XML
Powered by Domino
How to use XML with Lotus Domino

November 2000
Take Note!

Before using this information and the product it supports, be sure to read the general information in Appendix B, “Special notices” on page 279.

First Edition (November 2000)

This edition applies to Release 5.05 of Lotus Domino.

Comments may be addressed to:
IBM Corporation, International Technical Support Organization
Dept. TQH 1CP-5605E
1 Charles Park
Cambridge, Massachusetts 02142-1245

When you send information to IBM, you grant IBM a non-exclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 2000. All rights reserved.
Note to U.S Government Users – Documentation related to restricted rights – Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.
# Contents

**Figures** ................................................................. vii

**Tables** ........................................................................ xi

**Preface** ................................................................. xiii

The team that wrote this redbook .............................................. xv

Comments welcome ............................................................ xvi

## Chapter 1. Introduction to XML ...................................................... 1

1.1 XML overview ......................................................... 1

1.1.1 XML and HTML .................................................. 1

1.1.2 Why use XML ..................................................... 4

1.1.3 Connecting with XML ........................................... 4

1.2 XML document structure ............................................. 5

1.2.1 XML syntax ..................................................... 5

1.2.2 Syntax rules ..................................................... 7

1.2.3 Well-formed XML documents ................................. 8

1.2.4 Well-formed and valid XML documents ....................... 10

1.3 XHTML ........................................................................ 10

1.4 XML parser ............................................................. 12

1.4.1 Parsing XML ..................................................... 12

1.4.2 APIs for document handling .................................. 14

1.4.3 Document Object Model ...................................... 15

1.4.4 Simple API for XML (SAX) .................................. 17

1.5 Designing an XML data structure .................................. 21

1.5.1 DTD ................................................................ 22

1.5.2 XML schema ..................................................... 27

1.6 Namespace .............................................................. 27

1.7 XPath ......................................................................... 28

1.8 XSL ........................................................................... 29

1.8.1 XSLT ............................................................... 29

1.8.2 XSL-FO ........................................................... 36

1.9 SOAP ......................................................................... 36

1.10 Other XML technology ............................................. 39

1.10.1 Xlink ................................................................ 39

1.10.2 XPointer .......................................................... 39

1.11 Examples of using XML ............................................. 40

1.11.1 SMIL ................................................................ 40

1.11.2 MathML ............................................................ 40
# Chapter 2. Lotus and XML

2.1 Domino as an XML data-store ........................................... 43  
   2.1.1 Domino and XML architecture .................................... 43  
   2.1.2 Domino forms ................................................... 44  
   2.1.3 Domino views ................................................... 45  
   2.1.4 Domino pages ................................................... 46  
   2.1.5 Domino agents .................................................. 48  
2.2 Domino XML Language (DXL) .......................................... 49  
   2.2.1 What is DXL .................................................... 49  
   2.2.2 How to use DXL ................................................ 54  
   2.2.3 Lotus XML toolkit .............................................. 57  
2.3 Lotus/IBM XML source .................................................. 62  
2.4 IBM XML Parser for Java .............................................. 62  
   2.4.1 DOM with XML Parser for Java .................................. 63  
   2.4.2 SAX with XML Parser for Java .................................. 66  
2.5 LotusXSL .............................................................. 68  
2.6 Security and XML ...................................................... 69  
   2.6.1 XML security for stored data .................................... 70  
   2.6.2 XML security as communication medium ....................... 72  

# Chapter 3. Getting XML out of Domino

3.1 Getting XML out of Domino forms .................................... 75  
   3.1.1 Applying XML to existing forms .................................. 76  
   3.1.2 Using the same document info for XML, Domino, HTML ......... 76  
   3.1.3 Tips & tricks for generating XML from Domino forms & fields .. 76  
3.2 Example: A simple XML form with tags embedded ................... 79  
3.3 Using LotusScript agents to generate XML ............................ 81  
3.4 Example: Storing computed XML data in a field on a document ...... 81  
   3.4.1 Generating computed document XML via an agent ............... 83  
3.5 Getting XML out of Domino Views .................................... 85  
   3.5.1 Using ?ReadViewEntries on existing views ...................... 86  
   3.5.2 Displaying multiple documents in an XML tree structure ....... 90  
   3.5.3 Adding XML tags into a view/page design ...................... 91  
   3.5.4 Applying a DTD ................................................ 92  
   3.5.5 Conforming to the Domino DTD ................................ 94  
   3.5.6 Conforming to other DTDs ...................................... 95  
3.6 Advantages of Domino for data integration with XML ............... 96  
   3.6.1 A Web server in the third tier .................................. 97  
   3.6.2 Using Domino as an XML store .................................. 98  
   3.6.3 The difference between data integration and XML ............. 98  
   3.6.4 Data integration: getting data out of storage ............... 98  
   3.6.5 XML: formatting retrieved data for sharing .................. 99  
   3.6.6 Options for data integration with Domino ..................... 99  

iv XML Powered by Domino
3.6.7 Serving XML-formatted documents with Domino forms & DECS
3.6.8 The DECS environment: administration database settings . . . . 100
3.6.9 The DECS application database: Formatting results . . . . . . . . . . 102
3.7 Other data integration options ........................................ 110
3.7.1 Data integration example using LCLSX .......................... 111
3.8 Java agents and servlets .................................................. 116
3.8.1 Using an agent to generate computed XML to a browser . . . . . 117
3.8.2 The GenerateXML Java method .................................... 122
3.8.3 Passing XML to an external XML file ............................ 122
3.8.4 Storing computed XML data in a field on a document .......... 124
3.8.5 Using a servlet to serve computed XML .......................... 127
3.9 Working with the Lotus XML Toolkit ................................. 135
3.9.1 Components ....................................................... 135
3.9.2 Features ..................................................................... 136
3.9.3 Java API Classes .................................................... 137
3.9.4 Developing a stand-alone Java XML application for Domino . . 140

Chapter 4. Transforming XML data in Domino ........................ 155
4.1 XSLT primer ................................................................. 155
4.1.1 LotusXSL and Xalan .................................................. 155
4.1.2 XSLT syntax ............................................................ 156
4.1.3 XSLT stylesheets ...................................................... 157
4.1.4 XSLT template model ............................................... 165
4.1.5 Returning node values from XML document .................... 166
4.1.6 Including additional text ........................................... 171
4.1.7 Sorting data with xsl:sort ......................................... 174
4.1.8 How to access attributes ........................................... 175
4.2 Transforming XML to HTML ........................................... 176
4.2.1 Including simple HTML markup .................................. 176
4.2.2 Including more complex HTML markup ......................... 180
4.2.3 Working with numeric data and xsl:number .................... 184
4.2.4 Procedural XSLT commands ....................................... 187
4.2.5 Storing XSLT stylesheets in Domino ............................ 187
4.2.6 Applying XSLT stylesheets in Domino .......................... 190
4.2.7 Transforming XML data on the server .......................... 191
4.2.8 Storing remote XML in a Domino document ................. 197
4.2.9 Transforming XML data from the browser client .......... 202
4.2.10 Using Domino Design elements to transform and display data 211
4.3 Transforming from XML to XML ........................................ 212
4.3.1 Transforming to an alternate XML format ....................... 213
4.3.2 Transforming elements to elements ................................ 222
4.3.3 Transforming attributes to elements ............................ 224
4.3.4 Transforming elements to attributes and attributes to attributes 225
<table>
<thead>
<tr>
<th>Figures</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. HTML data file viewed from a Web browser</td>
</tr>
<tr>
<td>2. Tree structure of XML document</td>
</tr>
<tr>
<td>3. Example of browsing well-formed XML on IE5.0</td>
</tr>
<tr>
<td>4. Example of browsing non well-formed XML on IE5.0</td>
</tr>
<tr>
<td>5. DOM interface hierarchy</td>
</tr>
<tr>
<td>6. SAX API processing model</td>
</tr>
<tr>
<td>7. DTD showing tree structure</td>
</tr>
<tr>
<td>8. HTTP task and XSLT processor</td>
</tr>
<tr>
<td>9. XSLT transforms XML into other formats</td>
</tr>
<tr>
<td>10. XML document browsed with XSL file</td>
</tr>
<tr>
<td>11. Simple example of SOAP on HTTP connection</td>
</tr>
<tr>
<td>12. Domino document properties</td>
</tr>
<tr>
<td>13. XML and Domino architecture comparison</td>
</tr>
<tr>
<td>14. Example of simple XML form</td>
</tr>
<tr>
<td>15. Simple XML view</td>
</tr>
<tr>
<td>16. Page with an embedded XML view</td>
</tr>
<tr>
<td>17. Naming a page as an external file</td>
</tr>
<tr>
<td>18. The parent-child relationship of View elements</td>
</tr>
<tr>
<td>20. Transforming DXL to XML and HMTL using XSLT</td>
</tr>
<tr>
<td>21. Getting XML into a Domino database via DXL</td>
</tr>
<tr>
<td>22. Changing Domino objects by modifying DXL</td>
</tr>
<tr>
<td>23. Sample Domino document to be transformed into DXL</td>
</tr>
<tr>
<td>24. How to find “noteid” in Domino document</td>
</tr>
<tr>
<td>25. Example of a Domino form which will be transformed into DXL</td>
</tr>
<tr>
<td>26. Setting for database encryption with server ID</td>
</tr>
<tr>
<td>27. Treat Documents as HTML enabled to pass XML to a browser</td>
</tr>
<tr>
<td>28. Poorly formed XML in IE5</td>
</tr>
<tr>
<td>29. Well formed XML in IE5</td>
</tr>
<tr>
<td>30. The basic XML form</td>
</tr>
<tr>
<td>31. LotusScript XML form</td>
</tr>
<tr>
<td>32. Text properties for fields on the LotusScript XML form</td>
</tr>
<tr>
<td>33. Properly formatted results from the LotusScript XML form</td>
</tr>
<tr>
<td>34. Basic customer list in Notes</td>
</tr>
<tr>
<td>35. Basic customer list view in Domino Designer</td>
</tr>
<tr>
<td>36. Basic customer list view embedded on a page</td>
</tr>
<tr>
<td>37. Microsoft IE XML parser error message for a missing domino.dtd</td>
</tr>
<tr>
<td>38. Internally validated XML in the Microsoft IE XML parser</td>
</tr>
<tr>
<td>39. Typical 2-tier Web architecture</td>
</tr>
<tr>
<td>40. Typical 3-tier Web architecture including a Domino server</td>
</tr>
</tbody>
</table>
41. DECS fish connection for the XML sample database .................. 101
42. Fish by fish name DECS activity ...................................... 102
43. DECSML.NSF fish query form ........................................... 103
44. Fish query form in Domino Designer with $\$\$Return code .......... 103
45. Query results view of the DECSML database .......................... 104
46. Fish ID search XML results .............................................. 106
47. XMLFishIDResults form in Domino Designer ......................... 107
48. XMLFishNameResults form in Domino Designer ...................... 108
49. XMLMultiResults form ................................................... 109
50. LCLSXXMLFishIDResults form with WebQueryOpen event highlighted. 112
51. JDBCXMLFishIDResults form with WebQueryOpen event highlighted. 117
52. JDBCXMLFishIDResults with an added body field .................. 125
53. HTML page that calls a servlet on the Domino server ............... 129
54. Sample Application example shown with exported DXL ............. 141
55. JDK 1.1.8 Window showing compile & run commands for GetFishDXL 142
56. Stylesheet primer form design ......................................... 157
57. Transform primer document ............................................. 162
58. Transform prompt dialog box ............................................ 163
59. Notepad run command .................................................... 163
60. XSLT results viewed in Microsoft Notepad ............................ 164
61. XSLT results stored in primer document ............................... 164
62. DOM representation of XML data ....................................... 166
63. DOM tree of directory data ............................................. 167
64. Output from FirstName stylesheets .................................... 169
65. Text output in Microsoft Notepad ...................................... 172
66. XML declaration included in output ................................... 172
67. Text output in Microsoft Wordpad ..................................... 173
68. Sorted Text output ...................................................... 175
69. Attribute value included in output .................................... 175
70. Transformed XML viewed in Web browser ............................. 179
71. HTML output in Web browser ........................................... 181
72. HTML table produced with XSLT stylesheet ........................... 184
73. Sorted numerical data ................................................... 186
74. Properly sorted numerical output ..................................... 187
75. Page design in Domino storing an XSLT stylesheet .................. 189
76. HTML output from the TransformNewsFeed agent ................... 197
77. Form design to hold raw XML data .................................... 198
78. Raw XML document viewed in a Web browser ........................ 199
79. Transformed XML data stored in Domino document ................ 200
80. Domino form design with XML data island .......................... 203
81. Domino document with a data island ................................ 205
82. Document with XML data island displayed in Internet Explorer ... 206
83. Output from HTML using Microsoft ActiveX objects .............. 207
84. HTML using Microsoft XMLDOM with sort buttons ................. 209
85. Sorted XML output .................................................. 210
86. Domino form with hard coded XML tags .............................. 240
87. Form design for storing XML data in a field ......................... 241
88. Document with raw XML stored in field ............................. 242
89. Directory structure for storing static files .......................... 244
90. 3-tier XML processing environment ................................. 245
91. Form for raw XML data .............................................. 247
92. Agent open in Domino Designer client ................................ 248
93. Completed XML document from agent ................................ 251
94. Agent output to Java debug console .................................. 255
95. Hierarchical DOM tree ............................................... 256
96. Exchange rates DOM tree .............................................. 257
97. Java console output .................................................. 260
98. Import exchange rates form design ................................... 262
99. Raw XML data view ................................................... 263
100. Text child node of an Element node ................................. 268
101. Converted exchange rate form design ................................ 269
# Tables

1. Entity reference ......................................................... 8  
2. Well-formed XML versus Valid XML .................................. 10  
3. Methods of Document Handler interface ................................ 18  
4. Common data types ....................................................... 26  
5. Default value ............................................................ 26  
6. Domino page versus file system ........................................ 48  
7. Elements used in View DTD ............................................ 50  
8. Elements used in Document DTD ........................................ 52  
9. Classes provided in Lotus XML Toolkit ................................ 61  
10. DOM interface summary .................................................. 63  
11. SAX interface summary .................................................. 66  
12. Fields in form design ..................................................... 158  
13. TransformPrimerAgent Parameters ...................................... 158  
14. xsl:output command and attribute values ............................ 171  
15. Conditional XSLT commands ............................................ 187  
16. Domino Design elements with XML .................................... 211  
17. Elements that represent actual content in a DXL document ....... 215  
18. Elements that represent Domino application fields in a DXL document ... 216  
19. Elements that represent Domino document properties in DXL document 216  
20. DXL element to XML element transformation .......................... 224  
21. DXL attribute to XML element transformation ........................ 225  
22. DXL element to XML element and attribute transformation ......... 226  
23. XML Java methods from the Domino 5 Designer Help database ...... 243  
24. Agents versus servlets .................................................... 246  
25. Field definitions for form design ....................................... 247  
26. Agent parameters ......................................................... 248  
27. Common Java methods to process XML data ........................... 252  
28. ParseRawXML agent parameters ....................................... 253  
29. Node type constants and integer equivalents .......................... 258  
30. ParseRawXML agent parameters ....................................... 263  
31. CreateExRateDocs agent parameters ................................... 269  
32. Summary .................................................................. 275
In the not-too-distant past, if you needed to use information stored on a mainframe or relational database system with an external system such as a spreadsheet application, you exported the data as a comma-delimited ASCII file, then crossed your fingers and hoped for the best. And how many times did you end up with the information splayed diagonally across the spreadsheet instead of being inserted neatly into rows and columns? And how many countless hours did you spend with a magnifying glass searching for the missing commas?

Or consider the following simple scenario 10 years ago: Your company entered into a joint marketing agreement with another company. One of the terms of the agreement is that both companies will swap customer mailing lists and as the marketing manager, you are responsible for providing your company's customer data to the other company. You would simply contact your database administrator and a week later, he would build you a cryptic SQL query command and after about 10 hours of churning, a set of records would pop out. Life was great...only a week and 10 hours to get the data. To send it to the other company, you simply print reams of green bar then they pay a data-entry clerk to re-key the information into their systems.

By today's standards, that scenario seems outrageously time-consuming, unsecure, and costly. Today, you can assume that the two companies' systems can be physically linked via the Internet over a secure connection, thus eliminating both the potential breach in security and the extra work introduced by the printed records. But merely connecting the systems doesn't ensure that the data can be imported into the receiving system. That's where XML enters the picture. It is a standard format for data interchange between disparate systems.

But how would XML be used with Notes and Domino data?

Many Lotus customers use Notes and Domino for process approvals routing. For example, most organizations require managerial approval for materials purchasing, employee time-off requests, corporate policy changes, and so forth. Consider the following procurement example: For a number of years, employees have used Lotus Notes to fill out purchase requisition forms. The forms are subsequently routed electronically to the employee's manager for approval. Upon approval, the requisition is closed and a purchase order is created. In the past, the purchase order form would be printed and sent to the vendor. Today, the purchase order form could be sent electronically to the vendor; and using XML, it could be formatted so that the vendor could
automatically import it into their order fulfillment system. This would streamline the order processing system by eliminating the need for order entry. As a result, data entry errors are reduced and order turnaround time is decreased, both of which lead to cost savings and improved customer satisfaction.

Another common use of Notes and Domino is for distributed authoring of content. For example, workers throughout an organization can use Notes to author and edit descriptions of products and/or services for a catalog. This information could then be transformed to XML, a standard format that could then be sent to the catalog publisher to be formatted for print. A technology called XSLT (discussed in this Redbook along with XML) is used to transform the XML into formatted information. Using XSLT, the information could be automatically formatted for a printed customer catalog, a wholesale catalog, and for the Website. Using XML and XSLT, a single version of the information can be easily repurposed for multiple delivery mediums.

These are just two examples of the many uses of XML in Notes and Domino business applications. Since XML is such a simple, yet powerful technology for enabling businesses to conduct business with each other, Lotus is committed to allowing developers to use XML and XML-related technologies in many ways with the Domino platform. In the chapters that follow, developers can learn the flexible options that are available today that enable Notes and Domino applications to be key components of the business applications that will drive the New Economy.

In this IBM Redbook we will help you understand how to make use of XML within a Lotus Domino environment. In this book, we do four things:

- Provide some basic background information on what XML is, its history, syntax, and related tools to get you started.
- Show how you can get information out of Domino in an XML format by using the tools you are already accustomed to, such as views, forms, pages, and agents.
- Once we have extracted XML data from Domino, we show you how to transform it into other formats, such as HTML, text, or another XML format.
- Finally, we show you how you can get data that is already in an XML format into Domino using agents and the Lotus XML toolkit.
Throughout this book we use the Microsoft Internet Explorer 5 browser to show XML. At the time of this writing, it was the only browser that could display and format XML; however, we noticed some problems, such as the browser hanging and giving incorrect error messages, between different versions of the browser. We recommend that you update your browser to the latest version of Internet Explorer 5.5 before trying the examples in this book.

The latest version is available from http://windowsupdate.microsoft.com/?IE

The team that wrote this redbook

This redbook was produced by a team of specialists from around the world working at the International Technical Support Organization Cambridge Center.

David Morrison is an International Technical Support Specialist for Lotus Notes and Domino at the International Technical Support Organization Center at Lotus Development, Cambridge, Massachusetts. He manages projects that produce redbooks on all areas of Lotus products. Before joining the ITSO in 1999, he was a senior Lotus Notes consultant working for IBM e-business services in the United Kingdom.

Brian Benz is CEO of Benz Technologies, a Lotus Advanced Business Partner and IBM Partner in Development specializing in Domino infrastructure, Web application development, and knowledge management systems consulting. Brian is also a writer on Web topics for industry publications and is often on the road as a seminar presenter for the VIEW and Advisor Media. He also recently assisted IBM in developing the certification requirements for the new IBM e-business Solution Developer and Solution Integrator partner certifications. Brian has been certified since December 1998 as an R5 CLP Principal Application Developer and R5 CLP Systems Administrator.

Paul Calhoun is the President and CEO of NetNotes Solutions Unlimited, Inc. located in Houston, Texas. He is a Certified Lotus Instructor, R5 Principle Certified Lotus Professional in application development, an R5 Certified Lotus Professional in Administration, a Microsoft Certified Trainer and a Microsoft Certified Systems Engineer. For the past 11 years he has facilitated the classroom delivery of Lotus- and Microsoft-certified curriculum to developers.
and administrators nationwide. When not in the classroom, Paul consults with
companies on the deployment of Domino servers and applications. He has
written several XML tutorials and taught XML programming for the past 18
months.

**Yusuke Murakami** is an IT specialist at IBM Japan Systems Engineering, a
subsidiary of IBM Japan. His work involves technical support and consulting
on Lotus Notes/Domino, mainly in system infrastructure design, system
administration, and Website design. He started with Notes/Domino four years
ago and has constructed Notes/Domino systems for many major Japanese
companies.

Thanks to the following people for their invaluable contributions to this project:

Laurent Pele of fxtop.com for the use of his foreign exchange rate data

Gary Devendorf, Chris Reckling, Beverly DeWitt, Janet Thomas, Joe
Kesselman, and David Marston of Lotus Development

The Lotus XSL team

The XML Toolkit team

Alison Chandler from the ITSO

---

**Comments welcome**

**Your comments are important to us!**

We want our Redbooks to be as helpful as possible. Please send us your
comments about this or other Redbooks in one of the following ways:

- Fax the evaluation form found in “IBM Redbooks review” on page 291 to
  the fax number shown on the form.
- Use the online evaluation form found at [ibm.com/redbooks](http://ibm.com/redbooks)
- Send your comments in an Internet note to redbook@us.ibm.com
Chapter 1. Introduction to XML

In this chapter, we explain the basics of XML. While this book is not designed to teach you everything about XML, we believe it's important to understand the basic concepts. In the following sections we describe what XML is (and isn't), what parsers are, schemas, namespaces, XSL, SOAP, and other XML-related technologies. Finally, we present some examples of how XML can be used in practical applications.

1.1 XML overview

XML (eXtensible Markup Language) is a meta language - a language used to write markup languages, which is used to describe data. There are two major merits to using XML. The first is that XML is written in a plain text format. This allows it to be compatible with existing computing environments. The second advantage is the extensibility of XML: developers can create their own markup tags, or elements, to best represent the structure and nature of the data. As you will see later in this chapter, when you define your XML documents you are actually defining a language to suit your application’s needs.

HTML is a markup language that is very useful for specifying how data should be displayed, whereas XML is very powerful for specifying the structure and context of data. XML excels as a format for describing data in a way that can be shared by multiple applications on many platforms. People, as well as computers, can understand the meaning of the data because the author can describe the data by defining each tag in terms that relate to the data structure. XML can be used as a universal data format, and for the exchange of information between systems on intranets or the Internet using Web browsers and Java. XML is beginning to play an important role in e-business and B2Bs as a universal data format.

1.1.1 XML and HTML

It's important to start off by mentioning that XML and HTML are not the same; and while they both have their roots in SGML, they are not even close to being the same. HTML is a specific markup language and is an application of SGML (or is supposed to be, if properly written). XML itself operates at the same level as SGML, not that of HTML. If you look at examples of XML and HTML, you will see that they appear to be quite similar. That is because the look of XML is similar to that of SGML, not because it was designed to look like HTML.
It is interesting to note that HTML was not originally designed to visually present information. It started as an abstract document markup language, breaking things up into paragraphs and similar conceptual units, which the browser was supposed to decide how to render visually. That's still the best way to use HTML, if you're concerned about accessibility issues. Since then, HTML has become massively overloaded with visual rendering specific features, but these are an accretion upon what is, essentially, a very simple abstract description of the contents of a page.

XML itself only describes the structure of the data. But XML-based languages may be every bit as visually-specific as HTML. XHTML (see 1.3, “XHTML” on page 10) is an obvious case since it is HTML re-expressed in XML-compatible syntax, while XSL-FO (described in 1.8.2, “XSL-FO” on page 36) plays into the role HTML was originally intended to occupy in that it has an abstract page structure and is rendered to meet the needs of specific environments and users. XML facilitates the separation of data structure and rendering description, but that's in large part because XML is itself neutral, which allows tools to span that gap without having to step outside the XML boundaries.

While the look of XML and HTML can be very similar, there are many big differences, especially in the handling of data. HTML is a markup language designed to visually present information to a user from a Web browser, while XML is designed to describe the structure of the data.

The following is an HTML sample.

```html
<HTML>
<HEAD>
<TITLE>Information</TITLE>
</HEAD>
<BODY>
<H2>Customer ID Search Results</H2>
<TABLE border=1>
<TR><TD>CustomerID</TD><TD>0000002150</TD></TR>
<TR><TD>Last Name:</TD><TD>SMITH</TD></TR>
<TR><TD>First Name</TD><TD>AUBREY</TD></TR>
<TR><TD>Company</TD><TD>HILLSIDE DR</TD></TR>
<TR><TD>Address</TD><TD>PO BOX 2134</TD></TR>
<TR><TD>Zip</TD><TD>75034</TD></TR>
<TR><TD>Zone</TD><TD>Old Town</TD></TR>
</TABLE>
</BODY>
</HTML>
```
Figure 1 shows this HTML when viewed in a Web browser.

![Customer ID Search Results](image)

| CustomerID | 000002150 |
| Last Name  | SMITH     |
| First Name | AUBREY    |
| Company    | HILLSIDE DR |
| Address    | PO BOX 2134 |
| Zip        | 75034     |
| Zone       | Old Town  |

As this example shows, it would be very hard to use any data from the HTML source code. Even though lots of data is included in the HTML, the application programs do not know how the data is defined. So, HTML is good at providing data for people but not for application programs.

Now let's look at a simple example of XML.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<Customer>
  <ID>000002150</ID>
  <Lastname>SMITH</Lastname>
  <Firstname>AUBREY</Firstname>
  <Company>HILLSIDE DR</Company>
  <Address>PO BOX 2134</Address>
  <Zip>75034</Zip>
  <Zone>Old Town</Zone>
</Customer>
```

This time, there are tags enclosing each data item. It would be easy to use this data from any application program that knows the name of the tags. In this case, we do not care that this is not browsable data. Application programs would be able to interpret this data even though it is not suitable for browsing. In order to present this data to a web browser, we would need to transform it from XML into something better suited for reading via a browser.
1.1.2 Why use XML

As we’ve discussed, XML is a language for describing data. We should also ask the question, why use XML? There are two main reasons for using XML, especially with Domino:

- For data exchange
- To present to a browser client

As we mentioned previously, XML is a universal data format. If any system needs to exchange data with another system, there must be an agreed upon common data format. Using XML, any system can use data directly without any data format conversion.

XML does not include any formatting information for the purpose of display by a browser client. However, by using extensible style sheet transformation (XSLT) technology, we can transform any XML data format for presentation on any browser platform. The browser only needs to support XML and XSLT. Using the XSLT technology has many merits because we can have many ways to present a single data source. We will discuss XSLT later.

1.1.3 Connecting with XML

The real value of XML technology is realized when data is shared among various systems and applications. If you think in terms of what XML could be used for, for instance in knowledge management where data is passed to and from many disparate systems, the potential is enormous. Since XML is a universal data format, the best way to use it is to share data between systems. There are two types of connections using XML:

1. Connecting server to server
2. Connecting server to client

Since we can use XML as a data format, it is often used in connecting data from server to server. As for exchanging data, it is sometimes used to transfer...
data between servers for batch processing — for example, transferring with FTP. The other use for XML is in referring to data in real time. In this case, each application program directly handles XML between other. For example, an application program can access XML data from data storage and do some processing on it. The main transport technologies used to connect servers are FTP, HTTP, SMTP and IBM MQ series.

The other way of using XML is for connecting client to server, or server to client. Since XML does not contain the information necessary to display or present itself, we need to use XSL with XML. In this case we need to use clients which can view XML and XSL, or introduce middleware that uses XSL to “style” the data.

### 1.2 XML document structure

Using XML, we can describe many types of data which we are used to handling in a Notes database or relational database. Although it is up to us what we call the tags used in an XML document, there are many syntax rules that must be strictly followed. First, we will briefly explain the structure of this syntax.

#### 1.2.1 XML syntax

Let’s start with the following simple example of XML.

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE doc SYSTEM "doc.dtd">
<doc>
  <title>This is the most simple XML</title>
</doc>
```

An XML document consists of three parts.
1. XML declaration
2. An optional DTD reference
3. XML entities

The XML declaration does the following things:
- Declares that the document type is XML
- Specifies the XML version
- Specifies the character encoding
- Indicates whether this document is logically complete or references an external entity
The current version of XML is “1.0”; it is declared in the following part:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
```

In this example, encoding style is set as “UTF-8”. This is one of the good points to using XML as a universal data format: XML can handle unicode character encoding. XML authors must pay attention to the fact that every XML document must be written using the same encoding within the declaration. Although we used “UTF-8” in this example, we can also use other encodings, like “UTF-16” or any other encoding style. The last attribute, “standalone,” defines the relationship of this XML document with other external files like DTDs or XSLs. If we set “yes” in this attribute, this XML document will not refer to any other external files. When we omit this attribute, the value defaults to “no.”

The next part after the declaration is a DTD part. “DTD” means Document Type Definition; it is used to define the structure of an XML document. DTD is defined in either of the following two ways: include the DTD in the XML document itself, or create it in an external file and point to it from the XML document. We call the first the internal DTD and the second is the external DTD. The following is an example of placing the DTD outside of the XML document. (In this case, DTD is another file named “doc.dtd”)

```xml
<!DOCTYPE doc SYSTEM "doc.dtd">
```

In this example, the XML document has the “doc” element as a root element and will use “doc.dtd” to validate the data. “SYSTEM” means that this DTD is defined by a non-public entity. If the DTD used in an XML document is a public one, we would use “PUBLIC” in this declaration.

The last part, XML entities, contains the real body of the XML document. All the data elements are defined here. To define this data, we use elements and attributes in XML entities.

An element is a unit of data. Every element consists of a “start tag,” “contents,” and “end tag.” The following sentence is an example of an element.

```xml
<Redbook>XML powered by Domino</Redbook>
```

In an XML document, we can add some additional information to this element as an attribute. Attributes must be written inside of the start tag. An example of using attributes is as follows:

```xml
<Redbook BookID="LO-0053-R">XML powered by Domino</Redbook>
```
1.2.2 Syntax rules

Unlike HTML, we can use any names for these elements and attributes that we choose. However, XML documents need a very strict syntax structure. The following are the key points of the syntax structure.

- Every start tag needs to have a counterpart end tag. The end tag is the same as the start tag with a / character before it. For example, if you open an element with `<document>`, you must put `</document>` at the end of that element.

- Capital and small letters are regarded as different. Case matters.
  For example, the following is incorrect syntax:
  
  ```xml```
  <Message> Hello World! </message>
  ```xml```

- The XML tree structure must be nested perfectly.
  For example, the following is incorrect:
  
  ```xml```
  <Tree><NestedTree> leaf </Tree></NestedTree>
  ```xml```
  
  This must be defined as follows.
  
  ```xml```
  <Tree>
   <NestedTree> leaf </NestedTree>
  </Tree>
  ```xml```

- There can be only one root element in an XML document.
  For example, the `<doc>` tag is the root element in the following:
  
  ```xml```
  <?xml version="1.0" encoding="UTF-8" standalone="no" ?>
  <!DOCTYPE doc SYSTEM "doc.dtd">
  <doc>
   <title>This is the most simple XML</title>
  </doc>
  ```xml```

  So, the following document is incorrect because there are two root elements in a single XML document:
  
  ```xml```
  <?xml version="1.0" encoding="UTF-8" standalone="no" ?>
  <!DOCTYPE doc SYSTEM "doc.dtd">
  <doc>
   <title>doc for one</title>
  </doc>
  <doc>
   <title>doc for two</title>
  </doc>
  ```xml```

- Empty elements are allowed.
  For example, `<element></element>` is allowed. We can also write this as `<element />`. 
- Each element can have any number of attributes.

The following is an example of using an attribute:

```xml
<employee EmployeeID="00125">Tetsuya Miwa</employee>
```

All attributes must be enclosed with double quotes ("" or single quotes (''). Therefore, all attributes are ultimately strings. So, the following example is incorrect:

```xml
<employee EmployeeID=00125>Tetsuya Miwa</employee>
```

- Comment statements are allowed using the following tags “<!-- “-->”.

The following is an example of a comment.

```xml
<!-- This is a sample for comment. We must close the comment tag -->
```

- Some predefined entities are needed to replace certain characters.

This is just the same as HTML. For example, we cannot use `<` or `>` because these characters are predefined to describe tags. To solve this problem, we use entity references. The following table shows the entity references.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Replace with</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;</code></td>
<td>&lt;</td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>&gt;</td>
</tr>
<tr>
<td><code>&amp;</code></td>
<td>&amp;</td>
</tr>
<tr>
<td><code>'</code></td>
<td>'</td>
</tr>
<tr>
<td><code>*</code></td>
<td>&quot;</td>
</tr>
</tbody>
</table>

### 1.2.3 Well-formed XML documents

Every XML document must satisfy all of these syntax requirements. XML documents which satisfy the syntax rules are referred to as well-formed XML documents.

The following example is a more complex XML document.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE customerlist SYSTEM "sample.dtd">
<customerlist>
  <customer customerID="00123">
    <name>
      <last>smith</last>
      <first>aubrey</first>
    </name>
  </customer>
</customerlist>
```
This is a well-formed XML document. Every start tag has a counterpart endtag and all the elements compose a tree structure. The "customerlist" entry is the root element for this document and attributes are also described properly.

Figure 2 makes it clear that the example XML document is composed as a tree structure.

![Figure 2. Tree structure of XML document](image)

The preceding rules cover most of the basic syntax of XML. For more information about XML syntax, refer to the specification documents from the W3C (the World Wide Web Consortium is the organization that develops all the major Internet standards) at:

http://www.w3.org/TR/REC-xml
1.2.4 Well-formed and valid XML documents

In the previous section, we explained that XML must conform to a strong set of syntax rules. These XML documents can be classified into two categories. The first category is “Well-formed XML documents” and the other is “Valid XML documents.” Actually, every XML document must be a well-formed XML document. Well-formed means the XML document conforms to the XML syntax rules. Well-formed documents do not always need a DTD.

On the other hand, Valid XML documents must have a DTD and must strongly adhere to the structure defined in the DTD. Table 2 summarizes the difference between these two categories.

Table 2. Well-formed XML versus Valid XML

<table>
<thead>
<tr>
<th>Well-formed XML</th>
<th>Valid XML</th>
</tr>
</thead>
<tbody>
<tr>
<td>Must adhere to XML syntax guidelines</td>
<td>Must adhere to XML syntax guidelines</td>
</tr>
<tr>
<td>DTD is not required</td>
<td>DTDs are required</td>
</tr>
<tr>
<td>No defined structure</td>
<td>Must adhere to DTD structure</td>
</tr>
<tr>
<td></td>
<td>Must be well-formed</td>
</tr>
</tbody>
</table>

When we use XML Documents to transfer data between client and server, we usually are only concerned that the document is well-formed. On the other hand, when we use XML for exchanging data between two servers, using valid XML is very important as it enables us to verify that the document we are receiving is in the correct format before we process it.

1.3 XHTML

XML is often compared with HTML, but it’s important to note that XML is not an extension of HTML. As we mentioned at the beginning of this chapter, XML and HTML are quite different in their purpose. HTML is used for presentation and XML is used for describing data.

XHTML (eXtensible Hyper Text Markup Language) is the real extension of HTML.

XHTML is actually an extension of HTML4.0; it is HTML using XML syntax. The main differences between HTML4.0 and XHTML are the following:

- HTML elements and attributes are written in lower case characters.
- Every attribute must be enclosed in double quotes (""") or single quotes ("').
- The author cannot omit the end tag.
• The document must be in a tree structure (the tags must be strictly nested).

For example, we could write HTML the following way:

```html
<UL>
  <LI>List1
  <LI>List2
</UL>
```

This is not allowed in XHTML. The following example shows the correct XHTML:

```html
<ul>
  <li>List1</li>
  <li>List2</li>
</ul>
```

XHTML has the following three types of DTDs. Every XHTML document must use one of these DTDs.

- XHTML Transitional
- XHTML Strict
- XHTML Frameset

"XHTML Transitional" is the closest DTD to the HTML DTD. This DTD can make the most of XHTML functionality including style sheet and also make it easy to transit from the existing HTML. "XHTML Strict" is the strict DTD. We need to use W3C CSS (Cascading Style Sheets) with this DTD. "XHTML Frameset" is a DTD for using frameset.

An XHTML document must meet the following requirements:

- One of the three DTDs mentioned previously must be used.
- The root element of the document is `<html>`.
- XHTML namespaces must be declared in a root element.
- DOCTYPE declaration must be done along its DTD.

The following is a simple example of XHTML:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html
  PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <title>XHTML sample</title>
</head>
```
For more information about XHTML, refer to the specification documents from the W3C.

http://www.w3.org/TR/xhtml1/

1.4 XML parser

Understanding the XML parser is very important. Although XML syntax will let us describe data in a plain text format and can be used as a universal data format, it is still just plain text data. An XML parser will help us use this data directly. An XML parser will parse a plain text XML document into an in-memory format that can be manipulated programmatically via an API.

An XML parser is sometimes called an “XML processor.” The Domino product includes an XML parser, IBM’s XML for Java parser. IBM’s XML4J parser is primarily used with server-based applications. Microsoft Internet Explorer 5 also includes an XML parser, used to parse XML data that it accesses.

In this section we review these XML parsers.

1.4.1 Parsing XML

Each XML document must be verified as conforming to XML syntax guidelines. XML parsers verify the document syntax. An XML parser must check each XML document for well-formed syntax and alternately check to ensure that it is valid.

Let’s see an example of well-formed parsing. Microsoft Internet Explorer 5.0 can verify the XML syntax and display the XML code if the document is well-formed.

<?xml version="1.0" encoding="UTF-8" ?>
<Customer>
  <CustomerID>0000002150</CustomerID>
  <Lastname>SMITH</Lastname>
  <Firstname>AUBREY</Firstname>
  <Company>HILLSIDE DR</Company>
  <Address>PO BOX 2134</Address>
  <Zip>75034</Zip>
  <Zone>Old Town</Zone>
</Customer>
Figure 3 shows this XML document when viewed with Microsoft Internet Explorer version 5 and above.

If you can see the document displayed in the IE browser, it means at least the XML document is well-formed. On the other hand, let’s try sending an XML document with incorrect syntax to IE5.0. The following XML document is incorrect and not a well-formed document.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<Customer>
    <CustomerID>000002150</CustomerID>
    <Lastname>SMITH</Lastname>
    <Firstname>AUBREY</Firstname>
    <Company>HILLSIDE DR</Company>
    <Address>PO BOX 2134</Address>
    <Zip>75034</Zip>
    <Zone>Old Town</Zone>
</Customer>
```

When we try this XML document in the E5.0 browser we get the results shown in Figure 4 on page 14.
Figure 4. Example of browsing non well-formed XML on IE5.0

As this example shows, an XML parser can check for “well-formedness.” This example is checking just for its well-formedness. Some XML parsers can also check for the validity of a document against a DTD.

An XML parser also provides a structured interface to access the values of the elements and attributes of an XML document. There are two major APIs to handle XML objects.

1.4.2 APIs for document handling

An XML parser makes it possible to validate the XML document. Then, the next step is to access the data structure of XML document. XML parsers provide APIs to handle the XML data. There are two major APIs for XML:

- Document Object Model (DOM)
- Simple API for XML (SAX)

We usually use the Java programming language to access these APIs, especially in Notes/Domino. In Notes/Domino, we can use a JavaAgent or a JavaServlet on a Domino server. The Domino server has a parser for Java.
1.4.3 Document Object Model

Not to be confused with the Domino Object Model, the document object model is an API to access an XML data structure. Using DOM, we can access XML data represented as a tree object model. In DOM programming, every element is handled as a “node,” which is defined as an interface in the DOM.

DOM creates a tree of the entire XML document in memory and can access and control any of these objects using the DOM API. Sometimes DOM is not the best API to use when handling a large document since it holds the whole object tree on memory.

DOM is a set of interfaces which are mainly composed of nodes. There is a hierarchy between these interfaces. The following figure shows the hierarchy of the interfaces.

![DOM Interface Hierarchy Diagram]

As this figure shows, node object is the primary data type in this model. Using the DOM API, we can handle node objects in many ways. We introduce some of the methods which are frequently used in DOM programming.

There are two ways of handling node objects:

- Referring and setting a value to each node object
For the first way, referring and setting value to each node object, we often use the following four methods:

- **getNodeName()**
  This method returns the name of a node. The name depends on the type of node it is. Sometimes it is an element name and sometimes it is an attribute name.

- **getNodeValue()**
  This method returns the value of a node. The value depends on the type of node it is.

- **getNodeType()**
  This method returns the type of a node.

- **setNodeValue(arg)**
  This method sets a value to the node.

For the second way, accessing tree structure with node object, we often use the following methods:

- **getParentNode()**
  This method gets the parent node.

- **getPreviousSibling(), getNextSibling()**
  These methods get the same tree level of node.

- **getFirstChild(), getLastChild(), getChildNode(), getElementsByTagName()**
  These methods get children nodes.

- **appendChild(), removeChild(), replaceChild()**
  These methods are used for adding and deleting a child node.

In this way, we can control XML documents easily with the DOM API. The following code is a simple example of a Java program that uses the DOM API in a Notes Java Agent.

```java
import lotus.domino.*;
import org.w3c.dom.*;

public class JavaAgent extends AgentBase {
    public void NotesMain() {
        int i;
```
try {
    Session ns = getSession();
    AgentContext ac = ns.getAgentContext();
    lotus.domino.Document doc = ac.getDocumentContext();
    Item rawXML = doc.getFirstItem("XMLDATA");
    org.w3c.dom.Document xDoc = rawXML.parseXML(false);
    Element el = xDoc.getDocumentElement();

    //Using DOM API
    String rootTag = el.getTagName();
    System.out.println("The Root Element is " + rootTag);
    NodeList nl = xDoc.getElementsByTagName(rootTag);
    System.out.println("There is " + nl.getLength() + " node in the Root Node List");
    Node n = nl.item(0);
    nl = n.getChildNodes();
    System.out.println("The " + rootTag + " Root Tag has " + nl.getLength() + " child nodes");
    for ( i = 0 ; i < 10 ; i++) {
        n = nl.item(i);
        if (n.getNodeType() == n.ELEMENT_NODE) {
            System.out.println("Node Name is:" + n.getNodeName());
            System.out.println("Node Value is:" + n.getNodeValue());
            System.out.println("Node Value is:" + n.getNodeType());
            System.out.println("This node has child nodes:" + n.hasChildNodes());
        }
    }
    } catch(Exception e) {
    e.printStackTrace();
}

We will explain this agent in a later chapter. But let's have a look at this program. We can control the node with some methods, for example getChildNode, getNodeType, and GetNodeName.

For more information about DOM, refer to the specification documents from the W3C.

http://www.w3.org/DOM/

1.4.4 Simple API for XML (SAX)

Another API for handling XML documents is Simple API for XML (SAX). While DOM is the current W3C recommendation, SAX was developed by the people
on the xml-dev mailing list on the Web. SAX was originally written by this
group because they wanted a simple and lightweight API for processing XML
documents. SAX is becoming the de-facto standard for server to server XML
processing.

SAX does not create an object tree; instead, it is an event-driven lightweight
API. The XML document is processed just once, passing each element event
to the event handler. Every application which uses SAX must register SAX
event handlers to a parser object. If we write this in Java code, the following
shows this registration.

```java
parser.setDocumentHandler(new myDochandler() );
```

SAX provides three handler interfaces:

- DocumentHandler
- DTDHandler
- ErrorHandler

The most important interface is “DocumentHandler,” which handles elements.
The other two are handlers for DTD and XML errors.

Table 3 shows methods of the Document Handler interface.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>startDocument()</td>
<td>Receive notification of the beginning of the document</td>
</tr>
<tr>
<td>endDocument()</td>
<td>Receive notification of the end of the document</td>
</tr>
<tr>
<td>startElement(String name, AttributeList atts)</td>
<td>Receive notification of the beginning of an element</td>
</tr>
<tr>
<td>endElement(String name)</td>
<td>Receive notification of the end of an element</td>
</tr>
<tr>
<td>character(char ch[], int start, int length)</td>
<td>Receive notification of character data</td>
</tr>
<tr>
<td>ignorableWhitespace(char ch[], int start, int length)</td>
<td>Receive notification of ignorable white space</td>
</tr>
<tr>
<td>processingInstruction(String target, String data)</td>
<td>Receive notification of a processing instruction</td>
</tr>
<tr>
<td>setDocumentLocator(Locator locator)</td>
<td>Receive an object for locating the origin of SAX document events</td>
</tr>
</tbody>
</table>
The following code is a simple example of a SAX application that is used in a Notes/Domino application as a Java Agent.

```java
import org.xml.sax.*;
import org.xml.sax.Parser;
import lotus.domino.*;
import org.xml.sax.helpers.ParserFactory;
import java.io.*;

public class JavaAgent extends AgentBase {

    public void NotesMain() {
        int i;
        try {
            Session ns = getSession();
            AgentContext ac = ns.getAgentContext();
            lotus.domino.Document doc = ac.getDocumentContext();
            Item rawXML = doc.getFirstItem("XMLDATA");
            Reader r = rawXML.getReader();
            //The following line of code implements the parser directly
            //com.ibm.xml.parsers.SAXParser p = new
            com.ibm.xml.parsers.SAXParser();
            //or us this to implement the parser through ParserFactory
            Parser p =
            ParserFactory.makeParser("com.ibm.xml.parsers.SAXParser");
            p.setDocumentHandler(new myDochandler() );
            p.parse(new InputSource(r));

        } catch(Exception e) {
            e.printStackTrace();
        }
    }

    public class myDochandler extends HandlerBase {

        public void startDocument() {
            System.out.println("Begining of Document");
        }

        public void endDocument() {
            System.out.println("End of Document");
        }
    }

    myDochandler.java
import org.xml.sax.*;

public class myDochandler extends HandlerBase {

    public void startDocument() {
        System.out.println("Begining of Document");
    }

    public void endDocument() {
        System.out.println("End of Document");
    }
}
```

Chapter 1. Introduction to XML 19
This program will parse the XML document and print the result of its parsing with SAX events. For example, we show the sample data and its result.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<Customer>
  <CustomerID>0000002150</CustomerID>
  <Lastname>SMITH</Lastname>
  <Firstname>AUBREY</Firstname>
  <Company>HILLSIDE DR</Company>
  <Address>PO BOX 2134</Address>
  <Zip>75034</Zip>
  <Zone>Old Town</Zone>
</Customer>
```

The result of this document is the following:

```
Begining of Document
Start of Element - Customer
Start of Element - CustomerID
0000002150
End of Element - CustomerID
Start of Element - Lastname
SMITH
End of Element - Lastname
Start of Element - Firstname
AUBREY
End of Element - Firstname
```
1.5 Designing an XML data structure

XML documents can express data with many different structures. If the data is used in just one system, it does not always need to be defined with strict structure. But if the data is used for many purposes and on many systems, we need to agree on a strict data structure.

This section explains these the following data structures: DTD and XML schema.
1.5.1 DTD

DTD (Document Type Definition) defines the data structure of each XML document. It defines the names of elements and attributes, the tree structure of the data, the number of elements, and any rules related to data structure.

XML documents do not always need this DTD. Non-valid XML documents do not need a DTD to describe XML. On the other hand, valid XML documents need a DTD and must conform to its definition. XML documents which are not described along with the DTD definition are not valid XML documents; even DTD is defined as an XML document. In the real world, we usually use DTD when we exchange data with other systems, including Domino systems.

We do not always need to write a DTD itself in an XML document; the DTD can be written either inside or outside an XML document. Describing the DTD within the XML document means the DTD can be modified only for the specific document. It is more useful to use an external DTD if we want it to be accessible as a public DTD.

In the DTD, we declare the following things:

1. Every "element"
2. Every “attribute”
3. Every “entity”
4. Every “notation”

Let’s look at an example. The following is an example of an external DTD.

```xml
<!ELEMENT customerlist (customer)+>
<!ELEMENT customer (name,company)>
<!ATTLIST customerID type CDATA #REQUIRED>
<!ELEMENT name (last,first)>
<!ELEMENT last (#PCDATA)>
<!ELEMENT first (#PCDATA)>
<!ELEMENT company (#PCDATA)>
```

We save this DTD in a file, for example “sample.dtd,” because this is an external DTD.

An example of an internal DTD is the following.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE customerlist [ 
<!ELEMENT customerlist (customer)+>
<!ELEMENT customer (name,company)>
<!ATTLIST CustomerID type CDATA #REQUIRED> 
```
This DTD defines the tree structure of the data. It is illustrated in Figure 7.

![Diagram of DTD tree structure]

Figure 7. DTD showing tree structure

With this DTD, we can describe the data in the following way.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE customerlist SYSTEM "sample.dtd">
<customerlist>
  <customer CustomerID="00123">
    <name>
      <last>smith</last>
      <first>aubrey</first>
    </name>
  </customer>
  <company>HILLSIDE DR</company>
</customerlist>
```

When declaring elements, we define the name of the element, the parent-child relationship of the element, and so on. The basic structure of declaration is:

```xml
<!ELEMENT A (b,c)>
```

In this declaration, element A is a parent of b and c. The b and c must appear in the same order that they are declared in. With this sentence, we can define A's name and its children.
The simple declaration of a child is:

```xml
<!ELEMENT b (#PCDATA)>
```

(#PCDATA) means b is parsed character data. With these two basic syntax elements we can describe a data structure. A simple example of using this element declaration is the following:

```xml
<!ELEMENT name (last,first)>
<!ELEMENT last (#PCDATA)>
<!ELEMENT first (#PCDATA)>
```

A example XML document using this DTD is:

```xml
<name>
  <last>Miwa</last>
  <first>Tetsuya</first>
</name>
```

While this example is quite simple, DTDs can be used to define much more complex structures. In the previous definition between parents and children, the children must have appeared in an XML document because “,” means “and”. We can use “|” instead of “,” and in this case, one of the children will appear in its XML document. The following is a simple example of using “|”.

```xml
<!ELEMENT language (English|French)>
<!ELEMENT English (#PCDATA)>
<!ELEMENT French (#PCDATA)>
```

The data using this DTD is described in the following way. As you can see, we do not write both “English” and “French” elements, but either of these.

```xml
<language>
  <English>Hello world!</English>
</language>
```

The DTD can also control the number of times an element appears. In this definition, we use “*”, “+”, and “?”. The “*” character indicates that this element will appear zero or more times; “+” indicates that this element will appear one or more times; and the “?” character indicates that this element will appear zero or one times. The following XML document is an example of the syntax, using “*” in DTD.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE customerlist [
  <!ELEMENT customerlist (customer)*>]
<!ELEMENT customer (#PCDATA)>
]>
<customerlist>
Now we can define the data which has complex structure using the DTD element syntax. The following is a sample which is uses multiple syntax elements of DTD.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE customerlist [
<!ELEMENT customerlist (customer)>]
<!ELEMENT customer (name, company?, call)>  
<!ELEMENT name (#PCDATA)>  
<!ELEMENT company (#PCDATA)>  
<!ELEMENT call (#PCDATA)>  
]>
<customerlist>
  <customer>
    <name>Tetsuya Miwa</name>
    <company>TM Fish</company>
    <call>1-999-xxx-xooo</call>
    <call>1-999-xxx-xoxo</call>
  </customer>
  <customer>
    <name>Alexei White</name>
    <call>1-999-xxx-yyyy</call>
  </customer>
</customerlist>
```

In this XML document, we used more complex DTD syntax. Let's see the detail of this code.

```xml
<!ELEMENT customerlist (customer)>*
```

This line shows “customerlist” has a child element named “customer,” and this “customer” can appear in “customerlist” zero or more times. In this document, there are two “customer” elements. But we can even allow a document which does not have any “customer” element.

```xml
<!ELEMENT customer (name, company?, call)>  
```

This line means the element “customer” has three children: “name,” “company,” and “call,” and the element “company” can appear zero or one time. The other two elements must appear once, and these three elements must appear in this order. In this XML document, the first customer data has a
company element but the second one does not have this element. In this way, we can define lots of elements with DTD.

Attributes make it possible to put additional data in elements. The simple declaration of an attribute is like the following:

```xml
<!ATTLIST element attribute CTYPE DefaultValue>
```

This defines the attribute of an element, the name of the attribute, its data type and the default value. In this sample, CDATA means type text. We can use a lot of types and several kinds of default values. The following table shows some common data types.

<table>
<thead>
<tr>
<th>Data type</th>
<th>Explanation of type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDATA</td>
<td>Plain text strings</td>
</tr>
<tr>
<td>ID</td>
<td>Identification</td>
</tr>
<tr>
<td>enumerated</td>
<td>Defined multiple value</td>
</tr>
</tbody>
</table>

An example of an enumerated attribute is the following:

```xml
<!ATTLIST dinner drink (coffee|tea) "coffee">
```

In attribute declaration, we can set the default value. We can write the default value directly to this declaration or we can give this attribute just the value information, as shown in Table 5.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Explanation of keyword</th>
</tr>
</thead>
<tbody>
<tr>
<td>#REQUIRED</td>
<td>The attribute is required</td>
</tr>
<tr>
<td>#IMPLIED</td>
<td>The attribute is optional</td>
</tr>
<tr>
<td>#FIXED</td>
<td>The attribute is fixed</td>
</tr>
</tbody>
</table>

Entities are also an important concept in XML. An entity is basically a substitute string for the original; the concept is very similar to the “Const statement” in lotusscript. We can use both internal and external entities, just as we did with DTD files. These entities will be replaced using the original string by the XML parser. The following is an example of an entity.

```xml
<!ENTITY cam "cambridge">
```

We can use this entity in an XML document. The following is a sample of using the previous entity example. We use entity in an XML document with “&”
We can also use this entity as an external entity. Following is a simple example. This “System” means just the same as what we specified in the DTD declaration in XML documents.

```xml
<!ENTITY cam SYSTEM "system.xml">
```

Notation declarations define the data type of an external entity. We use this declaration when we reference binary files. After the definition of this notation, an XML parser can properly handle this data as a binary file. The following is a simple example of notation.

```xml
<!NOTATION BMP SYSTEM "Bitmap File">
<!ENTITY pic1 SYSTEM "pic1.bmp" NDATA "BMP">
```

### 1.5.2 XML schema

DTD is the current standard to define XML document structure. But DTDs have a few known limitations. There are two issues when using DTDs:

1. DTDs are not written using XML document syntax.
2. DTDs can handle only “text” data type.

The syntax of DTDs is different than the syntax of XML. This difference makes for two problems. The first is every XML programmer has to learn both of these syntaxes.

The other problem is data type. DTDs can only handle text data, described as “PCDATA.” It would be better if DTD could handle other types of data, for example, string, boolean, real, integer, date, time, and so on.

These limitations make other schema look much better. There are some schemas which can substitute for DTD. The most likely successor to DTD will be XML schema. The W3 has issued a call for recommendations. The status of the recommendations can be monitored at the W3C Web site.

### 1.6 Namespace

Tag names in XML are not limited to a predefined set. This can sometimes cause trouble. For example, a very common noun like “name” or “title” is an obvious name for an element; but such common words have some likelihood of being used in other XML documents as well, and quite possibly with different DTDs. This may cause problems since the system cannot distinguish which DTD the name is referencing. This idea is similar to field name conflict in Notes/Domino. Application developers can encounter problems with
handling data from other applications when the field names are exactly the same as what exists in the current system.

In the XML world, XML Namespaces will solve this problem. We can discriminate between DTDs by using XML namespace.

Namespace declaration is done in the following way.

```xml
xmlns:namespace = "URI"
```

We can declare a namespace name and make the URI (uniform resource identifier) the place pointed to for the DTD. The URI includes both the URL (uniform resource locator) and URN (uniform resource name). A URI is a string which is globally unique and which uses numbers and strings. At this time, a URI is considered almost the same as a URL. The detailed definition is in RFC 2396.

The following is an example XML document which uses namespaces.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<customer xmlns:CM="http://fish.net">
  <CM:customer>
    <CM:name>Aubrey Smith</CM:name>
    <CM:company>HILLSIDE DR</CM:company>
  </CM:customer>
</customer>
```

In this example, all elements related to “customer” are in the CM namespace and there is no conflict with other XML documents which also use the same name tag.

For more information about namespaces in XML, refer to the specification documents from the W3C.

http://www.w3.org/TR/REC-xml-names/

1.7 XPath

Before we discuss XSL, which is one of the most important technologies related to XML, we should mention XPath (XML Path Language). XPath technology is used in XSLT and XPointer.

XPath is a language for addressing parts of an XML document. To address XML data, XPath also provides the basic facilities for traversal. The concept behind XPath is to handle an XML document as a node tree. We trace this tree structure from the root tag of each XML document.
The XPath syntax is made up of expressions. After evaluation, these expressions create an object which is classified into one of the following basic types:

- Node-set
- Boolean
- Number
- String

XPath is often used as a location path. To describe the location, there is a lot of syntax in the location path. For example “child::para” means “select the para element children of the context node.” We can also use abbreviated syntax using this location path. In this abbreviated syntax, we can write just “para” in stead of writing “child::para.” Abbreviated syntax is quite simple and easy to understand. The way to handle tree structure is with “/”.

In addition to the location path, we can use lots of functions in XPath. These functions are also classified into four types:

- Node-set function
- Boolean function
- Number function
- String function

For more information about XPath, refer to the specification documents from the W3C.

http://www.w3.org/TR/xpath

1.8 XSL

XSL (eXtensible Stylesheet Language) is a language to define XML style. An XML document itself does not have any information to define how the document should be viewed; we reference XSL stylesheets to transform the XML data.

XSL is made up of XSLT and XSLFO. In this section, we describe these two technologies.

1.8.1 XSLT

XSLT (XSL Transformations) is a language which converts XML documents into XML documents, HTML or text documents. XSL also uses the XML
syntax. Note that XSLT is the transformation engine that transforms an XML document by applying an XSL stylesheet to it. You will sometimes see the two terms used to mean the same thing, which is basically changing an XML document from one format to another.

XSLT stylesheets are used for the following two primary purposes:
1. Converting XML into HTML
2. Converting XML into another form of XML

Converting XML into HTML is very important technology to provide data to clients, usually Web browsers. This conversion is usually done on the server side. But a Web client can also browse an XML document if the client software (usually a Web browser) supports rendering XML and XSL. In this case, the client program will view the browsable XML data which is converted by XSL.

Server-side conversion is very useful technology. With this transformation, every Web browser, which can only view HTML documents, can view the data contained in XML documents in HTML style. This server-side transformation is quite similar to the HTTP task in Notes/Domino. The HTTP task changes forms and documents into HTML that is browsable to Web browser.

XSLT can provide many styles of HTML documents from a single XML document. This means we can handle data and style separately: XML for
data description and XSL for data presentation. Actually, these concepts already exist in HTML and CSS. CSS (Cascading Style Sheet) is a style sheet syntax which can define the view information for each tag. We can even use CSS with XML, but XSL has much more functionality than CSS. With XSLT, we can even do some processing, including blocks, looping, and sorting.

XML can also be transformed into a different XML document with an XSL stylesheet. This technology is important to connect with other systems which have other DTDs. XSLT compensates for the differences between the DTDs.

The following figure shows a good reason for using XSLT. With XSLT, we can transform XML to any style with an XSL stylesheet. As illustrated in Figure 9, the transformation can be into one of three types. To do this we need three types of stylesheets; the XSLT processor is one.

![Figure 9. XSLT transforms XML into other formats](image)

An XSL document is written in XML syntax. The following is a sample of an XSL file which converts an XML file into a HTML file.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
  <xsl:template match="/">
    <html>
      <title>XSL sample</title>
      <body>
        <H2>Customer list for Fish.net</H2>
        <table border="1">
          <tr>
            <td>ID</td>
            <td>name</td>
            <td>company</td>
          </tr>
        </table>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```
As this sample shows, XSL is nothing but an XML document. It has an XML declaration in the head of the document. After the XML declaration is the stylesheet declaration part which is the root element of this document. In this part we also declare the XSLT namespace. The latest XSLT namespace is "http://www.w3.org/1999/XSL/Transform," but in this sample we use another namespace which IE5.0 can handle. In IE, the XSLT namespace is "http://www.w3.org/TR/WD-xsl."

As at the time of this writing, Microsoft was updating the way the IE 5.5 browser handled namespaces. We recommend that you check their Website first, before trying these examples, if you are using Internet Explorer 5.5 for the correct namespace.

We start to describe template rules after the stylesheet declaration. In this template, we describe the conversion rule to the new document (in this case,
an HTML document). In an XSL file, we describe the transformation rule. In this part we define two things: which elements are to be transformed and how they are to be transformed.

This XSL example will make XML into HTML which uses tables. The following example is an XML document which should be transformed by this XSL stylesheet.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<?xml-stylesheet type="text/xsl" href="xslsample.xsl"?>
<customerlist>
  <customer>
    <customerid>001</customerid>
    <name>Tetsuy Miwa</name>
    <company>TM Fish</company>
  </customer>
  <customer>
    <customerid>002</customerid>
    <name>David Johnson</name>
    <company>DJ Fish</company>
  </customer>
  <customer>
    <customerid>003</customerid>
    <name>Alex White</name>
    <company>AW Fish</company>
  </customer>
</customerlist>
```

This XML file indicates the XSL file directly. This XML file can be browsed with XSL in I.E5.0.
Figure 10. XML document browsed with XSL file

This XSL file and XSLT processor create the following HTML code from this sample XML code. We can easily get another HTML which is from the same XSL file and another XML file.

```html
<html>
<title>XSL sample</title>
<body>
<h2>Customer list for Fish.net</h2>
<table border="1">
<tr>
<td>ID</td><td>name</td><td>company</td>
</tr>
<tr>
<td>001</td><td>Tetsuy Miwa</td><td>TM Fish</td>
</tr>
<tr>
<td>002</td><td>David Johnson</td><td>DJ Fish</td>
</tr>
<tr>
<td>003</td><td>Alex White</td><td>AW Fish</td>
</tr>
</table>
</body>
</html>
```
Let's see how this XSL file creates this HTML code with XSLT processor. In the XSL file, we describe template rules. At first, this XSL will try to apply the following template to the root node of XML documents:

```xml
<xsl:template match="/">
  <html>
    <title>XSL sample</title>
    <body>
      <h2>Customer list for Fish.net</h2>
      <table border="1">
        <tr>
          <td>ID</td>
          <td>name</td>
          <td>company</td>
        </tr>
        <xsl:apply-templates select="//customer"/>
      </table>
    </body>
  </html>
</xsl:template>
```

This template creates the main points of HTML code. Just before this process closes the `<table>` tag, the XSLT processor will try to apply the next template, as follows:

```xml
<xsl:template match="//customer">
  <tr>
    <xsl:apply-templates select="id"/>
    <xsl:apply-templates select="name"/>
    <xsl:apply-templates select="company"/>
  </tr>
</xsl:template>
```

As in the first process, the XSLT processor will try to finish the next template process.

```xml
<xsl:template match="id">
  <td><xsl:apply-templates/></td>
</xsl:template>
<xsl:template match="name">
  <td><xsl:apply-templates/></td>
</xsl:template>
<xsl:template match="company">
  <td><xsl:apply-templates/></td>
</xsl:template>
```

For each customer, a new table row is created and the customer's ID is processed, followed by the name and company. The table row is then closed. As in the first process, the XSLT processor will try to finish the next template process.
In this template, XSLT will return the element values if the node is a Text node. Then these values will be given back to their parent node templates recursively. In this way, the previous HTML code will be created with this transformation.

This example is XSL client-side conversion. We can also use server-side conversion. In the real world, especially on the Internet, it would be better to use server-side transformation because we do not have to be concerned about the client browser version. Even if we do server-side transformation, the client can see the XML file just the same as in this example.

Conversion of XML to text allows data to be presented in any format that can be rendered in text, though it can be laborious. For example, you could convert XML into a CSV (comma separated values) file in order for it to be imported into a spreadsheet.

### 1.8.2 XSL-FO

XSL-FO (XSL Formatting Object) is another function of XSL. XSLT is used to transform an XML document into another XML document. XSL-FO defines how to display the resulting tree.

At the time of writing, XSL-FO is still in a theoretical stage; we used an XSL stylesheet to display the XML document using Notes/Domino. You can track the progress of the XSL-FO specification on the W3C website at:

http://www.w3c.org/

### 1.9 SOAP

HTTP is one of the most common protocols to connect systems on the Internet. However, the HTTP protocol is a very simple protocol that supports only a few methods, like GET and POST. Although HTTP is a protocol which is used for browsing data, HTTP applications need many functions because of their wide use. Now, almost everyone has a Web browser on their client as an all-purpose client. HTTP applications are becoming much more functional using other technology, for example CGI, Java Servlets, Java Applets and IIOP.

On the other hand, classical Client/Server systems, including Notes/Domino, use their own RPC (Remote Procedure Call) protocol. Using RPC, we can call
the server function from another terminal. CORBA applications with IIOP make it possible to do the same thing in intranet systems, as does Microsoft DCOM technology.

SOAP (Simple Object Access Protocol) is another way of making this possible. SOAP is a lightweight protocol for exchanging information in distributed environments. Using the SOAP technology, we can build an RPC system with HTTP and XML. SOAP makes it possible to call server functions directly from a client using HTTP and XML. SOAP applications are usually based on the HTTP protocol, which is allowed to pass through almost all firewalls (usually port 80), although other transports can be used as well. XML is used as a function call message in an HTTP POST command and the data we get from the system is described in XML. Figure 11 shows a simple example of a SOAP implementation.

Figure 11. Simple example of SOAP on HTTP connection

SOAP 1.1 (of which Lotus and IBM are co-authors) defines the SOAP envelope, SOAP encoding rules, and SOAP RFC representation. The SOAP envelope defines an overall framework for expressing what is in a message, who should deal with it, and whether it is optional or mandatory. A simple image of a SOAP envelope is the following:

```xml
<Envelope>
  <!-- envelope part -->
  SOAP message declaration
  Define encoding

  <Header>
  <!-- Header part -->
  Define the way of data processing
  </header>

  <Body>
```

```xml
</Body>
</Envelope>
```
A real example of a SOAP message, which is introduced in SOAP1.1, is the following. This example is uses the previous SOAP envelope described above.

```
POST /StockQuote HTTP/1.1
Host: www.stockquoteserver.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "Some-URI"

<SOAP-ENV:Envelope
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
 SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
 <SOAP-ENV:Body>
 <m:GetLastTradePrice xmlns:m="Some-URI">
  <symbol>DIS</symbol>
 </m:GetLastTradePrice>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

We can get the following message response against the previous request.

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP-ENV:Envelope
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
 SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
 <SOAP-ENV:Body>
 <m:GetLastTradePriceResponse xmlns:m="Some-URI">
  <Price>34.5</Price>
 </m:GetLastTradePriceResponse>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

For more information about SOAP, refer to the specification documents from the W3C.

http://www.w3.org/TR/SOAP/
1.10 Other XML technology

XML is not an application, but an architecture. There are many extended XML technologies that use this XML architecture. Several of the important ones that do so are Xlink and XPointer, which are described in this section.

1.10.1 Xlink

Linking technology is really necessary to use the Internet. One of the big reasons for the great success of HTML in the Internet is HTML linking capability. HTML linking makes it possible to connect all over the world just by one click of a mouse. The following is the simple example of HTML linking.

\[<a href="http://fish.net/">Fish.net</a>\]

XML also has a linking technology: Xlink (XML Linking Language). A simple use of Xlink is very similar to that of HTML linking. The following is the example of it.

\[<a xml:link="simple" href="http://fish.net/">Fish net</a>\]

But the HTML linking function is just for one way linking. So, Xlink has to refer to other technology, HyTime and TEI (Text Encoding Initiative), for multidirectional linking. Actually, Xlink has a lot of extensions compared with HTML linking. For example, Xlink supports the following functions:

- Mutual linking between two objects
- Linking from one object to multiple objects
- No need to have tags for linking
- Describing linking information which is not an object to be linked

In this way, Xlink is much more useful than HTML link, but its use is rare in developing Domino and XML applications. For more information about Xlink, refer to the specification documents from the W3C.

http://www.w3.org/TR/xlink/

1.10.2 XPointer

XPointer (XML Pointer Language) is a method to provide pointing. In XPointer, one defines the addressing expression for linking XML documents using XPath.

For more information about XPointer, refer to the specification documents from the W3C.

http://www.w3.org/TR/xptr
1.11 Examples of using XML

Because XML can be used in many ways to describe data, there are many XML applications in the world. Using XML with a strict rule in an industry, we can define a lot of data structures as standard. In the real world, XML is already used in many industries, for example in the scientific and medical fields, in publishing, broadcasting, communications, and financial services, among others.

In this section we show some examples of the world which XML made.

1.11.1 SMIL

In broadcasting and communications, XML is used to describe multimedia content such as sound, voice, and moving pictures. SMIL (Synchronized Multimedia Integration Language) is one of the major languages to describe multimedia using XML.

SMIL is not a protocol to describe these contents directly, but instead describes how to control the timing of playing and showing data. In SMIL, we can decide the layout of contents, synchronization timing, and so on, with many kinds of media files.

For more information about SMIL, refer to the specification documents from the W3C.

http://www.w3.org/AudioVideo/

1.11.2 MathML

In the scientific community, XML is also widely used because drawing complex expressions and characters is a suitable task for using XML.

MathML (Mathematical Markup Language) is an XML solution used to draw mathematical expressions. Mathematical expressions are sometime difficult to express in plain text format. MathML makes it possible to describe mathematical expression in plain text format.

For example, look at the following mathematical expression, which is difficult to express in a plain text editor.

\[
x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}
\]
This expression can be described in MathML as follows.

\[
\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}
\]

There are many tools to use this MathML. Using the tools which support MathML and XHTML, we can use MathML inside of HTML as XHTML and browse these expression directly.

For more information about MathML, refer to the specification documents from the W3C.

http://www.w3.org/Math/
Chapter 2. Lotus and XML

Notes and Domino are well-suited to handle XML because of similar architecture. Lotus and IBM provide lots of solutions to make the most of XML with Lotus products.

In this chapter, we explain the XML tools and solutions applicable to Lotus.

2.1 Domino as an XML data-store

Domino databases are really good for storing and handling XML. In this section we explain why.

2.1.1 Domino and XML architecture

Domino and XML architecture are quite similar in that they have separate data and presentation parts. In a Domino database, all the data is stored as documents. Each document is nothing but data itself. To browse this document, we use a form to give it structure; this is the presentation part. A Notes client displays the data on the form, with calculations in some cases, and a Web browser browses the HTML file which the HTTP task creates from the original data document and a form for HTTP. The viewing logic from the Notes client resembles that of client-side conversion with XML and XSL files. On the other hand, as we mentioned previously, browsing a Domino document from a Web browser with an HTML file created by the HTTP task resembles the process of server-side conversion with XSLT.

Domino documents are collections of values stored in Domino fields. Figure 12 shows the properties tab of a Domino document.

![Figure 12. Domino document properties](image-url)
As the figure shows, a Domino document does not have a presentation logic. Instead, it has a “form” field which indicates the form which the document will use. This is comparable to indicating the XSL file in an XML document. This is illustrated in Figure 13.

**Figure 13. XML and Domino architecture comparison**

The similar architecture of Domino and XML is the main reason that Notes/Domino is suitable to connect to other systems with XML. In addition, there are other features that make using Domino advantageous. They are described in this section.

### 2.1.2 Domino forms

Domino forms are used as the presentation logic in Notes/Domino. Even before XML, we were able to change the form we used to display data so that same information could be shown in different ways. For example in a workflow application, we can easily store all the fields (data) in a single document and then use different forms to display it differently to different people. Another example is an application which can be used by both Notes clients and Web browsers. We would create one form for a web browser and another for a Notes user. By switching the forms between Notes and Web versions, we can easily build this kind of bi-modal application.

Now let's consider an XML form. Just as we can use Domino forms to organize and display Domino fields, it is also possible to make a simple XML...
form, which is written in pass-thru HTML. Figure 14 is a simple example of this.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<customer>
  <id>ID</id>
  <lastname>Lastname</lastname>
  <firstname>Firstname</firstname>
  <company>Company</company>
  <address>Address</address>
  <zip>Zip</zip>
  <zone>Zone</zone>
</customer>
```

*Figure 14. Example of simple XML form*

As you can see, this XML form appears quite useless if viewed by a Notes client as we would see all its tags. However, we can use this form to easily display XML data. This is a very powerful way of using Domino documents and forms.

### 2.1.3 Domino views

Domino views let us display collections of documents, showing the value of specific fields stored within those documents. This view architecture is also suitable to handle XML. Even though we can use forms to display XML, we cannot make one big XML document composed of multiple documents, which is the mechanism Domino view uses. Domino accomplishes this with view selection formulas. Using these formulas, we can produce the same kind of single, browsable document with XML data.

For example, if we want to make an XML document with data only for records that include "company=Lotus," we create this view with following view selection formula.

```
SELECT company = "Lotus"
```

Another feature of views is the capability to add some plain text for each view column. With this technique, we can easily make a useful XML view, like the simple example shown in Figure 15 on page 46.
Figure 15. Simple XML view

We usually use this view with page because this view does not have XML declaration and root elements.

2.1.4 Domino pages

Domino pages are also an important way to handle XML. Pages are a new design element in Domino R5 and are used to store static information such as raw HTML for a web page, a cascading style sheet or maybe just plain text.

One way of using a page is as a container for an embedded view. In this way we can create a dynamic XML page by adding XML tags around values in a view. Figure 16 is an example of this kind of page with an embedded XML view. In this example, we add an XML declaration, create a root element on the page and call it <Customers> and by adding an embedded view, create the nodes for us automatically. We then add the closing </Customers> root element to the end of the page.

![Embedded view](image)

Figure 16. Page with an embedded XML view
Another way to make the most of pages is using them instead of the external file system. In using XML, there is sometimes a need to use an external plain text file, for example, XSL or DTD files. Since the HTTP function of the Domino server includes the HTTP daemon as well as the Notes to HTML conversion engine called the HTTP Task, we can put the file on the file system directly. Usually, we can use the file system located in the Domino-data-dir/domino/html. We can also change this by editing the server document.

A Domino page design element gives us a great substitute for these external files. Using pages is not just a substitution for file systems; using pages has a lot of merits compared with handling these files directly on the file system.

Using a page design element instead of an external file is very easy: we just enter the page name with an extension, for example, customer.dtd or style.xsl. Figure 17 is an example of using pages as external plain text files.

<table>
<thead>
<tr>
<th>Name/Comment</th>
<th>Alias</th>
<th>Last</th>
</tr>
</thead>
<tbody>
<tr>
<td>StyleExchangeRate</td>
<td>nasset.xsl</td>
<td>10/12</td>
</tr>
<tr>
<td>Use in MS Browser</td>
<td></td>
<td></td>
</tr>
<tr>
<td>StyleExchangeRate2</td>
<td>SERZ.XSL</td>
<td>09/08</td>
</tr>
<tr>
<td>Use with Transform XML agent.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>News DTD</td>
<td>News.DTD</td>
<td>09/08</td>
</tr>
<tr>
<td>News Feed Document Type Definition</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NewsFeedStyleSheet</td>
<td>NFSS.xsl</td>
<td>09/08</td>
</tr>
<tr>
<td>News feed style sheet for More Over †</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Confirmed</td>
<td></td>
<td>06/01</td>
</tr>
</tbody>
</table>

*Figure 17. Naming a page as an external file*

The additional advantages of using pages as external files relate to replication and security.

If we use the external files on a file system, we have to be concerned with how they will be downloaded and uploaded. We can download the files on HTTP, but we have to use FTP or some other solution to upload these files, which means we have to make the FTP protocol open to the server. When using pages, we do not need to worry about this. Domino has a strong security system, requiring cross-authentication to the server and also an appropriate ACL to the database.
In addition to the security issue, Notes/Domino has a great replication function. The Domino server does not support a file system replication, but we can get the benefit of replication if we use pages.

Table 6 summarizes the differences between using external file systems and Domino pages.

Table 6. Domino page versus file system

<table>
<thead>
<tr>
<th></th>
<th>File system</th>
<th>Domino page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Download to use</td>
<td>Via HTTP</td>
<td>Via HTTP</td>
</tr>
<tr>
<td>Upload to use</td>
<td>Via FTP (Server must have opened the FTP protocol)</td>
<td>Via NRPC, with strong Domino security</td>
</tr>
<tr>
<td>Replication</td>
<td>Manually, or you need some tool</td>
<td>With Domino replication</td>
</tr>
<tr>
<td>File modification</td>
<td>Use text editor or XML editor and do not need Domino Designer to create or modify</td>
<td>Need to use Domino Designer</td>
</tr>
<tr>
<td>File management</td>
<td>Need to remember the relationship with database</td>
<td>All the files are included in domino database</td>
</tr>
</tbody>
</table>

As this table shows, it is usually better to use Domino pages instead of external files.

2.1.5 Domino agents

The function of Domino agents is another good reason for using Domino for your XML needs. A Domino agent is a program which allows us to perform many different types of actions on a database and its contents. Agents are flexible to use since they can be run on a scheduled basis or they can be event-driven. With these agents, we can import, export and transform XML files easily.

We can use both Java and Lotuscript to write agent code. It is very important that we can use Java since it enables us to handle XML using many of the XML Java classes available. Domino has a built in XML parser (after R5.0.3) and we can use the parser from a Java agent. As you will see later, we used Domino agents as alot when preparing this redbook.

If you are using IE 5 and above, you can access the built-in XML COM objects in Internet Explorer from Lotuscript. The following example is a Lotuscript agent that would be used to send data from a Domino page element to the Microsoft XMLDOM object.
Sub Initialize
    Dim source as Variant
    sourcefile = "http://localhost/mydatabase.nsf/xmlviewpage?openpage"
    Set source = CreateObject("Microsoft.XMLDOM")
    source.async = False
    source.load(sourcefile)
    Print "Content-type: text/xml"
    Print "source.xml"
End Sub

2.2 Domino XML Language (DXL)

Domino architecture makes it simple to create XML by ourselves. To do this we have to create XML views, forms, and DTDs.

Lotus will provide a generic DTD for Domino. It will describe (almost) all aspects of the Domino database. In this section we introduce this DTD, called DXL (Domino XML Language).

2.2.1 What is DXL

DXL is an XML language for representing Domino data. Lotus defined a number of elements (tags) and attributes to represent Notes documents, forms, views, fields, rich text, and everything (well, almost) that is stored in a .nsf file. DXL is a quite different concept than what we explained in the previous section. DXL is a universal data format for Domino, and we can describe both document data and design elements in DXL. (Although it cannot describe all elements of Domino at present, that is the ultimate goal of development efforts.)

As of this writing, Domino DTD supports two types of DTD: View DTD and Document DTD. These two DTDs have some common elements and entities. (When the XML toolkit from Lotus ships, these will be combined into a single DTD).

The following entities are common and are mainly used to express the data types:

<!ENTITY % boolean "( true | false )">
<!ENTITY % integer "CDATA">
<!ENTITY % noteid "CDATA">
<!ENTITY % unid "CDATA">
<!ENTITY % string "CDATA">

DateTime is also a common element and attribute. The following "dst" is used as the flag for daylight saving:
Also, the following entities are defined:

```
<!ENTITY % simple.types "text | number | datetime | textlist |
numberlist | datetimelist"> 
```

These entities are used to describe the data in a document item or view column.

### 2.2.1.1 View DTD

View DTD represents the structure of a view in a Domino database. The ReadViewEntries URL command, which was added in R5.0.2, will create an XML document in accordance with this View DTD.

View DTD has a “ViewEntries” element as a root element, and this root element has two children elements: “ViewEntry” and “Entrydata.” The Entrydata element is actually a child of ViewEntry.

**Table 7. Elements used in View DTD**

<table>
<thead>
<tr>
<th>Element</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>ViewEntries</td>
<td>The root element of the result DXL.</td>
</tr>
<tr>
<td>ViewEntry</td>
<td>Represents a line in a view.</td>
</tr>
<tr>
<td>Entrydata</td>
<td>Represents a single column of data for a line in a view.</td>
</tr>
</tbody>
</table>

Each element also has many attributes. Figure 18 shows the relationship of the elements and their attributes.
The following is an XML sample created in accordance with View DTD.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- Lotus-Domino (Build V503_12211999 - December 21, 1999 on Windows NT/Intel) -->
<viewentries toplevelentries="2">
  <viewentry position="1" noteid="80000058" children="16" descendants="65" siblings="6">
    <entrydata columnnumber="0" name="Category" category="true">
      <text>General Feedback</text>
    </entrydata>
  </viewentry>
  <viewentry position="2" noteid="8000005C" children="356" descendants="734" siblings="6">
    <entrydata columnnumber="0" name="Category" category="true">
      <text>General Feedback for Iris</text>
    </entrydata>
  </viewentry>
</viewentries>
```

The details of View DTD we can see in Domino R5 Designer help are as follows:

```xml
<!ELEMENT viewentries ( viewentry* )>
<!ATTLIST viewentries
toplevelentries%integer;#IMPLIED
>
<!ELEMENT viewentry ( entrydata* )>
<!ATTLIST viewentry
position#CDATA#IMPLIED
name#CDATA#IMPLIED
"columnnumber","indent"
"category","category"
"unid","noteid","children","siblings"
"descendants","collapsed","response","conflict"
"categorytotal","markedfordel"
2.2.1.2 Document DTD

Document DTD represents the structure of a Domino document in a Domino database. The GenerateXML method will create an XML document in accordance to this Document DTD.

Document DTD has a “Document” element as a root element. This Document root element has two major children elements: the “NotesInfo” element is used to describe identity information about notes, and the “Item” element is used to describe an item in a note. The NoteInfo element also has two elements: “created” and “modified.” The elements are summarized in Table 8.

Table 8. Elements used in Document DTD

<table>
<thead>
<tr>
<th>Element</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Document</td>
<td>The root element of the result DXL.</td>
</tr>
<tr>
<td>NoteInfo</td>
<td>The noteinfo element contains identity information about notes.</td>
</tr>
<tr>
<td>Item</td>
<td>Represents an item in a note.</td>
</tr>
<tr>
<td>created</td>
<td>Represents the creation date and/or time of the note.</td>
</tr>
<tr>
<td>modified</td>
<td>Represents the date and/or time the note was modified.</td>
</tr>
</tbody>
</table>

Each elements also has many attributes. Figure 19 shows the relationship of the elements and their attributes.
The following is an XML sample created in accordance with Document DTD:

```xml
<document form='DocForm'>
  <noteinfo unid='D83C5EFA90BC3ED85256862000A1043'>
    <created><datetime>20000109T204955,23-05</datetime></created>
  </noteinfo>
  <item name='Text1'><text>This is some text!</text></item>
  <item name='DateTime1'><datetime>19990616</datetime></item>
</document>
```

The details of Document DTD we can see in Domino R5 Designer help are as follows:

```xml
<!ELEMENT document (noteinfo?, item*)>
<!ATTLIST document
  parent%unid;#IMPLIED
  response%boolean;"false"
  form%string;#IMPLIED
  conflict%boolean;"false"
>
<!ELEMENT noteinfo (created?, modified?)>
<!ELEMENT created (datetime)
<!ELEMENT modified (datetime)
<!ATTLIST noteinfo
  unid%unid;#IMPLIED
  noteid%noteid;#IMPLIED
>
<!ELEMENT item (%simple.types;)
<!ATTLIST item
  name%string;#REQUIRED
```
Both View DTD and Document DTD are supported in the current release; the DXL goal is to describe all contents in Domino databases, including documents, design elements, and any kind of object which is contained in a Domino database.

2.2.2 How to use DXL

DXL is just a representation of Domino data for XML; it means nothing by itself. What is the best way to use DXL? Because XML is a universal data format, we can use XML documents in many way. The following are some simple examples of ways in which DXL can be used:

- Exporting Domino data
- Exporting Domino data via XML which is transformed from DXL with XSLT
- Browsing Domino data via HTML which is transformed from DXL with XSLT
- Importing external DXL
- Importing XML transformed to DXL via XSLT
- Editing Domino design elements
- Creating Domino design elements
- Comparing and merging the design of two or more Domino databases or templates

The details of some of these examples are discussed in this section.

One of easiest ways to use DXL is exporting. A Domino database can be easily exported with DXL and data can be exchanged with other systems. To use this capability, the recipient of the data must understand DXL. The recipient can use the XML data either by importing it to Domino or transforming it to another XML format.

The second likely scenario for using DXL is exporting not the DXL file itself, but something which is transformed by XSLT. Since DXL files are XML documents, we can convert them to other XML or HTML documents. As we mentioned previously, we can transform XML documents to any other XML document, but DXL is not a data format used to exchange data with other systems or to provide data to Web browsers. XSLT will solve this problem.
Figure 20 illustrates the flow of this transformation. In this example we have to create XSL stylesheets to do the transformation.

Just as we can use it for exporting, we can also use DXL to import. We can easily import XML documents from DXL. It is a good idea to convert XML to DXL after we receive the XML documents to import to a Domino database. Although it is not always easy to transform every XML document into DXL, this XSLT will help to use DXL well. Figure 21 on page 56 illustrates this.
In this way, DXL can be used both for exporting and importing XML documents. Since DXL is an XML document and described in plain text, we can edit this text file to change the design elements of Domino database without using Domino Designer. We need to use the previous DXL export and import to change Domino objects via DXL. Figure 22 on page 57 shows a simple example of changing Domino objects by modifying DXL.

Figure 21. Getting XML into a Domino database via DXL
Figure 22. Changing Domino objects by modifying DXL

It is also possible to use a third-party tool to handle DXL. In this way, we can not only do imports, exports, and modification, but also create Domino objects via DXL.

2.2.3 Lotus XML toolkit

The toolkit allows developers to build applications that export and import DXL out of and into Domino (R5). It includes API class libraries for Java and C++ for use with Notes Java/C++ applications. The toolkit includes sample command-line utilities to import and export. For more information on using the toolkit, see 3.9, “Working with the Lotus XML Toolkit” on page 135.

Note

There may be some confusion about whether the command-line samples, DXLimport and DXLexport, are samples or products. They are samples whose primary purpose is to show how to call the API classes from an application and demonstrate the functionality of the toolkit. The following examples in this book emphasize the use of these samples. However, the main purpose of the toolkit is to create applications that have import and export functionality.
Since at the time of this writing the Lotus XML toolkit is still version 0.5 in Beta, and not complete, there is some possibility of changing the spec. An overview of what is in the toolkit follows.

The Lotus XML toolkit provides C++ classes and will also provide Java classes for handling DXL. By using these classes, we can make a powerful DXL utility program. The toolkit has very useful samples, “DXLexport” and “DXLimport.” These sample tools are command-line utilities which will enable us to create DXL documents from Domino database and also the DXL document back into Domino database.

You can download the Lotus XML Toolkit from the Lotus Web site at:

http://www.lotus.com/xml

Let’s see how a sample program works. First, we make an example of a DXL file. Figure 23 shows a simple Domino document, which has only two fields, “title” and “body.”

![Figure 23. Sample Domino document to be transformed into DXL](image)

Now let’s make this Domino document into DXL with the DXLexport tool. As the readme.txt shows, we issue the following command:

D:\LotusXML\bin\mswin32>dxlexport dxltest.nsf 0x8F6

This command will create a DXL document for the document which has “8F6” noteid in dxltest.nsf. We can find the noteid from the property of Domino Document, as shown in Figure 24.
The XML document resulting from the command is the following:

```xml
<?xml version='1.0' encoding='utf-8'?>
<!-- DXLExporter version 0.5, schema (DTD) version 0.13 -->
<!-- WARNING: This data does not conform to a supported DTD yet! -->
<!DOCTYPE document SYSTEM 'domino.dtd'>
<document xmlns='http://www.lotus.com/dxl' version='0.13' form='main'>
  <noteinfo noteid='8fa' unid='510BF367AE2232D78525696E000751F6'
            sequence='1'>
    <created><datetime>
      dst='true'>20001003T211957,34-04</datetime></created>
    <modified><datetime>
      dst='true'>20001003T212008,50-04</datetime></modified>
    <revised><datetime>
      dst='true'>20001003T212008,49-04</datetime></revised>
    <lastaccessed><datetime>
      dst='true'>20001003T212008,49-04</datetime></lastaccessed>
    <addedtofile><datetime>
      dst='true'>20001003T212008,49-04</datetime></addedtofile>
  </noteinfo>
  <item name='Title'><text>Test example</text></item>
  <item name='Body'><richtext>
    <par><run><font style='bold'/>richtext</run></par>
    <run><font color='red'/>red</run></font>
    <run><font color='blue'/>blue</run></font>
    <run><font color='lime'/>green</run></font>
  </richtext></item>
  <item name='$UpdatedBy' names='true'><textlist>
    <text>CN=Designer Fishnet/O=Fishnet</text>
  </textlist></item>
</document>
```

The DXLExport tool can also make Domino documents into DXL. Now let's try to make a DXL document which expresses design elements. Figure 25 on page 60 is a simple example of a Domino form.
This time we use the "element name" parameter instead of "noteid." In the following example, we will create the DXL document from the "main" element:

D:\LotusXML\bin\mswin32>dxlexport dxltest.nsf main

This command creates the following DXL:

```xml
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE form SYSTEM 'domino.dtd'>
<form xmlns='http://www.lotus.com/dxl' version='0.13' designerversion='5' name='main'>
  <noteinfo noteid='136' unid='126B76458B9B87188525696E0005EE25' sequence='4'>
    <created><datetime dst='true'>20001003T210446,45-04</datetime></created>
    <modified><datetime dst='true'>20001003T212752,56-04</datetime></modified>
    <revised><datetime dst='true'>20001003T212752,55-04</datetime></revised>
    <lastaccessed><datetime dst='true'>20001003T212752,55-04</datetime></lastaccessed>
    <addedtofile><datetime dst='true'>20001003T210523,29-04</datetime></addedtofile></noteinfo>
  <updatedby><name>CN=Designer Fishnet/O=Fishnet</name></updatedby>
  <body><richtext>
    <par/>
    <pardef id='1'/>
  </richtext>
</form>
```
In this way, we can make both documents and design elements into DXL.

We can use the `dxlimport` command just like the `dxlexport` command. The following command will create a Domino document in accordance with DXL. In this command, we add the dxltext.xml which is written in DXL into dxltest.nsf; `-i` and `-d` are parameters to designate the names of the imported XML file and database file.

```
D:\LotusXML\bin\mswin32>dxlimport -i dxltext.xml -d dxltest.nsf
```

These command utilities are just a sample; using the Lotus XML toolkit you can do much more complex work with DXL. XML toolkit is just a set of classes, which are described in Table 9.

**Table 9. Classes provided in Lotus XML Toolkit**

<table>
<thead>
<tr>
<th>Name of class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXL Base</td>
<td>Base class for DXLEnumer, DXLHandle, and DXLImporter.</td>
</tr>
<tr>
<td>DXLDatabase</td>
<td>Represents a Domino database, used for both import and export operations.</td>
</tr>
<tr>
<td>DXLException</td>
<td>Base class for DXLImportException and DXLExportException, which are used to return error information to applications.</td>
</tr>
<tr>
<td>DXLEnumer</td>
<td>Provides functions to export DXL from a Domino database.</td>
</tr>
<tr>
<td>DXLExportException</td>
<td>Used by DXLEnumer to return error information.</td>
</tr>
<tr>
<td>DXLHandle</td>
<td>Base class for DXLDatabase, DXLException, DXLStream, and DXLString.</td>
</tr>
<tr>
<td>DXLImporter</td>
<td>Provides functions to import DXL into a Domino database.</td>
</tr>
<tr>
<td>DXLImportException</td>
<td>Used by DXLImporter to return error information.</td>
</tr>
<tr>
<td>DXLInputStream</td>
<td>DXL input source for import operations. Represents a file, URI, or the standard input device.</td>
</tr>
</tbody>
</table>
2.3 Lotus/IBM XML source

Lotus and IBM place great importance on XML technologies. They provide lots of useful XML tools and technical information.

IBM Alphaworks (http://www.alphaworks.ibm.com/) provides many useful XML tools. Here you can directly access the alpha version of code which uses the latest software technologies. Alphaworks provides lots of the latest XML and Java tools, all of them are free to download.

The following tools are representative examples of alphaworks tools.

- XML parser for Java
- LotusXSL
- Xeena (Visual XML Editor)
- XML Viewer
- XML Security Suite

XML parser for Java and LotusXSL are actually included in Domino and Domino Designer. We can use these tools from Domino application to use XML.

IBM Developer works (http://www.ibm.com/developer/) also provides technical references for developers. Here you can read a lot of the latest news and technical information about XML.

2.4 IBM XML Parser for Java

As we mentioned previously, the XML parser is really an important concept in handling XML documents. IBM provides an XML parser, which was developed in accordance with XML specifications and is also a license-free tool. The

<table>
<thead>
<tr>
<th>Name of class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DXLOutputStream</td>
<td>DXL output target for export operations. Represents a file or the standard output device.</td>
</tr>
<tr>
<td>DXLStream</td>
<td>Base class for DXLInputStream and DXLOutputStream.</td>
</tr>
<tr>
<td>DXLString</td>
<td>Represents a string in the native character set, Unicode, or LMBCS (Lotus Multibyte Character Set, used in Domino databases).</td>
</tr>
</tbody>
</table>
IBM XML Parser for Java (XML4J) is a robust validating XML parser which is written in 100% pure Java.

XML and Java are a good combination. Just as XML is a universal data format, Java is a universal programming language that is suitable for any platform and can easily use uniques. The IBM XML Parser for Java provides both DOM and SAX APIs, and also supports namespace specifications. We can download this tool from the IBM Alphaworks Web site.

Domino and Domino Designer include XML4J. Domino uses this parser in the “parseXML” and “generateXML” methods. We can also use this parser directly in programming as a way of handling XML objects with DOM and SAX. We use this XML parser for Java in Domino Java Agent and Java Servlet.

In general, we have to do the following setup before we use the XML Parser for Java. The Java Development Kit (JDK) must be at release 1.1 or 1.2.

- Install JDK
- Install XML Parser for Java
- Set the CLASSPATH to contain the parser’s .jar file.

When we use this parser from Domino, we usually do not need to do these things because Domino is set correctly natively. However, we have to set the CLASSPATH for NotesJAR, XML4J.JAR, and LotusXSL.Jar when we use this application in a standalone environment.

### 2.4.1 DOM with XML Parser for Java

XML parser for Domino provides two main interfaces, DOM and SAX. In DOM programming, we can use the Org.W3C.DOM package, which has the interfaces described in Table 10.

#### Table 10. DOM interface summary

<table>
<thead>
<tr>
<th>Interface</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>org.w3c.dom.Attr</td>
<td>The Attr interface represents an attribute in an Element object.</td>
</tr>
<tr>
<td>org.w3c.dom.CDATASection</td>
<td>CDATA sections are used to escape blocks of text containing characters that would otherwise be regarded as markup.</td>
</tr>
<tr>
<td>org.w3c.dom.CharacterData</td>
<td>The CharacterData interface extends Node with a set of attributes and methods for accessing character data in the DOM.</td>
</tr>
<tr>
<td>Interface</td>
<td>Summary</td>
</tr>
<tr>
<td>-----------</td>
<td>---------</td>
</tr>
<tr>
<td>org.w3c.dom.Comment</td>
<td>This represents the content of a comment, i.e., all the characters between the starting '&lt;!--' and ending '--&gt;'.</td>
</tr>
<tr>
<td>org.w3c.dom.Document</td>
<td>The Document interface represents the entire HTML or XML document.</td>
</tr>
<tr>
<td>org.w3c.dom.DocumentFragment</td>
<td>DocumentFragment is a “lightweight” or “minimal” Document object.</td>
</tr>
<tr>
<td>org.w3c.dom.DocumentType</td>
<td>Each Document has a doctype attribute whose value is either null or a DocumentType object.</td>
</tr>
<tr>
<td>org.w3c.dom.DOMException</td>
<td>DOM operations only raise exceptions in “exceptional” circumstances, i.e., when an operation is impossible to perform (either for logical reasons, because data is lost, or because the implementation has become unstable).</td>
</tr>
<tr>
<td>org.w3c.dom.DOMImplementation</td>
<td>The DOMImplementation interface provides methods for performing operations that are independent of any particular instance of the document object model.</td>
</tr>
<tr>
<td>org.w3c.dom.Element</td>
<td>The Element interface represents an element in an HTML or XML document.</td>
</tr>
<tr>
<td>org.w3c.dom.Entity</td>
<td>This interface represents an entity, either parsed or unparsed, in an XML document.</td>
</tr>
<tr>
<td>org.w3c.dom.EntityReference</td>
<td>EntityReference objects may be inserted into the structure model when an entity reference is in the source document, or when you wish to insert an entity reference.</td>
</tr>
<tr>
<td>org.w3c.dom.NamedNodeMap</td>
<td>Objects implementing the NamedNodeMap interface are used to represent collections of nodes that can be accessed by name.</td>
</tr>
<tr>
<td>org.w3c.dom.Node</td>
<td>The Node interface is the primary datatype for the entire Document Object Model.</td>
</tr>
<tr>
<td>org.w3c.dom.NodeList</td>
<td>The NodeList interface provides the abstraction of an ordered collection of nodes, without defining or constraining how this collection is implemented.</td>
</tr>
</tbody>
</table>
These interfaces have some useful methods. We can see the detail of these methods in Domino R5 Designer Help.

Let's consider an example of using the XML Parser for Java in a Domino Java agent. The following code is a Domino agent in which we also include an of how to use the DOM.

```java
import lotus.domino.*;
import org.w3c.dom.*;

public class JavaAgent extends AgentBase {

    public void NotesMain() {
        int i;
        try {
            Session ns = getSession();
            AgentContext ac = ns.getAgentContext();
            lotus.domino.Document doc = ac.getDocumentContext();
            Item rawXML = doc.getFirstItem("XMLDATA");
            org.w3c.dom.Document xDoc = rawXML.parseXML(false);
            Element el = xDoc.getDocumentElement();
            //Using DOM API
            String rootTag = el.getTagName();
            System.out.println("The Root Element is " + rootTag);
            NodeList nl = xDoc.getElementsByTagName(rootTag);
            System.out.println("There is " + nl.getLength() + " node in the "
                             + "Root Node List");
            Node n = nl.item(0);
            nl = n.getChildNodes();
            System.out.println("The " + rootTag + " Root Tag has " +
                               nl.getLength() + " child nodes");
            for (i = 0; i < 10; i++) {
                n = nl.item(i);
            }
        }
    }
}
```

These interfaces have some useful methods. We can see the detail of these methods in Domino R5 Designer Help.
if (n.getNodeType() == n.ELEMENT_NODE) {
    System.out.println("Node Name is:" + n.getNodeName());
    System.out.println("Node Value is: " + n.getNodeValue());
    System.out.println("Node Value is: " + n.getNodeType());
    System.out.println("This node has child nodes: " +
    n.hasChildNodes());
}
} catch(Exception e) {
    e.printStackTrace();
}
}

We explain how we use the XML parser for Java with Notes/Domino in a later
chapter.

2.4.2 SAX with XML Parser for Java

XML Parser for Java also has SAX API; we can use this SAX from Domino
Java Agent or Servlet. In SAX programming, we can use the Org.W3C.SAX
package which has the interfaces identified in Table 11.

Table 11. SAX interface summary

<table>
<thead>
<tr>
<th>Interface</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>InputSource</td>
<td>A single input source for an XML entity.</td>
</tr>
<tr>
<td>SAXException</td>
<td>Encapsulates a general SAX error or warning.</td>
</tr>
<tr>
<td>SAXParseException</td>
<td>Encapsulates an XML parse error or warning.</td>
</tr>
</tbody>
</table>

Following is an example of using XML Parser for Java in Domino Java Agent.
The following code is a Domino Agent sample in which we also used an
eample of SAX code.

```java
import org.xml.sax.*;
import org.xml.sax.Parser;
import lotus.domino.*;
import org.xml.sax.helpers.ParserFactory;
import java.io.*;

public class JavaAgent extends AgentBase {

    public void NotesMain() {
        int i;
        try {
            Session ns = getSession();
```
AgentContext ac = ns.getAgentContext();
lotus.domino.Document doc = ac.getDocumentContext();
Item rawXML = doc.getFirstItem("XMLDATA");
Reader r = rawXML.getReader();
// The following line of code implements the parser directly
// com.ibm.xml.parsers.SAXParser p = new com.ibm.xml.parsers.SAXParser();
// or use this to implement the parser through ParserFactory
Parser p = ParserFactory.makeParser("com.ibm.xml.parsers.SAXParser");
p.setDocumentHandler(new myDochandler());
p.parse(new InputSource(r));

} catch(Exception e) {
    e.printStackTrace();
}

myDокументhandler.java
import org.xml.sax.*;

public class myDochandler extends HandlerBase {

    public void startDocument() {
        System.out.println("Begining of Document");
    }

    public void endDocument() {
        System.out.println("End of Document");
    }

    public void endElement(String name) throws SAXException {
        System.out.println("End of Element - " + name);
    }

    public void startElement(String name, AttributeList attbs) throws SAXException {
        System.out.println("Start of Element - " + name);
    }

    public void processingInstruction(String t, String d) throws SAXException {
        System.out.println("PI target - " + t);
    }

}
2.5 LotusXSL

LotusXSL is an XSLT processor which is included with Domino and Domino Designer. This XSLT tool was specifically implemented for the W3C Recommendation of 16 November 1999, the XSL Transformations (XSLT) Version 1.0 and XML Path Language (XPath) Version 1.0.

Originally, the LotusXSL processor was developed by Lotus, but has now been donated to the Apache Software Foundation and its name has been changed to “Xalan”. The LotusXSL.jar, which we can use from Domino, includes Xalan and also its wrapper package.

LotusXSL is very important for transforming XML to other XML documents or HTML documents. In this redbook, we use the term XSLT in many places; we could have used LotusXSL just as well in every place where we mention XSLT processor. When we use this XSLT processor, we must use XML Parser for Java at the same time.

Let’s see how LotusXSL is used in Domino. The following code is a Domino Java Agent which will do XSLT processing using LotusXSL:

```java
public class JavaAgent extends AgentBase {

    public void NotesMain() {
        try {
            XSLProcessor xp = new XSLProcessor(new XML4JLiaison4dom());

            String x = "http://p.moreover.com/cgi-local/page?index_food+xml";
            String s = "http://wickedxml/fishnet/fishnet.nsf/nfss.xsl";
            StringWriter rw = new StringWriter();
```

As this example shows, we use LotusXSL classes and transform the XML file with an XSL file written in another place. The details of LotusXSL are described in Chapter 5, “Getting XML into Domino” on page 237.

2.6 Security and XML

XML is a useful technology as a universal data format. Since this data is written in plain text format, it is important to consider its security.

There are several ways of achieving the security goals of an organization. Following are the major aspects of security requirements:

- Data integrity
- Confidentiality
- Authentication
- Identification
- Non-repudiation

There are many security technologies available to fill these security requests. The major technologies include the following:

- Encryption
- Digital Signature
Domino provides great security for storing and distributing data, so Domino is also a suitable platform for storing XML from the standpoint of security.

2.6.1 XML security for stored data

Since XML is written in plain text which everyone can read, we need to consider how we store the data. Domino is an excellent choice to provide the necessary secure storage.

XML data is stored in a Domino database as a Domino document, so we can make the most of Domino’s security features, which are described in this section. We can also import XML data into a .nsf format using the toolkit.

2.6.1.1 Authentication

When we want to access a Domino server, we have to authenticate against the server. In cases where we will permit access from a Web browser, we can easily configure the Web server to use not only basic authentication but session authentication. Session-based authentication is one of the new features of Domino R5; this new security feature provides automatic logoff, user monitoring, limits the number of accessing users, and provides “Single Signon” to one server (in contrast to basic authentication, which requires reauthentication when the user moves their directory to a different so-called “realm”). We can set session-based authentication on the Domino Server configuration document in Domino Directory.

Authentication helps the administrator not only with access control, but also with detail logging. Domino provides detail logging of access to both servers and databases.

2.6.1.2 Encryption

Domino provides strict encryption to the data, which can be implemented easily by any developers. Developers can encrypt both entire databases or individual fields with Domino keys.

Actually, we do not need to encrypt data when we save it in a Domino database since Domino database has strict access control. Encrypting data protects it in cases where the database is accessed or copied directly through a file system.

To encrypt a field, we need to use our user ID key, which is included in the userid file. Therefore, Web users who do not have a valid userid file cannot
see the encrypted field. In this example, we will use the encryption of the database itself, with a server key we can set from the database property dialog box, as shown in Figure 26.

![Encryption for FileNet XML](image)

**Figure 26. Setting for database encryption with server ID**

With this encryption, only the server ID can directly access the database. We can access the database via the server, but not directly from a file system.

Database encryption is one of the keys to maintaining data security. Another technique is using XML Security Suite, which we can download from Alphaworks. XML Security Suite provides encryption and digital signature for XML documents. With this tool we can encrypt the XML data itself, without using Domino security features. However, it is usually much easier to make the most of Domino security features to protect a database.

### 2.6.1.3 Access control for databases

One of the most important aspects of Domino security is the ability to control who can use a Domino database; this control is provided by the access control list (ACL). We can define up to seven levels of access control for any Domino database.
When we use many types of forms in one database, for example to handle many DTD or XML documents, using the role option also help the data keep safe.

2.6.1.4 Access control for documents
Domino security provides not only access control for databases, but also for documents. We can use the Readers field to control this.

2.6.1.5 Access control for file system
Domino R5 server provides the function of access control for file systems, which is similar to that of Domino. If we use the file systems for XSL or DTD files, it would be advantageous to use this function.

2.6.2 XML security as communication medium
While Domino provides security for data that is being stored, we must also consider the security of data while it is being exchanged. Since XML is a plain text format, we need to encrypt the data to maintain security.

We can use either of the following two encryption schemes to protect the data:

- Encrypt the data itself
- Encrypt the protocol data on the transport layer

Encrypting the data requires some encryption tool which is a private tool for XML. XML Security Suite is a good choice; it is a tool that provides encryption and digital signature for XML documents.

Encryption at the transport layer can make the best use of Domino server. We usually use Secure Socket Layer (SSL) to encrypt the communication data on transport layer.

2.6.2.1 What is SSL
SSL is a network protocol which is now the de facto standard of encryption technology on the Web. Although the main way of using SSL is encrypting the HTTP connection, we can also use SSL for TCP/IP protocol for LDAP, POP3, IMAP, SMTP and so on. Using SSL, we can implement the following security features:

- Encryption of the communication between the client and the server
- Authentication of the server
- Authentication of the client
Encryption is one of the major functions of SSL and it is very easy to use, especially in Domino. In SSL communication, all messages are encrypted with a unique secret key which is exchanged with encryption at the first communication using the server’s public key. In this case, the server has its certificate and the client trusts the server. This is a server authentication type of SSL. SSL server authentication is usually enough, used with encryption, to keep security. In this case, authentication to the server is used as the basic authentication.

For more strict security, we can also use client authentication SSL. SSL client authentication has a similar architecture to that of Notes Client authentication. We use X.509 certificates to authenticate. Using SSL client authentication is very safe because it does not use passwords to authenticate the client.

Basic and session-based authentication is usually used for communication within intranets. On the other hand, SSL is often used on the Internet. If data needs strong security, we need to use SSL even though Domino provides secure storage.
Chapter 3. Getting XML out of Domino

Domino is a great tool for building and organizing pages of structured and non-structured information. When this capability is added to existing security, directory, and application development features, it's easy to see why Domino is such an effective tool for building Web sites. All of the features that make Domino a superior Web application tool also make Domino an incredible XML storage and generation tool.

There are many methods for extracting XML from Domino. XML can be added to Domino forms, pages and views, and can also be generated by agents based on data in Domino documents. XML for Views can be updated automatically by using the built-in ?ReadViewEntries URL command. Also, if you need to apply custom elements to view entries or multiple Domino data documents, there are several flexible methods for wrapping your own XML tags around Domino data. Finally, the Lotus XML toolkit can be used to to extract XML data using Java and C++.

This chapter provides examples of using each one of these methods, including important hints and tips for making each one work. We will also outline best practices for security and performance when extracting XML from Domino and other database sources via Domino.

3.1 Getting XML out of Domino forms

The simplest and quickest method of transforming Domino data to XML is to create a form that contains fields that you want to convert to XML data, then add tags on either side of the field that will be interpreted as XML. We will show a simple example of this later in this chapter. However, as with most solutions that are quick and easy to develop, there is a price to pay for using this method. The cost comes in future inflexibility and maintenance. Since XML elements are hard-coded and directly associated with field names on the form, changing the field name on the form or associating a field with another XML element becomes confusing and difficult, especially if you're using a large number of forms to generate XML.

A more flexible solution is to build agents that run on a regular schedule or when a form is opened via the form WebQueryOpen event to generate XML on the fly. While this requires more work on the front-end, the agents themselves can provide automatic XML generation tools based on form data. These agents can be written in LotusScript or Java, as seen in examples later on in this chapter.
3.1.1 Applying XML to existing forms

At the time of writing, because of the need to enable the “Treat document contents as HTML” setting in Domino forms to generate well-formed, properly parsed XML, there is no way to create one form that can be read by Notes clients, Web browsers, and XML parsers. The XML form must be created and maintained separately from forms that can be interpreted by Notes and/or browsers. This creates some maintenance headaches because developers will commonly have more than one form to maintain for the same data. However, Domino Web application development features more than make up for this inconvenience with the flexibility in methods to generate XML.

3.1.2 Using the same document info for XML, Domino, HTML

Starting with Domino R5, there is a new @ formula, @BrowserInfo. This solves a great deal of problems when choosing the correct form to display document data between Notes clients and Web browsers. Unfortunately, there is no implicit way to decide if a Web client accessing a Domino server should be served XML or HTML, so for now we have to code XML requests explicitly. However, Domino has several methods for enabling self-service requests for XML data from Domino documents. These include flexible form selection that can be applied to Domino data based on input, or selection of a form based on URL commands to serve agents and documents.

3.1.3 Tips & tricks for generating XML from Domino forms & fields

Text that can be parsed by an XML parser is known as well-formed XML, and conversely, text that an XML parser cannot understand is known as badly or poorly formed XML. When building a Domino form that is parsed into XML, it's important to set the form setting “Treat document contents as HTML” on. Figure 27 on page 77 shows this.
If "Treat document contents as HTML" is not enabled, Domino automatically generates some JavaScript in the header of the document and (if you’re editing a document) adds a <FORM> HTML tag to the start of your page before the <?XML version.....?> tag, which renders the XML as HTML. If this is the case, you'll get a page that looks like the example in Figure 28 instead of well-formed XML.

<?xml version="1.0" encoding="#UTF-8" ?>
<Customer>
  <ID>0000002150</ID>
  <firstname>SMITH</firstname>
  <firstname>ARLENE</firstname>
  <Company>HILLSIDE DR</Company>
  <Address>PO BOX 2134</Address>
  <Zip></Zip>
  <Zone>Old Town</Zone>
</Customer>
If you look at this example, you may think it looks like XML, but keep in mind that XML is not intended for human eyes. Browsers have a hard time parsing this source code because the XML tags are not in the header of the page, but in the body, embedded in an HTML form. Because the source code does not explicitly identify this code as XML at the very top of the page, the browser interprets the XML tags as text. The browser cannot distinguish between the document data and the XML tags that have been added by the XML form. To the browser, this is simply an HTML page.

Figure 29 shows what the same Domino document results look like when "Treat document contents as HTML" is on.

![Well formed XML in IE5](image)

This example looks very similar to the non-XML text displayed in the previous example, but instead of text this time, the browser has parsed the data and displayed it to us as well formed XML. We can tell this because the field names are now indented inside the customer element, and if you could see this diagram in color, you would see the data in black, the brackets in blue and the elements in red. It should be pointed out that this is the standard display format for XML as generated by an IE 5.0 browser. If you’re using some other browser, the displayed results may vary, but the XML data elements will probably be visibly parsed in the same way.
Chapter 3. Getting XML out of Domino

One other important caveat when generating XML from Domino documents is to watch out for non-supported XML characters in field name strings. When an XML parser encounters a $ in an element name, for example, it will return an XML format error. This can be a particular problem for Domino system fields in documents, which often start with a $, such as $UpdatedBy. Another problem is case sensitivity of elements in an XML stream. There are several ways to deal with these issues, and we outline each way in the examples later in this chapter. The “trick” here is that by using IE5+ to view our XML we can quickly tell whether it is well-formed or not.

3.2 Example: A simple XML form with tags embedded

Simply creating a form that adds tags to either side of a field is the easiest way to generate well-formed XML in Domino. In the following example, we have added tags to the XMLCustID form in the fishnet query example database, which displays customer information. When the form is applied to Domino document customer data, the XML embedded in the form is wrapped around the data.

![Image of the basic XML form](image)

Figure 30. The basic XML form

When hard coding XML, it is very important to make sure that all values are in quotes, and that the beginning and ending quotations match. For example, this string has 4 errors:
<?XML version 1.0 encoding='UTF-8' ?>

This string would cause the data on a page to be formatted as HTML, not XML. In XML parlance, any XML information passed to the parser is an entity, and values passed in quotation marks (and they have to be in quotation marks) are called attributes. The sample string would fail because the XML version attribute 1.0 is not surrounded by quotations, and the encoding attribute UTF-8 has mismatched quotation marks. HTML will generally be more forgiving with quotations and case, but XML parsers are not, as they have to read the data and manipulate it. Finally, the version attribute is supposed to have an equal sign and 1.0 is supposed to be in quotations. Here’s the correct format for comparison:

<?xml version="1.0" encoding="UTF-8" ?>

After setting the XML entities and attributes, all the code in brackets that related to data on the XML form are called elements. Element names must be hard-coded with the same case sensitivity. In this case all elements have a case format that matches the case of the actual field names on the forms. This way the data is parsed correctly into an XML stream. It’s important to be careful when hard coding field names since matching letter case must be kept between the start and end tags of all XML elements to avoid XML parsing format errors, unlike HTML. For example, this string would be parsed as an incorrectly tagged element:

<id>745968963989</id>

ID and id would be interpreted as two different element names when parsing XML. To the parser it would look like we’re trying to establish a new id element inside the ID element, which would generate a format error. For this reason, it’s important to decide before going too far into your XML coding what case format all element names should be in. With few exceptions, you’ll find that the de facto standard is all lower case lettering. Data in between the element names does not have to be standardized, because, of course, it’s data and should be left the way it is. Lotus products have shielded developers from the vagaries of dealing with case sensitivity for a long time now, but they can protect us no more! Web technologies, including JavaScript and now XML, are case sensitive. Experienced Domino developers should look for case sensitivity problems first when debugging XML format errors.

Another important caveat is that field data formats should always be passed to the XML parser as text. Numeric, rich text, or any other non-text Domino data format will cause an error in a rich text parser. For hard-coded XML forms, the @Text function can be used to convert non-text formats to text, and LotusScript has a handy .text property which we cover later in this chapter.
3.3 Using LotusScript agents to generate XML

Although R5 XML methods are available in Java only, they are mostly used with data transformation using XSLT engines, a technique we get into in more detail in the next chapter. There may be times when you need to generate XML on the fly, without hard coding XML directly onto a form. In this case, a LotusScript agent that is either scheduled to run periodically or triggered by the WebQuerySave event is the best way to go.

3.4 Example: Storing computed XML data in a field on a document

In this example we show an agent that reads through all the fields on a form, and calculates element names and values for all of them. This is more flexible than hard coding fields and element names on a form, because to add or remove an element in the XML data the developer just needs to add or remove a field on the form. The agent picks up the fields each time the document is opened via the WebQuerySave event on the form, and the XML is generated based on the form's field names at that moment.

The following is the agent code in its entirety:

Sub Initialize
    Dim session As NotesSession
    Set session = New NotesSession
    Dim doc As NotesDocument
    Set doc = session.DocumentContext
    AllFields_LotusScript=""
    Forall i In doc.Items
        If ( Instr( i.name, "$" ) = 0 ) Then
            AllFields_LotusScript = AllFields_LotusScript + "<" + i.name+"">" + i.text + "</" + i.name+"">" + Chr(13)
        End If
    End Forall
    doc.AllFields_LotusScript=AllFields_LotusScript
End Sub

As you can see from the brevity of this agent, LotusScript has some distinct advantages when generating XML from Domino document data. This is all the code required to format all the fields on a form and serve them to a parser in XML. Let's have a look at the meat of the code line by line, and review the features:

    Dim session As NotesSession
    Set session = New NotesSession
Dim doc As NotesDocument
Set doc = session.DocumentContext

The preceding code simply sets the context for the agent by finding the document that is currently being served to the Web browser. This is the current document that called the agent via the WebQueryOpen event, and contains the data that we want to format as XML. The following code simply traverses all the fields in the items property of the NotesDocument and generates XML elements based on the document fields names. It also places the value of each field between the newly created element tags.

AllFields_LotusScript=""
Forall i In doc.Items
    If (Instr(i.name,"$" ) = 0 ) Then
        AllFields_LotusScript = AllFields_LotusScript + "<" + i.name+"">" + i.text + "</" + i.name+"">" + Chr(13)
    End If
End Forall

We should review a couple of options that could be used at this point. Note that we're using the text property of the Domino data items to pass to the XML browser. As we mentioned previously, XML parsers must be passed text in order to convert data to an XML format, and the .text property is an ideal way to handle this. Also, as explained previously, XML cannot deal with non-alphabetical characters, such as the $ that is a part of most Domino systems fields. Because the items property of NotesDocument contains all fields in a form, including system fields, this agent checks for field names that include a $ and simply excludes them from the XML data that is generated. There are other techniques to deal with this situation, as the next example illustrates.

Forall i In doc.Items
    XMLCompliantName = Evaluate("@ReplaceSubstring(i.name;"$";""'), doc)
    AllFields_LotusScript = AllFields_LotusScript + "<" + XMLCompliantName +"">" + i.text + "</" + XMLCompliantName +"">" + Chr(13)
End Forall

The preceding code converts Domino system field names to a format acceptable by XML parsers by simply removing the $ from the string, so $UpdatedBy becomes UpdatedBy.

However, there are often cases where Domino developers have used a field name similar to a system name to store variables. In this case, removing the $ will cause a duplicate XML element name, which will also cause an XML format error. A better way to substitute field names is something like this:

Forall i In doc.Items
XMLCompliantName = Evaluate("@ReplaceSubstring(i.name; ""$"";""dollarsign"")", doc)
   AllFields_LotusScript = AllFields_LotusScript + "<" + XMLCompliantName +"">" + i.text + "</" + XMLCompliantName +"">"+Chr(13)
   End Forall

In this example, $UpdatedBy would become dollarsignUpdatedBy, an acceptable XML format, and also a useful taxonomy for converting XML elements and data into Domino should this document find it's way back to your Domino database. Also, you may want to check for illegal XML characters in your text fields, such as < or >, which could render your XML poorly formed.

doc.AllFields_LotusScript=AllFields_LotusScript

The preceding code is the last line of the agent. It simply stores the data in a field on the form called AllFields_LotusScript. Actually generating a value for this data in XML is unnecessary, so it could also be programmatically excluded from your XML data.

3.4.1 Generating computed document XML via an agent

Now that we understand what the agent does to generate XML from Domino data, let's review the options for calling the agent and serving the XML to a parser. The agent could be run nightly to enhance performance. One other way to accomplish this is to attach a WebQueryOpen agent to a Domino form, so that each time the document is opened on the Web via a URL link, the agent runs and updates the field that contains the XML data. To use this option, as with the hard-coded form example in the previous section, it is necessary to turn the form setting “Treat document contents as HTML” on. Figure 31 on page 84 shows a form that uses the agent to serve XML data.
In addition to having the “Treat document contents as HTML” setting on, the fields that will be used to generate the XML have to be hidden from Web browsers by selecting the Hide Paragraph from Web Browsers setting in the text tab for those fields.

The WebQueryOpen event takes place when the document is opened on the server, but before the document is displayed to the browser. This allows the agent to generate the XML, but does not display the fields to the XML parser.

Figure 33 on page 85 shows the results of the XML agent being applied to a document via the WebQueryOpen event of a properly formatted Domino form.
Chapter 3. Getting XML out of Domino

3.5 Getting XML out of Domino Views

Forms and documents are great for generating XML that represents a single document of data. However, in many cases, XML that has been generated by Domino must represent multiple documents. When dealing with nested XML elements, one can begin to see the complexities associated with representing this data in a hierarchy of root element, attributes and nested elements that XML parsers can understand.

In a perfect world, there would be one format for all data and we could all use this format to exchange data. In an XML world, this would mean that all parties generating XML data for sharing could agree on a single Document Type Definition, or DTD. However, as the old saying goes, the wonderful thing about standards it that there are so many to choose from. At time of writing it
seems that everyone is creating their own data definitions for XML, which represents their type of data in text format for XML parsing. For example, there are several proposed standards for data interchange based on EDI standards, and several for news and content delivery. Each standard has good features to overcome challenges for that particular type of data, and once you get into the details it's hard to imagine one definition could be applicable for all data types.

In the real world you'll probably end up generating data in several different XML data formats based on the requirements of the parties you're dealing with, mostly your customers or business partners. This will be facilitated either by using XSL to transform generated XML data or by generating XML from Domino in the various required formats. In this chapter we cover the details involved in generating different types of XML from Domino.

There are many ways to get Domino to generate representations of multiple documents. The easiest is to create a view that contains the documents that you want to generate as XML. Once the view is created, there are several ways to create XML data out of the view columns.

Another flexible method is to create a set of multi-value fields that can be manipulated into generating nested XML elements from a single Domino document. In this section we will focus on the methods for generating XML from view data. In the data integration section we'll show you how to generate XML from multi value fields on a Domino document.

### 3.5.1 Using `?ReadViewEntries` on existing views

Of all the methods for generating XML from Domino, using the `?ReadViewEntries` URL command is the easiest, because it is a built-in function of Domino and requires no programming. However, this also makes it the least flexible option for generating XML from Domino data.

While the ReadViewEntries command is an easy way to generate XML, it is only useful if the XML parser used to receive the data conforms to the Domino view XML format. In this case, the system receiving the XML is probably a Domino system. If this is the case, there may be easier ways to transfer the data between two Domino databases without XML transformation and parsing.

Figure 34 on page 87 is an example of a view of customer data in the DECSML database.
Chapter 3. Getting XML out of Domino

Here's the XML generated by Domino for this view when you use the ?ReadViewEntries URL command:

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!-- Lotus-Domino (Release 5.0.4 - June 8, 2000 on Windows NT/Intel) -->
<viewentries toplevelentries="2">
  <viewentry position="1" unid="0B539D0B4268910B8525695E00714130"
             noteid="F2A" siblings="2">
    <entrydata columnnumber="0" name="ID">
      <text>0000001164</text>
    </entrydata>
    <entrydata columnnumber="1" name="Lastname">
      <text>Jones</text>
    </entrydata>
    <entrydata columnnumber="2" name="Firstname">
      <text>DEBORAH</text>
    </entrydata>
    <entrydata columnnumber="3" name="Company">
      <text>SANTA CLARA DR</text>
    </entrydata>
    <entrydata columnnumber="4" name="Address">
      <text>8582 SANTA CLARA DR</text>
    </entrydata>
    <entrydata columnnumber="5" name="Zip">
      <number>75034</number>
    </entrydata>
    <entrydata columnnumber="6" name="Zone">
      <text>Old Town</text>
    </entrydata>
  </viewentry>
</viewentries>
```

Figure 34. Basic customer list in Notes
This example provides a great introduction to multiple XML element hierarchies. Let's dissect this representation level by level:

Viewentries is the root level element for this XML document. Toplevelentries is the representation of how many nested descendant elements are defined within the root level element. In this particular case, the root level element contains two descendant elements.

Viewentry defines the next level of XML elements, one for each element represented in the XML root element. Because views can have multiple hierarchical levels of Domino documents, responses and response-to-responses, this definition tells us that we are at the highest level (1) and that there are two sibling elements at this level. Hierarchy levels of nested elements are often referred to as ancestors and descendants. The unid is used to reference the document, just as it is in Domino. It is the same
across all replicas of the database. The noteid is similar, but is unique for a
document in each replication of a database, unlike the unid, which is the
same for every replica.

```
<entrydata columnnumber="0" name="ID">
  <text>0000001164</text>
</entrydata>
<entrydata columnnumber="1" name="Lastname">
  <text>Jones</text>
</entrydata>
```

Now we’re actually into the first nested element. The first two data elements
represented here correspond to columns in the Domino view, hence the
columnnumber reference. It’s worth noting that the column numbers in XML
start at zero, not one. The element name corresponds to the Domino field
name, and the text element is the actual data itself.

### 3.5.1.1 ?ReadViewEntries URL arguments

At this point, it’s important to point out some URL arguments that can be used
with the ?ReadViewEntries command to make the XML representation more
useful. Refer to the Domino Designer online help for more details on usage
and examples.

**PreFormat**

This is the most useful argument, by far. It converts data formats to all text, so
that a regular view can be easily converted to XML and used as both a
regular and XML view. If you don’t use PreFormat, @Text and/or the view
setting “Treat all view entries as HTML” must be used to make the data
XML-compatible.

**RestrictToCategory = category**

A close second, this argument permits developers to programmatically restrict
the view display to a single category. The first column of a view must be
categorized to use this argument.

**StartKey = <first column value (represented by a string)>**

**Start = <numbered position in the view>**

**Count = <numbered position in the view>**

**CollapseView**

**ExpandView**

**Collapse = <collapse at numbered position in the view>**

**Expand = <expand at numbered position in the view>**

These commands can be used to generate the XML-generated data in
different formats. They are the same as other view URL arguments, and
format the view by controlling the start document, number of documents, and controlling if and how the view is collapsed or expanded.

3.5.2 Displaying multiple documents in an XML tree structure

As you can see from the previous example, the hierarchical display of the document data represented in just two lines of a Domino view can take a very complicated form when an XML element hierarchy and data format descriptions are applied. This is, in essence, the reason for complicated DTDs that validate data before parsing it back into non-XML data. In this case, the XML tags do a great job of describing the data as seen from a Domino view perspective, but are not very representative of the customer data that the view contains. Because of this, XML parsers that wanted to read this data would have to parse the view structure before being able to manipulate the underlying data values.

In most cases, data will be represented in XML in a format closer to the actual purpose for the data. In this case customer data would probably be formatted in a representation more useful to parsing customer information. The next example shows a view of the same XML data formatted with a customer document representation.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<Customers>
  <Customer>
    <ID>0000001164</ID>
    <Lastname>Jones</Lastname>
    <Firstname>DEBORAH</Firstname>
    <Company>SANTA CLARA DR</Company>
    <Address>8582 SANTA CLARA DR</Address>
    <Zip>75034</Zip>
    <Zone>Old Town</Zone>
  </Customer>
  <Customer>
    <ID>0000002150</ID>
    <Lastname>Smith</Lastname>
    <Firstname>AUBREY</Firstname>
    <Company>HILLSIDE DR</Company>
    <Address>PO BOX 2134</Address>
    <Zip> </Zip>
    <Zone>Old Town</Zone>
  </Customer>
</Customers>
```

As you can see in this example, we have a much more brief XML representation of the same data with the same number of documents.
However, it is not generated using ?ReadViewEntries, so it can be much simpler. The XML generated here represents the root document Customers, which contains two customer documents, which contain several elements representing customer data. This representation, however, makes several assumptions about the data each document contains. For example, it contains no attributes specifying data in a specific format, so all data is assumed to be in text. Because we’re not using the ?ReadViewEntries command, we cannot use the PreFormat URL argument to generate text for all columns in the view before generating XML, so we have to make sure that the data is converted to text before we begin. We do this by enabling the view property “Treat View Contents as HTML”. Figure 35 illustrates the view that generated the customer XML.

![Figure 35. Basic customer list view in Domino Designer](image)

### 3.5.3 Adding XML tags into a view/page design

When creating a view to generate XML, only half the work is done when the view is done. Note that the first column in the view is `<Customer>`, but that is not the root element when XML is generated. The view itself can contain the XML describing individual elements that represent each row of data in the view, but the XML identification tags and the root element information have to be added somehow. Without this information, the view itself will not be parsed and will be rendered as HTML instead, as there is nothing to tell the parser to treat it as XML. In order to add the XML definition and root element tags, the next step is to embed the XML view on an Domino form or page, and add the required XML tags above and below the view. Figure 36 on page 92 is an
example of the same view embedded on a page, with the XML identification tags and the root element tag for `<Customers>` added.

![Image of a page with an embedded view](image)

**Figure 36. Basic customer list view embedded on a page**

After embedding the view on the page, a couple of properties need to be set to ensure proper parsing of the page and view data into XML. The page property “Treat page contents as HTML” should be enabled. In the Embedded view properties, the “Display Using” property needs to be set to “HTML,” from the default, which is “Java Applet.” Once this is done, you just need to add the identification tags for XML and the XML root element tags for the `<Customers>` element, and that’s it!

### 3.5.4 Applying a DTD

Up until now, we’ve seen examples of how to get data out of Domino using various methods, but we’ve only really skimmed the surface of the capabilities of XML. Besides being able to represent text data, XML also allows the parsing and transport of non-text data at the markup language level, effectively representing non-text data as text that can be translated into its native format at its destination. This is really where the rubber hits the road when using XML, and when the XML you see on the screen begins to get very hard to follow. This is because although it’s made up of readable words and symbols, it’s not intended for human eyes to read, but parses very well using a capable XML parser. Not only are there multiple nested elements represented in a hierarchy, each nested element can have one or more attributes, which tell XML parsers what format the data should be interpreted in and where data should go once it arrives at it’s destination.
As with XML elements, methods abound for formatting attributes. There are several native data formats that can be handily described to parsers, but beyond that there can be custom formats for data and also custom formats of forms. These formats are not restricted and can be described in any way the XML developer sees fit. For a good example of XML attributes, let's revisit a fragment of code pulled out of the previous ?ReadViewEntries example.

```xml
<viewentries toplevelentries="2">
  <viewentry position="1" unid="0B539D0B4268910B8525695E00714130" noteid="P2A" siblings="2">
    <entrydata columnnumber="0" name="ID">
      <text>0000001164</text>
    </entrydata>
  </viewentry>
</viewentries>
```

Here we see several elements as defined by brackets: `<viewentries>`, `<viewentry>`, `<entrydata>`, and `<text>`. These all describe XML nested elements. For example, `<text>` describes a descendant element of the `<viewentry>` element. The attributes for an element are anything that follows the element text, which by default must have no spaces. The toplevelentries attribute for the viewentries element tells the XML parser to look for two top level entries in this view when converting data back from XML to Domino. The position attribute of the viewentry element tells the XML parser where to place the data in the first row of the view when it is parsed. The unid and noteid attributes are Domino document identifiers that will be translated into the document unique ID at the data’s destination. The siblings attribute of the viewentry element provides the same information as the toplevelentries attribute at the viewentries element level, but works at more than one level within the view hierarchy to describe elements on the same level. The columnnumber and name attribute for entrydata describe the position and Domino field name for this column in the view. Finally, we see the text element, with no attributes, as it is describing the actual value in the first field in the XML-generated view.

As you can see from this example, there is an inherent complexity to nesting elements and attributes to represent actual data in XML. Because of this, there needs to be a method for verifying that all the nesting that an XML representation takes is correct. This method needs to conform to the requirements of whatever particular data source the data originated from to ensure that the data made the XML transition in good condition, and may also be needed to ensure that data sent to a target system will be compatible when it's parsed from its XML format. This is what Document Type definitions, or DTDs were made to do. DTDs take the form of a separate file that describes the correct hierarchy and attributes for nested elements in an XML document. It is used to compare the XML a system receives with a
schematic diagram of the XML the system is designed to receive, and validate the XML if it passed, or pronounce the data invalid if it doesn't match.

However, like XML itself, DTDs are essentially text files that conform to certain formats. Because of this, the methods and structure of DTDs is as wide open as the methods and structure of XML.

### 3.5.5 Conforming to the Domino DTD

The Domino DTD is a document that forces validation with any XML document that contains DXL, the Lotus flavor of XML. It's maintained and created by the same people who bring you the LotusXML toolkit, described later in this chapter. At the time of writing, the domino.dtd has not shipped with Domino. It can be downloaded with the XML toolkit from:

www.lotus.com/xml

In order to conform to a DTD, the header of an XML document must look like this:

```xml
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE document SYSTEM 'domino.dtd'>
<!-- DXLExporter version 0.5, schema (DTD) version 0.13 -->
<document xmlns='http://www.lotus.com/dxl' version='0.13'
form='FishNameResults'>
```

It's worth noting that, at the time of writing, the domino.dtd is a work in progress. In fact, here's what the header comments in the domino.dtd file say:

domino.dtd
Version 0.13-in-progress

It's also important to make sure your system has access to both the XML file and the domino.dtd. In the case of our example, the domino.dtd must be in the same directory as the XML file that is being parsed and validated. Also, the XML file must contain, at the very minimum, the DOCTYPE argument to parse properly:

```xml
<!DOCTYPE document SYSTEM 'domino.dtd'>
```

If these conditions are not met, the user will receive error message shown in Figure 37 on page 95.
3.5.6 Conforming to other DTDs

Conforming to an external .dtd file is called external XML data validation, because the process relies on a file external to the XML document itself for validation. For this type of validation, the same issues apply to any .dtd file as they do to Domino.dtd.

By far the easiest way to ensure conformation with a frequently versioned dtd is to create an internal DTD along with the XML data at the time the data is generated. Because of the structure and nature of XML files, the DTD and the XML documents can coexist happily on the same physical document. Here's an illustration of an internally validated XML document:

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE customerlist [
  <!ELEMENT customerlist (customer)+>
  <!ELEMENT customer (name,company)> 
  <!ATTLIST CustomerID type CDATA #REQUIRED> 
  <!ELEMENT name (last,first)> 
  <!ELEMENT last (#PCDATA)> 
  <!ELEMENT first (#PCDATA)> 
  <!ELEMENT company (#PCDATA)> 
]>

<customerlist>
```
Figure 38 shows the results of a validated XML document.

For more details on making sure your generated XML data conforms to external DTDs, and for more details on validation, refer to Chapter 2, “Lotus and XML” on page 43.

### 3.6 Advantages of Domino for data integration with XML

Using Domino to serve XML data from non-Domino sources is not such an obvious solution at first. Most large database systems vendors either support XML directly or have XML integration options in the works. However, these solutions usually involve browser users getting data directly from the original...
data source or a subset. Figure 39 illustrates a simple two-tier Web site architecture.

### 2 Tier XML Structure

![Diagram of 2 Tier XML Structure]

**Figure 39. Typical 2-tier Web architecture**

A two-tier infrastructure most easily facilitates extracting data from a database format and serving it to a browser in a variety of formats, including XML. However, this type of direct access exposes data to security and data corruption risks. For these reasons, this is something that most systems administrators would not allow in a real-world environment.

Domino also allows you to efficiently and securely move data throughout the enterprise and beyond the firewall via replication.

#### 3.6.1 A Web server in the third tier

Most industrial strength e-business Web sites use a third tier to negotiate delivery of data from a back-end data source to a user via a browser client. Figure 40 illustrates a typical three-tier Web site architecture.

### 3 Tier XML Structure

![Diagram of 3 Tier XML Structure]

**Figure 40. Typical 3-tier Web architecture including a Domino server**
As shown in this example, the middle-tier Domino server negotiates connections between the browser client and the original data source. This provides an added physical and virtual level of security and balances out some of the workload of the database server. Users can be authenticated in this middle tier, and raw data can be converted to a variety of formats, including XML, without users having direct access to the back-end data.

3.6.2 Using Domino as an XML store

Domino is an excellent choice for middle-tier XML processing functionality for a number of reasons. Domino’s security infrastructure permits easy setup and maintenance of user and server security permissions. The Web application development features ensure that very flexible mid-tier display formats for data can be rapidly developed. But most importantly, Domino easily facilitates the separation of data and display format at the systems level before HTML is server to a browser client. This enables the application of flexible XML on the Domino server based on Domino forms. These Domino forms can contain @Formulas, LotusScript, and Java agents that calculate complex XML based on business rule requirements. XML formats stored on the Domino forms can be switched on the fly and applied to existing data in Domino documents. This means that the same data can be served in different XML formats at the system level, without having to transform generated XML using complex XSLT functionality. When this flexibility is added to Domino’s many built-in database integration options, Domino becomes the ideal mid-tier XML processor.

3.6.3 The difference between data integration and XML

Before the full potential of using Domino as a XML data store can be seen, it’s important to understand the distinction between data integration and XML. There is a lot of talk in the marketplace about XML being a data integration solution that will let corporate databases connect to each other and communicate without understanding each other’s data structure or format. While this is technically true in a limited way, it’s not an entirely accurate picture of what XML offers. The main thing to remember is that XML is not data integration, and data integration is not XML.

3.6.4 Data integration: getting data out of storage

In order to get a good picture of the difference between data integration and XML, imagine a company, let’s call it reallybigcompany.com, which has a DB2 database with ten million rows of inventory and pricing data. When customers come to the reallybigcompany.com Web site, they want to be able to look up inventory and get a price quotation based on pre-negotiated contract pricing. In order to ensure that customers don’t see other customers’ pricing,
authentication is required on the server before the customer can see any inventory or pricing data at all. When inventory items are queried, the database server calculates pricing based on contractual parameters and displays specific pricing for that customer. The idea here is that the database serves up only a small subset of data to each customer, and not all inventory and pricing data is available.

3.6.5 XML: formatting retrieved data for sharing

XML, like HTML, is a markup language, which formats text. The client that interprets the HTML data is a Web browser, which knows where to place graphics, text, JavaScript, and Java based on markup language instructions. An XML parser runs through the XML it is passed and pulls out data embedded in the XML based on tags. It can then manipulate this data for whatever purpose is required. It's important to note that authentication does not take place at the markup language level, meaning that the parser can only access data served up by the database server. XML by itself cannot make data requests directly into the data. This is the reason for several data security breaches you may have heard of recently in the press. Some companies were placing sensitive XML data outside of their corporate firewalls that was only accessible via unpublished URLs. The problem with this was that there is no way to authenticate who is actually accessing the data. Unauthorized parties discovered the URLs and were able to access data they wouldn't normally be able to get to.

In the reallybigcompany.com example, it is possible to publish all of the ten million rows of inventory and pricing data to the web as one huge XML file. This would make it very easy for clients to access the data using XML. However, the data is then accessible to anyone who found it on the Web and could be manipulated and read by anyone who could parse it with an XML parser. For this reason, a combination of data integration and XML tools are both requirements for any serious e-business Web site.

3.6.6 Options for data integration with Domino

As mentioned at the beginning of this section, Domino is an excellent second-tier resource for an e-business Web site because of the wealth of connectivity options to popular RDBMS servers, as well as the built-in capabilities to build connections to uncommon data types. The great thing about Domino data integration options is that they are almost all called the same way, with similar triggers. This means that one method embedded into your application can be swapped out for another method. For example, a LotusScript agent triggered by a document open event can be changed to a Java agent for the same document open event with no coding change.
required in the base application. While we go through the following example in DECS, keep in mind that the DECS example is just one of the easiest ways to achieve results in Domino, but not the only way. In the next section we explain how to get the same functionality with LotusScript using the LotusScript Connectors Extension library, or LCLSX.

3.6.7 Serving XML-formatted documents with Domino forms & DECS

A great example of using Domino for a mid-tier XML server is combining the data integration of Domino Enterprise Connectivity Services (DECS) to quickly prototype a Website solution that includes data integration with a back-end RDBMS, and Website, XML generation and user authentication using Domino. In this case, the FishNet sample XML Web site application is being integrated with a back-end RDBMS that stores customer and fish data. Data is served when a user or server logs on to the Website and fills out query forms. The query forms trigger a data query via DECS, which formats the data in HTML or XML on the front-end, depending on the needs of the query.

The Domino database that illustrates these examples is called DECSML.NSF. It contains a few views and forms for queries and results that trigger DECS queries on the back end. DECS handles the translation of queries and data sources via DECS activities and connections. For the purposes of this example, DECS activities watch Domino forms in a specified database and trigger back-end queries when a form is opened. DECS connections tell the activities where to look for back-end data and pass security parameters to the RDBMS for authentication.

3.6.8 The DECS environment: administration database settings

DECS is a code-free data integration tool that ships for free with R5. DECS is a server task that can be started automatically when the Domino server starts, as long as DECS is in the TASKS line of the server's NOTES.INI, or it can be started manually from the server console via the LOAD DECS command. The DECS server task is controlled by the DECS Administration database, DECSADMIN.NSF. It is created automatically the first time that DECS is started. It contains DECS connection and activity documents that are used via an easy to use wizard, or manually.

All our example data is stored in an ODBC-compatible data format on the same physical machine as the Domino server, although this is not a requirement for DECS. The tables are referenced via a custom ODBC driver on the Windows NT server. A DECS connection document is created to point
to the ODBC reference when queries are triggered by DECS. This is illustrated in Figure 41.

![Figure 41. DECS fish connection for the XML sample database](image)

Once the ODBC connection is established, the next step is to establish the DECS activities, which will trigger queries on the database using the DECS connection when a document using a specific form name is opened on the server. Figure 42 on page 102 shows the fish name activity, which will trigger a lookup on fish data when a document using the form name FishNameResults is opened. The query uses the information in the FishName field to find the first matching value in the ODBC table's ProductName column. If there is a successful match the data in the ODBC table is mapped to the fields in the Domino document using the mapping layout specified in the DECS activity.
3.6.9 The DECS application database: Formatting results

Once the DECS connection and activity documents are set up, the application itself can be developed to accept query data and format data after queries are returned.
To use our example, query data is entered via a Domino query form with a few fields for data, as shown in Figure 43. Depending on which button on the query form is pressed, a different query can be activated. When the query form is saved, the $$Return field reopens the form via a URL ?OpenDocument command, which triggers the query. This is the $$Return in the Query form shown in the previous illustration:

"[/DECSML.nsf/Query+Results/] + ButtonPressedValue + "?OpenDocument]"

Figure 43. DECSML.NSF fish query form

Figure 44. Fish query form in Domino Designer with $$Return code
$\text{Return}$ is a Web-only field in Domino that returns a response to a Web browser after the document is saved to the Domino server. This response is formatted in @formula syntax, but is generated by the browser, so it must be rendered as HTML. The response can take the form of any valid HTML, so this line simply instructs the Domino server to reopen the document that it just saved in the database. It does this by finding the first matching value in the Query Results view and opening it using the ?OpenDocument URL command. Figure 45 shows the Query Results view, listing query data in the first column. The first column is based on a value saved on the form, which represents the query value regardless of the form name.

![DECSML Query Results View](image)

Figure 45. Query results view of the DECSML database

When the $\text{Return}$ reopens the document, the DECS activity associated with this form automatically performs a query and adds back-end RDBMS data into the document. Even without DECS, this same event can be used via the document WebQueryOpen event to trigger an agent using any other valid Domino agent techniques calling various tools, such as LS:DO, LCLSX, JDBC, CORBA, LC for Java, or application-specific tools such as DB2LSX. Following is an overview of the series of events needed to make a Web Query happen from Domino.
In our example, JavaScript is used to save some information about the button the user pressed to trigger the query and the value that the user has entered for the query. Also, invalid query data can bog down a DECS server with useless queries, so JavaScript is used to ensure that query data is in an acceptable format for the query task it is meant to perform. Here's the JavaScript that is used to check the fish ID before the query is submitted:

```javascript
Function SubmitFID() {
    form = document.forms[0];
    if (form.FishID.value == "") {
        alert("Fish ID is required");
        element.focus();
        return"
    } else {
        form.ButtonPressed.value = "FID";
        form.ButtonPressedValue.value = form.FishID.value;
        form.submit();
    }
}
```

This script is activated when the user presses the Fish ID Search button on the Fish Query form. It first checks for a non-blank field and alerts the user if there is no value to be searched. If there is a value in the Fish ID field, the JavaScript saves a value in a hidden field on the form called ButtonPressed, indicating which button was pressed. The script also saves the search key value in another hidden field called ButtonPressedValue. This hidden value becomes the first column in the Query Results view, which is called by the $$Return to reopen the document, triggering the DECS query.

DECS queries use the form name to trigger a specific query. The WebQuerySave event of the Fish Query form is used to change the form name based on the information saved by the JavaScript. The script itself can’t change the form name, or any other Domino reserved field name. JavaScript can change the value while the document is in the browser, but when the document is saved back to the Domino server, the JavaScript-assigned form name is replaced with the original form name. To overcome this, the WebQuery event is used. Here’s the WebQuerySave event code:

```javascript
@If(ButtonPressed="FID"; @SetField("Form"; "FishIDResults"); ButtonPressed="XMLFID"; @SetField("Form"; "XMLFishIDResults"); ButtonPressed="FN"; @SetField("Form"; "FishNameResults"); ButtonPressed="XMLFN"; @SetField("Form"; "XMLFishNameResults"); @Success);
@All
```

The WebQuerySave event uses the ButtonPressed value saved by JavaScript on the form to change the form name just before the form is saved on the Domino server. As you can see from this code, there are actually four forms
behind the scenes that can be used to trigger one of four different queries. The form name controls which query will be performed because each form is associated with a separate DECS query. HTML data is generated by Domino for regular queries, and XML is generated for XML queries. The format of the generated data is controlled in the forms that are used when the document is opened. Figure 46 shows the XML results for the Fish ID Query.

![Image of XML results](image)

*Figure 46. Fish ID search XML results*

The Fish ID results XML is generated by the XMLFishIDResults form. This form contains XML elements wrapped around Domino document fields, and the form properties setting for “Treat Document Contents as HTML” is enabled. The form open activity in DECS supplies the data from the back-end RDBMS to populate the elements with real-time data. Figure 47 on page 107 shows the form in Domino Designer.
The Query by fish ID assumes there is only one unique fish ID for each fish in the back-end RDBMS, and displays the first matching result. The XML by fish name query performs the same function and generates XML in the same way for one matching query result. However, in the case of a fish name search, there can be more than one matching query result for the same query. For example, there are several types of sharks in the database, each with a unique subcategory. In order for the search functionality to be useful in this case, all matching shark records should be displayed, with their subcategories, if a user searches on the “shark” keyword. This is illustrated in Figure 48 on page 108.
In order to enable this functionality we have to have a mechanism for choosing the form that will display the data after the DECS query is finished serving up data from the back-end RDBMS. In this case, the WebQuerySave event is not useful, because there is no way to know if the query will return single or multiple results until the query is done. Also, the form name must be kept static for DECS to activate the correct query, fish by fish name. In this case, it's best to look to the view for functionality. The View Form formula is a little-known but useful feature that allows the view to control which form will display data based on data in individual documents. Here's the View Form formula for the Query Results view:

```plaintext
@If(@Elements(FishID)>1;@If(@Contains(form;"XML");"XMLMultiResults";"Multi Results");form)
```

This is a nested @IF formula that chooses which form to lay on top of the document data based on the FishID field in that document. The first @If
decides whether or not the FishID field contains a multi-value field with more than one element. When DECS processes multiple values for a query it places multiple rows of back-end data in a single document. Each matching RDBMS record field in the DECS activity maps to a single field in the Domino document, and if more than one matching record is found in the back-end RDBMS it is concatenated in the mapped Domino document field as a multi-value field. The @Elements function returns the number of elements in a multi-value field. This allows us to decide between the original form that will display a single result, or a more elaborate form that will format multiple results. The second nested @IF decides if the system should use a form that displays multiple record data as XML or HTML. If the form name contains “XML”, the XML form for formatting multiple results will be used to display data, otherwise the HTML form will be used. Because the last option in the nested @IF is to use the original form name, single record query results will display as either XML or HTML based on which button the user pressed. This allows the flexibility of using the same Domino document data to generate single or multiple record HTML, or single or multiple record XML.

The form for generating multiple record XML data is deceptively simple; it is shown in Figure 49.

![XML Multiple Results Form](image)

Figure 49. XMLMultiResults form

The form itself has only one field. This computed field uses the underlying document data to wrap hard-coded XML elements around each element in
the Domino document’s multi-value field. In this case we’re imposing a hierarchy on the nested XML elements to generate a root element called “FishList” that contains multiple nested elements called “Fish.” The XML element and attribute codes as well as the XML FishList element are hard-coded directly on to the form itself. The Fish records are generated by a computed field called FishRecord. Here’s a closer look at the default value formula for the computed field:

"<Fish>"+"<FishID>"+FishID+"</FishID>"+"<Fishdescription>"+FishDescription +"</Fishdescription>"+"<Subcategory>"+Subcategory+"</Subcategory>"+"</Fish>
"

In this computed Domino document field, hard-coded XML is wrapped around each element of the FishID, FishDescription, and Subcategory multi-value fields. These are fields that are contained on the original form for this document. Data is mapped to these fields by the DECS activity when the DECS query is performed on the document. The type of query done is determined by the original form name when the document is reopened by the $$Return URL command, as described earlier in this section. After the query is performed and the query results are saved to the document, the view form formula checks to see if the data returned contains multiple records by checking the number of multi-value field elements in a field on the form. If there are multiple records, the multi-value XML form is laid over the top of the document data, which generates well-formed XML based on document fields. Using this technique, multiple value fields become an easy and flexible way to format multi-value XML documents on the fly.

3.7 Other data integration options

Data integration options abound for connecting Domino to back-end data. Because of the flexibility of Domino to generate forms and pages on the fly, this integrated data can be easily generated as XML. Because of this flexibility, the XML generation functionality can be separate from the data integration functionality, meaning that data integration options can be interchangeable. For example, you may want to use DECS to build a prototype, then switch to a more robust data integration option for an e-business Web site. Because DECS watches for the form open event to execute a task on the server, you can easily replace the DECS functionality with an agent in the WebQueryOpen event, and this agent can be written in LotusScript or Java. The code in the WebQueryOpen event could use a variety of methods to trigger back-end RDBMS system queries. These back-end query tools employ a variety of methods for generating query triggers, from LotusScript Extension files to Java classes to C++ applications.
For a complete listing of data integration options that can be used with Domino, refer to the Lotus Enterprise Integration Zone, at:

www.lotus.com/eizone

3.7.1 Data integration example using LCLSX

As mentioned in the previous section, DECS is just one of many tools that Domino developers can use to integrate back-end data with the Web. Once a prototype application has been created, the query functionality facilitated by DECS can be swapped for a WebQueryOpen event agent. This agent can contain LotusScript or Java, and in each of these languages there are several options for connecting to back-end data via Domino. In this example we’ll show you how to use the Lotus Connector LSX (LCLSX) to create the same functionality as the DECS fish ID lookup, but in this case we’ll use an agent triggered by the WebQueryOpen event. To do this we simply copied the form and used the same functionality, but changed the WebQuerySave to open different forms in the same database. A copy of the fish ID query results form was made as well, and a WebQueryOpen event was added that calls a LotusScript agent. This agent uses LCLSX to pull data in from a back-end RDBMS. Here’s the new WebQuerySave event that refers to new query results form names for the LCLSX Fish Query Form:

@If(ButtonPressed="FID"; @SetField("Form"; "LCLSXFishIDResults"); ButtonPressed="XMLFID"; @SetField("Form"; "LCLSXXMLFishIDResults"); ButtonPressed="FN"; @SetField("Form"; "LCLSXFishNameResults"); ButtonPressed="XMLFN"; @SetField("Form"; "LCLSXXMLFishNameResults")); @Success); @All

Figure 50 on page 112 shows the LCLSXXMLFishIDResults form with the WebQueryOpen event highlighted.
Here's the agent code called by the WebQueryOpen event in it's entirety:

```vba
Option Public
Uselxs "*LSXLC"

Sub Initialize
    Dim session As New LCSession
    Dim srccon As LCConnection
    Dim keyValue As New LCFieldList
    Dim QueryResults As New LCFieldList
    Dim keyField As LCField
    Dim NotesSession As New NotesSession
    Dim count As Long
    Dim FishID As String

    On Error Goto ErrorHandler
    Set srccon = New LCConnection("odbc2")
    srccon.Server = "WickedXML"
    srccon.Metadata = "Fish"
    srccon.Connect

    Set doc = NotesSession.DocumentContext
    FishID = doc.FishID

    srccon.FieldNames = "FISHID,FISHNAME,SUBCATEGORY"
    Set keyField = keyValue.Append("FISHID", LCTYPE_TEXT)
```

Figure 50. LCLSXXMLFishIDResults form with WebQueryOpen event highlighted
keyField.Flags = LCFIELDF_KEY
keyField.Value = FishID
If Not (count = 0) Then
    count = srccon.Fetch(QueryResults)
Else
    MessageBox "No matching records found for Fish ID " + FishID, MB_OK, "No Matching Records Found"
End If

doc.Form = "XMLFishIDResults"
doc.FishID = FishID
doc.FishName = QueryResults.FISHNAME(0)
doc.SubCategory = QueryResults.SUBCATEGORY(0)
Exit Sub

ErrorHandler:
If (session.status <> LCSUCCESS) Then
    MessageBox session.GetstatusText, 0, _
    + "The following Lotus Connector error has occurred."
Else
    MessageBox Error$, 0, _
    + "The following LotusScript error has occurred."
End If
End

Now let's go through the code to show you the highlights of the LCLSX functionality.

Option Public
Uselsx "*LSXLC"

This code goes in the Option event of your LotusScript agent and instructs LotusScript to include the LC LSX functionality in your agent. As of this writing, this refers to the NLSXLX.dll in your notes subdirectory, which ships with Notes 5.02 and higher. The * wildcard character at the beginning allows for future compatibilities with any file matching this *LSXLC pattern. Now let's get into the meat of the code in the initialize statement:

Dim session As New LCSession
Dim srccon As LCConnection
Dim keyValue As New LCFieldList
Dim QueryResults As New LCFieldList
Here we set the initial variables we will be working with in the agent and their data types. LCConnection, LCSession, LCFieldList, and LCField are LCLSX classes. LCSession represents a Lotus Connector Session. It lets you get information about the connector environment on the server that the agent is running. LConnection represents a connection from the agent to a back-end data source via a specific Domino connector. LCFieldList represents metadata associated with the back-end data source field structure and is specified later in this agent. It is used to map back-end data source field structures. LCField represents one or more field values. The last line in this code segment instructs the agent to pass any errors, either LotusScript or LCLSX errors, to a subroutine at the bottom of the agent called ErrorHandler. We also need a value from the current document, so we need a NotesSession to capture document context, an ID variable for the FishID, and a count to be used to track how many records were found as a result of the back-end query. Now that we've defined our variables and error handling methods, we can get to work on the query:

```vbscript
Dim keyField As LCField
Dim NotesSession As New NotesSession
Dim count As Long
Dim FishID As String

On Error Goto ErrorHandler

Set srccon = New LCConnection("odbc2")
srccon.Server = "WickedXML"
srccon.Metadata = "Fish"
srccon.Connect

Set doc = NotesSession.DocumentContext
FishID = doc.FishID
```

This code specifies the connection type by telling LCConnection to use the ODBC2 Domino connector to connect to a back-end data source. We also instruct LCConnection to use the custom ODBC driver on the WickedXML server called Fish. The custom ODBC driver is mapped to a specific table in the Fish database. This provides us with all the column names in the table, which we can use as our query metadata. Now that we have the database located and we know where to look, let's specify what we're looking for:

```vbscript
Set doc = NotesSession.DocumentContext
FishID = doc.FishID
```

The first thing we need to do is get the Fish ID from the current document so we can pass it to the query. The DocumentConext method of the LotusScript NotesSession class facilitates this for us by getting current session and document information. Because we are working with the WebQueryOpen of a document, the document does not have to be explicitly specified. Now that we
have information on what we are looking for, let’s prepare the query by specifying the key on the back-end that the query will be based on:

```lisp
srccon.FieldNames = "FISHID,FISHNAME,SUBCATEGORY"
Set keyField = keyValue.Append("FISHID", LCTYPE_TEXT)
keyField.Flags = LCFIELDF_KEY
keyField.Value = FishID
```

The first thing we do is create a list of fields we want to retrieve from the back-end RDBMS table. This is added to the FieldNames property of the LCConnection class. The next step is to specify the key column on the back-end RDBMS to specify a matching back-end data value for FishID. In this case we’re using the FISHID column and instructing the LCFieldList object called KeyValue to treat this value as LCTYPE_Text, which is a string variable type. Next we set the KeyField.Flags property to LCFIELDF_KEY, which instructs the query to return key values in the back-end RDBMS FISHID column that match that FishID variable exactly. The last variable to be set is the query value itself, passed from the current document. Now that we have specified the key format and values to look for, let’s see if anything matches in the back-end database:

```lisp
If Not(count = 0) Then
    count = srccon.Fetch(QueryResults)
Else
    Messagebox "No matching records found for Fish ID "+FishID, MB_OK, "No Matching Records Found"
End If
```

Here is where we do the query work itself. To check to see if there are any matching records we first execute a select, which will not return data, but a count of matching records. Because we have specified to match key values exactly on the back-end RDBMS, the query should return no more than one unique Fish ID per query, but could return more. If no records are found, the select returns a zero. If this is the case, we don’t task the server by performing a fetch, instead we throw up a messagebox that tells the user that there are no matching records found for the Fish ID that was queried. If one or more records are found, the fetch is performed and the data is fetched into the LCFieldList called QueryResults. Now we need to add the query results to the current document:

```lisp
doc.Form="XMLFishIDResults"
doc.FishID="<Fishid>"+FishID+"</Fishid>"
doc.FishName= "<Fishname>"+QueryResults.FISHNAME(0)+"</Fishname>"
doc.SubCategory = "<Subcategory>"+QueryResults.SUBCATEGORY(0)+"</Subcategory>"
Exit Sub
```

Chapter 3. Getting XML out of Domino  115
In this case we already have the current document context via the WebQueryOpen event and the DocumentContext class of NotesSession. Adding the query values from the back-end RDBMS is as simple as assigning the variables and wrapping XML around the values as text. Once we are done assigning values to fields on the document, and there are no LotusScript or LCLSX errors, the agent exits and the document is generated by Domino, which creates a well-formed XML document.

```lcs
ErrorHandler:
    If ( session.status <> LCSUCCESS) Then
        MessageBox session.GetstatusText, 0, _ + "The following Lotus Connector error has occurred."
    Else
        MessageBox Error$, 0, _ + "The following LotusScript error has occurred."
    End If
End Sub
```

The last part of this agent is the error handling functionality. It's important to make sure that we distinguish between regular LotusScript Syntax errors and errors generated by the LCLSX classes. We can determine which type of error has occurred by checking the LCConnector sessionStatus system parameter. This parameter is always LCSUCCESS if no LCConnector errors have occurred. By process of elimination, if there are no LCConnector errors, then the error must be a standard LotusScript error. If we're dealing with an LCLSX error we display the LCConnector status property by using the LCConnector getStatusText() method to a messageBox. Otherwise we pass the LotusScript error$ string.

### 3.8 Java agents and servlets

Java is just one of the many options available to a Web developer using Domino as a development platform. However, outside of the Domino world, Java is usually the language most closely associated with XML. This is mainly due to the abundance of useful classes and objects available to Java developers to develop XML solutions and functionality. Lotus and Iris have begun to add XML functionality to the Domino classes by adding objects, properties and methods to Java classes first. The Domino classes connect to C++ classes on the Domino server to interact with Domino objects. Because of this, expect to see similar objects, properties, and methods for LotusScript objects in the future.
3.8.1 Using an agent to generate computed XML to a browser

Just as with the previous LCLSX data integration example, there are several options for real-time RDBMS queries using Java. In this case, we’re using an exact copy of the example form we used for LCLSX. This time, however, we’re calling an agent that uses a Java Database Connectivity (JDBC) driver to connect to an Open Database Connectivity (ODBC) data source using something called a JDBC:ODBC bridge. Here’s the WebQuerySave event for the JDBC Fish Query Form:

```java
@If(ButtonPressed="FID"; @SetField("Form"; "JDBCFishIDResults"));
ButtonPressed="XMLFID"; @SetField("Form"; "JDBCXmllFishIDResults"));
ButtonPressed="FN"; @SetField("Form"; "JDBCFishNameResults"));
ButtonPressed="XMLFN"; @SetField("Form"; "JDBCXmllFishNameResults")); @Success); @All
```

Figure 51 displays the JDBCXmllFishIDResults form with the WebQueryOpen event highlighted.

Here’s the Java JDBC:ODBC agent code called by the document WebQueryOpen event in it's entirety:

```java
import lotus.notes.*;
import java.sql.*;
import sun.jdbc.odbc.*;

public class GetFishbyID extends AgentBase
{
    public void NotesMain()
```
{  try  
    String url = "jdbc:odbc:fish";
    java.sql.Driver dr = (java.sql.Driver)
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver").newInstance();
    DriverManager.registerDriver(dr);
    Connection con = DriverManager.getConnection(url);
    Session session = getSession();
    AgentContext ac = session.getAgentContext();
    Document doc = ac.getDocumentContext();
    String FishID = doc.getItemValueString("FishID");
    String sql = "Select FISHID,FISHNAME, SUBCATEGORY from fish where LASTNAME = " + FishID + "";
    Statement stmt = con.createStatement();
    ResultSet results = stmt.executeQuery( sql );
    boolean rc = results.next();
    if (rc) {
      doc.replaceItemValue("FishID", FishID);
      doc.replaceItemValue("FishName", results.getString(1));
      doc.replaceItemValue("SubCategory", results.getString(2));
    }
    stmt.close();
    con.close();
  }  
  catch ( Exception e )
  {  
    e.printStackTrace();
  }  
}

Now let's review the code that makes this agent work. The first thing we do is import the various Java packages we have to work with in this agent:
import lotus.domino.*;
import java.sql.*;
import sun.jdbc.odbc.*;

These packages provide the base functionality that is required to make this agent work. The lotus.domino.* classes are located in the Notes.jar file and handle the accessing of UI elements such as the current document. The java.sql functionality translates sql requests into back-end data queries. The sun.jdbc.odbc bridge functionality allows Java-enabled servers to translate native JDBC requests into ODBC queries on back-end databases. Native JDBC functionality is available on a limited number of back-end database systems, including Domino, which ships a JDBC driver with Domino R5. However, many more systems support ODBC, so the ODBC bridge is an important part of enterprise integration functionality. It's worth noting at this point that Domino R5 ships the class files and packages used in this agent with the Domino server, so there is no need to download additional Java packages to get the JDBC:ODBC functionality; it's already installed on Domino R5 servers.

public class GetFishbyID extends AgentBase
{
  public void NotesMain()
  {
    try
    {
      Every Domino agent extends the AgentBase class and has NotesMain as it's root component. If the agent is multi-threaded, the NotesMain is where each new thread starts, so it's easy to follow recursive thread syntax in Domino agents using Java. The next step is to execute the Java thread:
    
    try
    {
      Every Domino agent has a try and a catch within the NotesMain. The next step is to execute the agent which will define the parameters of the connection and the query and retrieve data from the back-end data source.
      
      String url = "jdbc:odbc:fish";

      java.sql.Driver dr = (java.sql.Driver) Class.forName("sun.jdbc.odbc.JdbcOdbcDriver").newInstance();
      DriverManager.registerDriver(dr);

      The previous lines of code set the URL syntax to access the back-end fish database via the JDBC:ODBC bridge. In this case the URL points to an ODBC DSN (Data Source Name) called fish. The next step is to create a new instance of the JDBC:ODBC driver. The JDBC:ODBC bridge is actually a
native JDBC driver that handles ODBC, written by JavaSoft developers. This is similar to the native Domino JDBC driver, which was created to provide a connection to Domino databases, instead of ODBC compatible databases. Once we’ve created a new instance of the driver, we then register the driver within JDBC. Now that we have a driver and know where to point it to retrieve back-end data, we can try a connection to the data:

```java
Connection con = DriverManager.getConnection(url);
```

Once the connection has been made we need to pass query data to the back-end RDBMS. To do this we need to have data from the current document. As with the previous LCLSX example, we are using the WebQueryOpen event so we can retrieve context properties from Domino that will give us the current document associated with the document open:

```java
Session session = getSession();
AgentContext ac = session.getAgentContext();
Document doc = ac.getDocumentContext();
```

In this case, we’re defining a new NotesSession using the `getSession()` object. The next thing to do is to get the context that the agent is running in by using the `getAgentContext()` method of the NotesSession. Within the Agent Context object we can retrieve the DocumentContext for this agent, which gives us access to the current document’s fields though a NotesDocument object. Now we’re ready to pull the query string off the document, in this case the FishID field:

```java
String FishID = doc.getItemValueString("FishID");
```

Using this information we can now construct the SQL we need to query the back-end RDBMS:

```java
String sql = "Select FISHID,FISHNAME, SUBCATEGORY from fish where LASTNAME = "+ FishID + ";"
```

Once we’ve constructed our SQL statement, we need to send the query to the back-end system. To do this in JDBC you need to create an object, and bind that object to a connection. This object is referred to as a JDBC statement:

```java
Statement stmt = con.createStatement();
```

The next line of code uses the new JDBC statement to pass the SQL variable to the back-end data source via the execute method of the statement.

```java
ResultSet results = stmt.executeQuery( sql );
```

The query has now been processed by the back-end database system and results, if any, have been placed in a result set called results. Result sets are
multi-value vectors, very similar to arrays, or multi-value fields. Indexes of vectors are referred to by index number, which is the same concept as elements in arrays.

Because of the construction of JDBC data sources, the record pointer in the data set is set to the top of the data, specified as row zero. In order to find out if any data has been returned, the pointer has to be moved to the next row of data using the next method. If any records are contained in the result set, the rc variable will return one, if not, it will remain at its default, zero:

    boolean rc = results.next();

If data is returned, we are now on the first row. We now need to copy data from the result set into the current document. The ReplaceItemValue method of NotesDocument is used to do this. Values are pulled from the result set vector by index reference and assigned to fields on the document.

    if (rc)
    {
        doc.replaceItemValue("FishID", FishID);
        doc.replaceItemValue("FishName", results.getString(1));
        doc.replaceItemValue("SubCategory", results.getString(2));
    }

Once we are done with the connection and the statement, we have to explicitly close them:

    stmt.close();
    con.close();

Once they are closed, we're at the end of the agent task. The agent exits at this point and the document is served by Domino with the back-end data as objects in the document. As mentioned earlier, there is always a try and a catch in every Domino agent. This catch event simply prints a trace of any error message that is generated while this agent is running:

    }
    catch ( Exception e )
    {
        e.printStackTrace();
    }
3.8.2 The GenerateXML Java method

You may have noticed that the previous JDBC agent code example contains no code for generating XML. It just serves up regular notes data to the browser by querying a back-end database and applying data in a result set to the document in real time via the WebQueryOpen event. This is where the GenerateXML Java method comes into play. Instead of generating XML by hard-coding elements and values in an agent, as we did with the LCLSX example, we can now generate XML on the generated data with a single command. We can also save that XML to a file outside of Domino or as an attachment in a field on a Domino form with greater ease using the GenerateXML command.

3.8.3 Passing XML to an external .XML file

Let's see how the previous JDBC example would look when GenerateXML functionality is added. We'll keep everything the same up to the point where we actually save data to the current document from the back-end RDBMS:

```java
if ( rc ) {
    doc.replaceItemValue("FishID", FishID );
    doc.replaceItemValue("FishName", results.getString(1));
    doc.replaceItemValue("SubCategory", results.getString(2));
}
```

Now let’s add some functionality using the GenerateXML command to create XML out of the newly created document and save the well-formed XML to a text file:

```java
BufferedWriter bw = new BufferedWriter(new FileWriter("c:\temp\document.xml"));
    doc.generateXML(bw);
    bw.close();
```

By adding these four lines of code we were able to create an agent that saves generated XML data from Domino document data, then saves that data to an external XML file. We also needed to import the java.io classes at the top of the agent to handle the bufferedwriter class. BufferedWriter is an extension of the java.io.writer class that facilitates writing of arrays, strings, and single characters. In this case, we’re writing a string to a file. The string represents the XML generated by GenerateXML. java.io is imported into the agent by adding this line to the top of the agent:
import java.io.*;

Let's have a look at the XML that GenerateXML produced in the document.xml file:

```xml
<document form="JDBCXMLFishIDResults">
  <noteinfo noteid="1026" unid="410516635C5D9C4885256975005A4D50">
    <created>
      <datetime dst="true">20001011T122620,32-04</datetime>
    </created>
    <modified>
      <datetime dst="true">20001011T122632,45-04</datetime>
    </modified>
    <noteinfo>
      <item name="FishID">
        <textlist>
          <text>598</text>
        </textlist>
      </item>
      <item name="FishName">
        <textlist>
          <text>John Dory</text>
        </textlist>
      </item>
      <item name="Subcategory">
        <textlist>
          <text /></textlist>
      </item>
      <item name="$UpdatedBy" names="true">
        <textlist>
          <text>Anonymous</text>
        </textlist>
      </item>
    </noteinfo>
  </noteinfo>
</document>
```

This is a good illustration of the type of XML generated by Domino, associated with a document structure. As with our previous example, which was associated with a view structure, it is designed to identify objects in a Domino form and convert those objects to XML elements. The XML generated with a single line of Java code is well-formed and conforms to the Domino DTD. From here the data can be converted to other formats by transforming the XML data using an Extensible Stylesheet Transformation (XSMT) engine, which we will discuss using this example in Chapter 5, “Getting XML into Domino” on page 237.
As we can see from this example, using Java agents and existing Notes classes to generate XML is one of the easiest and fastest ways to develop an XML solution. However, there is a problem associated with this example that needs to be considered at this point. In order to make this example work, all users must be able to trigger agents to run on the server.

In the case of a regular Web site that doesn’t require users to authenticate for access, this leaves open the possibility that an agent could be created or copied onto the server that damages the server through built-in file system access rights associated with the server. Domino server security is excellent, but allowing anonymous users the ability to run agents on a server does open the possibility for system vulnerability if access to the agents is obtained. One way to avoid this problem is to allow agent rights to a limited number of authenticated users on the Domino server, or create and run a scheduled agent on the server that generates XML in batch. This way, a small group of authenticated users or only the server controls the agent, not the user, so you can disallow anonymous users from running agents on the server with no problem. Also, the user will have to explicitly retrieve the XML data from the attachment on the document, and will be prompted as to where to save the file on their local machine, avoiding the need for client file system access for the agent.

### 3.8.4 Storing computed XML data in a field on a document

In order to facilitate this functionality we have to use the `embeddedObject` and the `bufferedWriter` class in Java. This example also highlights a very important point regarding the `GenerateXML` class: it does not currently support rich text fields. When generating XML for a document, it simply ignores them and generates XML for any non-rich text field.
In the case of the form in Figure 52, the GenerateXML method creates an XML file identical to the previous example with or without the rich text field present. This is a current limitation of DXL, which will be remedied in future releases of DXL and the Domino DTD, once a cohesive standard for all types of data that can be stored in a rich text field is finalized. This may be a problem for developers who wish to capture rich text information in generated XML using the GenerateXML command, but for our purposes it works since we use the rich text field as an object store for our XML file only.

The first step is to add code to the current document WebQuerySave agent that will store the computed .XML file in the rich text field as an attachment. This way, the agent is only called by authenticated users who have edit rights to the document. Users with read-only rights can retrieve the XML by detaching and opening the attached XML code, but cannot trigger agents. The next level of security is to adapt this agent to run on a document collection at a scheduled time, and prevent users from running agents on the server completely. However, permitting smaller numbers of authenticated users to run the WebQuerySave agent ensures that data in the attached file is up to date as of the last time the document was saved from the Web. The scheduled agent's data could be out of date, as it's only as current as the last time the agent was run.

Here's the code that runs in the WebQuerySave event:

```java
import lotus.domino.*;
import java.io.*;
public class JavaAgent extends AgentBase {
```
public void NotesMain() {
try {
    Session session = getSession();
    AgentContext agentContext = session.getAgentContext();
    Document doc = agentContext.getDocumentContext();

    BufferedWriter bw = new BufferedWriter(new FileWriter("c:\temp\document.xml"));
    doc.generateXML(bw);
    bw.close();

    if (doc.hasItem("Body")) {
        Item item = doc.getFirstItem("Body");
        item.remove();
    }

    RichTextItem body = doc.createRichTextItem("Body");

    int att = EmbeddedObject.EMBED_ATTACHMENT;
    body.embedObject(att, "", "c:\temp\document.xml", "generateXMLCode");
    doc.save();
}
catch(Exception e) {
    e.printStackTrace();
}
}
}

Now let's go through this code line-by-line:

import lotus.domino.*;
import java.io.*;
public class JavaAgent extends AgentBase {
    public void NotesMain() {

As with the previous example, we're working with a couple of imported files, so we add them at the top. Also, every Notes agent extends agentbase, and contains a NotesMain component.

try {
    Session session = getSession();
    AgentContext agentContext = session.getAgentContext();
    Document doc = agentContext.getDocumentContext();

The next thing we do is get current document properties via the AgentContext class. The document is retrieved using the DocumentContext, exactly the same method used in the JDBC example.

```java
BufferedWriter bw = new BufferedWriter(new FileWriter("c:\temp\document.xml");
doc.generateXML(bw);
bw.close();
```

Now we get into the creation of the XML. In this case we’re still writing a string to the user’s file system containing the XML that we will reattach.

```java
if (doc.hasItem("Body")) {
    Item item = doc.getFirstItem("Body");
    item.remove();
}
```

Now that we have the XML file written to the file system, we can reattach it into the rich text item of the current document. Before we do this, however, we want to make sure that there is not already an instance of this attachment in the rich text field. We do this by finding the rich text item and removing it. We then recreate it using the createRichText item of the document property.

```java
RichTextItem body = doc.createRichTextItem("Body");
int att = EmbeddedObject.EMBED_ATTACHMENT;
body.embedObject(att,**,"c:\temp\document.xml","generateXMLCode");
doc.save();
}
```

Next we create (or re-create) the rich text field and attach the .XML created by the GenerateXML command. After this, we have our catch event, which is standard for all Domino agents.

```java
catch(Exception e) {
    e.printStackTrace();
}
```

### 3.8.5 Using a servlet to serve computed XML

Another way to generate XML from Domino is via a servlet that is triggered by an HTML URL command. Servlets are made up of Java code that has been developed using the Java 2 Enterprise Edition SDK for Servlets. They
function much the same way as applets, but use some very handy back-end classes, mainly for processing data and serving HTML.

Servlets were designed to overcome some of the current limitations associated with Java applets, such as slow download times, poor performance and lack of compatibility on Web clients. Instead of processing data on the client-side, a URL trigger is sent to a server, which runs the servlet. The servlet itself takes advantage of the horsepower of the server to generate data based on queries or computation, then serves results back to the client that called it in HTML. Because results are in HTML, most browsers and clients, including Notes, can handle results faster; and compatibility issues between different versions and flavors of Java Virtual Machines (JVMs) and Java Runtime Environments (JREs) are no longer a problem. Servlets provide significant performance advantages over Domino agents because they are loaded into memory when called the first time, and are not shut down after each use, as is the case with Domino agents.

Until now, this chapter has focused on LotusScript and Java agents that generate XML, which are very useful in a Domino context. Java agents are stored in Domino databases, and can be triggered by a Domino event such as a WebQueryOpen or a WebQuerySave. Agents can also be triggered directly by using the Domino ?OpenAgent URL command. By contrast, servlets are stored on a Domino server, in a directory specified by the server document or the virtual host document. They can only be triggered by URL. Servlet URLs are mapped to the server via its own URL, such as www.lotus.com/servlets/servlet

Aside from the performance enhancements associated with servlets, there may be times when you want to call Domino functionality from a non-Domino server. Servlets are an ideal solution for this type of functionality because of the inherent ease of passing values from a Web page to a servlet and back to the Web page. In the next example, which starts with Figure 53 on page 129, we have created a small HTML form that can be included in other Web sites by simply cutting and pasting HTML source code into an existing Web page. This enables other Web sites to provide access to Domino data via a servlet called by the form's post action.
Figure 53. HTML page that calls a servlet on the Domino server

When the form is submitted by clicking the Fish ID Search with XML Results button, the servlet is called on the Domino server. This servlet opens a database and looks up a document by Fish ID in a view. If the document is found, the servlet checks to see if there is an attachment containing XML in the document. If so, it returns an embeddedObject file and displays the attachment name on the screen.

Let's start by looking at the HTML code for the screen that calls the servlet:

```html
<HTML>
<HEAD>

<SCRIPT LANGUAGE="JavaScript">
<!--
function SubmitXMLFID() {
form = document.forms[0];
if (form.FishID.value == "")
    {alert("Fish ID is required");
     element.focus();
     return ""}
else
```
Most of this code is just standard HTML that formats the page. Let's pull out the main functionality in this form and have a look at it:

```
<INPUT NAME="FishID" VALUE="" STYLE="WIDTH=100"></TD></TR></TABLE>
</CENTER></BODY></HTML>
```

This is the line that creates the input box on the form. The value entered here in the FishID field is passed to the servlet when the form is saved by pressing the button:

```
<INPUT TYPE="button" Name=FishXMLIDSubmit VALUE="Fish ID Search with XML Results" onClick="SubmitXMLFID()"></TD></TR></TABLE>
</CENTER></BODY></HTML>
```

This is the code that creates the button. When the button is clicked, the JavaScript function SubmitXMLFID() is called:

```javascript
function SubmitXMLFID() {
form = document.forms[0];
if (form.FishID.value == ")
```
This JavaScript checks to make sure that FishID is a non-blank field that can be passed to the servlet as a parameter. If it is blank, the user is prompted to enter a Fish ID; if not, the form is submitted, which activates the Web page's POST method and its associated action:

```html
<FORM NAME = "ServletFishQuery" METHOD=post
ACTION="HTTP://WickedXML/servlet/GetFishXML">
```

This action tells the POST command to go to server 9.95.33.252 (The WickedXML server) and run the servlet called GetFishXML. The form itself passes the Fish ID to the servlet, which we will cover in the next section. Because servlets are activated via a URL, we could also call this servlet to look up Fish ID 598 manually by passing the parameter as a URL argument:

```
HTTP://WickedXML/servlet/GetFishXML?FishID=598
```

When the servlet is called, it passes the form as a request stream in MIME format. Parameters can be called from that form via CGI variables, including user client parameters and fields from the form that called the servlet. Here's the Servlet code that controls the query on the Domino server:

```java
import java.io.*;
import javax.servlet. *;
import javax.servlet.http. *;
import lotus.domino. *;

public class GetFishXML extends HttpServlet {

public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException { 

    HttpSession websession = request.getSession(true);
    response.setContentType("text/html");

    PrintWriter out = response.getWriter();

    try {
```
String fishid = request.getParameter("FishID");

NotesThread.sinitThread();
Session s = NotesFactory.createSession();

Database FishDB = s.getDatabase("", "DECSML");
if (FishDB != null) {

    View view = FishDB.getView("JDBCQuery");
    Document doc = view.getDocumentByKey(fishid, false);
    if (doc != null) {
        EmbeddedObject obj = doc.getAttachment("document.xml");
        if (obj != null) {
            out.println("<HTML><HEAD>");
            out.println("<TITLE>XML Record Found</TITLE></HEAD>");
            out.println("<BODY TEXT="#000000" BGCOLOR="#FFFFFF">");
            out.println("<CENTER>");
            out.println("Found " + obj.getName());
        } else {
            out.println("<HTML><HEAD>");
            out.println("<TITLE>Error Processing Query</TITLE></HEAD>");
            out.println("<BODY TEXT="#000000" BGCOLOR="#FFFF00">");
            out.println("<CENTER>");
            out.println("Database not found.");
        }
    }
    doc.recycle();
    FishDB.recycle();
    s.recycle();
    NotesThread.stermThread();
}
} catch (NotesException n) {
    n.printStackTrace();
} catch (Exception e) {
    out.println(e.getMessage());
    out.println("Error caught");
}

This servlet is used to look up a Fish ID passed from the field on a form that called the servlet. Let’s go over the functionality in segments:
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import lotus.domino.*;
public class GetFishXML extends HttpServlet {

These first few lines just set the context for the type of Java that we'll be working with. Java.io is used for the input/output of the servlet, javax.servlet and javax.servlet.http are servlet classes we'll need to run the servlet and pass it parameters over the HTTP protocol. The lotus.domino classes are used to access databases, views, documents and attachments on the Domino server.

public class GetFishXML extends HttpServlet {

public void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

These lines define the base class that our class will extend. The doGet subclass is where we name our request and response objects. These objects represent the request that was sent to the servlet and the response that will be generated by the servlet. All servlets must have a request and a response object. These objects are defined as mime type objects. For this example, the request object will pass the Fish ID from the HTML form that called the servlet. The response object will send data to the screen indicating whether or not the attachment was found in the document.

try {

String fishid = request.getParameter("FishID");
NotesThread.sinitThread();
Session s = NotesFactory.createSession();

Database FishDB = s.getDatabase("", "DECSML");
if (FishDB != null) {

    View view = FishDB.getView("JDBCQuery");
    Document doc = view.getDocumentByKey(fishid, false);

Now we begin the action. This piece of code gathers information from the HTML form that called this servlet and starts up a Domino session. This takes
us through the Notes session. We open the database, define the view and get the document we're looking for using the getDocumentByKey method of the View class.

```java
if (doc != null) {
    EmbeddedObject obj = doc.getAttachment("document.xml");
    if (obj != null) {
        out.println("<HTML><HEAD>\n        out.println("<TITLE>XML Record Found</TITLE></HEAD>\n        out.println("<BODY TEXT="#000000" BGCOLOR="#FFFFFF">\n        out.println("<CENTER>\n        out.println("Found " + obj.getName());
    } else {
        out.println("<HTML><HEAD>\n        out.println("<TITLE>Error Processing Query</TITLE></HEAD>\n        out.println("<BODY TEXT="#000000" BGCOLOR="#FFFFFF">\n        out.println("<CENTER>\n        out.println("Database not found.\n        
    }
}
```

If the document is found, we look inside the document for an attachment called document.xml. If the attachment is found, we return the attachment object as an embeddedObject and display the embeddedObject name on the screen. If not, we display an error message. The rest of the code is cleanup and exception catching:

```java
doc.recycle();
NotesThread.stermThread();
}
} catch (NotesException n) {
    n.printStackTrace();
} catch (Exception e) {
    out.println(e.getMessage());
    out.println("Error caught");
}
```

Because this attachment is retrieved as an embeddedObject and contains well-formed XML, there are a number of options that you could use to manipulate it using methods of the embeddedObject class. Here's a summary
of the methods of the embeddedObject class of particular use to XML developers:

- ParseXML parses the attachment's well-formed XML and creates a DOM document that conforms to the w3c.org specifications. See Chapter 2, “Lotus and XML” on page 43 for a full explanation of DOM documents, their structure and usage. In our example, the following line of code would produce a DOM document called domdoc that conforms to the w3c specifications:

  org.w3c.dom.Document domdoc = obj.parseXML(false)

- Transformation transforms one XML format to another using XSLT. For a complete explanation of XSLT, refer to Chapter 5, “Getting XML into Domino” on page 237. In our example, we create a new XSLT object by finding a stylesheet contained in a text field called XSLStorage in the same document. Then we define the XSL target object and perform the transformation, which writes XML conforming to the stylesheet into the XSLResultSetOut object.

  Item xsl = StyleDoc.getFirstItem("XSLStorage");
  if (xsl == null) System.out.println("xsl is null!");
  XSLTResultTarget out = new XSLTResultTarget();
  out.setFileName(name + "." + StyleName.getText() + ".out");
  xml.transformXML(xsl, out);

3.9 Working with the Lotus XML Toolkit

Lotus has developed a useful tool to assist Domino and XML developers in using XML, and the Domino flavor of XML, called DXL. This toolkit can enable an application to export all information contained in a Domino database to DXL and, conversely, import data in a DXL format into a .nsf database format. DXL is an XML format designed by Lotus. XML code can be considered DXL as long as it conforms to the Domino DTD. At the time of this writing, an updated Domino.DTD ships with each new version of the XML toolkit. The Lotus and Iris resources responsible for creating the XML toolkit are also the resources responsible for maintaining DXL and the Domino DTD. For more information on DXL see 2.2, “Domino XML Language (DXL)” on page 49.

3.9.1 Components

The Lotus XML Toolkit is available on the Lotus XML Web site at:

www.lotus.com/xml

It is a zipped file that contains several subdirectories. These subdirectories contain working files, documentation, C++ build files, and the DXLTools.jar file
for working with DXL in Java. There are also C++ and uncompiled Java
examples that provide a good start for learning syntax and how the classes
and objects can be used to generate and parse DXL code. Also included is
the latest version of the Domino.dtd, an integral component of DXL. It is used
by all DXL classes to validate XML data as valid DXL.

3.9.2 Features

The Lotus XML toolkit is designed to assist C++ and Java developers who
want to develop code that exports and imports data from Domino databases,
and manipulate XML data in the DXL format. The toolkit provides Java
classes, a .jar file, and some sample code. The classes themselves work at
the document level. This means that they import or export whole documents
between DXL and Domino databases. Any additional functionality to
manipulate Domino or DXL data at the field level can be developed using
pure Java or C++, or by coding transformation of data via XSLT.

3.9.2.1 The XML Toolkit and Notes security

When you run an XML Toolkit program, it uses the Notes ID specified in the
Notes initialization file (notes.ini) to access Domino databases. Your program
has the same level of access as the ID it uses. It cannot access anything that
it wouldn't be able to using the same ID with a Notes client or a Domino
server.

To prevent unauthorized access to design elements and data in Domino
databases, the XML Toolkit also implements the following security features:

- Imported design elements are not signed if the application is running
  under a server ID.
- Your application can export and import design elements only if the ID has
designer access. If the ID does not have designer access, you will receive
  an immediate error on export and an error on import when Notes attempts
to save a design element.
- Your application can export only non-private design elements, and then
  only if the database design is not hidden.
- When you use an ID with designer access to import design elements, you
  receive a password prompt.
- Your application cannot export a document if the ID is not on the readers
  list for the form used to create the document.
3.9.3 Java API Classes

The XML Toolkit provides C++ and Java API classes to access import and export functionality with LotusScript classes to follow in a later release. The core functionality of the toolkit is written in C++ which is wrapped by the Java API classes. Java applications using the toolkit must deploy the toolkit runtime library. At this time, the classes and syntax are still in beta. Refer to the Lotus XML Toolkit documentation which can be downloaded from http://www.lotus.com/XML for updated documentation and usage. For now, let's get acquainted with the classes available in Java:

**DXLBase** is the base class for all other toolkit classes.

**DXLSession** represents a toolkit session and initializes a Notes session. It is required by all applications.

**DXLDatabase** represents a Domino database for DXL import and export purposes. It should not be confused with the Java database class and cannot be converted or extended from it. It is not compatible with the database Java class and has a distinct group of the methods and properties.

**DXLException** is the base class for DXLImportException and DXLExportException, which are used to return error information to applications.

**DXLExporter** exports Domino database objects to DXL. Release 1.0 of the toolkit will export:

- Databases
- Database ACLs
- Documents
- Forms
- About Database documents
- Using Database documents
- Item types text, number, date/time, and rich text
- Pages
- Rich text elements:
- Attachments
- Buttons
- Image properties (no content)
• Paragraph settings and margins
• Tables
• Text and text attributes (font, color, etc.)
• Sections
• Links
• Shared actions
• Shared fields
• Subforms
• Views and folders (Calendar views are exported as <note> elements.)

   **Note:** Toolkit applications cannot export view and folder entries, only the design elements.

The following data items have been identified in the DTD, but can only be exported as generic notes, rich text CD (composite data) records, or "itemdata" items, depending on the options specified when exporting data:
• Database scripts
• Framesets
• Image resources
• Java resources
• Navigators
• Outlines
• Note item types not listed above
• Replication formulas
• Simple actions

What cannot be exported:
• Agents
• Agent data
• Script libraries

**DXLExportException** catches any DXL export exceptions and can be used to return or log DXL error messages.

**DXLEportOptions** sets parameters for the DXLExporter class. It creates a new DXLExportOptions class and sets all values to default settings. For a
complete listing of defaults and values, refer to the latest XML toolkit
documentation.

**DXLImporter** imports DXL objects into a Domino Databases. At the time of
this writing, some of the objects supported by DXLExporter cannot be
imported by DXLImporter. Release 1.0 of the toolkit will be able to import:

- Documents
- Text, number, and date/time items
- Rich text:
  - Text and text attributes (font, color, etc.)
  - Paragraph settings and margins
  - Tables (not nested tables)
  - Sections
  - Links
  - Buttons
- Database ACLs
- Forms
- Subforms
- Pages
- Views and folders

In some cases, the imported data does not include everything that was
exported. The following exported data is not yet imported by toolkit
applications:

- Action bars:
  - Button color
  - Font style
  - Shared actions
- Buttons:
  - HTML tags
  - JavaScript code
- Forms:
  - LotusScript events
  - Properties: no menus
- References to shared fields are imported only if the shared field already exists in the database

- Form fields:
  - LotusScript events
  - MIME fields

- Properties: "allow tab to exit field," hide delimiters, literalize, "run exiting event on change," RTL reading order

- Java code

- Pages:
  - LotusScript events
  - Rich text RTL reading order

- Subforms:
  - LotusScript events
  - multiple aliases

- Tables:
  - left margin as percentage
  - merged cells
  - right margin
  - row names

- URL links:
  - computed URL
  - URL target

- View/folder HelpRequest event formulas

**DXLImportException** catches any DXL import exceptions and can be used to return DXL error messages.

**DXLImportOptions** sets parameters for the DXLImporter class. It creates a new DXLImportOptions class and sets all values to default. For a complete listing of defaults and values, refer to the latest XML toolkit documentation.

### 3.9.4 Developing a stand-alone Java XML application for Domino

The Lotus XML Toolkit is designed to export and import data between Domino documents and DXL elements. It is designed primarily to be used to develop C++ applications and Java servlets, applications and applets outside of the
native Domino development environment. For our example we have developed an application which uses the java.awt (Advanced Windowing Toolkit) classes developed by Sun to create a platform-independent interface.

This interface opens the FishNet database and lists all the fish records that are available for extraction from the Query Results view. When a user double-clicks on one of the fish records, the application retrieves that record in DXL format and saves the record in a text file on the local machine. That text file is then read into the Java UI and displayed to the user in a text area of the application window. Because the output in the window is a text area, the DXL document can be cut and pasted from the UI or extracted from the text file on the local machine directly to an XML parser.

3.9.4.1 XML Toolkit and Domino system setup

Before we begin working with the XML Toolkit, we need to make sure that the toolkit files are in the right place. At the time of this writing the toolkit installs into a LotusXML subdirectory of a directory you choose when installing via a self-extracting .zip file. To access DXL functionality from Java, the DXLTools.jar file must be added to the CLASSPATH. The relative location for DXLTools.jar is \LotusXML\lib\DXLTools.jar. For simplicity, we moved it to the \notes directory of the development machine, where Notes.jar and NCSO.jar are stored. Also, if you get any Java runtime errors indicating libraries are not...
found, they may be fixed by moving any .dll files from LotusXML\lib\mswin32 to the \notes directory on a Notes client machine or the \domino directory on a Domino server, and make sure this directory is included in your PATH environment variable. Finally, the FishNet.nsf database must be in your default data directory for the file references in the Java application to work.

### 3.9.4.2 Java application environment and setup

Unlike Java agents, servlets and applets, there are several things about the development and runtime environments that have to be configured to make a standalone Java application operable. A Java development and runtime environment has to be present on the computer that will be used to compile and run the application. This application has been compiled and tested using JavaSoft’s Java Development Kit (JDK) tool version 1.1.8, downloadable for free at [www.javasoft.com](http://www.javasoft.com).

![Figure 55. JDK 1.1.8 Window showing compile & run commands for GetFishDXL](image)

By default the application installs in the jdk1.1.8 directory. The \bin subdirectory is where the javac command line compiler and the java command line runtime utilities are located. The source code should be installed in a jdk1.1.8\bin\XMLSource subdirectory to follow this example exactly. Once the source code is in the XMLSource subdirectory, GetFishDXL.java is compiled by opening a DOS window, changing to the jdk1.1.8\bin directory and typing this command:

```bash
doswindow> javac XMLSource\GetFishDXL.java
```

142 XML Powered by Domino
This creates several class files of the source code in the XMLSource subdirectory. If any compile errors are present they will be listed in the command window. Before the source code can be compiled, it has to be error-free. One of the most common mistakes for Domino developers is case. The case of the file names you type in to compile and run java code have to be exact matches to the actual file names or the class identifiers in the source code when using the javac and java utilities.

Because we compile source code by running the javac command line utility from the jdk1.1.8\bin subdirectory, there is no need to add a path reference on your system that points to this directory. However, when running compiled Java classes, the java command line utility has to have some way to get to it's own Java classes, the Notes Java classes, and the classes we just created for the GetFishDXL application. On Windows machines this is facilitated via the CLASSPATH setting, defined in the autoexec.bat file on win98 machines and via environment variables for other Windows environments. Here are the CLASSPATH settings for the development server:

```bash
SET CLASSPATH=d:\lotus\notes\data\domino\java\NCSO.jar;
d:\lotus\notes\Notes.jar; d:\lotus\notes\DXLTools.jar;
d:\jdk1.1.8\bin\XMLSource
```

The Java command line utility activates the local Java Virtual Machine (JVM) and has automatic access to all it's own Java classes by default. The CLASSPATH tells the JVM to look in Notes.jar, NCSO.jar and DXLTools.jar or the jdk1.1.8\bin\XMLSource paths for other classes. The JVM looks for compiled .class files that it can run. Because Notes and NCSO are .jar files that contain many .class files in them, they must be specified by their specific file name. Our files are .class files, so we need to point the JVM to the directory where the .class files are located.

Now that we have installed the XML Toolkit and the JDK, compiled the source code, and specified the CLASSPATH for our .class files, we can run the GetFishDXL application. GetFishDXL.class is run by opening a DOS window, changing to the jdk1.1.8\bin directory and typing this command:

```bash
java GetFishDXL.java
```

Now that we've created our runtime environment and seen the application in action, let's review the code. Because this is a standalone Java application, the largest proportion of the code is involved in running the GUI and accessing the Domino databases to display data, with the smallest part of the code actually serving up DXL. Here's the code in it's entirety, then we'll go over the components individually:
import java.io.*;
import java.util.*;
import lotus.notes.*;
import java.awt.*;
import java.awt.event.*;
import lotus.dxl.*;

public class GetFishDXL extends NotesThread {

    private static DXLExportOptions options;
    private static DXLExporter exporter;
    private static int noteId = 0;
    private static DXLDatabase DXLdb;
    private static String dbFileName;
    private static String dxlFileName;
    private static OutputStream output;
    private static int noteTypes = 0;
    private static boolean bGotNoteTypes = false;
    private static String noteName;

    public static void main(String[] args) {
        GetFishDXL DXLApp = new GetFishDXL();
        DXLApp.start();
    }

    public void runNotes() {
        try {
            Session s = Session.newInstance();
            Database db = s.getDatabase("", "FishNet.nsf");
            View view = db.getView("Fish Names");
            Gui gui = new Gui("XML Application Example - get Fish DXL");
            gui.updaters.addElement(this);
            Document doc = view.getFirstDocument();
            while (doc != null) {
                Vector v = doc.getColumnValues();
                String fishName = (String)v.elementAt(0);
                gui.list.addItem(fishName);
                doc = view.getNextDocument(doc);
            }
        } catch (NotesException e) {
            e.printStackTrace();
            return;
        }
    }
}
class Gui extends Frame implements ActionListener {
    public List list;
    public String fishName;
    public TextArea FishDXL;
    public Vector updaters;

    Gui(String title) {
        super(title);
        updaters = new Vector();
        layout = new GridLayout(1, 2);
        list = new List();
        list.addActionListener(this);
        add(list);
        FishDXL = new TextArea("", 50, 20);
        add(FishDXL);
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent evt) {
                int n = updaters.size();
                for (int i = 0; i < n; i++) {
                    Thread t = (Thread)updaters.elementAt(i);
                    if (t != null) {
                        try {
                            t.join();
                        } catch (InterruptedException e) {
                            System.out.println("Exception: " +
                                e.getMessage());
                            e.printStackTrace();
                        }
                    }
                }
                dispose();
            }
        });
        setSize(800, 600);
        show();
        toFront();
    }

    public void actionPerformed(ActionEvent evt) {
        if (evt.getSource().equals(list)) {
            fishName = list.getSelectedItem();
            GetDXL updater = new GetDXL(this);
            updaters.addElement(updater);
            updater.start();
        }
    }
}
class GetDXL extends NotesThread {
    Gui gui;

    GetDXL(Gui gui) {
        super();
        this.gui = gui;
    }

    public void runNotes() {
        try {
            output = new FileOutputStream("C:/TEMP/DXLFishRecord.XML");
            exporter = new DXLExporter();
            String fishName = gui.fishName;
            Session session = Session.newInstance();
            Database db = session.getDatabase("", "FishNet.nsf");
            View view = db.getView("Fish Names");
            Document doc = view.getDocumentByKey(fishName);
            if (doc == null) {
                System.out.println("Fish document not found");
                return;
            }
            String valStr = new String(doc.getNoteID());
            try {
                noteId = Integer.parseInt(valStr, 16);
            }
            catch (Exception e) {
                System.out.println("Exception: " + e.getMessage());
            }
            exporter.setExportOptions(options);
            DXLdb = new DXLDatabase("FishNet.nsf");
            exporter.exportDXL(DXLdb, output, noteId);
            BufferedReader in = new BufferedReader(new FileReader("C:/TEMP/DXLFishRecord.XML"));
            gui.FishDXL.setText("File Saved to C:/TEMP/DXLFishRecord.XML\nContents:\n\n");
            String line;
            while( ( line = in.readLine() ) != null)
            {
                gui.FishDXL.append(line+"\n");
            }
            //System.out.print(gui.FishDXL.getText());
            in.close();
        }
    }
}

XML Powered by Domino
catch (Exception e) {
    System.out.println("Exception: " + e.getMessage());
    e.printStackTrace();
}

Now let's have a look at each piece of the code and what it does for the application:

import java.io.*;
import java.util.*;
import lotus.notes.*;
import java.awt.*;
import java.awt.event.*;
import lotus.dxl.*;

public class GetFishDXL extends NotesThread {

    private static DXLEExportOptions options;
    private static DXLEExporter exporter;
    private static int noteId = 0;
    private static DXLDatabase DXLdb;
    private static String dbFileName;
    private static String dxlFileName;
    private static OutputStream output;
    private static int noteTypes = 0;
    private static boolean bGotNoteTypes = false;
    private static String noteName;

    public static void main(String[] args) {
        GetFishDXL DXLApp = new GetFishDXL();
        DXLApp.start();
    }

This first part of the code imports all of the Java classes the application will need to run. Once that is done we define the class and make it a multi-threaded Java application by extending NotesThread, an extension of the java.lang.thread class. Next we define some system variables that we will be using. Because they are private, they can only be passed to any other classes called by this class.

    public static void main(String[] args) {
        GetFishDXL DXLApp = new GetFishDXL();
        DXLApp.start();
    }
public void runNotes() {
    try {
        Session s = Session.newInstance();
        Database db = s.getDatabase("", "FishNet.nsf");
        View view = db.getView("Fish Names");
        Gui gui = new Gui("XML Application Example - get Fish DXL");
        gui.updaters.addElement(this);
        Document doc = view.getFirstDocument();
        while (doc != null) {
            Vector v = doc.getColumnValues();
            String fishName = (String)v.elementAt(0);
            gui.list.addItem(fishName);
            doc = view.getNextDocument(doc);
        }
    } catch (NotesException e) {
        e.printStackTrace();
        return;
    }
}

The public static void main is needed to make this class an application. It tells
the JVM that this code can be run from the command line, as opposed to an
applet, servlet or agent, which must be implemented by associated events.
DXLApp is a new instance of a thread. When a new thread is created via an
extension of NotesThread, as is the case here, the JVM is instructed to look
in the class for the first instance of a subclass called runNotes in the class
file and execute it. runNotes looks up documents in the FishNet database in a
view called FishNames. Next we create the UI using the AWT classes
and open the awt window. The view’s first column is displayed to the window via a
vector.

class Gui extends Frame implements ActionListener {
    public List list;
    public String fishName;
    public TextArea FishDXL;
    public Vector updaters;

    Gui(String title) {
        super(title);
        updaters = new Vector();
        setLayout(new GridLayout(1, 2));
        list = new List();
        list.addActionListener(this);
        add(list);
        FishDXL = new TextArea("", 50, 20);
add(FishDXL);
addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent evt) {
        int n = updaters.size();
        for (int i = 0; i < n; i++) {
            Thread t = (Thread)updaters.elementAt(i);
            if (t != null) {
                try {
                    t.join();
                } catch (InterruptedException e) {
                    System.out.println("Exception: "+ e.getMessage());
                    e.printStackTrace();
                }
            }
        }
        dispose();
    }
});
setSize(800, 600);
show();
toFront();
}
}

public void actionPerformed(ActionEvent evt) {
    if (evt.getSource().equals(list)) {
        fishName = list.getSelectedItem();
        GetDXL updater = new GetDXL(this);
        updaters.addElement(updater);
        updater.start();
    }
}
}

This code handles the creation, input and output of the AWT GUI interface. It is actually called from two different classes. The first class lists the fish records in the left pane of the window. The second call to this class is from code that we will review next. It creates the textArea in the right pane of the window and passes in the DXL code from a generated file.

class GetDXL extends NotesThread {
    Gui gui;
    
    GetDXL(Gui gui) {
        super();
        this.gui = gui;
    }
}
public void runNotes() {
    try {
        output = new FileOutputStream("C:/TEMP/DXLFishRecord.XML");
        exporter = new DXLExporter();
        String fishName = gui.fishName;

        This is where we actually get into the DXL processing for the application. This
        class also extends NotesThread and creates a new thread, which looks for
        the first instance of a runNotes subclass. runNotes in this instance is used to
        export from a Domino document to a DXL document on the file system. Next
        we specify a FileOutPutStream that we will write DXL, which is an extension
        of the java.io.outputstream. Next we define a new DXLExporter, which is a
        base class in the DXLTools.jar file. fishName defines the key that was passed
        from the UI when the user double-clicked on a fish document.

        Session session = Session.newInstance();
        Database db = session.getDatabase("", "FishNet.nsf");
        View view = db.getView("Fish Names");
        Document doc = view.getDocumentByKey(fishName);
        if (doc == null) {
            System.out.println("Fish document not found");
            return;
        }

        String valStr = new String(doc.getNoteID());
        try {
            noteId = Integer.parseInt(valStr, 16);
        } catch (Exception e) {
            System.out.println("Exception: " + e.getMessage());
        }

        This segment of code is devoted to looking up the NoteID of a document,
        based on the key passed via the fishName string Note that we look up key
        values using regular Domino Java sessions, databases, and other objects.
        DXL has its own DXLDatabase class which we will define separately, but the
        XML toolkit is exclusively for importing and exporting DXL, so DXLDatabase
        is focussed on this purpose only. To actually retrieve the NoteID that we want
        to include in the DXLExporter class we need to use the conventional Domino
        Java classes.

        exporter.setExportOptions(options);
        DXLdb = new DXLDatabase("FishNet.nsf");
        exporter.exportDXL(DXLdb, output, noteId);
    }
}

150  XML Powered by Domino
BufferedReader in = new BufferedReader(new FileReader("C:/TEMP/DXLFishRecord.XML"));
gui.FishDXL.setText("File Saved to C:/TEMP/DXLFishRecord.XML
Contents:

");
String line;
while( ( line = in.readLine() ) != null)
    { gui.FishDXL.append(line+"\n"); }
in.close();
}

This is where we actually get into the DXL processing for the application. The first thing to do in a DXL export is to create the DXLExportOptions object by invoking the setExportOptions method of the DXLexporter class. In this case the new DXLExportOptions class is created with no arguments, meaning with default settings for all options. Refer to the latest XML Toolkit documentation for full explanations and an up-to-date list of options and arguments. We then define the DXLdatabase that we will be pulling DXL data out of. The exportDXL method of DXLExporter is then invoked, which writes DXL from the DXLdb object to the stream specified by output. Including the noteID argument instructs exportDXL to only export a single document matching the noteID supplied. If this argument is not included, the default exportDXL setting exports all documents in the database. Once the DXL file has been created and the DXL has been exported, we can read the file back into the UI via a bufferedReader, which is an extension of java.io.reader. It returns a character representation of the file that we can pass to the UI line-by-line using the readLine method.

3.9.4.3 Working with the DXL Output file
Here’s the full listing of the DXL document produced by the exportDXL method:

<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE document SYSTEM 'domino.dtd'>
<!-- DXLExporter version 0.5, schema (DTD) version 0.13 -->
<!DOCTYPE document SYSTEM 'domino.dtd'>
<!-- DXLExporter version 0.5, schema (DTD) version 0.13 -->

<document xmlns="http://www.lotus.com/dxl" version="0.13"
form="FishNameResults">
  <noteinfo noteid="fde" unid="2ABA0B582793E9618525695F00567C7A"
sequence="1">
    <created>
      <datetime dst="true">20000919T114439,62-04</datetime>
    </created>
    <modified>
      <datetime dst="true">20000919T115208,01-04</datetime>
    </modified>
  </noteinfo>
</document>
<item name="FishName">
<text>Shark</text>
</item>

<item name="ButtonPressed">
<text>FN</text>
</item>

<item name="ButtonPressedValue">
<text>Shark</text>
</item>

<item name="$UpdatedBy" names="true">
<textlist>
<text>Anonymous</text>
</textlist>
</item>

<item name="FishID" summary="false">
<textlist>
<text>640</text>
<text>641</text>
<text>642</text>
<text>643</text>
<text>644</text>
<text>645</text>
<text>646</text>
<text>647</text>
<text>648</text>
<text>649</text>
<text>650</text>
<text>651</text>
<text>652</text>
<text>653</text>
</textlist>
</item>

<item name="Subcategory" summary="false">
<textlist>
<text>Blacktip</text>
<text>Bronze Whaler</text>
<text>Eastern School</text>
<text>Grey Nurse</text>
</textlist>
</item>
This is a representation of a single Domino document produced by the exportDXL method of the DXLExporter class. It represents an XML document that conforms to the domino.DTD. It is identified as a DXL document via the DXL namespace, http://www.lotus.com/dxl. As with the case of the generateXML method of the Domino document Java class, DXL does a great job of generating an XML document that describes a Domino document and all of its elements, but does not do a very good job of describing the nature of the content that the document contains. This is where Extensible Stylesheet
Language (XSL) is very useful, by transforming one XML format to another via Extensible Stylesheet Transformation (XSLT). In the next chapter we’ll discuss XSL and XSLT in detail and show you how to transform DXL to other XML formats.
Chapter 4. Transforming XML data in Domino

We introduced the basics of using an XSLT transformation engine to transform XML data into alternate results in Chapter 2. XSLT allows you to take XML data and transform it into alternate XML, HTML, WML, delimited text, or any other output format you might need.

In this chapter we discuss the use of the LotusXSL processing engine. We demonstrate how to transform XML data stored in Domino, as well as XML data that passes through the Domino, into an alternate result set.

4.1 XSLT primer

In this section we review some of the concepts we discussed in Chapter 2 and talk about the LotusXSL engine specifically.

4.1.1 LotusXSL and Xalan

LotusXSL is an XSLT transformation engine written by engineers at Lotus Development Corporation. In November of 1999 Lotus donated LotusXSL to the Apache Software Foundation under the title Xalan. The ongoing development of Xalan has shifted to the Apache organization. Updates and enhancements to the Xalan processor can be monitored at:

http://xml.apache.org/

Xalan can be used as a stand-alone XSLT transformation engine, or incorporated into Java applications.

LotusXSL is the transformation engine that is included in the Domino server and Domino Designer client starting with release 5.03. The transformation engine is implemented via a single Java jar file called lotusxsl.jar, which can be found in the installation directory of you server or designer client. This jar file is also the basis for the methods included in the predefined Java classes that are referenced in the back end Java classes for Item, RichTextItem, EmbeddedObject and MIMEEntity. This file contains the wrapper classes that allow access to the transformation engine interfaces.

4.1.1.1 The transformation engine

The process of transformation is always the same. There are three components that are required to perform a transformation:

- Input: a raw XML data source
- Stylesheet: an XSLT stylesheet that defines the desired output
• An XSLT processing engine: in our case, LotusXSL

The result will be the output defined by the XSLT stylesheet. It can be in the form of a file written to disk or as data presented to a requesting system.

4.1.2 XSLT syntax

XSLT syntax is well-formed XML. This means that every stylesheet that is written is itself a well formed XML document. XSLT stylesheets are written using a command syntax that is associated with a particular URI or namespace. This is how the XSLT processor identifies the difference between the XSLT commands and the values to be passed to the result set.

4.1.2.1 Namespaces

As we discussed in Chapter 1, there is a specific namespace that must be referenced when passing XSLT stylesheets to the processing engine. To understand the current namespace a little bit of history is in order.

XSLT has been a recommendation from the W3C since November 16th, 1999. Since it has become a recommendation, the official XSLT namespace is:

```xml
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

Prior to XSLT becoming a recommendation, the working draft namespace used was:

```xml
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
```

Some older XSLT processing engines still require the older namespace declaration, most notably IE5.0 (as IE5.5 can have a compliant processor installed). This only becomes a consideration when you will be sending the XML with a stylesheet reference directly to the browser client for processing.

If the processing all takes place on the Domino server using the LotusXSL processor, the recommended namespace should be used.

In our examples we will be using the LotusXSL processor.

Basic XSL instructions can be identified by the inclusion of the "xsl:" namespace in the tag name. For example, one of the most common XSLT instructions is the `<xsl:template>` instruction. This tag identifies the beginning of a new template definition, so the XSLT processing engine will process this instruction and apply the template instructions contained within it.
The complete recommendation for XSLT, including all of the appropriate instructions, can be found on the W3C Web site at:

www.w3.org/TR/xslt

4.1.3 XSLT stylesheets

In this section we start with some simple examples of transformation and then gradually progress to more complex transformations.

To facilitate learning and applying stylesheets in Domino, we first build a form design that allows us to write and apply stylesheets to XML data.

1. In the Domino Designer, create a form design that looks similar to the one shown in Figure 56.

![Figure 56. Stylesheet primer form design](image)

We have named the form “stylesheet Primer” with an alias of “SSPrimer.” The field definitions are defined in the following Table 12.
Table 12. Fields in form design

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Field Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>XSLStyleSheet</td>
<td>Editable Text</td>
</tr>
<tr>
<td>RawXML</td>
<td>Editable Text</td>
</tr>
<tr>
<td>TransResult</td>
<td>Editable Text</td>
</tr>
</tbody>
</table>

2. We will use the following Java agent to apply the stylesheet and view the results. The agent parameters are in Table 13.

Table 13. TransformPrimerAgent Parameters

<table>
<thead>
<tr>
<th>Agent Parameter</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>XSLPrimerTransToText</td>
</tr>
<tr>
<td>When should this agent run?</td>
<td>Manually from Agent List</td>
</tr>
<tr>
<td>Which document(s) should it act on?</td>
<td>Run Once(@ commands may be used)</td>
</tr>
<tr>
<td>Run</td>
<td>Java</td>
</tr>
</tbody>
</table>

Following is the Java source code for this agent:

```java
import lotus.domino.*;
import java.io.*;
import org.xml.sax.*;

public class TransformPrimer extends AgentBase {
    public void NotesMain() {
        try {
            Session s = getSession();
            AgentContext ac = s.getAgentContext();
            Database db = ac.getCurrentDatabase();
            Document doc = ac.getDocumentContext();

            // assign object variables to the fields that have the XML and XSLT code in them
            Item xmlItem = doc.getFirstItem("RawXML");
            Item xslItem = doc.getFirstItem("XSLStyleSheet");

            // create an inputsource out of the value from the XSLT field
            String sc = doc.getItemValueString("XSLStyleSheet");
            InputSource style = xslItem.getInputSource();
            // create a File Writer, transform the xml and write it to the following file.
            FileWriter fw = new FileWriter("C:\temp\output.txt");
            XSLTResultTarget result = new XSLTResultTarget(fw);
            xmlItem.transformXML(style, result);
        }
    }
}
```
// create a StringWriter and write the same results to the results field in the current document
style = new InputSource(new StringReader(sc));
StringWriter rw = new StringWriter();
result = new XSLTResultTarget(rw);
xmlItem.transformXML(style, result);
doc.replaceItemValue("TransResult", rw.toString());
doc.save(true, true);
}

} catch(Exception e) {
    e.printStackTrace();
}

3. Save this agent. This agent transforms the XML using the stylesheet code to transform the XML to an alternate output. This output is written to a file, C:\temp\output.txt, and also written to a field in the original document.

4. Create an action in the Stylesheet Primer form called “Transform XML to Text” that will call this previously defined agent using the following formula:

@Command([ToolsRunMacro];"(XSLPrimerTransToText)");
@Command([FileCloseWindow]);
@Prompt([OK];"Transformation Complete";"The Transformation is complete. Review text file or Document contents")

Now we have a form design that will hold the XML and the stylesheet together in one place.

Lets start by transforming a simple XML document with a simple stylesheet.

5. Create a new Stylesheet Primer document.

6. Either input or cut/paste this text into the XML code field

<?xml version="1.0" ?>
<Directory>
<Person>
    <FirstName>Paul</FirstName>
    <LastName>Calhoun</LastName>
    <StreetAddress>123 Main</StreetAddress>
    <City>Boston</City>
    <State>MA</State>
    <Zipcode>04938</Zipcode>
</Person>
<Person>
    <FirstName>Brian</FirstName>
    <LastName>Benz</LastName>
</Person>
We will use this XML code as the basis for writing our stylesheets.

7. Input or cut/paste the following XSLT code into the stylesheet field:

```xml
<?xml version="1.0" ?><xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

</xsl:stylesheet>
```

Before we perform the transformation, let's review the syntax so far. As mentioned at the beginning of this example, XSLT stylesheets are well-formed XML documents. That is why the first line will be the XML declaration statement:

```
<?xml version="1.0" ?>
```

Next we have the root tag. The root tag of an XSLT stylesheet will always be:

```
<xsl:stylesheet>
</xsl:stylesheet>
```

Just like data in an XML document, all of the XSLT processing commands will go between the root tag.

Attributes of the root tag declare the appropriate version number and namespace:

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

As of the current recommendation, the version attribute is 1.0 and must be enclosed in quotes. We already discussed the namespace at the beginning of this example. Make sure that you type it in exactly as it
appears, including the case of all the words. Most XSLT processing engines will not process properly if this namespace is not correct.

8. Now we will process the XML code and review the output. From the document you created, choose **File -> Save**.

9. Click the “Transform XML using stylesheet” action in the action bar. Verify that there were no processing errors by looking at the Java Console. From the Notes Client choose **File -> Tools -> Show Java Debug Console**. If there were any processing errors, or the XML data or the XSLT stylesheet were not well-formed, the errors will show up here. If there are errors to the syntax, correct them and click the action again. If there are no errors in the Java Console, the agent completed successfully. The primer document with the code examples will look like the one shown in Figure 57 on page 162.
Stylesheet Primer from the XML: Powered by Domino Redbook.

Input XSLT stylesheet syntax in the field below:

```xml
<?xml version="1.0"?>
<xlst:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
</xlst:stylesheet>
```

Input your raw XML data in the field below:

```xml
<?xml version="1.0"?>
<Directory>
<Person>
<FirstName>Pete</FirstName>
<LastName>Collins</LastName>
<StreetAddress>123 Main</StreetAddress>
<City>Boston</City>
<State>MA</State>
<PostalCode Country="US">02116</PostalCode>
</Person>
<Person>
<FirstName>Brian</FirstName>
<LastName>Brown</LastName>
<StreetAddress>937 Acme Ave.</StreetAddress>
<City>New York</City>
<State>NY</State>
<PostalCode Country="US">10040</PostalCode>
</Person>
<Person>
<FirstName>Yuki</FirstName>
<LastName>Murakami</LastName>
<StreetAddress>222 Marlborough Street</StreetAddress>
<City>Boston</City>
<State>MA</State>
<PostalCode Country="US">02120</PostalCode>
</Person>
</Directory>
```

Transformation output:

Figure 57. Transform primer document
The agent will apply the XSLT stylesheet to the XML data, save the document, and close the window. You should see a prompt box that looks like the one in Figure 58.

![Figure 58. Transform prompt dialog box](image)

Click OK. This will close the document and return to the document view. The XML has been saved in the C:\temp\output.txt file, as well as in the TransResult field of the document. To review the output, open the file in the temp directory in Microsoft Notepad (or any text editor).

10. Click **Start**, then the **Run Menu** option. Type in the command shown in Figure 59.

![Figure 59. Notepad run command](image)

The output will look like Figure 60 on page 164.
The corresponding result text in the primer document will look like Figure 61.

Transformation output:

```
<?xml version="1.0" encoding="UTF-8"?>
Paul
Calhoun
123 Main
Boston
MA
04938

Brian
Benz
987 Acme Ave.
New York
NY
03048

Yuhsuke
Morakami
222 Marlborough Street
Boston
MA
04920

<?xml version="1.0" encoding="UTF-8"?>
Paul Calhoun 123 Main Boston MA 04938
Brian Benz 987 Acme Ave New York NY 03048
Yuhsuke Morakami 222 Marlborough Street Boston MA 04920
```
Notice that in the results stored in the primer document there are several dark vertical lines. These lines represent carriage returns and line feeds that do not translate in Notes, but display properly in a text editor.

It is rather subtle, so look closely. Notice that all of the data from the document was returned with the exception of the country attribute from the PostalCode element.

So what took place? Why would applying a blank stylesheet to an XML data source return the values from every element in the document? This happens because of default templates that the XSLT processing engine uses to return values from the XML elements. If the stylesheet does not contain any templates specify which nodes to process, the default templates are applied to produce the output. The default templates are a part of most XSLT processing engines that conform to the W3’s 1.0 recommendation.

### 4.1.4 XSLT template model

The default templates are fine if we want to return every value of every node in the document. But if we want to do more sophisticated transformations and transform into alternate specific formats, we will need to create some specific templates to return the values we want in the order we want.

At the heart of most stylesheets will be the template. The standard template in a stylesheet is formatted in the following way:

```xml
<xsl:template [optional attributes] >
    <xsl:someTemplateCommand />
</xsl:template>
```

Notice that the template is well-formed XML. Remember, the entire stylesheet must be a well-formed XML document. So, if we wanted to create a stylesheet that references the root node of the document, the complete stylesheet syntax would be:

```xml
<?xml version="1.0" ?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

    <xsl:template match="/">
        <xsl:apply-templates />
    </xsl:template>

</xsl:stylesheet>
```

The forward slash character "/" is the xpath abbreviation for the “root” of the document tree. The first template in the stylesheet will usually reference the
4.1.5 Returning node values from XML document

When an XML document is processed by the XSLT transformation engine, it is parsed into a DOM tree.

The parent/child hierarchical representation would be structured as shown in Figure 62.

After the directory XML data is parsed, it would produce the DOM tree shown in Figure 63.
The XSLT stylesheet templates are applied to the DOM tree based upon matches that occur. A template is defined to match a node in the node tree representation of the XML data. So if we only wanted to return the first name values from the XML document, we could tell the XSLT processor to find all nodes that match a certain pattern. Using a parent/child hierarchical structure of the DOM tree, the path to the FirstName nodes would be Directory/Person/FirstName.

The statements that follow the select and match attributes in an XSLT stylesheet are called XPath expressions. We discussed the XPath recommendation from the W3 org in Chapter 2. An XPath expression is used to traverse a parsed XML document represented as a DOM tree. Using XPath expressions we can return individual node values, node sets, as well as perform string and numerical operations. The complete XPath specification can be found on the W3C organizations Web site at:

http://www.w3.org/TR/xpath
The `<xsl:apply-templates>` command is used to tell the XSLT processor to take the child node and search the stylesheet and process all templates. If a template match is found, then process the commands within that template. Using the `select` attribute of the apply-templates command tells the XSLT processor to take just the selected nodes in the stylesheet for matching template. If found, then process those commands. When the following stylesheet is applied to the Directory XML data, only the values in the FirstName nodes are returned.

```xml
<?xml version="1.0" ?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <xsl:apply-templates select="Directory/Person/FirstName" />
  </xsl:template>

  <xsl:template match="FirstName">
    <xsl:value-of select="." />
  </xsl:template>
</xsl:stylesheet>
```

Inside the XSLT processing engine the stylesheet is applied to the XML data with the following logic.

This match pattern in this template, "/", is referring to the root node of the DOM tree. Do not confuse the root node of the DOM tree with the root element of the XML document. In the DOM tree representation of any XML data the root node always refers to the top of the whole tree. It has only one child element. That child element is the root element of the XML document. In our example the Directory node would be the child of the Document node represented by "/". The apply-templates command tells the processing engine to apply all of the templates in the stylesheet against the DOM representation of the XML data. Because the document root is currently the context node (or the node that has focus) the select statement starts at the current node and then traverses the DOM tree to find all the Directory nodes that are children of the current node and that have Person child nodes that have FirstName Child nodes.

```xml
<?xml version="1.0" ?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <xsl:apply-templates select="Directory/Person/FirstName" />
  </xsl:template>
</xsl:stylesheet>
```

After the apply-templates command is executed, it will search for a template that matches the select statement. In this case FirstName:
The template with the match statement equal to FirstName is found by the transformation engine and the command enclosed is executed. The command `<xsl:value-of select="." />` tells the processing engine to return the string value of the current node or self as represented by a period in quotes, ".". The processing engine returns all three FirstNames with no formatting. The output document is shown in Figure 64.

![Figure 64. Output from FirstName stylesheet](image)

The previous example showed the way to access series nodes by referencing the node values directly. As with most scripting languages, there is usually more than one way to get the same results. The following stylesheet syntax also returns just the values from the FirstName nodes:

```xml
<?xml version="1.0" ?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="FirstName">
    <xsl:value-of select="." />
  </xsl:template>
  <xsl:template match="/">
    <xsl:apply-templates select="Directory" />
  </xsl:template>
  <xsl:template match="Directory">
    <xsl:apply-templates select="Person" />
  </xsl:template>
  <xsl:template match="Person">
    <xsl:apply-templates select="FirstName" />
  </xsl:template>
</xsl:stylesheet>
```
We have added some additional templates and ordered them in what might seem to be an illogical way. Trying to read the stylesheet from top to bottom is confusing. We have done this to prove a point. That point is that the XSLT engine does not read the document from top to bottom, processing the templates in the order they appear. The templates are applied based on pattern matching and the apply-templates command. The previous code has been annotated to help understand what is happening inside of the transformation engine.

```
<xsl:template match="FirstName"> 4.
  <xsl:value-of select="." />
</xsl:template>

<xsl:template match="/"> 1.
  <xsl:apply-templates select="Directory" />
</xsl:template>

<xsl:template match="Directory"> 2.
  <xsl:apply-templates select="Person" />
</xsl:template>

<xsl:template match="Person"> 3.
  <xsl:apply-templates select="FirstName" />
</xsl:template>
```

The templates are applied in the numbered order. When the stylesheet is applied to the XML data, an implied apply templates instruction is issued to find a template that matches the root node. The XSLT processing engine reads ALL of the templates in the stylesheet. There is only one that matches the Document root, line 1 in the previous code, so the context node (or current pointer) is set to the Document root and the instructions in the template are carried out. In this case the command is to `<xsl:apply-templates select="Directory" />`. The XSLT processing engine then gets the one Directory node and looks for a template that matches that node. It finds the templates at line 2. in the code listing. Once the match is made this changes the context node to the node in the match pattern. This is important because all statements are relative to the current context node. That is why the document root could find the Directory template and the apply-templates command, `<xsl:apply-templates select="Person" />` selects the three person nodes. The command `<xsl:apply-templates select="FirstName" />` is executed from the Person template at code listing, which causes the XSLT processing engine to again look at every template in the stylesheet. That is why the FirstName template is found and applied, even though it appears before the other three templates that have been processed. For readability
you would want your templates to follow a logical order, but as we have demonstrated the XSLT processor does not care.

Using this approach you can write a template that will return the values from any of the nodes you want processed.

### 4.1.6 Including additional text

One of the most widespread misconceptions concerning XSLT is that it is for transforming XML data into HTML. It is true that using XSLT an XML document can be transformed into HTML, but it can also be transformed into plain text or any other form of XML. There is a command that you can include that will tell the LotusXSL transformation engine the output that you desire. The command is described in Table 14.

<table>
<thead>
<tr>
<th>Command</th>
<th>Attribute</th>
<th>Attribute Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;xsl:output /&gt;</code></td>
<td>method</td>
<td>text</td>
</tr>
<tr>
<td></td>
<td></td>
<td>xml</td>
</tr>
<tr>
<td></td>
<td></td>
<td>html</td>
</tr>
</tbody>
</table>

The following code uses this command to inform the processing engine that the output should be text:

```xml
<?xml version="1.0" ?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:output method="text" />
    <xsl:template match="/">
        <xsl:apply-templates select="Directory/Person" />
    </xsl:template>
    <xsl:template match="Person">
        <xsl:value-of select="LastName" />
        <xsl:value-of select="FirstName" />
        <xsl:value-of select="City" />
        <xsl:value-of select="State" />
        <xsl:value-of select="PostalCode" />
    </xsl:template>
</xsl:stylesheet>
```
Notice also that in the template where match is equal to person, we are able to select each child node by name. This is because the “Person” node is the current context node and each of the child nodes can be addressed relative to its parent, in this case “Person”.

The result of this code displayed in Notepad is shown in Figure 65.

![Figure 65. Text output in Microsoft Notepad](image)

If the output command is omitted from the stylesheet the processing engines will default to an output of XML. That is why in our previous output the XML declaration statement was always included.

![Figure 66. XML declaration included in output](image)

Notice in the output shown in Figure 66 that the text is not formatted in any way. The processing engine does not include any formatting, carriage returns or line feeds in the data. To include output in the data, the values must be included in the stylesheet where you want them to appear. There are two ways to include additional text in your output:

- Include the plain text where it should appear
- Use the `<xsl:text>` command

The following code demonstrates examples of using both of these techniques.

```xml
<?xml version="1.0" ?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="text" />

</xsl:stylesheet>
```
<xsl:template match="/">
  <xsl:apply-templates select="Directory/Person" />
</xsl:template>

<xsl:template match="Person">
  <xsl:value-of select="LastName" />'xsl:text>, </xsl:text>
  <xsl:value-of select="FirstName" />
  Address:
  <xsl:value-of select="City" />'xsl:text>, </xsl:text>
  <xsl:value-of select="State" />'xsl:text> </xsl:text>
  <xsl:value-of select="PostalCode" />
  <xsl:text>&#010;&#013;</xsl:text>
</xsl:template>
</xsl:stylesheet>

Microsoft Notepad will not display the carriage returns/line feeds properly, so to see the formatted text, view the output.txt file in Microsoft Wordpad or a word processor. The output.txt file looks like Figure 67.

![Figure 67. Text output in Microsoft Wordpad](image)

Using the <xsl:text> command, punctuation can be inserted anywhere in the text output that it needs to appear. Also notice that actual text may be inserted directly into the stylesheet where it should appear in the output. When the processing engine encounters an entry in the stylesheet that is not
an XSLT command denoted by the xsl: namespace, that entry is written to the result output.

4.1.7 Sorting data with xsl:sort

With XSLT stylesheets, the nodes may be output in any order that fits the requirement for outputting this data. The values do not have to be written out to the result set in the same order as they appear in the data. Nodes that do not need to be included in this output can simply be omitted, as we did with the StreetAddress node. When a template is applied, the nodes that will be processed may also be ordered using the <xsl:sort /> command. The following code allows for the names to be sorted in the output:

```xml
<?xml version="1.0" ?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:output method="text" />
    <xsl:template match="/">
        <xsl:apply-templates select="Directory/Person">
            <xsl:sort />
        </xsl:apply-templates>
    </xsl:template>

    <xsl:template match="Person">
        <xsl:value-of select="LastName" /><xsl:text>, </xsl:text>
        <xsl:value-of select="FirstName" />
        Address:
        <xsl:value-of select="City" /><xsl:text>, </xsl:text>
        <xsl:value-of select="State" /></xsl:text>
        <xsl:value-of select="PostalCode" />
        <xsl:text>&#010;&#013;</xsl:text>
    </xsl:template>

</xsl:stylesheet>
```

When viewed in Microsoft Wordpad or a word processor, the output would look like that shown in Figure 68.
4.1.8 How to access attributes

Attribute values are not processed as child nodes from the perspective of the DOM Tree. However, the values of attributes can be accessed to be included in the desired output. The following code will return the “Country” attribute from the PostalCode Element and include it in the output. Attributes can be accessed by using the xpath abbreviation “@” followed by the name of the attribute node to be returned.

```xml
<?xml version="1.0" ?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text" />
  <xsl:template match="/">
    <xsl:apply-templates select="Directory/Person">
      <xsl:sort />
    </xsl:apply-templates>
  </xsl:template>
  
  <xsl:template match="Person">
    <xsl:value-of select="LastName" /><xsl:text>, </xsl:text>
    <xsl:value-of select="FirstName" />
    Address:
    <xsl:value-of select="City" /><xsl:text>, </xsl:text>
    <xsl:value-of select="State" />
  </xsl:template>
</xsl:stylesheet>
```

Figure 68. Sorted Text output
This address is located in the "<xsl:value-of select="PostalCode/@Country" />
"<xsl:text>&#010;&#013;</xsl:text>
</xsl:template>
</xsl:stylesheet>

The output from this stylesheet would look like Figure 69.

![Figure 69. Attribute value included in output](image)

### 4.2 Transforming XML to HTML

With XML becoming more and more common as a universal data source, transforming XML to HTML is one of the most common tasks that will be performed. Having the option of storing data in an XML format is a great opportunity. But there will still be a need to present that data to a Web browser, and that means HTML. One of the benefits of having data stored in an XML format is that it allows for the freedom of having multiple stylesheets that can be applied to the data. This means that there can be a single data source and multiple ways of presenting that data.

#### 4.2.1 Including simple HTML markup

Using the same example of directory data, we will create a stylesheet that transforms the XML data into a simple HTML document.
1. We need to enhance the agent we have been using to create the output.txt file so that it will create an output.html file. In the Domino designer, create a copy of the “XSLPrimerTransToText” agent. Edit the agent and change the following code.

   FileWriter fw = new FileWriter("C:\temp\output.txt");

And change it to:

   FileWriter fw = new FileWriter("C:\temp\output.html");

2. Save the agent as “XSLPrimerTransToHtml”

3. Open the SSPrimer form design and create an action that will call this new agent. The easiest way to do this is copy the existing “Transform to Text” agent and paste it back into the action list. Rename it as “Transform to HTML” and update the action code to call the agent with

   @Command([ToolsRunMacro];"(XSLPrimerTransToHtml)");

Now we are ready to transform the XML into HTML.

4. The basic structure of the stylesheet will remain the same except for the output method. The output method needs to be changed to “html”. The basic stylesheet will look like this:

   <?xml version="1.0" ?>
   <xsl:stylesheet version="1.0"
       xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
       <xsl:output method="html" />

   </xsl:stylesheet>

The process of building a stylesheet is similar regardless of the whether the output method is text, HTML, or XML. The general steps are as follows:

- Declare the output method.
- Build the templates to include the node and attribute values to be written to the result tree.
- Include any additional text or markup to be written to the result tree.

Remember from the last example that any value in the stylesheet that is not an XSLT command will be written to the result tree. So to include HTML markup in the result tree we just include the appropriate HTML tags directly in the stylesheet. One very important thing to keep in mind when writing a stylesheet to transform XML to HTML is that the stylesheet itself has got to be a well-formed XML document. This means that when you include HTML markup, the rules of well-formedness have to be adhered to. Every opening tag must have a close tag, and all attributes must be enclosed in quotes.

There are several HTML tags that are not normally closed. For example, the
<hr>, <br>, and <p> tags, just to name a few, are often written into HTML documents without being terminated. In the XSLT stylesheet these tags have to be closed or the stylesheet will not be well-formed. So to include these tags in an XSLT stylesheet the tags will have to be formatted in one of the following ways:

<hr></hr> or <hr />

Now that we have covered the rules, let's look at a stylesheet that outputs some simple HTML. Following is the syntax of an XSLT stylesheet that will transform our directory document to HTML.

```xml
<?xml version="1.0" ?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="html" />  
  <xsl:template match="/" >
    <html>
      <head>
        <title>Simple HTML Transformation</title>
      </head>
      <body>
        <xsl:apply-templates select="Directory" />
      </body>
    </html>
  </xsl:template>
  <xsl:template match="Directory" >
    <xsl:apply-templates select="Person" />
  </xsl:template>
  <xsl:template match="Person">  
    <p>
      <xsl:value-of select="LastName" />
      <xsl:value-of select="FirstName" />
      <xsl:value-of select="StreetAddress" />
      <xsl:value-of select="City" />
      <xsl:value-of select="State" />
      <xsl:value-of select="PostalCode" />
    </p>
  </xsl:template>
</xsl:stylesheet>
```

And this is the HTML code that it produces:
<html>
<head>
<title>Simple HTML Transformation</title>
</head>
<body>
<p>CalhounPaul123 Main Boston MA 04938</p>
<p>BenzBrian987 Acme Ave. New York NY 03040</p>
<p>MurakamiYuhhsuke222 Marlborough Street Boston MA 04920</p>
</body>
</html>

The HTML viewed in a Web browser is shown in Figure 70.

We know from the text example that the stylesheet is not “read” from top to bottom. To illustrate how the HTML was produced, the stylesheet syntax has been repeated with numbers next to the lines to indicate the order of when each line was processed. Following the numbered order it is easy to see how the HTML was written. The bold numbers indicate writing to the result tree.

1. <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform" />

2. <xsl:output method="html" />

3. <xsl:template match="/" >

4. <xsl:template match="/" >

5. <html>

6. <head>

7. <title>Simple HTML Transformation</title>
Notice that the order of processing jumps in the first template to the second template at line 10 `<xsl:apply-templates>` command. Every time an apply-templates command is executed, it will search the stylesheet for a template that matches the select statement and continue processing there. Lines 14 to 21 are written to the result tree for every occurrence of a person node. In other words, the content of this template is processed like it was a for loop that was equal to the number of Person nodes that are found.

### 4.2.2 Including more complex HTML markup

Now that we have covered the basics, we can use these techniques to include more complex HTML code that really formats the output nicely. The following syntax includes HTML tags to put each element on its own line with punctuation:

```xml
<xsl:template match="Person">
  <p>
    <xsl:value-of select="LastName"/>
    ,
    <xsl:value-of select="FirstName"/>
  </p>
  <br/>
  <xsl:value-of select="StreetAddress"/>
  <br/>
  <xsl:value-of select="City"/>
  ,
  <xsl:value-of select="State"/>
  <xsl:value-of select="PostalCode"/>
</xsl:template>
```

180 XML Powered by Domino
This syntax produces the following output:

Calhoun, Paul
123 Main
Boston, MA 04938

Benz, Brian
987 Acme Ave.
New York, NY 03040

Murakami, Yuhsuke
222 Marlborough Street
Boston, MA 04920

It is starting to take shape when viewed in the Web browser, as illustrated in Figure 71.

![Figure 71. HTML output in Web browser](image)

But this output is not much fancier than the standard text list. We need to add some additional items to make it look like an actual Web page.

Following is an XSLT stylesheet that places the XML data in a table with label headings.

```xml
<?xml version="1.0" ?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```
This is directory information produced from XML data.

Address Information

<table>
<thead>
<tr>
<th>Name</th>
<th>Street</th>
<th>City, State, Postalcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>(xml content)</td>
<td>(xml content)</td>
<td>(xml content)</td>
</tr>
</tbody>
</table>

182  XML Powered by Domino
The previous code produces the following HTML output:

```html
<html>
<head>
<title>Simple HTML Transformation</title>
</head>
<body>
<center>
<h1>
This is directory information produced from XML data.
</h1>
</center>
<table width="100%" border="3">
<tr align="center">
<th colspan="3">
Address Information
</th>
</tr>
<tr>
<td>Name</td><td>Street</td><td>City, State Postalcode</td>
</tr>
<tr>
<td>Calhoun, Paul</td><td>123 Main</td><td>Boston, MA 04938</td>
</tr>
<tr>
<td>Benz, Brian</td><td>987 Acme Ave.</td><td>New York, NY 03040</td>
</tr>
<tr>
<td>Murakami, Yuhsuke</td><td>222 Marlborough Street</td><td>Boston, MA 04920</td>
</tr>
</table>
</body>
</html>
```

Figure 72 on page 184 shows how it looks when displayed in the browser.
4.2.3 Working with numeric data and xsl:number

One of the current drawbacks to working with XML data is that every node value is treated as text. This becomes an issue especially when you want to order information as if it were a number. In our directory example we have added an element to contain an employee ID. The XML data looks like this:

```xml
<?xml version="1.0" ?>
<Directory>
  <Person>
    <FirstName>Paul</FirstName>
    <LastName>Calhoun</LastName>
    <StreetAddress>123 Main</StreetAddress>
    <City>Boston</City>
    <State>MA</State>
    <PostalCode Country="US">04938</PostalCode>
  </Person>
</Directory>
```
<EmpId>20</EmpId> 
</Person> 
<Person> 
<FirstName>Brian</FirstName> 
<LastName>Benz</LastName> 
<StreetAddress>987 Acme Ave.</StreetAddress> 
<City>New York</City> 
<State>NY</State> 
<PostalCode Country="US">03040</PostalCode> 
</Person> 
<Person> 
<FirstName>Yuhsuke</FirstName> 
<LastName>Murakami</LastName> 
<StreetAddress>222 Marlborough Street</StreetAddress> 
<City>Boston</City> 
<State>MA</State> 
<PostalCode Country="US">04920</PostalCode> 
</Person> 
</Directory> 

And of course we will be able to sort on this value in our stylesheet using the following syntax:

.....
<xsl:template match="Directory"> 
<xsl:apply-templates select="Person"> 
<xsl:sort select="EmpId" /> 
</xsl:apply-templates> 
</xsl:template> 
.....

This will produce the output shown in Figure 73 on page 186.
Which of course is not what we want. This is because the data is being sorted as text, and as text the ones will always appear before the twos, and so forth. There is an attribute we can add to our sort command that will tell it to treat the data as a number. It is the same as applying the number function to the output. The attribute is the data-type attribute, which allows for text or numerical data sorting. The addition to the sort command would look like this:

```xml
<xsl:sort select="EmpId" data-type="number" />
```

It produces the output shown in Figure 74 on page 187.
4.2.4 Procedural XSLT commands

XSLT has some procedural commands built into it that allow us to conditionally process data or process information in a loop.

The XSLT commands defined in Table 15 allow you to write stylesheets that include conditional selection and processing.

<table>
<thead>
<tr>
<th>XSLT Command</th>
<th>Attributes</th>
<th>Attribute Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;for-each&gt;</td>
<td>select</td>
<td>Node value to select</td>
</tr>
<tr>
<td>&lt;if&gt;</td>
<td>test</td>
<td>Expression returning true or false</td>
</tr>
<tr>
<td>&lt;choose&gt;</td>
<td>none</td>
<td>n/a</td>
</tr>
<tr>
<td>&lt;when&gt; (child of choose)</td>
<td>test</td>
<td>Expression returning true or false</td>
</tr>
<tr>
<td>&lt;otherwise&gt; (child of choose)</td>
<td>none</td>
<td>n/a</td>
</tr>
</tbody>
</table>

4.2.5 Storing XSLT stylesheets in Domino

The previous sections provided a foundation on the syntax of XSLT stylesheets that can now be used within Domino. Before accessing and
applying stylesheets we need a place to call them from. On the Domino server there are two places you can store your stylesheets:

- In the directory structure
- In a page design

If there is only one Domino server in the environment, and it will not be replicating design elements to or from other Domino servers, saving your stylesheets in the directory structure is an option. The files need to be saved into the following path:

```
<drive:><Domino install directory><Data\domino\html
```

The stylesheets can be written in any program editor or text editor and saved with the .xsl extension. If you use a word processing application to create the stylesheets, make sure that they are saved as plain text.

One of the disadvantages of storing the stylesheets in the directory structure is that they are now only available on that one server. This is fine if it is the only server in the environment. On the other hand, storing the stylesheets in the file directory will improve performance if the stylesheet is referenced often. The files that are stored in the file system will be cached by the server on system startup and be available to the calling applications. But in organizations that have multiple Domino application servers there is a better place to store the stylesheet documents.

That better place is Domino Pages. Domino pages were created for the purpose of storing static HTML documents in a Web server environment. XSLT stylesheets are static in nature. They generally have a one-to-many relationship with the XML documents that they transform. Storing the stylesheets in page designs allows them to be replicated between systems sharing the XML-based application.

To create a page that will store the XSLT stylesheet complete the following steps:

1. Create a new page.
2. Either type or paste the XSLT text onto the page.
3. From the page properties dialog box, check “Treat page contents as HTML.”
4. Save the page design with an .xsl extension.

Step 3 requires some further explanation. When this option is enabled it will ensure that the Domino server does not attempt to render the contents of this page through its HTTP rendering engine. What this option really does for us
is treat the page contents as plain text. Plain text is what we need in order to pass this stylesheet to the XSLT processing engine without any errors.

In Figure 75 we have created an XSLT stylesheet page design from the code in the previous example.

Figure 75. Page design in Domino storing an XSLT stylesheet
Now the we have the stylesheet stored in a page design how do we access it? The next lesson will demonstrate how to access the stylesheets stored in page design objects.

### 4.2.6 Applying XSLT stylesheets in Domino

Domino page designs are usually referenced by their URL syntax. So if the page that was created in the previous lesson was saved with the name HTMLTable.xsl on a server named “WickedXML” in a database named “fishnet.nsf” in a directory called “fishnet”, the URL would be:

http://wickedxml/fishnet/fishnet.nsf/HTMLTable.xsl

To reference the stylesheet from the Java agent we created, we simply need to alter where the inputsource reads the text of the stylesheet from.

1. In the Domino designer, create a copy of the XSLPrimerTransToHtml agent and save it as XSLPrimerTransToHtmlUsingPage.
2. Alter the agent code to remove the existing references to reading the stylesheet code from the field in the document and replace it with the bold lines from the following code.

```java
import lotus.domino.*;
import java.io.*;
import org.xml.sax.*;

public class TransformPrimer extends AgentBase {
    public void NotesMain() {
        try {
            Session s = getSession();
            AgentContext ac = s.getAgentContext();
            Database db = ac.getCurrentDatabase();
            Document doc = ac.getDocumentContext();
            // assign object variables to the fields that have the XML and XSLT code in them
            Item xmlItem = doc.getFirstItem("RawXML");
            InputSource style = new InputSource("http://wickedxml/fishnet/fishnet.nsf/HTMLTable.xsl");
            // create a File Writer, transform the xml and write it to the following file.
            FileWriter fw = new FileWriter("C:\temp\output.html");
            XSLTResultTarget result = new XSLTResultTarget(fw);
            xmlItem.transformXML(style, result);
            // write the same results to the results field in the current document
        }
    }
}
```

190 XML Powered by Domino
style = new InputSource("http://wickedxml/fishnet/fishnet.nsf/HTMLTable.xsl");
StringWriter rw = new StringWriter();
result = new XSLTResultTarget(rw);
xmlItem.transformXML(style, result);
doc.replaceItemValue("TransResult", rw.toString());
doc.save(true, true);

} catch(Exception e) {
    e.printStackTrace();
}

3. Save the agent.

4. Open the SSPrimer form design and copy the existing action “Transform XML to HTML” to the clipboard. Paste this action back into the action list. Edit the action name to “Transform XML to HTML Using Page Design.” Alter the formula so that it calls the correct agent:

@Command([ToolsRunMacro];"(XSLPrimerTransToHtmlUsingPage)"

5. Save the form design.

6. Return to the Notes client and open the document that contains the directory information. Click on “Transform XML to HTML Using Page design.”

7. Open c:\temp\output.html in a Web browser. You will notice that it looks exactly like the output from the last example. The difference is that we are reading the stylesheet from the page that is stored in the Domino database. This provides for an environment where all of the stylesheets that we need are accessible from the same Domino application that contains our XML data.

Using this same design concept, we could create a database on the Domino server that’s singular purpose is to be an XML object repository. This could be a database that stored all of the stylesheet designs and DTD schema definitions in pages. This database could then be replicated to servers throughout the environment to make these design elements accessible to all XML applications.

### 4.2.7 Transforming XML data on the server

We have discussed the mechanics of transforming XML data with an XSLT transformation engine. The transformation takes place wherever the
transformation engine lives. Transformation engines can live in standalone applications, Web clients and Web servers.

As developers, we have to decide which transformation engine we are going to use and if that processing will be done on the server or on the client.

In Domino and from the Domino platform this is an easy decision. It makes the most sense to use the XSLT transformation engine that is available with the product. The LotusXSL transformation engine has been included in the Domino server and Domino Designer client since release 5.03. This is also the engine that was used to create the predefined Java methods in the back end Java classes. The majority of Web clients today can not render XML data. This puts the burden of parsing the XML data and transforming it on the server. From the Domino server there are several processing models to choose from. In this lesson we explore some of these processing models for applying XSL stylesheets to XML data.

Applying stylesheets in Domino occurs in any of the following scenarios:

- Applying a stylesheet to live data from an outside source
- Applying a stylesheet to parsed XML data stored in a Domino document
- Applying a stylesheet to native Domino data represented as XML

There are many sources on the Internet, as well as subscription-based services, that supply content in an XML format. Many of these resources can be linked to directly and included in your application for real-time display of information. One of the best examples is accessing current news information. Once you have found a source for your news, you will need to evaluate the structure of the XML data so that you can author a stylesheet to return the information you are interested in. One of the free sources of news data on the Internet is:

http://w.moreover.com/

There are several categories of news topics. The data is available in XML as well as several other formats. We are interested in news about the food industry so the URL we will access is at:

http://p.moreover.com/cgi-local/page?index_food+xml

The raw XML for this news feed looks like this:

```xml
<?xml version="1.0" encoding="iso-8859-1" ?>
<!DOCTYPE moreovernews (View Source for full doctype... )>
<moreovernews>
<article id="_10693462">
  <url>http://c.moreover.com/click/here.pl?x10693350</url>
</article>
</moreovernews>
```
The root element of this XML is <moreovernews> with a series of first level <article> elements that are first level children. The good thing about this data is that the structure of the data is the same, regardless of the source we choose from this site. This means that we can author a stylesheet one time and have it applied to any of the news feeds that are supplied by this source.

After analyzing the data, we wrote the following stylesheet to transform the news feed into HTML on the fly. The following is the complete XSLT stylesheet code used to accomplished this:

```xml
<?xml version='1.0'?>
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:output method="html" />

    <xsl:template match="/">
        <html>
            <head>
                <title>News Feed</title>
```
This stylesheet is stored in a page design object called “NewFeedStyleSheet” with an alias of “NFSS.xsl”

We now have all of the elements that are needed to write an agent that will produce the HTML output on the fly from the Domino server. We have created a Java agent that parses the XML, transforms it to HTML and displays it to the requesting Web client. The Java code is stored in an agent called TransformNewsFeeds. The following is the complete Java code for this agent:

```java
import lotus.domino.*;
import com.lotus.xsl.*;
import com.lotus.xml.xml4j2dom.*;
import com.lotus.xsl.XSLTInputSource;
import com.lotus.xsl.XSLTResultTarget;
import java.io.*;

public class JavaAgent extends AgentBase {
    public void NotesMain() {
        try {
            XSLProcessor xp = new XSLProcessor(new XML4JLiaison4dom());
```
Unlike the agents code that we have reviewed up to this point, this agent is not using any native Domino objects to store or access the XML data. This means we will need to use the native methods of the LotusXSL package to process the XML and transform it. Starting at the beginning of the code we will review the functionality of the agent.

First we import the class files and methods we will use to complete the transformation:

```java
import lotus.domino.*;
import com.lotus.xsl.*;
import com.lotus.xml.xml4j2dom.*;
import com.lotus.xsl.XSLTInputSource;
import com.lotus.xsl.XSLTResultTarget;
import java.io.*;
```

The next few lines create the JavaAgent class and the NotesMain method. Next we create the processor using the XSLProcessor method.

```java
public class JavaAgent extends AgentBase {
    public void NotesMain() {
```
try {
    XSLProcessor xp = new XSLProcessor(new XML4JLiaison4dom());
    
    The first three lines of this code set up the variables for processing. In this example the sites are hard coded, but they could have easily been passed in from variables. Next we set up an input source to read the raw XML and XSLT data. The last line of code is where all of the work is done. The process method takes the XML and transforms it based upon the templates in the stylesheet document, and then writes the results to the XSLTResultTarget.

    String x = "http://p.moreover.com/cgi-local/page?index_food+xml";
    String s = "http://wickedxml/fishnet/fishnet.nsf/nfss.xsl";
    StringWriter rw = new StringWriter();

    XSLTInputSource xsrc = new XSLTInputSource(x);
    XSLTInputSource ssrc = new XSLTInputSource(s);
    XSLTResultTarget tout = new XSLTResultTarget(rw);

    xp.process(xsrc,ssrc,tout);

    If there is no error parsing the XML and applying the stylesheet, the HTML is written out to the requesting browser client.

    getAgentOutput().println("Content-type: text/html");
    getAgentOutput().println(rw.toString());

    The following URL will access this agent and execute the code:

    The output from this agent is shown in Figure 76.
Chapter 4. Transforming XML data in Domino

4.2.8 Storing remote XML in a Domino document

But what if there was a business need to store the data in Domino for archive purposes? It is nice to be able to process the information on the fly, but could it be stored in the database and transformed on demand later? Sure it can.

In Chapter 6 we will discuss a technique for reading an XML data source and storing the raw XML in a field in a Domino document. In this example we will process that raw XML from an agent. The data is stored in a document based upon the form design shown in Figure 77.
The XML is stored in a field named XMLDATA. A document that is based on this form design is created with a scheduled agent to read the news source and store it in the Domino database. When viewed through a Web client this document looks like Figure 78.
Chapter 4. Transforming XML data in Domino

Figure 78. Raw XML document viewed in a Web browser
The Display with Stylesheet action in the action bar uses the following formula:

@Command([ToolsRunMacro]; "TransformNewsFromDominoField")

This calls the TransformNewsFromDominoField agent that will read the raw XML data, transform it with our stylesheet and display it in the browser window. The output from the agent looks like that shown in Figure 79.

![Click Headlines to access news story!](image)

**Click Headlines to access news story!**

- **Kellogg eyeing Keebler?**
- **INDIA: Frito-Lay swallows rivals in India**
- **Do You Take Offense At Blueberry Bagels? Then Read No Further**
- **Kellogg seen as lead contender for Keebler**
- **Us Tsys Post Small Losses On Unilever Rte Locks; Stk Recovery**
- **Unilever to term out Bestfoods CP deal**
- **Salty snacks propel PepsiCo**
- **Genetic food labels can't be required, judge says**

Following is the Java agent code that produces this output:

```java
import lotus.domino.*;
import com.lotus.xsl.*;
import com.lotus.xml.xml4j2dom.*;
import com.lotus.xsl.XSLTInputSource;
import com.lotus.xsl.XSLTResultTarget;
```
import java.io.*;

public class JavaAgent extends AgentBase {

    public void NotesMain() {
        try {
            Session ns = getSession();
            AgentContext ac = ns.getAgentContext();
            Document doc = ac.getDocumentContext();
            Item it = doc.getFirstItem("XMLdata");

            XSLProcessor xp = new XSLProcessor(new XML4JLiaison4dom());

            String s = "http://wickedxml/fishnet/fishnet.nsf/nfss.xsl";
            StringWriter rw = new StringWriter();
            XSLTInputSource xsrc = new XSLTInputSource((it.getInputSource()));
            XSLTInputSource ssrc = new XSLTInputSource(s);
            XSLTResultTarget tout = new XSLTResultTarget(rw);
            xp.process(xsrc,ssrc,tout);
            getAgentOutput().println("Content-type: text/html");
            getAgentOutput().println(rw.toString());
        } catch(Exception e) {
            getAgentOutput().println("Content-type: text/html");
            getAgentOutput().println("<html><head><title>Agent Error</title></head><body>Agent Error</body></html>");
            getAgentOutput().println(e);
            getAgentOutput().println("</body></html>");
        }
    }
}

This agent code is similar to the code that was used to transform XML data from the direct URL. The differences in this code have been emphasized with bold type. Using the Java back-end classes we can get a handle to the current document object that includes the XMLDATA field where the raw XML is stored. Once we have a handle to that field we can call the getInputSource() method of the Item class to create and inputSource object from the raw XML that is then passed to the XSL transformation engine to be processed.
4.2.9 Transforming XML data from the browser client

When transforming XML data, the transformation will either occur on the server or on the browser client. From the development point of view this means writing applications that call the client's processing capabilities or processing the XML data on the server and then sending the data down to the client. From a performance perspective there are benefits to offloading the processing to the client. This frees up resources on the server to handle more requests. If all Web clients have the same capabilities, then this is the ideal processing model.

This processing model assumes all Web clients have the same capabilities for processing XML data and performing transformations. At the time of this writing that is not the case. There are two major Web browsers that are used in the corporate environment:

- Microsoft Internet Explorer
- Netscape Navigator

Currently only the 5.x versions of Microsoft's Web client includes capabilities for parsing XML data and performing transformations. These versions include an XML parser as well as an XSLT transformation engine. Although Netscape has done a great deal to include XML processing capabilities into its next browser, Netscape 6, it is only available in preview release. None of the earlier versions of Netscape support processing XML data.

Even after Netscape does ship the next version of Navigator, the XML processing capabilities will not be the same as the ones in IE5.0. What this means to the developer community is that we will have to take the client out of the equation, and process XML on the server side to supply data to any browser client.

How does this relate to Domino? Well, really nothing has changed. Most of what we have discussed up to this point has been server-side processing. Through the use of agents we can process XML data and provide the output to any requesting browser client.

But, if in a particular environment the browser client is known and will be consistent, then we can take advantage of the client-side processing capabilities of that client. In the next section we will discuss several of the client-side processing capabilities of Microsoft Internet Explorer.

4.2.9.1 XML data islands

A data island is an XML document that is embedded in an HTML page. Using a scripting language at the client, usually Javascript, calls can be made to the
client's native XML processing capabilities. The format of the data island would be:

```xml
<XML ID="XMLDATAID">
....XML document .....  
</XML>
```

Using a Domino document as the source for an HTML page makes creating data islands very easy. Just include a field that contains well-formed XML data in between the XML island tags. A form could be created that displays raw XML data in a field enclosed by the above tags. An example of the form design is shown in Figure 80.

![Figure 80. Domino form design with XML data island](image)

The text attribute “Pass-Thru HTML” has been applied to all of the lines. Let’s examine the syntax line by line. The fist three lines make up the data island itself. Using the ID tag, Internet Explorer can reference the island as an XML document object.
The very next line sets up a reference to an XSLT stylesheet. Notice that it has both an ID tag and an src or source tag. The src tag points to the location of the XSLT stylesheet relative to the XML document. In this case it is an XSLT stylesheet stored in a page design in the current database.

The only thing different about this stylesheet is the namespace being used. This is the syntax of the stylesheet:

```xml
<?xml version='1.0'?>
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/TR/WD-xsl">

<xsl:template match="/">
<html>
<head>
<title>News Feed</title>
</head>
<body>
<H1>Click Headlines to access News Item !</H1>
   <xsl:apply-templates select="moreovernews" />
</body>
</html>
</xsl:template>

<xsl:template match="moreovernews">
   <xsl:apply-templates select="article" />
</xsl:template>

<xsl:template match="article">
<H3>
   <xsl:element name="a">
      <xsl:attribute name="href"><xsl:value-of select="url"/></xsl:attribute>
      <xsl:value-of select="headline_text" />
   </xsl:element>
</H3>
</xsl:template>

</xsl:stylesheet>
```

Next is the Javascript code that does the actual work. Here, on the window event onLoad, the stylesheet is applied to the XML document and the
transformed output is displayed in the innerHTML source of the Div tag at the end of the form. It is important to point out that even though we are using Domino design elements, none of the processing is taking place on the Domino server. Referencing the ID value of the XML data island tag newsXML, the transformNode method is called to transform the output to HTML. The transformNode method is unique to IE5.x. This means that the only browser client that will be able to render this document correctly is IE 5.x.

```<SCRIPT FOR="window" EVENT="onLoad">
    XMLOutput.innerHTML=newsXml.transformNode(NewsStyle.XMLDocument);
</SCRIPT>
```

When displayed in the Notes client, the document would look like the one shown in Figure 81.

![Figure 81. Domino document with a data island](image)

Internet Explorer has both an XML parser and an XSLT transformation engine incorporated into it, so it can parse the XML data and transform it with the stylesheet and return the results to the DIV tag.

When displayed in Internet Explorer, the output would be like that shown in Figure 82 on page 206.
If the browser client is known to be Internet Explorer 5.x, for example on an intranet where all the Web clients are the same, then the specific features in IE 5.x can be used. The following code demonstrates how to point to a live XML data source and use a stylesheet that is stored in a page design element on a Domino server to style the output and render it to the Web client.

```html
<HTML>
<HEAD>

<SCRIPT LANGUAGE="JavaScript" FOR="window" EVENT="onLoad">
// Load data.
var source = new ActiveXObject("Microsoft.DOMXML");
source.async = false
source.load("http://p.moreover.com/cgi-local/page?index_food+xml");

// Load stylesheet.
var stylesheet = new ActiveXObject("Microsoft.DOMXML");
stylesheet.async = false
stylesheet.load("http://wickedxml/fishnet/fishnet.nsf/nfie.xsl");
XMLoutput=source.transformNode(stylesheet);
content.innerHTML = XMLoutput;

</SCRIPT>
```
The first thing you will notice is that this is a static HTML document. But because it points to a live XML source that is constantly updated, this page can be used to serve dynamic content.

The output from this HTML looks like Figure 83.

![Click Headlines to access News Item!](image)

**Figure 83. Output from HTML using Microsoft ActiveX objects**

We were able to accomplish the same results using a Domino agent. The difference between the two is in which resources are being utilized. With this scenario the only thing the Domino server has to do is serve the static page. All of the work to retrieve the XML source and transform the data is done by the client.
One of the benefits of client-side processing is downloading several stylesheets to allow the users to choose alternate views of the data. For example, the original output is in chronological order, but some might want to see it in sorted order. We can alter the page design with the following changes to accommodate this. The differences are in bold text.

```html
<HTML>
<HEAD>

<SCRIPT LANGUAGE="JavaScript" FOR="window" EVENT="onLoad">
XMLoutput = "";
// Load data.
source = new ActiveXObject("Microsoft.XMLDOM");
source.async = false
source.load("http://p.moreover.com/cgi-local/page?index_food+xml");

// Load stylesheet.
stylesheet = new ActiveXObject("Microsoft.XMLDOM");
stylesheet.async = false
stylesheet.load("http://wickedxml/fishnet/fishnet.nsf/nfie.xsl");

// Load second stylesheet
sortStyle = new ActiveXObject("Microsoft.XMLDOM");
sortStyle.async = false
sortStyle.load("http://wickedxml/fishnet/fishnet.nsf/nfie2.xsl");

// Transform XML
XMLoutput = source.transformNode(stylesheet);
content.innerHTML = XMLoutput;
</SCRIPT>

</HEAD>
<BODY>
<script>
function sort(){
XMLoutput = source.transformNode(sortStyle);
content.innerHTML = XMLoutput;
}
function unsort(){
XMLoutput = source.transformNode(stylesheet);
content.innerHTML = XMLoutput;
}</script>

<input type="button" value="Sort the Output" onClick="sort()" />
<input type="button" value="Display Original" onClick="unsort()" />

<!-- Output gets inserted into this DIV -->
</BODY>
</HTML>
```
We have created two buttons in the HTML document that use Javascript to call the sort function and the unsort function, these buttons apply the appropriate stylesheet. Once again the benefit here is that the work is being done by the client and no server resources are being utilized.

The result of these changes are displayed in Figure 84.

![HTML using Microsoft XMLDOM with sort buttons](http://wickedxml/fishnet/fishnet.net/XMLDOMSortOpenPage)

**Click Headlines to access News Item!**

- Unilever’s offer to buy IBL shares on Nov. 27
- Nestle to turn around ice cream business
- Safeway Recalls Taco Shells
- Vitamin makers pay US$442 million

*Figure 84. HTML using Microsoft XMLDOM with sort buttons*

After the Sort the Output button is clicked, it will look like the example shown in Figure 85 on page 210.
The stylesheet that we use to sort the output is slightly different from the one we used in the earlier examples. In the earlier examples we were able to use the \texttt{xsl:sort} command to sort the output returned by a particular template. But not all versions of IE5 support the \texttt{xsl:sort} command. The same sorting results can be accomplished by using the following stylesheet:

```xml
<?xml version='1.0'?>
<xsl:stylesheet
   xmlns:xsl="http://www.w3.org/TR/WD-xsl">

   <xsl:template match="/">
   <html>
   <head>
   <diluteness Feed</title>
   </head>
   <body>
   <H1>Click Headlines to access News Item!</H1>
   <xsl:for-each select="moreovernews/article" order-by=""headline_text"">
   <H3>
   <xsl:element name="a">
   <xsl:attribute name="href"><xsl:value-of select="url"/></xsl:attribute>
   
   <xsl:attribute name="href">ADM testing for presence of StarLink corn at all elevators and processing plants
   Baby boomers drive functional ingredient market
   Cadbury to take chair at Transense
   Canon Foods creates micro food guarantees
   </xsl:attribute>
   ```
To have the data appear in an ordered output we use the `xsl:for-each` command with the select and order-by attributes. This works like most other for loop processes in other languages. For each article element return the values enclosed in the for structure ordered by the headline_text element.

For a complete reference to the client-side capabilities of Internet Explorer, access the developer Web site at:

http://www.msdn.microsoft.com/xml/default.asp

For updated information on the XML capabilities in the next release of Netscape Navigator, access the Netscape Web site at:

http://developer.netscape.com/tech/xml/

### 4.2.10 Using Domino Design elements to transform and display data

In all of the previous examples we have used Domino agents, pages and forms to render and process XML data.

Table 16 summarizes the design elements and how they are used to process XML data.

<table>
<thead>
<tr>
<th>Design element</th>
<th>XML capability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agents</td>
<td>Domino agents are the most powerful tool at your disposal when processing XML data. They use both Java and LotusScript to process and render XML data in the Notes client as well as the browser client.</td>
</tr>
<tr>
<td>Servlets</td>
<td>Servlets can be used to send and receive XML-based data. The major benefit of a servlet is its ability to process server-to-server communications for transporting XML data.</td>
</tr>
<tr>
<td>Pages</td>
<td>Pages can be used to store XSLT stylesheets as well as Document Type Definitions. They can also be used to hold embedded view designs that are formatted as XML.</td>
</tr>
</tbody>
</table>
4.3 Transforming from XML to XML

Of course, one of the most useful purposes of XSLT is to transform one flavor of XML to another. In Chapter 2 we outlined the steps for transforming simple data from one XML source to another, and making sure that the transformed data conforms to a simple DTD. The fact is, however, as XML representations of data for specific purposes are formed, their data structures, and hence their DTDs, become more and more complex. For example, if one was to print out the Domino DTD they would find that it is more than 30 pages long, and at the time of this writing was not fully complete. This simply represents the data requirements for design elements in Domino databases, and doesn’t even try to model the content that could potentially be stored in the database (with the exception of rich text field containment.) However, keep in mind that XML data and DTDs are not meant to be read by human eyes, but are made as road maps to data that different machines can pass regardless of underlying data platforms. To get more of an idea of the types of standard XML representations and their associated DTDs, visit:

http://www.xml.org/xmlorg_registry/index.shtml

This is a listing of standard XML formats based on industry organization efforts, and grouped by industry.

Another cool Web site that is just getting off the ground is the XSL stylesheet registry at:

http://www.xmltree.com/xsl/results.cfm

This is an interesting format, and hopefully very useful in the future. A user has to enter the standard XML format they’re starting with, the destination XML format, and the transformation standard by which to transform XML. Search results list all stylesheets that enable that specific type of transformation.
4.3.1 Transforming to an alternate XML format

For a good example of data represented in a standardized XML format, let's have a look at the DXL that was generated by our standalone Java application at the end of Chapter 4. Even though the data itself only represents four fields on a Domino document, the XML wrapped around the content makes the DXL output file quite large. Here's the original DXL data in its entirety:

```xml
<?xml version="1.0" encoding="utf-8" ?>
<document xmlns="http://www.lotus.com/dxl" version="0.13"
form="FishNameResults">
  <noteinfo noteid="fde" unid="2ABA0B582793B9618525695F00567C7A"
sequence="1">  
    <created>
      <datetime dst="true">20000919T114439,62-04</datetime>
    </created>
    <modified>
      <datetime dst="true">20000919T115208,01-04</datetime>
    </modified>
    <revised>
      <datetime dst="true">20000919T114439,61-04</datetime>
    </revised>
    <lastaccessed>
      <datetime dst="true">20000919T115208,01-04</datetime>
    </lastaccessed>
    <addedtofile>
      <datetime dst="true">20000919T115208,01-04</datetime>
    </addedtofile>
  </noteinfo>
  <item name="FishName">
    <text>Shark</text>
  </item>
  <item name="ButtonPressed">
    <text>FN</text>
  </item>
  <item name="ButtonPressedValue">
    <text>Shark</text>
  </item>
  <item name="$UpdatedBy" names="true">
    <textlist>
      <text>Anonymous</text>
    </textlist>
  </item>
  <item name="FishID" summary="false">
    <textlist>
      <text>640</text>
      <text>641</text>
    </textlist>
  </item>
</document>
```
<textlist>
- <item name="Subcategory" summary="false">
- <textlist>
  - <text>Blacktip</text>
  - <text>Bronze Whaler</text>
  - <text>Eastern School</text>
  - <text>Grey Nurse</text>
  - <text>Gummy</text>
  - <text>Hammerhead</text>
  - <text>other</text>
  - <text>Pencil</text>
  - <text>Spurdog</text>
  - <text>Thickskin</text>
  - <text>Tiger</text>
  - <text>Whiskery</text>
  - <text>White Pointer</text>
  - <text>Wobbegong</text>
</itemlist>
</item>
- <item name="FishDescription" summary="false">
- <textlist>
  - <text>Shark</text>
  - <text>Shark</text>
  - <text>Shark</text>
  - <text>Shark</text>
  - <text>Shark</text>
  - <text>Shark</text>
  - <text>Shark</text>
  - <text>Shark</text>
  - <text>Shark</text>
  - <text>Shark</text>
  - <text>Shark</text>
  - <text>Shark</text>
</itemlist>
</item>

214  XML Powered by Domino
4.3.1.1 Elements of a DXL document
Let's have a look at the content represented in this DXL document. Aside from the root element and the DXL document attributes, there are three types of elements represented in the DXL.

The first type of element is the actual data that represents the content in this document. These elements need to be passed to other XML formats. They are identified in Table 17.

The second type of element is content used by the Domino application database while processing this record. These elements represent Domino fields that are used by the FishNet application to process data, but serve no useful purpose out of context of the original application. These types of elements can be stripped out of the XML document. They are listed in Table 18.

The third type of elements represent Domino document properties that are used to produce a valid DXL document. These include system settings that may or may not need to be used by Domino when validating a DXL document, if and when that document is imported back into Domino. Table 19 lists these elements.

Table 17. Elements that represent actual content in a DXL document

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Domino Field Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>FishName</td>
<td>Editable Text</td>
</tr>
<tr>
<td>Fish ID</td>
<td>Editable List</td>
</tr>
<tr>
<td>SubCategory</td>
<td>Editable List</td>
</tr>
<tr>
<td>FishDescription</td>
<td>Computed List</td>
</tr>
</tbody>
</table>
Table 18. Elements that represent Domino application fields in a DXL document

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Domino Field Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ButtonPressed</td>
<td>Computed Text</td>
</tr>
<tr>
<td>ButtonPressedValue</td>
<td>Computed Text</td>
</tr>
<tr>
<td>DECSACT___Fishname</td>
<td>Computed Text</td>
</tr>
<tr>
<td>$DECSCluster</td>
<td>Computed Text</td>
</tr>
</tbody>
</table>

Table 19. Elements that represent Domino document properties in DXL document

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Element Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>form</td>
<td>Domino form name</td>
</tr>
<tr>
<td>noteinfo</td>
<td>Domino UNID and noteid</td>
</tr>
<tr>
<td>created</td>
<td>Date of creation</td>
</tr>
<tr>
<td>modified</td>
<td>Date of last modification</td>
</tr>
<tr>
<td>revised</td>
<td>Date of last content revision</td>
</tr>
<tr>
<td>lastaccessed</td>
<td>Date last accessed</td>
</tr>
<tr>
<td>addedtofile</td>
<td>Date added to the database</td>
</tr>
<tr>
<td>$UpdatedBy</td>
<td>Domino UpdatedBy names field</td>
</tr>
</tbody>
</table>

As you can see from these tables, DXL data takes up a large part of DXL documents. If this DXL document was created for the purpose of transporting the data to another Domino document, then the format is fine the way it is. However, in a real-world context this data will probably have to be transformed to a format more closely representing the content in the document, rather than the document itself.

4.3.1.2 DXL transformation via XSLT

Let's imagine the following real-world scenario: ReallyBigFishRetailer.com is excited that your Web site has moved into the world of XML, but it needs you to transform your data to their own flavor of XML, RBFRXML, before it can be added to their site. They supply you with specifications and a DTD (which we will review later in this chapter). Because ReallyBigFishRetailer is one of your larger customers, you’re happy to oblige in order to keep them happy. The customer’s DTD is always right!
ReallyBigFishRetailer is definitely not interested in the name of the person who updated this document, when it was last updated, or any other internal fields that help you work with the document on your system. All they want is data on fish. This would include the FishID, FishName, Subcategories of Fish and any Descriptions.

4.3.1.3 Extensible StyleSheet format

In order to send this data, you need to transform your DXL using this Extensible Stylesheet. In the real world, the customer will probably provide you with a DTD document and you will have to write a stylesheet that transforms your data to something that will validate to the customer's DTD. For now, here's the sheet we've created for this example in it's entirety; we'll go over the functionality of the transformation using this sheet in subsequent sections:

```xml
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="xml"/>
  <xsl:template match="/">
    <FishList>
      <Fish>
        <xsl:apply-templates select="/document/item[@name='FishName']/FishName"/>
        <FishIDList><xsl:apply-templates select="/document/item[@name='FishID']"/></FishIDList>
        <FishTypes><xsl:apply-templates select="/document/item[@name='Subcategory']"/></FishTypes>
        <xsl:apply-templates select="/document/item[@name='FishDescription']"/>
      </Fish>
    </FishList>
  </xsl:template>

  <xsl:template match="item[@name='FishName']">
    <FishName>
      <xsl:value-of select="."/>
    </FishName>
  </xsl:template>

  <xsl:template match="item[@name='FishID']">
    <FishID>
    </FishID>
  </xsl:template>
</xsl:stylesheet>
```
We’ll cover the functionality of this stylesheet in more detail in later sections of this chapter.

### 4.3.1.4 Activating the transformation

Transformation is triggered via a Java agent from the Domino server. It picks up the DXL file from where we placed it in Chapter 4, located in c:/temp/DXLFishRecord.XML. It uses the stylesheet listed in the previous section, which is located in c:/temp/DXLtoFishList.xsl. XML output results are sent to C:\temp\output.xml. Here's a look at the Java Code that processes the XSLT transformation:

```java
import lotus.domino.*;
import com.lotus.xsl.*;
import com.lotus.xml.xml4j2dom.*;
import com.lotus.xsl.XSLTInputSource;
import com.lotus.xsl.XSLTResultTarget;
import java.io.*;
```
public class JavaAgent extends AgentBase {

    public void NotesMain() {
        try {
            XSLProcessor xp = new XSLProcessor(new XML4JLiaison4DOM());

            String x = "c:/temp/DXLFishRecord.XML";
            String s = "c:/temp/DXLFishList.xsl";

            XSLTInputSource xsrc = new XSLTInputSource(x);
            XSLTInputSource ssrc = new XSLTInputSource(s);

            FileWriter fw = new FileWriter("C:\temp\output.xml");
            XSLTResultTarget result = new XSLTResultTarget(fw);
            xp.process(xsrc, ssrc, result);

            getAgentOutput().println("Content-type: text/html");
            getAgentOutput().println("<html><head><title>XML Transformation Completed</title>
            Results are in C:\TEMP\OUTPUT.xml");
            getAgentOutput().println("</body></html>");
        }
        catch (Exception e) {
            getAgentOutput().println("Content-type: text/html");
            getAgentOutput().println("<html><head><title>Agent Error</title>
            Error</body></html>");
            getAgentOutput().println(e);
            getAgentOutput().println("</body></html>");
        }
    }
}

Let’s go through this code line by line and review the functionality:

import lotus.domino.*;
import com.lotus.xsl.*;
import com.lotus.xml.xml4j2dom.*;
import com.lotus.xsl.XSLTInputSource;
import com.lotus.xsl.XSLTResultTarget;
import java.io.*;

These lines import all of the classes we will be working with for this example. It’s worth noting at this point that LotusXSL.jar and XML4J.jar are required to
be on the system path or CLASSPATH when running these examples from an
agent. If you're running Domino 5.03 or higher, these jar files are installed
automatically in \lotus\notes on a client machine and \lotus\domino on the
server.

public class JavaAgent extends AgentBase {

    public void NotesMain() {

        try {
            XSLProcessor xp = new XSLProcessor(new XML4JLiaison4dom());

            String x = "c:/temp/DXLFishRecord.XML";
            String s = "c:/temp/DXLtoFishList.xsl";

            XSLTInputSource xsrc = new XSLTInputSource(x);
            XSLTInputSource ssrc = new XSLTInputSource(s);

            FileWriter fw = new FileWriter("C:\temp\output.xml");
            XSLTResultTarget result = new XSLTResultTarget(fw);

            xp.process(xsrc,ssrc,result);

            getAgentOutput().println("Content-type: text/html");
            getAgentOutput().println("<html><head><title>XML Transformation
Completed</title></head><body>");
            getAgentOutput().println("Results are in C:\TEMP\OUTPUT.xml");
            getAgentOutput().println("</body></html>");

These lines set up the Java agent environment by extending AgentBase and
creating a NotesMain. Next we identify the type of parser that we will use in
our transformation, which is the XML4j XML parsing processor using the
DOM interface. The DOM XSL processor needs three things to operate: two
input sources and one output. The first thing is the original XML file that we
want to manipulate, which in this case is the DXL output file from our
standalone Java application in Chapter 4. Next we need the XSL stylesheet
that will be used to transform the data. In this case we're using the stylesheet
listed previously in this section, which is on the local hard drive at
c:\temp\DXLtoFishList.xsl The last thing we need to do is define the output
format to be created by this transformation. In this case we'll create a new
.xml file called c:\temp\output.xml by employing the FileWriter class, an
extension of the java.io.writer class. Then we point the XSL transformation
output source to the file by creating a new instance of the XSLTResultTarget
class that instantiate the FileWriter.

xp.process(xsrc,ssrc,result);

getAgentOutput().println("Content-type: text/html");
getAgentOutput().println("<html><head><title>XML Transformation
Completed</title></head><body>");
getAgentOutput().println("Results are in C:\TEMP\OUTPUT.xml");
getAgentOutput().println("</body></html>");
The final step is to process the transformation, which writes the output to the file. This is done on a client machine by invoking the agent via URL from a browser:


4.3.1.5 Transformed data

In addition to removing all data associated with Domino document properties and other non-essential field data associated with processing this document on a Domino server, the stylesheet called by the Java agent also reorganizes and renames all the elements in the original document to match the needs of our customer. Here’s a look at the output generated by this transformation:

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<FishList>
  <Fish>
    <FishName>Shark</FishName>
    <FishIDList>
      <FishType ID="640">640</FishType>
      <FishType ID="641">641</FishType>
      <FishType ID="642">642</FishType>
      <FishType ID="643">643</FishType>
      <FishType ID="644">644</FishType>
      <FishType ID="645">645</FishType>
      <FishType ID="646">646</FishType>
      <FishType ID="647">647</FishType>
      <FishType ID="648">648</FishType>
      <FishType ID="649">649</FishType>
      <FishType ID="650">650</FishType>
      <FishType ID="651">651</FishType>
      <FishType ID="652">652</FishType>
    </FishIDList>
    <FishTypes>
      <FishName Name="Blacktip">Blacktip</FishName>
    </FishTypes>
  </Fish>
</FishList>
```
Now that we have looked at an overview of the original data, the methods of transformation via an extensible stylesheet, the Java agent that triggers the transformation, and the resulting transformed XML, let’s go through the process of transformation one step at a time.

### 4.3.2 Transforming elements to elements

Now it’s time to explain what the stylesheet is telling the XSLT transformation engine to do with the input and how that controls the output. In this case, the first thing we want to do is create a new XML document, then strip away all non-essential elements in the XML document, leaving only those elements that directly deal with the content contained in the document. Let’s go over the base functionality of the stylesheet used for this first part of the transformation:

```xml
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:output method="xml"/>

<xsl:template match="/"/>

<FishList>
  <Fish>
    <xsl:apply-templates select = "/document/item[@name='FishName']"/>
  </Fish>
</FishList>
```
The first thing we do is tell the StyleSheet that we are dealing with XML as output. The next thing we do is select the FishID, FishName, Subcategory and FishDescription fields specifically from the root note in the DOM. This means that all other elements will be ignored. Because the DXL output has written these Domino fields (and most standard output besides document properties) to elements called item, we need to retrieve the name attribute from each element to choose the elements we want. This is done using the @ symbol followed by the attribute name that we want to return the value of:

This command will select all XML elements that match the pattern /document/item containing an attribute called name whose value is FishName. At this point in the stylesheet we also identify some of the XML elements that will be in the output by wrapping the FishList, Fish, FishIDList, and FishTypes around the elements returned when templates are applied.

Now that we have selected our output elements and wrapped some XML elements around them for output, we can begin transformation of individual elements. The easiest way to transform the DXL data listed in the previous example to another XML format is to transfer data directly from one format to another, replacing element names from the original DXL file with element names in the new format. In this case we have one original element that we transfer to a new element, the FishDescription Field. This is shown in Table 20 on page 224.
Table 20. DXL element to XML element transformation

<table>
<thead>
<tr>
<th>DXL Element Data</th>
<th>XML Output Data</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;item name=&quot;FishDescription&quot; summary=&quot;false&quot;&gt;</code></td>
<td><code>&lt;FishDescription&gt;</code></td>
</tr>
</tbody>
</table>
Table 21. DXML attribute to XML element transformation

<table>
<thead>
<tr>
<th>DXML Element Data</th>
<th>XML Output Data</th>
</tr>
</thead>
</table>
| `<item name='FishName'>
  <text>Shark</text>
  </item>` | `<FishName>
  Shark
  </FishName>` |

In this case, we pull DXML data from an item element with a name attribute of FishName and transform it to an element called FishName. Here's the part of the stylesheet that does this:

```xml
<xsl:template match="item[@name='FishName']">
  <FishName>
    <xsl:value-of select = "."/>
  </FishName>
</xsl:template>
```

The FishName template match code instructs the XSLT DOM processor to match any values that match the pattern `/document/item` containing an attribute called name whose value is FishName. For each match that is found, we wrap XML tags around the value of the element and pass this new XML element back to the XML document. In this case, there is only one FishName, so the value is passed back to the XML processor as a single element.

4.3.4 Transforming elements to attributes and attributes to attributes

In our example there are two places where we transform DXML elements to XML elements and attributes, as shown in Table 22.
<table>
<thead>
<tr>
<th>DXL Element Data</th>
<th>XML Output Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>: &lt;item name=&quot;FishID&quot; summary=&quot;false&quot;&gt;</td>
<td>: &lt;FishIDList&gt;</td>
</tr>
<tr>
<td>: &lt;textlist&gt;</td>
<td>: &lt;FishType ID=&quot;640&quot;&gt;640&lt;/FishType&gt;</td>
</tr>
<tr>
<td>: &lt;text&gt;640&lt;/text&gt;</td>
<td>: &lt;FishType ID=&quot;641&quot;&gt;641&lt;/FishType&gt;</td>
</tr>
<tr>
<td>: &lt;text&gt;641&lt;/text&gt;</td>
<td>: &lt;FishType ID=&quot;642&quot;&gt;642&lt;/FishType&gt;</td>
</tr>
<tr>
<td>: &lt;text&gt;642&lt;/text&gt;</td>
<td>: &lt;FishType ID=&quot;643&quot;&gt;643&lt;/FishType&gt;</td>
</tr>
<tr>
<td>: &lt;text&gt;643&lt;/text&gt;</td>
<td>: &lt;FishType ID=&quot;644&quot;&gt;644&lt;/FishType&gt;</td>
</tr>
<tr>
<td>: &lt;text&gt;644&lt;/text&gt;</td>
<td>: &lt;FishType ID=&quot;645&quot;&gt;645&lt;/FishType&gt;</td>
</tr>
<tr>
<td>: &lt;text&gt;645&lt;/text&gt;</td>
<td>: &lt;FishType ID=&quot;646&quot;&gt;646&lt;/FishType&gt;</td>
</tr>
<tr>
<td>: &lt;text&gt;646&lt;/text&gt;</td>
<td>: &lt;FishType ID=&quot;647&quot;&gt;647&lt;/FishType&gt;</td>
</tr>
<tr>
<td>: &lt;text&gt;647&lt;/text&gt;</td>
<td>: &lt;FishType ID=&quot;648&quot;&gt;648&lt;/FishType&gt;</td>
</tr>
<tr>
<td>: &lt;text&gt;648&lt;/text&gt;</td>
<td>: &lt;FishType ID=&quot;649&quot;&gt;649&lt;/FishType&gt;</td>
</tr>
<tr>
<td>: &lt;text&gt;649&lt;/text&gt;</td>
<td>: &lt;FishType ID=&quot;650&quot;&gt;650&lt;/FishType&gt;</td>
</tr>
<tr>
<td>: &lt;text&gt;650&lt;/text&gt;</td>
<td>: &lt;FishType ID=&quot;651&quot;&gt;651&lt;/FishType&gt;</td>
</tr>
<tr>
<td>: &lt;text&gt;651&lt;/text&gt;</td>
<td>: &lt;FishType ID=&quot;652&quot;&gt;652&lt;/FishType&gt;</td>
</tr>
<tr>
<td>: &lt;text&gt;652&lt;/text&gt;</td>
<td>: &lt;FishType ID=&quot;653&quot;&gt;653&lt;/FishType&gt;</td>
</tr>
<tr>
<td>: &lt;textlist&gt;</td>
<td>: &lt;/FishIDList&gt;</td>
</tr>
<tr>
<td>: &lt;/item&gt;</td>
<td></td>
</tr>
<tr>
<td>: &lt;item name=&quot;Subcategory&quot; summary=&quot;false&quot;&gt;</td>
<td>: &lt;FishTypes&gt;</td>
</tr>
<tr>
<td>: &lt;textlist&gt;</td>
<td>: &lt;FishName Name=&quot;#Blacktip&quot;&gt;Blacktip&lt;/FishName&gt;</td>
</tr>
<tr>
<td>: &lt;text&gt;Blacktip&lt;/text&gt;</td>
<td>: &lt;FishName Name=&quot;#Bronze Whaler&quot;&gt;Bronze Whaler&lt;/FishName&gt;</td>
</tr>
<tr>
<td>: &lt;text&gt;Bronze Whaler&lt;/text&gt;</td>
<td>: &lt;FishName Name=&quot;#Eastern School&quot;&gt;Eastern School&lt;/FishName&gt;</td>
</tr>
<tr>
<td>: &lt;text&gt;Eastern School&lt;/text&gt;</td>
<td>: &lt;FishName Name=&quot;#Grey Nurse&quot;&gt;Grey Nurse&lt;/FishName&gt;</td>
</tr>
<tr>
<td>: &lt;text&gt;Grey Nurse&lt;/text&gt;</td>
<td>: &lt;FishName Name=&quot;#Gummy&quot;&gt;Gummy&lt;/FishName&gt;</td>
</tr>
<tr>
<td>: &lt;text&gt;Gummy&lt;/text&gt;</td>
<td>: &lt;FishName Name=&quot;#Hammerhead&quot;&gt;Hammerhead&lt;/FishName&gt;</td>
</tr>
<tr>
<td>: &lt;text&gt;Hammerhead&lt;/text&gt;</td>
<td>: &lt;FishName Name=&quot;#other&quot;&gt;other&lt;/FishName&gt;</td>
</tr>
<tr>
<td>: &lt;text&gt;other&lt;/text&gt;</td>
<td>: &lt;FishName Name=&quot;#Pencil&quot;&gt;Pencil&lt;/FishName&gt;</td>
</tr>
<tr>
<td>: &lt;text&gt;Pencil&lt;/text&gt;</td>
<td>: &lt;FishName Name=&quot;#Spurdog&quot;&gt;Spurdog&lt;/FishName&gt;</td>
</tr>
<tr>
<td>: &lt;text&gt;Spurdog&lt;/text&gt;</td>
<td>: &lt;FishName Name=&quot;#Tiger&quot;&gt;Tiger&lt;/FishName&gt;</td>
</tr>
<tr>
<td>: &lt;text&gt;Tiger&lt;/text&gt;</td>
<td>: &lt;FishName Name=&quot;#Thickskin&quot;&gt;Thickskin&lt;/FishName&gt;</td>
</tr>
<tr>
<td>: &lt;text&gt;Thickskin&lt;/text&gt;</td>
<td>: &lt;FishName Name=&quot;#Whiskery&quot;&gt;Whiskery&lt;/FishName&gt;</td>
</tr>
<tr>
<td>: &lt;text&gt;Whiskery&lt;/text&gt;</td>
<td>: &lt;FishName Name=&quot;#White Pointer&quot;&gt;White Pointer&lt;/FishName&gt;</td>
</tr>
<tr>
<td>: &lt;text&gt;White Pointer&lt;/text&gt;</td>
<td>: &lt;FishName Name=&quot;#Wobbegong&quot;&gt;Wobbegong&lt;/FishName&gt;</td>
</tr>
<tr>
<td>: &lt;/textlist&gt;</td>
<td>: &lt;/FishTypes&gt;</td>
</tr>
<tr>
<td>: &lt;/item&gt;</td>
<td></td>
</tr>
</tbody>
</table>
In both of the cases show in this illustration, we're transforming elements called item with generic item, textlist, and text nested elements to more meaningful element names and attributes in the context of the fish data that the document represents. We do this by pulling out the value of the attribute called name and using it as the basis for new elements and attributes. Here's the section of the stylesheet that produces these transformations:

```xml
<xsl:template match="item[@name='FishID']">
  <xsl:for-each select="textlist/text">
    <FishType>
      <xsl:attribute name="ID">
        <xsl:value-of select="."/>
      </xsl:attribute>
      <xsl:value-of select="."/>
    </FishType>
  </xsl:for-each>
</xsl:template>

<xsl:template match="item[@name='Subcategory']">
  <xsl:for-each select="textlist/text">
    <FishName>
      <xsl:attribute name="Name">
        <xsl:value-of select="."/>
      </xsl:attribute>
      <xsl:value-of select="."/>
    </FishName>
  </xsl:for-each>
</xsl:template>
```

The functionality is identical for both the FishID and Subcategory processing, so we'll focus on the second template, FishName. The FishName template match code instructs the XSLT DOM processor to match any values that match the pattern /document/item containing an attribute called name and a value of FishName. For each match that is found, we use a for-each instruction and an attribute tag to wrap an attribute into the returned element called ID. The XML element tags are wrapped around the returned data, transforming the nested textlist/text elements of an item element into a FishType Element with an ID attribute.

### 4.3.5 Validating your transformed data

Of course, all of this transformation work is for nothing if the data you send to the ReallyBigFishRetailer doesn't validate with their DTD. In order to make sure that the data generated is going to validate properly against the customer's, the bigfish.dtd must be in the same directory as the XML file that
is being parsed and validated, or available via URL syntax from the machine that is doing the parsing. Let’s have a look at the DTD sent to us by ReallyBigFishRetailer:

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!ELEMENT FishList (Fish)+>
<!ELEMENT Fish (FishName, FishIDList,FishTypes,FishDescription)>
<!ELEMENT FishIDList (FishType)>
<!ELEMENT FishTypes (FishName)>
<!ELEMENT FishName (#PCDATA)>
<!ELEMENT FishType (#PCDATA)>
<!ATTLIST FishType ID CDATA #REQUIRED>
<!ELEMENT FishDescription (#PCDATA)>
```

This is a mercifully simple stylesheet for validation. It simply instructs the parser to make sure that the XML output file meets certain criteria. In this case, the XML document must start with a FishList Element, and that element must contain at least one fish record. Inside the fish record should be nested a FishName, FishIDList, FishTypes and FishDescription element. The FishIDList element must contain a FishType element, and the FishTypes element must contain a FishNames element. Also, the FishType element in the FishIDList must have an attribute called ID and that ID is required. For more information on data validation using DTDs, refer to Chapter 2.

In order to force our newly transformed XML output to conform to this DTD, the output XML file must contain the DOCTYPE argument to instruct the XML parser to validate against the bigfish.dtd when the XML document is parsed:

```xml
<!DOCTYPE document SYSTEM 'bigfish.dtd'>
```

In order to pass this value to the XML output file via the XSL stylesheet, we need to add this line to the top of our stylesheet:

```xml
<xsl:output doctype-system="bigfish.dtd"/>
```

When the XSL processor transforms the data it will add the DOCTYPE line automatically to the top of the resulting output XML to force a validation of the transformed data.

### 4.4 Transforming XML to alternate formats

Of course, data has been around much longer than HTML and XML, so there are other formats that you may want to transform data to in order to pass data from one source to another. Let’s look at some of the issues involved in transforming XML to other data formats by reviewing examples of
transformation from our original DXL file to text, delimited text, and EDI formats.

4.4.1 Transforming to a simple text file

We've already done most of the work involved in setting up the environment to transform the same DXL output file we used in the last example for XML output. In this case, we've copied the original Domino agent TransformXMLtoXMLfile and renamed the new copy TransformXMLtoTextFile. In the new agent we've specified a new XSL stylesheet to use for the transformation and a new output file name. Refer to 4.3, "Transforming from XML to XML" on page 212 to get an overview of the environment and the Java agent that is used to generate this example. The new output file name is called c:\temp\textoutput.txt. The only real work is creating a new stylesheet to squirt out text instead of XML. The new stylesheet is called c:/temp/DXLtoFishText.xsl. Here is the stylesheet in it's entirety:

```xml
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text"/>
  
  <xsl:template match="/"
Fish IDs: <xsl:apply-templates select="/document/item[@name='FishID']"/>
Fish Names: <xsl:apply-templates select="/document/item[@name='FishName']"/>
Subcategories: <xsl:apply-templates select="/document/item[@name='Subcategory']"/>
Fish Descriptions: <xsl:apply-templates select="/document/item[@name='FishDescription']"/>
</xsl:template>

<xsl:template match="item[@name='FishID']">
  <xsl:value-of select="."/>
</xsl:template>

<xsl:template match="item[@name='FishName']">
  <xsl:value-of select="."/>
</xsl:template>

<xsl:template match="item[@name='Subcategory']">
  <xsl:value-of select="."/>
</xsl:template>

<xsl:template match="item[@name='FishDescription']">
  <xsl:value-of select="."/>
</xsl:template>

```

Chapter 4. Transforming XML data in Domino 229
In order to make a successful transformation from the DXL document to text, a few things have to be kept in mind. First of all, make sure that your output method is set to text:

```xml
<xsl:output method="text"/>
```

In this case, as with the previous XML to XML example, we wanted to strip away any extra Domino systems fields and document properties and focus on the fish content in the original DXL data. To do this we had to make some select statements:

```xml
<xsl:template match="/">
Fish IDs: <xsl:apply-templates select="/document/item[@name='FishID']"/>
Fish Names: <xsl:apply-templates select="/document/item[@name='FishName']"/>
Subcategories: <xsl:apply-templates select="/document/item[@name='Subcategory']"/>
Fish Descriptions: <xsl:apply-templates select="/document/item[@name='FishDescription']"/>
</xsl:template>
```

In this case we just want the FishID, FishName, Subcategory and FishDescription fields specifically from the root node in the DOM. This meant that all other elements will be ignored. Because the DXL output has written these Domino fields (and most standard output besides document properties) to elements called item, we need to retrieve the name attribute from each element to choose the elements we want. This is done using the `@` symbol followed by the attribute name that we want to return the value of. We also add some text to the beginning of each line, which will be appended before writing the values of the XSL match templates:

```xml
<xsl:template match="item[@name='FishID']">
  <xsl:value-of select="."/>
</xsl:template>
<xsl:template match="item[@name='FishName']">
  <xsl:value-of select="."/>
</xsl:template>
<xsl:template match="item[@name='Subcategory']">
  <xsl:value-of select="."/>
</xsl:template>
<xsl:template match="item[@name='FishDescription']">
  <xsl:value-of select="."/>
</xsl:template>
```
<xsl:template match="item[@name='FishDescription']">
  <xsl:value-of select = "."/>
</xsl:template>

This code simply returns all the values in the original DXL file that have a name attribute that matches the FishID, FishName, Subcategory, and FishDescription fields. This takes us to the end result, which is the output text itself, located in c:\temp\textoutput.txt:

Fish IDs: 640616426436464666467648649650651652653
Fish Names: Shark
Fish Descriptions: SharkSharkSharkSharkSharkSharkSharkSharkSharkSharkSharkSharkSharkShark

4.4.2 Transforming to a delimited text file

In order to add delimited text to this file, a few things had to be done to the previous example. We've copied the TransformXMLtoTextFile to a new agent called TransformXMLtoDelimitedTextFile. In the new agent we've specified a new XSL stylesheet to use for the transformation and a new output file name. Refer to the previous section, transforming XML to XML, to get an overview of the environment and the Java agent that is used to generate this example.

The new output file name is called c:\temp\delimitedtextoutput.txt. The new stylesheet is called c:/temp/DXLtoDelimitedText.xsl. There are only a few differences in the new XSL stylesheet. Here is the stylesheet in it's entirety:

```xml
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text"/>
  <xsl:template match="/">
    Fish IDs: <xsl:apply-templates select = "/document/item[@name='FishID']"/>
    Fish Names: <xsl:apply-templates select = "/document/item[@name='FishName']"/>
    Subcategories: <xsl:apply-templates select = "/document/item[@name='Subcategory']"/>
    Fish Descriptions: <xsl:apply-templates select = "/document/item[@name='FishDescription']"/>
  </xsl:template>
</xsl:stylesheet>
```
Most of the stylesheet is identical to the previous example for generating a text file from the original DXL file. The select templates at the top of the sheet are identical, the difference is in the templates for the various items. All the templates are identical, so we'll just review the first one for the item element with a name attribute of FishID:

```
<xsl:template match="item[@name='FishID']">
  <xsl:for-each select="textlist/text">
    <xsl:value-of select="."/>
    <xsl:if test="position() != last()">, </xsl:if>
  </xsl:for-each>
</xsl:template>
```

In this illustration, the FishID template match code instructs the XSLT DOM processor to match any values that match the pattern /document/item containing an attribute called name and a value of FishID. For each match that is found, we use a for-each instruction to wrap quotations around the returned data. The next line instructs the XSL processor to add a comma to the end of the value that is now wrapped in quotations, but only if the element we're dealing with is not the last element that matches this pattern, as we don't want an extra comma at the end of the delimited list.

Here's the output for this conversion type:


Fish Names:
4.4.3 Transforming to EDI

EDI (Electronic Data Interchange) has been around for over 30 years, and has a loyal following of hundreds of thousands of organizations, which use it for conducting secure business transactions between EDI-compatible organizations. EDI transactions follow a syntax called X12, which provides security and flexibility in a compact and rigidly standardized format. Because of this, transformations between EDI and XML are possible, but standards have to be just as rigid on the XML side to make sure the original transaction message is not lost in the transformation. In order to overcome any roadblocks, the United Nations CEFACT (United Nations body for Trade Facilitation and Electronic Business) has begun to put together some standards for XML and EDI integration, but has not published anything as yet. Detailed information on ebXML can be found at:

http://www.ebxml.org

Along with these standards, other commercial and non-commercial organizations are working on standards that pertain to a single vertical industry, or several industries with similar challenges. Among these are World Wide Web Consortium, the IETF, CommerceNet’s eCo Working Group, OBI, IOTP, RosettaNet, BizTalk, ebXML, and OASIS.

For this example, we’ll show you a simple transformation of our DXL data into something that EDI could consider compatible in terms of format, but not from a syntax point of view. EDI transactions have a very distinct file delimitation, using a ^ to separate different elements in a transaction file. For this example, we’ll build on the previous formatting examples using plain and delimited text, and produce a text file that could be transformed into an EDI transaction with further XML transformation.

We’ve copied the TransformXMLtoDelimitedTextFile to a new agent called TransformXMLtoEDIFormat. In the new agent we’ve specified a new XSL stylesheet to use for the transformation and a new output file name. Please refer to the previous section to get an overview of the environment and the Java agent that is used to generate this example. The new output file name is c:\temp\delimitedEDIoutput.txt. The new stylesheet is called c:/temp/DXLtoEDIFormat.xsl. Here’s the stylesheet in its entirety:

<?xml version="1.0"?>

As with the previous text output examples, the main code for the output is contained in the match templates. Let's have a look at the first match template for an example, as they all are identical:

In this illustration, the FishID template match code instructs the XSLT DOM processor to match any values that match the pattern /item containing an attribute called name whose value is FishID. For each match that is found, we
use a for-each instruction to wrap a ^ character around each end of the returned data. here's the EDI-formatted output:

Chapter 5. Getting XML into Domino

The XML data you need to run your business will more than likely come from a variety of sources. You will need access to data from your vendors, suppliers and customers among others. If the data you need is available in an XML format, you need to know how are you going to bring it into your own systems.

In this chapter, we concentrate on how to access XML data from outside of the Domino environment, how to bring it into Domino and create Domino objects with it, and finally, how we are able to manipulate that data via the standard Domino programming tools.

5.1 Where XML data comes from

There will be a variety of sources that you might have the opportunity to receive data from. Many large companies choose to provide their data in an XML format. Most of the major software vendors have provided native XML access to the data stored in their product lines. These companies include:

- IBM
- Lotus
- Sun
- Microsoft
- SAP
- Oracle
- Many others

What does this have to do with getting into Domino? If these major players have the ability to store data as XML, then applications written for these products will be able to produce XML output. That output may very well be the data you want to bring into your system. As a data integrator you may never have to worry about proprietary data formats again. If one of the people you do business with does not have Domino, you will still be able to work with them easily if they can provide you with the data in an XML format. XML has become the universal format for data transport. As more and more companies translate their existing data to an XML format, that data can then be brought into your Domino system.

5.1.1 XML data transports

The next question to consider is, how is the XML data going to be sent and received.
Sanjeev Varma, a GartnerGroup research director and XML expert, spoke at GartnerGroup’s “AD Summit 2000: Application Development in the New Economy” conference. He predicts that through 2002, no single XML transport protocol standard will be used in more than 5 percent of all new XML applications. But by 2005, only two or three protocol standards will be used to create the protocols expressed in more than 90 percent of all new XML applications. (For more details, see http://www.internetwk.com/story/INW20000913S0005).

Knowing this, we still need to consider the type of communication that will be taking place now and in the future. We can categorize these communications as:

- Server-to-server
- Client-to-server

When we talk about transporting data, we usually talk about the packaging or the protocol that will be used to move the data from point A to B.

As of today there is no standard for transporting XML data between systems. There is, however, a lot of work currently going on at the W3C organization concerning this very issue. You can access several open working drafts at the W3C website.

Transports in use today include:

- XML Remote Procedure Call (XML-RPC)
- Simple Object Access Protocol (SOAP)
- XML over FTP
- XML over HTTP
- XML over SMTP
- XML and message queuing

XML-RPC and SOAP are still relatively new and there is not yet consistent client or server implementations of them.

HTTP is the most commonly used in most open systems.

When looking at the type of communication and the available transports, most server-to-server transactions will take place between servlets over HTTP and client transactions will occur between browser clients and Web servers over HTTP.
5.2 Evaluating data accuracy

When receiving data from an outside source, one of the first questions that comes to mind is whether or not the data is accurate. As we explained in Chapter 2, there are two levels of conformity that can be tested for:

- Well-formedness
- Validity

You will want to validate your incoming data before you process it into your Domino environment.

5.2.1 Validating data with a common DTD

When data is received from an outside source, it can be tested for well-formedness by parsing it with an XML parser. To ensure that the data is valid, you and the person you are exchanging data with would have agreed upon a schema (currently in the form of a DTD). Using this schema you will be able to validate the data with a validating parser to ensure that the data is not only well-formed but valid according the schema as well.

Once the data has been validated it can be brought into your Domino environment without concern for processing errors.

5.3 Places you can store XML data in Domino

When XML data is brought into Domino it will either be processed in memory and rendered to a Notes or browser client or it will be written into document objects on the Domino server for processing later.

The techniques of processing the XML data will be similar for both of these options.

You learned in Chapter 4 about getting data out of Domino. In this chapter we will show you how it got there.

5.3.1 Notes documents

Notes documents are the ultimate XML container. Notes and XML documents share the same design philosophy of separating the data from the formatting of the data. In Domino the data is stored in documents and the presentation is stored in the form design. In XML the data is enclosed in tags that describe the data and the formatting of the data is stored in stylesheets. The important point to note is that with both Domino and XML, the presentation of the data comes together at run-time, allowing the developer to alter the presentation of
the data on the fly without having to alter the data structure or affecting the
data integrity by having to store multiple versions of the same data.

This allows for the creation of a form design like we used in Chapter 4, which
hard codes the tags around the field design.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<Customer>
  <ID>T</ID>
  <Lastname>lastname</Lastname>
  <Firstname>firstname</Firstname>
  <Company>company</Company>
  <Address>address</Address>
  <Zip>zip</Zip>
  <Zone>zone</Zone>
</Customer>
```

*Figure 86. Domino form with hard coded XML tags*

This format most closely resembles an actual XML document. This document
may now be served to a browser client or used as the basis for XML input to
an application.

### 5.3.2 Fields

What can be stored in a document field? Depending upon the field type, plain
text, rich text, and file attachments. You can design a Domino form containing
a single field that is used for storing the text of an XML document. This field
can either be:

- Text
- Rich Text

When storing XML data in a field on a Domino document, it should be stored
as plain text, an attachment, an embedded object, or a MIME entry. For ease
of processing, choose one of these options per field. Do not try to mix raw
XML data with an attachment in the same field.

When storing very large XML documents use a rich text field.
Storing XML data in a Notes field in a document allows for the use of the Domino database as an XML repository. The XML data stored in these fields can then be processed by the pre-defined Java methods that are included in the back-end classes on the Domino server and the Designer client.

Form design for storing XML data is shown in Figure 87.

![Figure 87. Form design for storing XML data in a field](image)

An example of a document created with the form design is shown in Figure 88 on page 242.
In the Domino 5 Designer help database, under “Using XML with Domino,” you will notice that most of the pre-defined Java methods are to process XML data that can be accessed by an instance of the Item, RichTextItem, MIMEEntity, or EmbeddedObject classes. Table 23 on page 243 identifies these methods.
This means that when accepting data into Domino to store it for later processing, it can be saved in an Item or RichTextItem, as a file attachment or EmbeddedObject, or it can be mailed into the database and processed as a MIMEEntry.

### 5.3.3 Pages

Pages are design elements that were introduced in Domino R5. Pages can be used to store static HTML or they can act as the container for an embedded view design.

Pages are design elements they can only be updated by someone with developer access to the database. As a result, they do not lend themselves to dynamic Web content or constant updates.

Since XML data is very dynamic, how can we use the page design in Domino? There are several aspects of working with XML data that would require the use of some static content. Two types of documents that you will use often with XML data are:

- DTDs
- Stylesheets

Both of these types of documents have a one to many relationship with XML documents. We use DTDs for the purpose of validating XML documents, and stylesheets to apply formatting of XML documents or transformation to other documents.

But what does this have to do with getting data into Domino?

When we receive data into the Domino system, the possibility exists that we will have to validate it with a DTD or transform it with a stylesheet.
The next question, then, is what is the benefit of storing these DTDs and stylesheets in page designs? Why not just keep them in the file system instead?

The answer once again lies in the power of using Domino as the container for not only our XML data, but the documents that work with that data.

If the files are stored in the file system, then the advantage of being able to replicate these elements to other Notes databases is lost. With the DTDs and stylesheets being replicated throughout the enterprise, there is no longer a dependency on files stored on a single machine.

5.3.4 File system

Although not the ideal storage location for your XML data, DTDs and stylesheets, it is still possible to use the Domino directory structure to store these files. Static HTML documents on a Domino Web server are stored in the following folder:

install directory >/data/Domino/HTML

This folder is also where you would store your XML, DTD and stylesheet documents.

Figure 89 shows a typical Domino file directory listing.
If you had a single stand-alone Domino Web server that did not replicate the XML application, then storing documents in the file system would be the same as storing them as page design elements. The XML application can access these documents from either location. There can be a performance enhancement to storing the schema documents and stylesheets in the directory structure. Documents in the directory structure are cached in memory upon server startup. If these documents were referenced often, serving them from the file system would be faster then having the Domino server render them from a page design.

5.3.5 Processing documents in memory

In some instances the XML data may not need to be stored at all. Using Domino as a middle tier in a three tier design environment would allow clients to access relational data on a back-end database system through a Notes client or Web browser. Some XML data will simply pass through and be processed in memory.

For example, data from a DB2 table could be retrieved in an XML format and then formatted with an XSLT stylesheet for presentation to a Web browser. This kind of on-demand system would not require the storage of XML data within the Domino server. The stylesheet could be applied from either an internal page design or an actual file stored in the file system.
5.4 What you can do with the XML data you receive

Once your have the XML data, what are you going to do with it? As discussed in the previous section, there are plenty of places to store XML data within the Domino architecture.

This section concentrates on taking the XML data and processing it to create Domino documents in the database.

5.4.1 Processing XML with an agent or servlet

Domino agents and servlets are the primary tools that will be used to process XML data.

5.4.1.1 Agents versus Servlets

Which should you choose to process your XML? That depends on the transaction model that you wanted to implement. Table 24 compares agents and servlets.

Table 24. Agents versus servlets

<table>
<thead>
<tr>
<th>Agents</th>
<th>Servlets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Triggered by schedule, object event or in response to user interaction.</td>
<td>Triggered by url request from Web client or Web server.</td>
</tr>
<tr>
<td>Can be coded using the formula language, lotusscript or java.</td>
<td>Coded in java only.</td>
</tr>
<tr>
<td>Code is stored as an Agent object in the Domino database.</td>
<td>Code is stored in a class file on the Domino Web server.</td>
</tr>
<tr>
<td>Can be initiated from the Notes client as well as the browser client.</td>
<td>Initiated by Web clients or servers.</td>
</tr>
<tr>
<td>Agent are read into memory each time they are run.</td>
<td>Servlets are cached in server memory each time the server starts.</td>
</tr>
</tbody>
</table>

5.4.2 Create single Notes document or series of Notes documents

If you are pulling the data from an XML source, you can write the raw XML content to a field in a Domino document and save it in the database to be processed later. You can also parse the incoming XML into a DOM tree and then use parser methods to “walk the tree” and write node values to Notes fields within a Notes document. If there are multiple node values to process, you can iterate through each creating individual Notes documents out of each one.
5.5 Example: How to add XML into a field on a Domino document

In this section we show the Domino design objects and code that enables the processing of raw XML data into a field in a document.

1. First, we need a form to hold the raw XML data. In the Domino Designer, create a form using the details in Table 25.

Table 25. Field definitions for form design

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Field Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Editable - Text</td>
</tr>
<tr>
<td>DataType</td>
<td>Editable - Text</td>
</tr>
<tr>
<td>Date</td>
<td>Editable - Time/Date</td>
</tr>
<tr>
<td>RawXMLData</td>
<td>Editable - Rich Text</td>
</tr>
</tbody>
</table>

In the Domino Designer client the form would look like Figure 91.

Figure 91. Form for raw XML data.

2. Save the form design.
3. Next, we will create an agent that will pull raw XML data from a URL and populate the XMLData field in the form design.

This is an agent that will read daily exchange rate data from a Web server and write the data into the field in the document. Create a new agent from the Domino Designer client.

Set the agent run-time parameters from the details in Table 26.

<table>
<thead>
<tr>
<th>Agent Parameter</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>CreateSingleDocumentExchangeRate</td>
</tr>
<tr>
<td>When should this agent run?</td>
<td>On Schedule</td>
</tr>
<tr>
<td>Which document(s) should it act on?</td>
<td>All new and modified document since last run</td>
</tr>
<tr>
<td>Run</td>
<td>Java</td>
</tr>
</tbody>
</table>

You can give the agent any name you want. In a production application this agent would be set to run on schedule daily, but can be triggered by any of the other agent triggers as well.

Once created, the agent should look similar to the one shown in Figure 92.

![Figure 92. Agent open in Domino Designer client](image-url)
4. Add the following Java code for the agent:

```java
import lotus.domino.*;
import java.net.*;
import java.io.*;

public class JavaAgent extends AgentBase {

    public void NotesMain() {

        try {
            //Create variables
            Session session = getSession();
            AgentContext ac = session.getAgentContext();
            Database db = ac.getCurrentDatabase();
            Document doc = db.createDocument();
            RichTextItem xml = doc.createRichTextItem("XMLData");
            DateTime dt = session.createDateTime("Today");
            //Set the value of the document fields
            doc.appendItemValue("Form","RawXMLData");
            doc.appendItemValue("Description","Daily Exchange Rates");
            doc.appendItemValue("DataType", "Exchange Rates");
            dt.setNow();
            doc.appendItemValue("Date", dt);
            //Open the URL of the Exchange rate web server and read the text
            into a Buffer
            URL EXRates = new URL("http://fxtop.com/xml/alldev.xml");
            BufferedReader in = new BufferedReader(new InputStreamReader(EXRates.openStream()));
            String inputLine;
            //Loop through every line of the URL and write it out to a Rich
            Text Field.
            while ((inputLine = in.readLine()) != null) {
                xml.appendText(inputLine.toString());
                xml.addNewLine();
            }
            //Save the document and close the reader.
            doc.save(true,true);
            in.close();
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

We will now examine each section of code and explain its purpose.
The following imports the appropriate Java classes and declares the public class that runs the agent:

```java
import lotus.domino.*;
import java.net.*;
import java.io.*;

public class JavaAgent extends AgentBase {
    public void NotesMain() {
        try {
            //Create variables
            Session session = getSession();
            AgentContext ac = session.getAgentContext();
            Database db = ac.getCurrentDatabase();
            Document doc = db.createDocument();
            RichTextItem xml = doc.createRichTextItem("XMLData");
            DateTime dt = session.createDateTime("Today");
            //Set the value of the document fields
            doc.appendItemValue("Form","RawXMLData");
            doc.appendItemValue("Description","Daily Exchange Rates");
            doc.appendItemValue("DataType", "Exchange Rates");
            dt.setNow();
            doc.appendItemValue("Date", dt);

            URL EXRates = new URL("http://fxtop.com/xml/alldev.xml");

            BufferedReader in = new BufferedReader(new
            InputStreamReader(EXRates.openStream()));
            String inputLine;
            //Loop through every line of the URL and write it out to a Rich
            Text Field.
            while ((inputLine = in.readLine()) != null) {
                xml.appendText(inputLine.toString());
                xml.addNewLine();
            }
        }
    }
}
```

Next, we enter a try/catch block where all of the variables are defined and set. All of these are standard Domino objects: Session, Agentcontext, Database, Document and Richtextitem.

We now open the URL to the source of the XML data using the URL class from the java.net package:

```java
URL EXRates = new URL("http://fxtop.com/xml/alldev.xml");
```

Now that we have the URL open for reading, we append each line from the buffer to a Domino rich text field followed by a new line to increase the readability of the text. We then process the entire URL inside the while loop, until all lines from the URL are processed.

```java
BufferedReader in = new BufferedReader(new
InputStreamReader(EXRates.openStream()));
String inputLine;
//Loop through every line of the URL and write it out to a Rich
Text Field.
while ((inputLine = in.readLine()) != null) {
    xml.appendText(inputLine.toString());
    xml.addNewLine();
}
```
Finally, we save the document and close the Java reader.

```java
    //Save the document and close the reader.
    doc.save(true, true);
    in.close();
```

The resulting document looks like the one shown in Figure 93.

![Figure 93. Completed XML document from agent.](image)

Once the raw XML is stored in a field in a Domino document, any of the predefined Java methods can be used to process the XML data. It can be parsed and written into multiple documents, or transformed with an XSLT stylesheet into an alternate representation of the data. The next section explores the predefined Java methods of the back-end Java classes that allow the processing of XML data.
5.5.0.1 Processing XML data stored in Domino.

In the beginning of this chapter we discussed the four options for storing XML data in Domino. The previous example demonstrated how raw XML data can be stored in a Rich Text Field. If the XML data was smaller, it could also be stored in a plain text field. The XML data could also arrive as a file attachment to the document and be accessed through the EmbeddedObject Class. Finally, if the XML data was mailed into the database, it could be processed via the MIMEEntry class.

However the XML data arrives, the methods available to process it are the same. Table 27 describes these methods.

Table 27. Common Java methods to process XML data

<table>
<thead>
<tr>
<th>Method name</th>
<th>Returns an instance of...</th>
</tr>
</thead>
<tbody>
<tr>
<td>inputSource</td>
<td>org.xml.sax.InputSource</td>
</tr>
<tr>
<td>inputStream</td>
<td>java.io.InputStream</td>
</tr>
<tr>
<td>Reader</td>
<td>java.io.Reader</td>
</tr>
<tr>
<td>parseXML</td>
<td>org.w3c.dom.Document</td>
</tr>
<tr>
<td>transformXML</td>
<td>specified XSLTResultTarget</td>
</tr>
</tbody>
</table>

These are the methods that will be used to process XML data stored in the Domino database.

The first thing that needs to be done is to turn the raw XML into a format that can be used for processing. There are two APIs that will allow us to accomplish this, DOM and SAX. The XML4J XML parser from IBM, which is available in the Domino Server and the Domino Designer Client, contains both DOM- and SAX-based parsers. When parsing XML data, either the native classes of the parser or one of the pre-defined methods described above can be used.

In the following example we will use the parseXML method of the RichTextItem class to parse the XML and return an instance of a w3 document in the form of a Document Object Model (DOM) tree.

Using the same form that we used to store the raw XML data into, we will create an agent that parses the XML and returns a DOM tree. Once we have access to the DOM tree, we will print some data out to the JAVA console.

The first thing we need to do is create the agent. Once again, this will be a Java agent to take advantage of the pre-defined methods of the existing object classes.
Create an agent with the parameters shown in Table 28.

<table>
<thead>
<tr>
<th>Agent parameter</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>ParseRawXML</td>
</tr>
<tr>
<td>When should this agent run?</td>
<td>Manually From Agent List</td>
</tr>
<tr>
<td>Which document(s) should it act on?</td>
<td>Run Once(@ Commands may be used)</td>
</tr>
<tr>
<td>Run</td>
<td>Java</td>
</tr>
</tbody>
</table>

This is the complete code for this agent:

```java
import lotus.domino.*;
import org.w3c.dom.*;

public class JavaAgent extends AgentBase {

  public void NotesMain() {
    int i;
    try {
      Session ns = getSession();
      AgentContext ac = ns.getAgentContext();
      lotus.domino.Document doc = ac.getDocumentContext();
      Item rawXML = doc.getFirstItem("XMLDATA");
      org.w3c.dom.Document xDoc = rawXML.parseXML(false);
      Element el = xDoc.getDocumentElement();
      String rootTag = el.getTagName();
      System.out.println("The Root Element is " + rootTag);
      NodeList nl = xDoc.getElementsByTagName(rootTag);
      System.out.println("There is " + nl.getLength() + " node in the Root
Node List");
      Node n = nl.item(0);
      nl = n.getChildNodes();
      System.out.println("The " + rootTag + " Root Tag has " + nl.getLength() + " child nodes");
      for ( i = 0 ; i < 15 ; i++) {
        n=nl.item(i);
        if (n.getNodeType() == n.ELEMENT_NODE) {
          System.out.println("Node Name is:" + n.getNodeName());
          System.out.println("Node Value is:" + n.getNodeValue());
          System.out.println("Node Value is:" + n.getNodeType());
          System.out.println("This node has child nodes:" + n.hasChildNodes());
        }
      }
    }
  }
}
```

Chapter 5. Getting XML into Domino  253
Save the agent with name ProcessRawXML.

Now edit the design of the RawXMLForm form. Include an action in the action bar that calls the agent we just created. The action will be a simple action with the following command:

Run '(ParseRawXMLDOM)' agent

Save the form design and then open the database in the Notes client. Access one of the existing raw XML documents that contains the exchange rate data.

Run the agent code by clicking the action in the action bar. To see the output from the agent, open the Java Debug Console. Choose File -> Tools -> Java Debug Console.

The output will look like Figure 94 on page 255.
Chapter 5. Getting XML into Domino

Now that you have seen the output, let's review the code to see how it was accomplished.

```java
import lotus.domino.*;
import org.w3c.dom.*;

public class JavaAgent extends AgentBase {
    
    public void NotesMain() {
        int i;
        try {
            Session ns = getSession();
            AgentContext ac = ns.getAgentContext();
            lotus.domino.Document doc = ac.getDocumentContext();
            Item rawXML = doc.getFirstItem("XMLDATA");
            org.w3c.dom.Document xDoc = rawXML.parseXML(false);
        }
    }

    // Other methods...
}
```

Figure 94. Agent output to Java debug console
In this code the org.w3c.dom package is imported to allow access to the methods that are used to process the raw XML data as a DOM tree. Another consideration is that both the org.w3c.dom and the lotus.domino packages contain a class called “Document”. We will use the explicit path when instantiating an object reference to these two classes. Through AgentContext we have access to the current document that contains the field that stores the raw XML data. We instantiate an instance of the Notes Item Class and set it equal to the field that contains the rawXML. To create the w3c document, the parseXML method of the Item class is invoked to provide a reference “xDoc” to the XML document represented as a DOM tree.

The DOM tree is represented as a hierarchical tree of nodes, starting with the document itself and then representing each element and all of their descendants until the tree is complete.

The hierarchy of a Generic DOM tree will look like Figure 95.

Figure 95. Hierarchical DOM tree
The document root is always the first node in the tree. The document node only has one child and that will always be the root element of the XML document. From there it is possible to process all nodes in the tree using the methods from the classes of the org.w3c.dom package.

Using our example of the currency data, the DOM tree that is produced would look similar to Figure 96.

Figure 96. Exchange rates DOM tree

ALLCURRENCIES is the root tag.

The child nodes of ALLCURRENCIES will include all text nodes, comments, processing instructions and elements. When returning the data from the DOM tree we are generally just interested in processing the ELEMENT nodes. Therefore we can do a node test based upon the values or constants defined in the Node class and listed in Table 29 on page 258.
<table>
<thead>
<tr>
<th>Constant name</th>
<th>Integer value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELEMENT_NODE</td>
<td>1</td>
</tr>
<tr>
<td>ATTRIBUTE_NODE</td>
<td>2</td>
</tr>
<tr>
<td>TEXT_NODE</td>
<td>3</td>
</tr>
<tr>
<td>CDATA_SECTION_NODE</td>
<td>4</td>
</tr>
<tr>
<td>ENTITY_REFERENCEC_NODE</td>
<td>5</td>
</tr>
<tr>
<td>ENTITY_NODE</td>
<td>6</td>
</tr>
<tr>
<td>PROCESSING_INSTUCTION_NODE</td>
<td>7</td>
</tr>
<tr>
<td>COMMENT_NODE</td>
<td>8</td>
</tr>
<tr>
<td>DOCUMENT_NODE</td>
<td>9</td>
</tr>
<tr>
<td>DOCUMENT_TYPE_NODE</td>
<td>10</td>
</tr>
<tr>
<td>DOCUMENT_FRAGMENT_NODE</td>
<td>11</td>
</tr>
<tr>
<td>NOTATION_NODE</td>
<td>12</td>
</tr>
</tbody>
</table>

The first child node of node type ELEMENT is DATE, the second is “PREVIOUSDATE”, and the rest are all “CURRENCY” nodes.

Following is the remainder of the code.

Remember the document only has one element, the root element. This code returns the root element and its name and prints them to the console.

```java
Element el = xDoc.getDocumentElement();
String rootTag = el.getTagName();
System.out.println("The Root Element is " + rootTag);
```

This code creates a node list from the root element name and prints the number of nodes in the list to the console.

```java
NodeList nl = xDoc.getElementsByTagName(rootTag);
System.out.println("There is "+nl.getLength()+" node in the Root Node List");
```

The next line creates a Node object from the first value in the list and then creates a node list of all of its children.

```java
Node n = nl.item(0);
nl = n.getChildNodes();
```
The rest of the code takes the first 10 nodes and tests them to determine if they are ELEMENT nodes or not. If they are, then their Name, Value, NodeType and whether they have any child nodes are printed to the console. (There is nothing special about using 10; we are just returning enough nodes to see example data.)

```java
for (i = 0; i < 10; i++) {
    n = nl.item(i);
    if (n.getNodeType() == n.ELEMENT_NODE) {
        System.out.println("Node Name is: " + n.getNodeName());
        System.out.println("Node Value is: " + n.getNodeValue());
        System.out.println("Node Value is: " + n.getNodeType());
        System.out.println("This node has child nodes: " + n.hasChildNodes());
    }
}
```

Again, the output from this agent on the Java console would look like Figure 97 on page 260.
Notice that there are only five nodes returned. The other five nodes, those that are not displayed, were not of node type ELEMENT. To see all of the node types that are returned you could remove the `if` test and then all of the nodes would be output.

5.5.0.2 Create multiple Domino documents from raw XML data

We have shown you how to bring raw XML data into Domino and parse it into a DOM tree for processing.

Using that same technique we will examine two ways to parse the raw XML data into multiple documents in the Domino database.
The first example we will look at is processing every node in the DOM tree and creating individual documents for each unique set of XML data. To review, the data we have been working with is exchange rate information. The basic structure of the raw XML looks like the following:

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<ALLCURRENCIES>
    <DATE>22/09/2000</DATE>
    <PREVIOUSDATE>22/09/2000</PREVIOUSDATE>
    <CURRENCY>
        <ISOCODE>ADF</ISOCODE>
        <LIBEN>Andorra (Franc)</LIBEN>
        <LIBFR>Andorre (Franc)</LIBFR>
        <LIBIT>Andorra (Frangi)</LIBIT>
        <LIBDE>Andorra (Franc)</LIBDE>
        <LIBES>Andorra (Franco)</LIBES>
        <LIBNL>Andorra (Franc)</LIBNL>
        <LIBNO>Andorra (Franc)</LIBNO>
        <LIBFI>Andorra (Frangi)</LIBFI>
        <LIBPT>Andorra (Franco)</LIBPT>
        <LIBSE>Andorra (franc)</LIBSE>
        <LIBDK>Andorra (franc)</LIBDK>
        <SHLIB>ADF</SHLIB>
        <NBDEC>2</NBDEC>
        <MODEC>0</MODEC>
        <DAYPRICE>5.7490</DAYPRICE>
        <PREVIOUSDAYPRICE>5.6311</PREVIOUSDAYPRICE>
    </CURRENCY>
    ...remainder of xml document containing currency references.
</ALLCURRENCIES>
```

As we pointed out in the previous example, ALLCURRENCIES is the Root Element. DATE, PREVIOUSDATE and CURRENCY are the first-level children of ALLCURRENCIES. We want to create a separate Domino document that represents each individual currency. This will require that we write an agent that processes the document that contains the raw XML and parses it into a DOM tree. From that DOM tree we will process the nodes to create individual Domino documents for each unique currency.

First we will need to create a form design that will store our currency documents. The form design we will use is shown in Figure 98 on page 262.
Notice that the field names that are being used are the same as the tag names from the XML document. This is not a requirement, but it will allow for easier processing of the data. If the field names and the tag names are the same, then the tag name can be used as a variable to populate a field of that name, without having to manually map all of the tag names to field names.
Next we will write an agent that will process the raw XML data stored in the documents we created in the previous demonstration.

These documents are stored in a view that determines whether or not they have been processed by the agent; this view is shown in Figure 99.

As this view shows, the outcome of running our agent to create a single Domino document from the Currency XML data and store it in a single field is one unprocessed document in the view.

Now we need to write an agent that will process this document in the view and create individual documents.

The agent will have the parameters shown in Table 30.

<table>
<thead>
<tr>
<th>Agent parameter</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>ParseExchangeRateDocs</td>
</tr>
<tr>
<td>When should this agent run?</td>
<td>Manually From Agent List</td>
</tr>
<tr>
<td>Which document(s) should it act on?</td>
<td>Run Once(@ Commands may be used)</td>
</tr>
<tr>
<td>Run</td>
<td>Java</td>
</tr>
</tbody>
</table>

Figure 99. Raw XML data view
Following is the complete Java code that will parse the raw XML and produce the individual documents. There are four methods that are used:

1. NotesMain, which calls processTree
2. processTree, which calls processNode
3. processNode, which calls processChildNodes
4. processChildNodes, which is called recursively

The method calls are in bold.

The complete code follows:

```java
import lotus.domino.*;
import org.w3c.dom.*;
import java.util.*;

public class ParseExRates extends AgentBase {

    String rateDate ;
    String prevRateDate;

    public void NotesMain() {
        try {
            Session ns = getSession();
            AgentContext ac = ns.getAgentContext();
            Database db = ac.getCurrentDatabase();
            View nv = db.getView("RawXMLdataView");
            Vector v = new Vector();
            v.addElement("Exchange Rates");
            v.addElement("Unprocessed");
            DocumentCollection dc = nv.getAllDocumentsByKey(v,true);
            if (dc.getCount() >= 1){
                lotus.domino.Document doc = dc.getFirstDocument();
                while (doc != null) {
                    Item it = doc.getFirstItem("XMLData");
                    org.w3c.dom.Document xDoc = it.parseXML(false);
                    Element el = xDoc.getDocumentElement();
                    processTree(el , xDoc);
                    doc.replaceItemValue("Status","Processed");
                    doc.save(true,true);
                    doc = dc.getNextDocument(doc);
                }  //end while
            } // end if
        } catch(Exception e) {
            e.printStackTrace();
        } catch(Exception e) {
            e.printStackTrace();
    }
```

264 XML Powered by Domino
public void processTree(Element e, org.w3c.dom.Document x) throws Exception {
    NodeList nl = x.getElementsByTagName(e.getTagName());
    Node n = nl.item(0);
    nl = n.getChildNodes();
    for (int i = 0 ; i < nl.getLength() ; i++) {
        n = nl.item(i);
        if (n.getNodeType() == n.ELEMENT_NODE) {
            processNode(n);
        } //end if
    } //end for
} //end processTree()

public void processNode(Node n) throws Exception {
    Session ns = getSession();
    AgentContext ac = ns.getAgentContext();
    Database db = ac.getCurrentDatabase();
    if (n.getNodeType() == n.ELEMENT_NODE) {
        if ((n.getNodeName()).equals("DATE")) {
            n = n.getFirstChild();
            rateDate = n.getNodeValue();
        } else if ((n.getNodeName()).equals("PREVIOUSDATE")) {
            n = n.getFirstChild();
            prevRateDate = n.getNodeValue();
        } else if (n.hasChildNodes()) {
            processChildNodes(n , db) ;
        } // end if-elseif
    } //end if
} //end processNode()

public void processChildNodes(Node n, Database db) throws Exception {
    lotus.domino.Document rDoc = db.createDocument();
    rDoc.appendItemValue("Form", "IEXRates");
    rDoc.appendItemValue("Date" , rateDate);
    rDoc.appendItemValue("PreviousDate" , prevRateDate);
    NodeList nl = n.getChildNodes();
    for (int y = 0 ; y < nl.getLength() ; y++) {
        n = nl.item(y);
        if (n.getNodeType() == n.ELEMENT_NODE) {
            Node cn = n.getFirstChild();
            rDoc.appendItemValue(n.getNodeName(),
            cn.getNodeValue());
        } //end if
    } //end for
} //end processChildNodes()
The NotesMain method sets up the objects that are needed to open a Notes session, the current database, and the raw XML view. Then all of the exchange rate documents with a status of “Unprocessed” are put in a document collection to be processed. If the document collection has at least one document in it, then we open the document and access the field with the raw XML data. The data is then parsed into a DOM tree object that is passed to the parseTree method. We also set a variable to the document element, in this case ALLCURRENCIES. This is passed to the processTree method as well.

The processTree method takes the DOM tree and the document element that is passed into it and creates a node list nl. In the first line of code, nl only contains one node and that is the ALLCURRENCIES node. The second line of code creates a node variable n and sets it equal to the ALLCURRENCIES node. The third line of code overwrites the current nl variable with a node list that contains all of ALLCURRENCIES children. These child nodes contain all node types, ELEMENT, TEXT, and so forth. The next line of code opens up a for loop to process every child node of ALLCURRENCIES. We only want to process the ELEMENT nodes, so each node is tested for nodeType. If the node type is equal to ELEMENT_NODE, then the node is passed into the processNode method.

```java
public void processTree(Element e, org.w3c.dom.Document x) throws Exception {
    NodeList nl = x.getElementsByTagName(e.getTagName());
    Node n = nl.item(0);
    nl = n.getChildNodes();
    for (int i = 0; i < nl.getLength(); i++) {
        n = nl.item(i);
        if (n.getNodeType() == n.ELEMENT_NODE) {
            processNode(n);
        } // end if
    } // end for
} // end processTree()
```

In the processNode method, each node is tested to see if it is either the DATE node or the PREVIOUSDATE node. The value of both of these nodes are
written to variables that can be used when we create the individual documents.

```java
if (n.getNodeType() == n.ELEMENT_NODE) {
    if ((n.getNodeName()).equals("DATE")) {
        n = n.getFirstChild();
        rateDate = n.getNodeValue();
    }
    else if ((n.getNodeName()).equals("PREVIOUSDATE")) {
        n = n.getFirstChild();
        prevRateDate = n.getNodeValue();
    }
}
```

All other nodes that contain children (in this case each CURRENCY node) are passed to the processChildNodes method.

```java
else if (n.hasChildNodes()) {
    processChildNodes(n, db);
} // end if-elseif
} // end if
```  

When the node is passed into processChildNodes, we create a node list out of all of its children. This node list returns all of the child nodes of CURRENCY. A for loop is called to process every node in the list. Once again, this includes nodes of type ELEMENT, TEXT, etc. We are only interested in processing the ELEMENT nodes, so if the node is an ELEMENT node we create a node `cn` equal to its child node. At this point `n` is equal to the current element node (ISOCODE, LIBEN, LIBFR, etc.) and `cn` is equal to its child node.

```java
NodeList nl = n.getChildNodes();
for (int y = 0; y < nl.getLength(); y++) {
    n = nl.item(y);
    if (n.getNodeType() == n.ELEMENT_NODE) {
        Node cn = n.getFirstChild();
        rDoc.appendItemValue(n.getNodeName(), cn.getNodeValue());
    } // end if
} //end for
```

This is the line of code that writes the XML data to the proper Domino fields:

```java
rDoc.appendItemValue(n.getNodeName(), cn.getNodeValue());
```

The node name of the current element is retrieved with the `n.getNodeName()` method. This name is the one that matches the field names we used when we designed the form to store all of the currency information. The next argument, `cn.getNodeValue()` is called on the element's child node. At first glance this...
seems to be incorrect. We should return the Node Name and the Node Value of the ELEMENT node, right? The answer is No! The reason for this is the way that the DOM tree is constructed by the DOM parser. When the parser parses the raw XML and creates the DOM tree, all elements that contain data have a single child of Node of Type Text that contains the value of the element. Figure 100 shows the node types and their values.

![Figure 100. Text child node of an Element node](image)

Once this data is written to the document for every child node in the node list the document is saved and the next node is processed.

```java
rDoc.save(true,true);
} // end processChildNodes
```

### 5.5.0.3 Create Domino documents from a subset of XML data

In the previous example we processed every node in the DOM tree. It is also possible to only process a subset of the DOM tree data. In the next example we will create multiple Domino documents from an XML stream, but only write a portion of the data to the document. Instead of writing the raw XML data to a document first, we will process it in memory. We also want to perform some calculations on the data prior to saving the new documents.

To start, we will need a different form design to hold the data. Figure 101 on page 269 shows the Converted Exchange Rate form design.
We will use this form to create the converted exchange rate documents from the raw XML data.

Next we will need to create a new agent that will create these documents. The new agent will have the parameters shown in Table 31.

Table 31. CreateExRateDocs agent parameters

<table>
<thead>
<tr>
<th>Agent parameter</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>CreateExRateDocs</td>
</tr>
<tr>
<td>When should this agent run?</td>
<td>OnSchedule Daily</td>
</tr>
<tr>
<td>Which document(s) should it act on?</td>
<td>All New and Modified Documents Since Last Run</td>
</tr>
<tr>
<td>Run</td>
<td>Java</td>
</tr>
</tbody>
</table>

Following is the complete code listing for this Java agent:

```java
import lotus.domino.*;
import org.xml.sax.Parser;
import org.xml.sax.helpers.ParserFactory;
import com.ibm.xml.parsers.NonValidatingDOMParser;
import org.w3c.dom.*;
import java.io.*;

public class JavaAgent extends AgentBase {
    Double crate;
    Double DP;
```
public void NotesMain() {

    try {
        String parserClass="com.ibm.xml.parsers.NonValidatingDOMParser" ;
        String xmlfile="http://fxtop.com/xml/alldev.xml" ;
        NodeList nl, inl, dnl ;
        Node n, nn, dn ;
        int i, y ;
        String srate, orate ;

        Session session = getSession();
        AgentContext ac = session.getAgentContext();
        Database db = ac.getCurrentDatabase();

        //Create a dom based parser and parse XML file
        Parser parser = ParserFactory.makeParser(parserClass);
        parser.parse(xmlfile);

        //Create a w3.org document object
        org.w3c.dom.Document doc=((NonValidatingDOMParser)parser).getDocument();

        //find the US currency Rate and convert it.
        nl = doc.getElementsByTagName("ISOCODE");
        for (i=0 ; i < nl.getLength() ; i++){
            n = nl.item(i);
            n = n.getFirstChild();
            if ((n.getNodeValue()).equals("USD")){
                n = n.getParentNode();
                n = n.getParentNode();
                n = n.getChildNodes();
                n=nl.item(0);
                while (n != null) {
                    if ((n.getNodeName()).equals("DAYPRICE")){
                        n = n.getFirstChild();
                        srate = n.getNodeValue();
                        orate = Double.valueOf(srate);
                    } // end if
                    n=n.getNextSibling();
                } // end while
            } //end if
        } // end for

        //Create a node list of only currency nodes.
        nl = doc.getElementsByTagName("CURRENCY");

        //Process all nodes in the node list.
        for (i=0 ; i < nl.getLength() ; i++){
            lotus.domino.Document ndoc = db.createDocument();
        } // end for
    } //end try
} //end NotesMain()
dnl = doc.getElementsByTagName("DATE");
dn = dnl.item(0);
dn = dn.getFirstChild();
ndoc.replaceItemValue("RateDay" , dn.getNodeValue());

ndoc.replaceItemValue("CRATE" , crate);
n = nl.item(i);
inl = n.getChildNodes();

for (y=0 ; y<inl.getLength() ; y++){
n= inl.item(y);
    if (((nn.getNodeName()).equals("LIBEN")) {  
        nn = nn.getFirstChild();
        ndoc.replaceItemValue("LIBEN" , nn.getNodeValue());
    }  
    if (((nn.getNodeName()).equals("ISOCODE")) {  
        nn = nn.getFirstChild();
        ndoc.replaceItemValue("ISOCODE" , nn.getNodeValue());
    }  
    if (((nn.getNodeName()).equals("DAYPRICE")) {  
        nn = nn.getFirstChild();
        if (Character.isDigit(new StringBuffer(nn.getNodeValue()).charAt(0)))
            DP = Double.valueOf(nn.getNodeValue());
        else
            DP = Double.valueOf("0.0");
        Double CDP = new Double(((DP.doubleValue())/(crate.doubleValue())));
        ndoc.replaceItemValue("DAYPRICE" ,CDP) ;  
    }  
    if (((nn.getNodeName()).equals("PREVIOUSDAYPRICE")) {  
        nn = nn.getFirstChild();
        Double DDP = Double.valueOf(nn.getNodeValue());
        if (DDP.isNaN())
            DDP = Double.valueOf("0.0");
        Double CDDP = new Double(((DDP.doubleValue())/(crate.doubleValue())));
        ndoc.replaceItemValue("PREVIOUSDAYPRICE" , CDDP) ;
    }  
    if (((nn.getNodeName()).equals("SHLIB")) {  
        nn = nn.getFirstChild();
        ndoc.replaceItemValue("SHLIB" , nn.getNodeValue());
    }  
    ndoc.replaceItemValue("Form" , "EXFRM");
    ndoc.computeWithForm(false, false);
    ndoc.save();} //end for y block
In the first few lines of code we import the classes needed to complete our agent. Notice that we are importing classes from org.xml.sax and org.ibm.xml. Both of these packages are part of the IBM XML4J parser that comes with Domino server and Domino Designer client starting with release 5.03.

```java
import lotus.domino.*;
import org.xml.sax.Parser;
import org.xml.sax.helpers.ParserFactory;
import com.ibm.xml.parsers.NonValidatingDOMParser;
import org.w3c.dom.*;
import java.io.*;
```

```java
public class JavaAgent extends AgentBase {
    Double crate;
    Double DP;

    The next few lines of code will instantiate the variables we use in the rest of the agent. Notice the two lines that are bold. The first one is the type of parser that we will use when we create our parser. Storing this value to a string makes it easier to reference an alternate parser.

```java
try {
    String parserClass="com.ibm.xml.parsers.NonValidatingDOMParser" ;
    String xmlfile="http://fxtop.com/xml/alldev.xml" ;
    NodeList nl,inl,dnl ;
    Node n, mn, dn ;
    int i , y ;
    String srate, orate ;
    Session session = getSession();
    AgentContext ac = session.getAgentContext();
    Database db = ac.getCurrentDatabase();
```

The next block of code is where the parser we use to parse the XML data is created. The first bold line of code uses the ParserFactory class to create the non-validating DOM parser. We read the parserClass string from above, so this instantiates a NonValidatingDOMParser. Alternately, we could have selected DOMParser, SAXParser, or ValidatingSAXParser. Next we call the parse method on the referenced XML data. The last line of code creates a W3C document or DOM tree.
This section of code is where we look up the current value of the dollar as a percentage of 1 euro. In the other examples we started at the root of the document and processed every node. But if you know the name of the node that contains the data you are seeking you can create a node tree of just those nodes. The first line of bold code creates a node list that consists solely of ISOCODE elements. This allows us to process this node list without having to test each node or traverse the entire tree. This is one of the major benefits of parsing XML via a DOM interface. As before, we open a for loop for the length of the node list and then test each child node to see if it equals “USD.” When the ISOCODE element with the USD value is found, we need a way to return the current rate. This value is stored in the DAYPRICE element. So the next line of code walks back up the node tree to the parent of the ISOCODE that had the correct value. Once we have a handle to this ELEMENT node we can process all of its child nodes looking for the one we want, DAYPRICE. When this one is found, we read the value out of its child node and convert it to a double and store it in the variable crate.

Now that we have the US dollar value we need, we can create the rest of the documents. The first line of code creates a node list of every CURRENCY element. This is so the entire DOM tree does not have to be processed. Next,
we enter a for loop that will process every CURRENCY node. We then create the fields that appear in every document, Form, RateDay and CRATE. Last, we populate a node list with all of the children of CURRENCY so that we can pull the other values we need.

```java
nl = doc.getElementsByTagName("CURRENCY");
for (i=0 ; i < nl.getLength() ; i++){
    lotus.domino.Document ndoc = db.createDocument();
    ndoc.replaceItemValue("Form" , "EXFRM");
    ndoc.replaceItemValue("RateDay" , docDate);
    ndoc.replaceItemValue("CRATE" , crate);
    n = nl.item(i);
    inl = n.getChildNodes();
    for (y=0 ; y<inl.getLength() ; y++){
        nn=inl.item(y);
        if ((nn.getNodeName()).equals("LIBEN")) {
            nn = nn.getFirstChild();
            ndoc.replaceItemValue("LIBEN" , nn.getNodeValue());
        }
        if ((nn.getNodeName()).equals("ISOCODE")) {
            nn = nn.getFirstChild();
            ndoc.replaceItemValue("ISOCODE" , nn.getNodeValue());
        }
        if ((nn.getNodeName()).equals("DAYPRICE")) {
            nn = nn.getFirstChild();
            if (Character.isDigit(new StringBuffer(nn.getNodeValue()).charAt(0))) {
                DP = Double.valueOf(nn.getNodeValue());
            } else {DP = Double.valueOf("0.0");}
            Double CDP = new Double(((DP.doubleValue())/(crate.doubleValue())));
            ndoc.replaceItemValue("DAYPRICE" ,CDP) ;
        }
        if ((nn.getNodeName()).equals("PREVIOUSDAYPRICE")) {
            nn = nn.getFirstChild();
            Double DDP = Double.valueOf(nn.getNodeValue());
            if (DDP isNaN()) {DDP = Double.valueOf("0.0");}
            Double CDDP = new Double(((DDP.doubleValue())/(crate.doubleValue())));
            ndoc.replaceItemValue("PREVIOUSDAYPRICE" , CDDP) ;
        }
        if ((nn.getNodeName()).equals("SHLIB")) {
```
5.6 Using the Lotus XML Toolkit

The last way of getting XML into Domino is to use the Lotus XML Toolkit, which is available from http://www.lotus.com/xml. Among other things, the toolkit provides C++ classes that you can use to build C++ applications. One of the major benefits of using the toolkit is that it enables you to update fields in a Notes document, create new documents or even create an entire new database with forms and views, and populate it with documents. For more information on the toolkit refer to 3.9, “Working with the Lotus XML Toolkit” on page 135.

5.7 Summary

In this chapter we looked at different ways to read XML data into your Domino environment. Table 32 summarizes the options we covered.

Table 32. Summary

<table>
<thead>
<tr>
<th>Read XML data from...</th>
<th>Save data in Domino at...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web URL</td>
<td>Field in a single Domino document</td>
</tr>
<tr>
<td>Field stored in Domino document</td>
<td>Multiple Domino documents</td>
</tr>
<tr>
<td>External XML source</td>
<td>Multiple Domino documents</td>
</tr>
</tbody>
</table>
Appendix A. Using the additional material

This redbook also contains additional Web material. See below for instructions on downloading each type of material.

A.1 Locating the additional material on the Internet

The Web material associated with this redbook is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser to:

ftp://www.redbooks.ibm.com/redbooks/SG246207

Alternatively, you can go to the IBM Redbooks Web site at:

ibm.com/redbooks

Select the Additional materials and open the directory that corresponds with the redbook form number.

A.2 Using the Web material

The additional Web material that accompanies this redbook includes the following:

<table>
<thead>
<tr>
<th>File name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sg246207.zip</td>
<td>Zipped Code Samples</td>
</tr>
</tbody>
</table>

A.2.1 System requirements for downloading the Web material

The following system configuration is recommended for downloading the additional Web material.

Lotus Notes and Domino R5.03+
Microsoft Internet Explorer 5.5
Lotus XSL Toolkit
Sun Java JDK

A.2.2 How to use the Web material

Create a subdirectory (folder) on your workstation and copy the contents of the Web material into this folder.
Appendix B. Special notices

This publication is intended to help managers and developers to understand how to use Lotus Domino R5.05 and XML together. The information in this publication is not intended as the specification of any programming interfaces that are provided by Lotus Domino or the XML language. See the PUBLICATIONS section of the IBM Programming Announcement for Lotus Domino for more information about what publications are considered to be product documentation.

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program or service.

Information in this book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Corporation, Dept. 600A, Mail Drop 1329, Somers, NY 10589 USA.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers
attempting to adapt these techniques to their own environments do so at their own risk.

Any pointers in this publication to external Web sites are provided for convenience only and do not in any manner serve as an endorsement of these Web sites.

The following terms are trademarks of the International Business Machines Corporation in the United States and/or other countries:

| IBM © | VisualAge |
| AS/400 | WebSphere |
| AT | XT |
| CT | 400 |
| Current | Lotus |
| DB2 | Lotus Notes |
| Netfinity | Domino |
| RS/6000 | Notes |
| SP | Redbooks |
| System/390 | Redbooks Logo |

The following terms are trademarks of other companies:

Tivoli, Manage. Anything. Anywhere., The Power To Manage., Anything. Anywhere., TME, NetView, Cross-Site, Tivoli Ready, Tivoli Certified, Planet Tivoli, and Tivoli Enterprise are trademarks or registered trademarks of Tivoli Systems Inc., an IBM company, in the United States, other countries, or both. In Denmark, Tivoli is a trademark licensed from Kjøbenhavns Sommer - Tivoli A/S.

C-bus is a trademark of Corollary, Inc. in the United States and/or other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and/or other countries.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States and/or other countries.

PC Direct is a trademark of Ziff Communications Company in the United States and/or other countries and is used by IBM Corporation under license.

ActionMedia, LANDesk, MMX, Pentium and ProShare are trademarks of Intel Corporation in the United States and/or other countries.
UNIX is a registered trademark in the United States and other countries licensed exclusively through The Open Group.

SET, SET Secure Electronic Transaction, and the SET Logo are trademarks owned by SET Secure Electronic Transaction LLC.

Other company, product, and service names may be trademarks or service marks of others.
Appendix C. Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

C.1 IBM Redbooks

For information on ordering these publications see “How to get IBM Redbooks” on page 287.

- *The XML Files: Using XML and XSL with IBM WebSphere V3.0*, SG24-5479
- *Using VisualAge for Java to Develop Domino Applications*, SG24-5424
- *Connecting Domino to the Enterprise Using Java*, SG24-5425
- *Domino and WebSphere Together*, SG24-5955

C.2 IBM Redbooks collections

Redbooks are also available on the following CD-ROMs. Click the CD-ROMs button at [ibm.com/redbooks](http://ibm.com/redbooks) for information about all the CD-ROMs offered, updates and formats.

<table>
<thead>
<tr>
<th>CD-ROM Title</th>
<th>Collection Kit Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM System/390 Redbooks Collection</td>
<td>SK2T-2177</td>
</tr>
<tr>
<td>IBM Networking Redbooks Collection</td>
<td>SK2T-6022</td>
</tr>
<tr>
<td>IBM Transaction Processing and Data Management Redbooks Collection</td>
<td>SK2T-8038</td>
</tr>
<tr>
<td>IBM Lotus Redbooks Collection</td>
<td>SK2T-8039</td>
</tr>
<tr>
<td>Tivoli Redbooks Collection</td>
<td>SK2T-8044</td>
</tr>
<tr>
<td>IBM AS/400 Redbooks Collection</td>
<td>SK2T-2849</td>
</tr>
<tr>
<td>IBM Netfinity Hardware and Software Redbooks Collection</td>
<td>SK2T-8046</td>
</tr>
<tr>
<td>IBM RS/6000 Redbooks Collection</td>
<td>SK2T-8043</td>
</tr>
<tr>
<td>IBM Application Development Redbooks Collection</td>
<td>SK2T-8037</td>
</tr>
<tr>
<td>IBM Enterprise Storage and Systems Management Solutions</td>
<td>SK3T-3694</td>
</tr>
</tbody>
</table>
C.3 Other resources

These publications are also relevant as further information sources:

- **Notes.Net Iris Sandbox**, available at:
  http://www.notes.net/sandbox.nsf

- **World Wide Web Consortium**, see
  http://www.w3.org/

- **Java and XML (O'ReillyJava Tools)**, B. McLaughlin, M. Loukides, ISBN: 0596000162

- **Annotated XML Specification**, Tim Bray, available at:
  http://www.xml.com/axml/axml.html


C.4 Referenced Web sites

These Web sites are also relevant as further information sources:

- http://www.w3.org/TR/REC-xml
  Extensible Markup Language (XML) 1.0 (Second Edition) W3C Recommendation 6 October 2000

- http://www.w3.org/TR/xhtml1/
  XHTML™ 1.0: The Extensible HyperText Markup Language

- http://www.w3.org/TR/REC-xml-names
  Namespaces in XML

- http://www.w3.org/TR/SOAP
  Simple Object Access Protocol (SOAP) 1.1

- http://www.w3.org/TR/xpath
  XML Path Language (XPath)

- http://www.w3.org/TR/xlink
  XML Linking Language (XLink) Version 1.0

- http://www.w3.org/TR/WD-xsl
  Extensible Stylesheet Language (XSL)

- http://www.lotus.com/xml
  Lotus XML homepage

- http://www.lotus.com/dxl
  Domino XML homepage
Appendix C. Related publications

- http://www.alphaworks.ibm.com
  IBM Alphaworks
  IBM developers page
- http://xml.apache.org/
How to get IBM Redbooks

This section explains how both customers and IBM employees can find out about IBM Redbooks, redpieces, and CD-ROMs. A form for ordering books and CD-ROMs by fax or e-mail is also provided.

- **Redbooks Web Site** [ibm.com/redbooks]
  
  Search for, view, download, or order hardcopy/CD-ROM Redbooks from the Redbooks Web site. Also read redpieces and download additional materials (code samples or diskette/CD-ROM images) from this Redbooks site.
  
  Redpieces are Redbooks in progress; not all Redbooks become redpieces and sometimes just a few chapters will be published this way. The intent is to get the information out much quicker than the formal publishing process allows.

- **E-mail Orders**
  
  Send orders by e-mail including information from the IBM Redbooks fax order form to:
  
  **e-mail address**
  
  - In United States or Canada: pubscan@us.ibm.com

- **Telephone Orders**
  
  - United States (toll free): 1-800-879-2755
  - Canada (toll free): 1-800-IBM-4YOU

- **Fax Orders**
  
  - United States (toll free): 1-800-445-9269
  - Canada: 1-403-267-4455

This information was current at the time of publication, but is continually subject to change. The latest information may be found at the Redbooks Web site.

---

**IBM Intranet for Employees**

IBM Redbooks fax order form

Please send me the following:

<table>
<thead>
<tr>
<th>Title</th>
<th>Order Number</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

First name

Last name

Company

Address

City

Postal code

Country

Telephone number

Telefax number

VAT number

- Invoice to customer number

- Credit card number

Credit card expiration date

Card issued to

Signature

We accept American Express, Diners, Eurocard, Master Card, and Visa. Payment by credit card not available in all countries. Signature mandatory for credit card payment.
IBM Redbooks review

Your feedback is valued by the Redbook authors. In particular we are interested in situations where a Redbook "made the difference" in a task or problem you encountered. Using one of the following methods, please review the Redbook, addressing value, subject matter, structure, depth and quality as appropriate.

- Use the online Contact us review redbook form found at ibm.com/redbooks
- Fax this form to: USA International Access Code + 1 914 432 8264
- Send your comments in an Internet note to redbook@us.ibm.com

<table>
<thead>
<tr>
<th>Document Number</th>
<th>SG24-6207-00</th>
</tr>
</thead>
<tbody>
<tr>
<td>Redbook Title</td>
<td>XML Powered by Domino How to use XML with Lotus Domino</td>
</tr>
</tbody>
</table>

Review

What other subjects would you like to see IBM Redbooks address?

Please rate your overall satisfaction: O Very Good O Good O Average O Poor

Please identify yourself as belonging to one of the following groups:

- O Customer
- O Business Partner
- O IBM, Lotus or Tivoli Employee
- O Solution Developer
- O None of the above

Your email address:
The data you provide here may be used to provide you with information from IBM or our business partners about our products, services or activities.

O Please do not use the information collected here for future marketing or promotional contacts or other communications beyond the scope of this transaction.

Questions about IBM’s privacy policy?
The following link explains how we protect your personal information.

ibm.com/privacy/yourprivacy/

© Copyright IBM Corp. 2000
XML, the extensible markup language, is the emerging standard for exchanging information between disparate systems. This simple yet powerful technology has many uses in Notes and Domino business applications. This redbook will help you understand XML and how to use it on a Domino platform.

In this IBM Redbook, we start with an introduction to XML, and describe its history, syntax, and related tools. We then describe how you can get information out of Domino in an XML format using tools you are already familiar with, such as views, forms, pages, and agents. Once XML data is extracted from Domino, we show how it can be transformed into other formats, like HTML, text, or other XML formats. Finally, we describe how to get data that is already in an XML format into Domino using agents and the Lotus XML toolkit.