MICROSOFT® PROGRAMMING SERIES

Microsoft®

E-Book Included

CD-ROM

CD-ROM includes sample code in Visual Basic®, VBScript, and JScript®!

Developing XML Solutions

Enable seamless business-to-business data exchange with XML, BizTalk™, and SOAP

Jake Sturm

MICROSOFT® PROGRAMMING SERIES

Microsoft®

Developing XML Solutions

Jake Sturm

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Cataloging-in-Publication Data
Sturm, Jake, 1961-
Developing XML Solutions / Jake Sturm.
p. cm.
Includes index.
ISBN 0-7356-0796-6
1. XML (Document markup language) 2. Electronic data processing--Distributed processing. 3. Web sites--Design. I. Title.
QA76.76.H94 S748 2000
005.7'2--dc21 00-031887

Printed and bound in the United States of America.

1 2 3 4 5 6 7 8 9 MLML 5 4 3 2 1 0

Distributed in Canada by Penguin Books Canada Limited.

A CIP catalogue record for this book is available from the British Library.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office or contact Microsoft Press International directly at fax (425) 936-7329. Visit our Web site at mspress.microsoft.com. Send comments to *mspinput@microsoft.com*.

Acquisitions Editor: Eric Stroo
Project Editor: Denise Bankaitis
Technical Editor: Julie Xiao
Manuscript Editors: Denise Bankaitis, Jennifer Harris

Contents

Part II XML and Windows DNA

Introduction

This book is intended for anyone who wants a glimpse into the next generation of enterprise development. If you want to develop an understanding of Extensible Markup Language (XML) and learn how to use XML for business-to-business (B2B) communications, learn what the Simple Object Access Protocol (SOAP) and BizTalk extensions are, and learn how to use Microsoft Internet Explorer 5 with XML, this book will provide the information you need. You are assumed to have a basic understanding of Microsoft Visual Basic and the Visual Basic Integrated Development Environment (IDE). Developers will find code samples, a discussion of the Internet Explorer 5 document object model, and many more topics. Web developers will find material on using XML to build Web pages. Senior developers and managers will find discussions on how XML can be integrated into the enterprise. Some of the World Wide Web Consortium (W3C) specifications discussed in this book are not final, and they are changing constantly. It is recommended that you visit the W3C Web site at *http://www.w3.org* often for the updated specifications.

What Is in This Book

This book provides a detailed discussion of what XML is and how it can be used to build a Digital Nervous System (DNS) using the Microsoft Windows DNA framework with SOAP 1.1, BizTalk Framework 2.0, and Internet Explorer 5. The book is divided into two parts. Part I covers all the essential elements of XML and enterprise development using SOAP and BizTalk. Part II covers XML and Windows DNA. It discusses how to use Internet Explorer 5 and the Windows DNA framework to build enterprise systems. Throughout the book, you will find code samples that will bring all the ideas together.

Part I: Introducing XML

Chapter 1 discusses how XML fits within the enterprise. It provides an overview of DNS, XML, and knowledge workers and includes a discussion of where XML solutions fit into the DNS.

Chapter 2 gives a general overview of markup languages. The chapter begins with a brief history of markup languages. Next, the three most important markup languages are discussed: Standard Generalized Markup Language (SGML), Hypertext Markup Language (HTML), and XML.

Chapter 3 covers the basic structure of an XML document. Topics include XML elements, attributes, comments, processing instructions, and well-formed documents. Some of the more common XML tools will be discussed and demonstrated in this chapter.

Chapter 4 introduces the *document type definition (DTD)*. The DTD is an optional document that can be used to define the structure of XML documents. This chapter provides an overview of DTDs, discusses the creation of valid documents, and describes the DTD syntax and how to create XML document structures using DTDs.

Chapter 5 examines DTD entities. This chapter shows you how to declare external, internal, general, and parameter entities and how these entities will be expanded in the XML document and the DTD.

Chapter 6 covers four of the specifications that support XML: XML Namespaces, XML Path Language (XPath), XML Pointer Language (XPointer), and XML Linking Language (XLink). This chapter provides an overview of namespaces, including why they are important and how to declare them. The chapter will also cover how XPath, XLink, and XPointers can be used to locate specific parts of an XML document and to create links in an XML document.

Chapter 7 covers XML schemas. This chapter discusses some of the shortcomings of DTDs, what a schema is, and the elements of a schema.

Chapter 8 is all about SOAP, version 1.1. This chapter covers the problems associated with firewalls and procedure calls and using SOAP for interoperability. Examples demonstrate how to use SOAP in enterprise solutions.

Chapter 9 examines the BizTalk Framework 2.0. A detailed discussion of BizTalk tags and BizTalk schemas is provided. The next generation of products that will support BizTalk is also discussed. The rest of the chapter focuses on using BizTalk in enterprise solutions.

Part II: XML and Windows DNA

Chapter 10 provides an overview of the Windows DNA framework and the two fundamental models of the Windows DNA framework: the logical and physical models. This chapter focuses on the logical three-tier model, which is defined by the services performed by components of the system. These services fall into three basic categories: user services components, business services components, and data services components. The chapter ends with a discussion of Windows DNA system design.

Chapter 11 covers the majority of the objects in the XML Document Object Model (DOM). This chapter examines how to use the DOM and provides numerous code samples showing how to work with the DOM objects. The DOM objects not covered in Chapter 11 are discussed in Chapter 12.

Chapter 12 discusses how to present XML data in a Web browser using Extensible Stylesheet Language (XSL), how to transform XML documents using XSL Transformations (XSLT), and how to build static user services components using XML. The rest of the chapter examines XSL and XSLT support in the XML DOM and programming with XSL and XSLT.

Chapter 13 covers the creation of dynamic Web-based user services components using Dynamic HTML (DHTML) and the XML Data Source Object (DSO) available in Internet Explorer 5. This chapter will discuss how to use DHTML to create user services components that can respond directly to input from users. The rest of the chapter covers how to use the XML DSO to work directly with XML data embedded in HTML code.

Chapter 14 examines how XML can be used to build business services components. This chapter shows you how to create business services components using HTML Components (HTC).

Chapter 15 explores using XML in the data services component. This chapter discusses using ActiveX Data Objects (ADO) with XML, the Microsoft XML SQL Server Internet Server Application Programming Interface (ISAPI) extension, and the XSL ISAPI extension. The SQL ISAPI extension allows data in a SQL Server 6.5 or 7.0 database to be retrieved directly through Microsoft Internet Information Server (IIS) as XML. The XSL ISAPI extension allows XSL documents to be automatically converted to XML when a browser other than Internet Explorer 5 requests data.

Chapter 16 introduces Microsoft BizTalk Server 2000. BizTalk Server 2000 allows corporations to pass information within the corporation and between the corporation and its partners using XML.

XML TOOLS

There are a number of XML tools available to assist you in developing XML applications. You will find some of these tools used in examples throughout this book. The tools I use are XML Authority from Extensibility, Inc., XML Spy from Icon Informations-System, and XML Pro from Vervet Logic. XML Authority provides a comprehensive design environment that accelerates the creation, conversion, and management of XML schemas. XML Spy is a tool for viewing and editing an XML document. XML Pro is an XML editing tool that enables you to create and edit XML documents using menus and screens. You can download Extensibility's tools from *www.extensibility.com*, XML Spy from *http://xmlspy.com*, and XML Pro from *www.vervet.com*.

Please note these products are not under the control of Microsoft Corporation, and Microsoft is not responsible for their content, nor should their reference in this book be construed as an endorsement of a product or a Web site. Microsoft does not make any warranties or representations as to third party products.

USING THE COMPANION CD

The CD included with this book contains all sample programs discussed in the book, Microsoft Internet Explorer 5, third-party software, and an electronic version of the book. You can find the sample programs in the Example Code folder.

To use this companion CD, insert it into your CD-ROM drive. If AutoRun is not enabled on your computer, run StartCD.exe in the root folder to display the Start menu.

Installing the Sample Programs

You can view the samples from the companion CD, or you can install them onto your hard disk and use them to create your own applications.

Installing the sample programs requires approximately 162 KB of disk space. To install the sample programs, insert the companion CD into your CD-ROM drive and run Setup.exe in the Setup folder. Some of the sample programs require that the full version of Internet Explorer 5 be installed to work properly. If your computer doesn't have Internet Explorer 5 installed, run ie5setup.exe in the MSIE5 folder to install Internet Explorer 5. If you have trouble running any of the sample files, refer to the Readme.txt file in the root directory of the companion CD or to the text in the book that describes the sample program.

You can uninstall the samples by selecting Add/Remove Programs from the Microsoft Windows Control Panel, selecting Developing XML Solutions Example Code, and clicking the Add/Remove button.

Electronic Version of the Book

The complete text of *Developing XML Solutions* has been included on the companion CD as a fully searchable electronic book. To view the electronic book, you must have a system running Microsoft Windows 95, Microsoft Windows 98, Microsoft Windows NT 4 Service Pack 3 (or later), or Microsoft Windows 2000. You must also have Microsoft Internet Explorer 4.01 or later and the latest HTML Help components installed on your system. If you don't have Internet Explorer 4.01 or later, the setup wizard will offer to install a light version of Internet Explorer 5, which is located in the Ebook folder. The Internet Explorer setup has been configured to install the minimum files necessary and won't change your current settings or associations.

System Requirements

The XML samples in this book can be run using a computer that has at least the following system requirements.

- 486 or higher processor
- Windows 95, Windows 98, Windows NT 4.0, or Windows 2000
- Visual Basic 6 (If you want to perform the Visual Basic examples in the book, you will need to have this installed on your computer.)

MICROSOFT PRESS SUPPORT INFORMATION

Every effort has been made to ensure the accuracy of this book and the contents of the companion CD. Microsoft Press provides corrections for books through the World Wide Web at the following address: *http://mspress.microsoft.com/support/*.

If you have comments, questions, or ideas regarding this book or the companion CD, please send them to Microsoft Press using either of the following methods:

Postal Mail:
Microsoft Press
Attn: Developing XML Solutions Editor
One Microsoft Way
Redmond, WA 98052-6399
E-mail:
MSPINPUT@MICROSOFT.COM

Please note that product support is not offered through these addresses.

Chapter 8

SOAP

Simple Object Access Protocol (SOAP) version 1.1 is an industry standard designed to improve cross-platform interoperability using the Web and XML. The Web has evolved from simply pushing out static pages to creating customized content that performs services for users. A user can be a customer retrieving specialized Web pages for placing orders or a business partner using a customized form for reviewing stock and sales figures. A wide range of components located on various computers are involved in performing these Web-based services. Because these systems consist of many computers, including the client computer, middle-tier servers, and usually a database server, these systems are called *distributed systems*. To understand how SOAP works, let's take a look at the distributed system first.

COMMUNICATION OVER DISTRIBUTED SYSTEMS

Distributed systems commonly use two models for communication: *message passing* (which can be combined with message queuing) and *request/response messaging system*. A message passing system allows messages to be sent at any time. Once a message has been sent, the application that sent the message usually moves on. This type of system is called asynchronous. An asynchronous system typically uses messages, but it can also be based on other models. With the request/response model, the request and the response are paired together and can be thought of as

a synchronous system. The request is sent by an application, and the application usually waits until a response is received before continuing. When one application calls an object on another computer by making a Remote Procedure Call (RPC), we can think of this call as synchronous request/response message passing.

The request/response model is commonly used to allow components on different computers to communicate with each other using RPCs. Over the last several years, many attempts have been made to develop a standard that would allow this communication between components on different computers. Currently, the two most commonly used standards are Distributed Component Object Model (DCOM) and the Object Management Group's Internet Inter-Orb Protocol (IIOP). Both of these standards work well; their greatest shortcoming is that they do not natively interoperate with each other. Therefore, you cannot arbitrarily make a call to a component on a server from a client without first knowing what standard that server is using. Usually, you will also have to configure the client so that it can communicate with the server, especially when there are security issues. DCOM works best when all the computers in the system are using Microsoft operating systems. An IIOP system works best when all the computers in the system use the same *CORBA Object Request Broker* (ORB).[1]

> **NOTE** IIOP is only a specification: it will be up to individual vendors to create an implementation of the specification in the form of an ORB. There are currently many different ORBs.

When you are working on an internal system, it might be possible to limit the system to one platform or the other. Once you start working with the Internet or expanding the intranet out to extranets (for example, networks that include the corporation and its partners), it will usually be impossible to have a uniform platform across the entire system. At this point, DCOM and IIOP will no longer allow communication between any two components within the system, and neither of these two standards allows users to cross trust domains easily. Thus, for larger systems expanding across computers with multiple platforms, we need a way to enable objects to communicate with each other. The solution to this problem is SOAP.

---

1. CORBA stands for Common Object Request Broker Architecture, a specification developed by the Object Management Group. This specification provides the standard interface definition between objects in different programs, even if these programs are written in different programming languages and are on different platforms. In CORBA, ORB acts as a "broker" between a client request for an object from a distributed object and the completion of that request.

SOAP AND THE REQUEST/RESPONSE MODEL

The SOAP standard introduces no new concepts—it's built completely from existing technology. It currently uses HTTP as its request/response messaging transport and is completely platform independent. As you know, HTTP connects computers across the entire world. HTTP can go through firewalls and is the easiest means to transport messages to any computer in the world. It's likely that SOAP will evolve to use other protocols in the future.

A SOAP *package* contains information that can be used to invoke a method. How that method is called is not defined in the SOAP specification. SOAP also does not handle distributed garbage collection, *message boxcarring,*[2] type safety, or bidirectional HTTP. What SOAP does allow you to do is pass parameters and commands between HTTP clients and servers, regardless of the platforms and applications on the client and server. The parameters and commands are encoded using XML. Let's take a look at how SOAP uses the standard HTTP headers.

HTTP HEADERS AND SOAP

Two types of headers are available in HTTP: request headers and response headers. When you are using your Web browser to surf the Internet, each time you navigate to a new URL the Web browser will create a request and send it to the Web server. These requests are written in plain text; each has headers in a standard format. When creating SOAP messages, you will be adding additional information to these standard formats. HTTP servers generate a response message upon receiving the client request. This message contains a status line and response headers. Let's look at the two headers in more detail.

Request Headers

A typical HTTP message in a SOAP request being passed to a Web server looks like this:

```
POST /Order HTTP/1.1
Host: www.northwindtraders.com
Content-Type: text/xml
Content-Length: nnnn
SOAPAction: "urn:northwindtraders.com:PO#UpdatePO"

Information being sent would be located here.
```

2. A boxcar message is a type of message that contains more than one business document.

The first line of the message contains three separate components: the request method, the request URI, and the protocol version. In this case, the request method is *POST*; the request URI is */Order*; and the version number is *HTTP/1.1*. The Internet Engineering Task Force (IETF) has standardized the request methods. The GET method is commonly used to retrieve information on the Web. The POST method is used to pass information from the client to the server. The information passed by the POST method is then used by applications on the server. Only certain types of information can be sent using GET; any type of data can be sent using POST. SOAP also supports sending messages using M-POST. We'll discuss this method in detail later in this chapter. When working with the POST method in a SOAP package, the request URI actually contains the name of the method to be invoked.

The second line is the URL of the server that the request is being sent to. The request URL is implementation specific—that is, each server defines how it will interpret the request URL. In the case of a SOAP package, the request URL usually represents the name of the object that contains the method being called.

The third line contains the content type, *text/xml*, which indicates that the *payload* is XML in plain text format. The payload refers to the essential data being carried to the destination. The payload information could be used by a server or a firewall to validate the incoming message. A SOAP request must use the *text/xml* as its content type. The fourth line specifies the size of the payload in bytes. The content type and content length are required with a payload.

The *SOAPAction* header field must be used in a SOAP request to specify the intent of the SOAP HTTP request. The fifth line of the message, *SOAPAction: "urn:northwindtraders.com:PO#UpdatePO"*, is a namespace followed by the method name. By combining this namespace with the request URL, our example calls the *UpdatePO* method of the *Order* object and is scoped by the *urn:northwindtraders.com:PO* namespace URI. The following are also valid SOAPAction header field values:

```
SOAPAction: "UpdatePO"
SOAPAction: ""
SOAPAction:
```

The header field value of the empty string means that the HTTP request URI provides the intent of the SOAP message. A header field without a specified value indicates that the intent of the SOAP message isn't available.

Notice that there is a single blank line between the fifth line and the payload request. When you are working with message headers, the carriage-return/line-feed sequence delimits the headers and an extra carriage-return/line-feed sequence is used to signify that the header information is complete and that what follows is the payload.

Response Headers

A typical response message that contains the response headers is shown here:

```
200 OK
Content-Type: text/plain
Content-Length: nnnn

Content goes here.
```

The first line of this message contains a status code and a message associated with that status code. In this case, the status code is *200* and the message is *OK*, meaning that the request was successfully decoded and that an appropriate response was returned. If an error had occurred, the following headers might have been returned:

```
400 Bad Request
Content-Type: text/plain
Content-Length: 0
```

In this case, the status code is *400* and the message is *Bad Request*, meaning that the request cannot be decoded by the server because of incorrect syntax. You can find other standard status codes in RFC 2616.

SIMPLE SOAP PAYLOADS

As you can see, SOAP uses HTTP as the request/response messaging transport. We can add a SOAP request payload in the request message and a response payload in the response message. In this way, we can issue an RPC to any component using HTTP.

The Payload for a Request Message

The SOAP specification defines several SOAP elements that can be used with a SOAP request: *envelope*, *head*, and *body*. The envelope is a container for the head and body. The head contains information about the SOAP message, and the body contains the actual message. Namespaces are used to distinguish the SOAP elements from the other elements of the payload. For example, *SOAP-ENV:Envelope*, *SOAP-ENV:Head*, and *SOAP-ENV:Body* are used in a SOAP document.

The SOAP schema for the envelope will look as follows:

```
<?xml version="1.0" ?>
<!-- XML Schema for SOAP v 1.1 Envelope -->
<!-- Copyright 2000 DevelopMentor, International Business
```

(continued)

```
Machines Corporation, Lotus Development Corporation,
Microsoft, UserLand Software -->
<schema xmlns="http://www.w3.org/1999/XMLSchema"
    xmlns:tns="http://schemas.xmlsoap.org/soap/envelope/"
    targetNamespace="http://schemas.xmlsoap.org/soap/envelope/">
    <!-- SOAP envelope, header, and body -->
    <element name="Envelope" type="tns:Envelope"/>
    <complexType name="Envelope">
        <element ref="tns:Header" minOccurs="0"/>
        <element ref="tns:Body" minOccurs="1"/>
        <any minOccurs="0" maxOccurs="*"/>
        <anyAttribute/>
    </complexType>
    <element name="Header" type="tns:Header"/>
    <complexType name="Header">
        <any minOccurs="0" maxOccurs="*"/>
        <anyAttribute/>
    </complexType>
    <element name="Body" type="tns:Body"/>
    <complexType name="Body">
        <any minOccurs="0" maxOccurs="*"/>
        <anyAttribute/>
    </complexType>
    <!-- Global Attributes.  The following attributes are
         intended to be usable via qualified attribute names on
         any complex type referencing them.
    -->
    <attribute name="mustUnderstand" default="0">
        <simpleType base="boolean">
            <pattern value="0|1"/>
        </simpleType>
    </attribute>
    <attribute name="actor" type="uri-reference"/>
    <!-- 'encodingStyle' indicates any canonicalization
     conventions followed in the contents of the containing
     element.  For example, the value
     'http://schemas.xmlsoap.org/soap/encoding/' indicates
     the pattern described in SOAP specification.
    -->
<simpleType name="encodingStyle" base="uri-reference"
    derivedBy="list"/>
    <attributeGroup name="encodingStyle">
        <attribute name="encodingStyle"
            type="tns:encodingStyle"/>
    </attributeGroup>
```

```
    <!-- SOAP fault reporting structure -->
    <complexType name="Fault" final="extension">
        <element name="faultcode" type="qname"/>
        <element name="faultstring" type="string"/>
        <element name="faultactor" type="uri-reference"
            minOccurs="0"/>
        <element name="detail" type="tns:detail" minOccurs="0"/>
    </complexType>
    <complexType name="detail">
        <any minOccurs="0" maxOccurs="*"/>
        <anyAttribute/>
    </complexType>
</schema>
```

A SOAP request including the payload defined in the schema would look like this:

```
POST /Order HTTP/1.1
Host: www.northwindtraders.com
Content-Type: text/xml
Content-Length: nnnn
SOAPAction: "urn:northwindtraders.com:PO#UpdatePO"

<SOAP-ENV:Envelope
xmlns:xsi="http://www.w3.org/1999/XMLSchema/instance"
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope"
   xsi:schemaLocation=
      "http://www.northwindtraders.com/schemas/NPOSchema.xsd">
   <SOAP-ENV:Body xsi:type="NorthwindBody">
      <UpdatePO>
         <orderID>0</orderID>
         <customerNumber>999</customerNumber>
         <item>89</item>
         <quantity>3000</quantity>
         <return>0</return>
      </UpdatePO>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

NOTE Notice that the default *Body* declaration is overridden by using an *xsi:type* attribute associating *NorthwindBody* with the *Body* element. A schema that defines *NorthwindBody* and the additional elements, such as *UpdatePO*, will be shown in the section "A Schema for the Body Content of the SOAP Message" later in this chapter.

Because the *Body* element contains the *any* element in the SOAP schema, you could also have written SOAP body as follows:

```
<SOAP-ENV:Envelope
   xmlns:xsi="http://www.w3.org/1999/XMLSchema/instance"
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope">
   <SOAP-ENV:Body xsi:type="NorthwindBody">
      <m:UpdatePO xmlns:m=
         "http://www.northwindtraders.com/schemas/NPOSchema.xsd">
         <orderID>0</orderID>
         <customerNumber>999</customerNumber>
         <item>89</item>
         <quantity>3000</quantity>
         <return>0</return>
      </m:UpdatePO>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

As you can see, the payload of a SOAP request is an XML document that contains the parameter values of the method. The HTTP header of this package has identified the *UpdatePO* method of the *Order* object as the recipient of this method call. The top-level element of the method call must have the same name as the method identified in *SOAPAction*.

The elements contained within the top element are the parameters for the method. The preceding example contains four parameters: *orderID*, *customerNumber*, *item*, and *quantity*. In Microsoft Visual Basic, this method could be written as follows:

```
Public Sub UpdatePO(byval orderID as Integer, _
    byval customerNumber as Integer, _
    byval item as Integer, _
    byval quantity as Integer, _
    byref return as Integer)
```

In Java, this method would look like this:

```
public class UpdatePO {public int orderID;
    public int customerNumber;
    public int item;
    public int quantity;
    public int return;}
```

When you are building the request, you will include one element for each *in* or *in/out* parameter. This technique of associating one element with each parameter is also known as the *element-normal form (ENF)*. The name of each element is the name of the parameter the element is associated with.

The request can also contain a *header* element that includes additional information. There are no predefined elements in the *header* element—you can include any element you want, as long as it is either prefixed by a namespace or the header type is overridden using *xsi:type* and a type defined in a schema.

We can add a *header* element to our payload example as shown here:

```
<SOAP-ENV:Envelope
   xmlns:xsi="http://www.w3.org/1999/XMLSchema/instance"
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope"
   xsi:schemaLocation=
      "http://www.northwindtraders.com/schemas/NPOSchema.xsd">
   <SOAP-ENV:Header xsi:type="NorthwindHeader">
      <GUID>
      10000000-0000-abcd-0000-000000000001
      </GUID>
   </SOAP-ENV:Header>
   <SOAP-ENV:Body xsi:type="NorthwindBody">
      <UpdatePO>
         <orderID>0</orderID>
         <customerNumber>999</customerNumber>
         <item>89</item>
         <quantity>3000</quantity>
         <return>0</return>
      </UpdatePO>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Because the *any* element is used in the *header* element in the SOAP schema, we could also rewrite the SOAP header as follows:

```
<SOAP-ENV:Header>
      <COM:GUID xmlns:COM="http://comobject.Northwindtraders.com">
         10000000-0000-abcd-0000-000000000001
      </COM:GUID>
</SOAP-ENV:Header>
```

In this case, we have created an element named *GUID*. It will be up to the receiving application to interpret this header, but it's likely that it will be used to instantiate the correct COM object. These additional elements can be considered processing instructions.

You can include a predefined attribute named *mustUnderstand* as an attribute of a header element. The *mustUnderstand* attribute can be used to indicate whether the header information is essential for processing of the information. If the header

information is essential, *mustUnderstand* should be set to *true*. If the receiving element cannot recognize the processing instruction and *mustUnderstand* is set to *1*, the message must be rejected. Thus, we could have the following *header* element:

```
<SOAP-ENV:Header xsi:type="Transaction">
      <transactionID mustUnderstand="1">
         10000000
      </transactionID >
</SOAP-ENV:Header>
```

In this case, we are creating an element named *TransactionID*. This element must be understood by the receiving application, or the message will be rejected.

Sending Messages Using M-POST

You can restrict messages coming through a firewall or a proxy server by using the M-POST method instead of POST. M-POST is a new HTTP method defined using the HTTP Extension Framework located at *http://www.w3.org/Protocols/HTTP/ietf-ext-wg*. This method is used when you are including mandatory information in the HTTP header, just as you used the *mustUnderstand* attribute in the SOAP *header* element.

As we mentioned, SOAP supports both POST and M-POST requests. A client first makes a SOAP request using M-POST. If the request fails and either a 501 status code (*Not Implemented* status message) or a 510 status code (*Not Extended* status message) returns, the client should retry the request using the POST method. If the client fails the request again and a 405 status code (*Method Not Allowed* status message) returns, the client should fail the request. If the returning status code is 200, the message has been received successfully. Firewalls can force a client to use the M-POST method to submit SOAP requests by blocking regular POSTs of the *text/xml-SOAP* content type.

If you use M-POST, you must use a mandatory extension declaration that refers to a namespace in the *Envelope* element declaration. The namespace prefix must precede the mandatory headers. The following example illustrates how to use M-POST and the mandatory headers:

```
M-POST /Order HTTP/1.1
Host: www.northwindtraders.com
Content-Type: text/xml
Content-Length: nnnn
Man: "http://schemas.xmlsoap.org/soap/envelope; ns=49"
49-SOAPAction: "urn:northwindtraders.com:PO#UpdatePO"
```

The *Man* header maps the URI *http://schemas.xmlsoap.org/soap/envelope* to the header prefix *49*. Any header that has a prefix of *49* will be associated with this URI and will therefore be a mandatory header. In this case, the *SOAPAction* will be associated with the URI and is a mandatory header.

> **NOTE** If you use M-POST and do not have any mandatory header elements, an error will occur, resulting in either a 501 or 510 status code.

The Payload for a SOAP Response

Just as our sample SOAP request message contained child elements for all the *in* and *in/out* parameters of the method, the SOAP response will contain child elements for each *out* and *in/out* parameter. Let's say you have the following Visual Basic function:

```
Public Function UpdatePO(byref OrderID as Integer, _
    byval CustomerNumber as Integer, _
    byval Item as Integer, _
    byval Quantity as Integer) as Integer
```

In this case, the server would set the *orderID* variable to some value and return the value to the client. Because the *orderID* parameter is *byref*, it is an *in/out* parameter. *UpdatePO* is now a function, and it will return a value (a Boolean value, in this case). The return value of the function can be considered an *out only* parameter.

For this *UpdatePO* function, the request payload containing all the *in* and *in/out* parameters might look like this:

```
<SOAP-ENV:Envelope
    xmlns:xsi="http://www.w3.org/1999/XMLSchema/instance"
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope"
    xsi:schemaLocation=
      "http://www.northwindtraders.com/schemas/NPOSchema.xsd">
   <SOAP-ENV:Body xsi:type="NorthwindBody">
      <UpdatePO>
      <orderID>0</orderID>
      <customerNumber>999</customerNumber>
      <item>89</item>
      <quantity>3000</quantity>
      </UpdatePO>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The response package including the payload that contains all the *out* and *in/out* parameters would look like this:

```
HTTP/1.1 200 OK
Content-Type: text/xml
Content-Length: nnnn

<SOAP-ENV:Envelope
   xmlns:xsi="http://www.w3.org/1999/XMLSchema/instance"
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope"
   xsi:schemaLocation=
      "http://www.northwindtraders.com/schemas/NPOSchema.xsd">
   <SOAP-ENV:Body xsi:type="NorthwindBody">
      <UpdatePO>
         <orderID>09877</orderID>
         <return>0</return>
      </UpdatePO>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Notice that the SOAP response message doesn't have the *SOAPAction* header field. This header field is required only in the SOAP request message. During request processing, a SOAP package will be passed to a SOAP application on the server that handles SOAP requests. This SOAP application will in turn pass on the request to the appropriate method.

Sometimes a request cannot be processed properly and errors might occur. Errors can be handled in several ways, depending on where the error occurs. If an error occurs during the transport of the package to the method, the error is usually handled by returning status codes other than 200. An example of this situation is when there is a problem getting the package through a firewall or the specified host does not exist or is down.

Once the information has been passed to the method, it is possible that the application handling the request will experience an error. In this case, you can return a custom HTTP code, use one of the predefined HTTP codes, use an element, such as the *return* element in the preceding example, or return a SOAP package that contains a *Fault* element to pass back the error information. Let's look at how to use the *Fault* element to report errors.

The *Fault* Element

A *Fault* element can contain four child elements: *faultcode*, *faultstring*, *faultactor*, and *detail*. The fault codes are identified at the URL *http://schemas.xmlsoap.org/soap/envelope*. The currently available code values are shown in the following table.

CURRENTLY AVAILABLE FAULT CODE VALUES

Name	*Meaning*
VersionMismatch	The call used an invalid namespace.
MustUnderstand	The receiver did not understand an XML element that was received containing an element tagged with *mustUnderstand="true"*.
Client	These are a class of errors that were caused by improper information in the actual SOAP message. For example, a new order could be missing a required value such as the item number or amount. These errors represent a problem with the actual message content and indicate that the message should not be resent without change. A class of client errors can be created using the dot (.) operator. For example, you could have *Client.InvalidPartNumber*, *Client.InvalidQuantity*, and so on.
Server	These errors are related to problems with the server and usually do not represent problems with the actual SOAP message. These messages might be resent at a later time to the server. A class of server errors can be created using the dot (.) operator.

The *faultstring* element is a string. It is not used by applications; it is used only as a message to users. This element is required. The *faultactor* element can provide information about which element in the message path caused the fault to occur. The *faultactor* will be a URI that identifies the source. If the fault occurs in an application that is not the ultimate destination of the message, the *faultactor* element must be included. If the fault occurs in the ultimate destination, the *faultactor* is not required but may be included. The *detail* element is required if the contents of the SOAP *Body* element couldn't be processed. It provides application-specific error information related to the *Body* element. You cannot include error information related to the header in the *detail* element. Any child elements of the *detail* element must be associated with a namespace.

A return package with a *Fault* element will look like this:

```
HTTP/1.1 200 OK
Content-Type: text/xml
Content-Length: nnnn

<SOAP-ENV:Envelope
   xmlns:xsi="http://www.w3.org/1999/XMLSchema/instance"
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope"
```

(continued)

```
   xsi:schemaLocation=
      "http://www.northwindtraders.com/schemas/NPOSchema.xsd">
      <SOAP-ENV:Fault>
         <SOAP-ENV:faultcode>200</SOAP-ENV:faultcode>
         <SOAP-ENV:faultstring>Must Understand Error
         </SOAP-ENV:faultstring>
         <SOAP-ENV:detail xsi:type="Fault">
            <errorMessage>
               Object not installed on server.
            </errorMessage>
         </SOAP-ENV:detail>
      </SOAP-ENV:Fault >
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Using the *faultstring* element, you can pass information back to the client that describes the exact error. The *faultstring* element can handle a wide range of errors.

A Schema for the Body Content of the SOAP Message

As you can see, we have not yet defined the NPO schema located at *http://www.northwindtraders.com/schemas/NPOSchema.xsd*. This schema can be defined as follows:

```
<xsd:schema xmlns:xsd="http://www.w3.org/1999/XMLSchema"
   targetNamespace="http://schemas.xmlsoap.org/soap/envelope"
   xmlns:SOAP="http://schemas.xmlsoap.org/soap/envelope">
   <xsd:complexType name="NorthwindHeader">
      <xsd:element name="GUID" type="string"/>
   </xsd:complexType>

   <xsd:complexType name="NorthwindBody">
      <xsd:element name="UpdatePO">
         <xsd:complexType>
            <element name="orderID" type="integer"/>
            <element name="customerNumber" type="integer"/>
            <element name="item" type="double"/>
            <element name="quantity" type="double"/>
         </xsd:complexType>
      </xsd:element>
   </xsd:complexType>
</xsd:schema>
```

This schema creates two elements: *NorthwindBody* and *NorthwindHeader*. Using the *xsi:type* attribute, we can extend the SOAP *body* element with the *NorthwindBody* complex type and extend the *Header* element with *NorthwindHeader* complex type. You can then create the following SOAP document:

```
<SOAP-ENV:Envelope
   xmlns:xsi="http://www.w3.org/1999/XMLSchema/instance"
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope"
   xsi:schemaLocation=
      "http://www.northwindtraders.com/schemas/NPOSchema.xsd">
<SOAP-ENV:Header xsi:type="NorthwindHeader">
      <COM:GUID xmlns:COM="http://comobject.northwindtraders.com">
         10000000-0000-abcd-0000-000000000001
      </COM:GUID>
   </SOAP-ENV:Header>

   <SOAP-ENV:Body xsi:type="NorthwindBody">
      <UpdatePO>
         <orderID>0</orderID>
         <customerNumber>999</customerNumber>
         <item>89</item>
         <quantity>3000</quantity>
      </UpdatePO>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

SOAP ENCODING

The SOAP encoding style provides a means to define data types similar to what is found in most programming languages, including types and arrays. The specification for encoding can be found at *http://schemas.xmlsoap.org/soap/encoding/*. SOAP defines simple and complex data types just as the schema standard does. The simple type elements are the same as those defined in the second schema standard. The complex type elements include those defined in the first SOAP standard and a special way of defining arrays. Structures follow the definitions of the complex type. For example, we could have the following structure:

```
<e:Customer>
   <CName>John Jones</CName>
   <Address>100 Main Street</Address>
   <ID>4</ID>
</e:Customer>
```

This structure would be defined as follows:

```
<element name=Customer>
   <element name="CName" type="xsd:string"/>
   <element name="Address" type="xsd:string"/>
   <element name="ID" type="xsd:string"/>
</element>
```

Chapter 11

The XML Document Object Model

The XML Document Object Model (DOM) is a platform-neutral and language-neutral interface that allows developers to create applications and scripts to access and update the content, style, and structure of XML documents. The XML DOM is based on the W3C DOM, which is a recommended specification that has been released by the DOM Working Group (WG) in several stages. The DOM Level 1 specification introduces the features that can be used to manipulate the content and structure of HTML and XML documents. The W3C is currently working on the DOM Level 2 specification. This specification extends DOM Level 1 with many new features. Microsoft Internet Explorer 5 fully supports the W3C Level 1 DOM specification.

> **NOTE** For more information about the features included in each DOM level, visit the W3C Web site at *http://www.w3.org/dom*.

In addition to the XML DOM support included with Internet Explorer 5, Microsoft XML parser version 2.6 and version 3.0 have been released that support several extensions of the XML DOM beyond the current W3C specification, including BizTalk

schemas, XPath, and XSL Transformations (XSLT). We will begin this chapter with a discussion of the implementation of the XML DOM in Internet Explorer 5 and end with a discussion of the new XML parser and the additional functionality that it adds.

INTERNET EXPLORER 5'S IMPLEMENTATION OF THE XML DOM

The implementation of the XML DOM in Internet Explorer 5 consists of a set of objects that can be used to load an XML document. Once the document is loaded, you can parse, navigate, and manipulate the information in the XML document. The DOM can also be used for retrieving information about the document.

There are four main objects included with Internet Explorer 5's implementation of the XML DOM: *XMLDOMDocument*, *XMLDOMNode*, *XMLDOMNodeList*, and *XMLDOMNamedNodeMap*. In addition, sixteen other objects are part of Internet Explorer 5's implementation of the XML DOM. All of these objects have properties and methods that you can use to gather information about the document (including its structure) and to navigate to other object *nodes* within a document tree. A node is a reference to any object that can exist in a document hierarchy. The ability to access different nodes of the document tree is a function that is also available using XPath and XPointer. Twelve different types of nodes are available in the DOM: *element*, *attribute*, *text*, *CDATA section*, *entity reference*, *entity*, *processing instruction*, *comment*, *document*, *document type*, *document fragment*, and *notation*. An interface exists for each of these node types that allows you to gather and manipulate information on the node. The most common node types are the *element*, *attribute*, and *text* nodes.

> **NOTE** Attributes are not actually child elements of any node in the tree, so they have a special programming interface called *IXMLDOMNamedNodeMap*.

The W3C DOM specification defines two types of programming interfaces: fundamental and extended. The fundamental DOM interfaces are required when writing applications that manipulate XML documents. The extended interfaces are not required, but make it easier for developers to write applications. The Internet Explorer 5 DOM implements both the fundamental and extended interfaces. In addition, it provides other interfaces to support Extensible Stylesheet Language (XSL), XSL patterns, namespaces, and data types.

For script developers, the most important object in the Internet Explorer 5's implementation of the XML DOM is the *XMLDOMDocument* object, which allows developers to navigate, query, and modify the content and structure of an XML document. This object implements the *IXMLDOMDocument* interface. We will look at this object first.

XMLDOMDocument Object

To navigate and get a reference to an XML document, you need to use the *XMLDOMDocument* object. Once you actually get a reference to the document, you can begin to work with it. The *XMLDOMDocument* object implements the *IXMLDOMDocument* interface.

Getting a reference to an XML document

Depending on the programming language you are using, you can get a reference to an XML document in several ways.

In Microsoft JScript, you can get a reference as follows:

```
var objXMLdoc = new ActiveXObject("Microsoft.XMLDOM);
objXMLdoc.load("http://www.northwindtraders.com/sales.xml");
```

In VBScript, the code for obtaining a reference appears as follows:

```
Dim objXMLdoc
Set objXMLdoc = CreateObject("Microsoft.XMLDOM")
objXMLdoc.load("http://www.northwindtraders.com/sales.xml")
```

In Microsoft Visual Basic, you should add a reference to Msxml.dll to your project by choosing References from the Project menu, and then choosing Microsoft XML version 2 from the References dialog box. The code to get a reference to an XML document appears as follows:

```
Dim objXMLdoc As DomDocument
Set objXMLdoc = New DomDocument
objXMLdoc.load("http://www.northwindtraders.com/Sales.xml")
```

You could also use the following code without setting the reference, though the above method is preferable:

```
Set objXMLdoc = CreateObject("Microsoft.XMLDOM")
objXMLdoc.load("http://www.northwindtraders.com/Sales.xml")
```

IXMLDOMDocument interface properties and methods

In the above examples, we used the *load* method to get a reference to an actual XML document. The following tables list the properties, methods, and events associated with the *IXMLDOMDocument* interface. Properties and methods that are extensions of the W3C DOM Level 1 specification will be marked with an asterisk (*) throughout this chapter.

> **NOTE** Code samples illustrating how to use the *IXMLDOMDocument* interface will be presented later in this chapter.

IXMLDOMDocument Properties

Name	Description
*async**	Downloads the XML document asynchronously if this property is set to *true* (the default).
attributes	Returns an *XMLDOMNamedNodeMap* object for nodes that can return attributes.
*baseName**	Returns the name of the node with any namespace removed.
childNodes	Returns all children of the current node for nodes that are allowed children.
*dataType**	Sets or returns the data type for an XML document node that uses a schema. For *entity references*, *elements*, and *attributes*, if a data type is specified in the schema it will return the data type as a string. If no value is specified, it returns *null*, and for all other nodes it returns *string*. Attempts to set the *dataType* property for nodes other than *attribute*, *element*, or *entity reference* are ignored.
*definition**	Returns the node that contains the DTD or schema definition for the entity referenced.
doctype	Returns a reference to an *XMLDOMDocumentType* node containing a reference to the DTD or schema.
documentElement	Returns a reference to the outermost document element of an XML document.
firstChild	Returns a reference to the first child of the current node.
implementation	Returns a reference to the *XMLDOMImplementation* object for the document.
lastChild	Returns a reference to the last child node of the current node.
*namespaceURI**	Returns the Uniform Resource Identifier (URI) for the namespace as a string.
nextSibling	Returns a reference to the next sibling node of the current node.
nodeName	Returns the name of the node.
*nodeTypeString**	Returns the node type as a string.
nodeType	Returns the node type as a number.
*nodeTypedValue**	Returns or sets the strongly typed value of the node.

Name	*Description*
nodeValue	Sets or returns the value of the node as text. Returns *attribute* value for *attribute* nodes. Returns the text within the *CDATA* section for *CDATASection* nodes. Returns the comment for *comment* nodes. Returns the processing instruction for *processing instruction* nodes. Returns the text for *text* nodes. For all other nodes, it returns *null* if you try to get the property and raises an error if you try to set the property.
ownerDocument	Returns the root of the document that contains this node.
parentNode	Returns the parent node of the current node for nodes that are allowed to have parents.
*parsed**	Returns *true* if the current node and all of its descendants have been parsed and instantiated.
*parseError**	Returns a reference to the *XMLDOMParseError* object that contains information about the last parsing error.
*prefix**	Returns the element namespace prefix as a string.
*preserveWhiteSpace**	Specifies if white space should be preserved. The default is *false*.
previousSibling	Returns a reference to the previous sibling node of the current node.
*readyState**	Indicates the current state of an XML document.
*resolveExternals**	Resolves the external entities, and the document is resolved against external DTDs, if this is *true*. The default is *false*.
*specified**	Returns *true* if a node value is specified. Returns *false* if a node value is derived from a default value. (This is normally used only with *attribute* nodes.)
*text**	Sets and returns the text content of the current node and all of its descendants.
*url**	Returns the URL of the last successfully loaded XML document or returns *null* if the XML document was built in memory.
*validateOnParse**	The document will validate on parsing when this property is set to *true*, but the parser will only check the document for being well formed if this property is set to *false*. Default is *true*. This property can be set or read.
*xml**	Returns the entire XML content of the current node and all of its descendant nodes.

IXMLDOMDocument Methods

Name	Description
*abort()**	Stops the asynchronous load if the *async* property is set to *true* and the document is loading. Any information that has been downloaded is discarded. If the *readyState* property is equal to *COMPLETED*, calling *abort* has no effect.
appendChild (newChild)	Appends *newChild* to the end of the child nodes list for the currently selected node.
cloneNode (deep)	Creates a *clone* node that is identical to the currently referenced node. If *deep* is set to *true*, all child nodes are also cloned.
createAttribute (name)	Creates an *attribute* node with the specified name.
createCDATASection (text)	Creates a *CDATASection* node containing *text*.
createComment (text)	Creates a *comment* node containing *text*. The comment delimiters (<!-- -->) will be inserted.
createDocumentFragment()	Creates an empty *DocumentFragment* node that is used to build independent sections of the XML document.
createElement (name)	Creates an instance of the specified element.
createEntityReference (name)	Creates an *EntityReference* node called *name*.
*createNode (type, name, namespace)**	Creates any type of node using the specified *type*, *name*, and *namespace* parameters.
createProcessingInstruction (target, data)	Creates a new processing instruction. The *target* parameter provides both the target and the node name. The *data* parameter is the actual instruction.
createTextNode (text)	Creates a *text* node containing the text specified in the *text* parameter.

Name	*Description*
getElementsByTagName (name)	Returns a collection of child elements that have the specified tag name. If the *name* parameter is *, it returns all elements.
hasChildNodes()	Returns *true* if the node has any child nodes.
insertBefore (newNode, beforeNode)	Inserts a new node object called *newNode* into the list of child nodes for the current node to the left of the *beforeNode* or at the end if *beforeNode* is left out.
*load (url)**	Loads an XML document from the specified URL.
*loadXML (string)**	Loads a string that contains well-formed XML.
*nodeFromID (value)**	Returns the node object that has an ID attribute matching the supplied value.
removeChild (node)	Removes the child node from the current node and returns it.
replaceChild (newNode, oldNode)	Replaces the child node *oldNode* with the node *newNode*.
*save (destination)**	Saves the file to the specified destination.
*selectNodes (pattern)**	Returns a node list object containing matching nodes. The *pattern* parameter is a string containing an XSL pattern.
*selectSingleNode (pattern)**	Returns the first node object matching the pattern of a string containing XSL.
*transformNode (stylesheet)**	Processes the node and its children using XSL pattern matching. The *stylesheet* parameter must be either an *XMLDOMDocument* node object or a node object in the XSL namespace.
transformNodeToObject (stylesheet, outputobject)	Transforms the node according to the XSL document and places the transformed document into the *outputobject* parameter.

IXMLDOMDocument Events

Name	Description
*ondataavailable**	Occurs whenever data becomes available. When the *async* property is *true*, this event fires several times as data comes in. Using the *readyState* property, you can obtain information on the incoming data, including when all of the data has been downloaded.
*onreadystatechange**	Fires whenever the *readyState* property changes.
*ontransformnode**	Fires when a node is transformed using the *TransformNode* method of the node object.

XMLDOMNode Object

The *XMLDOMNode* object implements the *IXMLDOMNode* interface. This interface contains the following properties: *attributes*, *baseName*, *childNodes*, *dataType*, *definition*, *firstChild*, *lastChild*, *namespaceURI*, *nextSibling*, *nodeName*, *nodeTypeString*, *nodeType*, *nodeTypedValue*, *nodeValue*, *ownerDocument*, *parentNode*, *parsed*, *prefix*, *previousSibling*, *specified*, *text*, and *xml*. The methods associated with *IXMLDOMNode* are *appendChild*, *clonenode*, *hasChildNodes*, *insertBefore*, *removeChild*, *replaceChild*, *selectNodes*, *selectSingleNode*, *transformNode*, and *transformNodeToObject*. There are no events associated with the *IXMLDOMNode* interface.

Looking at these properties and methods, you can see that they're all included in the *IXMLDOMDocument* interface and have been defined above. The same methods exist in both interfaces because *IXMLDOMNode* is used as the base interface for building all W3C DOM objects except for *IXMLDOMImplementation*, *IXMLDOMNodeList*, and *IXMLDOMNamedNodeMap*, as illustrated in Figure 11-1.

Besides these interfaces, Internet Explorer 5 has three additional interfaces: *IXTLRuntime*, *IXMLDOMParseError*, and *IXMLHTTPRequest*. You can see all the interfaces, including those specific to Internet Explorer 5, in Figure 11-2.

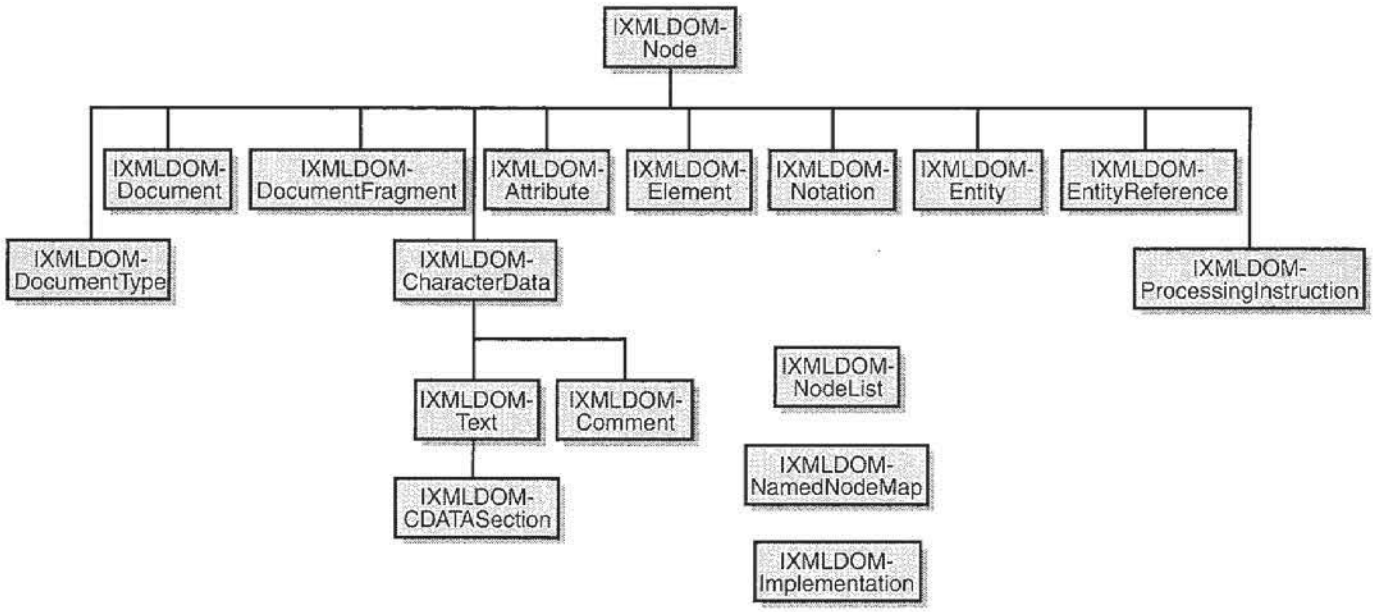

Figure 11-1. *The relationship between the W3C DOM object interfaces.*

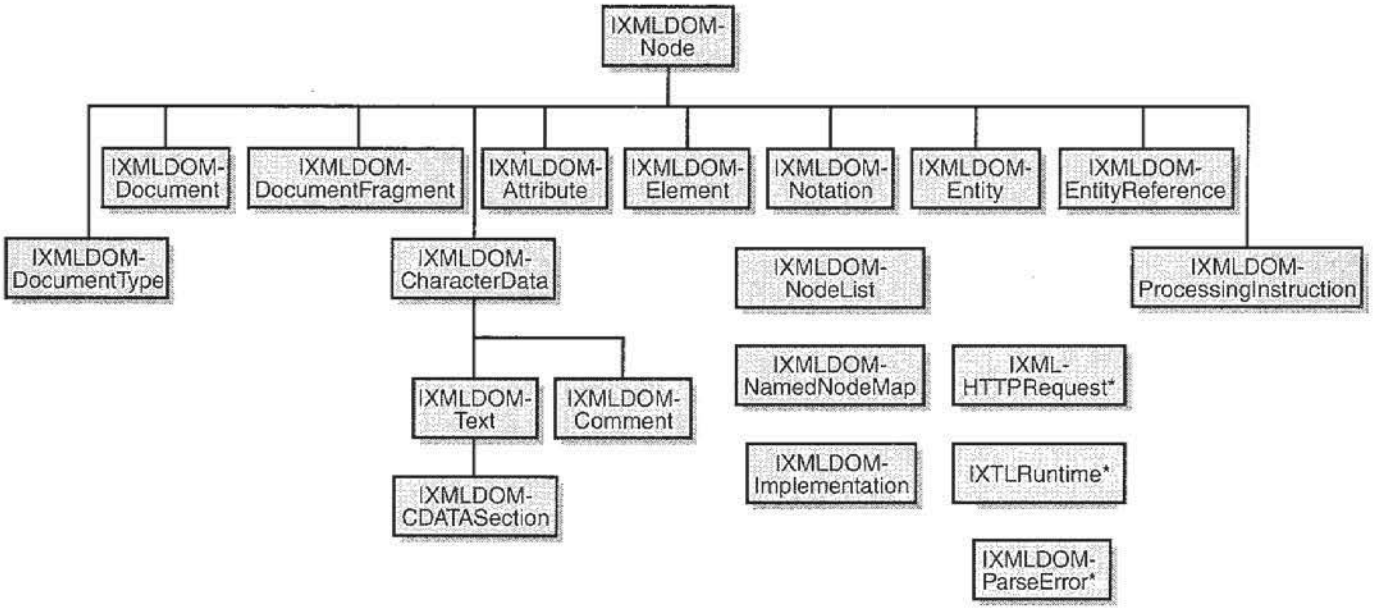

Figure 11-2. *Internet Explorer 5 DOM interfaces.*

Let's look at how to code some of the methods and properties that belong to the *IXMLDOMNode* interface.

> **NOTE** The following example will show you how to use some of the properties and methods of the *IXMLDOMNode* interface using Visual Basic. If you have access to Visual Basic, I highly recommend that you follow the examples. If you don't have Visual Basic, you can easily convert this example to a script sample. This example will print out the values of various properties to the Immediate Window.

To create the sample application, follow these steps:

1. Open Visual Basic, create a standard EXE project, and change the name of the default form to *frmDOMTest*.
2. Choose References from the Project menu, and add a reference to *Microsoft XML, version 2.0*.
3. Add a command button to *frmDOMTest* called *cmdNode* with a caption *Nodes*.
4. Add the following code to the click event handler of *cmdNode*:

```
Private Sub cmdNode_Click()
Dim objXMLDoc As DOMDocument
'Create a node object that is a reference to the root object.
Dim objRoot As IXMLDOMNode
'Create a node object that can be used to create a new node.
Dim objNewNode As IXMLDOMNode
Set objXMLDoc = New DOMDocument
'Turn off asynchronous load as we do not need it for this example.
objXMLDoc.async = False
'Open the file shown below.
objXMLDoc.Load ("c:\Books.xml")
'The documentElement will return the root element.
Set objRoot = objXMLDoc.documentElement
'Begin printing out properties for the root.
Debug.Print "XML for the root: " & vbCrLf & objRoot.xml
Debug.Print "BaseName for the root: " & objRoot.baseName
Debug.Print "Namespace prefix for the root: " & objRoot.prefix
Debug.Print "DataType for the root: " & objRoot.dataType
'We will begin to walk through the document starting at the first
'child.
Debug.Print "First Child XML for the root: " & vbCrLf & _
                objRoot.firstChild.xml
'We will get the next child, which is two elements down from
'the root.
```

```
Debug.Print "First Child of Child XML for the root: " & _
                objRoot.firstChild.firstChild.xml
'Nextsibling will return a node on the same level, in this case
'the same level as the second element down from the root.
Debug.Print "Second Child of Child XML for the root: " & _
                objRoot.firstChild.firstChild.nextSibling.xml
Debug.Print "Third Child of Child XML for the root: " & _
                objRoot.firstChild.firstChild.nextSibling. _
                nextSibling.xml
Debug.Print "Fourth Child of Child XML for the root: " & _
                objRoot.firstChild.firstChild.nextSibling. _
                nextSibling.nextSibling.xml
Debug.Print "Namespace URI for the root: " & _
                objRoot.namespaceURI
Debug.Print "Nodename for the root: " & objRoot.nodeName
Debug.Print "NodeType for the root: " & objRoot.nodeType
Debug.Print "NodeType String for the root: " & _
                objRoot.nodeTypeString
Debug.Print "NodeValue for the root: " & objRoot.nodeValue
Debug.Print "parentNode for the root: " & _
                objRoot.parentNode.xml
'Using XSL to get a single node
Debug.Print "XSL selecting first child node of the item node: " & _
     vbCrLf & objRoot.selectSingleNode("item/*").xml
Set objNewNode = objXMLDoc.createNode(1, "test", "")
objRoot.appendChild objNewNode
Debug.Print "Root XML after appending: " & vbCrLf & objRoot.xml
Set objNewNode = Nothing
Set objRoot = Nothing
Set objXMLDoc = Nothing

End Sub
```

Notice that we first get a reference to the document object. Using this reference, we can get a reference to the *XMLDOMNode* object. Then we start navigating the nodes in the XML document. Finally, we create a node named *test* and append it as a child node to the *root* node. To test this application, let's create an XML document called Books.xml in the C:\ directory with the following XML:

```
<?xml version="1.0" ?>
<northwind:BOOKS xmlns:northwind="www.northwindtraders.com/PO">
   <item>
      <title>Number, the Language of Science</title>
```

(continued)

```
      <author>Danzig</author>
      <price>5.95</price>
      <quantity>3</quantity>
   </item>
</northwind:BOOKS>
```

When you run the program and click the *Nodes* button, the results are as follows:

```
XML for the root:
<northwind:BOOKS xmlns:acme="www.northwindtraders.com/PO">
   <item>
      <title>Number, the Language of Science</title>
      <author>Danzig</author>
      <price>5.95</price>
      <quantity>3</quantity>
   </item>
</northwind:BOOKS>
BaseName for the root:BOOKS
Namespace prefix for the root:northwind
DataType for the root:
First Child XML for the root:
<item>
   <title>Number, the Language of Science</title>
   <author>Danzig</author>
   <price>5.95</price>
   <quantity>3</quantity>
</item>
First Child of Child XML for the root: <title>Number, the Language
   of Science</title>
Second Child of Child XML for the root: <author>Danzig</author>
Third Child of Child XML for the root: <price>5.95</price>
Fourth Child of Child XML for the root: <quantity>3</quantity>
Namespace URI for the root: www.northwindtraders.com/PO
Nodename for the root: northwind:BOOKS
NodeType for the root: 1
NodeType String for the root: element
NodeValue for the root:
parentNode for the root: <?xml version="1.0"?>
<northwind:BOOKS xmlns:northwind="www.northwindtraders.com/PO">
   <item>
      <title>Number, the Language of Science</title>
      <author>Danzig</author>
      <price>5.95</price>
      <quantity>3</quantity>
```

```
   </item>
</northwind:BOOKS>

XSL selecting first child node of ITEM:
<title>Number, the Language of Science</title>
Root XML after appending:
<northwind:BOOKS xmlns:northwind="www.northwindtraders.com/PO">
   <item>
      <title>Number, the Language of Science</title>
      <author>Danzig</author>
      <price>5.95</price>
      <quantity>3</quantity>
   </item>
   <test/></northwind:BOOKS>
```

Notice that the *test* element was inserted as the last child of the root, which is what we would have expected. Once this element is inserted, you can add text values or make other changes. All the sample programs discussed in this chapter, including the Visual Basic sample program and Books.xml, are available on the companion CD.

> **NOTE** Though we have not discussed XSL yet, we used XSL to get a single node in the previous application. XSL defines the location of elements using the XPath syntax. We'll discuss XSL in detail in Chapter 12. In this chapter, we will use the XPath syntax with the *selectSingleNode* method.

Several methods and properties belonging to the document object will return other objects in the hierarchy, such as *selectNodes* or *attributes*. We'll discuss these methods and properties while examining other object interfaces in the XML DOM.

XMLDOMNodeList Object

The *XMLDOMNodeList* object is a collection of node objects. It is primarily used to iterate through the element nodes. This object implements the *IXMLDOMNodeList* interface. The *IXMLDOMNodeList* interface reflects the current state of the nodes in the document, so a change in the nodes will be immediately reflected in the object. The property and methods of *IXMLDOMNodeList* are as follows:

IXMLDOMNodeList Property

Name	*Description*
length	Returns the number of nodes that are contained in the node list.

IXMLDOMNodeList Methods

Name	Description
item (index)	Returns the node located at position *index* in the node list. The first node is indexed as 0.
*nextNode()**	Returns the next node object in the node list. If there are no more nodes, it returns *null*.
*reset()**	Resets the pointer so that it points before the first node element.

For an example of the *IXMLDOMNodeList* interface, you can add an attribute to the XML document and another command button to the *frmDOMTest* form. To do so, follow these steps:

1. Open the XML document Books.xml and change the *title* element to the following:

```
<title language="English">Number, the Language of Science
</title>
```

2. Add another command button to the *frmDOMTest* form called *cmdNodeList* with the caption *NodeList*.

3. Add the following code to the click event handler of the *cmdNodeList* button:

```
Private Sub cmdNodeList_Click()
    Dim objNodeList As IXMLDOMNodeList
    Dim objXMLDoc As DOMDocument
    Set objXMLDoc = New DOMDocument
    objXMLDoc.async = False
    objXMLDoc.Load ("c:\Books.xml")
    Set objNodeList = _
        objXMLDoc.documentElement.firstChild.childNodes
    Debug.Print "The second item's basename is: " & _
        objNodeList.Item(2).baseName
    Debug.Print "The number of nodes are: " & objNodeList.length
    Debug.Print "The first node xml is: " & vbCrLf & _
        objNodeList.nextNode.xml
    Debug.Print "The second node xml is: " & _
        objNodeList.nextNode.xml
    Debug.Print "The third node xml is: " & _
```

```
        objNodeList.nextNode.xml
        objNodeList.Reset
    Debug.Print "After reset, the first node xml is: " & _
        vbCrLf & objNodeList.nextNode.xml
    Dim intNodeCounter As Integer
    For intNodeCounter = 0 To objNodeList.length - 1
        Debug.Print "The " & "xml of node" & _
        Str(intNodeCounter + 1) & " is: " & vbCrLf & _
        objNodeList.Item(intNodeCounter).xml
    Next
    Set objNodeList = Nothing
    Set objXMLDoc = Nothing
End Sub
```

Notice that, once again, we first get a reference to the document object. Once we have this reference, we can get a reference to the *IXMLDOMNodeList* interface. Then we use the *item*, *nextNode*, and *reset* methods of the *IXMLDOMNodeList* interface to navigate the document. Last, we print all the nodes in the collection with a loop. When you run this updated application and click the *NodeList* button, the results are as follows:

```
The second item's basename is: price
The number of nodes are: 4
The first node xml is:
<title language="English">Number, the Language of Science</title>
The second node xml is: <author>Danzig</author>
The third node xml is: <price>5.95</price>
After reset, the first node xml is:
<title language="English">Number, the Language of Science</title>
The xml of node 1 is:
<title language="English">Number, the Language of Science</title>
The xml of node 2 is:
<author>Danzig</author>
The xml of node 3 is:
<price>5.95</price>
The xml of node 4 is:
<quantity>3</quantity>
```

Notice that the *attribute* node was not included in the results. We will need to use the *IXMLDOMNamedNodeMap* interface to get a reference to *attribute* nodes.

XMLDOMNamedNodeMap Object

The *XMLDOMNamedNodeMap* object implements the *IXMLDOMNamedNodeMap* interface. This interface is similar to the *IXMLDOMNodeList* interface except that it allows you to iterate through attributes and *namespace* nodes. The *IXMLDOMNamedNodeMap* interface has the same *length* property as the *IXMLDOMNodeList* interface. *IXMLDOMNamedNodeMap* also has the same *item*, *nextNode*, and *reset* methods as the *IXMLDOMNodeList* interface. The additional methods that are associated with the *IXMLDOMNamedNodeMap* are as follows:

ADDITIONAL *IXMLDOMNAMEDNODEMAP* METHODS

Name	*Description*
getNamedItem (name)	Retrieves the node object with the specified name. This method is usually used to retrieve an attribute from an element.
*getQualifiedItem (baseName, namespace)**	Returns the node object with the specified *baseName* and namespace.
removeNamedItem (name)	Removes the node object that has the specified name from the named node map. This method is usually used to remove an attribute.
*removeQualifiedItem (baseName, namespace)**	Removes the node object with the specified *baseName* and *namespace*. This method is usually used to remove attributes from the collection.
setNamedItem (newNode)	Inserts a new node into the collection. If a node with the same name as the *newNode* already exists, it's replaced.

To illustrate how to use the methods and properties of *IXMLDOMNamedNodeMap*, add another command button to the *frmDOMTest* form with the name *cmdNamedNodeMap* and the caption *NamedNodeMap*. Add the following code to the click event handler of the *cmdNamedNodeMap* button:

```
Private Sub cmdNamedNodeMap_Click()
    Dim objNamedNodeMap As IXMLDOMNamedNodeMap
    Dim objXMLDoc As DOMDocument

    Set objXMLDoc = New DOMDocument
    objXMLDoc.async = False
    objXMLDoc.Load ("c:\Books.xml")

    Set objNamedNodeMap = objXMLDoc.documentElement.Attributes
    Debug.Print _
         "The root's first attribute node's basename is: " & _
         objNamedNodeMap.Item(0).baseName
    Debug.Print "The number of root's attribute nodes is: " & _
         objNamedNodeMap.length
    Debug.Print "The first node xml is: " & _
         objNamedNodeMap.nextNode.xml

    Set objNamedNodeMap = _
       objXMLDoc.documentElement.firstChild.firstChild.Attributes
    Debug.Print _
         "The title element's attribute node's" & _
         " basename is: " & objNamedNodeMap.Item(0).baseName
    Debug.Print "The number of the title element's " & _
         "attribute nodes is: " & objNamedNodeMap.length
    Set objNamedNodeMap = Nothing
    Set objXMLDoc = Nothing

End Sub
```

Once again, to move through the XML document you will begin by getting a reference to a document object. This time, you will use the *attributes* property of the document object to get a reference to the *IXMLDOMNamedNodeMap* interface. When you run this example and click the *NamedNodeMap* button, the results are as follows:

```
The root's first attribute node's basename is: northwind
The number of the root's attribute nodes is: 1
The first node xml is:xmlns: northwind="www.northwindtraders.com/PO"
The title element's attribute node's baseName is: language
The number of the title element's attribute nodes is: 1
```

Thus, using the *IXMLDOMDocument* interface's *attributes* property and the *IXMLDOMNamedNodeMap* interface we are able to get information about the *namespace* and *attribute* nodes.

XMLDOMDocumentType Object

The *XMLDOMDocumentType* object implements *IXMLDOMDocumentType* interface. The *doctype* property of the *IXMLDOMDocument* interface identifies the document's *IXMLDOMDocumentType* interface. The *IXMLDOMDocumentType* interface gets information on the document type declaration in the XML document. This interface also extends the *IXMLDOMNode* interface, so it has all the properties and methods of the *IXMLDOMNode* interface. The *IXMLDOMDocumentType* interface also implements the following extended properties:

ADDITIONAL *IXMLDOMDOCUMENTTYPE* PROPERTIES

Name	*Description*
entities	Returns a node list containing references to the entity objects declared in the DTD
name	Returns the name of the document type for the document
notations	Returns a node list containing references to the notation objects in the DTD

Now that the *IXMLDOMDocumentType* interface contains information associated with the DTD, let's create a DTD named Books.dtd for the document using the following text:

```
<!ELEMENT northwind:BOOKS  (item)>
<!ATTLIST northwind:BOOKS  xmlns:northwind CDATA  #FIXED
                           "www.northwindtraders.com/PO">
<!ENTITY % ItemElements "(title, author, price, quantity)">
<!ENTITY copyright "&#xA9;">
<!ELEMENT item  %ItemElements;>
<!ELEMENT title (#PCDATA)>
<!ATTLIST title language CDATA  #REQUIRED>
<!ELEMENT author (#PCDATA)>
<!ELEMENT price (#PCDATA)>
<!ELEMENT quantity (#PCDATA)>
```

Notice that we declared a general entity called *copyright* in the above DTD, thus we need to reference this entity in the Books.xml document. We also need to add a line of code to the XML document so that it will reference the DTD:

```
<?xml version="1.0" ?>
<!DOCTYPE northwind:BOOKS SYSTEM "c:\Books.dtd">
<northwind:BOOKS xmlns:northwind="www.northwindtraders.com/PO">
   <item>
      <title language="English">Number, the Language of Science
```

```
      &copyright;
      </title>
      <author>Danzig</author>
      <price>5.95</price>
      <quantity>3</quantity>
   </item>
</northwind:BOOKS>
```

> **NOTE** Remember that the DTD has no ability to resolve namespaces. Thus, you must declare the elements that use the namespace with the namespace prefix and define an attribute for the namespace. The XML document, though, can resolve the namespace information.

Now let's take a look at how to use the properties of the *IXMLDOMDocumentType* interface in our example application. First, add another command button to the *frmDOMTest* form with the name *cmdDocumentType* and the caption *Document Type*. Then add the following code to the click event of this button:

```
Private Sub cmdDocumentType_Click()
    Dim objDocumentType As IXMLDOMDocumentType
    Dim objXMLDoc As DOMDocument

    Set objXMLDoc = New DOMDocument
    objXMLDoc.async = False
    objXMLDoc.Load ("c:\Books.xml")

    Set objDocumentType = objXMLDoc.doctype
    Debug.Print objDocumentType.Name
    Debug.Print objDocumentType.xml
    Debug.Print objDocumentType.entities.length
    Debug.Print objDocumentType.entities.Item(0).xml
End Sub
```

When you run this example and click the *DocumentType* button, you'll see the following output:

```
northwind:BOOKS
<!DOCTYPE northwind:BOOKS SYSTEM "c:\Books.dtd">
1
<!ENTITY copyright "©">
```

Once again, you have created a reference to an *IXMLDOMDocument* interface. With this reference, you can use the *doctype* property to get a reference to an *IXMLDOMDocumentType* interface. Then you use the *name* and *xml* properties of the *IXMLDOMDocumentType* interface to get a node's name and its XML content. Notice that the *parameter* entity was not included in the entities collection.

XMLDOMDocumentFragment Object

The *XMLDOMDocumentFragment* object will be used to create fragments of documents that can be appended to another document. When the *XMLDOMDocumentFragment* object is inserted into a document object, the root node of the *XMLDOMDocument-Fragment* is not inserted, only its children. Thus, *XMLDOMDocumentFragment* objects are useful for inserting child elements into a document.

The *XMLDOMDocumentFragment* object implements the *IXMLDOMDocument-Fragment* interface. This interface inherits all the *IXMLDOMNode* interface's methods and properties, but it doesn't extend the interface, so this interface has no additional methods or properties of its own.

XMLDOMElement Object

The *XMLDOMElement* object contains the elements in the document and is the most common node. The text nodes belonging to an element object contain the content of the element. If there is no text content, the *XMLDOMText* object will be *null*. This object implements the *IXMLDOMElement* interface. When working with the *IXMLDOMElement* interface, you must know beforehand what the names of the elements and attributes are that you want to retrieve and place them in the code. This is because the *IXMLDOMElement* interface sets and retrieves attributes and elements by their names.

In addition to the methods and properties of the *IXMLDOMNode* interface, the *IXMLDOMElement* interface has the following extended property and methods:

EXTENDED *IXMLDOMELEMENT* PROPERTY

Name	*Description*
tagName	Sets or returns the name of the element

EXTENDED *IXMLDOMELEMENT* METHODS

Name	*Description*
getAttribute (attributeName)	Returns the value of the attribute with the specified *attributeName*.
getAttributeNode (attributeName)	Returns the attribute node object with the specified *attributeName*.
getElementsByTagName (elementName)	Returns an *XMLDOMNodeList* object that contains all the descendant elements named *elementName*.
normalize()	Combines the adjacent text nodes into one unified *text* node. Normalizes all descendant *text* nodes of the element.

Name	*Description*
removeAttribute (attributeName)	Removes the value of the attribute named *attributeName*.
removeAttributeNode (attributeNode)	Removes the *attribute* node named *attributeNode* and returns the node. If there is a default value in the schema or DTD, a new *attribute* node will be created with the default value.
setAttribute (attributeName, newValue)	Sets the *attribute* node named *attributeName* to the value *newValue*.
setAttributeNode (attributeName)	Adds a new *attribute* node to the element. An existing *attribute* node by the same name will be replaced.

You can get a reference to an *IXMLDOMElement* interface by using the *selectNodes* method and XSL. You will now create an example to select a single *element* node. To do so, follow these steps:

1. Add another command button to the *frmDOMTest* form with the name *cmdElement* and the caption *Element*.
2. Insert the following code into the click event handler of this button:

```
Private Sub cmdElement_Click()
Dim objXMLDoc As DOMDocument

Dim objElement As IXMLDOMElement

Set objXMLDoc = New DOMDocument
objXMLDoc.async = False
objXMLDoc.Load ("c:\Books.xml")

Set objElement = objXMLDoc.selectNodes("//item/*").Item(1)
Debug.Print objElement.xml

Set objElement = Nothing
Set objXMLDoc = Nothing
End Sub
```

This example application selects the second child node of the *item* element. Then it retrieves the entire XML content of this node by using the *xml* property. When you run this example and click the *Element* button, the result is as follows:

```
<author>Danzig</author>
```

You can get a reference to the *XMLDOMAttribute*, *XMLDOMEntity*, *XMLDOMEntityReference*, *XMLDOMNotation*, *XMLDOMCharacterData*, *XMLDOMText*, *XMLDOMCDATASection*, *XMLDOMComment*, and *XMLDOMProcessingInstruction* by using XSL. You can get references to these node objects just as we used the XSL statement *"//item/*"* to get references to the node *item* in the previous application. (We will discuss the XSL syntax in detail in Chapter 12.) So in the following sections, we'll examine these interfaces without demonstrating how to use them in the applications.

XMLDOMAttribute Object

The *XMLDOMAttribute* object represents an *attribute* node of the *XMLDOMElement* object. This object implements the *IXMLDOMAttribute* interface. In addition to the properties and methods it inherits from the *IXMLDOMNode* interface, the *IXMLDOMAttribute* interface has the following additional properties:

EXTENDED *IXMLDOMATTRIBUTE* PROPERTIES

Name	*Description*
name	Sets or returns the name of the attribute
value	Sets or returns the value of the attribute

NOTE The W3C specification lists this object as the *attr* object, instead of the *XMLDOMAttribute* object.

XMLDOMEntity Object

The *XMLDOMEntity* object represents a parsed or unparsed entity declared in a DTD. The *XMLDOMEntity* object is not the entity declaration. This object implements the *IXMLDOMEntity* interface. The properties of this interface are read-only. Like most of the other DOM interfaces, this interface inherits the *IXMLDOMNode* interface too. In addition to the *IXMLDOMNode* properties and methods, the *IXMLDOMEntity* object extends the *IXMLDOMNNode* object with the following properties:

EXTENDED *IXMLDOMENTITY* PROPERTIES

Name	*Description*
publicID	Returns the value of the PUBLIC identifier for the entity node
systemID	Returns the value of the SYSTEM identifier for the entity node
notationName	Returns the notation name

XMLDOMEntityReference Object

The *XMLDOMEntityReference* object represents an *entity reference* node contained in the XML document. Remember that an XML processor doesn't expand the entities until they are needed. Thus, if the XML processor doesn't expand the entities, there will be no *XMLDOMEntityReference* objects. The replacement text will be located in the *text* property. The *IXMLDOMEntityReference* interface implemented by the *XMLDOMEntityReference* object inherits all the methods and properties of, but does not extend, the *IXMLDOMNode* interface.

XMLDOMNotation Object

The *XMLDOMNotation* object represents a notation declared in the DTD with the declaration <!NOTATION>. The *XMLDOMNotation* object implements the *IXMLDOMNotation* interface that inherits all the methods and properties of the *IXMLDOMNode* interface and extends the *IXMLDOMNode* interface with the following properties:

ADDITIONAL *IXMLDOMNOTATION* PROPERTIES

Name	*Description*
publicID	Returns the value of the PUBLIC identifier for the *notation* node
systemID	Returns the value of the SYSTEM identifier for the *notation* node

XMLDOMCharacterData Object

The *XMLDOMCharacterData* object makes it easier to work with the text content in an XML document. The *IXMLDOMCharacterData* interface implemented by the *XMLDOMCharacterData* object also inherits the *IXMLDOMNode* interface, so it includes all the properties and methods of the *IXMLDOMNode* interface. Moreover, it extends the *IXMLDOMNode* interface with the following properties and methods:

EXTENDED *IXMLDOMCHARACTERDATA* PROPERTIES

Name	*Description*
data	Contains the node's data. The actual data will depend on the type of node.
length	Returns the number of characters in the data string.

EXTENDED *IXMLDOMCHARACTERDATA* METHODS

Name	*Description*
appendData (text)	Appends the *text* argument onto the existing data string
deleteData (charOffset, numChars)	Deletes *numChars* characters off the data string starting at *charOffset*
insertData (charOffset, text)	Inserts the supplied text into the data string at the *charOffset*
replaceData (charOffset, numChars, text)	Replaces *numChars* characters with the supplied text starting at *charOffset*
substringData (charOffset, numChars)	Returns the *numChars* characters as a string, starting at *charOffset*, in the data string

XMLDOMText Object

The *XMLDOMText* object represents the *text* node of an element or an attribute. You can use the *XMLDOMText* object to build *text* nodes and append them into an XML document. The *IXMLDOMText* interface implemented by the *XMLDOMText* object inherits the *IXMLDOMCharacterData* interface and extends it with the following method:

EXTENDED *IXMLDOMTEXT* METHOD

Name	*Description*
splitText (charOffset)	Splits the node into two nodes at the specified character offset and then inserts the new node into the XML document immediately following the node

XMLDOMCDATASection Object

An *XMLDOMCDATASection* object is used for sections of text that are not to be interpreted by the processor as markup. The *XMLDOMCDATASection* object implements the *IXMLDOMCDATASection* interface. This interface inherits the *IXMLDOMText* interface and has the same methods and properties as the *IXMLDOMText* interface.

XMLDOMComment Object

The *XMLDOMComment* object contains comments that are in the XML document. The *IXMLDOMComment* interface implemented by this object inherits the *IXMLDOMCharacterData* interface and also possesses the same methods and properties as *IXMLDOMCharacterData*. The *IXMLDOMComment* interface does not extend the *IXMLDOMCharacterData* interface.

XMLDOMProcessingInstruction Object

The *XMLDOMProcessingInstruction* object contains the processing instructions in the document between the <? tag and the ?> tag. The content enclosed in these two tags is divided into the target and data content. The *IXMLDOMProcessingInstruction* interface implemented by the *XMLDOMProcessingInstruction* object inherits the *IXMLDOMNode* interface and has the same methods as the *IXMLDOMNode* interface. It extends the *IXMLDOMNode* interface with the following properties:

EXTENDED *IXMLDOMPROCESSINGINSTRUCTION* PROPERTIES

Name	*Description*
data	Sets or returns the content of the processing instruction, which doesn't contain the target
target	Sets or returns the target application to which the processing instruction is directed

XMLDOMImplementation Object

Because different applications that support XML can support different features of XML, the W3C included the *XMLDOMImplementation* object, which can be used to determine whether certain features are supported in a particular application. The *XMLDOMImplementation* object implements the *IXMLDOMImplementation* interface. This interface has one method called *hasFeature* that returns *true* if the specified feature is implemented by the specified version of the XML DOM implementation. To see how this object works, add another command button to the *frmDOMTest* form with the name *cmdImplementation* and the caption *Implementation*. Add the following code to the click event of this button:

```
Private Sub cmdImplementation _Click()
    Dim objImplementation As IXMLDOMImplementation
    Dim objXMLDoc As DOMDocument
```

(continued)

```
    Set objXMLDoc = New DOMDocument
    objXMLDoc.async = False
    objXMLDoc.Load ("c:\Books.xml")
    Set objImplementation = objXMLDoc.implementation
    'Currently accepted values for feature: XML, DOM, and MS-DOM
    Debug.Print "MS-DOM: " & _
          objImplementation.hasFeature("MS-DOM", "1.0")
    Debug.Print "XML: " & _
          objImplementation.hasFeature("XML", "1.0")
    Debug.Print "DOM: " & _
          objImplementation.hasFeature("DOM", "1.0")
End Sub
```

If you have Internet Explorer 5 installed on your computer, running this application and clicking the *Implementation* button will give you the following results:

```
MS-DOM: True
XML: True
DOM: True
```

HasFeature returning true shows that Internet Explorer 5 supports XML, DOM, and the MS-DOM.

XMLDOMParseError Object

The *XMLDOMParseError* object is an extension to the W3C specification. It can be used to get detailed information on the last error that occurred while either loading or parsing a document. The *XMLDOMParseError* object implements the *IXMLDOMParseError* interface that has the following properties:

IXMLDOMPARSEERROR PROPERTIES

Name	*Description*
errorCode	Returns the error number or error code as a decimal integer.
filepos	Returns the absolute character position in the document where the error occurred.
line	Returns the number of the line where the error occurred.
linepos	Returns the absolute character position in the line where the error occurred.
reason	Returns a description of the source and reason for the error. If the error is in a schema or DTD, it can include the URL for the DTD or schema and the node in the schema or DTD where the error occurred.
srcText	Returns the full text of the line that contains the error. If the error cannot be assigned to a specific line, an empty line is returned.
url	Returns the URL for the most recent XML document that contained an error.

To see how the *IXMLDOMParseError* interface is used, we will make a change to the first line of code in the Books.dtd: *<!ELEMENT northwind:BOOKS (item)>* by removing the *northwind:* from the line, as shown here:

```
<!ELEMENT BOOKS  (item )>
```

Add another command button to the *frmDOCTest* form with the name *cmdParseError* and the caption *ParseError*. In the click event handler of this button, place the following code:

```
Private Sub cmdParseError_Click()
    Dim objXMLDoc As DOMDocument
    Dim objXMLParseError As IXMLDOMParseError
    On Error GoTo cmdParseErrorError
    Set objXMLDoc = New DOMDocument
    objXMLDoc.async = False
    objXMLDoc.Load ("c:\Books.xml")
    'Check whether there was an error parsing the file using the
    'parseError object.
    If objXMLDoc.parseError.errorCode <> 0 Then
        'If there was an error, raise it to jump into error trap.
        Err.Raise objXMLDoc.parseError.errorCode
    End If
    Exit Sub
'Error Trap
cmdParseErrorError:
    Set objXMLParseError = objXMLDoc.parseError
    With objXMLParseError
        'Because of the With objXMLParseError, .errorCode is the
        'same as objXMLParseError.errorCode.
        'First check whether the error was caused by a parse error.
        If .errorCode <> 0 Then
            Debug.Print "The following error occurred:" & vbCrLf & _
                 "error code: " & .errorCode & vbCrLf & _
                 "error file position: " & .filepos & vbCrLf & _
                 "error line: " & .Line & vbCrLf & _
                 "error line position: " & .linepos & vbCrLf & _
                 "error reason: " & vbCrLf & .reason & vbCrLf & _
                 "error source text: " & vbCrLf & .srcText & _
                           " Test:cmdParseError"
        Else
            'If the error was not caused by a parse error, use
            'regular Visual Basic error object.
            MsgBox "The following error has occurred:" & vbCrLf & _
                "Error Description: " & Err.Description & _
                                        vbCrLf & _
```

(continued)

```
                "Error Source: " & vbCrLf & Err.Source &  _
                        " Test:cmdParseError" & vbCrLf & _
                "Error Number: " & Err.Number
        End If
    End With

    Set objXMLParseError = Nothing
    Set objXMLDoc = Nothing
End Sub
```

Before you actually run this code, take a look at it to see what an error handler in production code should look like. A parse error does not raise its error. After the parse error occurs, the Visual Basic error number (*Err.Number*) is still 0. Thus, you must use the *ParseError* object to check for an error after you load an XML document. If there is a parse error, you can raise an error, as was done above, to bring you into the error trap.

The error trap provides a message if the error occurred in parsing the file. If the error was caused for some other reason, the standard Visual Basic *Err* error object is used. Also notice that the name of the application and the method are included in the source, making it easier to find and fix bugs.

Now you can run the updated application and click the *ParseError* button. With the change in the DTD, you will receive the following message:

```
The following error occurred:
error code: -1072898035
error file position: 137
error line: 3
error line position: 64
error reason:
The element 'northwind:BOOKS' is used but not declared in the
DTD/Schema.

error source text:
<northwind:BOOKS xmlns:northwind="www.northwindtraders.com/PO">
Test:cmdParseError
```

In this case, because the DTD has no awareness of namespaces, the namespace qualified northwind:BOOKS in the XML document no longer matches the DTD declaration.

To create a different error, change the DTD back to its original form by adding back the *northwind:*. Remove *#FIXED "www.northwindtraders.com/PO"* from the second line in the DTD, and the second line will look as follows:

```
<!ATTLIST northwind:BOOKS  xmlns:northwindCDATA>
```

Now run the application and click the *ParseError* button; the error message will look as follows:

```
The following error occurred:
error code: -1072896766
error file position: 86
error line: 2
error line position: 51
error reason:
A string literal was expected, but no opening quote character
  was found.

error source text:
<!ATTLIST northwind:BOOKS  xmlns:northwind CDATA>
Test:cmdParseError
```

Notice that the error source text is now the information from the DTD. The reason might not be that obvious, but by looking at the error reason you will see that you need to include #REQUIRED, #FIXED, or #IMPLIED in the DTD. You must use #FIXED because this is a namespace attribute.

XTLRuntime Object

The *XTLRuntime* object works with XSL style sheets. It implements the *IXTLRuntime* interface that has nine methods: *absoluteChildNumber*, *ancestorChildNumber*, *childNumber*, *depth*, *formatDate*, *formatIndex*, *formatNumber*, *formatTime*, and *uniqueID*. We will cover XSL style sheets and these methods in Chapter 12.

XMLHTTPRequest Object

The *XMLHTTPRequest* object, which is an extension of the W3C specification, can be used to send and receive HTTP messages to and from a Web server. Once a message is received, it can be parsed by XML DOM objects. You could use the *XMLHTTPRequest*

object to create applications that build and send SOAP messages to the server. This object implements the *IXMLHTTPRequest* interface, which has the following properties and methods:

IXMLHTTPREQUEST PROPERTIES

Name	*Description*
readyState	Indicates the current state of the document being loaded. The value changes as the XML document loads.
responseBody	Returns the response as an array of unsigned bytes.
responseStream	Returns the response object as an *IStream* object.
responseText	Returns the response object as a text string.
responseXML	Returns the response as an XML document. When this property is used, validation is turned off to prevent the parser from attempting to download a linked DTD or schema.
status	Returns a status code as a long integer.
statusText	Returns the status as a string.

IXMLHTTPREQUEST METHODS

Name	*Description*
*abort()**	Cancels the current HTTP request
*getAllResponseHeaders()**	Returns all the HTTP headers as name value pairs separated by carriage return-linefeeds
*getResponseHeader (headerName)**	Gets the response header with the name *headerName*
*open (method, url, async, userID, password)**	Initializes a request and specifies the HTTP method, the URL, and if the response is asynchronous, the user information
*send()**	Sends an HTTP request to the server and waits to receive a response
setRequestHeader (headerName, value)	Sets HTTP headers that are sent to the server

This chapter has described a complete set of objects that can allow you to manipulate XML information and send HTTP streams to and from a Web server. It's time we look at a more complete example of using some of these objects, including

the *XMLHTTPRequest* object. To do this, we will write the code to create a SOAP client and server application.

SOAP Application Using XML DOM

The SOAP application you are about to create will use Visual Basic on the client side and Active Server Pages (ASP) on the server side. The XML DOM will be used on both the client and the server.

Instead of loading XML data from a file or a string, this application will build an XML file using the XML DOM objects and show you how to write code to build XML documents dynamically.

In a real application, you could build the XML document using a Visual Basic application that processes SOAP requests or use JScript or VBScript in a browser application with the values that the user has inputted into the browser form. For this example, we will provide values for the XML elements.

We will start by creating a new Visual Basic EXE project. For this example, you will need a copy of the XML parser, which at the time of this printing is version 3. Thus, you should add a reference to the Microsoft XML 3.0 by choosing References from the Project menu, and then choosing Microsoft XML version 3 from the References dialog box.

> **NOTE** This book will use both the 3.0 and 2.6 versions of the XML parser. When you are working with these examples, you can use the most current version by changing the declarations of the *DOMDocument* objects.

Add a button to the *frmDOMTest* form with the name *cmdSoap* and the caption *Soap*. Enter the following code into the click event handler of this button:

```
Dim objXMLDoc As DOMDocument30
Dim objHTTPRequest As XMLHTTP30
Dim objXMLDocResponse As DOMDocument30
Set objHTTPRequest = New XMLHTTP30
Set objXMLDoc = New DOMDocument
Set objXMLDocResponse = New DOMDocument30
Set objXMLDocNode = New DOMDocument30
Dim lngResponse As Long
Dim lngOrderID As Long
Dim strResponse As String

objXMLDoc.async = False
'We will begin by using the loadXML string to load in the root
'node.
```

(continued)

```
objXMLDoc.loadXML _
    "<SOAP-ENV:Envelope xmlns:SOAP-ENV = " & _
    "'http://schemas.xmlsoap.org/soap/envelope'" & _
    " xmlns:xsi='http://www.w3.org/1999/XMLSchema/instance' " & _
    " xsi:schemaLocation=" & _
    "'http://www.northwindtraders.com/schemas/NPOSchema.xsd'>" & _
    "<SOAP-ENV:Body xsi:type='northwindBody'><UpdatePO>" & _
    "<OrderID>" & "</OrderID>" & _
    "<return>" & strReturn & _
    "</return></UpdatePO></SOAP-ENV:Body>" & _
    "</SOAP-ENV:Envelope>"

'The createNode method returns a node object.
'If the element has a namespace prefix, it must be included.
'The appendChild method will take the node object created with
'createNode and add it to XML document node collection.
objXMLDoc.documentElement.appendChild _
    objXMLDoc.createNode(NODE_ELEMENT, "SOAP-ENV:Body", _
    "'http://schemas.xmlsoap.org/soap'")
'For an attribute, you must use the attributes property.
objXMLDoc.documentElement.firstChild.Attributes.setNamedItem _
    objXMLDoc.createNode(NODE_ATTRIBUTE, "xsi:type", _
    "xmlns:xsi='http://www.w3.org/1999/XMLSchema/instance'")
objXMLDoc.selectSingleNode _
    ("SOAP-ENV:Envelope/SOAP-ENV:Body").appendChild _
    objXMLDoc.createNode(NODE_ELEMENT, "UpdatePO", "")
objXMLDoc.selectSingleNode _
    ("SOAP-ENV:Envelope/SOAP-ENV:Body/UpdatePO"). _
    appendChild objXMLDoc.createNode(NODE_ELEMENT, "OrderID", "")
objXMLDoc.selectSingleNode _
    ("SOAP-ENV:Envelope/SOAP-ENV:Body/UpdatePO"). _
    appendChild objXMLDoc.createNode _
    (NODE_ELEMENT, "CustomerNumber", "")
objXMLDoc.selectSingleNode _
    ("SOAP-ENV:Envelope/SOAP-ENV:Body/UpdatePO"). _
    appendChild objXMLDoc.createNode(NODE_ELEMENT, "Item", "")
objXMLDoc.selectSingleNode _
    ("SOAP-ENV:Envelope/SOAP-ENV:Body/UpdatePO"). _
    appendChild objXMLDoc.createNode(NODE_ELEMENT, "Quantity", "")
objXMLDoc.selectSingleNode _
    ("SOAP-ENV:Envelope/SOAP-ENV:Body/UpdatePO"). _
    appendChild objXMLDoc.createNode(NODE_ELEMENT, "return", "")

'We must now set the values for each node.
'We will use XSL in selectSingleNode to get the node.
'For the attribute, we must use getNamedItem to get the attribute.
objXMLDoc.selectSingleNode("SOAP-ENV:Envelope/SOAP-ENV:Body"). _
    Attributes.getNamedItem("xsi:type").nodeValue = "NorthwindBody"
```

```
objXMLDoc.selectSingleNode _
    ("SOAP-ENV:Envelope/SOAP-ENV:Body/UpdatePO/OrderID").Text = "0"
objXMLDoc.selectSingleNode _
    ("SOAP-ENV:Envelope/SOAP-ENV:Body/UpdatePO/CustomerNumber"). _
    Text = "999"
objXMLDoc.selectSingleNode _
    ("SOAP-ENV:Envelope/SOAP-ENV:Body/UpdatePO/Item").Text = "89"
objXMLDoc.selectSingleNode _
    ("SOAP-ENV:Envelope/SOAP-ENV:Body/UpdatePO/return").Text = "0"
objXMLDoc.selectSingleNode _
    ("SOAP-ENV:Envelope/SOAP-ENV:Body/UpdatePO/Quantity"). _
    Text = "35"
'Initialize HTTP request with Post.
objHTTPRequest.open "POST", "http://localhost/XMLSample/SOAP.asp"
'Set the SOAP headers.
objHTTPRequest.setRequestHeader "POST", "/Northwind.Order HTTP/1.1"
objHTTPRequest.setRequestHeader "Host", "www.northwindtraders.com"
objHTTPRequest.setRequestHeader "Content-Type", "text/xml"
objHTTPRequest.setRequestHeader "content-length", _
    Len(objXMLDoc.xml)
objHTTPRequest.setRequestHeader _
    "SOAPAction", "urn: northwindtraders.com:PO#UpdatePO"
'Send the message.
objHTTPRequest.send objXMLDoc.xml
'Set the response document object equal to the responseXML object.
Set objXMLDocResponse = objHTTPRequest.responseXML

'Wait to get result.
Dim lLoopNum As Long
Do While objXMLDocResponse.selectSingleNode _
("SOAP-ENV:Envelope/SOAP-ENV:Body/UpdatePO/return") Is Nothing _
    And lLoopNum < 10000
    DoEvents
    lLoopNum = lLoopNum + 1
Loop
'Get the return values.
strResponse = objXMLDocResponse.selectSingleNode _
    ("SOAP-ENV:Envelope/SOAP-ENV:Body/UpdatePO/return").Text
If strResponse = "" Then
    'Raise an error here.
Else
    MsgBox "Response = " & strResponse
End If

lngOrderID = CLng(objXMLDocResponse.selectSingleNode _
    ("SOAP-ENV:Envelope/SOAP-ENV:Body/UpdatePO/OrderID").Text)

Set objXMLDocResponse = Nothing
Set objXMLDoc = Nothing
Set objHTTPRequest = Nothing
```

To be able to run the application, you need to create an object named *objOrderPO* that contains the *UpdatePO* method. Since we will not actually create this object, we will comment out the code where the method will be created and create dummy variables so you can run the example. You also need to create an ASP page named SOAP.asp and put it on the local server in a folder called *XMLSample* under the default Web site. Now let's take a look at how to create an ASP page.

The ASP page uses two document objects, one called *objXMLDocRequest*, which holds the XML document sent from the server, and the other called *objXMLDocResponse*, which contains the XML document that is returned to the server. The names of the object and the method that need to be called are retrieved from the header first. Next you retrieve the XML document from the request object, retrieve the parameter values located in the XML body, and finally create the object and call the method. Once the method has been called, the return XML string is built and placed in the response object.

Create an ASP page and add the following code:

```
<%@ Language=VBScript %>
<SCRIPT LANGUAGE=vbscript RUNAT=Server>
Dim objXMLDocRequest
Dim objXMLDocResponse
Dim result, strObject, strMethod
Dim strCustomerNumber, strItem, strReturn
Dim strOrderID, strQuantity

Set objXMLDocRequest= Server.CreateObject ("Microsoft.XMLDOM")
Set objXMLDocResponse= Server.CreateObject ("Microsoft.XMLDOM")
'You must set the content type so that the returning document will
'be identified as XML.
Response.ContentType = "text/xml"
'Load the posted XML document.
strObject= Request.ServerVariables.Item ("HTTP_POST")
'Remove /
strObject= Right(strObject, len(strObject)-1)
'Remove HTTP...
strObject= Left(strObject, instr(1, strObject, " ") )
strMethod= Request.ServerVariables.Item ("HTTP_SOAPAction")
'Strip off URL.
strMethod=Right(strMethod, len(strMethod)-instr(1, strMethod,"#"))
'Use the load method to get the XML sent from the client out of the
'request object.
objXMLDocRequest.load Request
'Now that you have the XML data, use the values in the XML
'document to set the local variables.
strCustomerNumber=objXMLDocRequest.SelectSingleNode _
    ("SOAP-ENV:Envelope/SOAP-ENV:Body/UpdatePO/CustomerNumber").Text
```

```
strItem = objXMLDocRequest.SelectSingleNode _
    ("SOAP-ENV:Envelope/SOAP-ENV:Body/UpdatePO/Item").Text
strQuantity = objXMLDocRequest.SelectSingleNode _
    ("SOAP-ENV:Envelope/SOAP-ENV:Body/UpdatePO/Quantity").Text
'Using the name of the object passed in the header, we will
'instantiate the object.
'Because we do not actually have an object called objOrderPO, we
'will comment out the next line of code.
'Set objOrderPO = _
'    server.CreateObject (Request.ServerVariables.Item(strObject))
'Call the correct method passing in the parameters.
Select Case strMethod
    Case "UpdatePO"
'Because we do not actually have an object called objOrderPO, we
'will comment out the next two lines of code. We are also adding
'dummy values for the strReturn and strOrderID variables.
    'strReturn=objOrderPO.UpdatePO _
    '(strCustomerName, strItem, strQuantity, strOrderID)
    strReturn="0"
    strOrderID="100"
    Case "DeletePO"
    strReturn=objOrderPO.DeletePO _
         (strCustomerName, strItem, strQuantity, strOrderID)
End Select
'Create XML that is going back to the client using a string.
objXMLDocResponse.LoadXML _
    "<SOAP-ENV:Envelope xmlns:SOAP-ENV =" & _
"'http://schemas.xmlsoap.org/soap/envelope'" & _
    " xmlns:xsi='http://www.w3.org/1999/XMLSchema/instance' " & _
    " xsi:schemaLocation=" & _
    "'http://www.northwindtraders.com/schemas/NPOSchema.xsd'>" & _
    "<SOAP-ENV:Body xsi:type='northwindBody'><UpdatePO>" & _
    "<OrderID>" & strOrderID & "</OrderID>" & _
    "<return>" & strReturn & _
    "</return></UpdatePO></SOAP-ENV:Body>" & _
    "</SOAP-ENV:Envelope>"
'Return the XML.
Response.Write objXMLDocResponse.xml
</SCRIPT>
```

In this example, you can see how tightly bound the client-side code is to the object that is being called. The client-side object has to build an XML document that contains the correct parameters and also has to create a header with the right object and method names.

XML PARSER VERSION 2.6 AND VERSION 3.0

The XML parser version 2.6 and version 3.0 extend the older version that came with Internet Explorer 5. The XML parser version 2.6 is a separate DLL (Msxml2.dll) that can be installed in addition to the original XML parser. Various options are available for running the two DLLs together, but those options are beyond the scope of this book. You can check the XML SDK 2.5 (which can be viewed on Microsoft's Web site) for more information about the DLLs. Version 3.0 of the parser is installed as a new DLL (Msxml3.dll) with new version-dependent CLSIDs and ProgIDs to protect those applications that use Msxml.dll or Msxml2.dll and allow you to choose the version of the parser to use in your code.

Version 2.6 comes with five additional XML document objects: *XMLDOMDocument2*, *XMLDOMSchemaCache*, *XMLDOMSelection*, *XMLDOMXSLProcessor*, and *XMLDOMXSLTemplate*. We will discuss the *XMLDOMXSLTemplate* and *XMLDOMXSLProcessor* objects in Chapter 12 when we discuss XSL. Version 3.0 doesn't add any new features to version 2.6. Thus, we'll have a detailed discussion about the *XMLDOMDocument2*, *XMLDOMSchemaCache*, and *XMLDOMSelection* objects in the following section.

XMLDOMDocument2, *XMLDOMSchemaCache*, and *XMLDOMSelection* Objects

The *XMLDOMDocument2* object implements the *IXMLDOMDocument2* interface. This interface is an extension of the *IXMLDOMDocument* interface that supports schema caching and validation. The *IXMLDOMDocument2* interface inherits all the original properties and methods of the *IXMLDOMDocument* interface and adds the following new properties:

EXTENDED *IXMLDOMDOCUMENT2* PROPERTIES

Name	*Description*
namespaces	Returns a list of all of the namespaces in the document as an *XMLDOMSchemaCollection*(*schemaCache* object)
schemas	Locates all the schema documents using the *XMLDOMSchemaCollection* (*schemaCache* object)

The *XMLSchemaCache* object which implements the *IXMLDOMSchemaCollection* interface contains information about the schemas and namespaces used by an XML document. This interface has the following property and methods:

IXMLDOMSchemaCollection Property

Name	Description
length	Returns the number of namespaces that are currently in the collection

IXMLDOMSchemaCollection Methods

Name	Description
add (namespaceURI, schema)	Adds a new schema to the schema collection. The specified schema is associated with the given namespace URI.
addCollection (XMLDOMSchemaCollection)	Adds all the schemas from another collection into the current collection.
get (namespaceURI)	Returns a read-only DOM node containing the <Schema> element.
namespaceURI (index)	Returns the namespace for the specified index.
remove (namespaceURI)	Removes the specified namespace from the collection.

Now that we've examined the properties and methods in the *IXMLDOMDocument2* interface and the *IXMLDOMSchemaCollection* interface, it's time to see how they are used in the application. Create a new Visual Basic Standard EXE project, and name the default form *frmTestDOM2*. In the form *frmTestDOM2*, place a command button with the name *cmdSchemas* and the caption *Schemas*. Place the following code in the click event handler of this button:

```
Private Sub cmdSchemas_Click()
    Dim objXMLDoc As DOMDocument26
    Dim objXMLSchemas As XMLSchemaCache
    Dim lngSchemaCounter As Long
    Set objXMLDoc = New DOMDocument
    objXMLDoc.async = False
    objXMLDoc.Load ("c:\Books.xml")
    Set objXMLSchemas = objXMLDoc.namespaces
    For lngSchemaCounter = 0 To objXMLSchemas.length - 1
        Debug.Print "URL: " & _
            objXMLSchemas.namespaceURI(lngSchemaCounter)
        Debug.Print "Type: " & _
            objXMLDoc.selectSingleNode("//author").dataType
    Next
    Set objXMLSchemas = Nothing
    Set objXMLDoc = Nothing
End Sub
```

This application loads the Books.xml document that contains a reference to the Books.dtd. If you run this program and click the *Schemas* button when the XML document is referencing the DTD, you will get the following result:

```
URL: www.northwindtraders.com/PO
Type:
```

Looking back at the original XML document, you have the following two lines of code:

```
<!DOCTYPE northwind:BOOKS SYSTEM "c:\Books.dtd">
<northwind:BOOKS xmlns:northwind ="www.northwind.com/PO">
```

You can see that the !DOCTYPE declaration defined the location of the DTD. Because XML documents are aware of namespaces, the parser recognized that the prefix *northwind* was a namespace prefix for the *BOOKS* element. In the next line of code, the parser finds the declaration that associates a namespace with that name and with that declaration. There is no data type information in the result above because this is a DTD.

We will now use a BizTalk schema instead of a DTD. Create the following schema file called Books.xsd:

```
<Schema name="BOOKS" xmlns="urn:schemas-microsoft-com:xml-data"
   xmlns:dt="urn:schemas-microsoft-com:datatypes">
   <ElementType name="BOOKS" content="eltOnly" model="closed">
      <element type="item"/>
   </ElementType>
   <ElementType name="item" content="eltOnly" model="closed">
      <element type="title"/>
      <element type="author"/>
      <element type="price"/>
      <element type="quantity"/>
   </ElementType>
   <ElementType name="title" content="mixed" model="closed">
      <AttributeType name="language" dt:type="string"/>
      <attribute type="language"/>
   </ElementType>
   <ElementType name="author" content="textOnly" model="closed"
      dt:type="string"/>
   <ElementType name="price" content="textOnly" model="closed"
      dt:type="string"/>
   <ElementType name="quantity" content="textOnly" model="closed"
      dt:type="string"/>
</Schema>
```

Create a new XML document called Books2.xml and add the following code:

```
<?xml version="1.0" ?>
<northwind:BOOKS xmlns:northwind="x-schema:c:\Books.xsd">
   <northwind:item>
      <northwind:title language="English">Number, the
            Language of Science</northwind:title>
      <northwind:author>Danzig</northwind:author>
      <northwind:price>5.95</northwind:price>
      <northwind:quantity>3</northwind:quantity>
   </northwind:item>
</northwind:BOOKS>
```

Notice that you need to change the code so that it references Books.xsd. To get a reference to the schema we must use *x-schema* in the namespace declaration. The x-schema syntax is used by Internet Explorer 5 to identify where the schema is. The namespace prefix had to be added to all the elements in order for this code to work. While that should not have been necessary, errors would result if the namespace prefix was not added. This is a reminder that the way things are implemented might not always be what you expect. Finally, change the Visual Basic code so that it references Books2.xml and the namespace prefix is included in the XSL statement:

```
Private Sub cmdSchemas_Click()
    Dim objXMLDoc As DOMDocument26
    Dim objXMLSchemas As XMLSchemaCache
    Dim lngSchemaCounter As Long
    Set objXMLDoc = New DOMDocument
    objXMLDoc.async = False
    objXMLDoc.Load ("c:\Books2.xml")
    Set objXMLSchemas = objXMLDoc.namespaces
    For lngSchemaCounter = 0 To objXMLSchemas.length - 1
        Debug.Print "URL: " & _
            objXMLSchemas.namespaceURI(lngSchemaCounter)
        Debug.Print "Type: " & objXMLDoc.selectSingleNode _
            ("//northwind:author").dataType
    Next
End Sub
```

After you make the changes, running the program and clicking the *Schemas* button will result in the following output:

```
URL: x-schema:c:\Books.xsd
Type:
```

With the current release of the XML parser, the string data type is not being returned.

The *XMLDOMSelection* object represents a list of nodes that match an XSL pattern or an XPath expression. The *XMLDOMSelection* object can be created by using the *selectNodes* method of the *IXMLDOMDocument2* interface that is included in version 2.6 and later of Microsoft XML parser. This object implements the *IXMLDOMSelection* interface, which inherits from the *IXMLDOMNodeList* interface. In Visual Basic, the *XMLDOMSelection* object can be used as follows:

```
Dim objXMLDoc As DOMDocument26
Dim objXMLSelection As IXMLDOMSelection

Set objXMLDoc = New DOMDocument26
objXMLDoc.async = False
objXMLDoc.Load ("C:\Books.xml")
Set objXMLSelection = objXMLDoc.selectNodes("//item [quantity=3]")
Debug.Print objXMLSelection.expr
```

In this example, we select the *item* element that has an attribute with a value of *3*. The *expr* property of *IXMLDOMSelection* is used to retrieve the XPath expression string. In this case, *"//item [quantity=3]"* is returned.

SUMMARY

In this chapter, you reviewed the majority of the XML DOM objects. These objects allow you to manipulate an XML document in virtually any way that might be required by an application. These objects provide a powerful set of tools that allow you to begin building complete XML applications. You can also use the DOM objects to both send messages to and receive messages from a Web server, which allows you to create SOAP messages as shown in the example in this chapter. In the next chapter, we will look at XSL and learn how the DOM can be used with it.