be useful for resource management. For example, we might want to be able to still accept orders, even when our database is down for maintenance work. We can achieve this by sending all orders via a message queue. An interesting application of such messaging is for controlled resource throttling on a busy system. Rather than processing every single request (such as an order) completely as they are received, we can partially process each one. We can then create a message describing the rest of the work to be carried out, and place it in a queue for further processing. By using a queue you can decide how many messages are processed at once. So, rather than your server trying to process thousands of simultaneous orders requests SSLLOOWWLLYY, driving your customers away, you can do the initial processing, give the user the initial confirmation quickly, then forward the requests for complete processing at your leisure.

Windows 2000 uses **Microsoft Message Queue Server** (MSMQ) to provide messaging. A layer on top of this known as **queued components** provides the messaging service that most *component*-based Windows 2000 DNA applications will use. This additional layer allows you to asynchronously invoke methods on COM objects. The way this works is simple but very clever. When you create a COM object you actually get returned a reference to a **recorder** object, and not the actual object. You invoke the methods of the COM object as usual, and the queued components layer creates a message that records all the methods you called along with all the parameters etc. The message is forwarded to a **player** object that then creates the real COM object and invokes the various methods just as if you'd directly called them. This eliminates the need for you to implement the recorder/player infrastructure yourself. This is good news because it is difficult and time consuming to do, and it requires knowledge of fairly low-level C/C++ programming.

MSMQ supports and works together with Transaction Services. Using a salesman example, when a new order is taken we could log it against the value of sales, then create a message to process the sale later (check availablity, arrange delivery, charge the client) and place it in the new orders queue. Transaction services would ensure that, if the order was cancelled for any reason, the log entry would be deleted.

Last but not least, an important aspect of messaging is reliability. Window 2000 messaging ensures that once a message is placed in a queue it will be delivered once and only once.

Universal Data Access

Most applications today are centered around data in some way. Data is kept in spreadsheets, e-mail folders, databases and Mainframes etc. The business services tier of our Windows DNA application will need access to much of this data.

In order to access data from many diverse data stores, Microsoft has devised a strategy known as **Universal Data Access**. UDA describes how we can access structured and unstructured data in heterogeneous data sources. It is important to note that UDA isn't a database or data source API, but rather a strategy for data access. From an ASP perspective the key technologies for implementing UDA are: ADO, OLE DB, and ODBC.

Typically, data access is performed by an ASP page or data-access component within the business services tier using ActiveX Data Objects (ADO). This is very simple to use in ASP, as shown by this small VBScript example that simply opens a connection to an Access database:

```
<%
'Create a connection object.
Set objConn = Server.CreateObject("ADODB.Connection")
objConn.Open "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=" & _
             "C:\aspprogref\rja.mdb"
%>
```

Or in JScript:

```
<%@ LANGUAGE=JScript%>
<%
//Create a connection object
var objConn = Server.CreateObject("ADODB.Connection");
objConn.Open ('Provider=Microsoft.Jet.OLEDB.4.0;Data Source=' +
              'C:\\aspprogref\\rja.mdb');
%>
```

ADO is described in more detail in Section IV of this book (Chapters 26 to 33).

ADO is a layer on top of OLE DB, that consists of a set of low-level COM interfaces and components that define how a data source can be accessed and manipulated. OLE DB cannot be directly used in ASP script or VB. This limitation exists because VB and ASP script are not 100% COM capable, and cannot access COM interfaces that do not use automation-compatible data types, a subset of the full COM data types that can be used in C/C++.

OLE DB 2.5 (the version shipped with Windows 2000) is not defined as an API for relational databases like older APIs such as ODBC. Instead it is defined as an API for

accessing data sources, and is therefore far more flexible. A data source can be just about anything, a text file, your e-mail folders or the afore-mentioned relational database. OLE DB was originally geared towards structured data in the form of columns and rows, but OLE DB 2.5 makes accessing unstructured data, such as ASP files from your web site a lot easier, thanks to some new interfaces and components.

XML

Over the past year or two we've all heard a lot about the promise of XML: the new markup language of the Web. It differs from HTML is that it does not have predefined tags and it is primarily concerned with describing the data rather than how it should be displayed.

For example, an HTML page containing information about a book might look like this:

```
<H1>ASP Prog Ref 3.0</H1>
<P>The aim of this book is to provide a comprehensive reference to
both the ASP technology itself as well as the ways that ASP can be
used within the Microsoft environment.</P>
```

Whilst a browser understands how to display this, and as humans, we can guess that it is information about a book, HTML does not explicitly say that the text within the `H1` tags is the title of a book, and that the paragraph text is the abstract for the book. Here lies the biggest and most important different; with XML you can define your own tags, which will describe the content within them:

```
<Book>
   <Title>ASP Prog Ref 3.0</Title>
   <Abstract>The aim of this book is to provide a comprehensive
    reference to both the ASP technology itself as well as the
    ways that ASP can be used within the Microsoft environment.
    </Abstract>
</Book>
```

XML allows us to define our own tags that give semantic meaning to our data. Book, Title, and Abstract are far more evident and allow us more control over the data.

We can therefore use XML in many different ways within our DNA applications. We could pass XML between tiers, maybe to describe orders to be added to a database, or we could use XML as a simple client-side caching mechanism to reduce round trips. Microsoft is investing heavily in XML and this is reflected in Windows 2000 shipping with native support for XML in the form of MSXML. Originally released as an XML add-on component for IE4, MSXML version 2.0 is part of IE5, the native browser for Windows 2000.

In the next few months IE 5.5 will be released. This will ship by default with MSXML 2.6. This version of MSXML implements the latest W3C working drafts, and is significantly more scalable on multiple CPU boxes.

XML, XSLT, and XPath are covered in chapters 35 and 36.

Web Services – The Next Generation of Web Development

Currently, information that a company exposes on the Web is mostly only available in HTML format. When we have found it we can remember it, save it to disk, print it out, or write it down, but all of these involve interpretation and are time consuming. We can develop a parser to extract the information but this is dependant on the structure of that site's pages.

For example, Amazon.com lists most of the Wrox Press books including this one. As an author I like to go to Amazon and see how the books I've been involved with are selling. To do this I have to manually go to each page and view the ranking and reviews. This is a real pain as I'm not interested in the flashy graphics and plugs for other books; I want to see only the two elements of information I'm interested in, the ranking of my book and the reviews. In SQL terms, I'd like to write something like this:

```
SELECT ranking, reviews FROM WWW.AMAZON.COM
WHERE PUBLISHER = 'Wrox Press' and Author = 'ME'
```

It would be very useful for me if Amazon would expose function for getting this information. It's safe to assume that the data access infrastructure is there, so all we need is a simple access method which bypasses the adding of all that HTML noise.

Any solution must take into account platform compatibility issues, it must be fairly easy, and it must be secure. This brings us to XML, or XML over HTTP. If Amazon were to use XML pages, we could have access to the information in either HTML or XML. While I could surf to the page traditionally is which a fairly simple .asp file could convert it to HTML, if I were to specifically ask for it in XML format, the page would be returned in the following form:

```
<Books>
  <Book>
    <Title>ProXML</Title>
    <Ranking>81</Ranking>
    <Reviews>
      <Review>I am new to XML and ...</Review>
      <Review>This book is ...</Review>
      <Review>Another review of this book</Review>
    <Reviews>
  </Book>
  ...
</Books>
```

The URL for this request might look something like:

```
http://www.amazon.com/querybooks.asp?publisher=wrox&author=me
```

This output can be generated just like any other HTML page, by using ASP and ADO. We've used HTTP, and all we've done is returned an XML document rather than HTML. XML is easy to parse, in fact made to parse, so I could easily write an application to load this file and process it. The page is not design-specific, that is it would be contained in the ASP file, so it wouldn't change if Amazon redesigned its site. What we've achieved is the creation of a web service. Using this simple technique sites like Amazon can expose all sorts of function to clients. Furthermore, other sites can easily consume this data and expose it on their sites. For example, Wrox Press could automatically display the Amazon ranking and reviews of their books on wrox.com.

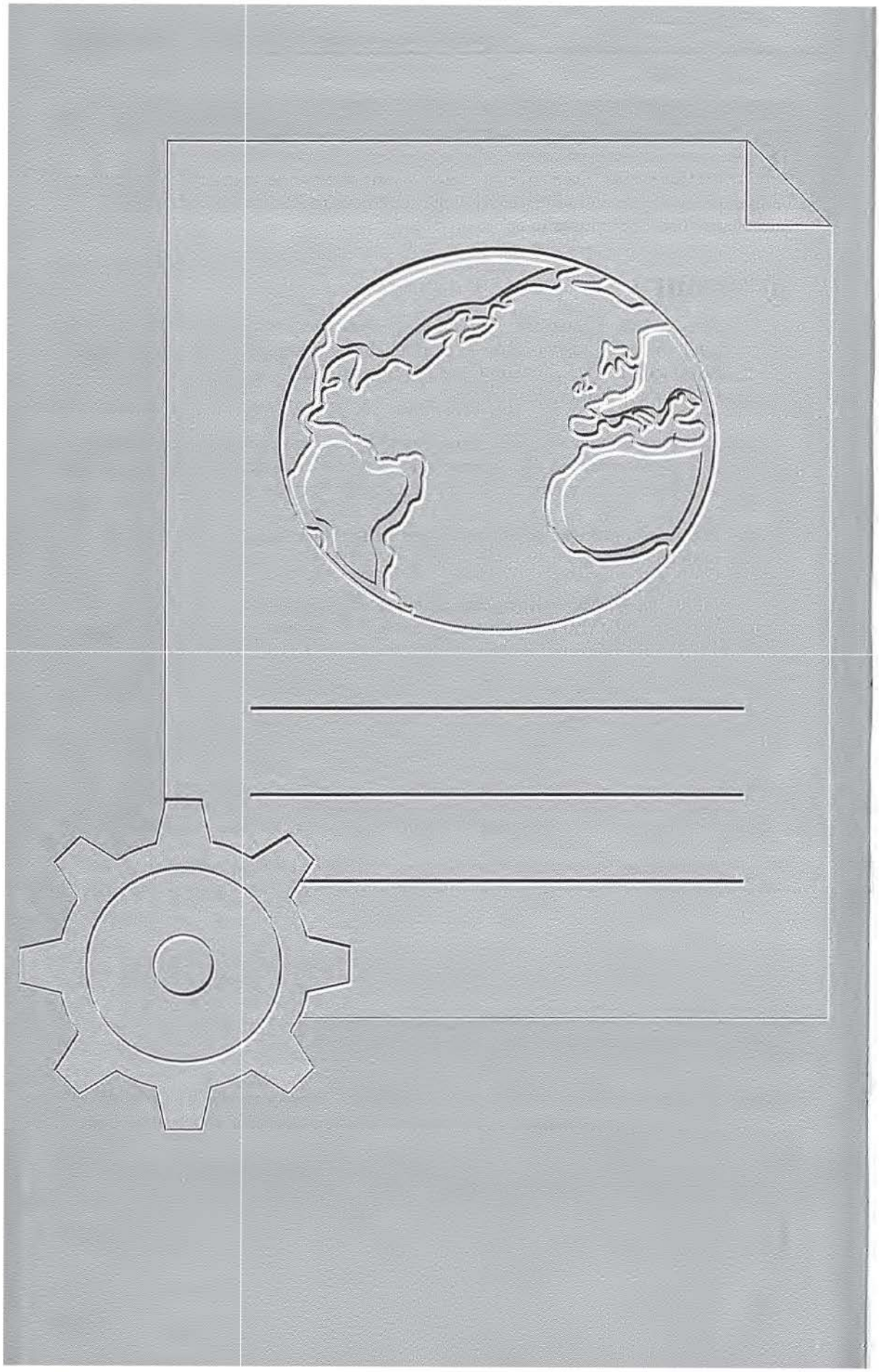
What I've defined as a web service here is what I call a first generation web service. XML is returned from the server to client, but the client doesn't use XML to talk to the server. In a second generation web service XML is used by the client to the talk to the server. This gives the client more flexibility because the input data base to the web service can be arbitrarily complex. This model also means that more advanced services, potentially object-based, can be implemented. This second generation web service is what SOAP addresses.

More details on SOAP can be found at `http://www.develop.com/soap`.

Summary

This introductory chapter contains:

- An overview of Windows 2000.
- An overview of Windows DNA.
- An overview of Component Services and COM+.
- An introduction to the topics covered in the rest of the book .



35

The XML DOM

XML, or **Extensible Markup Language**, is rapidly becoming an important tool for persisting and transmitting structured and semi-structured information. Microsoft has provided a library for the manipulation of XML documents. This library is Microsoft's implementation of the World Wide Web Consortium's (W3C) Document Object Model (DOM) specification for XML. It may be found in `msxml.dll`, which is installed automatically when Internet Explorer 5.0 or higher is installed, and is a standard component of Windows 2000.

What is the XML DOM?

The XML DOM is the mechanism defined by the W3C for structural access and modification of XML documents. It allows XML documents to be viewed as a tree of objects, rather than serialized text. This has several advantages over manipulating the text directly:

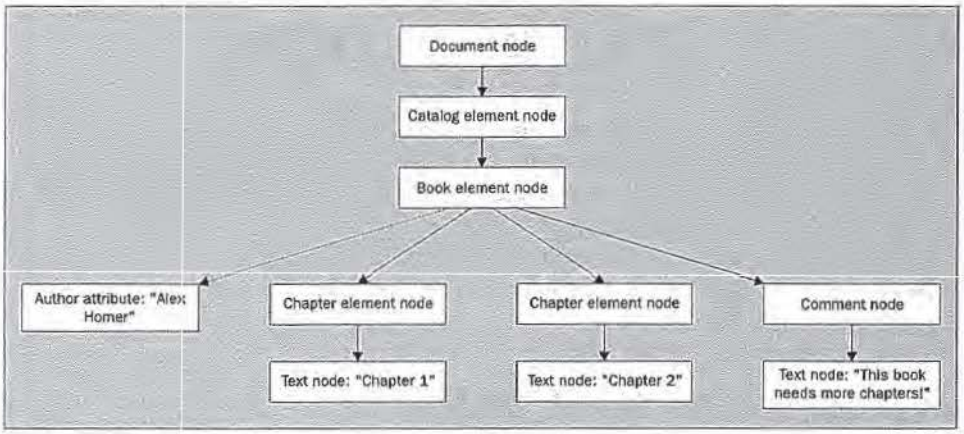
- Using the XML DOM guarantees that any document you create will be well-formed; since the document is created as a tree of nodes, rather than serialized text, problems like forgetting to write the end tag for an element to a stream just cannot occur.
- Using the XML DOM allows documents being read to be parsed for well-formedness and validity (if a DTD or schema is supplied for the document).
- Using the XML DOM allows documents to be created ad-hoc, with the serialization to a stream only happening after the document has been constructed as a tree.
- Using the XML DOM allows the use of XSL patterns (and, in version 2.6, XPath expressions) to search XML documents and retrieve nodes quickly.

Let's take a look at a quick example to understand how the tree view of XML documents works.

For example, if we have the XML document below:

```
<Catalog>
   <Book author="Kevin Williams">
      <Chapter>Chapter 1</Chapter>
      <Chapter>Chapter 2</Chapter>
      <!-- This book needs more chapters! -->
   </Book>
</Catalog>
```

This document is parsed by the DOM into a tree of nodes: a document node (which is always the topmost node in any document), containing a `<Catalog>` element. This contains a single `<Book>` element, which itself contains two `<Chapter>` elements and a comment. The `<Chapter>` elements and the comment contain text nodes representing their string values. The DOM generates a tree in memory that looks something like this:



Note the dotted line going to the `Author` attribute – this is because attributes are treated as special node types in the DOM. There are methods and properties specifically dedicated to the manipulation of attribute values.

Which Version Should You Use?

The version of `msxml.dll` that ships with Internet Explorer 5 is version 2.0. This version provides support for XSL-style transformations, as well as all of the DOM functionality. Version 2.6 is available as a download from Microsoft (as of press time), and adds full support for newer XSLT-style transformations. The new objects in version 2.6 are clearly marked in the reference section; if you don't need to take advantage of this functionality, you don't need to obtain the newer version of the DLL. Also, if you choose to use version 2.6, be sure to read the documentation carefully – to ensure backwards compatibility, Microsoft has elected to embed the 2.6 functionality (for the purposes of this release, at least) in a separate DLL, `msxml2.dll`. You can choose to replace the existing `msxml.dll` with the newer `msxml2.dll` at install time if you like, or run both side-by-side. The package also includes a version of `msxml.dll` that fixes some performance problems with multiple connections from IIS, so it's worth a look as well.

Version 2.6 of the Microsoft XML libraries, together with an SDK containing documentation on the new objects, may be downloaded from:

```
http://msdn.microsoft.com/downloads/webtechnology/xml/msxml.asp
```

Using XML from ASP

Before we go on to look at the objects provided by the DOM in detail, we'll first have a quick look at some of the tasks that are commonly performed when manipulating XML documents from ASP, and give some introductory examples:

- Accessing stand-alone documents from ASP
- Creating an XML document from scratch
- Sending an XML document to the client
- Storing an XML document to a file

Accessing Stand-alone Documents from ASP

Stand-alone documents are accessed from ASP using the objects encapsulated in `msxml.dll`. The `XMLDOMDocument` object represents an entire XML document, and is used to load and parse XML files, either from in-memory strings or streams. This object should always be the first one created when working with the XML DOM objects; the other objects will then be accessible via factory methods or properties of the `XMLDOMDocument` object. To access a stand-alone document from a URL, you need to create an instance of the `XMLDOMDocument` object:

```
Dim xmldocument
Set xmldocument = Server.CreateObject("Microsoft.XMLDOM")
```

For the purposes of this example, we'll set the document to load synchronously (if you want to load asynchronously, you'll need to set an event handler on the `onreadystatechange` event):

```
xmldocument.async = false
```

Then, the document can be loaded and parsed by calling the `load` method of the newly-created object:

```
xmldocument.load("http://myServer/xml/books.xml")
```

We must specify either a full URL or a full physical path in the `Load` method.

This will load the document from the remote resource and parse it. This document is retrieved using HTTP, so all the same restrictions for other operations that take place over HTTP pertain here: do all intervening firewalls allow HTTP traffic? Is the document protected? If a proxy server is being used, is the system configured to allow proxied access to the document over HTTP?

Since we set the document to load synchronously, once control is returned to us we know that the document has been loaded and parsed. We can now check to see if the document parsed correctly:

```
If xmldocument.parseError.errorCode <> 0 Then
   ' the parse failed - take some corrective action
Else
   ' the parse succeeded - continue normally
End If
```

The JScript equivalent for this code is:

```
var xmldocument = Server.CreateObject('Microsoft.XMLDOM');
xmldocument.async = false;
xmldocument.load('books.xml');

if (xmldocument.parseError.errorCode != 0) {
   // the parse failed - take some corrective action
} else {
   // the parse succeeded - continue normally
}
```

A complete list of the errors that may be generated by the MSXML parsing engine may be found in Appendix M.

Alternatively, if you want to load the document asynchronously, you can set the `async` property to `true`, and then set an event handler for the `onreadystatechange` event:

```
xmldocument.async = true
xmldocument.onreadystatechange = "HandleDocStateChange"
```

Or, in JScript:

```
xmldocument.async = 1;
xmldocument.onreadystatechange = "HandleDocStateChange";
```

After calling the load method, script execution will continue while the document is loaded and parsed in the background. The handler you have specified (in this case, `HandleDocStateChange`) will then be called as the `XMLDOMDocument` object's state changes, from `"uninitialized"` to `"loading"` to `"loaded"` to `"interactive"` (some data is available, but the parsing has not yet been completed) to `"complete"`. You may write code in the event handler to take action once the parse of the document has completed.

Creating an XML Document from Scratch

To create an XML document in memory without loading a document, we must first create an empty instance of the `XMLDOMDocument` object:

```
Dim xmldocument
Set xmldocument = Server.CreateObject("Microsoft.XMLDOM")
```

Or, in JScript:

```
var xmldocument;
xmldocument = Server.CreateObject("Microsoft.XMLDOM");
```

Next, we create our various nodes and attach them to the node tree as we create them. For example, to create the root <Book> element for our document, we could do the following:

```
Dim xmlelement
Set xmlelement = xmldocument.createElement("Book")
xmldocument.appendChild(xmlelement)
```

Or, in JScript:

```
var xmlelement;
xmlelement = xmldocument.createElement("Book");
xmldocument.appendChild(xmlelement);
```

The entire document may be built in memory in this way.

If we have information in a serialized format that would lend itself to generation of XML text directly, we can also construct the document this way. Simply construct the XML text in a string:

```
Dim xmltext
xmltext = "<Book>"
xmltext = xmltext & "<Author>"
xmltext = xmltext & "Kevin Williams"
xmltext = xmltext & "</Author>"
xmltext = xmltext & "</Book>"
```

Or, in JScript:

```
var xmltext;
xmltext = "<Book>";
xmltext += "<Author>";
xmltext += "Kevin Williams";
xmltext += "</Author>";
xmltext += "</Book>";
```

If we then want to manipulate the document using the XML DOM, we can create an `XMLDOMDocument` and use the `loadXML` method on that document:

```
Dim xmldocument
Set xmldocument = Server.CreateObject("Microsoft.XMLDOM")
xmldocument.loadXML(xmltext)
```

Or, in JScript:

```
var xmldocument;
xmldocument = Server.CreateObject("Microsoft.XMLDOM");
xmldocument.loadXML(xmltext);
```

The parser will parse the document and populate the `parseError` object accordingly, making this method a good way to verify that you have created a well-formed document in your string variable:

```
If xmldocument.parseError.errorCode <> 0 Then
   ' the parse failed - we must not have created it correctly
Else
   ' the parse succeeded - continue normally
End If
```

Or, in JScript:

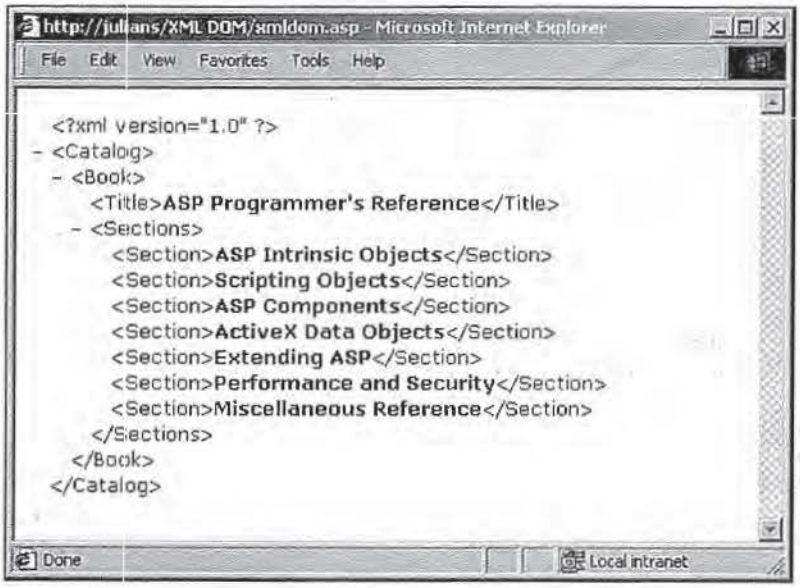
```
if (xmldocument.parseError.errorCode != 0) {
   // the parse failed - we must not have created it correctly
} else {
   // the parse succeeded - continue normally
}
```

Sending an XML Document to the Client

Once you have modified an existing XML document or created a new one in memory, you will probably want to do something with it. The simplest possibility is just to send the document to the client without formatting. To do this, we just need to set the `Response.ContentType` to `"text/xml"` and then write the contents of the document to the `Response` object:

```
Response.ContentType = "text/xml"
Response.Write xmldocument.xml
```

Any associated stylesheet will be applied in the browser. Otherwise, the browser's default stylesheet will be used:



However, this will only produce meaningful results if we can guarantee that the browser supports XML (at the moment, this effectively means IE5+). A more useful technique over the Internet is to use an XSLT stylesheet to transform the XML into HTML before sending it to the client by calling the `XMLDOMDocument`'s `transformNode` method:

```
' VBScript
Dim objXML, objXSL

' Create XMLDOMDocument objects for the XML file and the stylesheet
objXML=Server.CreateObject("Microsoft.XMLDOM")
objXSL=Server.CreateObject("Microsoft.XMLDOM")
```

```
' Set synchronous loading
objXML.async=false
objXSL.async=false

' Load the XML files into our objects
objXML.load "http://myserver/books.xml"
objXSL.load "http://myserver/books.xsl"

' Transform the XML document and send it to the browser
strXML = objXML.transformNode(objXSL.documentElement)
Response.Write strXML
```

```
// JScript

// Create XMLDOMDocument objects for the XML file and the stylesheet
var objXML=Server.CreateObject("Microsoft.XMLDOM");
var objXSL=Server.CreateObject("Microsoft.XMLDOM");

// Set synchronous loading
objXML.async=false;
objXSL.async=false;

// Load the XML files into our objects
objXML.load("http://myserver/books.xml");
objXSL.load("http://myserver/books.xsl");

// Transform the XML document and send it to the browser
var strXML=objXML.transformNode(objXSL.documentElement);
Response.Write(strXML);
```

XSLT stylesheets are discussed in depth in the next chapter.

Storing an XML Document to a File

An alternative is to **persist** the document to a file. This is simple – the Microsoft implementation of the XML DOM includes a `save` method of the `XMLDOMDocument` object for just this purpose:

```
xmldocument.save("c:\xml\newbooks.xml")
```

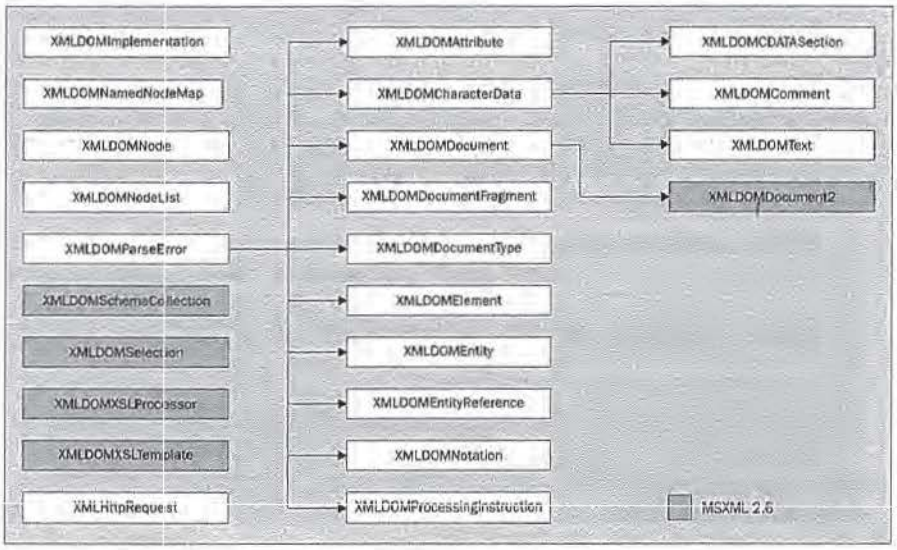
Or, in JScript:

```
xmldocument.save("c:\\xml\\newbooks.xml");
```

Note that this only allows persistence to a local file, not across the Internet. Also, this method cannot be used inside a secured environment such as a web browser, as these environments are prevented from accessing the local file system directly because of the potential for abuse by hackers. We should therefore only use this method on the server.

The XML Document Object Model (DOM)

Manipulation of XML documents is performed via the Document Object Model libraries provided as part of Internet Explorer 5.0 and above. Some of the objects exposed by the Microsoft XML DOM libraries are base objects that other objects inherit from. The inheritance tree for the Microsoft XML DOM is shown in the following picture:



Only three of these objects may be instantiated directly: the `XMLDOMDocument`, which is the root node of the entire document; `XMLHttpRequest`, which is a helper object created to facilitate the transmission of XML documents across the Web; and the `XSLTemplate` object, which is used to cache compiled XSL templates. The other objects must be either referenced through properties of, or created by factory methods of, the `XMLDOMDocument` object.

XMLDOMAttribute

The `XMLDOMAttribute` object represents an attribute in an XML document. It has a `name` property representing the name of the attribute, and a `value` property representing the value stored in the attribute. For example, in the following document fragment:

```
<Book Author="Kevin Williams" />
```

There would be one `XMLDOMAttribute` object in the node tree, with a name of `Author` and a value of `"Kevin Williams"`. This value is actually contained in a child text node, but can be accessed directly through the `XMLDOMAttribute` object.

The `XMLDOMAttribute` objects representing the attributes associated with a particular element may be accessed by examining the `getattribute`, `getattributenode`, `setattribute`, `setattributenode`, `removeattribute`, and `removeattributenode` methods of the `XMLDOMElement` object, as well as its `attributes` property.

For example, to retrieve a reference to an `XMLDOMAttribute` using the `getAttributeNode` method of the element node to which the attribute belongs:

```
' VBScript
Dim xmlAuthorAttribute
' get the Author attribute for the Book element referenced by the
' xmlBookElement variable
Set xmlAuthorAttribute = xmlBookElement.getAttributeNode("Author")
```

```
// JScript
/* get the Author attribute for the Book element referenced by the
   xmlBookElement variable */
var xmlAuthorAttribute = xmlBookElement.getAttributeNode("Author");
```

Alternatively, we can access an attribute through the `XMLDOMNamedNodeMap` for the element:

```
' VBScript
Dim xmlNamedNodeMap, xmlAuthorAttribute
' get the Author attribute for the Book element referenced by the
' xmlBookElement variable
Set xmlNamedNodeMap = xmlBookElement.attributes
Set xmlAuthorAttribute = xmlNamedNodeMap.getNamedItem("Author")
```

```
// JScript
/* get the Author attribute for the Book element referenced by the
   xmlBookElement variable */
var xmlNamedNodeMap = xmlBookElement.attributes;
var xmlAuthorAttribute = xmlNamedNodeMap.getNamedItem("Author");
```

We can also create an attribute using its parent element's `setAttribute` method:

```
' VBScript
' create an Author attribute for the Book element referenced by the
' xmlBookElement variable
xmlBookElement.setAttribute "Author", "Kevin Williams"
```

```
// JScript
/* create an Author attribute for the Book element referenced by the
   xmlBookElement variable */
xmlBookElement.setAttribute("Author","Kevin Williams");
```

The `XMLDOMAttribute` object is derived from the `XMLDOMNode` object. In addition to the `name` and `value` properties described below, see the methods, and properties defined for the `XMLDOMNode` object for additional functionality available through the `XMLDOMAttribute` object.

Methods	Properties	
appendChild*	attributes*	nodeTypedValue*
cloneNode*	baseName*	nodeTypeString*
hasChildNodes*	childNodes*	nodeValue*
insertBefore*	dataType*	ownerDocument*
removeChild*	definition*	parentNode*

Methods	Properties	
replaceChild*	firstChild*	parsed*
selectNodes*	lastChild*	prefix*
selectSingleNode*	name	previousSibling*
transformNode*	namespaceURI*	specified*
transformNodeToObject*	nextSibling*	text*
	nodeName*	value
	nodeType*	xml*

* See section on XMLDOMNode methods and properties.

Additional Methods

There are no additional methods defined for the XMLDOMAttribute object.

Additional Properties

The XMLDOMAttribute object has two properties which are not inherited from XMLDOMNode: name and value.

name

The read-only name property returns a string containing the name of the attribute.

```
String = XMLDOMAttribute.name
```

For example, the following code checks the name of the first attribute of an element, and if it is equal to "Author", stores the value in the variable strAuthor.

```
' VBScript
' get the name of the first attribute for the xmlBookElement element
Dim xmlAttribute, strAuthor
xmlAttribute = xmlBookElement.attributes(0)
If xmlAttribute.name = "Author" Then
   strAuthor = xmlAttribute.value
End If
```

```
// JScript
// get the name of the first attribute for the xmlBookElement element
var xmlAttribute, strAuthor;
xmlAttribute = xmlBookElement.attributes(0);
if (xmlAttribute.name == "Author") {
   strAuthor = xmlAttribute.value;
}
```

value

The value property gives the value of the attribute.

```
Variant = XMLDOMAttribute.value
```

If the attribute has more than one child node, this will be the value after parsing all entity references and concatenating the text. For example, suppose we have an entity `&wrox;` defined as:

```
<!ENTITY wrox "Wrox Press Limited">
```

If this is used within the `Copyright` attribute of an element such as:

```
<Book Copyright="Copyright 2000 &wrox;"> ... </Book>
```

Then the `value` property of this attribute will return:

```
"Copyright 2000 Wrox Press Limited"
```

XMLDOMCDATASection

The `XMLDOMCDATASection` object represents an escaped block of text in the XML document; this text is ignored by the XML parser, and may therefore contain characters which are otherwise illegal in XML, without needing each character to be escaped individually. They are therefore used to enclose sections of text which include many illegal characters.

The objects may be found in the child node lists in the node tree corresponding to their positions in the original document. For example, in the following document:

```
<SampleXML>
   <![CDATA[<Book Author="Kevin Williams"></Book>]]>
</SampleXML>
```

The `SampleXML` `XMLDOMElement` node would have one `XMLDOMCDATASection` node in its child node list, with the value `<Book Author="Kevin Williams"></Book>`.

`XMLDOMCDATASection` nodes may be accessed by finding them in the `XMLDOMNodeList` returned by the `childNodes` property of any node type that is allowed to contain escaped text blocks. For example, to get a reference to the `CDATA` section above, we would use:

```
xmlSampleXMLElement.childNodes.item(0)
```

The `XMLDOMCDATASection` object is derived from the `XMLDOMCharacterData` object, that itself is derived from the `XMLDOMNode` object. In addition to the `splitText` method of the `XMLDOMCDATASection` object, see the methods and properties defined for the `XMLDOMCharacterData` and `XMLDOMNode` objects for additional functionality available through the `XMLDOMCDATASection` object.

Methods	Properties	
`appendChild*`	`attributes*`	`ownerDocument*`
`appendData†`	`baseName*`	`parentNode*`
`cloneNode*`	`childNodes*`	`parsed*`

Table Continued on Following Page

Methods	Properties	
deleteData†	data†	prefix*
hasChildNodes*	dataType*	previousSibling*
insertBefore*	definition*	specified*
insertData†	firstChild*	text*
removeChild*	lastChild*	xml*
replaceChild*	length†	
replaceData†	namespaceURI*	
selectNodes*	nextSibling*	
selectSingleNode*	nodeName*	
splitText	nodeType*	
substringData†	nodeTypedValue*	
transformNode*	nodeTypeString*	
transformNodeToObject*	nodeValue*	

* See section on XMLDOMNode methods and properties.
† See section on XMLDOMCharacterData methods and properties.

We can embed an entire XML document in a CDATA section and then create a new node tree from that:

```
' VBScript
' Create a new node tree from the XML embedded in the xmlCatalogXML
' XMLDOMCDataSection object
Dim xmlCatalogDocument
Set xmlCatalogDocument = Server.CreateObject("Microsoft.XMLDOM")
xmlCatalogDocument.loadXML xmlCatalogXML.data
If xmlCatalogDocument.parseError.errorCode <> 0 Then
   ' the document did not parse - act accordingly
Else
   ' the document parsed - act accordingly
End If
```

```
// JScript
// Create a new node tree from the XML embedded in the xmlCatalogXML
// XMLDOMCDataSection object
var xmlCatalogDocument;
xmlCatalogDocument = Server.CreateObject("Microsoft.XMLDOM");
xmlCatalogDocument.loadXML (xmlCatalogXML.data);
if (xmlCatalogDocument.parseError.errorCode != 0) {
   // the document did not parse - act accordingly
} else {
   // the document parsed - act accordingly
}
```

Additional Methods

splitText

The `splitText` method splits the specified node into two nodes, breaking the data content apart at the *`offset`* location. It then creates an adjacent sibling node of the same type and inserts it at the appropriate location in the node tree.

```
xmlNewCDATASection = XMLDOMCDATASection.splitText(offset)
```

Parameter	Data Type	Description
`offset`	Long	Position at which node is split into two siblings

For example, if we take the XML element given above:

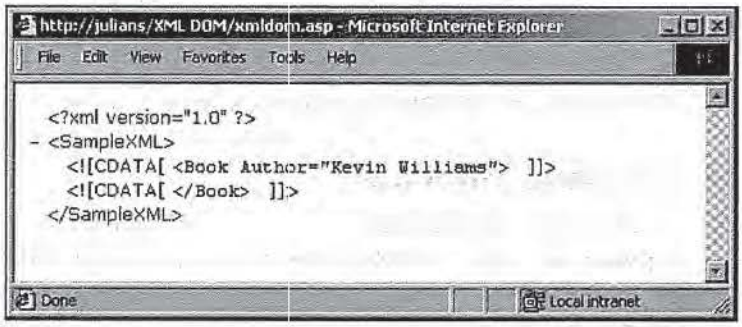
```
<SampleXML>
   <![CDATA[<Book Author="Kevin Williams"></Book>]]>
</SampleXML>
```

We can split this into two separate `CDATA` sections using the `splitText` method:

```
' VBScript
' xmlCatalogXML points to the CDATA section
Dim xmlNewCatalogCDATA
xmlNewCatalogCDATA = xmlCatalogXML.splitText(30)
```

```
// JScript
var xmlNewCatalogCDATA = xmlCatalogXML.splitText(30);
```

The `<SampleXML>` element now contains two `CDATA` sections:



Additional Properties

There are no additional properties associated with the `XMLDOMCDATASection` object.

XMLDOMCharacterData

The `XMLDOMCharacterData` object is not used directly by the Microsoft XML libraries. Instead, it is used as a common ancestor to other objects representing character data, namely `XMLDOMCDataSection`, `XMLDOMComment`, and `XMLDOMText`. Whenever a method or property of an object in the DOM returns one of these three types of objects, that object will support the methods and properties listed below.

The XMLDOMCharacterData object is derived from the XMLDOMNode object. In addition to the methods and properties described below, see the properties, methods, and events defined for the XMLDOMNode object for additional functionality available through the XMLDOMCharacterData object.

Methods	Properties	
appendChild*	attributes*	nodeValue*
appendData	baseName*	ownerDocument*
cloneNode*	childNodes*	parentNode*
deleteData	data	parsed*
hasChildNodes*	dataType*	prefix*
insertBefore*	definition*	previousSibling*
insertData	firstChild*	specified*
removeChild*	lastChild*	text*
replaceChild*	length	xml*
replaceData	namespaceURI*	
selectNodes*	nextSibling*	
selectSingleNode*	nodeName*	
substringData	nodeType*	
transformNode*	nodeTypedValue*	
transformNodeToObject*	nodeTypeString*	

* See section on XMLDOMNode methods and properties.

Additional Methods

There are five methods which are not inherited from the XMLDOMNode object: appendData, deleteData, insertData, replaceData, and substringData.

appendData

The appendData method appends the string in the *data* parameter to the end of the data currently contained in this node.

```
XMLDOMCharacterData.appendData(data)
```

Parameter	Data Type	Description
data	String	The data to be appended.

For example:

```
' VBScript
' modify the XMLDOMText node in xmlAuthorText to contain the author's whole
' name
If xmlAuthorText.data = "Williams" Then
   xmlAuthorText.appendData(", Kevin")
End If
```

```
// JScript
/* modify the XMLDOMText node in xmlAuthorText to contain the author's whole
   name */
if (xmlAuthorText.data == "Williams") {
   xmlAuthorText.appendData(", Kevin");
}
```

deleteData

The `deleteData` method deletes the substring of the data currently contained in this node beginning at the *`offset`* position (with zero indicating the first character in the string) and extending for *`count`* characters. If this extends beyond the end of the data currently in the node, the method deletes the data to the end of the string.

```
XMLDOMCharacterData.deleteData(offset, count)
```

Parameter	Data Type	Description
`offset`	Long	The position of the data to be deleted, i.e. characters from the start of the string
`count`	Long	The number of characters of data to be deleted

For example:

```
' VBScript
' modify the XMLDOMText node in xmlAuthorText to remove the author's first
' name
If xmlAuthorText.data = "Kevin Williams" Then
   xmlAuthorText.deleteData 0, 6
End If
```

```
// JScript
/* modify the XMLDOMText node in xmlAuthorText to remove the author's first
   name */
if (xmlAuthorText.data == "Kevin Williams") {
   xmlAuthorText.deleteData(0, 6);
}
```

insertData

The `insertData` method inserts the text specified in the `data` parameter into the data currently contained in this node at the `offset` position (with zero indicating the first character in the string).

```
XMLDOMCharacterData.insertData(offset, data)
```

Parameter	Data Type	Description
offset	Long	The position where the data is to be added, i.e. characters from the start of the string.
data	String	The data to be added.

```
' VBScript
' insert the author's first name into the xmlAuthorText XMLDOMText node
If xmlAuthorText.data = "Williams" Then
   xmlAuthorText.insertData 0, "Kevin "
End If
```

```
// JScript
// insert the author's first name into the xmlAuthorText XMLDOMText node
if (xmlAuthorText.data == "Williams") {
   xmlAuthorText.insertData(0, "Kevin ");
}
```

replaceData

The `replaceData` method deletes the substring of the data currently contained in this node beginning at the *offset* position (with zero indicating the first character in the string) and extending for *count* characters. It then inserts the text specified in the *data* parameter into the data currently contained in this node at the *offset* position.

```
XMLDOMCharacterData.replaceData(offset, count, data)
```

Parameter	Data Type	Description
offset	Long	The position of the data to be replaced, i.e. characters from the start of the string.
count	Long	The number of characters of data to be replaced.
data	String	The data to be added.

For example:

```
' VBScript
' correct the author's name in the XMLDOMText node xmlAuthorText
If xmlAuthorText.data = "George Williams" Then
   xmlAuthorText.replaceData 0, 6, "Kevin"
End If
```

```
// JScript
// correct the author's name in the XMLDOMText node xmlAuthorText
if (xmlAuthorText.data == "George Williams") {
   xmlAuthorText.insertData(0, 6, "Kevin");
}
```

substringData

The `substringData` method returns the portion of the data currently contained in this node beginning at the *offset* position (with zero indicating the first character in the string) and extending for *count* characters.

```
String = XMLDOMCharacterData.substringData(offset, count)
```

Parameter	Data Type	Description
offset	Long	The position of the data to be returned, i.e. characters from the start of the string.
count	Long	The number of characters of data to be returned.

For example:

```
' VBScript
' check to see if this book was written by a person named Kevin
Dim blnWrittenByKevin
blnWrittenByKevin = False
If xmlAuthorText.substringData(0, 5) = "Kevin" Then
   blnWrittenByKevin = True
End If
```

```
// JScript
// check to see if this book was written by a person named Kevin
var blnWrittenByKevin;
blnWrittenByKevinWilliams = false;
if (xmlAuthorText.substringData(0, 5) == "Kevin") {
   blnWrittenByKevin = true;
}
```

Additional Properties

As well as the properties inherited from `XMLDOMNode`, the `XMLDOMCharacterData` object exposes two properties of its own: `data` and `length`.

data

The `data` property sets or returns a string representing the data contained in this node.

```
XMLDOMCharacterData.data = String
String = XMLDOMCharacterData.data
```

```
' VBScript
' set the xmlAuthorText node's value to Unknown
xmlAuthorText.data = "Unknown"
```

```
// JScript
// set the xmlAuthorText node's value to Unknown
xmlAuthorText.data = "Unknown";
```

length

The read-only `length` property returns the number of characters in the data contained in this node.

```
Long = XMLDOMCharacterData.length
```

```
' VBScript
' get the length of the xmlAuthorText node's text and store it in a variable
Dim intAuthorLength
intAuthorLength = xmlAuthorText.length
```

```
// JScript
// get the length of the xmlAuthorText node's text and store it in a variable
var intAuthorLength = xmlAuthorText.length;
```

XMLDOMComment

The XMLDOMComment object represents a comment in the original XML document. The objects may be found in the child node lists in the node tree corresponding to their position in the original document. For example, in the following document:

```
<!-- This is the first comment -->
<Book Author="Kevin Williams">
   <!-- This is the second comment -->
</Book>
```

Two XMLDOMComment nodes would appear in the node tree: one with the text "This is the first comment", as the first child of the XMLDOMDocument object, and the second with the text "This is the second comment", as the first child of the XMLDOMElement object corresponding to the <Book> element.

XMLDOMComment nodes may be accessed by finding them in the XMLDOMNodeList returned by the childNodes property of any node type that is allowed to contain comments. For example, the first comment in the XML document above could be referenced using:

```
xmlDocument.childNodes.item(0)
```

The XMLDOMComment object is derived from the XMLDOMCharacterData object, that itself is derived from the XMLDOMNode object. There are no methods or properties unique to the XMLDOMComment object, so please see the methods and properties defined for the XMLDOMCharacterData and XMLDOMNode objects for the functionality available through the XMLDOMComment object.

Methods	Properties	
appendChild*	attributes*	nodeValue*
appendData†	baseName*	ownerDocument*
cloneNode*	childNodes*	parentNode*
deleteData†	data†	parsed*
hasChildNodes*	dataType*	prefix*
insertBefore*	definition*	previousSibling*
insertData†	firstChild*	specified*
removeChild*	lastChild*	text*
replaceChild*	length†	xml*
replaceData†	namespaceURI*	
selectNodes*	nextSibling*	
selectSingleNode*	nodeName*	
substringData†	nodeType*	
transformNode*	nodeTypedValue*	
transformNodeToObject*	nodeTypeString*	

* See section on XMLDOMNode methods and properties.
† See section on XMLDOMCharacterData methods and properties.

The following code retrieves all comment nodes for a <Book> element and concatenates them together into one string:

```
' VBScript
Dim i, strAllComments
strAllComments = ""
For i = 0 To xmlBookElement.childNodes.length - 1
   ' Check if this child node is a comment (COMMENT_NODE = 8)
   If xmlBookElement.childNodes(i).nodeType = NODE_COMMENT Then
      strAllComments = strAllComments & " " &
xmlBookElement.childNodes(i).data
   End If
Next
```

```
// JScript
var i, strAllComments;
strAllComments = "";
for (i = 0; i < xmlBookElement.childNodes.length; i++) {
   // Check if this child node is a comment (COMMENT_NODE == 8)
   if (xmlBookElement.childNodes(i).nodeType == 8) {
      strAllComments += " " + xmlBookElement.childNodes(i).data;
   }
}
```

XMLDOMDocument

The XMLDOMDocument object represents an entire XML document. This is therefore the topmost node of the XML DOM tree. It will always be the first object your script references – typically by either creating a new instance and building the node tree from scratch, or using the load or loadXML methods to initialize the node tree.

A new XMLDOMDocument object may be obtained by using the appropriate object creation function to create an instance of XMLDOMDocument:

```
' VBScript
Dim xmlDocument
Set xmlDocument = Server.CreateObject("Microsoft.XMLDOM)
```

```
// JScript
var = Server.CreateObject("Microsoft.XMLDOM");
```

The XMLDOMDocument object is derived from the XMLDOMNode object. In addition to the 15 methods, 10 properties, and three events of the XMLDOMDocument object detailed below, see the properties, methods, and events defined for the XMLDOMNode object for additional functionality available through the XMLDOMDocument object.

Methods	Properties	Events
abort	async	ondataavailable
appendChild*	attributes*	onreadystatechange
cloneNode*	baseName*	ontransformnode
createAttribute	childNodes*	
createCDATASection	dataType*	
createComment	definition*	
createDocumentFragment	doctype	
createElement	documentElement	
createEntityReference	firstChild*	
createNode	implementation	
createProcessing Instruction	lastChild*	
createTextNode	namespaceURI*	
getElementsByTagName	nextSibling*	
hasChildNodes*	nodeName*	
insertBefore*	nodeType*	
load	nodeTypedValue*	
loadXML	nodeTypeString*	
nodeFromID	nodeValue*	
removeChild*	ownerDocument*	
replaceChild*	parentNode*	
save	parsed*	
selectNodes*	parseError	
selectSingleNode*	preserveWhite Space	
transformNode*	previousSibling*	
transformNodeToObject*	readyState	
	resolveExternals	
	specified*	
	text*	
	url	
	validateOnParse	
	xml*	

* See section on XMLDOMNode methods and properties.

Additional Methods

abort

The `abort` method aborts the current load and parse operation if there is one in progress. Any portion of the XML node tree that has been built is discarded.

```
XMLDOMDocument.abort()
```

createAttribute

The `createAttribute` method creates a new `XMLDOMAttribute` object with the given name.

```
XMLDOMAttribute = XMLDOMDocument.createAttribute(name)
```

Parameter	Data Type	Description
name	String	The name of the attribute to be created.

This returns the new `XMLDOMAttribute` object. Note that the attribute will still need to be added to the child attribute list of the element to which it corresponds. If you need to create a namespace-qualified attribute, use the `createNode` method instead. The value of the attribute may be set by calling the `setAttribute` method on the new object.

For example, the following code creates an `Author` attribute in the `xmlCatalogDocument` document on the `xmlBookElement` element:

```
' VBScript
Dim xmlAuthorAttribute
Set xmlAuthorAttribute = xmlCatalogDocument.createAttribute("Author")
xmlAuthorAttribute.value = "Kevin Williams"
xmlBookElement.setAttributeNode(xmlAuthorAttribute)
```

```
// JScript
var xmlAuthorAttribute = xmlCatalogDocument.createAttribute("Author");
xmlAuthorAttribute.value = "Kevin Williams";
xmlBookElement.setAttributeNode(xmlAuthorAttribute);
```

createCDATASection

The `createCDATASection` method creates a new `XMLDOMCDATASection` object with the given data.

```
XMLDOMCDATASection = XMLDOMDocument.createCDATASection(data)
```

Parameter	Data Type	Description
data	String	The data to be inserted into the new `XMLDOMCDATASection` object.

The new XMLDOMCDATASection object is returned. Note that the XMLDOMCDATASection object will still need to be added to the node tree in the appropriate location.

```
' VBScript
' Create a CDATASection child node in the xmlRawCodeElement element
Dim xmlRawCodeCDATA
Set xmlRawCodeCDATA = xmlCatalogDocument.createCDATASection("<Book></Book>")
xmlRawCodeElement.appendChild xmlRawCodeCDATA
```

```
// JScript
// Create a CDATASection child node in the xmlRawCodeElement element
var xmlRawCodeCDATA;
xmlRawCodeCDATA = xmlCatalogDocument.createCDATASection("<Book></Book>");
xmlRawCodeElement.appendChild(xmlRawCodeCDATA);
```

createComment

The createComment method creates a new XMLDOMComment object with the given text.

```
XMLDOMComment = XMLDOMDocument.createComment(data)
```

Parameter	Data Type	Description
data	String	The data to be inserted into the newly-created XMLDOMComment object.

This method returns the new XMLDOMComment object. Note that the XMLDOMComment object will still need to be added to the node tree in the appropriate location.

createDocumentFragment

The createDocumentFragment method creates a new, empty XMLDOMDocumentFragment object.

```
XMLDOMDocumentFragment = XMLDOMDocument.createDocumentFragment(name)
```

Parameter	Data Type	Description
name	String	The name of the XMLDOMDocumentFragment object to be created.

This method returns the new XMLDOMDocumentFragment object. Note that the XMLDOMDocumentFragment object will still need to be added to the node tree in the appropriate location once it has been populated.

createElement

The createElement method creates a new XMLDOMElement object with the given tag name.

```
XMLDOMElement = XMLDOMDocument.createElement(tag_name)
```

Parameter	Data Type	Description
tag_name	String	The name of the element to be created.

This method returns the new `XMLDOMElement` object. Note that the `XMLDOMElement` object will still need to be added to the node tree in the appropriate location. If you need to create a namespace-qualified element, use the `createNode` method instead.

createEntityReference

The `createEntityReference` method creates a new `XMLDOMEntityReference` object with the given name.

```
XMLDOMEntityReference = XMLDOMDocument.createEntityReference(name)
```

Parameter	Data Type	Description
name	String	The name of `XMLDOMEntityReference` object to be created.

This method returns the new `XMLDOMEntityReference` object. Note that the `XMLDOMEntityReference` object will still need to be added to the node tree in the appropriate location.

createNode

The `createNode` method creates a new `XMLDOMNode` object with the given type, name, and namespace URI. The possible types are enumerated in the `XMLDOMNodeType` enumeration. If the specified name contains a namespace prefix, that namespace prefix will be associated with the provided URI; otherwise, it will be treated as the default namespace.

```
XMLDOMNode = XMLDOMDocument.createNode(type, name, namespaceURI)
```

Parameter	Data Type	Description
`type`	Long	The type of `XMLDOMNode` object to be created. For possible values, see the `nodeType` property of the `XMLDOMNode` object.
`name`	String	The name of the node to be created.
`namespace URI`	String	The definition for the node's namespace.

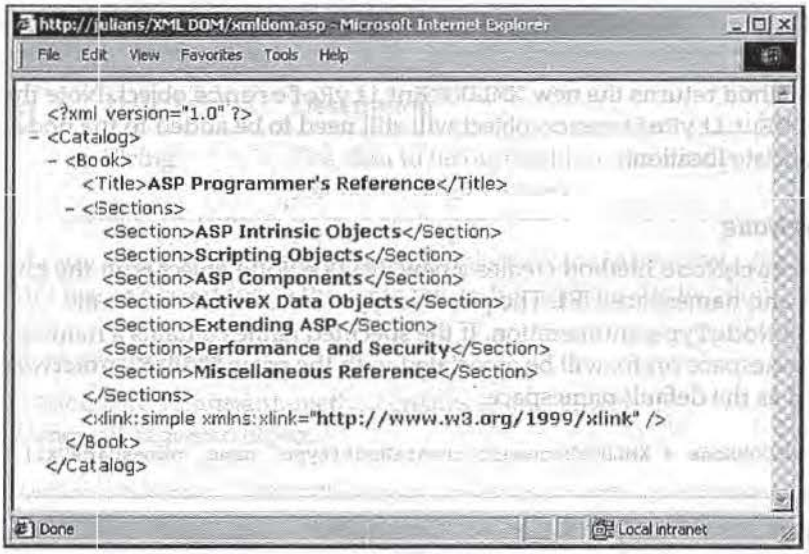
This method returns the new `XMLDOMNode` object. For nodes that do not have names, the `name` parameter should be passed as an empty string. Note that the `XMLDOMNode` object will still need to be added to the node tree in the appropriate location.

For example:

```
' VBScript
' Create an xlink:simple element in the xmlBookElement element
Dim xmlXLinkSimpleElement
Set xmlXLinkSimpleElement = xmlCatalogDocument.createNode (NODE_ELEMENT, _
                                                           "xlink:simple", _
                                                           "http://www.w3.org/1999/xlink")
xmlBookElement.appendChild(xmlXLinkSimpleElement)
```

```
// JScript
// Create an xlink:simple element in the xmlBookElement element
var xmlXLinkSimpleElement;
xmlXLinkSimpleElement = xmlCatalogDocument.createNode (NODE_ELEMENT,
                                                       "xlink:simple",
                                                       "http://www.w3.org/1999/xlink");
xmlBookElement.appendChild(xmlXLinkSimpleElement);
```

This adds an `<xlink:simple>` element as the final child node of the `<Book>` element:



createProcessingInstruction

The `createProcessingInstruction` method creates a new `XMLDOMProcessingInstruction` object with the given target and data.

```
XMLDOMProcessingInstruction =
                    XMLDOMDocument.createProcessingInstruction(target, data)
```

Parameter	Data Type	Description
target	String	The name of the application which will process the instruction
data	String	The data to be passed to the external application

This method returns the new `XMLDOMProcessingInstruction` object. Note that the `XMLDOMProcessingInstruction` object will still need to be added to the node tree in the appropriate location.

createTextNode

The `createTextNode` method creates a new `XMLDOMText` object with the given text data.

```
XMLDOMTextNode = XMLDOMDocument.createTextNode(data)
```

Parameter	Data Type	Description
data	String	The data the new `XMLDOMText` object will contain.

This method returns the new `XMLDOMText` object. Note that the `XMLDOMText` object will still need to be added to the node tree in the appropriate location.

getElementsByTagName

The `getElementsByTagName` method returns an `XMLDOMNodeList` object containing all the elements in the document with the specified *tagname*, in the order in which they appear in the original document. If no elements in the document have the specified *tag_name*, an empty `XMLDOMNodeList` is returned.

```
XMLDOMNodeList = XMLDOMDocument.getElementsByTagName(tag_name)
```

Parameter	Data Type	Description
tag_name	String	The name of the elements to search for.

```
' VBScript
' Select all the Book elements in the document
Dim xmlBookNodeList
Set xmlBookNodeList = xmlCatalogDocument.getElementsByTagName("Book")
```

```
// JScript
// Select all the Book elements in the document
var xmlBookNodeList = xmlCatalogDocument.getElementsByTagName("Book");
```

load

The `load` method loads the XML document at the specified URL and parses it.

```
Boolean = XMLDOMDocument.load(filename)
```

Parameter	Data Type	Description
filename	Variant	The full URL or physical path of the XML file to be loaded into the document.

This method returns `true` if the document was loaded successfully, or `false` otherwise. If the URL cannot be resolved or does not contain an XML document, this method returns an error. Calling this method clears the current node tree for the document. If parsing errors are encountered when parsing the document, they are indicated in the `parseError` property of the `XMLDOMDocument` object.

```
' VBScript
Dim xmlCatalogDocument
Set xmlCatalogDocument = Server.CreateObject(Microsoft.XMLDOM)
xmlCatalogDocument.load "http://myserver/books.xml"
```

```
// JScript
var xmlCatalogDocument = Server.CreateObject("Microsoft.XMLDOM");
xmlCatalogDocument.load("http://myserver/books.xml");
```

loadXML

The `loadXML` method loads the XML document contained in the *`xmlString`* parameter and parses it.

```
Boolean = XMLDOMDocument.loadXML(xmlString)
```

Parameter	Data Type	Description
`xmlString`	String	The XML document to be loaded.

The method returns `true` if the XML string was loaded successfully, or `false` otherwise. Calling this method clears the current node tree for the document.

nodeFromID

The `nodeFromID` method returns the `XMLDOMNode` object for the element in the document with an `ID` attribute that matches the *`idString`* parameter. If no node matches the provided `ID`, this method returns `NULL`.

```
XMLDOMNode = XMLDOMDocument.nodeFromID(idString)
```

Parameter	Data Type	Description
`idString`	String	The value of the `ID` attribute to be sought in the document.

For this method to work, the node being sought must have an attribute defined as type `ID` in the DTD. For example, if we have a `<Book>` element with a unique identifying attribute called `BookID`, this attribute might be defined in the DTD as:

```
<!ATTLIST Book BookID ID #REQUIRED>
```

We can then use this attribute to provide a unique identifier for each `<Book>` element:

```
<Book BookID="ASP3ProgRef">
```

And we can easily retrieve this node (without needing to know its position in the XML document) using the `nodeFromID` method:

```
' VBScript
Dim xmlBookElement
Set xmlBookElement = xmlCatalogDocument.nodeFromID("ASP3ProgRef")
```

```
// JScript
var xmlBookElement = xmlCatalogDocument.nodeFromID("ASP3ProgRef");
```

save

The `save` method saves the XML for the current document to the target specified in the *`objTarget`* parameter.

```
XMLDOMDocument.save(objTarget)
```

Parameter	Data Type	Description
`objTarget`	Variant	If `objTarget` is a string, it represents a local filename to which the XML is to be written. If the file already exists, it is replaced. In a multi-user scenario, this could result in the file being overwritten and data lost, unless precautions are taken to ensure that the file is given a unique name each time it is saved. If `objTarget` is the `Response` object, the XML is sent back as the response to the calling client. If `objTarget` is an `XMLDOMDocument` object, the entire object is cloned and recreated in the new `XMLDOMDocument` object's node tree. Note that this actually persists the original document as text and then reparses it to create the new object's node tree, making this a good way to verify the persistability of your document. If `objTarget` is any other COM object that provides an `IStream`, `IPersistStream`, or `IPersistStreamInit` interface, the XML will be passed to that interface and the object will act on the XML accordingly.

We can use the `save` method to send an XML document to the browser by specifying the `Response` object as the target:

```
' VBScript
Dim xmlCatalogDocument
Set xmlCatalogDocument = Server.CreateObject("Microsoft.XMLDOM")
xmlCatalogDocument.async = false
xmlCatalogDocument.load "http://servername/xml/books.xml"
xmlCatalogDocument.save Response
```

```
// JScript
var xmlCatalogDocument = Server.CreateObject("Microsoft.XMLDOM");
xmlCatalogDocument.async = false;
xmlCatalogDocument.load("http://servername/xml/books.xml");
xmlCatalogDocument.save(Response);
```

Additional Properties

async

The `async` property is a boolean flag that indicates whether asynchronous operation is permitted.

```
XMLDOMDocument.async = Boolean
Boolean = XMLDOMDocument.async
```

When set to `true` (the default), control will return immediately to your script after calling the `load` or `loadXML` methods. If you want to use the `XMLDOMDocument` object asynchronously, you should set an event handler for the `onreadystatechange` event (see the `onreadystatechange` property below) before loading your document.

doctype

The `doctype` property is a read-only property that returns the `XMLDOMDocumentType` object that contains information about the Document Type Definition associated with the document. If there is no DTD for the document, or the document is not XML, this property returns `NULL`.

```
XMLDOMDocumentType = XMLDOMDocument.doctype
```

documentElement

The `documentElement` property sets or returns the `XMLDOMElement` node representing the outermost element of the document. Every XML document may have only one element immediately below the document root (although it may have other nodes, such as comments or processing instructions).

```
XMLDOMDocument.documentElement = XMLDOMElement
XMLDOMElement = XMLDOMDocument.documentElement
```

For example:

```
' VBScript
Dim xmlNewDocument
Set xmlNewDocument = Server.CreateObject("Microsoft.XMLDOM")
Set xmlNewDocument.documentElement = xmlNewDocument.createElement("Catalog")
```

```
// JScript
var xmlNewDocument = Server.CreateObject("Microsoft.XMLDOM");
xmlNewDocument.documentElement = xmlNewDocument.createElement("Catalog");
```

Implementation

The read-only `implementation` property returns the `XMLDOMImplementation` object that contains information about the version of the XML parser used to create or parse this document.

```
XMLDOMImplementation = XMLDOMDocument.implementation
```

parseError

The read-only `parseError` property returns an `XMLDOMParseError` object describing the status of the last parse performed by this instance of the XML DOM. You should always check the `parseError` property of a document after calling the `load` or `loadXML` methods to ensure that the document has been parsed correctly. A list of errors raised by the MSXML parse can be found in Appendix M.

```
XMLDOMParseError = XMLDOMDocument.parseError
```

For example:

```
' VBScript
' Check to see if the document parsed successfully
If xmldocument.parseError.errorCode <> 0 Then
   ' the parse failed - take some corrective action
Else
   ' the parse succeeded - continue normally
End If
```

```
// JScript
// Check to see if the document parsed successfully
if (xmlDocument.parseError.errorCode != 0) {
  // the parse failed - take some corrective action
} else {
  // the parse succeeded - continue normally
}
```

preserveWhiteSpace

The `preserveWhiteSpace` property is a Boolean flag indicating whether whitespace should be preserved in the values of the `text` and `xml` properties for nodes in this document. If this flag is set to `true`, whitespace will always be preserved as if there were an `xml:space="preserve"` attribute on every element in the document. If the flag is set to `false`, whitespace will only be preserved for nodes where the `xml:space="preserve"` attribute is present. The default is `false`.

```
XMLDOMDocument.preserveWhiteSpace = Boolean
Boolean = XMLDOMDocument.preserveWhiteSpace
```

This property is unlikely to have any effect when sending an XML document to the browser.

readyState

The read-only `readyState` property returns a long integer that describes the status of the `load` or `loadXML` method.

```
Long = XMLDOMDocument.readyState
```

The possible values are:

- 0 ("unitialized"), i.e. the object has been created but the `load` method has not yet been executed.
- 1 ("loading"), i.e. the document is loading, but has not yet entered the parsing step.
- 2 ("loaded"), i.e. the document has been loaded and is being parsed.
- 3 ("interactive"), i.e. the document has been partially parsed, and a read-only version of the object model is available.
- 4 ("complete"), i.e. the document has been completely parsed, successfully or unsuccessfully.

For example:

```
' VBScript
Function checkCatalogDocumentState()
   ' This function handles the onreadystatechange event for the Catalog
   ' document load
   If xmlCatalogDocument.readyState = 4 Then ' Complete
      If xmldoc.parseError.errorCode = 0 Then
         blnCatalogParsedSuccessfully = True
      Else
         blnCatalogParsedSuccessfully = False
      End If
   End If
Set checkCatalogDocumentState = xmlCatalogDocument
End Function
```

```
// JScript
function checkCatalogDocumentState() {
   // This function handles the onreadystatechange event for the Catalog
   // document load
   if (xmlCatalogDocument.readyState == 4) { // Complete
      if (xmldoc.parseError.errorCode == 0) {
         blnCatalogParsedSuccessfully = 1;
      } else {
         blnCatalogParsedSuccessfully = 0;
      }
   }
}
```

resolveExternals

The `resolveExternals` property is a boolean flag indicating whether external definitions (resolvable namespaces, DTD external subsets, and external entity references) should be resolved at parse time. The default value is `true`.

```
XMLDOMDocument.resolveExternals = Boolean
Boolean = XMLDOMDocument.resolveExternals
```

url

The read-only `url` property returns a string containing the URL of the document that was last successfully loaded using the `load` method. If no document has been loaded with this method, this property returns `NULL`.

```
String = XMLDOMDocument.url
```

validateOnParse

The `validateOnParse` property is a Boolean flag indicating whether the parser should validate the document against a DTD or schema if one is present. If this flag is false, the parser will ignore a provided DTD or schema and only parse for well-formedness. The default value is `true`.

```
XMLDOMDocument.validateOnParse = Boolean
Boolean = XMLDOMDocument.validateOnParse
```

Events

The event handlers for the XML DOM are defined in VBScript and JScript by setting the value of a write-only property with the same name as the event to the name of the callback procedure used to respond to the event. Note that in VBScript a Type mismatch error will be raised if this event handler isn't a function which returns an object.

ondataavailable

The `ondataavailable` event is fired whenever a new chunk of data has been successfully loaded from a remote source. It could be used to indicate the status of a large document load to the user.

```
' VBScript
' set up an event handler for the ondataavailable event
xmlCatalogDocument.ondataavailable = checkCatalogDataAvailable

Function checkCatalogDataAvailable()
   ' handle the ondataavailable event for the Catalog document
   Set checkCatalogDataAvailable = xmlCatalogDocument
End Function
```

```
// JScript
// set up an event handler for the ondataavailable event
xmlCatalogDocument.ondataavailable = checkCatalogDataAvailable;

function checkCatalogDataAvailable() {
   // handle the ondataavailable event for the Catalog document
}
```

onreadystatechange

The `onreadystatechange` event is fired whenever the value of the `readyState` property for the object changes. When used in conjunction with the `async` property, this allows your script to continue executing while waiting for an XML document to be loaded.

```
' VBScript
' set up an event handler for the onreadystatechange event
xmlCatalogDocument.onreadystatechange = checkCatalogReadyStateChange

Function checkCatalogReadyStateChange()
   ' handle the onreadystatechange event for the Catalog document
   Set checkCatalogReadyStateChange = xmlCatalogDocument
End Function
```

```
// JScript
// set up an event handler for the onreadystatechange event
xmlCatalogDocument.onreadystatechange = checkCatalogReadyStateChange;

function checkCatalogReadyStateChange() {
   // handle the onreadystatechange event for the Catalog document
}
```

ontransformnode

The `ontransformnode` event is fired each time a node in a source document is about to be transformed using a node in a stylesheet. This event could be used to report the status of a long transformation to the user.

```
' VBScript
' set up an event handler for the ontransformnode event
xmlCatalogDocument.ontransformnode = checkCatalogTransformNode

Function checkCatalogTransformNode()
   ' handle the ontransformnode event for the Catalog document
   Set checkCatalogTransformNode = xmlCatalogDocument
End Function
```

```
// JScript
// set up an event handler for the ontransformnode event
xmlCatalogDocument.ontransformnode = checkCatalogTransformNode;

function checkCatalogTransformNode() {
   // handle the ontransformnode event for the Catalog document
}
```

XMLDOMDocument2

The `XMLDOMDocument2` object extends the `XMLDOMDocument` object with additional functionality. It does not replace the original `XMLDOMDocument` object for reasons of backward compatibility. It supports schema caching and additional validation features, as well as providing support for XPath.

> **This object is only available in version 2.6 of MSXML.**

A new `XMLDOMDocument2` object may be obtained by using the appropriate object creation function to create an instance of `XMLDOMDocument2`:

```
' VBScript
Dim xmlDocument2
Set xmlDocument2 = Server.CreateObject(MSXML2.DOMDocument)
```

```
// JScript
var xmlDocument2;
xmlDocument2 = Server.CreateObject("MSXML2.DOMDocument");
```

The `XMLDOMDocument2` is by default apartment-threaded, but it is also available in a free-threaded version. This version exposes exactly the same methods and properties, and can be instantiated using the Prog ID "`MSXML2.FreethreadedDOMDocument`". As a general rule, it is probably a good idea to avoid using free-threaded objects in ASP.

The XMLDOMDocument2 object extends the functionality of the XMLDOMDocument object. In addition to the 4 methods and 2 properties of the XMLDOMDocument object detailed below, see the properties, methods, and events defined for the XMLDOMDocument object for additional functionality available through the XMLDOMDocument2 object.

Methods	Properties	Events
abort†	async†	ondataavailable†
appendChild*	attributes*	onreadystatechange†
cloneNode*	baseName*	ontransformnode†
createAttribute†	childNodes*	
createCDATASection†	dataType*	
createComment†	definition*	
createDocumentFragment†	doctype†	
createElement†	documentElement†	
createEntityReference†	firstChild*	
createNode†	implementation†	
createProcessing Instruction†	lastChild* namespaces	
createTextNode†	namespaceURI*	
getElementsByTagName†	nextSibling*	
getProperty	nodeName*	
hasChildNodes*	nodeType*	
insertBefore*	nodeTypedValue*	
load†	nodeTypeString*	
loadXML†	nodeValue*	
nodeFromID†	ownerDocument*	
removeChild*	parentNode*	
replaceChild*	parsed*	
save†	parseError†	
selectNodes‡	preserveWhiteSpace†	
selectSingleNode*	previousSibling*	
setProperty	readyState†	

Table Continued on Following Page

Methods	Properties	Events
transformNode*	resolveExternals†	
	schemas	
transformNodeToObject*	specified*	
validate	text*	
	url†	
	validateOnParse†	
	xml*	

* See section on XMLDOMNode methods and properties.
† See section on XMLDOMDocument methods properties and events.
‡ The XMLDOMDocument2 object redefines this method.

Additional Methods

getProperty

The getProperty method is used to obtain the value of a named property for the document. In version 2.6, the only supported property is "SelectionLanguage" which may have the value "XPath" or "XSLPattern". The value of this property governs the way queries passed to selectNode and selectSingleNode are interpreted for nodes in this document.

```
String = XMLDOMDocument2.getProperty(name)
```

Parameter	Data Type	Description
name	String	The name of the property to be returned by this call.

selectNodes

The selectNodes method is used to retrieve a set of nodes from the document. For the XMLDOMDocument2 object, an XMLDOMSelection object is returned. The parameter still represents the XSLT or XSL pattern query (based on the value of the SelectionLanguage property – see setProperty and getProperty) that will be used to create the result.

```
XMLDOMSelection = XMLDOMDocument2.selectNodes(strExpr)
```

Parameter	Data Type	Description
strExpr	String	The XSL pattern or XPath expression string used to execute the query.

For example, to find all the <Chapter> elements in the document:

```
' VBScript
Dim xmlSelectionChapterElements
xmlSelectionChapterElements = xmlCatalogDocument2.selectNodes("//Chapter")
```

```
// JScript
// find all the Chapter elements in the node
var xmlSelectionChapterElements;
xmlSelectionChapterElements = xmlCatalogDocument2.selectNodes("//Chapter");
```

setProperty

The `setProperty` method is used to set the value of a named property for the document. In version 2.6, the only supported property is `"SelectionLanguage"`; it may have the value `XPath` or `XSLPattern`. The value of this property governs the way queries passed to `selectNode` and `selectSingleNode` are interpreted for nodes in this document.

```
XMLDOMDocument2.setProperty(name, value)
```

Parameter	Data Type	Description
name	String	The name of the property that is to be set.
value	String	The value to which the property is to be set.

For example, to set the selection language to XPath:

```
' VBScript
xmlCatalogDocument2.setProperty("SelectionLanguage", "XPath")
```

```
// JScript
xmlCatalogDocument2.setProperty("SelectionLanguage", "XPath");
```

validate

The `validate` method performs runtime validation of a document using the currently loaded DTD, schema, or schema collection. It returns error information indicating what error, if any, was encountered while validating the document.

```
Integer = XMLDOMDocument2.validate
```

Additional Properties

namespaces

The `namespaces` property returns an `XMLDOMSchemaCollection` object that contains a list of the namespaces used in the document. This property is read-only.

```
XMLDOMSchemaCollection = XMLDOMDocument2.namespaces
```

schemas

The `schemas` property sets or returns an `XMLDOMSchemaCollection` object that contains a list of the precached schemas to be used in the validation of a document loaded with the `load` or `loadXML` methods.

```
XMLDOMSchemaCollection = XMLDOMDocument2.schemas
XMLDOMDocument2.schemas = XMLDOMSchemaCollection
```

XMLDOMDocumentFragment

The XMLDOMDocumentFragment object does not correspond directly to any part of an XML document *per se*. Instead, it represents a well-formed fragment of a node tree, and may be used to construct a portion of an XML document before attaching it to the main document. When the XMLDOMDocumentFragment node is attached to the main document's node tree, all of the child nodes of the fragment are attached in its place.

A new XMLDOMDocumentFragment object for a document may be obtained by calling the createDocumentFragment method on the XMLDOMDocument object for that document.

The XMLDOMDocumentFragment object is derived from the XMLDOMNode object. The XMLDOMDocumentFragment object exposes no additional methods or properties, so see the XMLDOMNode section for the functionality available through this object.

Methods	Properties	
appendChild*	attributes*	nodeTypedValue*
cloneNode*	baseName*	nodeTypeString*
hasChildNodes*	childNodes*	nodeValue*
insertBefore*	dataType*	ownerDocument*
removeChild*	definition*	parentNode*
replaceChild*	firstChild*	parsed*
selectNodes*	lastChild*	prefix*
selectSingleNode*	namespaceURI*	previousSibling*
transformNode*	nextSibling*	specified*
transformNodeToObject*	nodeName*	text*
	nodeType*	xml*

* See section on XMLDOMNode methods and properties.

XMLDOMDocumentType

The XMLDOMDocumentType object contains information about the document type definition (DTD) associated with a particular document. If there is no DTD associated with a document, no XMLDOMDocumentType object will be available.

The XMLDOMDocumentType object for a particular document may be obtained through the doctype property of the XMLDOMDocument object for that document. For example, if we have an XML document with an internal DTD such as:

```
<?xml version="1.0"?>
<!DOCTYPE Catalog [
   <!ELEMENT Catalog (Book+)>
   <!ELEMENT Book (Publisher, Sections)>
   <!ELEMENT Publisher (#PCDATA)>
   <!ELEMENT Sections (Section+)>
```

```
    <!ELEMENT Section (#PCDATA)>
    <!ENTITY wrox "Wrox Press Limited">
]>
<Catalog>
   <Book>
      <Publisher>&wrox;</Publisher>
      <Sections>
         <Section>ASP Intrinsic Objects</Section>
         <Section>Scripting Objects</Section>
         <Section>ASP Components</Section>
         <Section>ActiveX Data Objects</Section>
         <Section>Extending ASP</Section>
         <Section>Miscellaneous Reference</Section>
      </Sections>
   </Book>
</Catalog>
```

We can get a reference to the `XMLDOMDocumentType` using the code:

```
' VBScript
Dim xmlCatalogDocument, xmlDoctype
Set xmlCatalogDocument = Server.CreateObject("Microsoft.XMLDOM")
xmlCatalogDocument.async = false
xmlCatalogDocument.load("http://servername/xml/books.xml")
Set xmlCatalogDoctype = xmlCatalogDocument.doctype
```

```
// JScript
var xmlCatalogDocument = Server.CreateObject("Microsoft.XMLDOM");
xmlCatalogDocument.async = false;
xmlCatalogDocument.load("http://servername/xml/books.xml");
var xmlCatalogDoctype = xmlCatalogDocument.doctype;
```

The `XMLDOMDocumentType` object is derived from the `XMLDOMNode` object. In addition to the three properties detailed below, see the methods and properties defined for the `XMLDOMNode` object for additional functionality available through the `XMLDOMDocumentType` object.

Methods	Properties	
appendChild*	attributes*	nodeTypedValue*
cloneNode*	baseName*	nodeTypeString*
hasChildNodes*	childNodes*	nodeValue*
insertBefore*	dataType*	notations
removeChild*	definition*	ownerDocument*
replaceChild*	entities	parentNode*
selectNodes*	firstChild*	parsed*
selectSingleNode*	lastChild*	prefix*
transformNode*	name	previousSibling*
transformNodeToObject*	namespaceURI*	specified*
	nextSibling*	text*
	nodeName*	xml*
	nodeType*	

* See section on `XMLDOMNode` methods and properties.

Additional Methods

There are no additional methods associated with the XMLDOMDocumentType object.

Additional Properties

entities

The read-only entities property returns an XMLDOMNamedNodeMap object containing a list of all the entities, both internal and external, that are declared in the document's DTD.

```
XMLNamedNodeMap = XMLDOMDocumentType.entities
```

For example:

```
' VBScript
Dim xmlCatalogEntities, xmlCatalogDoctype
Set xmlCatalogDoctype = xmlCatalogDocument.doctype
Set xmlCatalogEntities = xmlCatalogDoctype.entities
```

```
// JScript
var xmlCatalogDoctype = xmlCatalogDocument.doctype;
var xmlCatalogEntities = xmlCatalogDoctype.entities;
```

For the XML document above, the returned XMLDOMNamedNodeMap will contain just one item: the entity &wrox;.

name

The read-only name property returns the name of the document type. This will be the same as the tag of the root element of the document tree (in the example given above, this will be "Catalog").

```
String = XMLDOMDocumentType.name
```

notations

The read-only notations property returns an XMLNamedNodeMap object containing a list of all the notations declared in the document's DTD.

```
XMLNamedNodeMap = XMLDOMDocumentType.notations
```

XMLDOMElement

The XMLDOMElement object represents an element in the XML document. For example, in this document:

```
<Book>
   <Chapter>Chapter 1</Chapter>
</Book>
```

There would be two XMLDOMElement elements: one for the <Book> element and one for the <Chapter> element.

A new XMLDOMElement object may be created for a document by calling the createElement method on the XMLDOMDocument object for that document. Existing XMLDOMElement objects may be obtained by calling the getElementsByTagName method on any document or element:

```
' VBScript
' Get first Chapter element in document
Dim xmlChapterElement
Set xmlChapterElement = xmlDocument.getElementsByTagName("Chapter").item(0)
```

```
// JScript
// Get first Chapter element in document
var xmlChapterElement = xmlDocument.getElementsByTagName("Chapter").item(0);
```

Alternatively, we can look in the `childNodes` list directly:

```
' VBScript
Dim xmlChapterElement
Set xmlChapterElement = xmlBookElement.childNodes.item(0)
```

```
// JScript
var xmlChapterElement = xmlBookElement.childNodes.item(0);
```

The `XMLDOMElement` object is derived from the `XMLDOMNode` object. In addition to the methods and properties described below, please see the properties, methods, and events defined for the `XMLDOMNode` object for additional functionality available through the `XMLDOMElement` object.

Methods		Properties	
appendChild*	selectNodes*	attributes*	nodeType String*
cloneNode*	selectSingleNode*	baseName*	nodeValue*
getAttribute	setAttribute	childNodes*	ownerDocument *
getAttributeNode	setAttributeNode	dataType*	parentNode*
getElementsBy TagName	transformNode*	definition*	parsed*
hasChildNodes*	transformNodeTo Object*	firstChild*	prefix*
insertBefore*		lastChild*	previous Sibling*
normalize		namespace URI*	specified*
removeAttribute		nextSibling*	tagName
removeAttribute Node		nodeName*	text*
removeChild*		nodeType*	xml*
replaceChild*		nodeTyped Value*	

* See section on `XMLDOMNode` methods and properties.

Additional Methods

getAttribute

The `getAttribute` method returns a string containing the value of the attribute with the specified name for this element. If the attribute is not present but has a default value in the document's DTD, that value is returned; otherwise, this method returns the empty string.

```
Variant = XMLDOMElement.getAttribute(name)
```

Parameter	Data Type	Description
name	String	Name of the string attribute whose value is sought.

For example, if we have a `<Book>` element with an author attribute:

```
<Book Author="Kevin Williams">
```

We can retrieve the value from this attribute using the following code:

```
' VBScript
Dim strAuthor
strAuthor = xmlBookElement.getAttribute("Author")
```

```
// JScript
var strAuthor = xmlBookElement.getAttribute("Author");
```

getAttributeNode

The `getAttributeNode` method returns an `XMLDOMAttribute` object representing the attribute with the specified name for this element. If the attribute does not exist for this element, this method returns `NULL`.

```
XMLDOMAttribute = XMLDOMElement.getAttributeNode(name)
```

Parameter	Data Type	Description
name	String	Name of the attribute node to retrieve.

We could modify the previous example to retrieve a reference to the attribute node rather than to the value of the attribute:

```
' VBScript
Dim xmlAuthorAttribute
Set xmlAuthorAttribute = xmlBookElement.getAttributeNode("Author")
```

```
// JScript
var xmlAuthorAttribute = xmlBookElement.getAttributeNode("Author");
```

getElementsByTagName

The `getElementsByTagName` method returns an `XMLDOMNodeList` object containing a list of all the elements in this element's descendant tree with the specified *tag_name*, in the order they appear in the original document. If no elements in this element's descendant tree have the specified *tag_name*, an empty `XMLDOMNodeList` is returned.

```
XMLDOMNodeList = XMLDOMElement.getElementsByTagName(tag_name)
```

Parameter	Data Type	Description
tag_name	String	The name of the elements which will be retrieved.

For example, to retrieve all the <Chapter> descendants of a <Book> element into a node list:

```
' VBScript
Dim xmlChapterNodeList
Set xmlChapterNodeList = xmlBookElement.getElementsByTagName("Chapter")
```

```
// JScript
var xmlChapterNodeList = xmlBookElement.getElementsByTagName("Chapter");
```

normalize

The `normalize` method normalizes the descendant nodes of this element into a form where no two text nodes are adjacent. If adjacent text node siblings exist in the descendant nodes, they are combined into one node and the extra node is removed from the node tree.

```
XMLDOMElement.normalize
```

removeAttribute

The `removeAttribute` method removes the attribute with the specified name from this element's attribute list. If the attribute has a default value in the document's DTD, the attribute is instead replaced with one containing the default value for it defined in the DTD.

```
XMLDOMElement.removeAttribute(name)
```

Parameter	Data Type	Description
name	String	Name of attribute to be removed or changed.

removeAttributeNode

The `removeAttributeNode` method removes the specified attribute from this element's attribute list. If the attribute has a default value in the document's DTD, the attribute is instead replaced with one containing the default value for it defined in the DTD. This method returns the removed `XMLDOMAttribute` object.

```
XMLDOMNode = XMLDOMElement.removeAttributeNode(attr_node)
```

Parameter	Data Type	Description
attr_node	`XMLDOMAttribute` object	The attribute node removed from the element.

setAttribute

The `setAttribute` method sets the value of the attribute with the provided name to the specified value. If the attribute does not already exist for this element, it is created.

```
XMLDOMElement.setAttribute(name, value)
```

Parameter	Data Type	Description
name	String	Name of attribute whose value is to be set.
value	Variant	Value of attribute.

For example, to set the value of an existing `Author` attribute to `"Kevin Williams"`, or to create a new attribute with that value, we would write:

```
' VBScript
xmlBookElement.setAttribute("Author", "Kevin Williams")
```

```
// JScript
xmlBookElement.setAttribute("Author", "Kevin Williams");
```

setAttributeNode

The `setAttributeNode` method adds the attribute provided to the element. If an attribute with the same name already exists for this element, it is replaced with the new attribute. This method differs from `setAttribute` in that it returns the `XMLDOMAttribute` object.

```
XMLDOMAttribute = XMLDOMElement.setAttributeNode(attr_node)
```

Parameter	Data Type	Description
attr_node	`XMLDOMAttribute` object	Attribute node to be added to element.

Additional Properties

tagName

The read-only `tagName` property returns a string containing the element's name.

```
String = XMLDOMElement.tagName
```

XMLDOMEntity

The `XMLDOMEntity` object represents a parsed or unparsed entity in the XML document. The Microsoft XML DOM expands all external entities (except for binary ones) before returning the node tree. For example, in the following fragment of a DTD:

```
<!ENTITY ProgRef "Programmer's Reference">
```

The xmlBook entity would be represented by a XMLDOMEntity object. Note that this object only represents the entity definition; it does not represent the references to this entity within the XML document; these can be accessed through the XMLDOMEntityReference object.

We can reference this entity using the entities property of the XMLDOMDocumentType object:

```
' VBScript
Dim xmlDoctype, xmlEntity
Set xmlDoctype = xmlDocument.doctype
' Get reference to first entity in DTD
Set xmlEntity = xmlDoctype.entities(0)
```

```
// JScript
var xmlDoctype = xmlDocument.doctype;
// Get reference to first entity in DTD
var xmlEntity = xmlDoctype.entities(0);
```

The XMLDOMEntity object is derived from the XMLDOMNode object. In addition to the notationName, publicId, and systemId properties, please see the properties, methods, and events defined for the XMLDOMNode object for additional functionality available through the XMLDOMEntity object.

Methods	Properties	
appendChild*	attributes*	nodeValue*
cloneNode*	baseName*	notationName
hasChildNodes*	childNodes*	ownerDocument*
insertBefore*	dataType*	parentNode*
removeChild*	definition*	parsed*
replaceChild*	firstChild*	prefix*
selectNodes*	lastChild*	previousSibling*
selectSingleNode*	namespaceURI*	publicId
transformNode*	nextSibling*	specified*
transformNodeToObject*	nodeName*	systemId
	nodeType*	text*
	nodeTypedValue*	xml*
	nodeTypeString*	

* See section on XMLDOMNode methods and properties.

Additional Methods

There are no additional methods associated with the XMLDOMEntity object.

Additional Properties

notationName

The read-only `notationName` property returns a string containing the notation name for this entity. This is the name following the `NDATA` declaration, if there is one. For parsed entities, this property returns an empty string.

```
String = XMLDOMEntity.notationName
```

publicId

The read-only `publicId` property returns a string containing the public identifier for this entity. This is the string following the `PUBLIC` declaration, if one exists. If the public identifier is not specified, this property returns an empty string.

```
String = XMLDOMEntity.publicId
```

systemId

The read-only `systemId` property returns a string containing the system identifier for this entity. This is the string following the `SYSTEM` declaration, if one exists. If the system identifier is not specified, this property returns an empty string.

```
String = XMLDOMEntity.systemId
```

XMLDOMEntityReference

The `XMLDOMEntityReference` object represents an entity reference within an XML document.

For example, suppose we have an element such as the following containing an entity reference:

```
<Title>ASP &ProgRef;</Title>
```

The `<Title>` element has two child nodes: the first is a text node containing the string `"ASP "`, and the second is an entity reference with the name `"ProgRef"`. Since this is the second child node, we can get a reference to it using `xmlTitleElement.childNodes(1)`.

If the entity information about the entity being referenced is available, the `XMLDOMEntityReference` object's child tree will reflect the internal structure of that entity. For example, if the `ProgRef` entity is defined as:

```
<!ENTITY ProgRef "Programmer's Reference">
```

The entity reference will have a single child node: a text node containing the expanded text, `"Programmer's Reference"`.

The `XMLDOMEntityReference` object is derived from the `XMLDOMNode` object. It exposes no new methods, properties or events, so see the `XMLDOMNode` object for the functionality available through the `XMLDOMEntityReference` object.

Methods	Properties	
appendChild*	attributes*	nodeTypedValue*
cloneNode*	baseName*	nodeTypeString*
hasChildNodes*	childNodes*	nodeValue*
insertBefore*	dataType*	ownerDocument*
removeChild*	definition*	parentNode*
replaceChild*	firstChild*	parsed*
selectNodes*	lastChild*	prefix*
selectSingleNode*	namespaceURI*	previousSibling*
transformNode*	nextSibling*	specified*
transformNodeToObject*	nodeName*	text*
	nodeType*	xml*

* See section on XMLDOMNode methods and properties.

XMLDOMImplementation

The XMLDOMImplementation object allows the developer to query msxml.dll to determine what features it supports. This may be used to discover whether newer features are available from the version of the parser being used.

The XMLDOMImplementation object for a particular document may be accessed via the implementation property of the XMLDOMDocument object for that document.

Methods
hasFeature

Methods

hasFeature

The hasFeature method returns a boolean indicating whether the specified feature is supported by the current version of msxml.dll. For msxml.dll, valid features are "XML", "DOM", and "MS-DOM". At the time of writing, only version 1.0 of the DOM is supported by msxml.dll.

```
Boolean = XMLDOMImplementation.hasFeature(feature, version)
```

Parameter	Data Type	Description
feature	String	Specified feature
version	String	Version of that feature to check

The following example use the hasFeature method to check whether the current parser supports MSXML version 2.6 methods:

```
' VBScript
Dim xmlCatalogImplementation
Set xmlCatalogImplementation = xmlCatalogDocument.implementation
If xmlCatalogImplementation.hasFeature("MS-DOM", "2.6") Then
   Response.Write "2.6 objects, methods, and properties are supported."
Else
   Response.Write "2.6 objects, methods, and properties are not supported."
End If
```

```
// JScript
var xmlCatalogImplementation = xmlCatalogDocument.implementation;
if (xmlCatalogImplementation.hasFeature("MS-DOM", "2.6")) {
   Response.Write("2.6 objects, methods, and properties are supported.");
} else {
   Response.Write("2.6 objects, methods, and properties are not supported.");
}
```

Properties

There are no properties associated with the XMLDOMImplementation object.

XMLDOMNamedNodeMap

The XMLDOMNamedNodeMap collection contains an unordered set of nodes. They may be referenced by their name as well as by an ordinal position, although the ordinal position does not imply anything else about the node (it's simply a convenience for iterating through the set). An XMLDOMNamedNodeMap is used to represent the collection of attributes for an element node, and may be accessed through the attributes property of an element node.

Methods		Properties
getNamedItem	removeNamedItem	length
getQualifiedItem	removeQualifiedItem	
item	reset	
nextNode	setNamedItem	

Methods

getNamedItem

The getNamedItem method returns the XMLDOMNode object for the node in the collection with the specified name. If the node with the specified name is not in the collection of nodes, this method returns NULL.

```
XMLDOMNode = XMLDOMNamedNodeMap.getNamedItem(name)
```

Parameter	Data Type	Description
name	String	Name of node object.

For example:

```
' VBScript
' get the attribute list for the Book element
Dim xmlBookAttributes, xmlAuthorAttribute
xmlBookAttributes = xmlBookElement.attributes
' find the Author attribute in the attribute list
xmlAuthorAttribute = xmlBookAttributes.getNamedItem("Author")
```

```
// JScript
// get the attribute list for the Book element
var xmlBookAttributes, xmlAuthorAttribute;
xmlBookAttributes = xmlBookElement.attributes;
// find the Author attribute in the attribute list
xmlAuthorAttribute = xmlBookAttributes.getNamedItem("Author");
```

getQualifiedItem

The `getQualifiedItem` method returns the `XMLDOMNode` object for the attribute in the collection with the specified attribute name and namespace prefix. Note that, even though the second parameter is called *namespace_uri*, it actually corresponds to the namespace prefix. If an attribute matching the specified base name and namespace prefix is not in the collection of nodes, this method returns `NULL`.

```
XMLDOMNode = XMLDOMNamedNodeMap.getQualifiedItem(base_name, namespace_uri)
```

Parameter	Data Type	Description
base_name	String	The name of the item excluding the namespace prefix.
namespace_uri	String	The namespace prefix of the item.

Item

The `item` method returns the `XMLDOMNode` at the position indicated by the `index` parameter. The `XMLDOMNamedNodeMap` collection is zero-based, so the `index` parameter must be between `0` and `length - 1`. If the index is out of range, this method returns `NULL`.

```
XMLDOMNode = XMLDOMNamedNodeMap.item(index)
```

Parameter	Data Type	Description
index	Long	Position of `XMLDOMNode` object.

For example, to get the first attribute for a <Book> element:

```
' VBScript
' get the attribute list for the Book element
Dim xmlBookAttributes, xmlFirstAttribute
Set xmlBookAttributes = xmlBookElement.attributes
' get the first attribute for the Book element
Set xmlFirstAttribute = xmlBookAttributes.item(0)
```

```
// JScript
// get the attribute list for the Book element
var xmlBookAttributes, xmlFirstAttribute;
xmlBookAttributes = xmlBookElement.attributes;
// get the first attribute for the Book element
xmlFirstAttribute = xmlBookAttributes.item(0);
```

nextNode

The `nextNode` method returns the next node in the list from the current iterator position. When the `XMLDOMNamedNodeMap` object is created, the iterator starts before the first node in the list, so all of the nodes may be iterated through by repeatedly calling `nextNode` until it returns a `NULL`, indicating that there are no more nodes in the list beyond the iterator position. Note again that the iterator is a convenience only and does not imply any sort of order to the nodes in the named node map.

```
XMLDOMNode = XMLDOMNamedNodeMap.nextNode()
```

removeNamedItem

The `removeNamedItem` method removes the node with the specified name from the collection. It returns an `XMLDOMNode` object representing the removed node. If the node with the given name does not exist in the collection, this method returns `NULL`.

```
XMLDOMNode = XMLDOMNamedNodeMap.removeNamedItem(name)
```

Parameter	Data Type	Description
name	String	Name of item to be removed.

removeQualifiedItem

The `removeQualifiedItem` method removes the `XMLDOMNode` object with the specified attribute name and namespace prefix from the collection. Note that, even though the second parameter is called *namespace_uri*, it actually corresponds to the namespace prefix. The removed `XMLDOMNode` object is returned. If an attribute matching the specified base name and namespace prefix is not in the collection of nodes, this method returns `NULL`.

```
XMLDOMNode = XMLDOMNamedNodeMap.removeQualifiedItem(base_name, namespace_uri)
```

Parameter	Data Type	Description
base_name	String	Name of `XMLDOMNode` object.
namespace_uri	String	Namespace of `XMLDOMNode` object

reset

The `reset` method resets the position of the iterator on the collection to before the first node in the list.

```
XMLDOMNamedNodeMap.reset()
```

setNamedItem

The `setNamedItem` method adds the specified `XMLDOMNode` object to the collection. If the collection already contains an attribute node with the same name as the new node, it is replaced. If the node being added is not an attribute, `setNamedItem` returns an error.

```
XMLDOMNode = XMLDOMNamedNodeMap.setNamedItem(new_item)
```

Parameter	Data Type	Description
new_item	XMLDOMNode	Name of XMLDOMNode to be set

For example, to add an `Author` attribute with the value `"Kevin Williams"` to a `<Book>` element:

```
' VBScript
' get the attribute list for the Book element
Dim xmlBookAttributes, xmlAuthorAttribute
Set xmlAuthorAttribute = xmlCatalogDocument.createAttribute("Author")
xmlAuthorAttribute.value = "Kevin Williams"
Set xmlBookAttributes = xmlBookElement.attributes
' set the Author attribute
Set xmlAuthorAttribute = xmlBookAttributes.setNamedItem(xmlAuthorAttribute)
```

```
// JScript
// get the attribute list for the Book element
var xmlBookAttributes, xmlAuthorAttribute;
xmlAuthorAttribute = xmlCatalogDocument.createAttribute("Author");
xmlAuthorAttribute.value = "Kevin Williams";
xmlBookAttributes = xmlBookElement.attributes;
// set the Author attribute
xmlAuthorAttribute = xmlBookAttributes.setNamedItem(xmlAuthorAttribute);
```

Properties

length

The read-only `length` property returns the number of `XMLDOMNode` objects in the collection.

```
Long = XMLDOMNamedNodeMap.length
```

XMLDOMNode

The `XMLDOMNode` object is the base object for all of the different node types expressed in the DOM node tree. It includes many generic methods that may be used to manipulate the various nodes without regard to the type of node represented by the object. Also, the `XMLDOMNode` object must be used to create objects with namespaces, as the specific node implementation objects do not provide this functionality.

Methods	Properties	
appendChild	attributes	nodeTypedValue
cloneNode	baseName	nodeTypeString
hasChildNodes	childNodes	nodeValue
insertBefore	dataType	ownerDocument
removeChild	definition	parentNode
replaceChild	firstChild	parsed
selectNodes	lastChild	prefix
selectSingleNode	namespaceURI	previousSibling
transformNode	nextSibling	specified
transformNodeToObject	nodeName	text
	nodeType	xml

Methods

appendChild

The appendChild method appends the newChild node to the end of the child list for this node. It returns a reference to the newChild node.

```
XMLDOMNode = XMLDOMNode.appendChild(new_child)
```

Parameter	Data Type	Description
new_child	XMLDOMNode object	New child node to be appended to child list.

The appendChild method must be called to add an element to the node tree after it has been created with the XMLDOMDocument's createElement method:

```
' VBScript
' create a new Chapter element
Dim xmlChapterElement
xmlChapterElement = xmlCatalogDocument.createElement("Chapter")
' append it to the Book element
xmlBookElement.appendChild(xmlChapterElement)
```

```
// VBScript
// create a new Chapter element
var xmlChapterElement;
xmlChapterElement = xmlCatalogDocument.createElement("Chapter");
// append it to the Book element
xmlBookElement.appendChild(xmlChapterElement);
```

cloneNode

The `cloneNode` method makes an exact copy of the current node and returns it. If the `deep` flag is set to `true`, it also recursively copies all of the child nodes in this node's subtree; otherwise, the new node will only have the attributes of the original, and its child node list will be blank.

```
XMLDOMNode = XMLDOMNode.cloneNode(deep)
```

Parameter	Data Type	Description
deep	Boolean	Flag which indicates whether child nodes are to be copied

For example, to clone a node, including all its descendant nodes:

```
' VBScript
' clone the Book node and all of its subtree
Dim xmlBookElement2
Set xmlBookElement2 = xmlBookElement.cloneNode(True)
```

```
// JScript
// clone the Book node and all of its subtree
var xmlBookElement2;
xmlBookElement2 = xmlBookElement.cloneNode(1);
```

Note that this creates a new, independent node, not a new reference to the existing node (so changing the text in one of these elements will not change the text in the other).

hasChildNodes

The `hasChildNodes` method returns a Boolean value indicating whether this node has children. It will always return `false` for node types that cannot have children.

```
Boolean = XMLDOMNode.hasChildNodes()
```

We should call this method before iterating through an element's children (for example, when parsing a document recursively):

```
' VBScript
' Check to see if the Book element has any children
If xmlBookElement.hasChildNodes Then
   ' there are children - process them
Else
   ' there are no children - continue
End If
```

```
// JScript
// Check to see if the Book element has any children
if (xmlBookElement.hasChildNodes()) {
   // there are children - process them
} else {
   // there are no children - continue
}
```

insertBefore

The `insertBefore` method inserts the child specified by the *`new_child`* parameter into the child list of this node, to the left of the node specified by the *`ref_child`* parameter. If the *`ref_child`* parameter is NULL, the *`new_child`* node is added at the end of the parameter list.

```
XMLDOMNode = XMLDOMNode.insertBefore(new_child, ref_child)
```

Parameter	Data Type	Description
`new_child`	XMLDOMNode object	Child node to be inserted.
`ref_child`	Variant	The node before which the new child is to be inserted.

It returns the *`new_child`* node. Note that if a document fragment node is added using this method, instead of the document fragment node being inserted, all of the root-level elements of the fragment will be inserted at the same location. If the insertion being attempted contravenes any of the rules of the DOM (for example, that attributes may not be the children of other attributes), this method will return an error code.

```
' VBScript
' create a new Chapter element
Dim xmlChapterElement, xmlFirstChildNode
Set xmlChapterElement = xmlCatalogDocument.createElement("Chapter")
Set xmlFirstChildNode = xmlBookElement.firstChild
' insert it before the first child node of the Book element
xmlBookElement.insertBefore(xmlChapterElement, xmlFirstChildNode)
```

```
// JScript
// create a new Chapter element
var xmlChapterElement = xmlCatalogDocument.createElement("Chapter");
var xmlFirstChildNode = xmlBookElement.firstChild;
// insert it before the first child node of the Book element
xmlBookElement.insertBefore(xmlChapterElement, xmlFirstChildNode);
```

removeChild

The `removeChild` method removes the specified child node from the child list of the current node and returns a reference to that child node. Note that even though the child node has been "disconnected" from the rest of the document, it will still exist, along with any nodes that make up its subtree; reattaching the node somewhere else in the tree will also reattach all of the subnodes. If the specified child node cannot be removed from the tree, this method will return an error code.

```
XMLDOMNode = XMLDOMNode.removeChild(old_child)
```

Parameter	Data Type	Description
`old_child`	XMLDOMNode object	Child to be removed from child list

For example, to remove the first child of an element:

```
' get the first subnode of the Book element
Dim xmlFirstElement
xmlFirstElement = xmlBookElement.firstChild
' remove it
xmlBookElement.removeChild(xmlFirstElement)
```

```
// JScript
// get the first element in the Book element
var xmlFirstElement;
xmlFirstElement = xmlBookElement.firstChild;
// remove it
xmlBookElement.removeChild(xmlFirstElement);
```

replaceChild

The `replaceChild` method allow us to replace the child node of the current node specified by the *`old_child`* parameter with the node specified by the *`new_child`* parameter. If a document fragment is added using this method, instead of the document fragment node being used to replace the old child node, the document fragment node's children will be inserted at the same location. If the replacement cannot be made, or the replacement would contravene any of the rules of the DOM, this method will return an error.

```
XMLDOMNode = XMLDOMNode.replaceChild(new_child, old_child)
```

Parameter	Data Type	Description
new_child	XMLDOMNode object	The new node which will replace the old
old_child	XMLDOMNode object	Old child node to be replaced

For example:

```
' VBScript
' create a new Chapter element
Dim xmlChapterElement
xmlChapterElement = xmlCatalogDocument.createElement("Chapter")
' replace the first child node of the Book element with the new chapter
' element
xmlBookElement.replaceChild(xmlBookElement.childNodes(0), xmlChapterElement)
```

```
// JScript
// create a new Chapter element
var xmlChapterElement;
xmlChapterElement = xmlCatalogDocument.createElement("Chapter");
// replace the first child node of the Book element with the new chapter
// element
xmlBookElement.replaceChild(xmlBookElement.childNodes(0), xmlChapterElement);
```

selectNodes

The `selectNodes` method applies the XSL pattern query specified by the *`pattern_string`* parameter using this node as the context node, and returns an `XMLDOMNodeList` containing the list of nodes selected by the query. If no nodes match the pattern query, an empty `XMLDOMNodeList` is returned.

```
XMLDOMNode = XMLDOMNode.selectNodes(pattern_string)
```

Parameter	Data Type	Description
pattern_string	String	Query to be executed on the subtree

For example, to select all <Book> elements with an Author attribute set to "Kevin Williams":

```
' VBScript
Dim xmlBookElements
Set xmlBookElements = xmlCatalogDocument.selectNodes
                      ("//Book[@Author='Kevin Williams']")
```

```
// JScript
var xmlBookElements = xmlCatalogDocument.selectNodes
                      ("//Book[@Author='Kevin Williams']");
```

selectSingleNode

The selectSingleNode method applies the XSL pattern query specified by the patternString parameter using this node as the context node, and returns an XMLDOMNode that corresponds to the first node that matches the query. If no nodes match the query, this method returns NULL.

```
XMLDOMNode = XMLDOMNode.selectSingleNode(pattern_string)
```

Parameter	Data Type	Description
pattern_string	String	Query to be executed on the subtree

For example, to find the first <Chapter> element at any level in the document:

```
' VBScript
Dim xmlChapterElement
xmlChapterElement = xmlCatalogDocument.selectSingleNode("//Chapter")
```

```
// JScript
var = xmlCatalogDocument.selectSingleNode("//Chapter");
```

transformNode

The transformNode method applies the XSL stylesheet or fragment specified in the *stylesheet* parameter to the current node and returns a string containing the output of that transformation. The stylesheet may be specified either as an XMLDOMDocument object that references an entire stylesheet, or an XMLDOMNode object that references a fragment of a stylesheet.

```
XMLDOMNode = XMLDOMNode.transformNode(stylesheet)
```

Parameter	Data Type	Description
stylesheet	XMLDOMNode or XMLDOMDocument object	The stylesheet to be applied to the subtree

The following code transforms an XML document, storing the result as a string, and then sends this XML string to the browser. Transforming an XML document on the server in this way avoids problems caused by browsers which don't support XML and/or XSL.

```
' VBScript
Dim xmlDocument, xslDocument
Set xmlDocument=Server.CreateObject("Microsoft.XMLDOM")
Set xslDocument=Server.CreateObject("Microsoft.XMLDOM")

xmlDocument.async=false
xslDocument.async=false

xslDocument.load "http://myserver/books.xml"
xslDocument.load "http://myserver/books.xsl"

strXML = xmlDocument.transformNode(xslDocument.documentElement)
Response.Write strXML
```

```
// JScript
var xmlDocument=Server.CreateObject("Microsoft.XMLDOM");
var xslDocument=Server.CreateObject("Microsoft.XMLDOM");

xmlDocument.async=false;
xslDocument.async=false;

xslDocument.load("http://myserver/books.xml");
xslDocument.load("http://myserver/books.xsl");

strXML = xmlDocument.transformNode(xslDocument.documentElement);
Response.Write(strXML);
```

transformNodeToObject

The `transformNodeToObject` method applies the XSL stylesheet or fragment specified in the *`stylesheet`* parameter to the current node. If the *`output_object`* parameter is an `XMLDOMDocument` object, the document is constructed with the results of the transformation; if the *`output_object`* parameter is a stream, the results of the transformation are written to that stream.

```
Variant = XMLDOMNode.transformNodeToObject(stylesheet, output_object)
```

Parameter	Data Type	Description
`stylesheet`	XMLDOMNode stylesheet	Stylesheet to be applied to subtree.
`output_object`	Variant	If the *`output_object`* parameter is an `XMLDOMDocument` object, the document is constructed with the results of the transformation. If the *`output_object`* parameter is a stream, the results of the transformation are written to that stream.

This code does the same as the previous example, but this time the transformed XML is stored in an XMLDOMDocument object rather than a string, so if we want, we could perform additional operations on the XML before sending it to the client:

```
' VBScript
Dim xmlDocument, xslDocument, xmlNewDocument
Set xmlDocument = Server.CreateObject("Microsoft.XMLDOM")
Set xslDocument = Server.CreateObject("Microsoft.XMLDOM")

xmlDocument.async=false
xslDocument.async=false

xslDocument.load "http://myserver/books.xml"
xslDocument.load "http://myserver/books.xsl"

Set xmlNewDocument = Server.CreateObject("Microsoft.XMLDOM")
xmlDocument.transformNodeToObject xslDocument.documentElement, xmlNewDocument
Response.Write xmlNewDocument.xml
```

```
// JScript
var xmlDocument, xslDocument, xmlNewDocument;
xmlDocument = Server.CreateObject("Microsoft.XMLDOM");
xslDocument = Server.CreateObject("Microsoft.XMLDOM");

xmlDocument.async = false;
xslDocument.async = false;

xslDocument.load("http://myserver/books.xml");
xslDocument.load("http://myserver/books.xsl");

xmlNewDocument = Server.CreateObject("Microsoft.XMLDOM");
xmlDocument.transformNodeToObject(xslDocument.documentElement,
                                  xmlNewDocument);
Response.Write(xmlNewDocument.xml);
```

Properties

attributes

The read-only `attributes` property returns an `XMLNamedNodeMap` containing an unordered list of the attributes for this node. Note that this will only be meaningful for nodes whose `nodeType` is `NODE_ELEMENT`, `NODE_ENTITY`, or `NODE_NOTATION`; all other `nodeTypes` will return `NULL`. For entity and notation nodes, the attributes must be either `PUBLIC`, `SYSTEM`, or `NDATA`.

```
XMLDOMNamedNodeMap = XMLDOMNode.attributes
```

baseName

The read-only `baseName` property returns the string containing the base name of the node. For non-namespace-qualified nodes, this will always be the name of the node; for namespace-qualified nodes, this will be the portion of the name following the colon – i.e., if the node's name is `abc:def`, `baseName` will return `def`.

```
String = XMLDOMNode.baseName
```

childNodes

The read-only `childNodes` property returns an `XMLDOMNodeList` containing an ordered list of the children of this node. If the node does not have children, or is not permitted to have children because of its type (for example, a `NODE_COMMENT` node), an `XMLDOMNodeList` is still returned with a length of zero.

```
XMLDOMNodeList = XMLDOMNode.childNodes
```

We can use this property to parse an XML document recursively. The following example defines a function `fnRecurse()` which takes a node object and an integer as parameters. The integer defines the number of spaces which will be printed out before the node's name and value, according to its depth in the node tree. After storing the requisite number of non-breaking spaces in a string, the function adds the name and value of the node. We then iterate through its `childNodes` and call the *same* function on each of these, before returning the resulting string. The rest of the script simply loads an XML document, calls our `fnRecurse` function on it, and finally writes this string into the contents of a `<SPAN>` element.

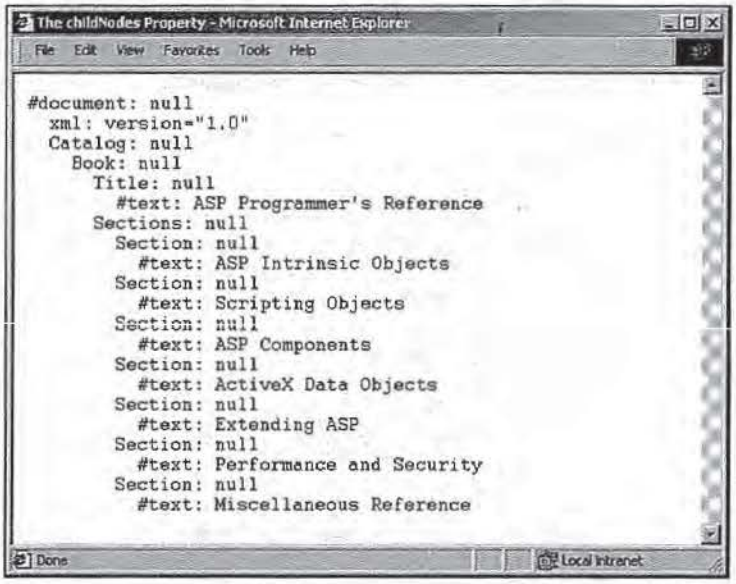
```
<HTML>
<%
Dim xmlDocument
Set xmlDocument=Server.CreateObject("Microsoft.XMLDOM")
xmlDocument.async=false
xmlDocument.load "http://myserver/xml/books.xml"
strXML = fnRecurse(xmlDocument, 0)

Function fnRecurse(objNode, intSpaces)
   Dim strXML
   strXML = ""
   For intCount=1 To intSpaces
      strXML = strXML & " "
   Next
   strXML = strXML & objNode.nodeName & ": " & _
            objNode.nodeValue & "<BR>"
   Dim intChildren
   intChildren = objNode.childNodes.length
   Dim intChildCount
   For intChildCount = 0 To intChildren - 1
      Dim objChild
      Set objChild=objNode.childNodes(intChildCount)
      strXML = strXML & fnRecurse(objChild, intSpaces+2)
   Next
   fnRecurse = strXML
End Function
%>
<SPAN STYLE="font-family: courier"><%= strXML %></SPAN>
</HTML>
```

Or in JScript:

```
<HTML>
<%
var xmlDocument=Server.CreateObject("Microsoft.XMLDOM");
xmlDocument.async=false;
xmlDocument.load("http://myserver/xml/books.xml");
strXML = fnRecurse(xmlDocument, 0);
```

```
function fnRecurse(objNode, intSpaces) {
   var strXML="";
   for (var i=0; i<intSpaces; i++) {
      strXML+=" ";
   }
   strXML+=objNode.nodeName+": "+objNode.nodeValue+"<BR>";
   var intChildren=objNode.childNodes.length;
   for (var j=0; j<intChildren; j++) {
      var objChild=objNode.childNodes[j];
      strXML+=fnRecurse(objChild, intSpaces+2);
   }
   return strXML;
}
%>
<SPAN STYLE="font-family: courier"><%= strXML %></SPAN>
</HTML>
```

The result is a display of the document's node tree, with the name and value of each node:



dataType

The `dataType` property sets or returns the string description of the data type for this node. For attribute and element nodes, this will be the string representation of the data type specifier included in the schema, or `NULL` if no data type was defined for the element or attribute. For entity references, this value will only be non-`NULL` if the entity referenced has exactly one top-level element; otherwise, it will return `NULL`. Attempts to assign the data type for an entity reference will be ignored. This property will always return `NULL` if the XML document is not associated with a schema.

```
XMLDOMNode.dataType = String
Variant = XMLDOMNode.dataType
```

definition

The read-only `definition` property returns an `XMLDOMNode` that is the definition of this node in the DTD or schema.

```
XMLDOMNode = XMLDOMNode.definition
```

If the node is an entity reference, the value returned will be the node in the DOCTYPE node's entity collection that corresponds to the referenced entity. If the node is an unparsed entity, the value returned will be the node in the DOCTYPE node's notation collection that corresponds to the entity. If the node is an attribute, and an XML-Data schema is being used, the node corresponding to the AttributeType declaration in the schema will be returned. If the node is an element and an XML-Data schema is being used, the node corresponding to the ElementType declaration in the schema will be returned. All other node types return NULL for this property.

firstChild

The read-only firstChild property returns the XMLDOMNode that is the first child node of this node. If this node has no children, or the node type is not allowed to have children, this property returns NULL.

```
XMLDOMNode = XMLDOMNode.firstChild
```

lastChild

The read-only lastChild property returns the XMLDOMNode that is the last child node of this node. If this node has no children, or the node type is not allowed to have children, this property returns NULL.

```
XMLDOMNode = XMLDOMNode.lastChild
```

namespaceURI

The read-only namespaceURI property returns the string corresponding to the URI of the namespace containing this node. If there is no namespace declared for the node, this property returns an empty string.

```
String = XMLDOMNode.namespaceURI
```

nextSibling

The read-only nextSibling property returns an XMLDOMNode that is the next sibling node of this node (that is, the next child node in the ordered child list of this node's parent). If the node does not have siblings following it, or the node is not allowed to have siblings (document nodes, document fragment nodes, and attribute nodes), this property returns NULL.

```
XMLDOMNode = XMLDOMNode.nextSibling
```

nodeName

The read-only nodeName property returns a string containing the fully-qualified name of the node. This string will include the namespace prefix for the node name if it is present. For nodes that do not have names, a constant string will always be returned; for example, a text node will always have a nodeName of #text.

```
String = XMLDOMNode.nodeName
```

The standard values for nodeName are:

Node Type	Node Name
Text node	"#text"
CDATA section	"#cdata-section"
Comment	"#comment"
Document	"#document"
Document fragment	"#document-fragment"

nodeType

The read-only nodeType property returns a value in the DOMNodeType enumeration indicating the type of node. See the description of the DOMNodeType enumeration later in the text for more details on the possible values for the nodeType property.

```
DOMNodeType = XMLDOMNode.nodeType
```

The possible values for nodeType are:

Name	Value	Definition
NODE_INVALID	0	The node is not a valid XML node.
NODE_ELEMENT	1	This node represents an element.
NODE_ATTRIBUTE	2	This node represents an attribute of an element.
NODE_TEXT	3	This node represents the text content of a tag.
NODE_CDATA_SECTION	4	This node represents a CDATA section (an escaped, unparsed block of text).
NODE_ENTITY_REFERENCE	5	This node represents a reference to an entity.
NODE_ENTITY	6	This node represents an expanded entity.
NODE_PROCESSING_INSTRUCTION	7	This node represents a processing instruction.
NODE_COMMENT	8	This node represents a comment.
NODE_DOCUMENT	9	This node represents a document. Note that it must have exactly one element child node, representing the root element in the document.
NODE_DOCUMENT_TYPE	10	This node represents the document type declaration.
NODE_DOCUMENT_FRAGMENT	11	This node represents a document fragment.
NODE_NOTATION	12	This node represents a notation.

If you want to use the constants rather than the numerical values, you will need to add a reference to the type library to your page, using a METADATA directive:

```
<!-- METADATA TYPE="TypeLib"
              FILE="C:\WinNT\System32\msxml2.dll" -->
```

If using the earlier version of MSXML, you will need to make the reference to `msxml.dll` *rather than* `msxml2.dll`.

nodeTypedValue

The `nodeTypedValue` property sets or returns the typed value for the node. For instance, if an attribute is declared to be of type float in a document's schema, this value will be a variant containing a floating-point integer value. Note that this works for text values in elements as well; if the containing element is typed, reading this property on the text node will return a value in the type of the containing element.

```
XMLDOMNode.nodeTypedValue = Variant
Variant = XMLDOMNode.nodeTypedValue
```

nodeTypeString

The read-only `nodeTypeString` property returns a string value for the node type.

```
String = XMLDOMNode.nodeTypeString
```

Possible values are: `"attribute"`, `"cdatasection"`, `"comment"`, `"document"`, `"documentfragment"`, `"documenttype"`, `"element"`, `"entity"`, `"entityreference"`, `"notation"`, `"processinginstruction"`, and `"text"`.

nodeValue

The `nodeValue` property sets or returns the string value for a node. If you want to work with the typed value of the node instead, use `nodeTypedValue`.

```
XMLDOMNode.nodeValue = Variant
Variant = XMLDOMNode.nodeValue
```

Note that the `nodeValue` of an element node will always return `NULL`; the value of any text content of the element is contained in a child text node.

ownerDocument

The read-only `ownerDocument` property returns the `XMLDOMDocument` object that corresponds to the document that contains this node. If the node has been removed from its document, this property will return the document that last contained this node.

```
XMLDOMDocument = XMLDOMNode.ownerDocument
```

parentNode

The read-only `parentNode` property returns the `XMLDOMNode` object for the parent node of this node. If the node does not have a parent (attribute nodes, document nodes, and document fragment nodes), this property returns `NULL`.

```
XMLDOMNode = XMLDOMNode.parentNode
```

parsed

The read-only `parsed` property returns a Boolean indicating whether the node has been completely parsed yet or not. This property is useful for asynchronous operation, to determine whether the parser has finished parsing a node.

```
Boolean = XMLDOMNode.parsed
```

prefix

The read-only `prefix` property returns a string containing the namespace prefix for the node; that is, if the node's name is `abc:def`, this property returns `abc`. If no namespace is declared for the node, or if the node's name may not contain a namespace, this property returns an empty string.

```
String = XMLDOMNode.prefix
```

previousSibling

The read-only `previousSibling` property returns an `XMLDOMNode` that is the previous sibling node of this node (that is, the previous child node in the ordered child list of this node's parent). If the node does not have siblings preceding it, or the node is not allowed to have siblings (document nodes, document fragment nodes, and attribute nodes), this property returns `NULL`.

```
XMLDOMNode = XMLDOMNode.previousSibling
```

specified

The read-only `specified` property returns a Boolean indicating whether the value for this node was specified, or was assigned by a default definition in a DTD or schema. This value will only be `false` for attribute nodes that take their values from defaults in the DTD or schema; for all other node types, the value returned will always be `true`.

```
Boolean = XMLDOMNode.specified
```

text

The `text` property sets or returns the text for a node.

```
XMLDOMNode.text = String
String = XMLDOMNode.text
```

This actually corresponds to the concatenated text of all the children of this node as well as this node, in document order, with whitespace normalized according to the current whitespace settings. For example, for a document like this:

```
<Book>ASP Programmer's Reference
   <Section>Intrinsic Objects</Section>
   <Section>Scripting Objects</Section>
   <Section>ASP Components</Section>
</Book>
```

The `text` property would return the string `"ASP Programmer's Reference Intrinsic Objects Scripting Objects ASP Components"`.

xml

The read-only `xml` property returns a string that contains the XML representation of this node and all its descendants. This property should be used when a document has been manipulated in script using the DOM and the modified version now needs to be sent to the client or persisted to a file.

```
String = XMLDOMNode.xml
```

XMLDOMNodeList

The `XMLDOMNodeList` object is a collection containing an ordered list of `XMLDOMNode` objects. It is returned when requesting the children of a node, or when executing an XSL pattern query against an XML document. For example:

```
' VBScript
Dim xmlNodeList
Set xmlNodeList = xmlBookElement.childNodes
```

```
// JScript
var xmlNodeList = xmlBookElement.childNodes;
```

Normal ASP collection coding techniques (such as a VBScript `For...Next` loop or a JScript `Enumerator`) may be used to access the individual members of this collection:

```
' VBScript
For Each xmlNode In xmlBookElement.childNodes
   Response.Write xmlNode.nodeName & ": " & xmlNode.text & "<BR>"
Next
```

```
// JScript
var enmNodeList = new Enumerator(xmlBookElement.childNodes);
for (; !enmNodeList.atEnd(); enmNodeList.moveNext()) {
   var xmlNode = enmNodeList.item();
   Response.Write(xmlNode.nodeName + ": " + xmlNode.text + "<BR>");
}
```

Methods	Properties
`item`	`length`
`nextNode`	
`reset`	

Methods

item

The `item` method returns the `XMLDOMNode` at the position indicated by the *`index`* parameter. The `XMLDOMNodeList` collection is zero-based, so the *`index`* parameter must be between `0` and `length - 1`. If the index is out of range, this method returns `NULL`.

```
XMLDOMNode = XMLDOMNodeList.item(index)
```

Parameter	Data Type	Description
index	Long	The position of the node in the nodelist.

nextNode

The `nextNode` method returns the next node in the list from the current iterator position. When the `XMLDOMNodeList` object is created, the iterator starts before the first node in the list, so all of the nodes may be iterated through by repeatedly calling `nextNode` until it returns a `NULL`, indicating that there are no more nodes in the list beyond the iterator position. The iterator position can be reset to the start using the `reset` method.

```
XMLDOMNode = XMLDOMNodeList.nextNode()
```

reset

The `reset` method resets the position of the iterator on the collection to before the first node in the list.

```
XMLDOMNodeList.reset()
```

Properties

length

The read-only `length` property returns the number of `XMLDOMNode` objects in the collection.

```
Long = XMLDOMNodeList.length
```

XMLDOMNotation

The `XMLDOMNotation` object represents a notation declaration in the document's DTD or schema. Notations are used to associate certain special attributes with external applications which are used to process data which cannot be handled by the XML parser (such as images). For example, for the following notation declaration:

```
<!NOTATION jpeg PUBLIC "www.jpeg.org">
```

There would be one `XMLDOMNotation` object in the `XMLDOMDocumentType` associated with the document.

The `XMLDOMNotation` objects for a document may be accessed by accessing the `notations` property of the `XMLDOMDocumentType` object associated with the document.

The `XMLDOMNotation` object is derived from the `XMLDOMNode` object. In addition to the `publicId` and `systemId` properties of this object, see the methods and properties, methods defined for the `XMLDOMNode` object for additional functionality available through the `XMLDOMNotation` object.

Methods	Properties	
appendChild*	attributes*	nodeTypeString*
cloneNode*	baseName*	nodeValue*
hasChildNodes*	childNodes*	ownerDocument*
insertBefore*	dataType*	parentNode*
removeChild*	definition*	parsed*
replaceChild*	firstChild*	prefix*
selectNodes*	lastChild*	previousSibling*
selectSingleNode*	namespaceURI*	publicId
transformNode*	nextSibling*	specified*
transformNodeToObject*	nodeName*	systemId
	nodeType*	text*
	nodeTypedValue*	xml*

* See section on XMLDOMNode methods and properties.

Additional Methods

There are no additional methods associated with the XMLDOMNotation object.

Additional Properties

publicId

The read-only publicId property returns a string containing the public identifier for this entity (e.g., for the example above, this would return "www.jpeg.org"). If the public identifier is not specified, this property returns an empty string.

```
Variant = XMLDOMNotation.publicId
```

systemId

The read-only systemId property returns a string containing the system identifier for this entity. If the system identifier is not specified, this property returns an empty string.

```
Variant = XMLDOMNotation.systemId
```

XMLDOMParseError

The XMLDOMParseError object contains information about parsing errors encountered by the XMLDOMDocument object while attempting to parse an XML document. We can get a reference to this object using the parseError property of the XMLDOMDocument object:

```
' VBScript
Set xmlParseError = xmlCatalogDocument.parseError
```

```
// JScript
var xmlParseError = xmlCatalogDocument.parseError;
```

The `XMLDOMParseError` object may be tested after loading an XML document to see if errors were encountered while attempting to parse the document. A list of the possible parse errors can be found in Appendix M.

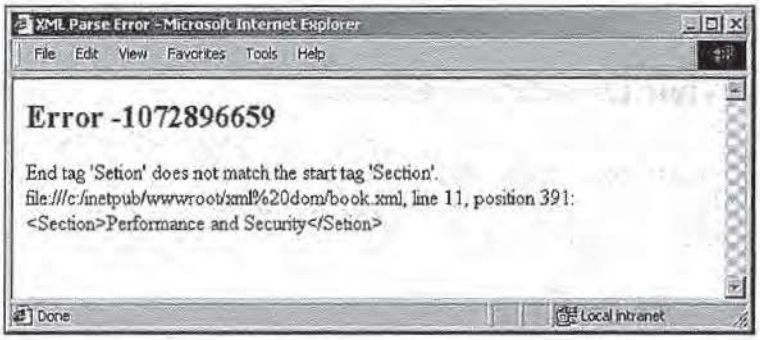
Properties
errorCode
filepos
line
linepos
reason
srcText
url

The following sample displays all the properties of the `XMLDOMParseError` object:

```
' VBScript
Dim xmlParseError
Set xmlParseError = xmlCatalogDocument.parseError
If xmlParseError.errorCode <> 0 Then
   strError = "<H2>Error " & xmlParseError.errorCode & "</H2>"
   strError = strError & xmlParseError.reason & "<BR>"
   strError = strError & xmlParseError.url
   strError = strError & ", line " & xmlParseError.line
   strError = strError & ", position " & xmlParseError.filepos & ":<BR>"
   strError = strError & Server.HTMLEncode(xmlParseError.srcText) & "<BR>"
   Response.Write strError
End If
```

```
// JScript
var xmlParseError = xmlCatalogDocument.parseError;
if (xmlParseError.errorCode != 0) {
   strError = "<H2>Error " + xmlParseError.errorCode + "</H2>";
   strError += xmlParseError.reason + "<BR>";
   strError += xmlParseError.url;
   strError += ", line " + xmlParseError.line;
   strError += ", position " + xmlParseError.filepos + ":<BR>";
   strError += Server.HTMLEncode(xmlParseError.srcText) + "<BR>";
   Response.write(strError);
}
```

This produces a simple generic error page:



Methods

There are no methods associated with the `XMLDOMParseError` object.

Properties

errorCode

The read-only `errorCode` property returns the error code of the last parse error encountered, in decimal.

```
Long = XMLDOMParseError.errorCode
```

filepos

The read-only `filepos` property returns the absolute file position (offset in characters expressed as a long integer) where the reported error occurred.

```
Long = XMLDOMParseError.filepos
```

line

The read-only `line` property returns the line number (expressed as a long integer) where the reported error occurred.

```
Long = XMLDOMParseError.line
```

linepos

The read-only `linepos` property returns the offset in the current line (expressed as a long integer) where the reported error occurred.

```
Long = XMLDOMParseError.linepos
```

reason

The read-only `reason` property returns a string containing a description of the error.

```
String = XMLDOMParseError.reason
```

srcText

The read-only `srcText` property returns a string containing the entire line on which the error occurred.

```
String = XMLDOMParseError.srcText
```

url

The read-only `url` property returns a string containing the URL of the file being parsed when the error was encountered. If the document is being built in memory, this property will return `NULL`.

```
String = XMLDOMParseError.url
```

XMLDOMProcessingInstruction

The XMLDOMProcessingInstruction object corresponds to a processing instruction declared in the document. Processing instructions are data passed directly to the application which will process the XML document. For example, in the following document fragment:

```
<Book>
   <?Indexer IGNORE_BOOK ?>
</Book>
```

The <Book> element would have one XMLDOMProcessingInstruction object in its list of child nodes. We could therefore get a reference to this XMLDOMProcessingInstruction object using:

```
' VBScript
Dim xmlPI
Set xmlPI = xmlBookElement.childNodes(0)
```

```
// JScript
var xmlPI = xmlBookElement.childNodes(0);
```

The XMLDOMProcessingInstruction object is derived from the XMLDOMNode object. In addition to the data and target properties described below, please see the properties, methods, and events defined for the XMLDOMNode object for additional functionality available through the XMLDOMProcessingInstruction object.

Methods	Properties	
appendChild*	attributes*	nodeTypeString*
cloneNode*	baseName*	nodeValue*
hasChildNodes*	childNodes*	nodeTypedValue*
insertBefore*	data	ownerDocument*
removeChild*	dataType*	parentNode*
replaceChild*	definition*	parsed*
selectNodes*	firstChild*	prefix*
selectSingleNode*	lastChild*	previousSibling*
transformNode*	namespaceURI*	specified*
transformNodeToObject*	nextSibling*	target
	nodeName*	text*
	nodeType*	xml*

* See section on XMLDOMNode methods and properties.

Additional Methods

There are no additional methods associated with the XMLDOMProcessingInstruction object.

Additional Properties

data

The read-only `data` property sets or returns the string representing the content of the processing instruction with the exception of the target.

```
XMLDOMProcessingInstruction.data = String
String = XMLDOMProcessingInstruction.data
```

target

The read-only `target` property returns the string containing the target of the processing instruction.

```
String = XMLDOMProcessingInstruction.target
```

In the example above, the `target` would be `"Indexer"`.

XMLDOMSchemaCollection

The `XMLDOMSchemaCollection` object provides a way to store namespace and schema information as a collection. This object allows us to manage the schemas and namespaces associated with an XML document and to access the nodes in a schema.

This object is only available in version 2.6 of MSXML.

An `XMLDOMSchemaCollection` object may be obtained by reading the `namespaces` and `schemas` properties of `XMLDOMDocument2`. For example, if we have an XML document like this:

```
<?xml version="1.0"?>
<Catalog xmlns="x-schema:schema.xml">
   <Book>
      <Sections>
         <Section>ASP Intrinsic Objects</Section>
         <Section>Scripting Objects</Section>
         <Section>ASP Components</Section>
         <Section>ActiveX Data Objects</Section>
         <Section>Extending ASP</Section>
         <Section>Miscellaneous Reference</Section>
      </Sections>
   </Book>
</Catalog>
```

The schema for this might be as follows:

```
<Schema name="bookschema" xmlns="urn:schemas-microsoft-com:xml-data"
        xmlns:dt="urn:schemas-microsoft-com:datatypes">
   <ElementType name="Section" content="textOnly"
                model="closed" dt:type="string" />
   <ElementType name="Sections" content="eltOnly"
                model="closed">
```

```
      <element type="Section" minOccurs="1" maxOccurs="*" />
   </ElementType>
   <ElementType name="Book" content="eltOnly"
                model="closed">
      <element type="Sections" minOccurs="1" maxOccurs="1" />
   </ElementType>
   <ElementType name="Catalog" content="eltOnly"
                model="closed">
      <element type="Book" minOccurs="1" maxOccurs="*" />
   </ElementType>
</Schema>
```

To access the `XMLDOMSchemaCollection` for this document, we would use:

```
' VBScript
Dim xmlSchemas
Set xmlSchemas = xmlCatalogDocument.namespaces
```

```
// JScript
var xmlSchemas = xmlCatalogDocument.namespaces;
```

Methods	Properties
add	length
addCollection	
get	
namespaceURI	
remove	

Methods

add

The `add` method is used to add a new schema to the collection. The first parameter is the definition of the namespace to be associated with the schema. The second parameter is the schema to be loaded. If the parameter is a string, it is treated as the URL of a schema. If the parameter is an `XMLDOMDocument` object, that object is loaded as if it were a schema and added to the collection. If the parameter is `NULL`, any schema for the namespace specified is removed from the collection.

```
XMLDOMSchemaCollection.add(namespaceURI, schema)
```

Parameter	Data Type	Description
namespaceURI	String	The namespace definition to be associated with the schema
schema	Object	The schema to be loaded into the collection, or `NULL` if the schema corresponding to the specified namespace is to be removed

For example:

```
' VBScript
' Add the schema in the xmlCatalogSchema document to the
' XMLCatalogSchemaCollection
xmlCatalogSchemaCollection.add("http://www.wrox.com/Catalog",
xmlCatalogSchema)
```

```
// JScript
// Add the schema in the xmlCatalogSchema document to the
// XMLCatalogSchemaCollection
xmlCatalogSchemaCollection.add("http://www.wrox.com/Catalog",
xmlCatalogSchema);
```

addCollection

The `addCollection` method is used to add all of the schemas from another existing `XMLDOMSchemaCollection` to the current `XMLDOMSchemaCollection`. If there are namespace collisions, the schemas in the existing `XMLDOMSchemaCollection` are replaced by the ones in the new `XMLDOMSchemaCollection` with the same namespace.

```
XMLDOMSchemaCollection.addCollection(objXMLDOMSchemaCollection)
```

Parameter	Data Type	Description
objXMLDOMSchemaCollection	`XMLDOMSchemaCollection`	Collection of schemas to be added to the current collection.

get

The `get` method is used to obtain the `XMLDOMNode` for the `<Schema>` element for the cached schema in the collection with the specified *namespaceURI*.

```
XMLDOMNode = XMLDOMSchemaCollection.get(namespaceURI)
```

Parameter	Data Type	Description
namespaceURI	String	The namespace for which the schema is to be returned.

For example, to access the `<Schema>` element of the schema given above, we would use:

```
' VBScript
Dim xmlSchemaNode
Set xmlSchemaNode = xmlSchemas.get("x-schema:schema.xml")
```

```
// JScript
var xmlSchemaNode = xmlSchemas.get("x-schema:schema.xml");
```

We can then access the other nodes in the schema through this. For example, we could access the first `<ElementType>` element as `xmlSchemaNode.firstChild`.

namespaceURI

The `namespaceURI` method is used to retrieve the namespace at the specified index in the collection. Information about any associated schema can then be discovered by using the `get` method on the returned `namespaceURI`.

```
String = XMLDOMSchemaCollection.namespaceURI(index)
```

Parameter	Data Type	Description
index	Long	Index of the namespace in the collection to be returned.

In the example XML document and associated schema given above, `xmlSchemas.namespaceURI(0)` would return `"x-schema:schema.xml"`.

remove

The `remove` method is used to remove the specified namespace from the collection.

```
XMLDOMSchemaCollection.remove(namespaceURI)
```

Parameter	Data Type	Description
namespaceURI	String	The namespace to be removed from the collection.

Properties

length

The `length` property returns the number of namespaces currently in the collection. This property is read-only.

```
Integer = XMLDOMSchemaCollection.length
```

XMLDOMSelection

The `XMLDOMSelection` object represents the list of nodes that match a particular XSL pattern or XPath expression. XPath expressions are discussed in-depth in the next chapter.

This object is only available in version 2.6 of MSXML.

An `XMLDOMSchemaCollection` object is obtained by calling the `selectNodes` method of `XMLDOMDocument2`. For example, given the following XML document:

```
<?xml version="1.0"?>
<Catalog>
   <Book>
      <Sections>
         <Section Author="Alex Homer">ASP Intrinsic Objects</Section>
         <Section Author="Alex Homer">Scripting Objects</Section>
         <Section Author="Alex Homer">ASP Components</Section>
         <Section Author="Brian Francis">ActiveX Data Objects</Section>
         <Section Author="Various">Extending ASP</Section>
         <Section Author="Wrox">Miscellaneous Reference</Section>
      </Sections>
   </Book>
</Catalog>
```

We could retrieve an `XMLDOMSelection` containing all of the `<Section>` elements where the `Author` attribute is set to `"Alex Homer"` using:

```
' VBScript
Dim xmlCatalogDocument, xmlSelection
Set xmlCatalogDocument = Server.CreateObject("MSXML2.DOMDocument")
xmlCatalogDocument.async = false
xmlCatalogDocument.load("http://servername/xml/books.xml")
Set xmlSelection = xmlCatalogDocument.selectNodes("//Section[@Author='Alex
                                                    Homer']")
```

```
// JScript
var xmlCatalogDocument = Server.CreateObject("MSXML2.DOMDocument");
xmlCatalogDocument.async = false;
xmlCatalogDocument.load("http://servername/xml/books.xml");
var xmlSelection = xmlCatalogDocument.selectNodes("//Section[@Author='Alex
                                                    Homer']");
```

Methods	Properties
clone	context
matches	expr
getProperty	length
item	
nextNode	
peekNode	
removeAll	
removeNext	
reset	

Methods

clone

The `clone` method creates an exact copy of the current `XMLDOMSelection` object with the same position and context.

```
XMLDOMSelection = XMLDOMSelection.clone()
```

matches

The `matches` method checks to see if the `XMLDOMNode` object passed in is part of the result set in the `XMLDOMSelection` object.

```
XMLDOMNode = XMLDOMSelection.matches(objXMLDOMNode)
```

Parameter	Data Type	Description
objXMLDOMNode	XMLDOMNode	The node to be checked against the `XMLDOMSelection`.

During testing, this method always returned `Nothing`.

getProperty

The `getProperty` method is used to look up a property of the `XMLDOMSelection` object. For version 2.6 of MSXML, the only valid value for this property is `"SelectionLanguage"`; it will be either `"XSLPattern"` or `"XPath"`, based on the selection method used to create the `XMLDOMSelection` object.

```
Variant = XMLDOMSelection.getProperty(name)
```

Parameter	Data Type	Description
name	String	The name of the property to look up the value of. Currently, this can only be `"SelectionLanguage"`.

item

The `item` method is used to access individual nodes in the `XMLDOMSelection` collection.

```
XMLDOMNode = XMLDOMSelection.item(index)
```

Parameter	Data Type	Description
index	Long	The ordinal position of the node in the selection which will be retrieved.

nextNode

The `nextNode` method is used to retrieve the next node in the `XMLDOMSelection` collection. Calling this method increments the iterator for the collection, so that calling it repeatedly will retrieve all elements in the collection.

```
XMLDOMNode = XMLDOMSelection.nextNode()
```

peekNode

The `peekNode` method is used to retrieve the next node in the `XMLDOMSelection` collection. Calling this method does not increment the iterator for the collection, so that calling it repeatedly will continue to retrieve the node just beyond the current iterator position.

```
XMLDOMNode = XMLDOMSelection.peekNode()
```

removeAll

The `removeAll` method is used to remove all of the nodes in the `XMLDOMSelection` object from the document.

```
XMLDOMSelection.removeAll()
```

removeNext

The `removeNext` method is used to remove the next node in the `XMLDOMSelection` collection from the document.

```
XMLDOMSelection.removeNext()
```

reset

The `reset` method is used to reset the position of the iterator on the `XMLDOMSelection` object to the beginning of the list.

```
XMLDOMSelection.reset()
```

Properties

context

The `context` property sets or returns the node for which the results in the `XMLDOMSelection` object apply. Setting this property reinitializes the object for the specified node.

```
XMLDOMNode = XMLDOMSelection.context
XMLDOMSelection.context = XMLDOMNode
```

expr

The `expr` property sets or returns the XSL pattern or XPath expression for which the results in the `XMLDOMSelection` object apply. In the example above, `expr` would return `"//Section[@Author='Alex Homer']"`. Setting this property reinitializes the object using the specified query string.

```
String = XMLDOMSelection.expr
XMLDOMSelection.expr = String
```

length

The `length` property returns the number of nodes in the `XMLDOMSelection` object. This property is read-only.

```
Long = XMLDOMSelection.length
```

XMLDOMText

The XMLDOMText object represents a text node which is used to contain the parsed character data contained in an XML element or attribute: element and attribute nodes do not themselves contain any text; their content is always contained in a child text node.

We can access a text node using the normal XMLDOMNode object properties, so if we have an XML element:

```
<Title>ASP Programmer's Reference</Title>
```

We can get a reference to the text node using:

```
' VBScript
Dim xmlTextNode
Set xmlTextNode = xmlTitleElement.firstChild
```

```
// JScript
var xmlTextNode = xmlTitleElement.firstChild;
```

We can also access the text node's content as a string using the element's `text` property.

This object is derived from the XMLDOMCharacterData object, that itself is derived from the XMLDOMNode object. In addition to the `splitText` method of the XMLDOMText object, see the methods and properties defined for the XMLDOMCharacterData and XMLDOMNode objects for additional functionality available through the XMLDOMText object.

Methods	Properties	
appendChild*	attributes*	ownerDocument*
appendData†	baseName*	parentNode*
cloneNode*	childNodes*	parsed*
deleteData†	data†	prefix*
hasChildNodes*	dataType*	previousSibling*
insertBefore*	definition*	specified*
insertData†	firstChild*	text*
removeChild*	lastChild*	xml*
replaceChild*	length†	
replaceData†	namespaceURI*	
selectNodes*	nextSibling*	
selectSingleNode*	nodeName*	
splitText	nodeType*	
substringData†	nodeTypedValue*	
transformNode*	nodeTypeString*	
transformNodeToObject*	nodeValue*	

* See section on XMLDOMNode methods and properties.
† See section on XMLDOMCharacterData methods and properties.

Additional Methods

splitText

The `splitText` method splits the specified node into two nodes, breaking the data content apart at the *`offset`* location. It then creates an adjacent sibling node of the same type and inserts it at the appropriate location at the node tree. The nodes can be rejoined using the element's `normalize` method.

```
XMLDOMText = XMLDOMText.splitText(offset)
```

Parameter	Data Type	Description
`offset`	Long	Position in text node where it is to be split.

Additional Properties

There are no additional properties associated with the `XMLDOMText` object.

XMLHttpRequest

The `XMLHttpRequest` object is used to transmit XML documents over HTTP. It is used primarily on the client side to access information that is available through the `Request` and `Response` objects on the server, so we won't look at it in too much detail here. A typical application of this object might be to transmit an XML document from a client to a server application of some kind, which processes the XML document and returns another XML document to the client. It can be instantiated using the Prog ID `"Microsoft.XMLHTTP"`.

Methods	Properties	Events
`abort`	`readyState`	`onreadystatechange`
`getAllResponseHeaders`	`responseBody`	
`getResponseHeader`	`responseStream`	
`open`	`responseText`	
`send`	`responseXML`	
`setRequestHeader`	`status`	
	`statusText`	

Methods

abort

The `abort` method cancels the current request. This returns the object to an uninitialized state, and the `open` method must be called again before continuing.

```
XMLHttpRequest.abort()
```

getAllResponseHeaders

The `getAllResponseHeaders` method returns the header information sent in response to a call to the `send` method. The information is returned as a sequence of name-value pairs, separated by carriage-return linefeed pairs.

```
String = XMLHttpRequest.getAllResponseHeaders()
```

getResponseHeader

The `getResponseHeader` method returns the value of the header with the specified name from the headers sent in response to a call to the `send` method.

```
String = XMLHttpRequest.getResponseHeader(header)
```

Parameter	Data Type	Description
header	String	The name of the header to be retrieved.

For example, to get the `"last-modified"` header:

```
' VBScript
Dim strCatalogLastModified
strCatalogLastModified =
                    xmlCatalogHttpRequest.getResponseHeader("last-modified")
```

```
// JScript
var sCatalogLastModified =
                    xmlCatalogHttpRequest.getResponseHeader("last-modified");
```

open

The `open` method initializes a request, and sets information regarding that request. The *method* parameter is the HTTP method to be used for the request: `GET`, `POST`, and so on. The `url` parameter is the URL of the request. The `async` flag is used to set synchronous or asynchronous operation for the request (the default is asynchronous). The `user` and `password` parameters are used to specify security information for the requested object. Note that the actual request does not take place until the `send` method is called.

```
XMLHttpRequest.open(method, url, [async], [user], [password])
```

Parameter	Data Type	Description
method	String	The HTTP method, e.g. `GET`, `POST` etc.
url	String	URL of the request.
async	Variant	Optional. Determines whether synchronous or asynchronous operation will be used.
user	Variant	Optional security information.
password	Variant	Optional security information.

For example:

```
' VBScript
' Open the catalog XML document
xmlCatalogHttpRequest.open "GET", "catalog.xml"
```

```
// JScript
// Open the catalog XML document
xmlCatalogHttpRequest.open("GET", "catalog.xml");
```

send

The `send` method issues the request to the URL selected in the call to the `open` method. From VBScript, you can pass it a string containing the request information, an array of unsigned bytes, or an `XMLDOMDocument` object. For strings and arrays of unsigned bytes, use `setRequestHeader` to assign the content type and character set.

```
XMLHttpRequest.send(var_header)
```

Parameter	Data Type	Description
var_header	Variant	String of request information, array of unsigned bytes, or an `XMLDOMDocument` object.

setRequestHeader

The `setRequestHeader` method sets a value for a header on a request, and should be called if necessary after calling the `open` method but before calling the `send` method. The two parameters are the name of the header and its value, respectively. If another header has already been set with the same name, it is replaced.

```
XMLHttpRequest.setRequestHeader(bstr_header, bstr_value)
```

Parameter	Data Type	Description
bstr_header	String	The HTTP request header to be set.
bstr_value	String	The value to be set on the request header.

Properties

readyState

The read-only `readyState` property returns a long integer that describes the status of the request.

```
Long = XMLHttpRequest.readyState
```

The possible values are:

- 0 ("uninitialized"), i.e. the object has been created but the load method has not yet been executed.
- 1 ("loading"), i.e. the reply is loading, but has not yet entered the parsing step.
- 2 ("loaded"), i.e. the reply has been loaded and is being parsed.
- 3 ("interactive"), i.e. the reply has been partially parsed and a read-only version of the object model is available.
- 4 ("complete"), i.e. the reply has been completely parsed, successfully or unsuccessfully.

responseBody

The read-only responseBody property returns an array of unsigned bytes containing the raw response from the server. Note that this may contain encoded data depending on what the server transmitted.

```
Variant = XMLHttpRequest.responseBody
```

responseStream

The read-only responseStream property returns a stream object representing the raw response from the server. Note that this may contain encoded data depending on what the server transmitted.

responseText

The read-only responseText property returns a string representing the response from the server. The object attempts to decode the response as a Unicode string, so this field may contain meaningless information if the response is a BLOB or some other non-text response.

```
String = XMLHttpRequest.responseText
```

responseXML

The read-only responseXML property returns an XMLDOMDocument representing the returned document as parsed by the Microsoft parser. This property will only be populated if the MIME type of the response is correctly set to "text/xml". Note that the returned document is not validated against its DTD or schema (if one is present) by the parser before being returned.

```
XMLDOMDocument = XMLHttpRequest.responseXML
```

status

The read-only status property returns a long integer containing the HTTP status code received in response to a request.

```
Long = XMLHttpRequest.status
```

statusText

The read-only `statusText` property returns a string containing the HTTP response line status received in response to a request.

```
String = XMLHttpRequest.statusText
```

Events

onreadystatechange

The `onreadystatechange` event is fired whenever the value of the `readyState` property for the object changes. When used in conjunction with the `async` property, this allows your script to continue executing while waiting for an XML HTTP request to return. Event handlers are defined in VBScript and JScript by setting the value of a write-only property with the same name as the event to the name of the callback procedure used to respond to the event. See below for an example of this.

```
XMLHttpRequest.onreadystatechange = String
```

```
' VBScript
' set up an event handler for the onreadystatechange event
xmlCatalogHttpRequest.onreadystatechange = checkCatalogReadyStateChange
...
Sub checkCatalogReadyStateChange()
   ' handle the onreadystatechange event for the Catalog document
   ...
End Sub
```

```
// JScript
// set up an event handler for the onreadystatechange event
xmlCatalogHttpRequest.onreadystatechange = checkCatalogReadyStateChange;
...
function checkCatalogReadyStateChange() {
   // handle the onreadystatechange event for the Catalog document
   ...
}
```

XSLProcessor

The `XSLProcessor` object is used to perform transformations using compiled stylesheets.

This object is only available in version 2.6 of MSXML.

An `XSLProcessor` object may be obtained by calling the `createProcessor` method of the `XSLTemplate` object:

```
' VBScript
Dim xslTemplate, xslProcessor
Set xslTemplate = Server.CreateObject("MSXML2.XSLTemplate")
Set xslProcessor = xslTemplate.createProcessor
```

```
// JScript
var xslTemplate = Server.CreateObject("MSXML2.XSLTemplate");
var xslProcessor = xslTemplate.createProcessor();
```

The following code shows an example of using these two objects. We must first load our XML document and stylesheet as normal, except that we must use the free-threaded version of the `XMLDOMDocument2` object, or an error will be generated (due to the use of mixed threading models). We then instantiate our `XSLTemplate` object and set its `stylesheet` property to our XSL document and create the `XSLProcessor` object. Now we just need to set its `input` property (the XML node object which will be transformed, in this case our `xmlCatalogDocument`) and `output` property (the object to which the transformed XML will be sent; here, we send it directly to the browser). Finally we call the `XSLProcessor` object's `transform` method to perform the actual transformation.

The VBScript version of this code is:

```
Set xmlCatalogDocument = _
                         Server.CreateObject("MSXML2.FreeThreadedDOMDocument")
xmlCatalogDocument.async = false
xmlCatalogDocument.load("http://servername/xml/books.xml")

Set xslCatalogDocument = _
                         Server.CreateObject("MSXML2.FreeThreadedDOMDocument")
xslCatalogDocument.async = false
xslCatalogDocument.load("http://servername/xml/books.xsl")

Set xmlXSLTemplate = Server.CreateObject("MSXML2.XSLTemplate")
Set xmlXSLTemplate.stylesheet = xslCatalogDocument
Set xmlXSLProcessor = xmlXSLTemplate.createProcessor

xmlXSLProcessor.input = xmlCatalogDocument
xmlXSLProcessor.output = Response
xmlXSLProcessor.transform
```

And the JScript:

```
var xmlCatalogDocument =
                         Server.CreateObject("MSXML2.FreeThreadedDOMDocument");
xmlCatalogDocument.async = false;
xmlCatalogDocument.load("http://servername/xml/books.xml");

var xslCatalogDocument =
                         Server.CreateObject("MSXML2.FreeThreadedDOMDocument");
xslCatalogDocument.async = false;
xslCatalogDocument.load("http://servername/xml/books.xsl");

var xmlXSLTemplate = Server.CreateObject("MSXML2.XSLTemplate");
xmlXSLTemplate.stylesheet = xslCatalogDocument;
var xmlXSLProcessor = xmlXSLTemplate.createProcessor();

xmlXSLProcessor.input = xmlCatalogDocument;
xmlXSLProcessor.output = Response;
xmlXSLProcessor.transform();
```

Methods	Properties
addObject	input
addParameter	output
reset	readyState
setStartMode	startMode
transform	startModeURI
	stylesheet
	ownerTemplate

Methods

addObject

The `addObject` method is used to pass objects to a stylesheet. Numbers are converted to doubles, other values are converted to strings, and objects return an error.

```
XSLProcessor.addObject(obj, [namespaceURI])
```

Parameter	Data Type	Description
obj	Object	The object to be passed to the stylesheet.
namespaceURI	String	Optional. The namespace URI of the object to be passed to the stylesheet.

For example:

```
' VBScript
' Pass catalog information as parameters to the xmlCatalogProcessor object
xmlCatalogProcessor.addObject("catalog", "http://www.wrox.com/Catalog")
```

```
// JScript
// Pass catalog information as parameters to the xmlCatalogProcessor object
xmlCatalogProcessor.addObject("catalog", "http://www.wrox.com/Catalog");
```

addParameter

The `addParameter` method is used to pass values to a stylesheet that may be referenced by using the `<xsl:param>` element.

```
XSLProcessor.addParameter(baseName, parameter, [namespaceURI])
```

Parameter	Data Type	Description
baseName	String	The base name of the parameter to be passed to the stylesheet.
parameter	String	The value of the parameter to be passed to the stylesheet.
namespaceURI	String	Optional. The namespace URI of the parameter.

reset

The `reset` method is used to abort any processing currently being performed by the `XSLProcessor` object and discard any results.

```
XSLProcessor.reset()
```

setStartMode

The `setStartMode` method is used to set the processing mode to be used by the `XSLProcessor` object. This will only apply those templates in the stylesheet that match the mode provided.

```
XSLProcessor.setStartMode(mode, [namespaceURI])
```

Parameter	Data Type	Description
mode	String	The mode in which the template is to be processed.
namespace URI	String	Optional. The namespace of the mode in which the template is to be processed.

For example:

```
' VBScript
' Set the catalog processor to operate in Public mode
xmlCatalogProcessor.setStartMode "public"
```

```
// JScript
// Set the catalog processor to operate in Public mode
xmlCatalogProcessor.setStartMode("public");
```

transform

The `transform` method starts or resumes a transformation operation using the `XSLProcessor` object. It returns `false` if the transformation did not complete successfully, or `true` if it did.

```
Boolean = XSLProcessor.transform()
```

Properties

input

The `input` property specifies the node at the top of the node tree to be transformed. Setting this property resets the state of the `XSLProcessor` object.

```
XMLDOMNode = XSLProcessor.input
XSLProcessor.input = XMLDOMNode
```

output

The `output` property specifies a target for the `XSLProcessor` object to write its output to. This may be any object that supports the `IStream` or `IPersistStream` interface (such as the ADO `Stream` object or the `Response` object); it may also be an `XMLDOMDocument` object. If this property is read from, it returns a string that is the incrementally-buffered output of the transformation process; successive reads to this property will return successive portions of the transformation result.

```
XSLProcessor.output = Variant
Variant = XSLProcessor.output
```

ownerTemplate

The `ownerTemplate` property returns the `XSLTemplate` object for the template that was used to create this `XSLProcessor` object.

```
XCMLDOMXSLTemplate = XSLProcessor.ownerTemplate
```

readyState

The read-only `readyState` property returns a long integer that describes the status of the current transformation.

```
Long = XSLProcessor.readyState
```

The possible values are:

- `0 ("READYSTATE_UNINITIALIZED")` i.e. the object has been created but the required properties has not yet been set.
- `1 ("READYSTATE_LOADED")` i.e. all properties have been set but the `transform` method has not yet been called.
- `2 ("READYSTATE_INTERACTIVE")` i.e. the transformation has begun but not completed.
- `3 ("READYSTATE_COMPLETE")` i.e. the transformation has completed and all the resultant output is available.

startMode

The `startMode` property returns the basename part of the starting mode for the processor. The default start mode is an empty string (i.e., no mode). This property is read-only – to set the start mode, invoke the `setStartMode` method.

```
String = XSLProcessor.startMode
```

startModeURI

The `startModeURI` property returns the URI of the namespace part of the starting mode for the processor. The default start mode is an empty string (i.e., no mode). This property is read-only – to set the start mode, invoke the `setStartMode` method.

```
String = XSLProcessor.startModeURI
```

stylesheet

The `stylesheet` property sets or returns the `XMLDOMNode` object representing the stylesheet that is to be used by the processor.

```
XMLDOMNode = XSLProcessor.stylesheet
XSLProcessor.stylesheet = XMLDOMNode
```

XSLTemplate

The `XSLTemplate` object is used to cache compiled XSL templates.

> **This object is only available in version 2.6 of MSXML.**

An `XSLTemplate` object may be instantiated using the `Server.CreateObject` function with a Prog ID of `"MSXML2.XSLTemplate"`:

```
' VBScript
Dim xslTemplate
Set xslTemplate = Server.CreateObject("MSXML2.XSLTemplate")
```

```
// JScript
var xslTemplate = Server.CreateObject("MSXML2.XSLTemplate");
```

Methods	Properties
createProcessor	stylesheet

Methods

createProcessor

The `createProcessor` method is used to create a new `XSLProcessor` object that may be used to transform documents using the cached stylesheet.

```
XSLProcessor = XSLTemplate.createProcessor()
```

Properties

stylesheet

The `stylesheet` property sets or returns the `XMLDOMNode` object representing the stylesheet to be used when creating processors from this template. Setting this property will replace the currently cached stylesheet.

```
XMLDOMNode = XSLProcessor.stylesheet
XSLProcessor.stylesheet = XMLDOMNode
```

Summary

In this rather long chapter, we've had a look at the objects of Microsoft's implementation of the XML DOM and their methods and properties. These objects allow us to manipulate an XML document before it is sent to the client, and also to transform it through an XSL stylesheet. Because of the amount of material we've had to cover, we've had to be very concise. We don't expect this chapter to help you learn XML from scratch, but once you've mastered the basics, we think you'll find that this reference will be more and more useful.

"Professional XML" by Wrox Press deals with the concept of XML with particular focus on real-world applications.