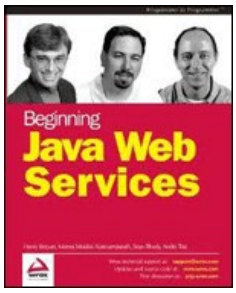


Beginning Java Web Services

by Henry Bequet

Chapter 2: Parsing Data Using XML

Overview

In the last chapter we had a whirlwind tour of web services. From this chapter onwards we will cover the individual technologies involved in implementing web services in further detail. We will focus on how XML is used in web services and the various aspects of XML that are relevant to web services. The topics that are covered in this chapter are as follows:

- **XML Namespaces**
  We will look at XML Namespaces that are used quite extensively in both **Web Service Description Language (WSDL)** documents that describe web services, and **Simple Object Access Protocol (SOAP)** envelopes that represent web service requests and responses.
- **XML Schema**
  We will look at XML Schema, which addresses many shortcomings that are associated with **Document Type Definitions (DTD),** traditionally used for constraining the contents of XML documents.
- **Processing XML documents**
  We will look at the various options available for producing and consuming XML documents. Here we will cover the **Document Object Model (DOM)** and the **Simple API for XML (SAX).** We will also cover the **Java API for XML Processing (JAXP)** for writing XML applications that are not tightly coupled to XML parser vendors.
- **XML Protocols**
  We will look at the various protocols available for using XML to represent remote procedure requests and responses. Since SOAP is the dominant protocol used in web services, we will cover that in more detail. We will also cover the simpler but less efficient XML-RPC protocol.

Important Please note that this chapter doesn't cover the basics of XML. It assumes basic knowledge of XML. For a more comprehensive coverage of XML please refer to *Professional Java XML* published by Wrox Press (ISBN 1-86100-285-8).

The Evolution of XML

In the past few years XML has evolved as the de facto standard for document markup with human readable tags. Since its advent, XML has found its way into a variety of applications such as:

- Configuration Information
- Publishing
- Electronic Data Interchange
- Voice Mail Systems
- Vector Graphics
- Remote Method Invocation
- Object Serialization

XML can be used to customize tags that are relevant to the domain we work on. As a result of this extensibility XML has been adopted by diverse industries for representing the data relevant to their domain.

What makes XML an integral element of enterprise computing is that XML documents are simple text documents with a platform-neutral way of representing data. An XML document produced by an application running on Microsoft Windows can easily be consumed by an application running on Sun Solaris.

XML is a descendant of the **Standard Generalized Markup Language (SGML),** which is a very powerful but complicated markup language. XML simplifies most of SGML's complexity and provides a powerful way of creating document markup. The XML 1.0 specification was made a Recommendation by the W3C in early 1998. Since then, it has evolved into one of the most powerful technologies in enterprise computing. Over the past few years many other complementary technologies have also evolved that extend the power of XML. These technologies include:

- **XSL Transformations**

  Used for transforming XML documents from one form to another
- **XSL Formatting Objects**

  Used for describing layout of viewable XML documents
- **XPath**

  Used for selecting nodes arbitrarily from XML documents

- **XLink**

  Used for linking XML documents in a manner similar to HTML hyperlinks

- **XML Base**

  Used in conjunction with XLink for defining a base URI for XML documents (this is similar to the functionality provided by the HTML base tag)

- **XML Schema**

  Used for defining rules for a valid XML document

- **XPointer**

  Used as an extension of XPath for pointing to arbitrary structures within an XML document

- **XML Query**

  Used for extracting data from XML documents

- **SOAP**

  Used for exchanging structured and typed information between applications in a decentralized, distributed environment using XML

- **XML Encryption**

  Used for encrypting specific portions of an XML document

- **XML Key Management**

  Used for describing the protocols for distributing and registering public keys

- **XML Signature**

  Used for digitally signing specific portions of an XML document

In the next few sections we will be covering those XML-related technologies that are relevant to web services. We can obtain all the XML-related specifications from http://www.w3c.org/XML/.

Processing XML Documents

In this section we will cover the various technologies for processing XML documents. Processing XML documents mainly involves:

- Creating XML documents from scratch
- Parsing XML data available from external sources to ensure the well formedness and validity of the XML content

The various examples that we will cover in this chapter will be built around a stock quote XML example that will be used throughout the book. The structure of the XML document is shown below:

```
<?xml version="1.0"?>

<stock_quotes>

  <!-- EDS -->
  <stock_quote>
    <symbol>EDS</symbol>
    <when><date>2002-6-21</date><time>13:33</time></when>
    <price type="ask" value="32.32"/>
    <price type="open" value="32.8"/>
    <price type="dayhigh" value="33.1"/>
    <price type="daylow" value="32.16"/>
    <change>+0.239</change><volume>67552600</volume>
  </stock_quote>

  <!-- Sun-->
  <stock_quote>
    <symbol>SUNW</symbol>
    <when><date>2002-6-21</date><time>13:33</time></when>
    <price type="ask" value="5.93"/>
    <price type="open" value="5.67"/>
    <price type="dayhigh" value="6.01"/>
    <price type="daylow" value="5.56"/>
```

```
      <change>+0.239</change>
      <volume>67552600</volume>
    </stock_quote>

    <!-- IBM-->
    <stock_quote>
      <symbol>IBM</symbol>
      <when><date>2002-6-21</date><time>13:33</time></when>
      <price type="ask" value="69.01"/>
      <price type="open" value="69.51"/>
      <price type="dayhigh" value="71.39"/>
      <price type="daylow" value="71.39"/>
      <change>+0.239</change>
      <volume>67552600</volume>
    </stock_quote>

    <!-- Microsoft -->
    <stock_quote>
      <symbol>MSFT</symbol>
      <when><date>2002-6-21</date><time>13:33</time></when>
      <price type="ask" value="51.25"/>
      <price type="open" value="51.31"/>
      <price type="dayhigh" value="52.79"/>
      <price type="daylow" value="50.64"/>
      <change>+0.239</change>
      <volume>67552600</volume>
    </stock_quote>

    <!-- DOW Jones -->
    <stock_quote>
      <symbol>^DJI</symbol>
      <when><date>2002-6-21</date><time>13:33</time></when>
      <price type="ask" value="8448.19"/>
      <price type="open" value="8540.47"/>
      <price type="dayhigh" value="8621.95"/>
      <price type="daylow" value="8448.19"/>
      <change>+0.239</change>
      <volume>67552600</volume>
    </stock_quote>

    <!-- NASDAQ -->
    <stock_quote>
      <symbol>^IXIC</symbol>
      <when><date>2002-6-21</date><time>13:33</time></when>
      <price type="ask" value="1360.62"/>
      <price type="open" value="1390.41"/>
      <price type="dayhigh" value="1395.29"/>
      <price type="daylow" value="1360.22"/>
      <change>+0.239</change>
      <volume>67552600</volume>
    </stock_quote>
    <! --Standard & Poor -->
    <stock_quote>
      <symbol>^GSPC</symbol>
      <when><date>2002-6-21</date><time>13:33</time></when>
      <price type="ask" value="884.58"/>
      <price type="open" value="905.36"/>
      <price type="dayhigh" value="907.84"/>
      <price type="daylow" value="884.46"/>
      <change>+0.239</change>
      <volume>67552600</volume>
    </stock_quote>

  </stock_quotes>
```

The XML document above defines the quotes for some well-known stocks and market indices. Note that in a real-world scenario, this document would come from a web site such as Reuters, Yahoo Finance, or Financial Times that provides online stock quotes.

Each quote shows the ticker symbol of the stock that is quoted, the date on which the quote was received, the ask price, open price, highest

and lowest prices for the day, change in price since open, and the volume of the stocks. We will have a look at how the stock quote XML can be created from scratch, how an external file that contains the stock quote XML can be parsed to check for validity and well formedness, and how an in-memory XML structure representing the `stock_quote` can be queried to find various information regarding the stock quotes.

The two prevalent APIs available for processing XML documents are:

- **DOM**
  DOM is a W3C recommendation for processing XML documents. The DOM API loads the whole XML document into memory as a tree structure and allows the manipulation of the structure of the document by adding, removing, and amending the nodes.
- **SAX**
  SAX is an event-driven approach for processing XML documents. The SAX API parses XML documents dynamically instead of loading the whole document into memory.

Both DOM and SAX define an API in terms of interfaces and exceptions. To use them in our applications, we need to have classes that implement these interfaces. Fortunately, there are high quality XML parsers available, which implement both SAX and DOM APIs. In this chapter, we will be using the Xerces parser. It can be obtained from http://xml.apache.org/xerces2-j/index.html.

Document Object Model

DOM is a W3C recommendation that provides an API for treating XML documents as a tree of objects. It loads an entire XML document into memory and allows us to manipulate the structure of the XML document by adding, removing, and amending the elements and/or attributes.

Evolution of DOM

The initial DOM specification was put forward to provide portability to HTML documents. This allowed various elements of an HTML document to be treated as part of an object tree.

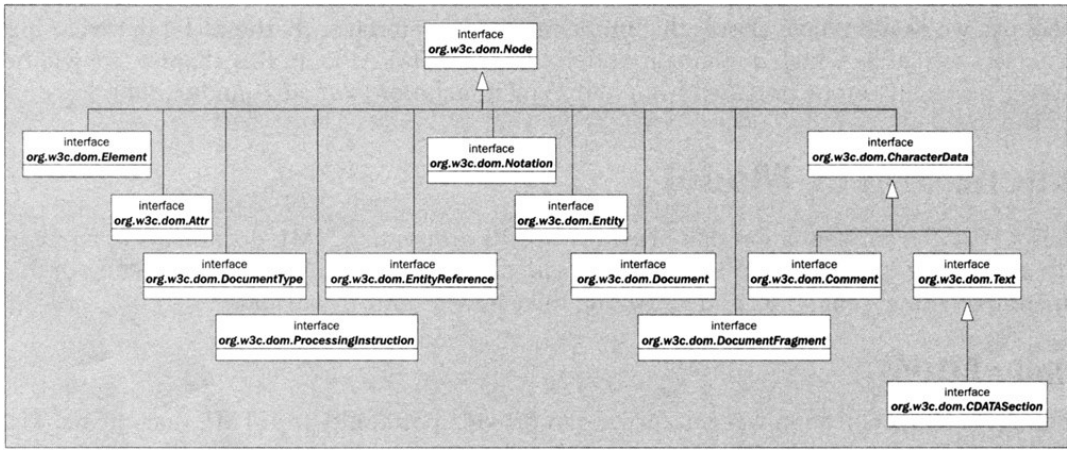
The list below illustrates the various levels of DOM that evolved over the past few years:

- **DOM Level 1**
  This became a W3C recommendation in mid 1998. It defined the basic interfaces that represent the various components of an XML document like the document itself, elements, attributes, PIs, and CDATA sections.
- **DOM Level 2**
  This became a recommendation in late 2000 and most importantly added support for XML namespaces. Level 2 also modularized DOM into:
  - **Core**
    This builds on Level 2, defining interfaces for manipulating the structure of XML documents
  - **Views**
    Covers presentation of a document in different types of views
  - **Events**
    Allows events and event listeners to be associated with XML nodes
  - **Style**
    Deals mainly with style sheets
  - **Traversal and Range**
    Deals with traversal of XML documents and the definition of ranges between arbitrary points in an XML document
- **DOM Level 3**
  This is currently a working draft and builds on Level 2. Main additions include loading and storing of XML documents from and to external sources.

In this section we will be covering the DOM Level 2 core API. However, the Xerces implementation provides full support to all the Level 2 modules, and also provides experimental support to the Level 3 features.

DOM Level 2 Core Architecture

The diagram below depicts the main interfaces in the DOM Level 2 core API:



In DOM, every component that makes up an XML document is treated as a **node.** Each of these nodes is represented as an interface that defines methods specific to that node. The different types of nodes include:

- `org.w3c.dom.Element`
- `org.w3c.dom.Attr`
- `org.w3c.dom.DocumentType`
- `org.w3c.dom.ProcessingInstruction`
- `org.w3c.dom.Notation`
- `org.w3c.dom.Entity`
- `org.w3c.dom.Document`
- `org.w3c.dom.DocumentFragment`
- `org.w3c.dom.Comment`
- `org.w3c.dom.Text`
- `org.w3c.dom.CDATASection`

The behavior common to all the node types is defined in the `org.wc.dom.Node` interface.

Node Interface

The `Node` interface defines methods that represent the common behavior of all types of nodes. The methods defined on this interface are mainly used for:

- Getting information regarding the node's parent node, child nodes, and sibling nodes
- Getting information about the node itself, such as the local name, attributes, and namespace URI
- Adding and removing of nodes and attributes

Document Interface

The `Document` interface extends the `Node` interface and represents the XML document itself. The most important aspect of this interface is that it acts as a factory for creating other types of nodes such as elements, attributes, texts, comments, CDATA sections, and so on. Nodes created by a document can be attached only to the owning document. However, this interface provides methods for importing nodes owned by other documents as well. Let us now look at an example.

Try It Out: Creating the Stock Quote Document

Now we will discuss a small example that uses the DOM API to build the stock quote XML document from scratch and then save it to a file called `stock_quote.xml`.

1. We will read the stock quotes data from a database table. The SQL scripts shown overleaf will create the required table and add the sample data:

```
CREATE TABLE Quotes(
    symbol VARCHAR(30),
    ask_price FLOAT,
```

```
    open_price FLOAT,
    dayhigh_price FLOAT,
    daylow_price FLOAT,
    change_price FLOAT,
    volume FLOAT
);

INSERT INTO Quotes VALUES
  ('EDS', 32.32, 32.80, 33.10, 32.16, -1.63, 5470100);
INSERT INTO Quotes VALUES
  ('SUNW', 5.93, 5.67, 6.01, 5.56, +0.239, 67552600);
INSERT INTO Quotes VALUES
  ('IBM', 69.01, 69.51, 71.39, 71.39, -1.99, 10428600);
INSERT INTO Quotes VALUES
  ('MSFT', 51.25, 51.31, 52.79, 50.64, -0.55, 48166900);
INSERT INTO Quotes VALUES
  ('^DJI', 8448.19, 8540.47, 8621.95, 8448.19, -94.29, -1);
INSERT INTO Quotes VALUES
  ('^IXIC', 1360.62, 1390.41, 1395.29, 1360.22, -36.63, -1);
INSERT INTO Quotes VALUES
  ('^GSPC', 884.58, 905.36, 907.84, 884.46, -21.46, -1);
```

2. We will then run the SQL scripts shown above in our database management system to create the `Quotes` table. We have used MySQL in this case.

   Note To learn more about MySQL please refer to *Beginning Databases with MySQL* from Wrox Press (ISBN 1-86100-692-6).

3. Download Xerces from http://xml.apache.org/dist/xerces-j/ and install it on your machine. (This should simply be a case of unzipping the files to a directory such as `C:\Xerces2`.)

4. Store the class shown below in a file called `StockCoreDOMGenerator.java`. (You can obtain the source code from http:\\www.wrox.com).

```
package com.wrox.jws.stockcore;

import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Attr;

import org.apache.xerces.dom.DocumentImpl;

import java.io.PrintWriter;
import java.io.FileWriter;

import java.sql.*;
import java.util.GregorianCalendar;

public class StockCoreDOMGenerator {
```

   This is the document that holds the stock quotes data:

```
private Document doc;
```

   This string array holds the various types of prices associated with the stock:

```
private static String TYPES[] = {"ask", "open", "dayhigh", "daylow"};
```

   This variable holds the name of the file to which the stock data is written:

```
private static final String STOCK_FILE = "stock_quote.xml";
```

   This interface contains the names of the elements and attributes used to build the stock quotes data:

```
private static interface Markup {

  public static final String STOCK_QUOTES = "stock_quotes";
  public static final String STOCK_QUOTE = "stock_quote";

  public static final String SYMBOL = "symbol";

  public static final String WHEN = "when";
  public static final String DATE = "date";
  public static final String TIME = "time";
```

```
    public static final String PRICE = "price";
    public static final String TYPE = "type";
    public static final String VALUE = "value";

    public static final String CHANGE = "change";
    public static final String VOLUME = "volume";

  }
```

The interface shown below contains the various properties required for connecting to the database. The driver and the **JDBC URL** we have used here are specific to **MySQL** database. If you are using a different **DBMS,** you will have to change this to match your **DBMS**:

```
  private static interface Database {

    public static final String URL =
                        "jdbc:mysql:/localhost:3306/Quotes";
    public static final String USER = "sa";
    public static final String PASSWD = "";
    public static final String SQL = "SELECT * FROM Quotes";
    public static final String DRIVER = "org.gjt.mm.mysql.Driver";

  }
```

The constructor creates the document and the root element:

```
  public StockCoreDOMGenerator() {
    doc = new DocumentImpl();

    Element root = doc.createElement(Markup.STOCK_QUOTES);
    doc.appendChild(root);
  }
```

This method adds the specified stock data to the **XML** document:

```
  private void addStock(String symbol, String quote[], String change,
      String volume) {

    GregorianCalendar cal = new GregorianCalendar();
    String date = cal.get(cal.YEAR) + "-" + cal.get(cal.MONTH) +
      "-" + cal.get(cal.DATE) ;
    String time = cal.get(cal.HOUR_OF_DAY) + ":" + cal.get(cal.MINUTE);

    Element root = doc.getDocumentElement();

    Element stockQuoteEl = doc.createElement(Markup.STOCK_QUOTE);
    root.appendChild(stockQuoteEl);

    Element symbolEl = doc.createElement(Markup.SYMBOL);
    symbolEl.appendChild(doc.createTextNode(Symbol));
    stockQuoteEl.appendChild(symbolEl);

    Element whenEl = doc.createElement(Markup.WHEN);
    Element dateEl = doc.createElement(Markup.DATE);
    dateEl.appendChild(doc.createTextNode(date));
    whenEl.appendChild(dateEl);
    Element timeEl = doc.createElement(Markup.TIME);
    timeEl.appendChild(doc.createTextNode(time));
    whenEl.appendChild(timeEl);
    stockQuoteEl.appendChild(whenEl);

    for(int i = 0; i < 4; i++) {
      Element priceEl = doc.createElement(Markup.PRICE);
      priceEl.setAttribute(Markup.TYPE, TYPES[i]);
      priceEl.setAttribute(Markup.VALUE, quote[i]);
      stockQuoteEl.appendChild(priceEl);
    }

    Element changeEl = doc.createElement(Markup.CHANGE);
    changeEl.appendChild(doc.createTextNode("+0.239"));
    stockQuoteEl.appendChild(changeEl);
    Element volumeEl = doc.createElement(Markup.VOLUME);
```

```
      volumeEl.appendChild(doc.createTextNode("67552600"));
      stockQuoteEl.appendChild(volumeEl);

    }
```

This method serializes and saves the **XML** document to an external file:

```
    private void saveDocument() throws Exception {
      PrintWriter writer = new PrintWriter(new FileWriter(STOCK_FILE));
      writer.println(((DocumentImpl)doc).saveXML(doc));
      writer.close();
    }
```

The `main()` method adds a set of stock quote data to the XML by reading the data from the database and saves it to an external file:

```
    public static void main(String args[]) throws Exception {
      StockCoreDOMGenerator generator = new StockCoreDOMGenerator();

      Class.forName(Database.DRIVER);
      Properties props = new Properties();
      props.put("user", Database.USER );
      props.put("password", Database.PASSWD );

      Connection con = DriverManager.getConnection(Database.URL, props);
      Statement stmt = con.createStatement();
      ResultSet res = stmt.executeQuery(Database.SQL);

      try {
        while(res.next()) {
          generator.addStock(res.getString(1), new String[] {res.getString(2),
            res.getString(3), res.getString(4), res.getString(5)},
            res.getString(6), res.getString(7));
        }
      } finally {
        if(res != null) res.close();
        if(stmt != null) stmt.close();
        if(con != null) con.close();
      }

      generator.saveDocument();

      System.out.println("Stock quotes saved successfully");

      }
    }
```

5. Set the classpath to include the `xercesImpl.jar` and `xml ParserAPI.jar` files available with Xerces distribution:

```
set classpath=%classpath%;C:\Xerces2\xercesImpl.jar;
              C:\Xerces2\xmlParserAPIs.jar;
```

   Here, `C:\Xerces2` is our Xerces installation directory.

6. Now compile the file `StockCoreDOMGenerator.java` (assuming we are in the `\Chp02\src` directory) by giving the following command:

```
javac -d ..\classes com\wrox\jws\stockcore\StockCoreDOMGenerator.java
```

7. Run the class. Please make sure that the database driver is available in the classpath when running the application. You will need to switch to the `\Chp02\classes` directory now in order for this to run:

```
java -classpath %classpath%;%mysql_home%\mm.mysql-2.0.4.bin.jar
     com.wrox.jws.stockcore.StockCoreDOMGenerator
```

   This will produce the following output in the command window:

```
Stock quotes saved successfully
```

How It Works

Now we will have a look at some of the key aspects of the class `StockCoreDOMGenerator`. Here we create an instance of the **XML** document by using the Xerces-specific implementation of the **DOM** `Document` interface:

```
doc = new DocumentImpl();
```

The document object is used to create the root element and it is attached to the document:

```
Element root = doc.createElement(Markup.STOCK_QUOTES);
doc.appendChild(root);
```

This method adds a quote to the stock quotes data. The method takes the symbol, various prices, and the net volume:

```
private void addStock(String symbol, String quote[], String change,
    String volume) {
  GregorianCalendar cal = new GregorianCalendar();
  String date = cal.get(cal.YEAR) + "-" cal.get(cal.MONTH) + "-"
    + cal.get(cal.DATE) ;
  String time = cal.get(cal.HOUR_OF_DAY) + ":" + cal.get(cal.MINUTE);
```

First the root element is retrieved from the document object:

```
Element root = doc.getDocumentElement();
```

Create the `stockQuoteEl` element and append it to the root element:

```
Element stockQuoteEl = doc.createElement(Markup.STOCK_QUOTE);
root.appendChild(stockQuoteEl);
```

Create the `symbolEl` element and append it to the `stockQuoteEl` element:

```
Element symbolEl = doc.createElement(Markup.SYMBOL);
symbolEl.appendChild(doc.createTextNode(symbol));
stockQuoteEl.appendChild(symbolEl);
```

Create the `whenEl` element:

```
Element whenEl = doc.createElement(Markup.WHEN);
```

Create the `dateEl` element and append it to the `whenEl` element:

```
Element dateEl = doc.createElement(Markup.DATE);
dateEl.appendChild(doc.createTextNode(date));
whenEl.appendChild(dateEl);
```

Create the `timeEl` element and append it to the `whenEl` element:

```
Element timeEl = doc.createElement(Markup.TIME);
timeEl.appendChild(doc.createTextNode(time));
whenEl.appendChild(timeEl);
```

Append the `whenEl` element to the `stockQuoteEl` element:

```
stockQuoteEl.appendChild(whenEl);
```

Append the various `priceEl` elements to the `stockQuoteEl` element:

```
for(int i = 0; i < 4; i++) {
  Element priceEl = doc.createElement(Markup.PRICE);
```

Add the `TYPE` and `VALUE` attributes to the `priceEl` element:

```
  priceEl.setAttribute(Markup.TYPE, TYPES[i]);
  priceEl.setAttribute(Markup.VALUE, quote[i]);
  stockQuoteEl.appendChild(priceEl);
}
```

Create the `changeEl` element and append it to the `stockQuoteEl` element:

```
Element changeEl = doc.createElement(Markup.CHANGE);
changeEl.appendChild(doc.createTextNode("+0.239"));
stockQuoteEl.appendChild(changeEl);
```

Create the `volumeEl` element and append it to the `stockQuoteEl` element:

```
Element volumeEl = doc.createElement(Markup.VOLUME);
volumeEl.appendChild(doc.createTextNode("67552600"));
stockQuoteEl.appendChild(volumeEl);
```

The following method uses the **DOM** Level 3 functionality provided by the Xerces implementation to serialize the document to an external file:

```
private void saveDocument() throws Exception {
```

Open a print writer to the file to which the **XML** data is written:

```
  PrintWriter writer = new PrintWriter(new FileWriter(STOCK_FILE));
```

Use the Xerces-specific document implementation to serialize and save the **XML** data to an external file:

```
writer.println(((DocumentImpl)doc).saveXML(doc));
writer.close();
```

Now we will have a look at the `main()` method. First, we construct an instance of the class:

```
StockCoreDOMGenerator generator = new StockCoreDOMGenerator();
```

Load the database driver and get a connection to the database:

```
Class.forName(Database.DRIVER);
Connection con = DriverManager.getConnection(Database.URL, props);
```

Create a SQL statement and execute the query:

```
Statement stmt = con.createStatement();
ResultSet res = stmt.executeQuery(Database.SQL);
```

Iterate through the resultset and call the method for adding stocks by passing the information read from the database:

```
while(res.next()) {
  generator.addStock(res.getString(1), new String[]{res.getString(2),
    res.getString(3), res.getString(4), res.getString(5) },
    res.getString(6), res.getString(7));
}
```

Then we close the database resources:

```
if(res != null) res.close();
if(stmt != null) stmt.close();
if(con != null) con.close();
```

Finally we store the **XML** document to an external file:

```
generator.saveDocument();

System.out.println("Stock quotes saved successfully");
```

In this example we created the stock quote **XML** from scratch using the DOM interfaces. In the following example we will use the Xerces **DOM** parser to parse the contents of the text file we generated in the last example to an in-memory **DOM** structure.

Try It Out: Parsing Stock Quote Data

Here, the parser will verify the contents of the text file for the well formedness constraints for **XML** documents. The text file we will parse will contain the stock quote for more than one symbol.

1. Store the contents of the class below to a file called `StockCore.java`:

```
package com.wrox.jws.stockcore;

import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Attr;
import org.w3c.dom.NodeList;

import org.apache.xerces.parsers.DOMParser;
import org.xml.sax.InputSource;

public class StockCore {
```

The string variable that contains the name of the file containing the stock quote **XML** data:

```
private static final String STOCK_FILE = "stock_quote.xml";

private Document doc;
public Document getDocument() { return doc; }
```

The constructor parses the **XML** document:

```
public StockCore() throws Exception {
   InputSource in = new InputSource(
     getClass().getClassLoader().getResourceAsStream(STOCK_FILE));

   DOMParser domParser = new DOMParser();
   domParser.parse(in);
   doc = domParser.getDocument();
```

```
    }
```

The `main()` method simply calls the constructor to parse the **`XML`** document:

```
    public static void main(String args[]) throws Exception {
      StockCore stockCore = new StockCore();
      System.out.println("Stock quotes loaded.");
    }
  }
```

2. Compile the file `StockCore.java` (again making sure the Xerces JAR files are in the classpath):

```
javac -d ..\classes com\wrox\jws\stockcore\StockCore.java
```

3. The `StockCoreDOMGenerator` class should have saved the file `stock_quote.xml` in the `\chp02\classes` directory. If not copy it there yourself.

4. Switch to the \chp02\classes directory and run the class:

```
java com.wrox.jws.stockcore.StockCore
```

This will produce the following output:

```
Stock quotes loaded
```

How It Works

Let's look at some of the key aspects of the `StockCore` class:

We first create an input source pointing to the file containing the stock quote data:

```
InputSource in =
  new InputSource(getClass().getResourceAsStream(STOCK_FILE));
```

Then we create the Xerces DOM parser:

```
DOMParser domParser = new DOMParser();
```

Finally we parse the document itself and store the reference to the parsed document:

```
domParser.parse(in);
doc = domParser.getDocument();
```

Try It Out: Examining the Contents of a DOM Structure

In this example, we will expand the `StockCore` class to add functionality for inspecting the contents of the stock quotes DOM structure. Here, the application will take the ticker symbol as a command-line argument and print the ask price if the symbol is available in the stock quotes data.

1. Add the following method to our current `StockCore.java` file. This is the new method that is added to get the ask price for a specified symbol:

```
  public String getQuote(String symbol) {
    Element root = doc.getDocumentElement();
    NodeList stockList = root.getElementsByTagName("stock_quote");

      for(int i = 0; i < stockList.getLength(); i++) {
      Element stockQuoteEl =(Element) stockList.item (i);
      Element symbolEl =
        (Element) stockQuoteEl.getElementsByTagName ("symbol") .item (0);
      if(!symbolEl.getFirstChild(). getNodeValue() .equals (symbol))
        continue;

      NodeList priceList = stockQuoteEl.getElementsByTagName ("price");

      for(int j = 0; j < priceList.getLength(); j++) {
        Element priceEl = (Element) priceList.item (0);
        if(priceEl.getAttribute ("type").equals ("ask"))
          return priceEl.getAttribute("value");
      }
    }
    return" ";

  }
```

2. We also need to modify the `main()` method so that it first parses the XML by calling the constructor and then calls the method to get the ask price for the symbol passed in as the commandline argument:

```
    public static void main(String args[])throws Exception {
      if (args. length != 1) {
        System.out.println ("Usage: java dom.StockCore <symbol>");
        System.exit (0);
    }

    StockCore stockCore = new StockCore();
    System.out.println ("The ask price for " + args [0] + " is " +
      stockCore.getQuote (args [0]));
    }
```

3. Recompile the file `StockCore.java` (with the Xerces JARs in the classpath):

```
javac -d ..\classes com\wrox\jws\stockcore\StockCore.java
```

4. Run the class:

```
java com.wrox.jws.stockcore.StockCore EDS
```

   This should produce the following output:

The ask price of EDS is 32.32.

How It Works

Now let us take a look at some of the key aspects of this new version of the class `StockCore` and see how it works.

The first step of our `getQuote()` method is to get the `root` element of the document:

```
Element root = doc.getDocumentElement();
```

The method `getElementsByTagname()` returns all the elements in the document with the specified tag name. `NodeList` is a DOM interface that represents a list of nodes. Use this method to get all the `stock_quote` elements in the document:

```
NodeList stockList = root.getElementsByTagName("stock_quote");

for(int i = 0; i < stockList.getLength(); i++) {
  Element stockQuoteEl = (Element) stockList.item(i);
```

Then we get the `symbol`, which is the child element of the `stock_quote` element:

```
  Element symbolEl =
   (Element) stockQuoteEl.getElementsByTagName("symbol").item(0);
```

The `getFirstChild()` method returns the first child of the current node. The `getNodeValue()` method returns the value of the node. The value of the node depends on the type of the node. In this case the first child of the `symbol` element is the text node and this will return the content of the `symbol` element. If this content doesn't match the passed `symbol`, process the next `stock_quote` element:

```
  if(!symbolEl.getFirstChild() .getNodeValue() .equals (symbol))
    continue;
```

Get the list of prices for the matched symbol:

```
  NodeList priceList =
    stockQuoteEl.getElmentsByTagName("price");

  for(int j = 0; j < priceList.getlength(); j++) {
    Element priceEl = (Element) priceList.item (0);
```

The `getAttribute()` method returns the value of the named attribute. In this case we return the value of the attribute called `value` of the `price` element, if the value of the `type` attribute is "ask":

```
    if(priceEl.getAttribute("type") .equals("ask"))
      return priceEl.getAttribute("value");
```

SAX

DOM is a powerful API for loading an XML document into memory, and inspecting and manipulating its contents. However, this is not very efficient for processing large documents or in cases where we are not interested in the contents of the whole document or where we just want to verify the validity of the document. This is where SAX comes into the picture. SAX is the acronym for **Simple API for XML** parsing.

Members of the `xml-dev` mailing list defined the SAX 1.0 API in mid 1998. The current release of SAX version 2.0 provides advanced features including support for namespaces, which is significantly different from version (1.0) and this section will be concentrating on the 2.0 API.

SAX provides an event-driven approach for parsing XML documents. SAX parsers parse XML documents sequentially and emit events

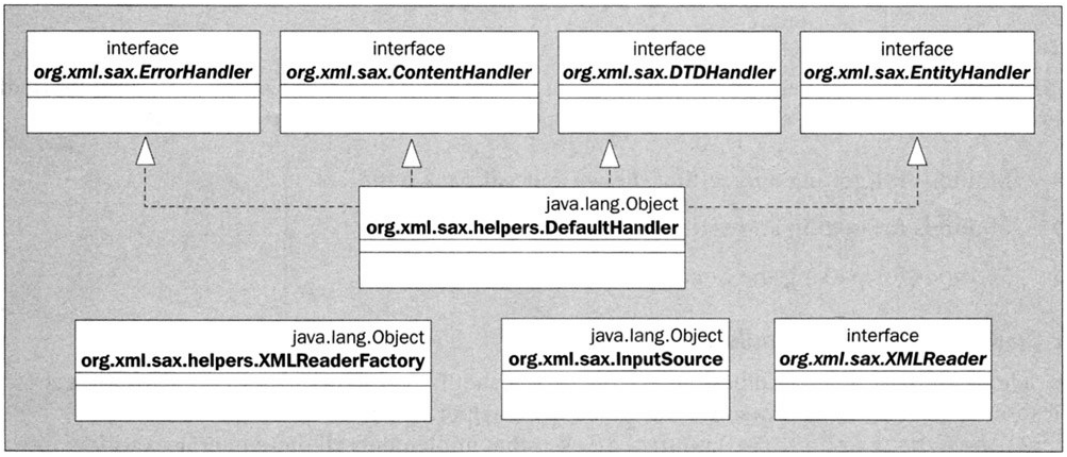
indicating start and end of the document, elements, text content etc. Applications interested in processing these events can register implemenatations of callback interfaces provided by the SAX API with the parser.

SAX Architecture

In SAX, the parser can be configured with a variety of callback handlers. When the parser scans an external stream that contains XML markup, it will report the various events involved to these callback handlers. These events include the following:

- Beginning of the document
- End of the document
- Namespace mapping
- Errors in well-formedness
- Validation errors
- Text data
- Start of an element
- End of an element

The SAX API provides interfaces that define the contract for these callback handlers. When we write XML applications that use SAX, we can write implementations for these interfaces and register them with a SAX parser. The diagram below depicts the important classes and interfaces in the SAX API:



SAX Event Handlers

In this section, we will see the various event handlers available in the SAX 2.0 API, which can be used for receiving relevant events associated with parsing XML documents:

- `org.xml.sax.ContentHandler`

  This interface is implemented by classes that need to be notified about events associated with the contents of the document. The events handled by this interface include:

  - Start and end of the document
  - Start and end of prefix mappings
  - Start and end of elements
  - Occurrence of character data, processing instructions, and ignorable whitespace

- `org.xml.sax.DTDHandler`

  This interface is implemented by classes that need to be notified about events associated with the contents of the DTD. The events handled by this interface include notation and unparsed entity declarations.

- `org.xml.sax.EntityResolver`

  Entity resolver can be used for resolving entities like external DTDs and so on.

- `org.xml.sax.ErrorHandler`

  This interface is implemented by classes that need to be notified about errors that occur during parsing the XML document. The events handled by this interface include errors, fatal errors, and warnings encountered during parsing the document.

Input Source

The SAX parser uses the class `org.xml.sax.InputSource` that can wrap a variety of input streams like:

- Byte stream
- Character stream
- A source identified by a public ID

SAX Parser

The SAX 2.0 parser class needs to implement the interface `org.xml.sax.XMLReader`. Normally, the parser vendor provides the class that implements this interface. This interface includes:

- Methods for getting and setting the various callback handlers
- Methods for getting and setting properties and features
- Methods for parsing the documents

Adapter Class for Callback Handlers

Earlier we had looked at four callback interfaces, which should be registered with the parser to get notification about the various parsing events. The SAX helper API provides the `org.xml.sax.helpers.DefaultHandler` class that implements all these interfaces with empty implementations. Hence, the only thing we need to do is to extend this class and override the required methods.

If we are interested only in the start end of the document, we may write an implementation as shown below:

```
import org.xml.sax.helpers.DefaultHandler;

public class MyDocumentHandler extends DefaultHandler
{
  public void startDocument() { System.out.println("Document started"); }

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

Try It Out: Parsing the Stock Quotes XML Using SAX

In this section we will write a version of `StockCore` class that will parse the `stock_quote.xml` file using the SAX API. Please note that the SAX parser will not create the document for us. Hence we will create the DOM document from the SAX events emitted by the parser during parsing.

1. Store the contents of the following class to a file called `StockCoreHandler.java`. This class is the handler, which we will register with our SAX parser to receive notifications about key events:

   ```
   package com.wrox.jws.stockcore.sax;

   import org.xml.sax.helpers.DefaultHandler;
   import org.xml.sax.Attributes;
   import org.xml.sax.SAXParseException;
   import org.xml.sax.SAXException;

   import org.w3c.dom.Document;
   import org.w3c.dom.Element;
   import org.apache.xerces.dom.DocumentImpl;

   import java.util.Stack;

   public class StockCoreHandler extends DefaultHandler {
   private Stack elements;
   ```

   This is the reference to the document that is created from the parser events:

   ```
   private Document doc;
   public Document getDocument() { return doc; }
   ```

   This method is called when the parser encounters character data:

   ```
   public void characters(char[] ch, int start, int length) {
   ```

```
      Element current =(Element) elements.peek();
      current.appendChild(doc.createTextNode(new String(ch, start, length)));
    }
```

This method is called when the parser encounters the end of the document:

```
    public void endDocument() {
      doc.appendChild((Element) elements.pop() );
    }
```

This method is called when the parser encounters the end of an element:

```
    public void endElement(String namespaceURI, String localName,
        String qName) {
      if (elements.size() != 1) elements.pop();
    }
```

This method is called when the parser encounters the start of the document:

```
    public void startDocument() {
    elements = new Stack();
    doc = new DocumentImpl();
    }
```

This method is called when the parser encounters the start of an element:

```
    public void startElement(String namespaceURI, String localName,
        String qName, Attributes atts) {
      Element child = doc.createElement (qName);

      for (int i = 0; i <atts.getLength(); i++) {
        child.setAttribute (atts.getQName (i), atts.getValue (i));
      }

      if (elements.empty()) {
        elements.push (child);
      } else {
        Element parent = (Element) elements.peek();
        parent.appendChild (child);
        elements.push (child);
      }
    }
```

The three methods shown below are called in the case of parsing errors:

```
    public void error(SAXParseException ex) throws SAXException {
      throw ex;
    }

    public void fatalError (SAXParseException ex) throws SAXException {
      throw ex;
    }

    public void warning (SAXParseException ex) throws SAXException {
      throw ex;
    }
  }
```

2. Store the class shown below into a file called `StockCore.java`. Most of it is similar to the DOM version, the major difference being the use of SAX parser instead of the DOM parser. Hence only the bits that are different from the DOM version are shown below:

```
    package com.wrox.jws.stockcore.sax;
```

Import the SAX parser:

```
    import org.apache.xerces.parsers.SAXParser;

    public class StockCore {
      ...
```

The constructor now uses the SAX parser to parse the document:

```
      public StockCore() throws Exception {
        InputSource in = new InputSource(
          getClass() .getClassLoader().getResourceAsStream(STOCK_FILE));
```

```
        SAXParser saxParser = new SAXParser();
        StockCoreHandler handler = new StockCoreHandler();
        saxParser.setContentHandler (handler);
        saxParser.setErrorHandler (handler);

        saxParser.parse (in);
        doc = handler.getDocument();
      }
      ...
    }
```

3. Compile the files `StockCoreHandler.java` and `StockCore.java` (we are in the `\Chp02\src` directory):

   ```
   javac -d ..\classes com\wrox\jws\stockcore\sax\*.java
   ```

4. Switch to the `\Chp02\classes` directory and run the class:

   ```
   java com.wrox.jws.stockcore.sax.StockCore EDS
   ```

   This will produce the same output as the DOM example:

The ask price of EDS is 32.32.

How It Works

In this example we use a SAX parser.Since the SAX parser doesn't generate the document for us, we create a callback handler called `StockCoreHandler` by extending the `org.xml.sax.helper.DefaultHandler` class and override the required callback methods to create the document from the information provided by the callback method arguments. We will use the same handler instance as our content and error handlers.

First we will have a look at the key aspects of the class `StockCoreHandler`. The `Stack` stores the elements when they are parsed by the SAX parser:

```
private Stack elements;
```

The `Document` object stores the parsed stock quotes data:

```
private Document doc;
public Document getDocument() { return doc; }
```

The parser calls this method when it encounters the start of the `Document`. Here we initialize the `Document` and the `Stack` that stores the `elements` that are parsed:

```
public void startDocument() {
  elements = new Stack();
  doc = new DocumentImpl();
}
```

The parser calls this method when it encounters the start of an element. It passes the namespace URI, local name, qualified name, and attributes associated with the element. Please note that namespaces, qualified name, local names, etc. will be covered in detail a little later. At this moment, just note that `qName` will give us the name of the element that is currently being parsed:

```
public void startElement(String namespaceURI, String localName,
    String qName, Attributes atts) {
```

Create the `Element` and set the attributes. The `Attributes` class represents the aggregation of all attributes present in the element:

```
  Element child = doc.createElement(qName);
  for (int i = 0; i < atts.getLength(); i++) {
    child.setAttribute (atts.getQName (i), atts.getValue (i));
}
```

If the `Stack` is non-empty, the parser is parsing the `Document` element. Hence push the `Document` element to the `Stack`:

```
  if(elements.empty()) {
    elements.push (child);
```

If the `Stack` is non-empty, we are at an intermediate element. Hence retrieve the last element from the stack and append the current element as a child to the last element:

```
  } else {
    Element parent =(Element) elements.peek();
    parent.appendChild (child);
    elements.push (child);
  }
```

The parser calls this method when it encounters character data. Here we use it to add the text content for the elements:

```
public void characters(char[] ch, int start, int length) {
  Element current =(Element)elements.peek();
  current.appendChild(doc.createTextNode(new String(ch, start, length)));
}
```

This method is called when the parser encounters the end of an element. We use this method to remove the last element that was added to the stack, unless there is only one element left in the stack. In that case it is the `Document` element:

```
public void endElement(String namespaceURI, String localName,
    String qName) {
  if(elements.size() != 1) elements.pop();
}
```

This method is called at the end of the document and we use it to retrieve the `Document` element from the stack and add it to the document:

```
public void endDocument() {
  doc.appendChild((Element)elements.pop());
}
```

The three methods shown below come from the `ErrorHandler` interface and are used to notify errors during parsing. These methods take an exception that represents the parsing error as their argument. We simply throw this exception back to abort parsing:

```
public void error(SAXParseException ex) throws SAXException {
  throw ex;
}

public void fatalError(SAXParseException ex) throws SAXException {
  throw ex;
}

public void warning(SAXParseException ex) throws SAXException {
  throw ex;
}
```

Now we will have a look at the changes made to the `StockCore` class, to use the SAX parser. The main changes are in the constructor and are shown below:

```
public StockCore() throws Exception {
  InputSource in = new InputSource(
  getClass() .getClassLoader() .getResourceAsStream(STOCK_FILE));
```

Instantiate the SAX parser:

```
SAXParser saxParser = new SAXParser();
```

Create the callback handler used for content and error handling:

```
StockCoreHandler handler = new StockCoreHandler();
saxParser.setContentHandler(handler);
saxParser.setErrorHandler(handler);
```

Parse the document and retrieve the document from the handler:

```
saxParser.parse(in);
doc = handler.getDocument();
```

Please note that we will also have to import the `SAXParser` class instead of the `DOMParser` class.

Factory-Based Approach for Creating Parsers

As already mentioned, the `XMLReader` is generally implemented by parser vendors. If we start hard-coding the class names of the vendor implementations in our applications, our applications will become tightly coupled to the vendor implementations. To avoid this, the SAX helper package provides a factory-based approach for creating parsers using `org.xml.parser.helper.XMLReaderFactory`.

This class provides two methods for writing vendor-neutral parser code. The first method is:

```
static XMLReader createXMLReader(String className) throws SAXException.
```

This method takes the name of the class that implements the interface `XMLReader` and returns an instance. Please note that the specified class should be available in the classpath.

The second method is:

```
static XMLReader createXMLReader() throws SAXException.
```

This method performs the following logic to return an `XMLReader` instance:

- The system property `org.xml.sax.driver` is checked for the class name

- The file `META-INF/services/org.xml.sax.driver` is checked in the available JAR files at run time
- Parser vendors may provide a default implementation

Features and Properties

Even though the SAX helper API provides a vendor-neutral way of creating parsers using a factory approach, the vendors may need to provide specific features and properties in their parser implementations.

This can be done in a vendor-neutral way by specifying them using the `setFeature()` and `setProperty()` method specified in the `XMLReader` interface. Both these methods will throw a `SAXNotRecognizedException` if the specified feature/property is not recognized and `SAXNotSupportedException` if it is recognized and not supported. The features are set using Boolean values true or false, whereas properties are set as Java objects.

The table below summarizes some of the general features supported by the Xerces parser:

Feature	Description	Default
http://xml.org/sax/features/namespaces	Performs Namespace processing	True
http://xml.org/sax/features/validation	Validates the document	False
http://apache.org/xml/features/validation/dynamic	Validates only if grammar is specified	False
http://apache.org/xml/features/validation/schema	Turns on schema validation	False
http://apache.org/xml/features/validation/schema--full- checking	Enables full schema grammar constraint	False
http://apache.org/xml/features/validation/schema/normal ized-value	Exposes normalized values for elements	True
http://xml.org/sax/features/external-general-entities	Includes external general entities	True
http://xml.org/sax/features/external-parameter-entities	Includes external parameter entities	True
http://apache.org/xml/features/validation/warn-on-duplicate- attdef	Warns on duplicate attribute declaration	True

The table below summarizes some of the properties supported by the Xerces parser:

Property	Description	Type
http://apache.org/xml/properties/schema/external-schemaLocation	Defines a list of schema locations for schemas with target namespace	String
http://apache.org/xml/properties/schema/external-noNamespace SchemaLocation	Defines a list of schema locations for schemas without target namespace	String

We will have a look at using features and properties in a parsing example in a later section on XML schemas for validating XML documents.

The Java API for XML Processing

In the examples we have seen so far, we have been using Xerces-specific classes. One drawback of using this approach in XML-based applications, is that at a later stage if we decide to change our parser vendor, we will have to change all the hard-coded class names in our application. JAXP offers a vendor-neutral approach for writing SAX and DOM applications. It provides a factory-based approach for creating DOM and SAX parsers and wrappers around vendor-specific DOM and SAX parsers encapsulating DOM and SAX parsers respectively. It also provides support for XSL transformations.

JAXP is part of both J2EE 1.3 and J2SE 1.4 and includes the following packages:

- `java.xml.parsers`

  Contains classes that provide a factory-based approach for DOM-and SAX-based XML processing
- `java.xml.transform`

  Contains generic transformation API
- `java.xml.transform.dom`

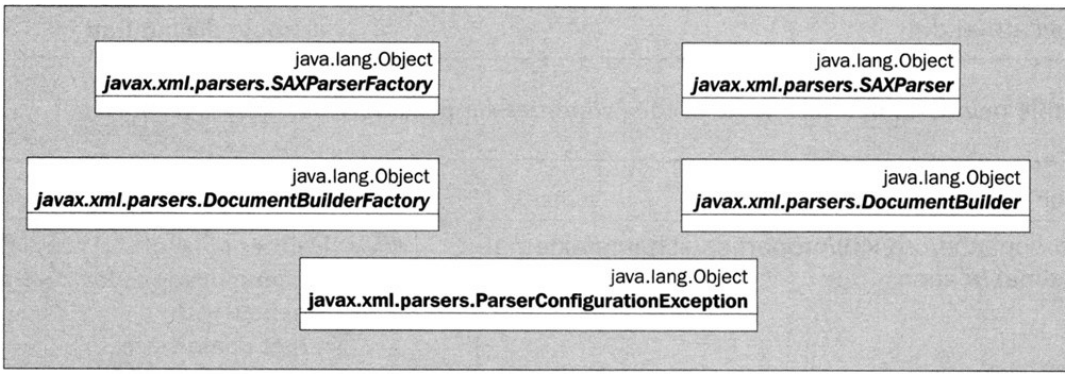
  Contains DOM-specific transformation APIs
- `java.xml.transform.sax`

  Contains SAX-specific transformation APIs
- `java.xml.tranform.stream`

  Contains stream-and URI-specific transformation API

JAXP Factory Classes for XML Processing

The diagram below depicts the classes provided by JAXP for writing vendor-neutral XML applications:



The `SAXParserFactory` class provides factory methods for creating `SAXParser` instances. The `SAXParser` class acts as a thin wrapper around the SAX `XMLReader` interface. The parser implementation to be used at run time is defined using the system property `javax.xml.parsers.SAXParserFactory` to specify the factory to be used.

This can also be specified in the `jaxp.properties` file in the `\lib` directory of the JRE installation or it may also use the JAR services API to look in the file `META-INF/services/javax.xml.parsers.SAXParserFactory`. If none of these things are specified the parser vendor may choose to use a default parser factory.

The `DocumentBuilderFactory` class provides factory methods for creating `DocumentBuilder` instances. The `DocumentBuilder` class provides methods for parsing XML documents into DOM trees, creating new XML documents, and so on. The parser implementation to be used at run time is defined using the system property `javax.xml.parsers.DocumentBuilderFactory` to specify the factory to use.

This can also be specified in the `jaxp.properties`, or it may also use the JAR services API to look in the file `META-INF/services/javax.xml.parsers.DocumentBuilderFactory`. If none of these things specified the parser vendor may choose to use a default parser factory.

Try It Out: Using JAXP with DOM StockCore

In this section, we will look at removing the Xerces-specific code from the DOM version of the `StockCore` class.

1. Take the `StockCore.java` file that we developed for the DOM parser example and copy it into a new directory for the JAXP examples. Make the following changes:

```
package com.wrox.jws.stockcore.jaxp;

import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Attr;
import org.w3c.dom.NodeList;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;

import org.xml.sax.InputSource;

public class StockCore {
```

   The constructor now uses the JAXP classes instead of the Xerces DOM parser:

```
public StockCore() throws Exception {
  InputSource in = new InputSource(
  getClass() .getClassLoader() .getResourceAsStream (STOCK_FILE));

    DocumentBuilder domParser =
      DocumentBuilderFactory.newInstance() .newDocumentBuilder();
    doc = domParser.parse (in);
}
```

2. Compile the new `StockCore.java` file:

```
javac -d ..\classes com\wrox\jws\stockcore\jaxp\StockCore.java
```

3. Switch directories and run the class:

```
java com.wrox.jws.stockcore.jaxp.StockCore EDS
```

This will produce the following output yet again:

The ask price of EDS is 32.32.

XML Namespaces

As we have seen in the last chapter, web services use XML to define data representing the requests sent by the clients to the web services and the responses the clients receive from web services. These XML documents are defined as SOAP envelopes, and information about these web services is published using the standard WSDL documents. These documents are also XML documents that define information like parameter types, return types, protocol, and transport bindings about the web services.

The SOAP documents that represent web services and the WSDL documents that describe web services, contain markup coming from a variety of sources. For example, a WSDL document contains markup related to description of web services, the encoding scheme used, and XML Schema for defining the types of the information that are exchanged using web services.

When a single document contains markup originating from more than one source, a potential problem called 'name collision' arises. A single document can contain markup vocabulary from more than one source, especially if we are using XML Schema to constrain the contents of the document. XML Schema is covered in detail in the next section.

Name collision occurs when two elements with same names but totally different meanings are present in the same XML document. This problem is not as severe for attributes since the scope of an attribute is always the enclosing element and an element can never have duplicate attributes. This problem of name conflicts is solved by the simple yet elegant solution of XML namespaces. In this section we will look at how XML solves the age-old problem of name conflicts using namespaces.

We shall illustrate this problem with the help of an example. In the XML document representing our stock quotes, each `stock_quote` element has four `price` elements. However, each `price` element has a different meaning, as they represent `ask, open, dayhigh`, and `daylow` prices. These four elements have the same names and the only way we can distinguish one from the other is by looking at the value of the `type` attribute.

Implementing Namespaces

Namespaces solve the problem of name ambiguity of elements and attributes by assigning **prefixes** to names. These prefixes are qualified to unique URIs. Namespaces allow us to take advantage of elements defined in other namespaces, and leverage that knowledge in our own XML documents. The ambiguity problem in the case of identical names is resolved by declaring that a particular element's (or attribute's) context is taken from a definition established in a separate namespace. We leverage that namespace by using the prefix format that qualifies the specified element.

The reason for using a combination of URIs and prefixes is that URIs are often very long and may contain characters that are not allowed in XML markup. Hence these URIs are mapped to shorter prefixes that are used to qualify the element and attribute names.

The following example shows how we can use XML namespaces to distinguish the price elements from one another, instead of using the `type` attribute:

```
<stock_quotes
  xmlns:ask="http://www.acme.com/Ask"
  xmlns:open="http://www.acme.com/Open"
  xmlns:dayHigh="http://www.acme.com/DayHigh"
  xmlns:dayLow="http://www.acme.com/DayLow">

  <!-- EDS -->
  <stock_quote>
    <symbol>EDS</symbol>
    <when>
      <date>7/18/2002</date>
      <time>3:12pm</time>
    </when>
    <ask:price value="32.32" />
    <open:price value="32.80" />
    <dayHigh:price value="33.10" />
    <dayLow:price value="32.16" />
    <change>-1.63</change>
    <volume>5470100</volume>
  </stock_quote>

...
<stock_quotes>
```

In the example shown above each `price` element is attached to a prefix qualified by a unique URI. By doing this we have removed the need of

using the `type` attribute to distinguish the business meaning of the `price` element.

Here the prefixes `ask, open, dayHigh`, and `dayLow` attached to the elements representing the same are qualified by the http://www.acme.com/Ask, http://www.acme.com/Open, http://www.acme.com/DayHigh, and http://www.acme.com/DayLow namespaces respectively. Namespaces are defined using the following syntax:

```
xmlns:<Namespace prefix>="<Namespace URI>"
```

Each namespace prefix defined in an XML document should be bound to exactly one URI. In the example above, for the element `<open:price>, open` is called the namespace prefix and `price` is called the local part. The complete name including the colon is called the **QName** or **qualified name.**

Assigning a namespace prefix to a URI is called **namespace binding**. This is done using the `xmlns` attribute. The scope of a namespace binding is the element within which the namespace binding is defined and all its child elements. Hence the prefixes in the above example can be used to qualify the `stock_quotes` element and all its child elements. We can declare the namespace binding in any element we like as shown below:

```
<stock_quotes
  xmlns:ask="http://www.acme.com/Ask"
  xmlns:open="http://www.acme.com/Open"
  xmlns:dayHigh="http://www.acme.com/DayHigh"
  xmlns:dayLow="http://www.acme.com/DayLow">
  <!-- EDS -->
  <stock_quote>
    <symbol>EDS</symbol>
    <dt:when xmlns:dt="http://www.acme.com/Dt">
      <dt:date>7/18/2002</dt:date>
      <dt:time>3:12pm</dt:time>
    </dt:when>
    <ask:price value="32.32" />
    <open:price value="32.80" />
    <dayHigh:price value="33.10" />
    <dayLow:price value="32.16" />
    <change>-1.63</change>
    <volume>5470100</volume>
  </stock_quote>

...
<stock_quotes>
```

Here the namespace prefix `dt` is declared in the scope of the `when` element and can be used to qualify only that specific instance of `when` element and its children - `date` and `time` elements. Please note that in the above example, none of the elements and attributes apart from the `price` elements is qualified by a namespace and hence they don't belong to any namespace. In XML applications of considerable size and complexity, it is recommended that the namespace bindings be defined in the root element to avoid confusion.

Default Namespaces

In an XML application where the majority of the markup comes from one source, prefixing all the markup elements can be tedious. To simplify this, we can define a default namespace for an element and its child elements. This is done by defining a namespace binding without the prefix for that element and not specifying any prefix for that element, and its children. However, default namespaces apply only for elements. Attributes without a prefix are not considered to be a part of any namespace.

The example below shows that all the elements in the stock quote XML belong to the default namespace http://www.acme.com/Stock apart from the `price` elements that are explicitly declared to belong to different namespaces:

```
<stock_quotes
  xmlns:ask="http://www.acme.com/Ask"
  xmlns:open="http://www.acme.com/Open"
  xmlns:dayHigh="http://www.acme.com/DayHigh"
  xmlns:dayLow="http://www.acme.com/DayLow"
  xmlns="http://www.acme.com/Stock">

  <!-- EDS -->
  <stock_quote>
    <symbol>EDS</symbol>
    <when>
      <date>7/18/2002</date>
      <time>3:12pm</time>

      </when>
    <ask:price value="32.32" />
    <open:price value="32.80" />
```

```
    <dayHigh:price value="33.10" />
    <dayLow:price value="32.16" />
    <change>-1.63</change>
    <volume>5470100</volume>
  </stock_quote>


...
<stock_quotes>
```

Now let's modify our `StockCore` example to use namespaces support.

Try It Out: Using Namespaces to Get the Ask Price

In the versions of the `StockCore` class that we have so far covered in this chapter, we have been getting all the price elements for the specified symbol and checking whether the value of the `type` attribute is `ask` before returning the value of the value attribute. In this section we will modify that code to use namespaces support provided by the DOM API instead.

1. First we need to create the namespace-qualified version of our `stock_quote.xml` file. We've already seen the changes we're going to be making in the above section on namespaces. For simplicity, we'll just modify the `stock_quote.xml` file and save it as `stock_quote_ns.xml`:

```
    <stock_quotes
      xmlns:ask="http://www.acme.com/Ask"

      <!-- EDS -->
      <stock_quote>
        <symbol>EDS</symbol>
        <when>
          <date>7/18/2002</date>
          <time>3:12pm</time>
        </when>
        <ask:price value="32.32" />
        <price value="32.80" />
        <price value="33.10" />
        <price value="32.16" />
        <change>-1.63</change>
        <volume>5470100</volume>
      </stock_quote>
```

   Change all the other ask price elements as appropriate.

2. Store the contents of the class shown below in a file called `StockCore.java`:

```
    package com.wrox.jws.stockcore.ns;

    import org.w3c.dom.Document;
    import org.w3c.dom.Element;
    import org.w3c.dom.Attr;
    import org.w3c.dom.NodeList;

    import javax.xml.parsers.DocumentBuilder;
    import javax.xml.parsers.DocumentBuilderFactory;

    import org.xml.sax.InputSource;

    public class StockCore {

      private static final String STOCK_FILE = "stock_quote_ns.xml";

      private Document doc;
      public Document getDocument getDocument() { return doc; }
```

   The constructor now makes the JAXP document-builder class namespace aware:

```
      public StockCore() throws Exception {

        InputSource in = new InputSource(
          getClass().getClassLoader().getResourceAsStream(STOCK_FILE));
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        factory.setNamespaceAware(true);

        DocumentBuilder domParser = factory.newDocumentBuilder();
```

```
        doc = domParser.parse(in);
      }
```

This method now uses the namespace support offered by the DOM API to get the value of the `ask` price:

```
   public String getQuote(String symbol) {

     Element root = doc.getDocumentElement();
     NodeList stockList = root.getElementsByTagName("stock_quote");

     for(int i = 0; i < stockList.getLength(); i++) {
       Element stockQuoteEl =(Element)stockList.item(i);
       Element symbolEl =
       (Element)stockQuoteEl.getElementsByTagName("symbol").item(0);

     if(!symbolEl.getFirstChild().getNodeValue().equals(symbol)) {
       continue;
     }

     Element priceEl = (Element)stockQuoteEl.getElementsByTagNameNS(
       "http://www.acme.com/Ask", "price").item(0);

     return priceEl.getAttribute("value");
   }

   return " ";
   }
 }
```

3. Compile the file:

   ```
   javac -d ..\classes com\wrox\jws\stockcore\ns\StockCore
   ```

4. Make sure we have `stock_quote_ns.xml` in the `\Chp02\classes` directory and run the class:

   ```
   java com.wrox.jws.stockcore.ns.StockCore EDS
   ```

   This will produce the familiar output:

The ask price of EDS is 32.32.

How It Works

By default the JAXP `DocumentBuilderFactory` creates DOM parsers that are not namespace aware. However, if we call `setNameSpaceAware` method on `DocumentBuilderFactory` with a Boolean value of `true`, it will create `DocumentBuilder` instances that are namespace aware:

```
factory.setNamespaceAware(true);
```

Here instead of listing through all the price elements, we select only those price elements belonging to the namespace http://www.acme.com/Ask using the method `getElementsByTagNameNS()`. This method takes the namespace URI and the local name of the element as arguments:

```
Element priceEl =(Element)stockQuoteEl.getElementsByTagNameNS(
  "http://www.acme.com/Ask", "price").item(0);
```

Now we will have a detailed look at XML Schema.

XML Schema

The contents of an XML document may be optionally constrained by a set of rules, defined either internally or externally, to the document. Traditionally, these rules were defined using DTD.

Shortcomings of DTD include:

- DTDs have very limited support for types
- DTDs have very limited support for namespaces
- DTDs use a non-XML syntax
- DTDs don't provide any mechanism for defining types and type extensions

This led to a far more powerful mechanism for constraining XML documents - XML Schema. XML Schema are now a W3C recommendation.

XML Schema is an XML application that can be used for defining the content model for XML applications. From a web services perspective, XML Schema is used extensively in WSDL documents for defining the types used in the web services. In this section, we will give a brief overview of XML Schema. Please note that a comprehensive coverage of XML Schema is beyond the scope of this chapter.

XML Schema in Practice

We will cover the various aspects of XML Schema by developing a schema to constrain our stock quotes XML document.

If we look at the content model of the elements in the XML document that we discussed earlier, we will find the following types of elements:

- **Elements that contain child elements and/or attributes**
  The example below shows an element that contain child elements:
  `<when><date>2002-6-21</date><time>13:33</time></when>`
  The example below shows an element that contains attributes:
  `<price type="daylow" value="32.16"/>`

- **Elements that contain only text contents**
  The snippet below shows an element that contains only text content:
  `<time>13:33</time>`

XML Schema can be used to represent constraints on the content model of elements in an XML document. Elements that contain child elements and/or attributes are defined in a Schema document using complex types. Elements that don't contain child elements and attributes are defined using simple types.

Before we delve into the details of defining an XML Schema, we will take a look at a schema document for our stock quotes XML. An XML document that complies with the rules defined in a schema document is referred to as an **instance document** of that schema:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      Schema for stock quotes.
    </xsd:documentation>
  </xsd:annotation>

  <xsd:element name="stock_quotes" type="StockQuotes"/>

  <xsd:complexType name="StockQuotes">
    <xsd:sequence>
      <xsd:element name="stock_quote" minOccurs="0" maxOccurs="unbounded">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="symbol" type="Symbol"/>
            <xsd:element name="when" type="When"/>
            <xsd:element name="price" type="Price" minOccurs="4"
              maxOccurs="4"/>
            <xsd:element name="change" type="xsd:double"/>
            <xsd:element name="volume" type="xsd:double"/>
          </xsd:Sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="When">
    <xsd:sequence>
      <xsd:element name="date" type="xsd:string"/>
      <xsd:element name="time" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="price">
    <xsd:attribute name="type" type="xsd:string"/>
    <xsd:attribute name="value" type="xsd:decimal"/>
  </xsd:complexType>

  <xsd:simpleType name="Symbol">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="EDS"/>
      <xsd:enumeration value="SUNW"/>
```

```
        <xsd:enumeration value="IBM"/>
        <xsd:enumeration value="MSFT"/>
        <xsd:enumeration value="^DJI"/>
        <xsd:enumeration value="^IXIC"/>
        <xsd:enumeration value="^GSPC"/>
      </xsd:restriction>
    </xsd:simpleType>

  </xsd:schema>
```

The schema shown above defines elements, attributes, and types. Elements and attributes defined in the schema are associated with a type. Elements can be one of the following types:

- A user-defined complex type like `When`
- A built-in simple type like `xsd:string`
- A user-defined simple type derived from a built-in simple type like `Symbol`

Attributes can either be built-in or user-defined simple types.

The markup elements peculiar to an XML Schema such as `element, attribute, sequence, simpleType, complexType, restriction, enumeration`, and so on belong to the namespace http://www.w3.org/2001/XMLSchema and are, by convention, qualified by the prefix `xsd`. In the section on namespaces we have seen that an XML document can contain markup vocabulary from more than one source. XML Schema documents are a classic example for this scenario.

Let's look at the different types in more detail.

Simple Types

XML Schema defines a variety of simple types that can be used for attributes or elements containing only text content. These types include:

- `string`: Character strings in XML
- `integer`: Integer values
- `positiveInteger`: Positive integer values
- `negativeInteger`: Negative integer values
- `short`: Short integer values
- `decimal`: Arbitrary-precision decimal numbers
- `float`: IEEE single-precision 32-bit floating point
- `double`: IEEE double-precision 64-bit floating point
- `boolean`: Data type to support binary logic; either true or false
- `date`: A calendar date like 1973-06-31
- `time`: An instance of time that recurs every day
- `ID`: Represents an XML ID attribute type
- `IDREF`: Represents an XML IDREF attribute type
- `ENTITY`: Represents an XML entity attribute types

Please refer to the XML Schema data types for a complete list of built-in simple types.

User-Defined Simple Types

In addition to the built-in simple types, we can define our own simple types that are derived from the built-in simple types. XML Schema provides a variety of mechanisms for restricting the values of built-in simple types for user-defined simple types. Please refer to the XML Schema specifications for a complete list of these mechanisms at http://www.w3.org/XML/Schema.

When a new simple type is defined, the base type is defined using the `restriction` element. Various elements are used within the `restriction` element for defining the restriction rules.

The example below defines a data type called `Symbol` that is derived from `xsd:string` for defining stock symbols restricted by a set of values:

```
<xsd:simpleType name="Symbol">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="EDS"/>
    <xsd:enumeration value="SUNW"/>
    <xsd:enumeration value="IBM"/>
    <xsd:enumeration value="MSFT"/>
    <xsd:enumeration value="^DJI"/>
    <xsd:enumeration value="^IXIC"/>
    <xsd:enumeration value="^GSPC"/>
  </xsd:restriction>
</xsd:simpleType>
```

The example above uses the enumeration facet to define that the content of the symbol element should be one among the pre-defined set of values.

Complex Types

Complex types are used to define content model for elements that contain child elements and/or attributes. The content model of an element defines the structure of its child nodes and the attributes it can and can't have. Complex types are defined using the XML Schema markup `complexType` as shown below:

```
<xsd:complexType name="when">
  <xsd:sequence>
    <xsd:element name="date" type="xsd:string"/>
    <xsd:element name="address" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>
```

Complex type definitions generally contain element and attribute declarations that make up the content model for that type. Elements are declared using the element `element`, and attributes using the element `attribute`. The content model of complex types can be both-either complex types or simple types. The schema above defines that the `When` element should have exactly one `date` element with text content and one `time` element that also contains only text content, strictly in that order.

Constraining Element Occurrence

The number of times an element may appear in a complex type is defined using the attributes `minOccurs` and `maxOccurs` in the type definition as shown below:

```
<xsd:complexType name="StockQuotes">
  <xsd:sequence>
    <xsd:element name="stock_quote" minOccurs="0" maxOccurs="unbounded">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="symbol" type="Symbol"/>
          <xsd:element name="when" type="When"/>
          <xsd:element name="price" type="Price" minOccurs="4"
            maxOccurs="4"/>
          <xsd:element name="change" type="xsd:double"/>
          <xsd:element name="volume" type="xsd:double"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>
```

The listing above states that the complex type `StockQuotes` may contain zero or more `stock_quote` elements and the `stock_quote` element should contain exactly four `price` elements.

The values for this `minOccurs` and `maxOccurs` may be a positive integer or the string `unbounded`. If neither the attributes is specified, the element should appear exactly once in the content model for the type. If both, `minOccurs` and `maxOccurs` are specified, the value of `minOccurs` should be less than or equal to that of `maxOccurs`.

If only `minOccurs` is specified, its value should be either 0 or 1 and similarly, if only `maxOccurs` is specified, its value should be greater than or equal to 1. However the value of `minOccurs` can be greater than one if it is specified with `maxOccurs`, but should not be greater than the value of the `maxOccurs` attribute.

The table below depicts the various possible combinations for the `minOccurs` and `maxOccurs` attributes:

minOccurs	maxOccurs	Result
1	1	Element must appear exactly once

2	unbounded	Element must appear at least two times
0	1	Element appear once
0	2	Element may be absent, or present once or twice
4	4	Element must appear exactly four times
0	0	Element must not appear

Constraining Attribute Occurrence

Attributes may appear either once or not at all. Attributes are declared with the `use` attribute to indicate whether they are `required`, `optional`, or `prohibited`. The `default` attribute can be used in attribute declaration to give a default value to the attribute when it is absent.

The example below states that the type of the attribute `type` is `string` and it is a required attribute. If the attribute is not present the schema processor will give the default value `ask`:

```
    <xsd:attribute name="type" type="xsd:string" use="required" default="ask"/>
```

We can use the `fixed` attribute in an attribute declaration to state that the value of the attribute should be fixed. The example below states that the `type` attribute should be present and its value should always be `ask`:

```
    <xsd:attribute name="type" type="xsd:string" use="required" fixed="ask"/>
```

The snippet below states that the type attribute shouldn't be present at all:

```
    <xsd:attribute name="type" type="xsd:string" use="prohibited"/>
```

Anonymous Data Types

XML Schema also allows types to be defined without associating a name with them. This is useful when an element can appear in only one context. The listing below shows the stock quotes type, but it doesn't define the type for the `stock_quote` type explicitly:

```
    <xsd:complexType name="StockQuotes">
```

The stock quotes type defines zero or more `stock_quote` element without explicitly specifying its type:

```
      <xsd:sequence>
        <xsd:element name="stock_quote" minOccurs=1"0" maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:sequence>
```

The `stock_quote` element should contain exactly one `symbol`, one `when`, four `price`, one `change`, and one `volume` elements, exactly in that order:

```
              <xsd:element name="symbol" type="Symbol"/>
              <xsd:element name="when" type="When"/>
              <xsd:element name="price" type="price" minOccurs="4"
                maxOccurs="4"/>
              <xsd:element name="change" type="xsd:double"/>
              <xsd:element name="volume" type="xsd:double"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
```

XML Schema and Namespaces

In the root element of an XML Schema document we may use the `targetNamespace` attribute to define the **target namespace,** a namespace that applies to all the types defined in the schema. In our example this includes `Symbol, When, StockQuotes`, etc.

A schema document can have only one target namespace. Target namespaces are important in validating an XML document against a schema. The listing below shows our original schema document with target namespace defined in it:

```
    <xsd:schema
      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      targetNamespace="http://www.acme.com"
      xmlns:stock="http//www.acme.com">

      <xsd:annotation>
        <xsd:documentation xml:lang="en">
          Schema for stock quotes.
        </xsd:documentation>
      </xsd:annotation>
```

```
    <xsd:element name="stock_quotes" type="stock:Quotes"/>

    <xsd:complexType name="StockQuotes">
      <xsd:sequence>
        <xsd:element name="stock_quote" minOccurs="0" maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="symbol" type="stock:Symbol"/>
              <xsd:element name="when" type="stock:When"/>
              <xsd:element name="price" type="stock:Price" minOccurs="4"
                maxOccurs="4"/>
              <xsd:element name="change" type="xsd:double"/>
              <xsd:element name="volume" type="xsd:double"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="When">
      <xsd:sequence>
        <xsd:element name="date" type="xsd:string"/>
        <xsd:element name="time" type="xsd:string"/>
      </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="Price">
      <xsd:attribute name="type" type="xsd:string"/>
      <xsd:attribute name="value" type="xsd:decimal"/>
    </xsd:complexType>

    <xsd:simpleType name="Symbol">
      <xsd:restriction base="xsd:string">
        <xsd:enumeration value="EDS"/>
        <xsd:enumeration value="SUNW"/>
        <xsd:enumeration value="IBM"/>
        <xsd:enumeration value="MSFT"/>
        <xsd:enumeration value="^DJI"/>
        <xsd:enumeration value="^IXIC"/>
        <xsd:enumeration value="^GSPC"/>
      </xsd:restriction>
    </xsd:simpleType>

  </xsd:schema>
```

Note that now all the new types defined in the schema document are qualified with the target namespace. Now, we will see how to validate an XML document with a schema by using the target namespace:

```
  stock: stock_quotes xmlns:stock="http://www.acme.com">

    <stock_quote>
      <symbol>EDS</symbol>
      <when><date>2002-6-21</date><time>13:33</time></when>
      <price type="ask" value="32.32"/>
      <price type="open" value="32.8"/>
      <price type="dayhigh" value="33.1"/>
      <price type="daylow" value="32.16"/>
      <change>+0.239</change><volume>67552600</volume>
    </stock_quote>
    ...

  </stock:stock_quotes>
```

In the instance document above, the namespace declaration for the root element matches the target namespace of the schema document. If an instance document needs to be valid according to a schema, the target namespace of the schema should be the same as the namespace for the elements in the instance document. The content of this document is validated against the schema as we shall seen soon.

Qualified and Unqualified Locals

In the instance document shown above, only the root element is qualified by the target namespace. If we want all the elements and attributes to be qualified by the namespace during validation, we can use the `elementFormDefault` and `attributeFormDefault` attributes in the schema definition:

```
<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.employee.com"
  xmlns:emp="http://www.employee.com"
  elementFormDefault="qualified"
  attributeFormDefault="qualified">
```

In this case all the elements and attributes should be explicitly qualified by the target namespace:

```
<stock:stock_quotes xmlns:stock="http://www.acme.com">

<stock:stock_quote>
  <stock:symbol>EDS</stock:symbol>
  <stock:when>
    <stock:date>2002-6-21</stock:date>
    <stock:time>13:33</stock:time>
  </stock:when>
  <stock:price stock:type="ask" stock:value="32.32"/>
  <stock:price stock:type="open" stock:value="32.8"/>
  <stock:price stock:type="dayhigh" stock:value="33.1"/>
  <stock:price stock:type="daylow" stock:value="32.16"/>
  <stock:change>+0.239</stock:change>
  <stock:volume>67552600</stock:volume>
</ stock:stock_quote>
...

</stock:stock_quotes>
```

Note that the default value for `elementFormDefault` and `attributeFormDefault` is `unqualified`. The `form` attribute can be used for overriding the global values defined for individual local elements and attribute declarations using the `elementFormDefault` and `attributeFormDefault`:

```
<attribute name="type" type="xsd:string" form="qualified"/>
```

Schema Definition in Instance Documents

Instance documents can provide applications with the location of the schema document to be used for validation using the `schemaLocation` attribute:

```
<emp:employees
  xmlns:emp="http://www.employee.com"
```

The namespace http://www.w3c.org/2001/XMLSchema-instance contains the `schemaLocation` attribute used to specify hints about the schema that should be used to validate the document:

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.employee.com
http://www.employee.com/Employee.xsd">
```

This attribute contains pairs of values. The first member of the pair is the target namespace and the second member is the physical location of the schema. When the schema documents don't have a target namespace their locations are defined using the attribute `noNamespaceSchemaLocation` as shown below:

```
<stock_quotes
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="stock_quote.xsd">
```

Advanced Concepts

The XML features that are not covered in the chapter include:

- Including schema documents in other schema documents
- Importing types from one schema document to another
- Deriving complex types from other complex types
- Defining groups of elements and attributes that can be reused by different complex types
- List types

- Identity constraints similar to DTD IDs and IDREFs
- Substitution groups for designating certain element declarations as substitutes for others

Note Please refer to *Professional XML Schema* from Wrox Press (ISBN-1-86100-547-4) for a comprehensive coverage of XML Schema.

Try It Out: Validating the Stock Quotes XML

In this section we will validate our stock quotes XML document with the schema we have developed in this section.

1. First, we need to specify the schema location in the document element of our XML document as shown below, and save it as `stock_quote_schema.xml`:

```
<stock_quotes
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xsi:noNamespaceSchemaLocation='stock_quote.xsd'>
```

2. Store the contents of the schema document from the section above in a file called `stock_quote.xsd`.

3. We will use a JAXP SAX implementation of the `StockCore` class this time. The listing below shows the relevant modifications made to the `StockCore` class:

```
package com.wrox.jws.stockcore.schema;

import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Attr;
import org.w3c.dom.NodeList;

import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;
import org.xml.sax.XMLReader;

import org.xml.sax.InputSource;
import org.apache.xerces.dom.DocumentImpl;

public class StockCore {
```

This interface defines the various schema constants used for validating the stock quotes document:

```
private static interface FeatureId {

  public static final String NAMESPACES =
    "http://xml.org/sax/features/namespaces";
  public static final String VALIDATION =
    "http://xml.org/sax/features/validation";
  public static final String SCHEMA_VALIDATION =
    "http://apache.org/xml/features/validation/schema";
  public static final String SCHEMA_FULL_CHECKING =
    "http://apache.org/xml/features/validation/schema-full-checking";

}

private static final String STOCK_FILE = "stock_quote_schema.xml";

private Document doc;
public Document getDocument() { return doc; }
```

The constructor now validates the stock quotes XML using the schema:

```
public StockCore() throws Exception {

  InputSource in = new InputSource(
    getClass().getClassLoader().getResourceAsStream(STOCK_FILE));

  SAXParser saxParser = SAXParserFactory.newInstance().newSAXParser();
  XMLReader reader = saxParser.getXMLReader();

  reader.setFeature(FeatureId.NAMESPACES, true);
  reader.setFeature(FeatureId.VALIDATION, true);
  reader.setFeature(FeatureId.SCHEMA_VALIDATION, true);
  reader.setFeature(FeatureId.SCHEMA_FULL_CHECKING, true);
```

```
      StockCoreHandler handler = new StockCoreHandler();
      reader.setContentHandler(handler);
      reader.setErrorHandler (handler);

      reader.parse (in);
      doc = handler.getDocument();
    }
```

4. We will also need a JAXP version of the `StockCoreHandler` class. This is very similar to the class we saw earlier when we first looked at SAX parsing:

```
    package com.wrox.jws.stockcore.schema;

    import org.xml.sax.helpers.DefaultHandler;
    import org.xml.sax.Attributes;
    import org.xml.sax.SAXParseException;
    import org.xml.sax.SAXException;

    import org.w3c.dom.Document;
    import org.w3c.dom.Element;

    import javax.xml.parsers.DocumentBuilder;
    import javax.xml.parsers.DocumentBuilderFactory;

    import java.util.Stack;

    public class StockCoreHandler extends DefaultHandler
```

   The only other change is that the `startDocument()` method now uses the JAXP calls to create the XML document instead of using the Xerces-specific document implementation:

```
    public void startDocument() {
      try {
        elements = new Stack();
        DocumentBuilder builder =
          DocumentBuilderFactory.newInstance().newDocumentBuilder();
        doc = builder.newDocument();
      } catch(Exception ex) {
        ex.printStackTrace();
      }
    }
```

5. Now compile the files `StockCoreHandler.java` and `StockCore.java`:

```
javac -d ..\classea com\wrox\jws\stockcore\schema\*.java
```

6. Make sure the files stock_quote.`xsd` and `stock_quote_schema.xml` are in the execution directory and run the class:

```
java com.wrox.jws.stockcore.schema.StockCore EDS
```

   This will produce the all too familiar output:

The ask price of EDS is 32.32.

How It Works

The `FeatureID` interface defines the SAX features to enable schema-based validation:

```
    private static interface FeatureId {
```

This feature makes the parser namespace aware:

```
    public static final String NAMESPACES =
      "http://xml.org/sax/features/namespaces";
```

This turns validation on:

```
    public static final String VALIDATION =
      "http://xml.org/sax/features/validation";
```

This turns schema validation on:

```
    public static final String SCHEMA_VALIDATION =
      "http://apache.org/xml/features/validation/schema";
```

This turns full compliance with schema features on:

```
public static final String SCHEMA_FULL_CHECKING =
  "http://apache.org/xml/features/validation/schema-full-checking";
```

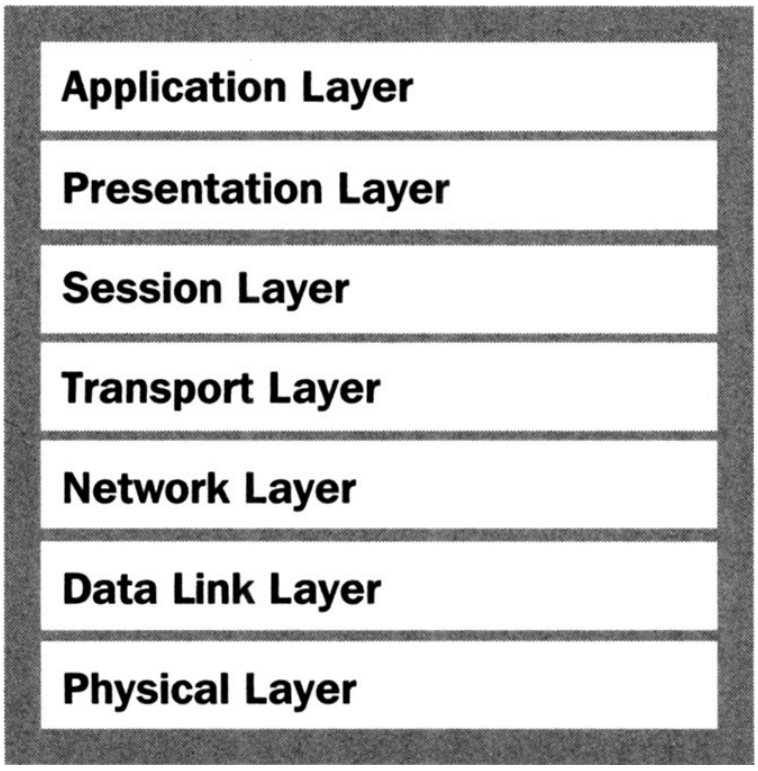
Now we will have a look at the changes we need in the constructor to enable schema validation. Please note that we can also make the parser namespace aware by calling the `setNamespaceAware()` method on `SAXParserFactory` by passing the Boolean value `true`. Similarly to set validation on we can call the `setValidating()` method passing the Boolean value `true`:

```
reader.setFeature(FeatureId.NAMESPACES, true);
reader.setFeature(FeatureId.VALIDATION, true);
reader.setFeature(FeatureId.SCHEMA_VALIDATION, true);
reader.setFeature(FeatureId.SCHEMA_FULL_CHECKING, true);

StockCoreHandler handler = new StockCoreHandler();
reader.setContentHandler(handler);
reader.setErrorHandler (handler);

reader.parse (in);
doc = handler.getDocument();
```

Now let us take a look at XML protocols.

XML Protocols

All protocols used for inter-process and inter-machine communication like HTTP, FTP, Telnet, TCP/IP, JRMP, and IIOP are based on the seven-layered **Open Systems Interconnect (OSI)** model determined by the **International Organization for Standardization (ISO).** The seven-layered model is depicted in the diagram shown below:



In the diagram above, the *Physical* layer takes care of how the bits are transmitted and the *Data Link* layer handles synchronization, blocking, and error and flow control. The network card driver provides the Physical and Data Link layers. The *Network* layer isolates the top layers from the Data Link layer and provides an abstraction of how a connection is made. Examples of network layer protocol are IP and X.25.

The *Transport* layer takes care of how the data is reliably transferred between two endpoints, and the *Session* layer provides an application-level abstraction of the connection. Protocols that handle these two layers include TCP and UDP. The *Presentation* layer decouples the applications from dependence on specific structured information and the *Application* layer interacts with the services. Examples of application-layer protocols are HTTP, FTP, Telnet, and so on.

The main goal of the layered architecture is to loosely couple the different aspects of data communication. When data is sent by the application layer it goes through the subsequent layers and each layer adds header information to the data frame. Similarly, when the physical layer receives data, each layer in the model strips off specific header information before it reaches the services. As the OSI model and the TCP/IP protocol stack were designed long before the advent of XML, in the TCP/IP implementation the application layer is a combination of

both application and presentation layers.

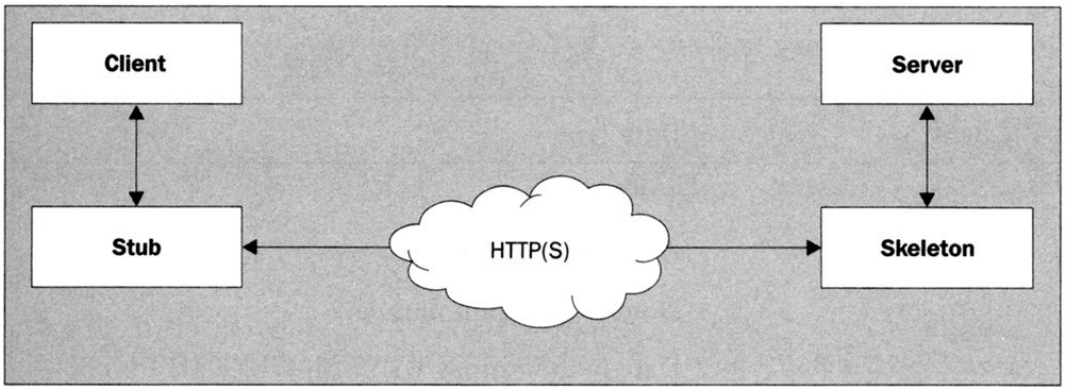
Even though people were excited with the advent of XML as a standard for representing structured data, it still didn't become a de facto standard for data interchange and RPC calls between applications. This is because XML doesn't specify any semantics for representing remote procedures and it doesn't specify a standard way of representing the data that is interchanged between applications.

XML-RPC

XML-RPC is a specification for representing remote procedure calls and results using XML in a simple yet powerful way. We can find the specification and information on the tools that implement the specification at http://www.xmlrpc.com. Note that XML-RPC is not an international standard, but it is increasingly becoming a de facto standard for performing remote procedure calls through an HTTP tunnel.

The most important thing about using XML for representing remote procedure calls is that it can sit on top of HTTP and pass through firewalls. This allows RPC calls to be made though firewalls, where most of the binary protocols like JRMP and IIOP won't work.

The basic way XML-RPC works is depicted in the diagram shown below:



1. The client calls the remote procedure
2. The client stub marshals the call on to HTTP as a message
3. The server skeleton receives and marshals the message
4. The method call is executed on the server skeleton and the result is sent back to the client

The XML-RPC calls are sent to the server as HTTP post requests. There are currently implementations available in Java, C/C++, COM, Perl, Python, Tcl, PHP, Lisp, BASIC, JavaScript, and ASP.

XML-RPC Request

The root element of the XML document that represents an XML-RPC request is represented by `methodCall`. The root element must have a mandatory child `methodName` that represents the name of the remote method and optional children representing the method parameters. The parameters are encapsulated in a `params` element that may contain zero or more `param` elements. An example is shown below:

```
<methodCall>
  <methodName>
    getSquareRoot
  </methodName>
  <params>
    <param>
      <value>
        <i4>64</i4>
      </value>
    </param>
  </params>
</methodCall>
```

The remote procedure `getSquareRoot()` is called with passing a parameter of type `i4`. The parameters and results passed back and forth during the remote procedure calls are encapsulated in `value` elements containing child elements representing the type of the value being passed with the element content as the value itself. The actual mapping of the XML-RPC types to the types within the implementation platform is specific to the implementation.

XML-RPC defines the following data types:

Element	Data type

<i4> or <int>	4 byte integer
<string>	A string of characters
<double>	Double-precision numbers
<dateTime.iso8601>	Date in the format YYYYMMDDTHH:mm:SS

Complex Types

XML-RPC also prvides complex types for serializing application-specific data types. The complex data types provided by XML-RPC are:

- Structures
- Arrays

Structures are represented by the element `struct` with embedded `member` elements to represent the members of the structure. The example below shows how a Java class may be represented by using structures:

```
public class Stock
{
  private String isin;
  private String sedol;
  private String shortCode;
  private double closingValue;
}
```

The `struct` elements may have other `struct` elements as nested members. The `struct` representation of the above class in XML-RPC may be represented as:

```
<struct>

  <member>
    <name>
      isin
    </name>
    <value>
      <string>
        GB12345
      </string>
    </value>
  </member>

  <member>
    <name>
      sedol
    </name>
    <value>
      <string>
        00897645
      </string>
    </value>
  </member>

  <member>
    <name>
      name
    </name>
    <value>
      <string>
        Chelsea Village
      </string>
    </value>
  </member>

  <member>
    <name>
      closingValue
    </name>
    <value>
      <double>
        1223.67
```

```
        </double>
      </value>
    </member>
  </struct>
```

How a specific instance of the class is serialized to its `struct` representation is specific to the implementation.

Arrays are represented in XML-RPC using `array` elements with a nested data element. The `data` element encapsulates all the array members as value elements. The example below represents an integer array:

```
  <array>
    <data>
      <value><i4>1</i4></value>
      <value><i4>2</i4></value>
      <value><i4>3</i4></value>
    </data>
  </array>
```

XML-RPC Response

XML-RPC responses are represented by XML documents with the root element as `methodResponse`. The content of the root element is governed by the result of the method call. For remote requests that don't cause any lower-level communication errors, the HTTP response status header should always be `200`. If the method call is successful, the response element will contain a list of values represented using the `value` elements that represents the result of the method call. An example is shown below:

```
  <methodResponse>
    <value><i4>567</i4></value>
    <value><string>Test</string></value>
  </methodResponse>
```

If the call fails due to business logic violation or an incorrect way of calling the method, the root element should contain a `fault` element. An example of the `fault` element is shown below:

```
  <fault>
    <value>
      <struct>

        <member>
          <name>faultCode</name>
          <value>
            <i4>4</i4>
          </value>
        </member>

        <member>
          <name>faultString</name>
          <value>
            <string>Too many parameters</string>
          </value>
        </member>

      </struct>
    </value>
  </fault>
```

Note that the `fault` element is represented as a structure with an integer element `faultCode`, representing an implementation-specific fault code and a string element `faultString`, representing the fault description.

XML-RPC Implementations

Many implementations providing the framework for writing client and server code use XML-RPC. One popular Java implementation is available at http://xml.apache.org/xmlrpc/. This framework provides both a client-side and server-side API. The server-side API supports remote procedures bound to both standalone servers and J2EE web containers.

SOAP

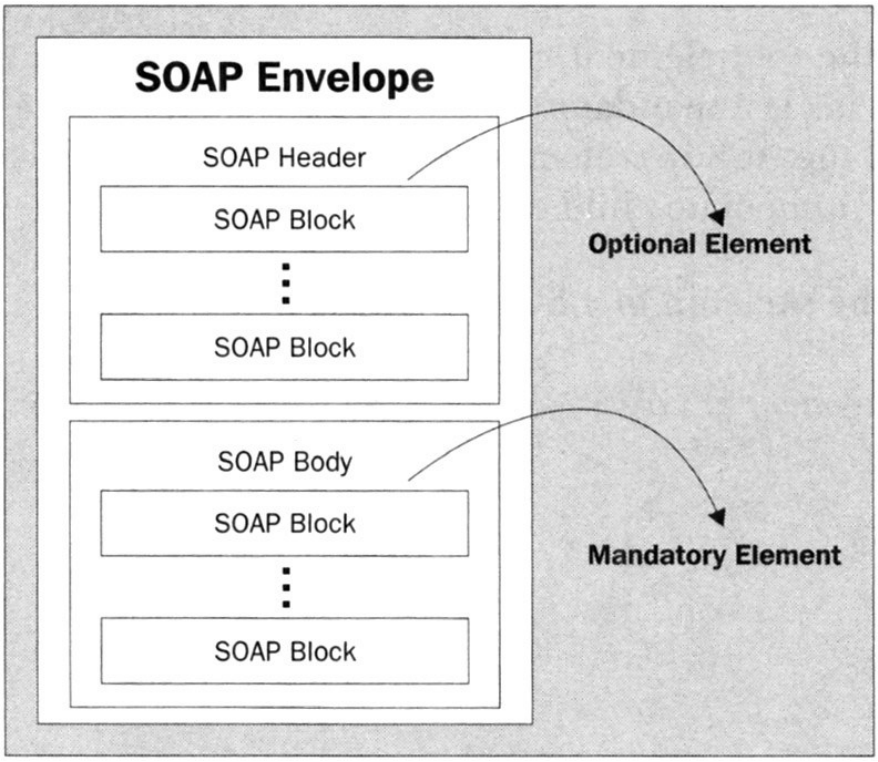
SOAP is a text-based protocol that uses XML for exchanging typed information in a decentralized environment. Please note that an exhaustive coverage of SOAP is beyond the scope of this chapter and book. In this section, we will take a whirlwind tour of SOAP envelopes, so that we can use them in our sample application.

Basic Concepts

The party that sends the SOAP message is called a SOAP sender and the one who receives it is called a SOAP receiver. The path a SOAP message takes from the initial sender to the ultimate receiver is called a message path. A message path will contain an initial sender, an ultimate receiver, and zero or more SOAP intermediaries. Not all the information in the envelope may be intended to the ultimate receiver. The entities that process messages according to the rules defined by SOAP are called SOAP nodes.

The atomic unit of information exchanged between SOAP nodes is called a SOAP message. A SOAP message is an XML document with the root element SOAP envelope. The SOAP envelope may have a SOAP header and has a mandatory SOAP body. The header and body may have multiple units of information encapsulated within them qualified by namespace URIs, called header blocks and body blocks respectively. A SOAP fault is a special body block that contains the fault information generated by a SOAP Node.

The diagram below shows the basic structure of a SOAP message:



Message Exchange Model

As we have already seen, SOAP nodes can be any one of the initial sender, ultimate receiver, or the intermediary. The roles performed by each node in the message path are defined by the SOAP actor name, specified as a URI. A SOAP node with an anonymous actor role is assumed as the ultimate receiver of the message. Each node in the message path is supposed to assume the roles of a special actor identified by the URI http://www.w3.org/2001/06/soap-envelope/actor/next. The SOAP specification doesn't define a way to link SOAP actor names to message routing.

The root elements of the SOAP header blocks may have an optional `actor` attribute qualified by the SOAP namespace to target them to specific SOAP nodes in the message path. Blocks without an explicit actor attribute are targeted for the ultimate receiver in the message path. The header block may also have an attribute called `mustUnderstand` belonging to the SOAP namespace, when set to 1, it mandates that the intended actor should process that block. If it can't process the block then it should raise a SOAP fault and stop doing any further processing.

Message Processing Rules

If header blocks are targeted towards specific SOAP nodes by using the `actor` attribute and have the `mustUnderstand` attribute set to 1, are not understood by the targeted nodes then such nodes should generate a SOAP fault and stop further processing. For intermediary SOAP nodes, the messages may need to be passed to the next node in the message path. These messages should retain all the header and body blocks except the header blocks targeted to the node that processed the messages.

SOAP Envelope

A SOAP envelope is an XML document that is exchanged between communicating peers. An envelope comprises the following:

- The document element of the SOAP envelope should have the local name `Envelope` and the namespace URI http://www.w3.org/2001/06/soap-envelope.
- This element may have an optional child element with local name as `Header` and the same namespace URI. Please note that, if this

element is present it should be the first immediate child of the root element, and may be used for adding information to the envelope without prior agreement with the peers that are invloved in the interchange.

- The next child of the root element must have the local name as `Body` and the same namespace URI. This is a mandatory element and should be the second immediate child of the root element if the `Header` element is present. If the `Header` element is absent this should be the first immediate child of the root element.

The snippet below shows the skeleton of a SOAP message:

```
<env:Envelope xmlns:env="http://www.w3.org/2001/06/soap-envelope">
  <env:Header>
  ...
  </env:Header>
  <env:Body>
  ...
  </env:Body>
</env:Envelope>
```

The `Header` and the `Body` elements may contain elements from different namespaces, for example, our application-specific data that we intend to exchange.

SOAP Header

SOAP header is the first immediate child element of the envelope. All immediate children of the header are called header blocks. Header blocks are used for extending the messages in a decentralized way by adding functionalities like authentication, transaction management, and so on.

All `header` elements must be namespace qualified. The header blocks may have the `actor` and `mustUnderstand` attributes to identify and mandate the SOAP node that processes the message. An example of a header attribute with `mustUnderstand` attribute set to `1` is shown below. The block is intended for the final receiver because the `actor` attribute is missing:

```
<env:Header>
  <authentication xmlns="http://myDomain/app" env:mustUnderstand="1">
    <user>Meeraj</user>
    <password>******</password>
  </authentication>
</env:Header>
```

In the above example the prefix, `env`, belongs to the SOAP envelope namespace.

SOAP Body

SOAP body is used for encapsulating the mandatory information intended for the ultimate receiver of the message. A SOAP `body` element should be the first immediate child of the envelope if the header is absent and the second immediate child if the header is present. The SOAP `body` element may have multiple immediate children called body blocks, which are treated as separated entities. SOAP body content is used for marshaling RPC calls, exchanging messaging information, and so on. The example shows SOAP envelope with both header and body elements defined:

```
<env:envelope xmlns:env="http://www.w3.org/2001/06/soap-envelope">
  <env:Header>
    <auth:authentication xmlns:auth="http://myDomain/auth">
      <auth:user>Meeraj</auth:user>
      <auth:password>******</auth:password>
    </auth:authentication>
  </env:header>
  <env:body>
    <stock:getStockQuoteResponse
      xmlns:stock="http://www.acme.com/service">
      <stock_quote>
        <symbol>EDS</symbol>
        .......
      </stock_quote>
    </stock:getStockQuoteResponse >
  </env:body>
</env:envelope>
```

SOAP Fault

SOAP fault, if present, should be the first body block in the envelope and should appear only once in the envelope. The fault block is used for conveying error and status information in processing the messages. The fault block will have the following child elements:

- `faultcode`
  This will carry a code for the fault. The SOAP specification defines a small set of standard SOAP fault codes.

- `faultstring`
  This gives a more detail description of the message.
- `faultactor`
  This defines the SOAP node that caused the fault.
- `detail`
  This can be used for providing details of the fault like stack traces and so on.

The things that we have not covered in this section about SOAP are SOAP encoding, transport protocol binding, and RPC conventions. However, now we have enough grounding in SOAP to write simple web services, which accept and send SOAP envelopes.

The listing below shows an example SOAP fault:

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Body>
    <SOAP-ENV:Fault>
      <faultcode>SOAP-ENV:Server</faultcode>
        <faultstring>Server Error</faultstring>
```

We can use the detail element to pass application specific error messages. Some of the Java implementations use this to pass stack traces:

```
        <detail>
          <e:myfaultdetails xmlns:e="Some-URI">
            <message>
              My application didn't work
            </message>
            <errorcode>
              1001
            </errorcode>
          </e:myfautdetails>
        </detail>
      </SOAP-ENV:Fault>
    </SOAP-ENV:Body>
  </SOAP-ENV:Envelope>
```

Summary

We stated this chapter by covering the evolution of XML over the past few years. Then we looked at the processing of XML documents – both generation and parsing. We studied DOM and SAX; they are the two prevalent APIs for processing XML documents. We looked at generating an XML document from the data stored in a database. Then we went on to discuss using XML namespaces for avoiding name collisions.

XML Schemas were discussed to illustrate how to constrain the contents of an XML document. We then covered the various XML protocols available for exchanging structured and typed information between applications; here we covered both XML-RPC and SOAP.

Throughout the chapter we have been using the stock quotes data to illustrate how to create, parse, and validate XML documents in our various examples. This XML example will also be used in the subsequent chapters to illustrate the various aspects of web services.