CHAPTER 9

XML Databases

We have been using XML documents throughout this book. Considering our brokerage house, imagine the number of XML documents that the brokerage would create and maintain over a period of time. These documents need to be organized properly so that they can be located on demand. Also, these documents may need to be queried to locate those containing a specified search string. Storing these documents in an XML database can greatly enhance your ability to effectively carry out these tasks and more.

An XML database stores the information in a hierarchical structure that enables users to quickly search by using keyword indexes. It stores the data (that is, the information) within XML format itself, foregoing the need to transform the data when performing inserts or retrievals. Because the documents do not need any transformation, storing documents that have complicated structures is easy. Compare this to storing data in a conventional relational database, where the data must often undergo complicated transformations to meet the storage requirements defined by that technology.

However, you must be wondering how you interact with the database. To query XML databases, we use XPath specifications. As you saw in earlier chapters, XPath provides a powerful syntax for locating nodes in an XML document. Thus, we use XPath expressions to locate documents containing the desired elements and attributes. You may also want to update documents stored in the database without extracting them from storage. We use the XML:DB XUpdate language for writing such update queries.

You will learn all these techniques of database creation, data retrieval, and data updates in this chapter. You will use command-line tools for creating, querying, and updating XML databases. You will also learn about the administration tasks for the database and managing the database through programmatic control. The XML:DB project provides the API for managing the XML databases. You will learn this API in this chapter.

Introducing Apache Xindice

The Apache Xindice (pronounced zeen-dee-chay) project, initially known as dbXML Core, defines the structure and programming API for XML databases. The dbXML source code was donated to the Apache Software Foundation in 2001. You will use Apache Xindice in this chapter to learn about XML databases.

Apache Xindice is a native XML database designed to store XML documents. Using Xindice, you store XML documents in their native format. Thus, there is no need to transform these documents into any other format. Contrast this to the data storage in a typical Indexed Sequential Access Method (ISAM) database or Relational Database Management System

(RDBMS). The database can be queried by using command-line tools or through programmatic control. Similarly, the data in XML documents stored in the database can be modified without the need for retrieving them.

Understanding the XML Database Structure

Xindice is structured in a hierarchical fashion, meaning there are no tables, rows, columns, nor relations. You can think of the database as consisting of a series of XML documents organized in a tree structure, much like you might organize files and folders in a directory structure on your hard drive. In fact, just like a directory structure, the database has a root folder. Under the root you create various subfolders in which to add your XML documents. Figure 9-1 shows a typical database structure.

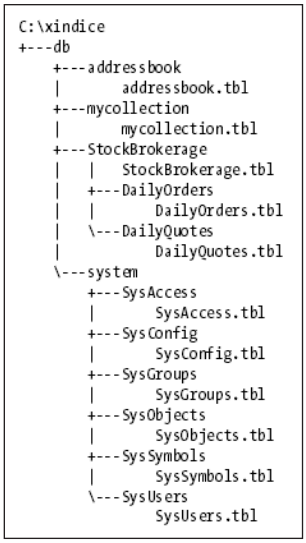

Figure 9-1. *Xindice database structure*

In Figure 9-1, `db` is the root folder of the database. Under this folder are four collections called `addressbook`, `mycollection`, `StockBrokerage`, and `system`. Under `StockBrokerage` are two subcollections called `DailyOrders` and `DailyQuotes`.

Each collection can store any number of XML document instances. Each collection can have additional subcollections. After a database is built, you can query these document instances by using XPath queries. You can also use XUpdate to update these documents without first retrieving them.

Installing Xindice

Before you try creating a database, you must install the necessary software. In this section, you'll download Xindice and then start the Xindice server.

Downloading the Software

You can download Xindice binaries from the Apache site:

`http://xml.apache.org/xindice/download.cgi`

The stable version as of this writing for the Windows platform is `xml-xindice-1.0.zip`. Download this file and unzip it to your local drive. Next perform the following tasks:

1. Set the `XINDICE_HOME` environment variable to `<installation folder>`.
2. Add the several JAR files from `%XINDICE_HOME%\java\lib` to your classpath.
3. Add the path `%XINDICE_HOME%\bin` to your environment `PATH`.

After you complete these steps, you are ready to test your installation.

Starting the Xindice Server

After installing the software, you need to start the Xindice database server. You do so by running the `startup` command in your Xindice installation folder. When you run this script, you will see screen output similar to the following:

```
C:\xindice>startup
java -classpath ".;C:\xindice\java\lib\xindice.jar;C:\soap-2_3_1\lib\soap.jar;C:
\Sun\AppServer\lib\activation.jar;C:\Sun\AppServer\lib\mail.jar;c:\ApacheAnt162\
config;C:\xindice\java\lib\ant-1.4.1.jar;C:\xindice\java\lib\examples.jar;C:\xin
dice\java\lib\infozone-tools.jar;C:\xindice\java\lib\openorb-1.2.0.jar;C:\xindic
e\java\lib\openorb_tools-1.2.0.jar;C:\xindice\java\lib\xalan-2.0.1.jar;C:\xindic
e\java\lib\xerces-1.4.3.jar;C:\xindice\java\lib\xindice.jar;C:\xindice\java\lib\
xml-apis-1.0.jar;C:\xindice\java\lib\xmldb-sdk.jar;C:\xindice\java\lib\xmldb-xup
date.jar;C:\xindice\java\lib\xmldb.jar;C:\Java\jdk1.5.0\lib\tools.jar" -noverify
org.apache.xindice.core.server.Xindice C:\xindice\config\system.xml

Xindice 1.0 (Birthday)

Database: 'db' initializing
Script: 'GET' added to script storage
Service: 'db' started
Service: 'HTTPServer' started @ http://DrSarang:4080/
Service: 'APIService' started

Server Running
```

After the server starts running, you can open the server URL (`http://localhost:4080`) in your browser. When you do so, you will see the Xindice server's home page, as shown in Figure 9-2.

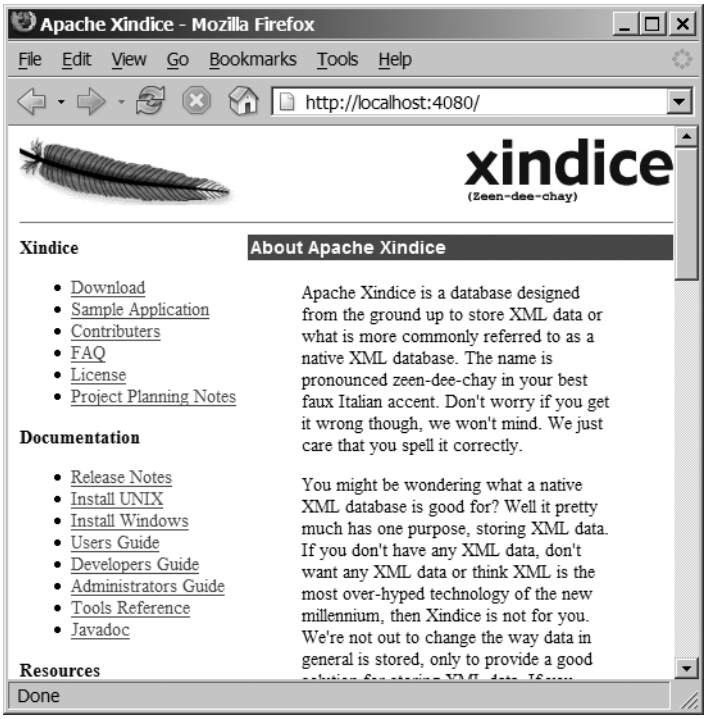

Figure 9-2. *Xindice server home page*

You are now ready to create a database.

Creating an XML Database

We will create a database used to store documents for the stock brokerage discussed in Chapter 2. The stock brokerage regularly creates several XML documents such as purchase orders, end-of-day quotes, and more. We will organize this data in a hierarchy of collections in the default `db` database.

To create and manage a Xindice database, Apache provides a command-line administrator utility called `xindiceadmin`. This utility allows you to create collections and subcollections in an existing database. A collection and a subcollection are the equivalent of a folder and a subfolder, respectively, in your directory hierarchy. A collection can contain a subcollection just as a folder can contain a subfolder.

It is also possible to create collections and subcollections through application code. This is explained later, in the "XML:DB API" section. For now, we will create collections for our stock brokerage by using the `xindiceadmin` utility.

Creating a Collection

Open a command prompt and change your working folder to your Xindice installation folder. You will create a collection called `StockBrokerage` under `db`.

To add a `StockBrokerage` collection to the `db` database, you use the `add_collection` (abbreviated `ac`) command switch on the `xindiceadmin` utility. Create the collection with the command line shown in the following screen output:

```
C:\xindice>xindiceadmin ac -c /db -n StockBrokerage
Created : /db/StockBrokerage
```

The `-c` switch specifies the context (`/db` in this case) under which the collection is to be added, and the `-n` switch specifies the name for the new collection. The full syntax for the `add_collection` command is as follows:

```
xindiceadmin add_collection {-c context} {-n name} [-v ]
```

The `-v` switch specifies verbose output.

If you examine the folder structure on your hard drive after running the command, you will notice that a subfolder called `StockBrokerage` has been created under the `<xindice installation folder>\db` folder. Also, note the creation of a TBL file called `StockBrokerage.tbl` in the created folder. This is used internally by the Xindice database server to save and track the documents added to this collection.

Creating Subcollections

You will now create two subcollections for our stock brokerage. The first collection, which we will call `DailyOrders`, will hold the XML document instances of the daily orders. The second collection, `DailyQuotes`, holds the documents containing the daily end-of-day prices of the various stocks traded on the exchange.

You add the `DailyOrders` subcollection by using the command line shown in the following screen output:

```
C:\xindice>xindiceadmin ac -c /db/StockBrokerage -n DailyOrders
Created : /db/StockBrokerage/DailyOrders
```

Note that, in this case, we specify the context as `/db/StockBrokerage`. The `DailyOrders` folder is now created under this context (folder). If you examine the folder structure after running the command, you will notice that a file called `DailyOrders.tbl` is added under the `DailyOrders` folder. Likewise, add a subcollection called `DailyQuotes` under `StockBrokerage` with the following command line:

```
C:\xindice>xindiceadmin ac -c /db/StockBrokerage -n DailyQuotes
```

Adding Documents

You will now add the daily orders documents and end-of-day quote files to the respective collections created in the previous sections. To add a document to a collection, you use another command-line utility called `xindice` with the `add_document` (abbreviated `ad`) switch.

Assume that our brokerage stores the daily orders files under the name `OrderMMDDYYYY.xml`, where `MMDDYYYY` specifies the date. Likewise, assume that it stores the daily quotation files under the name `EODMMDDYYYY.xml`.

■ **Note** The sample document files are available in the `Ch09\XMLDocs` folder of the code downloaded from the Source Code area of the Apress website (`http://www.apress.com`).

First, you will add the orders to our `DailyOrders` collection. The following screen output shows how to add an orders document:

```
C:\xindice>xindice ad -c /db/StockBrokerage/DailyOrders -f Ch09\X
MLDocs\Orders\Order01022006.xml -n Order01022006
Added document /db/StockBrokerage/DailyOrders/Order01022006
```

As in the case of `xindiceadmin`, the `-c` switch specifies the context (the collection) in which the document is to be added. The `-f` switch specifies the full path of the document on your local drive, and `-n` specifies the key to be assigned to the document. The key uniquely identifies the given document in the current context. If the `-n` switch is not specified, `xindice` will autogenerate the key. You will need this key to access this document in the database. To delete a document from the database, for example, you need to specify its key.

Adding Multiple Documents

You may occasionally want to add multiple documents to the database collection by using a single command. To do so, you use `add_multiple_documents` (alias `addmultiple`) on the `xindice` utility. The following screen output shows how the rest of the orders documents are added to the `DailyOrders` collection:

```
C:\xindice>xindice addmultiple -c /db/StockBrokerage/DailyOrders -f Ch
09\XMLDocs\Orders -e xml -v
Reading files from: XMLDocs
Added document /db/StockBrokerage/DailyOrders/Order01022006.xml
Added document /db/StockBrokerage/DailyOrders/Order01032006.xml
Added document /db/StockBrokerage/DailyOrders/Order01042006.xml
Added document /db/StockBrokerage/DailyOrders/Order01052006.xml
Added document /db/StockBrokerage/DailyOrders/Order01062006.xml
Added document /db/StockBrokerage/DailyOrders/Order01092006.xml
Added document /db/StockBrokerage/DailyOrders/Order01102006.xml
```

```
Added document /db/StockBrokerage/DailyOrders/Order01112006.xml
Added document /db/StockBrokerage/DailyOrders/Order01122006.xml
Added document /db/StockBrokerage/DailyOrders/Order01132006.xml
```

When you add multiple documents in this manner, `xindice` creates a unique key for each added document. This key has a value that is the same as its filename. Thus, the key for the `Order01132006.xml` file is `Order01132006.xml`. Note that even the file extension is used in the key value.

Likewise, add the provided `EOD` files to the `DailyQuotes` collection. The following screen output shows how to do this:

```
C:\xindice>xindice addmultiple -c /db/StockBrokerage/DailyQuotes -f Ch09\XMLDocs ➥
\Quotes -e xml -v
Reading files from: EOD
Added document /db/StockBrokerage/DailyQuotes/EOD01022006.xml
Added document /db/StockBrokerage/DailyQuotes/EOD01032006.xml
Added document /db/StockBrokerage/DailyQuotes/EOD01042006.xml
Added document /db/StockBrokerage/DailyQuotes/EOD01052006.xml
Added document /db/StockBrokerage/DailyQuotes/EOD01062006.xml
Added document /db/StockBrokerage/DailyQuotes/EOD01092006.xml
Added document /db/StockBrokerage/DailyQuotes/EOD01102006.xml
Added document /db/StockBrokerage/DailyQuotes/EOD01112006.xml
Added document /db/StockBrokerage/DailyQuotes/EOD01122006.xml
Added document /db/StockBrokerage/DailyQuotes/EOD01132006.xml
```

Now your database is ready for some real use.

Querying the Database

After a database is created, it can be queried to get a list of documents in the database, to retrieve a specified document from the database, to retrieve all the documents matching a specified search criterion, and more. In this section, I will introduce several commands to demonstrate these techniques.

Listing Documents in the Database

Imagine that our stock brokerage will retrieve a list of orders documents stored in the database. You use the `xindice` utility for this with the command `list_documents` (abbreviated `ld`). The screen output that results when you run the list query on your `DailyOrders` collection is as follows:

```
C:\xindice>xindice ld -c /db/StockBrokerage/DailyOrders

        Order01022006.xml
        Order01032006.xml
        Order01042006.xml
        Order01052006.xml
        Order01062006.xml
        Order01092006.xml
        Order01102006.xml
        Order01112006.xml
        Order01122006.xml
        Order01132006.xml

Total documents: 10
```

The `-c` switch specifies the context to search.

Retrieving a Document

Consider that the brokerage wants to retrieve the orders document for the date January 5, 2006 from the database. The following screen output shows how to perform this operation:

```
C:\xindice>xindice rd -c /db/StockBrokerage/DailyOrders -f c:\TempOrder.xml -n ➥
Order01052006.xml
Writing...
Wrote file c:\TempOrder.xml
```

The command switch used for retrieving documents is `rd` (`retrieve_document`). The `-n` switch specifies the document key. Note the use of the file extension in the key name. You specify the name of the output file with the switch `-f`. You can check the presence of the `TempOrder.xml` file in your root folder of drive C after running the preceding command. Check the file's contents to verify that it matches the original orders document for the said date.

Selecting Records Based on a Selection Criterion

You will now create a few queries that will select the records from our database based on the user-specified selection criterion. We specify the criterion with the help of an appropriate XPath[1] query. You will create a few such XPath queries to query our database.

Retrieving a Specific Order

Suppose the broker wants to retrieve an order for a specific customer on a specific date from the database. This can be achieved by using an XPath expression such as `"/customers[@date='01022006']/customer[@ID=1]"`. This would select the orders placed

1. Refer to Chapter 5 for details on XPath syntax.

on January 2, 2006 by the customer whose ID is 1. You specify this XPath query as a value to the `-q` switch on the `xindice` command-line utility.

The command line to run the query and its output are shown here:

```
C:\xindice>xindice xpath -c /db/StockBrokerage/DailyOrders -q "/customers➥
[@date='01022006']/customer[@ID=1]"

<?xml version="1.0"?>
<customer ID="1" xmlns:src="http://xml.apache.org/xindice/Query" src:col="/
db/StockBrokerage/DailyOrders" src:key="Order01022006">
    <name>Pat Irwin</name>
    <order>MSFT</order>
    <quantity>50</quantity>
    <price>25.35</price>
  </customer>
```

■ **Tip** The Xindice release notes given in `file:///C:/xml-xindice-1.0/docs/README` state the following: "On Windows, command-line queries can have problems with the quote handling of the Windows shell. In general, you should put double quotes around the entire query string and use single quotes in your XPath." If you encounter problems while running some of these queries, try them on Linux. I have successfully run all the examples in this chapter on Windows 2000, but had a few problems running them on Windows XP.

Retrieving All Orders for a Specific Customer

Our broker may want to retrieve the history of all the orders placed by a certain customer. In this case, you would use an XPath expression such as `/customers/customer[@ID=1]`. The result of running this query on the sample database is as follows:

```
C:\xindice>xindice xpath -c /db/StockBrokerage/DailyOrders -q "/customers/➥
customer[@ID=1]"

<?xml version="1.0"?>
<customer ID="1" xmlns:src="http://xml.apache.org/xindice/Query" src:col="/
db/StockBrokerage/DailyOrders" src:key="Order01022006">
    <name>Pat Irwin</name>
    <order>MSFT</order>
    <quantity>50</quantity>
    <price>25.35</price>
  </customer>
<?xml version="1.0"?>
<customer ID="1" xmlns:src="http://xml.apache.org/xindice/Query" src:col="/
```

```
db/StockBrokerage/DailyOrders" src:key="Order01032006">
    <name>Pat Irwin</name>
    <order>MSFT</order>
    <quantity>70</quantity>
    <price>25.30</price>
  </customer>
<?xml version="1.0"?>
<customer ID="1" xmlns:src="http://xml.apache.org/xindice/Query" src:col="/
db/StockBrokerage/DailyOrders" src:key="Order01042006">
    <name>Pat Irwin</name>
    <order>MSFT</order>
    <quantity>20</quantity>
    <price>25.80</price>
  </customer>
C:\xindice>
```

Retrieving Orders for a Specific Stock

Say you want to retrieve all the trade orders for IBM for a specified date. The XPath query for selecting this is `/customers[@date='01022006']/customer[order='IBM']`. Note that on the `customers` element you specify the `date` for selection of the orders, and for the `customer` element you specify the `order` element. The `date` is an attribute on the `customers` element and is thus preceded by an `@` sign. The following is the result of running this query on our sample database:

```
C:\xindice>xindice xpath -c /db/StockBrokerage/DailyOrders -q ➥
"/customers[@date='01022006']/customer[order='IBM']"

<?xml version="1.0"?>
<customer ID="2" xmlns:src="http://xml.apache.org/xindice/Query" src:col="/
db/StockBrokerage/DailyOrders" src:key="Order01022006">
    <name>Nancy Scheffler</name>
    <order>IBM</order>
    <quantity>75</quantity>
    <price>25.30</price>
  </customer>
<?xml version="1.0"?>
<customer ID="3" xmlns:src="http://xml.apache.org/xindice/Query" src:col="/
db/StockBrokerage/DailyOrders" src:key="Order01022006">
    <name>Cheryl Zuckschwerdt</name>
    <order>IBM</order>
    <quantity>30</quantity>
    <price>25.50</price>
  </customer>
<?xml version="1.0"?>
```

```
<customer ID="4" xmlns:src="http://xml.apache.org/xindice/Query" src:col="/
db/StockBrokerage/DailyOrders" src:key="Order01022006">
    <name>Bennie Furlong</name>
    <order>IBM</order>
    <quantity>65</quantity>
    <price>25.55</price>
  </customer>
```

Determining Trades for a Specific Stock by a Specified Customer

You might want to retrieve a history of all the trades performed by a specified customer on a specified stock. The XPath query for this is given by `/customers/customer[@ID=4][order='IBM']`.

The output after running this query on a sample database is as follows:

```
C:\xindice>xindice xpath -c /db/StockBrokerage/DailyOrders -q "/customers/➥
customer[@ID=4][order='IBM']"

<?xml version="1.0"?>
<customer ID="4" xmlns:src="http://xml.apache.org/xindice/Query" src:col="/
db/StockBrokerage/DailyOrders" src:key="Order01022006">
    <name>Bennie Furlong</name>
    <order>IBM</order>
    <quantity>65</quantity>
    <price>25.55</price>
  </customer>
<?xml version="1.0"?>
<customer ID="4" xmlns:src="http://xml.apache.org/xindice/Query" src:col="/
db/StockBrokerage/DailyOrders" src:key="Order01032006">
    <name>Bennie Furlong</name>
    <order>IBM</order>
    <quantity>60</quantity>
    <price>25.25</price>
  </customer>
<?xml version="1.0"?>
<customer ID="4" xmlns:src="http://xml.apache.org/xindice/Query" src:col="/
db/StockBrokerage/DailyOrders" src:key="Order01042006">
    <name>Bennie Furlong</name>
    <order>IBM</order>
    <quantity>70</quantity>
    <price>25.20</price>
  </customer>
```

Determining Orders for a Specified Stock at a Specified Price

A broker may want to consolidate all the orders by different customers placed on a particular stock at a particular requested price, before the order is finally placed on the stock exchange. The XPath query to select such records is given by `/customers[@date='01032006']/customer[price='$25.25']`. The following is the result of running this query on the sample database:

```
C:\xindice>xindice xpath -c /db/StockBrokerage/DailyOrders -q "/customers➥
[@date='01032006']/customer[price='$25.25']"

<?xml version="1.0"?>
<customer ID="3" xmlns:src="http://xml.apache.org/xindice/Query" src:col="/
db/StockBrokerage/DailyOrders" src:key="Order01032006">
    <name>Cheryl Zuckschwerdt</name>
    <order>IBM</order>
    <quantity>100</quantity>
    <price>25.25</price>
  </customer>
<?xml version="1.0"?>
<customer ID="4" xmlns:src="http://xml.apache.org/xindice/Query" src:col="/
db/StockBrokerage/DailyOrders" src:key="Order01032006">
    <name>Bennie Furlong</name>
    <order>IBM</order>
    <quantity>60</quantity>
    <price>25.25</price>
  </customer>
```

Using the XML:DB API

So far you have learned how to create, maintain, and query the XML database through command-line utilities. Now you will learn to do the same through a programmatic interface.

The XML:DB Working Group has developed the API for XML databases. This API can be implemented in multiple languages. A CORBA API is designed to access the database. The Xindice libraries provide a Java-based API that uses this CORBA API for the following:

- Creating and maintaining the database collections
- Querying the database
- Deleting the data
- Updating the database

In this section, you will learn all these techniques. First, you will learn to create and maintain a collection of XML documents in a database. You will use the database structure described earlier in this chapter for the command-line interface.

Creating Collections

You will write an application that creates a collection called `Brokerage` in the default database `db`. You will use a different name than the earlier case. Earlier in this chapter you used the name `StockBrokerage` while creating the database by using a command-line tool. If you try to create a collection with an existing name, the program will throw an application exception. Thus, the change in name!

Under the `Brokerage` collection, you will create two subcollections called `DailyQuotes` and `DailyOrders`. You will use the same names for the subcollections as in the earlier case. Note that the fully qualified path for these two subcollections would be different from the path for the subcollections created earlier.

The program that creates this database is presented in Listing 9-1.

Listing 9-1. *Program to Add Collections and Subcollections to a Xindice Database (*`Ch09\src\BrokerDatabase.java`*)*

```
package database;

import org.xmldb.api.base.*;
import org.xmldb.api.modules.*;
import org.xmldb.api.*;
import org.apache.xindice.client.xmldb.services.*;
import org.apache.xindice.xml.dom.*;

public class BrokerDatabase {
  public static void main(String[] args) throws Exception {
    Collection col = null;
    try {
      // Load the database driver
      String driver = "org.apache.xindice.client.xmldb.DatabaseImpl";
      Class c = Class.forName(driver);

      // Create a Database instance and register it to the DatabaseManager
      Database database = (Database) c.newInstance();
      DatabaseManager.registerDatabase(database);

      // Get the reference to the root collection
      col = DatabaseManager.getCollection("xmldb:xindice:///db");

      // Set up name for the new collection
      String collectionName = "Brokerage";

      // Obtain an instance
      CollectionManager service = (CollectionManager)
      col.getService("CollectionManager", "1.0");
```

```
// Build up the Collection XML configuration.
String collectionConfig =
    "<collection compressed=\"true\" name=\"" +
    collectionName + "\">" +
    " <filer class=" +
    "\"org.apache.xindice.core.filer.BTreeFiler\" " +
    "gzip=\"true\"/>" +
    "</collection>";

service.createCollection(collectionName,
    DOMParser.toDocument(collectionConfig));

System.out.println("Collection " + collectionName + " created.");

col = DatabaseManager.getCollection("xmldb:xindice:///db/Brokerage");
collectionName = "DailyQuotes";
service = (CollectionManager)
col.getService("CollectionManager", "1.0");

// Build up the Collection XML configuration.
collectionConfig =
    "<collection compressed=\"true\" name=\"" +
    collectionName + "\">" +
    " <filer class=" +
    "\"org.apache.xindice.core.filer.BTreeFiler\" " +
    "gzip=\"true\"/>" +
    "</collection>";

service.createCollection(collectionName,
    DOMParser.toDocument(collectionConfig));

System.out.println("Collection " + collectionName + " created.");

collectionName = "DailyOrders";
// Build up the Collection XML configuration.
collectionConfig =
    "<collection compressed=\"true\" name=\"" +
    collectionName + "\">" +
    " <filer class=" +
    "\"org.apache.xindice.core.filer.BTreeFiler\" " +
    "gzip=\"true\"/>" +
    "</collection>";

service.createCollection(collectionName,
    DOMParser.toDocument(collectionConfig));
```

```
      System.out.println("Collection " + collectionName + " created.");
    } catch (XMLDBException e) {
      System.err.println("XML:DB Exception occured " +
        e.getLocalizedMessage());
    } finally {
      if (col != null) {
        col.close();
      }
    }
  }
}
```

Let's examine the program in detail. To begin, the `main` function first loads the database driver:

```
// Load the database driver
String driver = "org.apache.xindice.client.xmldb.DatabaseImpl";
Class c = Class.forName(driver);
```

The database driver is implemented in the `org.apache.xindice.client.xmldb.DatabaseImpl` class. The `forName` method of the class `Class` loads this driver file in memory.

Next, the program obtains an instance of the `Database` class and registers it with the `DatabaseManager` by calling its `registerDatabase` static method:

```
// Create a Database instance and register it to the DatabaseManager
Database database = (Database) c.newInstance();
DatabaseManager.registerDatabase(database);
```

You now obtain a reference to the root collection as follows:

```
// Get the reference to the root collection
col = DatabaseManager.getCollection("xmldb:xindice:///db");
```

The `/db` specifies the XPath to the root collection, and `xmldb:xindice` specifies the protocol used for accessing the database collections.

At this point, you are ready to add a collection to the root you have obtained. First, you create a name for the collection to be added:

```
// Set up name for the new collection
String collectionName = "Brokerage";
```

Next, you obtain a reference to the `CollectionManager` object:

```
// Obtain an instance
CollectionManager service = (CollectionManager)
col.getService("CollectionManager", "1.0");
```

The `getService` method of the `Collection` class returns a reference to the `CollectionManager`. The `CollectionManager` class, as the name suggests, is responsible for managing collections.

Next, you need to set up the collection configuration XML fragment as follows:

```
<collection compressed="true" name="Brokerage">
  <filer class="org.apache.xindice.core.filer.BTreeFiler" gzip="true"/>
</collection>
```

While creating the collection, the `CollectionManager` will use this configuration, which specifies that the collection should be stored in compressed form and that the name of the collection is `Brokerage`. It also specifies the class that is used for filing the document instances. The `BTreeFiler` is a Xindice-provided class that files (arranges) the documents in a binary tree.

You construct this XML document fragment by creating a `String` object as follows:

```
// Build up the Collection XML configuration.
String collectionConfig =
    "<collection compressed=\"true\" name=\"" +
    collectionName + "\">" +
    " <filer class=" +
    "\"org.apache.xindice.core.filer.BTreeFiler\" " +
    "gzip=\"true\"/>" +
    "</collection>";
```

Now, you will add the collection to the root by calling the `createCollection` method on the service manager. The method takes the collection name and the instance of the DOM[2] tree representing the configuration:

```
service.createCollection(collectionName,
    DOMParser.toDocument(collectionConfig));
```

On success, the collection is added to the desired path in the database. Next, you will add a subcollection to the newly created collection. The procedure for adding a subcollection is similar to that of adding a collection, except that you need to set the appropriate root path for adding the subcollection. This is done by obtaining the reference to the new desired root as follows:

```
col = DatabaseManager.getCollection("xmldb:xindice:///db/Brokerage");
```

Note that you specify the path as `/db/Brokerage`. The collections will now be added to this path. As in the earlier case, you set the name for the new collection:

```
collectionName = "DailyQuotes";
```

Next, you will add this collection by using the steps as outlined earlier. Likewise, you add another subcollection called `DailyOrders` to the `Brokerage` collection.

Running the Application

Compile the preceding program by using the following command line:

```
C:\{working folder}>javac -d . BrokerDatabase.java
```

2. Refer to Chapter 2 for DOM details.

Run the application by using the following command line:

```
C:\{working folder}>java database.BrokerDatabase
```

If the program succeeds, you will find that collections are created in the specified database. You can verify this by using the `list_collection` method on the `xindice` command-line utility and specifying the context as `/db` and `/db/Brokerage`, respectively. The output of these commands is shown here:

```
C:\xindice>xindice lc -c /db

        Brokerage
        system
        StockBrokerage

Total collections: 3
```

```
C:\xindice>xindice lc -c /db/Brokerage

        DailyOrders
        DailyQuotes
Total collections: 2
```

■ **Note** The list of collections may be different on your machine, depending on the existing collections in your database.

If you run the application a second time, you will get a duplicate collection exception. To avoid the exception, you'll first need to delete the created collections. You can do this by using the `delete_collection` (abbreviated `dc`) command on the `xindice` utility. The screen output when deleting the collection is shown here:

```
C:\xindice>xindiceadmin dc -c /db -n Brokerage
Are you sure you want to delete the collection Brokerage ? (y/n)
y
Deleted: /db/Brokerage
```

Alternatively, you can delete the collections through your application program by calling the `removeCollection` method on the service manager.

■ **Note** The program for removing the `Brokerage` collection is available in the file `Ch09\src\DeleteCollection.java` in the source downloaded from the Source Code area of the Apress website (`http://www.apress.com`).

Adding Documents to Collections

Previously you learned how to create collections and subcollections in your XML database. Now you will learn to add XML documents to the created collection. An XML document is added to the collection as its resource object. For this purpose, the XML:DB API provides a class called `XMLResource`. You will create an instance of this class, initialize its contents with the contents from the XML file resource, and then add it to the desired collection. The program that does all this is given in Listing 9-2.

Listing 9-2. *Application for Adding Documents to a Database Collection* (`Ch09\src\AddDocument.java`)

```
package database;

// import required packages

public class AddDocument {
  static AddDocument doc;
  Collection col = null;
  public static void main(String[] args) throws Exception {
    if (args.length != 2)
    {
      System.out.println("Usage: java AddDocument path extension");
      System.exit(1);
    }
    // Copy the command line arguments
    String path = args[0];
    String ext = args[1];
    try {
      // Create the class instance
      doc = new AddDocument();
      // Obtain the database collection reference
      doc.col = doc.getDatabaseRoot();
      // Add documents to the database
      doc.addFiles(doc.col, path, ext);
    } catch (XMLDBException e) {
      System.err.println("XML:DB Exception occured " + e.errorCode);
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
```

```
    // Close the collection
    if (doc.col != null) {
       doc.col.close();
    }
  }
}

/*
 * getDatabaseRoot makes a database connection and returns a reference
 * to the collection given by an XPath expression.
 */
private Collection getDatabaseRoot() throws XMLDBException, Exception {
  Collection coll;
  // Load the database driver
  String driver = "org.apache.xindice.client.xmldb.DatabaseImpl";
  Class c = Class.forName(driver);
  // Create a database instance
  Database database = (Database) c.newInstance();
  // Register the database instance with the Manager
  DatabaseManager.registerDatabase(database);
  // Obtain a reference to the predefined collection
  coll = DatabaseManager.getCollection
         ("xmldb:xindice:///db/StockBrokerage/DailyOrders");
  // Return the collection reference to the caller
  return coll;
}

/*
 * addFiles adds the files to the specified collection from the
 * specified folder and having a specified extension
 */
private void addFiles(Collection c, String path, String ext)
  throws IOException, Exception {
  // Create a directory object
  File f = new File(path);
  // Get the list of files having the specified extension
  File[] list = f.listFiles(new ExtensionFilter(ext));
  // Iterate through the file collection
  for (int i=0; i<list.length;i++) {
    // Read the file contents into a string buffer
    String data = readFileFromDisk(list[i]);
    // Create a new resource on the specified collection
    XMLResource document =
                  (XMLResource) col.createResource(null,"XMLResource");
    // Set the contents of the created resource
    document.setContent(data);
```

```
      // Add the resource to the collection
      col.storeResource(document);
      // Print a message to the user
      System.out.println
                  ("Document " + list[i].getCanonicalFile() + " inserted");
    }
  }

  /*
   * readFromDiskFile reads the contents of the specified File object
   * into a string buffer and returns it to the caller
   */
  public String readFileFromDisk(File fileName) throws IOException {
    // Open an input stream on the given File
    FileInputStream in = new FileInputStream(fileName);
    // Create a buffer for reading file contents
    byte[] fileBuffer = new byte[(int)fileName.length()];
    // Read the file contents into the created buffer
    in.read(fileBuffer);
    // Close the file object
    in.close();
    // Return the buffer to the caller
    return new String(fileBuffer);
  }

  /*
   * ExtensionFilter creates a filter class for selecting the files
   * with the specified extension from a folder. The class implements the
   * accept method of the FilenameFilter interface that filters the files
   * using the specified extension.
   */
  public class ExtensionFilter implements FilenameFilter {
    private String extension ;

      // Constructor that initializes the extension filter
    public ExtensionFilter(String ext) {
      extension="." + ext;
    }
    // The accept method returns files matching the specified extension
    public boolean accept(File dir, String name) {
      return name.endsWith(extension);
    }
  }
}
```

The program accepts two command-line parameters. The first parameter specifies the folder name in which the documents are stored, and the second parameter specifies the filename extension on which the files from this folder will be filtered out. You copy the command-line arguments into local variables:

```
// Copy the command line arguments
String path = args[0];
String ext = args[1];
```

You create a class instance so that you can call its nonstatic methods:

```
doc = new AddDocument();
```

You obtain the reference to a predefined database collection in which you will add the documents:

```
// Obtain the database collection reference
doc.col = doc.getDatabaseRoot();
```

You add the documents by calling the `addFiles` method (discussed later) of our class:

```
// Add documents to the database
doc.addFiles(doc.col, path, ext);
```

You provide the exception processing for the entire application in the `main` method and finally close the database collection:

```
} catch (XMLDBException e) {
  System.err.println("XML:DB Exception occured " + e.errorCode);
} catch (Exception e) {
  e.printStackTrace();
} finally {
// Close the collection
if (doc.col != null) {
    doc.col.close();
}
}
```

You will now look at the implementation of the `getDatabaseRoot` method:

```
private Collection getDatabaseRoot() throws XMLDBException, Exception {
```

The method throws two types of exceptions that are then processed by the caller (this is the `main` method in our case).

The method first loads the database driver into the system:

```
// Load the database driver
String driver = "org.apache.xindice.client.xmldb.DatabaseImpl";
Class c = Class.forName(driver);
```

You then create a `Database` instance and register it with the database manager:

```
// Create a database instance
Database database = (Database) c.newInstance();
// Register the database instance with the Manager
DatabaseManager.registerDatabase(database);
```

You obtain a reference to the predefined collection in the database. Remember that you created the `StockBrokerage/DailyOrders` collection in the previous example. If you have not run this program (Listing 9-1) earlier, you may do so now or you may create this collection by using the command tool `xindiceadmin` described earlier:

```
// Obtain a reference to the predefined collection
coll = DatabaseManager.getCollection
("xmldb:xindice:///db/StockBrokerage/DailyOrders");
```

The method returns a reference to the obtained `Collection` object:

```
// Return the collection reference to the caller
return coll;
```

You will now learn about the implementation of the `addFiles` method. The `addFiles` method accepts the reference to the collection object obtained in the previous method (`getDatabaseRoot`) call, the folder path in which the documents are stored, and the extension that is used as a filter string:

```
private void addFiles(Collection c, String path, String ext)
    throws IOException, Exception {
```

You first construct a `File` object by using the specified path as its argument:

```
// Create a directory object
File f = new File(path);
```

This opens the folder with the specified path. You call the `listFiles` method on this folder object to obtain a list of files from this folder:

```
// Get the list of files having the specified extension
File[] list = f.listFiles(new ExtensionFilter(ext));
```

While obtaining the list of files, you specify the filter by instantiating the custom class `ExtensionFilter` (discussed later) that accepts the filter string to be used as an argument to its constructor.

You now iterate through the list of retrieved files:

```
// Iterate through the file collection
for (int i=0; i<list.length;i++) {
```

For each `File` object, you read its contents by calling the `readFileFromDisk` method (discussed later) that returns the file contents into a string buffer:

```
// Read the file contents into a string buffer
String data = readFileFromDisk(list[i]);
```

You create an XMLResource object for adding it to the collection:

```
// Create a new resource on the specified collection
XMLResource document =
            (XMLResource) col.createResource(null,"XMLResource");
```

You set the contents of the resource to the string data returned by the readFileFromDisk method:

```
// Set the contents of the created resource
document.setContent(data);
```

You add the resource to the collection by calling its storeResource method:

```
// Add the resource to the collection
col.storeResource(document);
```

You print a confirmation message to the user and go into another iteration of the file list:

```
// Print a message to the user
System.out.println
           ("Document " + list[i].getCanonicalFile() + " inserted");
```

You will now look at the implementation of the readFromDiskFile method. The method takes a File object as its argument and returns a string containing the file contents:

```
public String readFileFromDisk(File fileName) throws IOException {
```

You open an input stream on the specified file:

```
// Open an input stream on the given File
FileInputStream in = new FileInputStream(fileName);
```

You create a buffer equal to the length of the file for storing the file contents:

```
// Create a buffer for reading file contents
byte[] fileBuffer = new byte[(int)fileName.length()];
```

You read the file contents into the created buffer and then close the file:

```
  in.read(fileBuffer);
  in.close();
```

You construct a String object from the byte buffer and return it to the caller:

```
  return new String(fileBuffer);
```

You will now learn about the implementation of the ExtensionFilter class that is declared interior to our AddDocument class:

```
public class ExtensionFilter implements FilenameFilter {
```

The ExtensionFilter class implements the FilenameFilter interface. The object of this class is used in the listFiles method of the File class to filter the file selection. The FilenameFilter provides a method called accept that you need to implement in this class.

The class constructor accepts a string argument that represents the file extension to be used for filtering and initializes a private string variable by using this extension string:

```
// Constructor that initializes the extension filter
public ExtensionFilter(String ext) {
    extension="." + ext;
}
```

The `accept` method returns a `boolean` depending on whether the given name ends with the specified extension:

```
// The accept method returns files matching the specified extension
public boolean accept(File dir, String name) {
  return name.endsWith(extension);
}
```

Running the Application

Compile the application by using the following command line:

```
C:\{working folder}>javac -d . AddDocument.java
```

Run the application by using the following command:

```
C:\{working folder }>java database.AddDocument ..\XMLDocs\Demo xml
```

The first command-line argument (`..\XMLDocs\Demo`) specifies the folder in which the desired documents are stored, and the second argument (`xml`) specifies the file extension for filtering. On successful run of the application, you will see output similar to the following:

```
Document C:\apress\Ch09\XMLDocs\Demo\Order01162006.xml inserted
Document C:\apress\Ch09\XMLDocs\Demo\Order01172006.xml inserted
```

Next, you will write an application to list all the documents stored in our database starting from a specified collection path.

Listing All Documents in the Specified XPath

The XML database is arranged in a binary tree hierarchy. To visit every node of the tree, you need to write a recursive function. The program in Listing 9-3 illustrates how to traverse the tree recursively and list all the documents in each visited subcollection.

Listing 9-3. *Application to List All the Documents Under a Given Node (*Ch09\src\ListDocs.java*)*

```
package database;

// import required packages

public class ListDocs {
  public static void main(String[] args) throws Exception {
    if (args.length != 1) {
      System.out.println("Usage: java ListDocs XPathtoDesiredCollection");
      System.exit(1);
    }
    try {
      // Load the database driver
      String driver = "org.apache.xindice.client.xmldb.DatabaseImpl";
      Class c = Class.forName(driver);
      // Create a database instance
      Database database = (Database) c.newInstance();
       // Register the database instance with the Manager
      DatabaseManager.registerDatabase(database);
      // Construct the XPath starting from the root
      String strPath = "xmldb:xindice:///db" + args[0];
      System.out.println("Printing the list of resources at " + strPath);
      // Get the reference to the desired collection
      Collection coll = DatabaseManager.getCollection(strPath);
      // Recursively get a list of subcollections and documents therein
      ListAllDocuments(coll);
    } catch (XMLDBException e) {
      System.err.println("XML:DB Exception occured " + e.errorCode);
    } catch (Exception e) {
       e.printStackTrace();
    }
  }

  /*
   * ListAllDocuments accepts a Collection object as an argument and
   * traverses the tree recursively listing all the sub-collections and
   * documents therein.
   */
  private static void ListAllDocuments(Collection coll) throws XMLDBException {
    // Get string names of all child collections
    String [] str = coll.listChildCollections();
    // Iterate through the list of subcollections
    for (int i=0; i<str.length; i++) {
      // Print the subcollection name
      System.out.println("Collection: " + str[i]);
```

```
      // Get a list of resources within each subcollection
      String [] docs = (coll.getChildCollection(str[i])).listResources();
      // Print the names of document resources within a subcollection
      for (int j=0; j<docs.length; j++) {
        System.out.println(docs[j]);
      }
      // Revisit the function using the subcollection
      ListAllDocuments(coll.getChildCollection(str[i]));
    }
  }
}
```

The application accepts the XPath expression to the desired collection as an argument. In the `main` method, you construct the reference to this collection. To do this, you first load the database driver in memory:

```
      // Load the database driver
      String driver = "org.apache.xindice.client.xmldb.DatabaseImpl";
      Class c = Class.forName(driver);
```

You create a database instance and register it with the database manager:

```
      // Create a database instance
      Database database = (Database) c.newInstance();
       // Register the database instance with the Manager
      DatabaseManager.registerDatabase(database);
```

You construct the path to the desired collection starting from the database root and using the `xmldb:xindice` protocol:

```
      String strPath = "xmldb:xindice:///db" + args[0];
```

You now obtain a reference to the desired collection by calling the `getCollection` method on the database manager:

```
      // Get the reference to the desired collection
      Collection coll = DatabaseManager.getCollection(strPath);
```

Now you are ready to traverse the tree with this collection as the root element. You do so by calling the `ListAllDocuments` method of our class:

```
      // Recursively get a list of sub-collections and documents therein
      ListAllDocuments(coll);
```

You will now learn about the implementation of the `ListAllDocuments` method. This method accepts the reference to the root collection, from where the search to subcollections begins. It throws an `XMLDBException` to the caller:

```
  private static void ListAllDocuments(Collection coll) throws XMLDBException {
```

First, you obtain the list of child collections by calling the `listChildCollections` method on the collection object:

```
// Get string names of all child collections
String [] str = coll.listChildCollections();
```

You iterate through the list of subcollections and print the name of each subcollection on the user console as you visit it:

```
// Iterate through the list of subcollections
for (int i=0; i<str.length; i++) {
  // Print the subcollection name
  System.out.println("Collection: " + str[i]);
```

You get the list of document resources within each subcollection by calling the `listResources` method on the `Collection` object:

```
  // Get a list of resources within each sub-collection
  String [] docs = (coll.getChildCollection(str[i])).listResources();
```

You print the name of each document by iterating through the entire list of documents:

```
  // Print the names of document resources within a sub-collection
  for (int j=0; j<docs.length; j++) {
    System.out.println(docs[j]);
  }
```

You revisit the `ListAllDocuments` method by passing the new `Collection` reference to it. The reference to the subcollection is obtained by calling the `getChildCollection` method on the `Collection` object. The method takes the name of the desired collection as an argument and returns a `Collection` object having this name:

```
  // Revisit the function using the subcollection
  ListAllDocuments(coll.getChildCollection(str[i]));
```

Running the Application

Compile the code in Listing 9-3 by using the following command line:

```
C:\{working folder}>javac -d . ListDocs.java
```

Run the application by using the following command line:

```
C:\{working folder}>java database.ListDocs /StockBrokerage
```

If you have followed the instructions in this chapter and created the suggested sample database, you will see output similar to the following when you run the preceding program:

```
Printing the list of resources at xmldb:xindice:///db/StockBrokerage
Collection: DailyOrders
Order01022006.xml
Order01032006.xml
Order01042006.xml
Order01052006.xml
Order01062006.xml
Order01092006.xml
Order01102006.xml
Order01112006.xml
Order01122006.xml
Order01132006.xml
Collection: DailyQuotes
EOD01022006.xml
EOD01032006.xml
EOD01042006.xml
EOD01052006.xml
EOD01062006.xml
EOD01092006.xml
EOD01102006.xml
EOD01112006.xml
EOD01122006.xml
EOD01132006.xml
```

Deleting Documents

You will now write an application to remove all the documents recursively, starting from a specified node. Listing 9-4 provides the recursive function that does this. The rest of the code is similar to the code in Listing 9-3.

■ **Tip** The full source of this application is available at `Ch09\src\RemoveDocs.java` in the source download.

Listing 9-4. *Method to Remove All Documents from a Given Collection*

```
private static void RemoveAllDocuments(Collection coll) throws XMLDBException
 {
   // Get string names of all child collections
   String [] str = coll.listChildCollections();
   // Iterate through the list of sub-collections
   for (int i=0; i<str.length; i++) {
```

```
      // Print the subcollection name
      System.out.println("Collection: " + str[i]);
      // Get a list of resources within each subcollection
      String [] docs = (coll.getChildCollection(str[i])).listResources();
      for (int j=0; j<docs.length; j++) {
        // Obtain the resource reference
            Resource resource =
                coll.getChildCollection(str[i]).getResource(docs[j]);
        // Remove the resource
        coll.getChildCollection(str[i]).removeResource(resource);
            System.out.println("Resource " + resource.getId() + "removed");
      }
      // Revisit the function using the subcollection
      RemoveAllDocuments(coll.getChildCollection(str[i]));
    }
  }
```

The `RemoveAllDocuments` method accepts the `Collection` reference under which the documents are to be deleted. This is a recursive method similar to the `ListAllDocuments` method discussed earlier:

```
private static void RemoveAllDocuments(Collection coll) throws XMLDBException {
```

We first obtain the list of child collections:

```
// Get string names of all child collections
String [] str = coll.listChildCollections();
```

We iterate through the list and print the name of each subcollection as we visit it:

```
// Iterate through the list of subcollections
for (int i=0; i<str.length; i++) {
  // Print the subcollection name
  System.out.println("Collection: " + str[i]);
```

We obtain the list of resources within each subcollection by calling the `listResources` method on the `Collection` object:

```
      // Get a list of resources within each subcollection
      String [] docs = (coll.getChildCollection(str[i])).listResources();
```

For each listed resource, we obtain its `Resource` object reference by calling the `getResource` method on the `Collection` object. The `getResource` method accepts the resource name as its argument and returns a reference to the `Resource` object:

```
      for (int j=0; j<docs.length; j++) {
        // Obtain the resource reference
          Resource resource =
              coll.getChildCollection(str[i]).getResource(docs[j]);
```

The resource is removed by calling the `removeResource` method on the `Collection` object. The method accepts the reference to the resource that is to be removed as a parameter.

```
// Remove the resource
coll.getChildCollection(str[i]).removeResource(resource);
    System.out.println("Resource " + resource.getId() + "removed");
```

We revisit the method for a recursive call:

```
// Revisit the function using the sub-collection
RemoveAllDocuments(coll.getChildCollection(str[i]));
```

So far, you have learned how to create collections and subcollections, and how to manage the database hierarchy of collections. You have also learned how to add documents to a collection, how to list documents in a given collection, and how to remove documents from a collection. Next, you will learn how to use XUpdate to modify the content of a document without removing it from the database.

Using the XUpdate Language

XUpdate is a language for updating XML documents and is the result of the work done as a part of the XML:DB project. To update a document, you create an XUpdate query and then run it by using the provided update query service. The query itself is written as a well-formed XML document fragment.

A typical XUpdate query is shown here:

```
<xu:modifications version="1.0" xmlns:xu="http://www.xmldb.org/xupdate">
   <xu:remove select="/customers[@date='01022006']/customer[@ID=5]"/>
</xu:modifications>
```

The query is specified in the `modifications` element, which is defined in the `http://www.xmldb.org/xupdate` namespace. The `remove` subelement is a subelement to `modifications` and indicates that we are interested in removing something (obviously an element). An element to be removed is located by using an XPath query. This is specified as a value to the `select` attribute. This query is explained further in the "Updating a Node in the Orders Collection" section. Let's first look at the full syntax of the XUpdate query.

Creating the XUpdate Query

As I said, the XUpdate query is defined by using the `modifications` element. The modifications that are allowed consist of Insert/Update/Delete operations. Thus, it is possible to insert a node in an existing document by using the XUpdate query. For this, the XUpdate specifications define the following three subelements:

- `xupdate:insert-before`: Inserts a node before the specified record (given by an XPath expression)
- `xupdate:insert-after`: Inserts a node after the specified record
- `xupdate:append`: Adds a node to the end of the document

A record position is specified by using an XPath expression assigned to the `select` attribute of the appropriate child element.

To insert a record, you must first construct it. The record is constructed with the help of the following subelements:

- `xupdate:element`
- `xupdate:attribute`
- `xupdate:text`
- `xupdate:processing-instruction`
- `xupdate:comment`

You will learn to construct a record and add it to an existing document in the programming section that follows shortly.

To modify an existing record, you use the `update` subelement of the `modifications` element. Similarly, to remove a record, you use the `remove` subelement.

Let's now look at some coding examples so you can understand how to perform database updates by using an XUpdate query.

Removing a Node

Suppose you want to remove an order placed by a customer from the orders document. You will use an XUpdate query to do this. Listing 9-5 shows the program that removes a record from the specified orders document for a specified customer ID.

Listing 9-5. *Application to Remove a Node (*`Ch09\src\DeleteNode.java`*)*

```
package database;

import org.xmldb.api.base.*;
import org.xmldb.api.modules.*;
import org.xmldb.api.*;
import java.io.*;

public class DeleteNode {
  public static void main(String[] args) throws Exception {
    Collection col = null;
    // Check arguments
    if (args.length != 2) {
      System.out.println("Usage: java DeleteNode CustomerID");
      System.exit(1);
    }
    try {
      // Load driver class
      String driver = "org.apache.xindice.client.xmldb.DatabaseImpl";
      Class c = Class.forName(driver);
```

```
      // Create and register database instance
      Database database = (Database) c.newInstance();
      DatabaseManager.registerDatabase(database);
      // Retrieve a reference to DailyOrders collection
      col = DatabaseManager.getCollection
          ("xmldb:xindice:///db/StockBrokerage/DailyOrders");
       // Construct a Remove query
      String xupdate =
        "<xu:modifications version=\"1.0\"" +
        "xmlns:xu=\"http://www.xmldb.org/xupdate\">\n" +
        "xu:remove " +
        "select=\"/customers[@date='" +
        args[0] +
        "']/customer[@ID=" +
        args[1] +
        "]\"/>"
        + "\n</xu:modifications>";
      // Obtain a reference to the update query service
      XUpdateQueryService service =
          (XUpdateQueryService)col.getService
          ("XUpdateQueryService", "1.0");
      // Run the query using the service
      System.out.println("Running Remove Query: ");
      System.out.println(xupdate);
      service.update(xupdate);
    } catch (XMLDBException e) {
        System.err.println("XML:DB Exception occured " + e.errorCode + " " +
            e.getMessage());
    } finally {
        // Close the collection
        if (col != null) {
            col.close();
        }
    }
  }
}
```

The program accepts the two command-line parameters. The first parameter specifies the date for the orders document in our `DailyOrders` collection. The date is specified in MMDDYYYY format. The second argument specifies the customer ID as a string. The record matching this customer ID will be removed from the specified document.

The `main` function first loads the driver:

```
      // Load driver class
      String driver = "org.apache.xindice.client.xmldb.DatabaseImpl";
      Class c = Class.forName(driver);
```

As in the earlier examples, you create an instance of the database and register it with the database manager:

```
// Create and register database instance
Database database = (Database) c.newInstance();
DatabaseManager.registerDatabase(database);
```

Next, you obtain a reference to the `DailyOrders` collection:

```
// Retrieve a reference to DailyOrders collection
col = DatabaseManager.getCollection
    ("xmldb:xindice:///db/StockBrokerage/DailyOrders");
```

You now construct the XML document fragment to create a `remove` query by using the XUpdate language syntax:

```
 // Construct a Remove query
String xupdate =
  "<xu:modifications version=\"1.0\"" +
  "xmlns:xu=\"http://www.xmldb.org/xupdate\">\n" +
  "xu:remove " +
  "select=\"/customers[@date='" +
  args[0] +
  "']/customer[@ID=" +
  args[1] +
  "]\"/>"
  + "\n</xu:modifications>";
```

The XML document fragment that is generated by the preceding statement for the command-line arguments `01022006` and `8` is given here:

```
<xu:modifications version="1.0" xmlns:xu="http://www.xmldb.org/xupdate">
   <xu:remove select="/customers[@date='01022006']/customer[@ID=8]"/>
</xu:modifications>
```

Note the use of the `date` attribute to select the orders document and the use of the `ID` attribute to select a particular customer.

To run the query, you need to obtain a reference to the provided query service. You do this by calling the `getService` method of the `Collection` class:

```
// Obtain a reference to the update query service
XUpdateQueryService service =
    (XUpdateQueryService)col.getService
    ("XUpdateQueryService", "1.0");
```

Finally, you run the query itself by calling the `update` method on the `service` object obtained in the previous step:

```
// Run the query using the service
System.out.println("Running Remove Query: ");
System.out.println(xupdate);
service.update(xupdate);
```

If the query runs successfully, the specified record will be deleted from the selected document.

Running the Application

Compile the preceding source (Listing 9-5) by using the following command line:

```
C:\{working folder}>javac -d . DeleteNode.java
```

Run the application by using the following command:

```
C:\{working folder}>java database.DeleteNode 01022006 8
```

This will delete the record for the customer with an ID equal to 8 from the `Order01022006.xml` document in our database. Before you run the application, you may want to list the document content. You can do so by running the following XPath query on your `xindice` command line:

```
C:\xindice>xindice rd -c /db/StockBrokerage/DailyOrders -n Order01022006.xml
```

You will see the following output on your screen:

```
<?xml version="1.0"?>
<customers date="01022006">
...
  <customer ID="7">
      <name>John-Weigert</name>
      <order>IKJ</order>
      <quantity>20</quantity>
      <price>25.50</price>
  </customer>

  <customer ID="8">
    <name>Marilia-Oliver</name>
    <order>IKL</order>
    <quantity>87</quantity>
    <price>26.00</price>
  </customer>

  <customer ID="9">
    <name>Marilia-Oliver</name>
    <order>IKM</order>
    <quantity>90</quantity>
    <price>25.20</price>
  </customer>
...
</customers>
```

Now run the application. When the application runs successfully, you will see the following screen output:

```
Running Remove Query:
<xu:modifications version="1.0" xmlns:xu="http://www.xmldb.org/xupdate">
   <xu:remove select="/customers[@date='01022006']/customer[@ID=8]"/>
</xu:modifications>
```

Now, if you list the document content again by using the previous command line, you will notice that the record for the customer with ID equal to 8 has been removed from the document.

Next, you will study how to insert a record in the orders database.

Inserting a Node in the Orders Collection

You can insert a record in your document by constructing an `insert` XUpdate query. You will now insert an order record in our orders document. The XUpdate query for doing this is given here:

```
String xupdate = "<xu:modifications version=\"1.0\"" +
        "      xmlns:xu=\"http://www.xmldb.org/xupdate\">\n" +
        "   <xu:insert-after select=\"/customers/customer[@ID=17]\">\n" +
        "       <xu:element name=\"customer\">\n"+
        "           <xu:attribute name=\"id\">2</xu:attribute>\n"+
        "           <name>pradeep</name>\n"+
        "           <order>IBM</order>\n"+
        "           <quantity>50</quantity>\n"+
        "           <price>25.22</price>\n"+
        "       </xu:element>\n" +
        "    </xu:insert-after>\n"+
        "</xu:modifications>";
```

■ **Tip** The full source for the update node application is available in the file `Ch09\src\InsertNode.java` in the source download.

The XML document fragment generated by the preceding statement is as follows:

```
<xu:modifications version="1.0"      xmlns:xu="http://www.xmldb.org/xupdate">
  <xu:insert-after select="/customers/customer[@ID=17]">
    <xu:element name="customer">
      <xu:attribute name="id">2</xu:attribute>
      <name>pradeep</name>
      <order>IBM</order>
      <quantity>50</quantity>
      <price>25.22</price>
    </xu:element>
  </xu:insert-after>
</xu:modifications>
```

When you run the query by calling the `update` method on the query service object, the following XML fragment will be added after the record with the customer ID 17.

```
...
<customer id="2">
        <name>pradeep</name>
        <order>IBM</order>
        <quantity>50</quantity>
        <price>25.22</price>
</customer>
...
```

Note that the position for inserting the record is specified by the XPath expression `/customers/customer[@ID=17]` as a value of the `select` attribute.

Updating a Node in the Orders Collection

I will now discuss the construction of an XUpdate query for updating a specified record in our database. Imagine that we want to modify the order placed by the customer with ID equal to 1 on January 2, 2006. We will modify the stock code from an existing value of `MSFT` to a new value of `IKJ`. The code that constructs the XUpdate query is shown here:

```
// Construct an update query
String xupdate = "<xu:modifications version=\"1.0\"" +
        " xmlns:xu=\"http://www.xmldb.org/xupdate\">\n" +
        "     <xu:update select=" +
        "\"/customers[@date='01022006']/customer[@ID=1]/order\">\n"
        + "         IKJ" +
        "\n     </xu:update>\n" +
        "</xu:modifications>";
```

Tip The full source for the update node application is available in the file `Ch09\src\UpdateNode.java` in the source download.

The following XML document fragment is generated by the preceding code:

```
<xu:modifications version="1.0" xmlns:xu="http://www.xmldb.org/xupdate">
     <xu:update select="/customers[@date='01022006']/customer[@ID=1]/order">
         IKJ
     </xu:update>
</xu:modifications>
```

Note how the XPath expression selects the `order` element of customer ID equal to 1 from the January 2, 2006 document. The new value for the order node is set to `IKJ`.

As in the earlier cases, after constructing the query, we will execute it by calling the `update` method of the query service.

Performing Database Administration

Every database requires some sort of administration to keep it efficient while it is being accessed. This administration consists of organizing the database collection structure, adding/removing resources, creating indexes for faster access, backing up the data, and so on. In this section, you will learn about the various facilities available for a Xindice database administrator.

You use the `xindiceadmin` utility for administering the Xindice database. You have used this utility previously for adding/removing collections and more. In this section, I will summarize the functionalities provided by the `xindiceadmin` utility. The functionalities can be classified as management commands and actions, as listed here:

- Management commands
  - Collection management
  - Document management
- Actions
  - Indexer actions
  - XPath query actions
  - Miscellaneous actions

Most of the management commands have been discussed earlier and I will not repeat them here. I will discuss whatever I have not yet covered.

■ **Tip** You can refer to the Command Line Tools Reference guide for full details on the command syntax (`C:\<xindice installation folder>\docs\ToolsReference.html`).

Creating/Managing Indexes

An index improves database access performance. To create an index, you use the `add_indexer` command on the `xindiceadmin` utility. To illustrate how to add an index, you will create an index on the `order` element of our `DailyOrders` collection. The command to add such an index is as follows:

```
C:\xindice>xindiceadmin add_indexer -c /db/StockBrokerage/DailyOrders -n ➥
orderindex -p order
CREATED : orderindex
```

The `-c` switch specifies the collection context (which in this case is the `DailyOrders` subcollection in our `StockBrokerage` collection). The `-n` switch specifies the name of the index, and the `-p` switch specifies the pattern used to create an index.

You can also create indexes on the element attributes. For example, if you want to create an index on the ID attribute of the customer element, you would use a command as follows:

```
C:\xindice>xindiceadmin add_indexer -c /db/StockBrokerage/DailyOrders ➥
-n IDindex -p customer@ID
```

In this case, the pattern used is customer@ID.

Listing Indexes

You use the list_indexers (abbreviated li) command on the xindiceadmin to list the available indexes. The following screen output shows how to list the indexes in our DailyOrders collection:

```
C:\xindice>xindiceadmin li -c /db/StockBrokerage/DailyOrders
Indexes:

IDindex
orderindex

Total Indexes: 2
```

Deleting an Index

You delete an index by using the delete_indexer (abbreviated di) command on the xindiceadmin utility. To delete the ID index you just created, use the command shown in the following screen output:

```
C:\xindice>xindiceadmin di -c /db/StockBrokerage/DailyOrders -n IDindex
Continue deletion process? (y/n)
y
DELETED: IDindex
```

Backing Up Your Data

Every database administrator is concerned with the periodic backups of the database. The process of backing up the Xindice database is straightforward and simple. Simply copy the db folder along with its subfolders to any other storage device. Restoring the database requires the reverse process of copying from the backup folders to the db folder.

Exporting Data

Sometimes you may want to export the database contents to its native format. You do this by using the export command on the xindiceadmin utility. The screen output from running this command is shown here:

```
C:\xindice>xindiceadmin export -c /db/StockBrokerage -f c:\apress\Ch09\backup

Creating directory c:\apress\Ch09\backup\StockBrokerage
Extracting 0 files from /db/StockBrokerage
Creating directory c:\apress\Ch09\backup\StockBrokerage\DailyOrders
Extracting 10 files from /db/StockBrokerage/DailyOrders
Creating directory c:\apress\Ch09\backup\StockBrokerage\DailyQuotes
Extracting 0 files from /db/StockBrokerage/DailyQuotes
```

Here we export the contents of the `StockBrokerage` collection to the `backup` folder on the C drive. The utility creates an appropriate folder structure under the `backup` folder and copies all the documents in the subcollections to the appropriate subfolders. Each document is copied in its native format.

Importing Data

The data that you have exported earlier can be imported into the database by using the `import` command of `xindiceadmin`. You use the `import` command as follows:

```
C:\{xindice installation folder}>xindiceadmin import -c /db -f c:\apress\Ch09\import
```

This imports the data from the `import` folder and its subfolders into the `db` database. During the imports, the folder hierarchy is maintained in the database.

Shutting Down the Server

To shut down the Xindice server, you use the `shutdown` command shown here:

```
c:\xindice>xindiceadmin shutdown -c /db
```

Summary

XML databases provide an easy and powerful way to store your XML documents so they can be quickly searched later. The XML database follows a hierarchical storage structure analogous to the folder structures in your file system.

The XML documents stored in the database are organized into collections. You learned how to create collections, how to create subcollections, and how to delete collections and subcollections by using command-line utilities. You also learned how to add and remove documents to the collections by using command-line utilities.

In addition, you learned to perform all these activities through a program interface. You learned the XUpdate language syntax that allows you to write queries for updating the contents of the XML database.

Finally, you learned about database administration used to manage the database and its contents. You learned the techniques of backing up and restoring the database. You also learned to export the database content into its native format and to import the earlier exported content into the database again.