

## CHAPTER

## 8

# XML Processing and Data Binding with Java APIs

Java and XML are known to be a perfect fit for building portable business applications, where Java provides the portability of code that can be executed on various platforms and XML provides the portability of data that can be processed on diverse platforms. These two technologies can be ported from one platform to another without any effort, and in addition, Java applications can talk to non-Java applications by using XML as the communication protocol. The first generation of Java XML APIs dealt with the parsing of XML covering industry standards like SAX and DOM. By high demand from the developer community, other standards followed such as XML transformations and XML data binding. These emerging XML technologies are a key solution in the Web services arena, where parsing is used for processing XML messages and binding provides an object view of the XML data.

This chapter provides an overview of Java API for XML processing and data binding, focusing on the following:

- Overview of XML and its complementing standards
- Simple API for XML
- Document Object Model (DOM)
- The JAXP processing model and its features

## 314 Chapter 8

---

- Using JAXP in Web services development
- Using Java XML binding

The specification of Java API for XML processing is currently at version 1.2 and provides support for XML parsing and XML transformation. Also, the specification for XML data binding API is at version 1.0. These two specifications are very important because they set the fundamentals for XML processing and are essential in understanding the processing as well as benefits of using XML. Before describing the APIs, let's look at the basics of XML.

### Extensible Markup Language (XML) Basics

---

XML is an ASCII-based structured meta-data language that has been widely adopted in the industry. Currently, it is adopted in many areas such as security (for example, SAML and XACML), meta data (for example, XML Schema and TopicMaps), and presentation (for example, XHTML and XSLT).

XML was created in 1996 and embodied by the W3C group since early 1998. It is a derivative of the well-known SGML markup language, which has been around for a long time but has not gotten as much acceptance as XML due to its complexity. XML was created as an extensible way to represent data. It is considered extensible because it does not define a standard and well-defined set of tags (such as HTML) but rather provides the capability to create custom tags. It has the flexibility to define complex data structures in a modular way, thus promoting clarity and consistency.

HTML has tags `<>`, also referred to as markup, that have a specific meaning. When interpreted by a browser, the HTML tags have a particular presentation-oriented function within the document. For example, the tag `<Body>` marks the beginning of a page. Anything that is put inside the `<Body>` tag is rendered by the browser and displayed on the Web page. The `<Body>` tag also contains a closing tag, which is represented by the `</Body>` tag. In XML, the same set of tags could have hundreds of different meanings. For example, consider an XML file that describes the characteristics of a person. The Person tag contains a Body tag that will contain the weight and height of a person. The following is an XML representation of such a description:

```
<?xml version="1.0" ?>
<Friends>
  <Person>
    <Name>Jane Doe</Name>
    <Age>21</Age>
    <Body>
```

---

**XML Processing and Data Binding with Java APIs 315**

---

```
<Weight Unit="lbs">126</Weight>
<Height Unit="inches">62</Height>
</Body>
</Person>
<Person>
  <Name>John Doe</Name>
  <Age>26</Age>
  <Body>
    <Weight Unit="kg">80</Weight>
    <Height Unit="meters">1.67</Height>
  </Body>
</Person>
</Friends>
```

The document starts with a prolog, which is a processing instruction statement identifying the XML document and the version of the document. The XML structure represents data about two people, each identified with the `<person>` tag. Within the person tag, there is a description of the body characteristics of each person such as weight and height. Body characteristics can have different types of measuring units such as kilograms (kg) or pounds (lbs). This is a representation of hierarchical data where the `<Name>`, `<Age>`, and `<Body>` tags are enclosed within the scope of the `<Person>` tag. This makes XML very extensible, where the creator of the XML determines what meaning and content the markup must have.

XML is currently being used in various areas of enterprise computing. One area that all J2EE developers are familiar with is the area of the deployment descriptors, which is used for the configuration of the J2EE components hosted in an application server. Some application servers use XML for storing their setup, configuration, and deployment information. XML also is known to be a perfect solution for integration with legacy systems, because it is a platform and vendor-neutral solution.

Note that XML as a meta language, in most circumstances, is very easy to understand. It only uses ASCII encoding, thus making it readable with simple text editors. By looking at the previous example, one could easily interpret what is meant by the XML structure. To put things into perspective, the following is an equivalent comma-delimited data file representing the equivalent data:

```
Rima P, 18,100, 54
Urszula M, 21, 126, 62
Slawomir S, 45, 80, 1.55
Robert S, 26, 90, 1.67
```

In this particular scenario, commas are used to delimit the data, which was a very frequently used option prior to the standardization of XML. Just by looking at this sample, you will be hard pressed to determine what

## 316 Chapter 8

---

each entry means. There is no explanation of the data in an intuitive way. Also, the mixing of units into one single file is impossible, because there is no indication of the type of unit being used. Even though it's possible to use these types of data files, they are extremely difficult to maintain because they are very difficult to understand and validate. In fact, they can cause a maintenance nightmare.

XML provides a structured and well-defined syntax that enables data to be defined in a uniform way. This syntax is not very hard to learn, but understanding the different concepts is important. By understanding all of these concepts, a developer will be able to use various tools that will aid in XML development.

Before starting with the API discussion, let's look at the very basics of XML.

### XML Syntax

This section will discuss some terminology associated with XML that describes XML-specific syntax and what it means.

#### *XML Naming Conventions*

Naming has to be respected for the XML document to be well formed. Blank spaces are not permitted in XML names. A name must start with an alphabetical letter (A to Z or a to z) or an underscore (\_). It then can be followed with more letters, digits [0 to 9], underscores, hyphens (-), periods (.), and colons (:). Although a colon is permitted, it is mostly used when a document uses namespaces (see the section titled *Namespaces* that follows). Names are also case-sensitive in an XML document, therefore `<product>` and `<Product>` are considered to be two different elements. It is up to the developer to choose whether the structure is only in lowercase, uppercase, or mixed. For example, the following XML structure would not be valid due to the case mixing of the product:

```
<product>
  <id>1234</id>
  <price>19.99</price>
</Product>
```

The following is the correct product representation because both of the product elements are of the same case:

```
<product>
  <id>1234</id>
  <price>19.99</price>
</product>
```

## Prolog

Prolog is the declaration statement that identifies the document as an XML document. It is the first line in an XML file. It identifies the version of the XML specification used, the encoding being used, and whether it is standalone. The prolog is not necessary, but it is a good practice to use it in declarations for internationalization and future extensions set by the W3C. The version attribute is mandatory where the other two are optional. The following is a sample of a prolog:

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

or

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes" ?>
```

Some additional and optional attributes can be provided, including the following:

**Encoding.** Identifies the character set used for encoding the data.

**Standalone.** Identifies whether the source accesses external data sources.

## Root

A root is the topmost element in an XML document. For a document to be syntactically correct, it must contain only one root element.

The following is not a correct XML document because it does not contain a single root element:

```
<?xml version="1.0"?>
<person1>
  <name>john</name>
</person1>
<person2>
  <name>jane</name>
</person2>
```

The following is a correct version of the same document with the employee element as the root.

```
<?xml version="1.0"?>
<Employee>
  <Person1>
    <Name>john</Name>
  </Person1>
```

## 318 Chapter 8

---

```
<Person2>
  <Name>jane</Name>
</Person2>
<Employees>
```

### ***Processing Instructions***

Processing instructions (PIs) are used for providing information to the XML processing application. Things like scripts could be embedded in XML documents for extra processing. A prolog is considered to be a processing instruction but is reserved for XML standards. Processing instructions usually are used in special applications to perform special tasks.

The following is an example of a processing instruction:

```
<?target instructions?>
```

where the following are indicated:

**target.** It is the name of the application that will be performing the processing.

**instruction.** String containing information that is passed to the target application. An example of a PI that is seen in the majority of XML documents is the prolog:

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

### ***Comments***

Comments are used for documenting parts of an XML structure. Comments are defined using the same syntax as HTML. The following is an example of a comment:

```
<!-- this is a comment -->
```

### ***Tags***

A tag is the element markup identified by the angle brackets. Each tag must have a start tag and a close tag. An XML file that contains a closing tag for every tag in a nested form is considered to be a well-formed file. A tag is considered to be an empty tag if it stands by itself without any attributes. An empty tag serves the purpose of an identifier in a XML structure.

For example,

```
<ListItem Product_No="210020" Quantity="4000"/>
```

can be equivalently represented as

```
<ListItem>
  <Product_No>210020</Product_No>
  <Quantity>4000</Quantity>
</ListItem>
```

The first example shows how the attributes can be used in a tag. This also is an example of an empty tag where no data is present. It simply defines the name and age and closes itself with the slash (/).

A number of things need to be kept in mind when an XML structure is designed. For example, when data is very large, it makes more sense to use elements rather than attributes for clarity. Data that contains various HTML or formatting tags also should be defined as an element. On the other hand, if the data is short and does not change very often, defining it as an attribute might be the right approach to take.

## ***Elements***

An element is the data delimited by a start tag and an end tag. It is the building block used for creating XML documents. An XML document essentially is composed of many elements. The topmost element is called the root element. All elements that are directly under the root are referred to as child elements of the root element. In the following code, the root element is the `Catalog` element and the child elements of the root elements are the `CatalogId`, `Product`, and `EuroProducts` elements. These three elements are considered to be siblings in relation to one another, and the root element is considered to be an ancestor of the three siblings. This tree structure can span multiple levels, nesting much deeper than we are able to show in the following simple example code:

```
<!-- Catalog Is a Start or Root element -->
<Catalog>
  <CatalogId Id='123456' />
  <!--Product element contains children elements -->
  <Product>
    <!-- Id element contains character data -->
    <Id>1234</Id>
    <!-- Price contains attribute currency -->
    <Price currency='USD'>199.99</Price>
```

## 320 Chapter 8

---

```
</Product>
<!-- Empty element -->
<EuroProducts/>
<!-- Closing element -->
</Catalog>
```

### Attributes

Attributes provide additional information about an element. They have a key and value pair that identifies the different attribute. Many attributes can exist in one element. If attributes are used, the XML document can have a reduced number of tags, as shown in the following code:

```
<Price currency="USD">12.99</Price>
<Price currency="CND">21.99</Price>
```

In the code, the `Price` attribute specifies the currency type for the enclosed data. Attributes also apply to all of the elements that are nested within the element holding the attribute. The following example shows how the currency attribute applies to different scotch brands:

```
<Price currency="CND">
  <Scotch>
    <Name>Lagavulin</Name>
    <Value>49.99</Value>
  </Scotch>
  <Scotch>
    <Name>Talisker</Name>
    <Value>54.99</Name>
  </Scotch>
</Price>
<Price currency="USD">
  <Scotch>
    <Name>Cardhu</Name>
    <Value>29.99</Value>
  </Scotch>
</Price>
```

The *Lagavulin* and *Talisker* brands have CND currency, where the *Cardhu* brand holds the USD price.

### Entities

Entities are variables used to define common text or shortcuts to text. Table 8.1 shows the common ones used in XML specification. For example, the less-than sign (<) can be interpreted as the beginning of a tag; it is therefore important to make use of entities in these situations. Entities are interpreted and expanded at parsing time.

**Table 8.1** Entities Supported in XML Specification

ENTITY	CHARACTER
&lt;	<
&gt;	>
&amp;	&
&quot;	"
&apos;	'

The following is a sample of an XML structure representing a purchase order; it includes most of the concepts discussed in this section:

```

<!-- Prolog -->
<?xml version="1.0"?>
<!-- Root element -->
<PurchaseOrder>
  <Header>
    <PO_Number>2123536673005</PO_Number>
    <Date>02/22/2002</Date>
    <Customer_No>0002232</Customer_No>
    <!-- This Is the shipping address -->
    <Address>
      <Street1>233 St-John Blvd</Street1>
      <Street2>Building A42</Street2>
      <City>Boston</City>
      <State>MA</State>
      <Zip>03054</Zip>
      <Country>USA</Country>
    </Address>
    <!-- This Is the payment Information -->
    <PaymentInfo>
      <Type>Visa</Type>
      <Number>0323235664664564</Number>
      <Expires>02/2004</Expires>
      <Owner>John Doe</Owner>
    </PaymentInfo>
  </Header>
  <!-- The following section contains a list of -->
  <!-- ordered Items -->
  <Products/>
  <LineItem type="Software">
    <Product_No>21112</Product_No>
    <Quantity>250</Quantity>
  </LineItem>
  <LineItem type="Software">

```

## 322 Chapter 8

```

        <Product_No>343432</Product_No>
        <Quantity>1000</Quantity>
    </LineItem>
    <LineItem type="Hardware">
        <Product_No>210020</Product_No>
        <Quantity>4000</Quantity>
    </LineItem>
</PurchaseOrder>

```

This XML structure is a representation of a purchase order. It starts with the XML prolog, followed by the root element of the structure called `PurchaseOrder`. `PurchaseOrder` has child elements, starting with `Header`, which contain the buyer information followed by many `LineItem` elements that contain the product number and quantity of the ordered products. Between the `Header` and first `LineItem` is an empty tag called `Products`. This empty tag serves as a delimiter between the header and the line items. Each `LineItem` contains the type of product that it represents. For example, `Product_No 21112` is of type software.

### Namespaces

Namespaces in an XML document are used to prevent naming collisions within same or different XML documents. The namespace syntax enables a prefix definition and an associated URI/URL to exist. By specifying the URL, the namespace becomes a unique identifier. A URL is usually combined with a prefix to make the different elements distinguishable from each other. The URL does not refer to any particular file or directory on the Web, it simply acts as a unique association or label for the defined namespace. The XML namespaces specification indicates that each XML element is in a namespace. If the namespaces are not explicitly defined, the XML elements are considered to reside in a *default namespace*.

Consider the following example XML structure where the `<Name>` tag is found in two distinct places:

```

Buyer.xml
<!-- This Is a fragment of the xml file -->
<Buyer>
  <Name>Urszula M</Name>
  <email>urszulam@acme.com</email>
</Buyer>
<!-- This Is a fragment of the xml file -->
<Product>

  <Id>090902343</Id>
  <Name>Foo</Name>
</Product>

```

## XML Processing and Data Binding with Java APIs 323

This is a simple example, which calls for namespace support to avoid conflicts when both documents are used together. The most common conflicts arise when multiple XML documents use identical tags that have different meanings. `<Name>` is a child of both the product and buyer elements. The parser must understand to which `<Name>` tag the application is referring. Having said that, the syntax using namespaces specification will convert the `<Name>` tags into something less ambiguous, such as `<Person:Name>` and `<Item:Name>`. Namespaces can be found in XML-related documents, schema documents, and XSL stylesheets. The following sample shows the distinction of both tags:

```
<!-- Buyer.xml provides Information about the buyer -->
<PersonInfo:Buyer xmlns:PersonInfo=
"http://www.acme-computers.com/warehouse/personinfo/">
  <PersonInfo:Name>Robert S</Name>
  <PersonInfo:email>roberts@acme.com</email>
</PersonInfo:Buyer>

<!-- Catalog.xml listing of all Items available for sale -->
<Catalog:Product xmlns:Catalog=
"http://www.acme-computers.com/warehouse/catalog/"
xmlns="http://www.acme-computers.com/warehouse/default/">
  <!-- uses default namespace -->
  <Header>
    <LastUpdated>05/20/2001</LastUpdated>
  </Header>
  <Catalog:Item>
    <Catalog:Id>090902343</Catalog:Id>
    <Catalog:Name>Futsji</Catalog:Name>
  </Catalog:Item>
  <Catalog:Item>
    <Catalog:Id>123242343</Catalog:Id>
    <Catalog:Name>Sony</Catalog:Name>
  </Catalog:Item>
</Catalog:Product>
```

Element collision is prevented by placing prefixes in front of each XML element. The Header element of the catalog XML document does not use the Catalog namespace, instead, it uses the default namespace without any prefixes.

### Validation of XML Documents

Before the parser processes a document, it is checked for well-formedness. A well-formed document is a document in which every tag has an

## 324 Chapter 8

---

equivalent closing tag meaning; it conforms to the XML specification. A document that is well formed may not necessarily be valid. A valid document is a document that conforms to certain constraints defined in a schema definition. Validity is used for checking whether a document conforms to certain standards agreed upon by collaborating parties (for example, two businesses conducting the exchange of computer parts). These two businesses must provide the data in such a way that they both understand what is represented and what is meant by it.

The following example demonstrates a structure that is not well formed:

```
<!-- Not well-formed document -->
<?xml version="1.0"?>
<Employees>
  <Person>Jane</Person>
  <Age>21
</Employee>
</Age>
```

In the previous example, the `Age` and `Employee` elements are not properly nested. The order in which the paired tags are opened and closed is very important for a document to be considered well formed.

This error is corrected in the following example, in which the two elements are properly nested:

```
<!-- Well-formed document -->
<?xml version="1.0"?>
<Employee>
  <Person>Jane</Person>
  <Age>21</Age>
</Employee>
```

Well-formedness is very important because it enables the parser to process the XML document in a more efficient way.

In order for validity to be checked, a definition document must be provided to define what the document is allowed to have as tags and attributes and the type of elements that should be present within a particular tag.

Consider a simple example in which the `Product` element contains a type defining the type of product that a company is offering. The company offers two types of products (hardware or software). Suppose that its `Product` element is defined as follows:

```
<Product type="hardware">
  <Name>Generic Mouse</Name>
  <Price>19.99</Price>
```

```
</Product>
<Product type="service">
  <Name>Upgrade Services</Name>
  <Price>100.00</Price>
</Product>
```

In the previous code, a new type called `service` is introduced. In this case, this document would not be considered valid, because the receiving party would not know what the additional type means.

The following sections describe the different standards used for creating XML schemas for validating XML documents. Document Type Definitions (DTDs) were among the first specifications for validating XML data. The newest generation of validation standards is based on the XML Schema Definition, which is a more complete feature set that enables developers to define restrictions using XML syntax.

### ***Document Type Definition***

A Document Type Definition (DTD)—commonly known as a DOCTYPE—is a document containing the element restrictions an XML data document must follow in order to be considered *valid*. A DTD can be defined within the XML document or saved in an external file with a `.dtd` extension.

XML elements are declared with an element declaration using the following syntax:

```
<!ELEMENT element-name (element-content)>
```

Element-name is the XML element definition. Element-content defines the type of element. It defines whether the element is a data type or a compound type consisting of other elements and data. Various element types can be defined in an element, among which are the following:

**EMPTY.** Empty tag.

**#CDATA.** Character data; should not be parsed by the XML parser.

**#PCDATA.** Parsed character data; parsed by the XML parser. If elements are declared in `#PCDATA`, then these elements also must be defined.

**ANY.** Any content.

If a DTD is defined inside the XML, the declaration is included in the DOCTYPE construct. The following examples show how to define DTDs in

## 326 Chapter 8

both internal and external ways. These examples demonstrate how to define an element `Product` with children sequences of `Id` and `Price`:

```
<?xml version="1.0"?>
<!DOCTYPE Product [
  <!ELEMENT Product (Id,Price)>
  <!ELEMENT Id (#PCDATA)>
  <!ELEMENT Price (#PCDATA)>
]>
<Product>
  <Id>3124090231</Id>
  <Price>49.99</Price>
</Product>
```

The following example is an equivalent DTD, but it is defined in an external file called `Product.dtd`:

```
<?xml version="1.0"?>
  <!ELEMENT Product (Id, Price)>
  <!ELEMENT Id (#PCDATA)>
  <!ELEMENT Price (#PCDATA)>
```

The XML file size is reduced to the following:

```
<?xml version="1.0"?>
<!DOCTYPE Product SYSTEM "product.dtd">
<Product>
  <Id>3124090231</Id>
  <Price>49.99</Price>
</Product>
```

It also is possible to control the occurrence of the children within an element. See Table 8.2 for a list of the most common occurrence controls and a description of what they do.

**Table 8.2** DTD Element Occurrence Controls

ELEMENT DEFINITION	ATTRIBUTE	DESCRIPTION
<code>&lt;!ELEMENT Product (Price)&gt;</code>	None	The Product element may only contain one instance of the child element.
<code>&lt;!ELEMENT Product (Price*)&gt;</code>	*	The Product element can contain multiple child elements.

Table 8.2 (Continued)

ELEMENT DEFINITION	ATTRIBUTE	DESCRIPTION
<code>&lt;!ELEMENT Product (Price+)&gt;</code>	+	The Product element can contain one or more instances of the child elements.
<code>&lt;!ELEMENT Product (Price?)&gt;</code>	?	The Product element can contain zero or one instance of the child element.

DTD attributes are used in cases where XML elements contain attributes that need validation. The syntax for a single attribute is as follows:

```
<!ATTLIST element-name attribute-name CDATA "default-value">
<!ATTLIST Product type CDATA "hardware">
```

The syntax for a multi-attribute element is as follows:

```
<!ATTLIST element-name attribute-name (enum1|enum2... ) "default-value">
<!ELEMENT Product (#PCDATA)>
<!ATTLIST Product type(hardware|software|services) "hardware">
```

The following XML code respects the definitions of the enumerated attribute values defined previously. `Product` could be of the type hardware, software, or services. The default value, if not provided in the attribute list definition, is hardware because this is the first entry in the enumeration.

```
<Product type="hardware">
  <Id>34254030546</Id>
</Product>
<Product type="software">
  <Id>99321254030122</Id>
</Product>
```

In the case where text is reused multiple times, entities can be used to define a variable once and then be reused throughout the document thereafter.

Entity can be used for internal definitions, as follows:

```
<!ENTITY entity-name "entity-value">
<!ENTITY companyname "ABC Sports">
```

## 328 Chapter 8

---

Entities also can be used for external definitions, as follows:

```
<!ENTITY entity-name SYSTEM "URI/URL">
<!ENTITY companyname SYSTEM "http://www.myxml.com/entities/myentity.xml ">
```

The XML data is the same whether or not an internal or external entity source is used.

```
<Information>&companyname;</Information>
```

DTD has been the first and only constraint language for validating XML documents. It has solved many problems that developers were facing. In the current wave of technological evolutions, DTDs are not able to handle some requirements with ease.

For example, one disadvantage of the DTD is that it is hard to read and does not use XML as the definition format. DTDs are not very good at expressing sophisticated constraints on elements, such as the number of maximum and minimum occurrences of a particular element. DTDs do not have the capability to reuse previously defined structures. They also do not have support for type inheritance, subtyping, and abstract declarations.

A more flexible standard that covers most of the limitations of DTDs is XML Schema. This standard is becoming more popular as the definition format, because it is easier to understand and maintain.

### **XML Schema**

XML Schema is currently a W3C recommendation ([www.w3.org/XML/Schema](http://www.w3.org/XML/Schema)). XML Schema is hierarchical and enables type, structure, and relationship definitions to exist as well as field validations within elements. XML Schema is harder to learn and create than DTDs but solves the major limitations of DTDs. The schema definition is written in XML, which seems like a natural fit in the XML world with great tool support for creating and editing XML documents. XML Schema is not part of the XML 1.0 specification, which means that valid XML documents only apply to documents that are validated by DTDs using the DOCTYPE declaration.

### **Comparing DTD to XML Schema**

The following demonstrates the difference between a DTD and an XML Schema definition for the same XML file:

```
<Product>123456</Product>
<Price>49.99</Price>
```

The DTD for the previous XML code is as follows:

```
<?xml version="1.0"?>
<!DOCTYPE Product [
    <!ELEMENT Product (Id,Price)>
    <!ELEMENT Price (#PCDATA)>
]>
```

The XML Schema for the same XML code is as follows:

```
<!-- This Is the xml schema for the XML above -->
<element name='Product' type='string' />
<element name='Price' type='string' />
```

### **XML Schema Declaration Using Namespace**

The schema declarations begin the same way a regular XML document begins, with a prolog. It starts with `<schema>` as the root element. It then includes references to an XML Schema namespace declaration. Namespace declarations are needed in this case because the document being operated upon references specific elements from the schema. The schema elements provide the semantics for constraining elements in the other namespace of the XML document being processed. The following is a fragment of an XML Schema declaration:

```
<?xml version="1.0" ?>
<schema targetNamespace="http://www.acme.com/warehouse/catalog"
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:Catalog="http://www.acme.com/warehouse/catalog">

    <element name='Product' type='string' />
    <element name='Price' type='string' />
```

The root element `<schema>` contains a `targetNamespace` attribute, which specifies the target that the schema will constrain. The previous example shows that two different namespaces are defined. One is the default namespace (unqualified) used by the XML Schema and the second (qualified) is the target document defined using the `Catalog` prefix. There are no general rules for choosing a namespace as the default. There are suggestions to make the default namespace the same as the `targetNamespace`. There is not an optimal solution to this problem; which default namespace is optimal truly depends upon whether the schema will be extended, whether it will import different schemas, and any number of other things.

## 330 Chapter 8

### Using Multiple Schemas

There could be a case where the XML document refers to names of elements found in multiple namespaces that are defined in multiple schemas. In that case, the location of the XML schemas must be specified using the `schemaLocation` attribute. The following example shows a fragment of an XML document:

```
<?xml version="1.0" ?>
<Catalog:Product xmlns:Catalog="http://www.acme.com/warehouse/catalog/"
xmlns:Buyer="http://www.acme.com/warehouse/buyers/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.acme.com/warehouse/catalog
                    http://www.acme.com/schema/catalogs.xsd
                    http://www.acme.com/warehouse/buyers
                    http://www.acme.com/schema/customers.xsd">
  <!-- uses default namespace -->
  <Buyer:Name>
    John Doe
  </Buyer:Name>
  <Catalog:Item>
    <Catalog:Id>090902343</Catalog:Id>
    <Catalog:Name>Futsji</Catalog:Name>
  </Catalog:Item>
</Catalog:Product>
```

Another way to use multiple schemas is to import one schema into another, which is achieved in the following code:

```
<schema targetNamespace="http://www.acme.com/warehouse/catalog"
xmlns="http://www.w3.org/2001/XMLSchema"
xmlns:Catalog="http://www.acme.com/warehouse/catalog"
xmlns:Buyer="http://www.acme.com/warehouse/buyers">

  <import namespace="http://www.acme.com/warehouse/buyers"
    schemaLocation="http://www.acme.com/schema/customer.xsd" />
```

The `import` element is used to specify the namespace along with the URI location of the schema definition.

Now that we have the root element defined, the rest should be straightforward. A schema definition is composed of elements and attributes that describe the content of the XML document.

### Elements

XML schema elements provide definitions for the content of an XML data document. A name and a type represent an element. The type sets a restriction to which the XML document elements must conform. Element types

**Table 8.3** XML Schema Data Types

TYPE	DESCRIPTION
String	Character string
Binary	Binary data
Boolean	Logic value (true or false)
Decimal	Positive or negative integer value
Double	64-bit floating point value
Float	32-bit floating point value
Uri	Uniform resource Indicator
TimeInstant	Date and time stamp
TimeDuration	Duration of time
RecurringInstant	A recurrence of time occurring over a timeDuration

come in two different forms: primitive or complex. Primitive or simple element types are defined by the XML Schema specification. Table 8.3 lists most of the data types supported by the XML Schema specification.

Simple data types cannot contain other elements or any other attributes. The schema element, using simple data types, is defined using the following syntax:

```
<element name="[name of element]" type="[type of element] [option(s)]>
```

For example:

```
<element name="Price" type="decimal" />
```

Name is used to identify the XML element that the schema is constraining. `type` refers to the type of data that is expected to be stored between the elements. There also are other possibilities of various options, such as the occurrence of an element in the XML document.

Similarly to DTDs, XML Schema provides an equivalent to (+, \*, ?) attributes, which are called `minOccurs` and `maxOccurs`. Revisiting the syntax, we get

```
<element name="[name of element]"  
  type="[type of element]"
```

## 332 Chapter 8

```

    minOccurs="[Min occurrences]"
    maxOccurs="[Max occurrences]"
  >

```

When unspecified, both options default to 1, meaning that one occurrence exists per definition. On the other hand, if a finite occurrence number is not defined, a wildcard (\*) character is used. For example, the following is a definition for the `LineItem` element, which has to occur at least many times and has no `maxOccurs` limit:

```
<element name=LineItem type=complexType minOccurs=1>
```

Complex or user-defined elements are used to define elements that consist of other elements and attributes. The syntax used for complex types is represented in the following order:

```

<complexType name="[name of type]">
  <[Element specification] />
  <[Element specification] />
  ...
</complexType>

```

For example, a `Product` contains a name, description, and price, and is considered to be a complex type, as shown in the following fragment:

```

<Product>
  <Name>Product A</Product>
  <Description>
This Is a description for Product A
  </Description>
  <Price>20.99</Price>
</Product>

<complexType name="ProductType">
  <element name="Name" type="string" />
  <element name="Description" type="string" />
  <element name="Price" type="decimal" />

</complexType>

```

In this hierarchical structure of elements, the lowest level of elements is considered to be a simple type, the rest are all complex types. The nesting of elements could be very deep; schema does not impose any restrictions on this. For example, a `PurchaseOrder` is composed of `POID`, `Buyer`, and `Product`. The `POID` (Purchase Order ID) is an integer and is considered to be a simple type. `Buyer` and `Product` are user-defined types that contain more embedded elements. The purchase order requires one `POID`, one

buyer, and many products, which can be accomplished by setting `minOccurs` and `maxOccurs` on the elements. One last restriction is that the product description appears zero or one times. This is realized by setting the description element to `maxOccurs="0"`. The following is an example of this scenario:

```
<schema targetNamespace="http://www.acme.com/warehouse/po"
  xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:PO="http://www.acme.com/warehouse/po"
  xmlns:Catalog="http://www.acme.com/warehouse/catalog"
  xmlns:Buyer="http://www.acme.com/warehouse/buyers"
>

<element name="PurchaseOrder" type="PO:PurchaseOrderType">
<complexType name="PurchaseOrderType">
  <element name="PurchaseOrderNumber" type="Integer" />
  <element name="PurchaseDate" type="string"/>
  <element name="BuyerID" type="Integer" />
  <element name="BuyerName" type="string"/>
  <element name="Order" type="PO:OrderType"/>
</complexType>

<complexType name="OrderType">
  <element name="LineItem" type="LineItemType" />
</complexType>

<complexType name="LineItemType">
  <element name="ProductNumber" type="decimal" />
  <element name="Quantity" type="decimal"/>
</complexType>
```

The types used in the previous example are called explicit types. Explicit types are defined in such a way that they can be reused in the same or different document. There also are cases where the element contains a type, which is unique to a specific element definition and will not be reused anywhere else. This is referred to as an *implicit type* or a *nameless type*. Implicit type enables the definition of a user-defined type within an element. The following example demonstrates the use of an implicit type. The `BuyerInfo` type cannot be reused anywhere else in this XML document except in the definition of `PurchaseOrderType`, which is shown in the following fragment of code:

```
<complexType name="PurchaseOrderType">
  <element name="POID" type="Integer" />
  <element name="BuyerInfo">
    <complexType>
      <element name="BuyerName" type="string" />
      <element name="BuyerPhone" type="string" />
    </complexType>
  </element>
</complexType>
```

## 334 Chapter 8

---

```
</complexType>
</element>
</complexType>
```

In addition, there is a notion of local and global definitions for elements, which are shown in the following example.

This example demonstrates an instance of a local definition because the elements of `Name`, `Description`, and `Price` belong to the `ProductType` element.

```
<element name="Product" type="ProductType" />
<complexType name="ProductType">
  <element name="Name" type="string" />
  <element name="Description" type="string" />
  <element name="Price" type="decimal" />
</complexType>
```

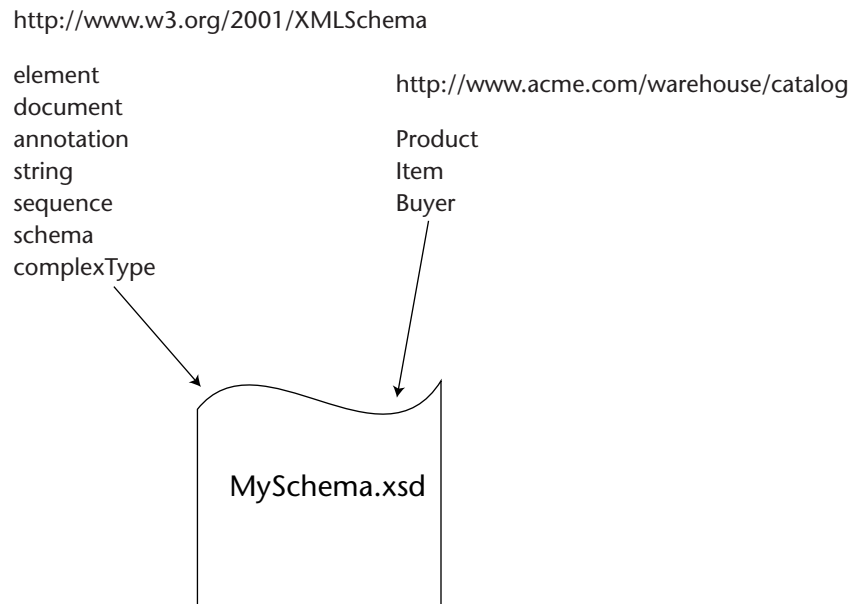
The following example uses global definitions, where `Name`, `Description`, and `Price` can be reused in other elements throughout the document:

```
<element name="Name" type="string" />
<element name="Description" type="string" />
<element name="Price" type="decimal" />
<element name="Product" type="ProductType" />

<complexType name="ProductType">
  <element ref="Name" />
  <element ref="Description" />
  <element ref="Price" />
</complexType>
```

The use of global and local definitions is really a matter of taste. Some of the best practices suggest making the declarations local if the elements are specific to the element being defined. Otherwise, if the elements can be reused throughout the document, then make the declarations global.

Figure 8.1 demonstrates how one XML Schema Definition file can be composed of many different schema files. In this case, XML Schema is used as the default to provide a set of simple data types that can be used to define certain elements in an XML document. In addition, the Catalog schema is used to provide more extensible user-defined types used within the XML document.



**Figure 8.1** XML Schema definition.

Previous sections of this chapter have discussed empty tags. These empty tags also can be constrained by the XML Schema definitions. This is accomplished by defining an element name containing a `complexType` of type `empty`, as follows:

```
<element name="SomeFlag">
  <complexType content="empty" />
</element>
```

### Attributes

Attributes are used to provide additional information to an XML data element. For example, the element `Price` can have an attribute identifying the currency of the price, as shown in the following:

```
<Price currency="USD">20.99</Price>
```

XML schemas can be used to enforce constraints on element attributes. The syntax for attribute definitions is as follows:

```
<attribute name="[name of attribute]"
           type="[type of attribute]"
           [Option(s)]
>
```

## 336 Chapter 8

If an attribute definition for the `Product` element were provided, the following would be the result:

```
<complexType name="ProductType">
  <element name="Name" type="string" />
  <element name="Description" type="string" minOccurs="0" />
  <element name="Price">
    <complexType content="decimal">
      <attribute name="currency" type="string" />
    </complexType>
  </element>
</complexType>
```

Simple types cannot have attributes; therefore they must be defined as a `complexType` because of the attribute addition. The attribute for `Price` is `currency`, which identifies the kind of currency of the price. The power of this definition can be extended by adding an option for making the attribute mandatory, thus providing a default currency or restricting the attribute to a list of possible currencies.

To make an attribute mandatory, use the `minOccurs` option by setting it to "1". The `minOccurs` option defaults to "0" because it is not always required:

```
<attribute name="currency" type="string" minOccurs="1" />
```

To add a default value for an attribute, use the "default" option:

```
<attribute name="currency" type="string" default="USD" />
```

To add a list of restricted values for an attribute, use an enumeration option:

```
<attribute name="currency" default="USD" />
  <simpleType base="string">
    <enumeration value="USD" />
    <enumeration value="CND" />
    <enumeration value="ERO" />
  </simpleType>
</attribute>
```

XML Schema is more powerful and more intuitive than DTDs, but it does not solve all of the problems. There are still limitations that cannot be solved with this standard alone. Instead, other technologies must be added and combined to solve more complex problems revolving around data restrictions and validations. For example, imagine an element, which has the following restrictions: The `<Price>` element of a `Product` is smaller or equal to the `<TotalPrice>` element of a `Purchase Order`. Additional XML

schema languages must be used to solve this problem. There are many different standards and tools that solve problems that DTDs and XML schemas cannot handle, such as the following languages:

- TREX
- Schematron
- SOX
- XDR
- RELAX

In the following sections, XML validation will be used to constrain XML data. We will see how to use XML validation with JAXP parsing and XML data binding using Castor. Parsing of the data is an essential mechanism used for interpreting the tags and elements that compose an XML data structure. There have been many vendor implementations that enable the parsing of XML, but one major factor was lacking: the standardization of XML processing. This is how Java APIs for XML processing and binding were born—these standards also are known as JAXP and JAXB. These two standards provide a standardized and vendor-neutral approach for XML processing. The following sections describe the two standards by going through the APIs and examples.

## Java API for XML Processing (JAXP)

Now that we have seen an overview of XML, we can begin looking at the various APIs available for the processing of XML. The process in which the XML is being interpreted is called parsing and the process where various templates are applied to XML to produce a different output is called transformation. The API that embraces both of these processes is called Java API for XML Processing, or JAXP. We give an overview of JAXP in Chapter 7, “Introduction to the Java Web Services Developer Pack (JWSDP).” JAXP also is an integral part of JWSDP and it facilitates the role of XML processing in Java Web services.

### JAXP

In the last couple of years, XML parsing has been standardized to two distinct processing models: Simple API for XML (SAX) and Document Object Model (DOM). These two standards address different parsing needs by providing APIs that enable developers to work with XML data. Many vendors have implemented their own specific parsers that address the two

## 338 Chapter 8

---

most common parsing models of SAX and DOM. This led to API standardization complexities where applications using one specific vendor were using specific method calls to perform equivalent parsing tasks. In the case where the parsers needed to be swapped, developers had to rewrite the parsing code to adapt to the new specific vendor standard.

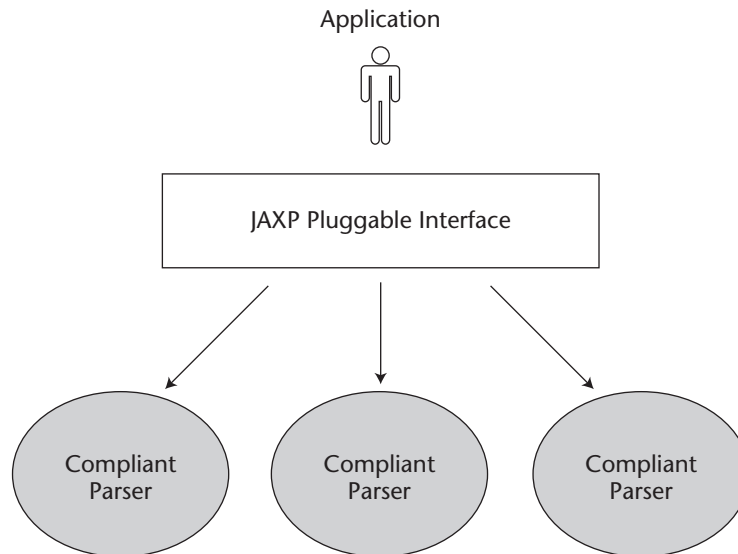
To address these complexities, in late 1999, Sun Microsystems came up with a draft of the Java API for XML Parsing (JAXP) 1.0 specification. The purpose of the specification was to provide a standard high-level API layer for abstracting lower-level details of XML parsing. The first version of JAXP (1.0) supported SAX 1.0 and DOM Level 1 parsing specifications. The feature set focused around parsing, which was a limitation that many have complained about because it did not provide support for any XML transformation processing. The latest version of JAXP 1.1 supports SAX 2.0, SAX 2 extensions, and DOM level 2. It also includes transformation support based on the eXtensible Stylesheet Language Transformations (XSLT) specification. The purpose behind JAXP is to provide an abstraction so that parsing using SAX or DOM looks the same no matter what type of SAX or DOM parser is being used by the application. This abstraction is referred to as pluggable interface.

### Uses for JAXP

JAXP provides a pluggability layer, which enables application developers to change parser implementations without affecting the application logic. One JAXP-compliant parser can be exchanged for another parser seamlessly, without much effort. JAXP provides a set of standard interfaces that encapsulate the details behind parser interactions. These interfaces act as abstractions that prevent the developer from working with XML directly. These abstractions are implementations of the SAX and DOM parsing standards and the XSLT transformation standard.

Figure 8.2 shows the high-level blocks that compose the JAXP model. In order for a parser and transformer to be compliant, it must follow the JAXP specification. The flexibility of having the freedom to choose any parser is very important. It enables an application developer to choose a parser provider that best suits the requirements of the service.

In addition to the pluggability layer, JAXP is made simple. The APIs are very straightforward and the learning curve is much smaller than learning a new proprietary API. The following section steps through the API for parsing and transforming XML documents.



**Figure 8.2** JAXP pluggable interface.

## JAXP API Model

The JAXP API model is quite easy to understand and simple to use. The parsing classes and interfaces are packaged under the `javax.xml.parsers` package and sit on top of existing SAX or DOM APIs. Transformation classes and interfaces are packaged in `javax.xml.transform` and provide utilities for performing XSLT transformations. Something really important to understand about JAXP is that it ships with Sun's Crimson parser, which is the reference implementation of the JAXP parser. The parser implementation is packaged in `parser.jar` with package name `com.sun.xml.parser`, which is *not* part of the JAXP specifications. Developers have the tendency to refer to the `com.sun.xml.parser` package as JAXP. The use of this package also misuses the concept of JAXP by bypassing the high-level interfaces by calling the parser API directly.

Table 8.4 is a listing of classes and interfaces of the JAXP package (`javax.xml.parsers.*` and `javax.xml.transform.*`).

**340 Chapter 8****Table 8.4** Classes and Interfaces of JAXP Package

<b>NAME</b>	<b>USE</b>	<b>CLASS OR INTERFACE NAME</b>
SAX Parser Factory	Instantiating the SAX Parser	<code>javax.xml.parsers.SAXParserFactory</code>
SAXParser	JAXP Implementation used for parsing XML	<code>javax.xml.parsers.SAXParser</code>
Document Builder Factory	Instantiating DOM Builder	<code>avax.xml.parsers.DocumentBuilderFactory</code>
Document Builder	Creating the DOM element tree	<code>javax.xml.parsers.DocumentBuilder</code>
Parser Configuration	Exception class used for handling configuration errors	<code>javax.xml.parsers.ParserConfigurationException</code>
Factory Configuration Error	Exception class used for handling factory configuration errors	<code>javax.xml.parsers.FactoryConfigurationError</code>
Transformer Factory	Used for Instantiating the Transformer class	<code>javax.xml.transform.TransformerFactory</code>
Transformer	Transformer class used for XSL transformations	<code>javax.xml.transform.Transformer</code>
Transformer Factory Configuration Error	Error class used for catching Factory configuration errors	<code>javax.xml.transform.TransformerFactoryConfigurationError</code>

**Table 8.4** (Continued)

<b>NAME</b>	<b>USE</b>	<b>CLASS OR INTERFACE NAME</b>
Stream Source	Input stream used for XSL transformations	<code>javax.xml.transform.stream.StreamSource</code>
Stream Result	Result stream used for XSL transformations	<code>javax.xml.transform.stream.StreamResult</code>
DOM Source	DOM tree source	<code>javax.xml.transform.dom.DOMSource</code>
DOM Result	DOM tree result	<code>javax.xml.transform.dom.DOMResult</code>
DOM Locator	DOM locator	<code>javax.xml.transform.dom.DOMLocator</code>
SAX Source	SAX Source	<code>javax.xml.transform.sax.SAXSource</code>
SAX Result	SAX Result	<code>javax.xml.transform.sax.SAXResult</code>

## 342 Chapter 8

---

The factory class is the fundamental pattern of JAXP. It provides the transparent instantiation of the parser implementation. In reality, this is the mechanism that promotes the pluggability approach by enabling the developers to define the factory implementation as a parameter passed to the Java virtual machine. The definition of the factory class can be done in many ways (for example, using `SAXParserFactory`):

System Property	<code>java -Djavax.xml.parsers.SAXParserFactory</code>
Property file	<code>\$JAVA_HOME/jre/lib/jaxp.properties</code>
JAR Service Provider	<code>META-INF/services/javax.xml.parsers.SAXParserFactory</code>
Default	Use the platform default as the last fallback

The same configuration settings apply to `DocumentBuilderFactory` and `TransformerFactory` classes.

### JAXP Implementations

Many implementations of JAXP 1.1 currently exist, including Sun Crimson, Apache Xerces, XML parser from IBM, and Oracle. For this chapter and throughout the book, we will use the Apache Xerces 2 XML parser and Xalan 2 Transformer, which are JAXP 1.1-compliant implementations.

A new change found in the new Java Development Toolkit (JDK) 1.4 is the addition of JAXP APIs and reference implementations. The reference implementations include Crimson for XML parsing and Xalan for XML transformations. The Endorsed Standards Override Mechanism must be used in order to override the JAXP implementation classes found in the JDK 1.4 with other JAXP compliant ones. For more information on how the mechanism works, refer to <http://java.sun.com/j2se/1.4/docs/guide/standards/>.

The following sections discuss the API and the different standards that it supports.

### Processing XML with SAX

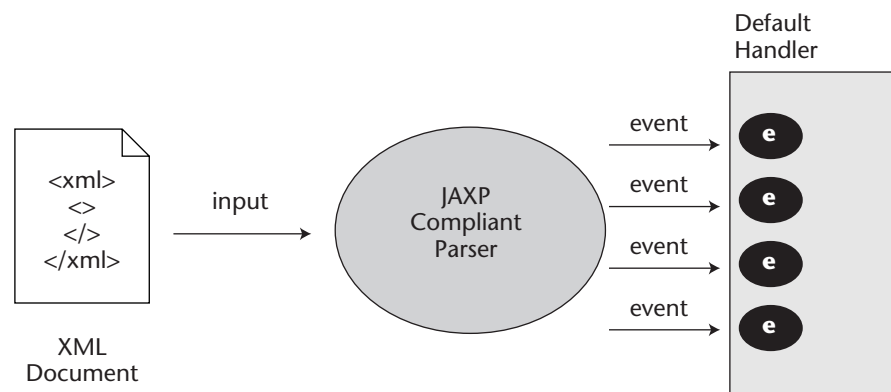
The Simple Access for XML (SAX) API is based on an event-driven processing model where the data elements are interpreted on a sequential basis and callbacks are called based on selected constructs. It is similar to the AWT 1.1 Event Delegation Model, where UI components generate events based on user input and where event listeners perform actions when these events are triggered. SAX's biggest advantage is that it does

not load any XML documents into memory; therefore it is considered to be very fast and lightweight. SAX supports validation but does not enforce the use of it. By having validation, a document is checked for conformance against a schema document (DTD or XML Schema). It uses a sequential read-only approach and does not support random access to the XML elements.

There are several ways to process XML data using SAX APIs. One way of doing it is to use a JAXP (`javax.xml.parsers.*`) API, which abstracts many low-level SAX specific calls or uses the SAX (`org.xml.sax.*`) API directly. Using the SAX API directly is more difficult because it requires more steps to perform equivalent functionality that is encapsulated in the JAXP API. In this section, we will focus around the JAXP specifics and will not cover vendor-specific SAX APIs in great detail because this topic spans a very large area.

SAX as a processing model is very simple (see Figure 8.3). The basics consist of the following three steps:

1. Implement a class that extends the `DefaultHandler` and holds callback methods for every type of construct found in XML that contains implementation based on your needs.
2. Instantiate a new SAX Parser class. The Parser reads the XML source file and triggers a callback implemented in the `DefaultHandler` class.
3. Read the XML source sequentially. With the sequential reading, it is not possible to randomly access elements in the structure. The rest depends upon your implementation residing in the `Handler` class.



**Figure 8.3** JAXP using the SAX processing model.

## 344 Chapter 8

---

The following sections describe JAXP-specific SAX processing. These steps consist of the following:

1. Getting `Factory` and `Parser` classes to perform XML parsing
2. Setting options such as namespaces, validation, and features
3. Creating a `DefaultHandler` implementation class

### **Getting a Factory and Parser**

The JAXP `SAXParser` class is responsible for parsing the XML data, and the implementation of the `org.xml.sax.DefaultHandler` handles all of the callbacks for specific tags. In order for an application to obtain an instance of the parser, it must get an instance of the `SAXParserFactory`. It then gets a `SAXParser` upon which it can call parse methods.

### **How to Get a SAX Parser**

The following is a step-by-step demonstration on how to create a SAX parser.

1. Instantiate the implemented `Handler` class. JAXP1.1 requires that the handler extend from `DefaultHandler` class as opposed to `HandlerBase`, which was an implementation used in JAXP 1.0:

```
DefaultHandler handler = new MyImplOfHandler();
```

2. Obtain a factory class using the `SAXParserFactory`'s static `newInstance()` method.

```
SAXParserFactory factory =  
    SAXParserFactory.newInstance();
```

3. Obtain the SAX parser class from the factory by calling the `newSAXParser()` static method:

```
SAXParser parser = factory.newSAXParser();
```

4. Parse the XML data by calling the parse method on `SAXParser` and passing in the XML input as the first parameter and the `DefaultHandler` implementation as the second. See `SAXParser` javadoc for different options available in the parse method.

```
parser.parse("PurchaseOrder.xml", handler);
```

The factory class used in the second step is provided as a system property, jar service, or platform default. This type of implementation is referred to as the JAXP pluggability layer, because it is very generic the way the factory class is retrieved. Parser vendors can be swapped without much effort as long as they comply to the JAXP 1.1 specification. If the

property is not found at runtime, a `FactoryConfigurationError` is thrown with a message describing the origin and cause of the problem. The parser factory can additionally be configured to instantiate validating and/or namespace-aware parsers. A `ParserConfigurationException` is thrown when the parser is not returned properly.

### **Configuration of the Factory**

The following code shows how to define a factory class by using a key/value pair set as a system property. This line can be placed in a configuration file (`java.util.Properties`) or simply passed as an argument to the Java call. The most common implementations known to developers are the Sun Crimson reference implementation and Apache Xerces. The following is an example of setting Xerces 2 and Crimson JAXP `SAXParserFactory`.

#### **APACHE XERCES 2**

```
javax.xml.parsers.SAXParserFactory=org.apache.xerces.jaxp.SAXParser  
FactoryImpl
```

#### **SUN REFERENCE IMPLEMENTATION (CRIMSON)**

```
javax.xml.parsers.SAXParserFactory=com.sun.xml.parser.SAXParserFactory  
Impl
```

### **Setting Namespaces**

A parser that is namespace aware is a parser that can handle naming collisions. This is important when multiple documents are used with the same application.

To configure the parser to be namespace aware, perform the following steps:

```
/** Set to namespace aware parsers */  
factory.setNameSpaceAware(true);
```

To verify if the parser supports namespaces, perform the following steps:

```
If (parser.isNamespaceAware()) {  
    System.out.println("Parser supports namespaces");  
} else {  
    System.out.println("Parser does not support namespaces");  
}
```

## 346 Chapter 8

---

### **Setting Validation**

Validation is based on providing a validation document that imposes certain constraints on the XML data. Many standards provide validating capabilities based on DTDs and W3C XML schemas.

To configure the parser to be a validation-aware parser, perform the following steps:

```
/** Set to validating parsers */
factory.setValidating(true);
```

The factory class will try to look for a validation-capable parser. If the parser is not available, a factory configuration exception is thrown.

To verify whether a parser supports validation, the following method is available from the parser class:

```
If (parser.isValidating()) {
    System.out.println("Parser supports validation.");
} else {
    System.out.println("Parser does not support validation.");
}
```

### **Setting Features**

SAX 2.0 enables a more flexible configuration option to exist through a method called `setFeature()`. This method is called on the factory class just like `setNamespaceAware()` and `setValidating()`. This option sets various features implemented by the vendors to be configured in a parser. For example, setting a schema validation on/off would have the following syntax:

```
Factory.setFeature("http://www.acme.com/xml/schema", true);
```

The `getFeature()` method enables the application to verify whether a particular feature is set by returning a boolean.

### **Creating a Default Handler**

JAXP 1.1 supports SAX 2.0, which promotes the extension from `DefaultHandler` rather than from `HandlerBase` like in SAX 1.0. The parser's parse method provides backward compatibility by exposing methods that take both `DefaultHandler` and `HandlerBase`. The following sample shows a sample handler that extends from the `DefaultHandler`. Because

---

## XML Processing and Data Binding with Java APIs 347

---

the SAX parser consists of a sequential reading of XML documentation, callback methods from the handler class will be invoked for every occurrence of a document, element, and character. The developer has full control over the action that can take place while the XML document is read. The following code lists the methods that need to be implemented when extending from `DefaultHandler`:

```
public class MySAXExampleHandler extends DefaultHandler {

    public MySAXExampleHandler() {
    }

    public void startDocument() throws SAXException {
    }

    public void endDocument() throws SAXException {
    }

    public void characters(char buf [],
                          int offset,
                          int len)
        throws SAXException {
    }

    public void startElement(String namespaceURI,
                            String localName,
                            String rawName,
                            Attributes attrs)
        throws SAXException {
    }

    public void endElement(String namespaceURI,
                          String localName,
                          String rawName)
        throws SAXException {
    }
}
```

In this code, the following callback methods are called:

**startDocument()**. This method is called only once at the start of the XML document.

**endDocument()**. This method is called when the parser reaches the end of the XML document.

**characters()**. This method is called for character data residing inside an element.

## 348 Chapter 8

---

**startElement()**. This method is called every time a new opening tag of an element is encountered (for example, `<element>`).

**endElement()**. This method is called when an element ends (for example, `</element>`).

In the SAX sample code section, the default handler will be implemented to read and output the XML to the terminal.

### *Using SAX Parser*

Data can be parsed with SAX in many different ways. The JAXP-compliant parser is obtained from the factory class and is called `SAXParser`. This parser is an implementation that is provided by many vendors. Because it conforms to the specification, it can be easily exchanged. Obtaining the parser object is pretty straightforward. Developers must instantiate the factory class, and from the factory class they must instantiate a new parser class:

```
SAXParserFactory saxFactory = SAXParserFactory.newInstance();
javax.xml.parsers.SAXParser saxParser =
    saxFactory.newSAXParser();
```

While getting an instance of the factory and parser, there are a couple of things that can go wrong. The factory configuration can be wrong, and an exception will be thrown to indicate this problem. The exception is called `FactoryConfigurationException`, and it must be caught during the instantiation of the factory object. The second area where the application can fail is while the parser object is instantiated. In this case, the `ParserConfigurationException` must be caught while the factory instantiates a new parser. The following is a listing of more complete code:

```
try {
    SAXParserFactory saxFactory = SAXParserFactory.newInstance();
    javax.xml.parsers.SAXParser saxParser =
        saxFactory.newSAXParser();
} catch (FactoryConfigurationException fce) {
    // handle exception here
} catch (ParserConfigurationException pce) {
    // handle exception here
}
```

The next step, once the parser is obtained, is to call the `parse()` method by passing the XML data to be parsed and a `Handler` class to handle the data that is being passed. The following are possibilities in which the parser can be used:

```
/** Passing file as a parameter */
saxParser.parse(new File(someFileString), mySAXExampleHanlder);

/** Passing a URI as a String parameter */
saxParser.parse("http://www.acme.com/xml/PurchaseOrder.xml",
               mySAXExampleHandler);

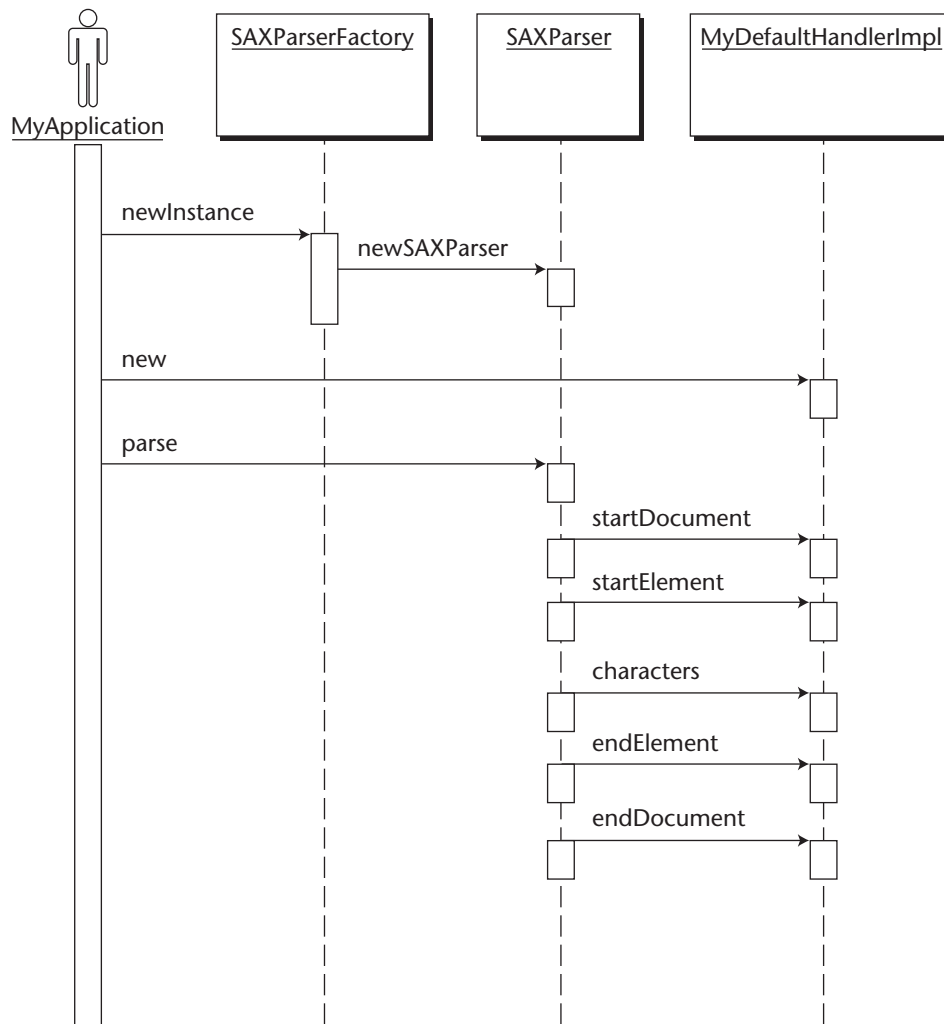
/** Parsing Input stream as a parameter */
saxParser.parse(someInputStream, mySAXExampleHanlder);
```

There are options with `HandlerBase` in the `parse()` method, but these are SAX 1.0 implementations. The current JAXP version implements those for backward compatibility, but you should use `DefaultHandler` when JAXP 1.1 is used.

Data also can be parsed with non-JAXP parsers such as the underlying `org.xml.sax.Parser`, but these approaches aren't vendor-neutral and will be omitted in this book. SAX 2.0 introduced the `XMLReader`, which is essentially a parser class that is wrapped by the `SAXParser` implementation. `XMLReader` is obtained by calling `getXMLReader()` from the `XMLParser`. Because low-level SAX parsing is not in the scope of this book, it will not be covered in more detail.

### ***Reading and Writing XML***

Reading and writing XML documents with SAX requires a parsing handler class (extended from `DefaultHandler`) to be implemented. This handler class provides logic coded in the callback methods defined by the programmer. The parser then processes the input stream and invokes the handler's callback methods to perform the actual work (see Figure 8.4). The `parse` method accepts various input parameters, including `java.io.InputStream`, `org.xml.sax.InputSource`, `java.io.File`, and `java.lang.String`. The second parameter is the implementation of the `DefaultHandler` class.

**350 Chapter 8**

**Figure 8.4** Sequence diagram showing JAXP parsing using SAX.

### ***Sample SAX Source Code***

The following is a sample `Handler` class and an implementation class, which uses the SAX approach to parse an XML file and to output the whole file into the console (system out).

Before running the code, compile the java files using the ANT script provided with the examples. To run, execute the following command:

```
java -classpath d:\xerces-1_4_4\xerces.jar; . -
Djavax.xml.parsers.SAXParserFactory=org.apache.xerces.jaxp.SAXParserFact
oryImpl jws.ch08.sax.MySAXExample
```

Listing 8.1 is a code listing for `MySAXExampleHandler.java`.

```
package jws.ch08.sax;

import org.xml.sax.*;
import org.xml.sax.helpers.*;
import org.xml.sax.ext.*;

public class MySAXExampleHandler extends DefaultHandler {
    public MySAXExampleHandler() {
    }

    public void startDocument() throws SAXException {
        System.out.println("START DOCUMENT");
        System.out.println("<?xml version='1.0'?>");
    }

    public void endDocument () throws SAXException {
        System.out.println("END DOCUMENT");
    }

    public void characters (char buf [],
                           int offset,
                           int len)
        throws SAXException {
        String s = new String(buf, offset, len);
        System.out.println (s);
    }

    public void startElement(String namespaceURI,
                             String localName,
                             String rawName,
                             Attributes attrs)
        throws SAXException {

        System.out.print("<"+localName);
        int length = attrs.getLength();
        for (int i=0; i < length; i++) {
            System.out.print(" "+attrs.getLocalName(i)+
                             "="+attrs.getValue(i));
        }
        System.out.println(">");
    }
}
```

**Listing 8.1** `MySAXExampleHandler`. (continues)

## 352 Chapter 8

```
public void endElement(String namespaceURI,
                      String localName,
                      String rawName)
    throws SAXException {
    System.out.println("</"+localName+">");
}
}
```

**Listing 8.1** MySAXExampleHandler. (continued)

Listing 8.2 shows a code listing for MySAXExample.java.

```
package jws.ch08.sax;

import javax.xml.parsers.*;

class MySAXExample {

    public MySAXExample () {
    }

    public static void main (String [] args) {
        try {
            SAXParserFactory factory =
                SAXParserFactory.newInstance();
            SAXParser parser = factory.newSAXParser();
            parser.parse("PurchaseOrder.xml",
                new MySAXExampleHandler() );
        } catch (FactoryConfigurationError fce) {
            System.out.println("FactoryConfigurationError occurred : "+fce);
        } catch (ParserConfigurationException pce) {
            System.out.println("ParserConfigurationException occurred :
                "+pce);
        } catch (Exception e) {
            System.out.println("Exception occurred : "+e);
        }
    }
}
```

**Listing 8.2** MySAXExample.java.

Listing 8.3 is a partial result output of the classes implemented previously. The SAX implementation class reads the XML document and outputs the same XML in the console. When it starts the parsing and reaches

the beginning of the XML document, it prints out *START DOCUMENT*. At the end of the document, it prints *END DOCUMENT*.

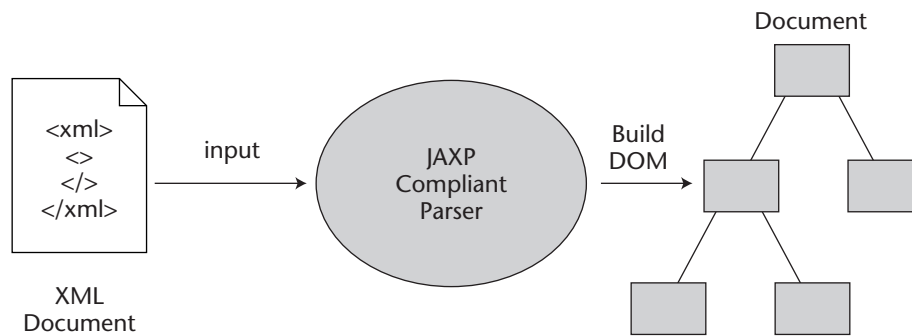
```
START DOCUMENT
<?xml version='1.0'?>
<PurchaseOrder>
  <Header>
    <PurchaseOrderNumber>
      2123536673005
    </PurchaseOrderNumber>
    <Date>
      02/22/2002
    </Date>
    <BuyerNumber>
      0002232
    </BuyerNumber>
    <BuyerName>
      John Doe
    </BuyerName>
    ...

  <LineItem type=SW>
    <ProductNumber>
      33333
    </ProductNumber>
    <Quantity>
      145
    </Quantity>
  </LineItem>
</Order>
</PurchaseOrder>
END DOCUMENT
```

**Listing 8.3** Sample SAX parsing output.

## Processing XML with DOM

The Document Object Model (DOM) API was defined and is maintained by the W3C working group. The W3C definition ([www.w3.org/TR/WDDOM/](http://www.w3.org/TR/WDDOM/)) states that “the Document Object Model is a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents.”

**354 Chapter 8****Figure 8.5** JAXP using DOM processing model.

The DOM processing model consists of reading the entire XML document into memory and building a tree representation of the structured data (see Figure 8.5). This process can require a substantial amount of memory when the XML document is large. By having the data in memory, DOM introduces the capability of manipulating the XML data by inserting, editing, or deleting tree elements. Unlike the SAX, it supports random access to any node in the tree. DOM supports validation by using DTD or a W3C XML Schema, but it does not enforce the use of validation.

The DOM processing model is more complete than the SAX alternative. It is considered more intensive on the resources, because it loads the entire XML document into memory. The basic steps for using DOM processing are as follows:

1. Instantiate a new `Builder` class. The `Builder` class is responsible for reading the XML data and transforming it into a tree representation.
2. Create the `Document` object once the data is transformed.
3. Use the `Document` object to access nodes representing elements from the XML document.

The XML source is read entirely into memory and is represented by the `Document` object. This enables the application to access any node randomly, which is something that SAX cannot do. One disadvantage of DOM is that it can be inefficient when large data is read into memory.

The following sections describe JAXP-specific SAX processing. These steps consist of the following:

- Getting a `Factory` and `Builder` class
- Setting namespaces, validation, and features

- Obtaining a Document object representing the XML element tree
- Traversing through the DOM node tree

### **Getting a Factory and Builder Class**

DOM provides a document builder class for parsing XML data. The DOM model is slightly different from SAX, however, because in most implementations it does rely on the SAX model for reading the XML into memory. This is not something that is mandated by the JAXP 1.1 specification, but it makes sense to leverage to existing capabilities of the API to build the document object model. The process of processing XML is very similar to that of SAX with a few class name changes. The following is an example showing the steps for getting a Document (`org.w3.dom.Document`) object, which consists of an in-memory tree structure composed of nodes. The nodes are representations of XML elements and attributes read from the XML input document.

1. Obtain a new instance of the DOM Factory class by calling the `DocumentBuilderFactory` class's `newInstance()` static method:

```
DocumentBuilderFactory factory=  
    DocumentBuilderFactory.newInstance();
```

2. Instantiate a new document builder class, once the factory is obtained, by calling the `newDocumentBuilder()` static method:

```
DocumentBuilder builder =  
    DocumentBuilder.newDocumentBuilder();
```

The document builder is used for loading the XML data into memory and for creating the Document object representation of the XML data.

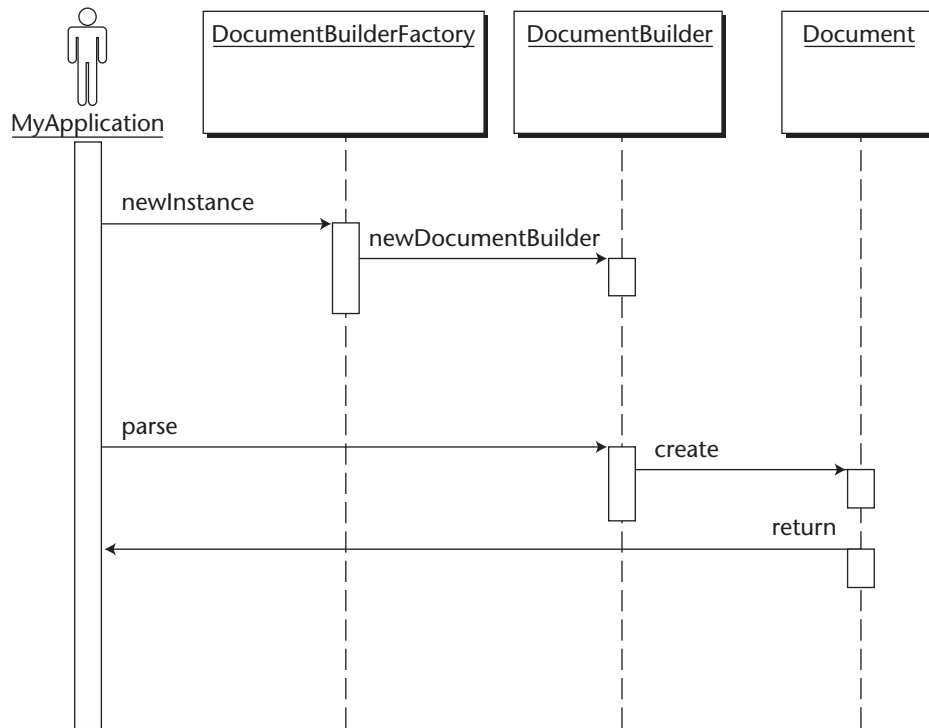
3. Parse, using the Builder class, which contains a parse method that accepts an input stream representing the XML data:

```
Document document =  
    builder.parse(http://www.acme.com/warehouse/PurchaseOrder.xml);
```

The result of the parse method returns a Document object that is used for accessing nodes of the tree representing the XML data.

The configuration of the factory class is exactly the same as the SAX configuration, except for the class names. The following is an example of the java system property option for a `DocumentBuilderFactory` class:

```
-Djavax.xml.parsers.DocumentBuilderFactory=  
org.apache.xerces.jaxp.DocumentBuilderFactoryImpl
```

**356 Chapter 8**

**Figure 8.6** Sequence diagram showing JAXP parsing using DOM.

Figure 8.6 shows the different steps that are required in order to parse an XML document into a DOM structure. It uses the steps explained previously to produce a `Document` object that represents the parsed XML data.

### Using Namespaces

A parser that is namespace-aware is a parser that is able to handle naming collisions. This is important when multiple documents are used with the same application.

To configure the parser to be namespace-aware, the factory class must be configured. To configure the factory class, perform the following steps:

```

/** Set to namespace aware parsers */
factory.setNameSpaceAware(true);

```

Once configured, the factory class will return a parser that is namespace-aware. The application can always perform checks by calling `isNameSpaceAware()` on the `DocumentBuilder` class to verify whether namespaces are supported by the parser:

```
If (builder.isNamespaceAware()) {  
    // Builder provides namespace support  
} else {  
    // Builder does not support namespace support  
}
```

### **Using Validation**

Validation is based on providing a validation document that imposes certain constraints on the XML data. Many standards provide validating capabilities, including DTDs and W3C XML Schemas.

To configure the parser to have validation turned on, the factory class must be configured to return validation-aware parsers only. To configure the factory class to do so, perform the following steps:

```
/** Set to validating parsers */  
factory.setValidating(true);
```

Once configured, the factory class will return a parser that is validation capable. The application can always perform checks by calling `isValidating()` on the `DocumentBuilder` class to verify whether validation is supported by the parser:

```
If (builder.isValidating()) {  
    // Builder is validation capable  
} else {  
    // Builder is not validation capable  
}
```

If the factory class is configured to be namespace-aware or validating, it will try to locate a parser that supports these settings. If it cannot return a correct parser that supports the configurations, it will throw a `ParserConfigurationException`.

### **Using DocumentBuilder**

The document builder is analogous to the `SAXParser` class and is used for parsing XML data. `DocumentBuilder` is used for building the `Document` object (`org.w3c.xml.Document`). This `Document` object is the object graph, which consists of nodes representing the elements and attributes contained in the parsed XML input. In order to obtain the `Document` object, the builder must parse the input XML by using one of the methods shown in Table 8.5.

**358 Chapter 8****Table 8.5** Methods for Parsing Input XML

<b>METHOD</b>	<b>PURPOSE</b>
<code>builder.parse (java.io.File file)</code>	Passing file Input as a parameter
<code>builder.parse(java.lang.String uri)</code>	Passing a string URI as a parameter
<code>builder.parse(java.io.InputStream input)</code>	Passing <code>InputStream</code> as a parameter
<code>builder.parse(org.xml.sax.InputSource input)</code>	Passing <code>InputSource</code> as a parameter

The following is a snippet of code for obtaining a Document object:

```
try {
    DocumentBuilderFactory factory =
        DocumentBuilderFactory.newInstance();
    DocumentBuilder builder = factory.newDocumentBuilder();

    if (builder.isNamespaceAware()) {
        System.out.println("Builder is namespace aware");
    } else {
        System.out.println("Builder is not namespace aware");
    }

    if (builder.isValidating()) {
        System.out.println("Builder is validation capable");
    } else {
        System.out.println("Builder is not validation capable");
    }

    Document document =
        builder.parse(new File("PurchaseOrder.xml"));

} catch (ParserConfigurationException pce) {
    System.out.println("ParserConfigurationException occurred:"+pce);
} catch (FactoryConfigurationError fce) {
    System.out.println("FactoryConfigurationError occurred:"+fce);
} catch (FileNotFoundException fnfe) {
    System.out.println("FileNotFoundException occurred:"+fnfe);
}
```

While getting an instance of the factory and builder objects, there is the possibility of the same exceptions being thrown as in SAX. The factory configuration can be wrong in this case—an exception is thrown to indicate this problem. The exception is called `FactoryConfigurationException`, and it is required that it be caught during the instantiation of the factory object. The second area where the application can fail is while instantiating the builder object. In this case, the `ParserConfigurationException` is required to be caught while the factory instantiates a new document builder. The following sections outline some of the important concepts when dealing with DOM trees. These sections cover some aspects that will help you understand the naming of each element or node within a tree.

### ***Traversal of a DOM Tree***

A DOM tree is composed of nodes. A node is the most essential object of the DOM tree; it is the representation of the XML data structure. When a parser processes the XML input, it produces a `Document` object. This `Document` object extends the node interface. The following are names of properties provided for each type of node in a Document tree. These properties help in the traversal of the tree.

- `nodeName` The name of the XML element that this node represents.
- `nodeValue` Text value that resides between the start and close element.
- `nodeType` A code representing the type of object (for example, element, attribute, text, and so on . . .).
- `parentNode` The parent of the current node (if any).
- `childNodes` List of children of the current node.
- `firstChild` Current node's first child.
- `lastChild` Current node's last child.
- `previousSibling` The node immediately preceding the current node.
- `nextSibling` The node immediately following the current node.
- `attributes` List of attributes of the current node (if any).

## 360 Chapter 8

### Sample Source Code

The following (`MyDOMExample`) is an implementation class that uses the DOM builder for reading the XML input into memory and then prints out the contents to the console.

Before running the code, compile the Java files using the ANT script provided with the examples. To run the code, execute the following command:

```
java -classpath D:\jaxp-1.1\crimson.jar;D:\jaxp-1.1\jaxp.jar;. -  
Djavax.xml.parsers.DocumentBuilderFactory=org.apache.crimson.jaxp.Docume  
ntBuilderFactoryImpl jws.ch08.dom.MyDOMExample
```

Listing 8.4 is a code listing for `MyDOMExample.java`.

```
package jws.ch08.dom;  
  
import javax.xml.parsers.*;  
import java.io.*;  
import org.w3c.dom.*;  
  
class MyDOMExample {  
  
    private int indent = 0;  
    private final String basicIndent = "  ";  
  
    public MyDOMExample () {}  
  
    private void printlnCommon(Node n) {  
        String val = n.getNamespaceURI();  
        if (val != null) {  
            System.out.print(" uri=\"" + val + "\"");  
        }  
  
        val = n.getPrefix();  
        if (val != null) {  
            System.out.print(" pre=\"" + val + "\"");  
        }  
  
        val = n.getLocalName();  
        if (val != null) {  
            System.out.print(" local=\"" + val + "\"");  
        }  
  
        val = n.getNodeValue();  
        if (val != null) {  
            if (!val.trim().equals("")) {  
                System.out.print(" nodeValue=\"" +  
                    n.getNodeValue() + "\"");  
            }  
        }  
    }  
}
```

**Listing 8.4** `MyDOMExample.java`.

```
}
}
System.out.println();
}

private void outputIndentation() {
    for (int i = 0; i < indent; i++) {
        System.out.print(basicIndent);
    }
}

private void printDocTree(Node n) {
    outputIndentation();

    int type = n.getNodeType();
    // verify what type of node we are dealing with
    switch (type) {
        case Node.DOCUMENT_TYPE_NODE:
            printlnCommon(n);
            NamedNodeMap nodeMap =
                ((DocumentType)n).getEntities();
            indent += 2;
            for (int i = 0; i < nodeMap.getLength(); i++) {
                Entity entity = (Entity)nodeMap.item(i);
                printDocTree(entity);
            }
            indent -= 2;
            break;
        case Node.ELEMENT_NODE:
            printlnCommon(n);
            // verify for more nodes
            NamedNodeMap atts = n.getAttributes();
            indent += 2;
            for (int i = 0; i < atts.getLength(); i++) {
                Node att = atts.item(i);
                printDocTree(att);
            }
            indent -= 2;
            break;
        default:
            printlnCommon(n);
            break;
    }

    // Print children if any
    indent++;
    for (Node child = n.getFirstChild(); child != null;
        child = child.getNextSibling()) {
        printDocTree(child);
    }
}
```

**Listing 8.4** MyDOMExample.java. (*continues*)

**362 Chapter 8**

```
    }
    indent--;
}

public static void main (String [] args) {
    try {
        DocumentBuilderFactory factory =
            DocumentBuilderFactory.newInstance();
        DocumentBuilder builder = factory.newDocumentBuilder();

        if (builder.isNamespaceAware()) {
            System.out.println("Builder is namespace aware");
        } else {
            System.out.println("Builder is not namespace aware");
        }

        if (builder.isValidating()) {
            System.out.println("Builder is validation capable");
        } else {
            System.out.println("Builder is not validation capable");
        }

        Document document =
            builder.parse(new File("PurchaseOrder.xml"));
        new MyDOMExample().printDocTree(document);
    } catch (ParserConfigurationException pce) {
        System.out.println("ParserConfigurationException occurred :
            "+pce);
    } catch (FactoryConfigurationError fce) {
        System.out.println("FactoryConfigurationError occurred :
            "+fce);
    } catch (FileNotFoundException fnfe) {
        System.out.println("FileNotFoundException occurred : "+fnfe);
    } catch (Exception e) {
        System.out.println("Exception occurred : "+e);
    }
}
}
```

**Listing 8.4** MyDOMExample.java. (*continued*)

Listing 8.5 is a sample output for the DOM example. It prints the XML data file to the console and prints the type of node that is being processed.

```
local="PurchaseOrder"
local="Header"
local="PurchaseOrderNumber"
```

**Listing 8.5** Sample DOM output.

```
    nodeValue="2123536673005"
  local="Date"
    nodeValue="02/22/2002"
  local="BuyerNumber"
    nodeValue="0002232"
  local="BuyerName"
    nodeValue="John Doe"
  local="BuyerAddress"
    local="Street"
      nodeValue="233 St-John Blvd"
    local="City"
      nodeValue="Boston"
    local="State"
      nodeValue="MA"
    local="Zip"
      nodeValue="03054"
    local="Country"
      nodeValue="USA"
  local="ShippingAddress"
    local="Street"
      nodeValue="233 St-John Blvd"
    local="City"
      nodeValue="Boston"
    local="State"
      nodeValue="MA"
    local="Zip"
      nodeValue="03054"
    local="Country"
      nodeValue="USA"
  local="PaymentInfo"
    local="Type"
      nodeValue="Visa"
    local="Number"
      nodeValue="0323235664664564"
    local="Expires"
      nodeValue="02/2004"
    local="Owner"
      nodeValue="John Doe"
  local="Order"
    local="LineItem"
      local="type" nodeValue="SW"
      local="ProductNumber"
        nodeValue="221112"
      local="Quantity"
        nodeValue="250"
    local="LineItem"
      local="type" nodeValue="HW"
      local="ProductNumber"
        nodeValue="343432"
```

**Listing 8.5** Sample DOM output. (*continues*)

## 364 Chapter 8

```
    local="Quantity"  
      nodeValue="12"  
  local="LineItem"  
    local="type" nodeValue="HW"  
    local="ProductNumber"  
      nodeValue="210020"  
    local="Quantity"  
      nodeValue="145"  
  local="LineItem"  
    local="type" nodeValue="SW"  
    local="ProductNumber"  
      nodeValue="33333"  
    local="Quantity"  
      nodeValue="145"
```

**Listing 8.5** Sample DOM output. (*continued*)

The following section is the new addition to JAXP 1.1. It deals with XML transformations and is referred to as Extensible Stylesheet Language Transformations (XSLT). Before we dive into this fascinating technology, we will look at an overview of the Extensible Stylesheet Language (XSL), which is the building block used for XSLT.

### XSL Stylesheets: An Overview

A stylesheet is used to apply a set of rules to transform input in order to produce a desired output format. By nature, XML does not focus on formatting, instead it provides data to the business logic. Stylesheets such as cascading style sheets (CSSs) or Extensible Stylesheet Language (XSL) are used for defining the way input will be formatted to take on a new output form.

Cascading style sheets are mostly used for HTML formatting, where they define the fonts, margins, colors, and so on. CSSs do have a very limited capability when it comes to transformations. XSL is a standard that has a more sophisticated model and provides richer capabilities for XML transforming.

The XSL constructs comply with the XML specification and therefore must be well formed and valid before they are used. This means that the syntax of XSL is restricted to ensure that the processing constructs are correct.

Conceptually, XSL deals with tree structures of data, where the processing instructions indicate what elements should be processed. Templates are used to match the root and leaf elements of the XML document. A large amount of data can be processed at a time, thus making searching and processing really efficient. For example, we can have thousands of `<LineItem>` elements under the `<Order>` element. If you select the `<Order>` element and apply a template to it, then the `<LineItem>` elements also can be affected by the selection. This makes XSL processing really efficient, because a large amount of data is processed at one time. The traversal of the tree is achieved by using the XPath standard. This standard provides the foundation for XSLT processing, where expression patterns are defined using XPath.

XPath is a standard that provides the mechanism for accessing the elements of an XML document. XPath expressions are important in XSLT because they enable stylesheet instructions to flexibly identify the parts of the input document to be transformed.

The specification itself is complex. While including extensive coverage would be impossible in this section, enough material is presented for you to understand the fundamentals of XSLT processing. These fundamentals enable you to traverse an XML document and select the set of elements that need to be included or excluded from the processing.

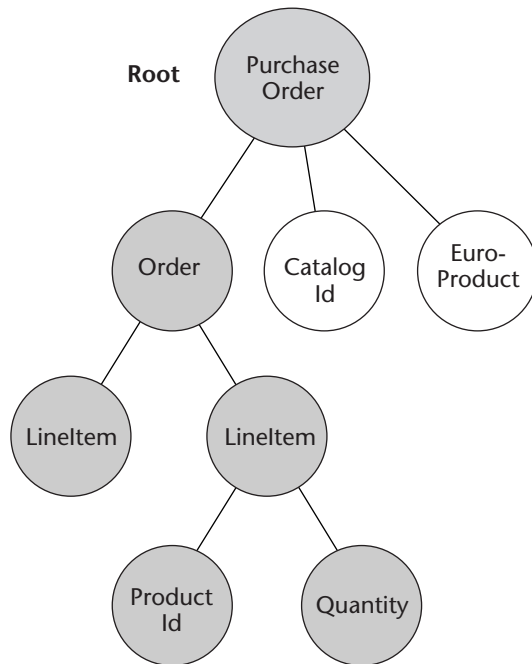
When anyone is operating on an XML document, the most common approach is to use relative paths to the element or attribute that is being worked on. When selecting a node to use as a starting point in our traversal, we identify what is called an axis. An axis is the path of the tree that is being used for the transformation.

Figure 8.7 shows a path traversed during the transformation. It starts at the root node and traverses only the order containing `LineItems`.

In the following paragraphs, some examples of XPath expressions are given. Consider the following XML:

```
<PurchaseOrder>
  <Header>
    <BuyderName>Robert</BuyerName>
  </Header>
  <Order>
    <LineItem type=SW>Item A</LineItem>
    <LineItem type=HW>Item B</LineItem>
    <LineItem type=SW>Item c</LineItem>
  </Order>
</PurchaseOrder>
```

## 366 Chapter 8



**Figure 8.7** Sample axis path used for transformation.

The following XPath expressions are used to access elements and attributes of the previous XML expression:

```

<!-- Order is an element that can be accessed from the current element
(PurchaseOrder) -->
<xsl:value-of select="Order" />

<!-- Match the LineItem element that is nested in the Order element -->
<xsl:value-of select="Order/LineItem" />

<!-- Match LineItem using the absolute path starting from the root
element -->
<xsl:value-of select="/PurchaseOrder/Order/LineItem" />

<!-- Match the 'type' attribute of the current element -->
<xsl:value-of select="@type" />

<!-- Match the 'type' element of LineItem element while being at the
root element -->
<xsl:value-of select="/PurchaseOrder/Order/LineItem/@type" />

```

After evaluating XML using the XPath syntax, we get an element or a set of elements as a result. This also is referred to as a node or node set,

because it represents the structured tree-like data. The resulting node set can be operated using additional XPath functionality. The following is the syntax that will help you understand the powerful yet complex mechanism for retrieving XML data for XML transformations. Because XSL is defined using XML, it must contain the following constructs in order to be considered valid:

- XML Declaration
- XSL root element: `<xsl:stylesheet>` `</xsl:stylesheet>`
- Declaration of used namespaces

### ***XML Declaration***

An XSL stylesheet uses the following standard XML declaration:

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

### ***XSL Root Element***

Because XSL stylesheets are defined in XML, they must contain a root element. This root element is defined by the following:

```
<xsl:stylesheet ...>  
...  
</xsl:stylesheet>
```

### ***XSL Namespaces***

The root element defines the namespaces used within the template and it also provides the version number, which is required. For example,

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/2002/XSL/Transform"  
version="1.0">  
  
</xsl:stylesheet>
```

The namespace `www.w3.org/2002/XSL/Transform` recognizes XSL-specific elements. A namespace used for any transformation-related work is prefixed with `"xsl:"`. For example,

```
<xsl:template match="PurchaseOrder" />  
...  
</xsl:template>
```

## 368 Chapter 8

---

A document may refer to a stylesheet using the following processing instruction:

```
<?xml-stylesheet type="text/xsl" href="order.xsl"?>
```

### ***XSL Syntax***

Many operations can be used to process the data in XML, including locating a particular element, iterating through a document, the conditional processing of elements, sorting, and numbering. In the following text, we look closely at the following topics:

- Templates used for locating parts of a document
- Conditional processing, which is done only when certain criteria are examined
- Looping used for iterating through results
- Sorting used for displaying the output in some logical order

### **Templates**

Templates are mostly used for locating one or more elements of an XML document. Using a template consists of using various rules that have specific requirements or conditions, which also are referred to as patterns. The template matching is defined by specific XPath expressions. To locate a particular element within a document, the Match attribute is used with an XPath expression that will match to zero or more elements in the document. The following is an example that will match the root element (`PurchaseOrder`) of our sample XML document:

```
<xsl:template match="PurchaseOrder">  
  /* The content in this block is written to the output.  
</xsl:template>
```

In this example, we matched the `PurchaseOrder` element, which is the first element in our path. At this stage of processing, we have matched the portion of the document that we would like to process, and the template has that portion of the XML hierarchy loaded into its memory. This hierarchy is an immutable structure that is used for XSL processing. This means that if you select an element, the process simply selects it but does not remove it. Also, if some elements of the hierarchy should become excluded, there must be a set of templates that tell the processor to ignore them. The most basic way to achieve access to elements is to use relative

---

## XML Processing and Data Binding with Java APIs 369

---

path expressions. Therefore, in relation to where we stand within the document, we can access various sections by simply specifying the paths to them. For instance, having

```
<xsl:template match="Order\LineItem">
```

as the first matching entry of the template would not take us there, because according to the root, the relative path to `LineItem` would be `PurchaseOrder\Order\LineItem`.

The `apply-templates` construct tells the processor to match the elements defined in the `select` attribute or if nothing is specified to match any element relative to the current path with any template defined within the XSL stylesheet:

```
/* apply pattern on selected nodes, the default node is *all* */  
<xsl:apply-templates select="node set expression" ... />
```

If a particular element must be fetched, it sometimes may be overkill to create a template for it. To retrieve particular elements, the `xsl:value-of` construct is used. For instance, getting the buyer ID number from the purchase order would look like the following:

```
<xsl:template match="PurchaseOrder\Header">  
  Buyer Number is : <xsl:value-of select="BuyerNumber" />  
</xsl:template>
```

In the case where we want to ignore an element, a simple solution is to create a template for that element with no action. This is a case where some information needs to be ignored, for instance, if we don't want to transform sensitive information, such as a credit card number or personal information of the buyer. This would result in the following template:

```
<xsl:template match="PurchaseOrder\Header\PaymentInfo">
```

Of course, this isn't the only solution for this sort of processing. We also could use conditional processing, which will be covered in the following section.

An example of conditional processing would look like the following:

```
<xsl:template match="PurchaseOrder">  
  <h1><xsl:apply-templates select="header" /></h1>  
</xsl:template>  
  
<xsl:template match="PurchaseOrder\Header\PaymentInfo">
```

## 370 Chapter 8

Apart from locating or matching elements in a document, we can perform a lot of other tasks by applying certain rules, which will traverse the XML tree and apply the expected changes under certain conditions.

### Conditional Processing

We can have conditional constructs that will apply to an XML document under certain conditions. A good example is one used in the previous section where personal information should not be transformed. In order to ignore the particular elements, we constructed an empty template. While it was a working solution, this example is not as clean as the one proposed in this section. Taking the same example, let's build a condition that will not select the "PurchaseOrder\Header\PaymentInfo" element and anything within it. This is accomplished by using the `not()` node function provided by XPath. The following is the syntax when using the `not()` function:

```
<xsl:apply-templates select="*[not(some expression)]" />
```

Our example would look like the following:

```
<xsl:template match="PurchaseOrder\Header">
  <xsl:apply-templates select="*[not(self:PaymentInfo)]" />
</xsl:template>
```

The self-parameter in the expression indicates what the frame of reference is. This lets the processor know that anything after the self-element is a child of the element.

Other conditional constructs can evaluate expressions based on expressions similar to if-then statements. One of these constructs is the `xsl:if` construct, which returns nodes that conform (evaluate to TRUE) to the expression. For example,

```
/* If expression is TRUE evaluate the template */
<xsl:if test="expression"> ... </...>
```

The test attribute contains the pattern that is evaluated to determine whether the condition is processed or not. In our example, we will separate the hardware (HW) and software (SW) type items:

```
<xsl:template match="PurchaseOrder\Order\LineItem">
  <xsl:if test="@type='SW'">
    Software Product # :<xsl:value-of select="ProductNumber">
  </xsl:if>
</xsl:template>
```

---

## XML Processing and Data Binding with Java APIs 371

---

The @ sign indicates that we are referring to an attribute rather than an element, and the single quotes around the SW string refers us to a static string. Another way to process conditional code is to use the `xsl:choose` construct. This construct behaves as an if-then-else type statement, where `<xsl:when ...>` evaluates the conditions, and if those fail then it executes `<xsl:otherwise>`. The `<xsl:when ...>` statement can occur many times within the `xsl:choose` construct, as shown as follows:

```
/* Apply a template to the conditions that evaluate to TRUE */
<xsl:choose>
  <xsl:when test="expression"> ...</xsl:when>
  [<xsl:when test="expression"> ...</xsl:when>]
  ...
  <xsl:otherwise> ...</xsl:otherwise>
</xsl:choose>
```

The following code is an example of an XSL stylesheet. It takes in the `PurchaseOrder.xml` file as input and sorts the `LineItems` according to their type: hardware, software, or unknown in the case of an unknown type:

```
<xsl:template math="PurchaseOrder\Order\LineItem">
  <xsl:choose>
    <xsl:when test="@type='SW'">
      Software Product # :<xsl:value-of select="ProductNumber">
    </xsl:when>
    <xsl:when test="@type='HW'">
      Hardware Product # :<xsl:value-of select="ProductNumber">
    </xsl:when>
    <xsl:otherwise>
      Unknown Product # :<xsl:value-of select="ProductNumber">
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

### Looping

Sometimes looping is needed when traversing a tree. XSL provides these constructs to facilitate the definition of repeating occurrences. The `xsl:for-each` construct is one of the constructs used for looping. In our example, we will iterate over the order and print out all of the product numbers:

```
<xsl:template math="PurchaseOrder\Order">
  <xsl:for-each select="LineItem">
```

## 372 Chapter 8

```

    Product # :<xsl:value-of select="ProductNumber">
  </xsl:for-each>
</xsl:template>

```

Although this can be produced with templates, it is much clearer to use the `for-loop` construct.

### Sorting

Sorting may be essential when formatting output based on a certain criteria, such as ascending or descending numbers or names. When sorting `<xsl:apply-templates ...>`, `<xsl:for-each ...>` constructs are used when evaluating the XML input. The following is a sorting construct that is used for sorting elements retrieved by templates (`xsl:apply-templates` or `xsl:for-each`) in its body:

```
<xsl:sort select="expression" .../>
```

The constructs have additional parameters that can be provided:

- `order="ascending|descending"`
- `data-type="text|number"`
- `case-order="upper-first|lower-first"`
- `lang="..."`

The following example takes the product number and sorts it in ascending order based on the `ProductNumber`:

```

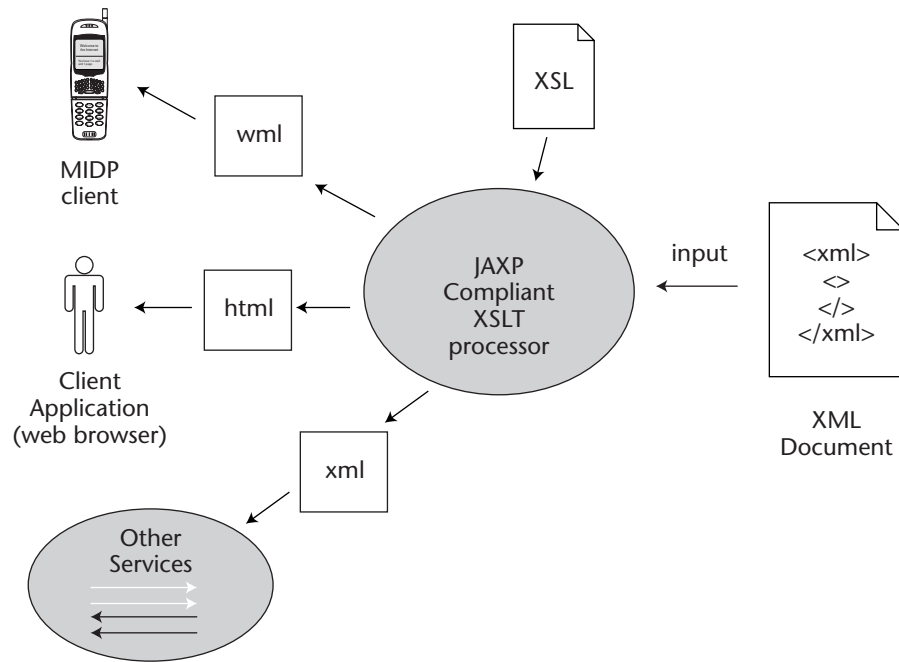
<xsl:template math="PurchaseOrder\Order">
  <xsl:apply-template select="LineItem">
    Product # : <xsl:sort select="ProductNumber"/>
  </xsl:apply-template>
</xsl:template>

```

The following section will use some of the important constructs discussed in this section and apply transformation to an XML document.

## Transforming with XSLT

Extensible Stylesheet Language Transformation (XSLT) is an XML processing standard used with XSL for XML-based document transformation. This is the process by which XSL templates are applied to XML documents in order to create new documents in desired formats (for example, XML, HTML, PDF, and WML). XSL provides the syntax and semantics for specifying formatting, and XSLT is the processor that performs the formatting task.



**Figure 8.8** Use of JAXP for XSLT transformation.

XSLT is often used for the purpose of generating various output formats for an application that enables access to heterogeneous client types (such as Web browsers, cell phones, and Java applications). It also is used to format one XML representation to another—this is typical in a B2B-type environment. Figure 8.8 depicts a scenario in which an application hosts a component that is able to generate various types of output based on the client type.

XSLT is supported in JAXP 1.1. This was achieved by incorporating Transformations for XML (TRaX) to provide a vendor-neutral solution for XML transformations. The various implementations of XSL processors drastically changed from one vendor to the next and a need for a standard solution was inevitable. The JAXP specification supports most of the XML transformations and is discussed in the next sections.

### ***Processing Model***

The processing model of XSLT consists of various rules defined by templates. A rule is defined as a combination of a template and pattern. The output is obtained by passing an input XML source (for example, Input-Stream, InputSource, File, or URL) to the processor. A node is processed by

## 374 Chapter 8

---

locating a template rule that contains the matching pattern. Once located, the template is instantiated and a result is created. This process continues until traversed recursively through the data. When many nodes are processed, the end result is known as the node list, which is a concatenation of all the node results. The following is a snippet of a template rule definition in an XSL stylesheet file:

```
<xsl:template match="some pattern">
  template
</xsl:template>
```

Patterns are sets of XPath expressions (location paths) that are used for evaluating node sets. A node matches a defined pattern when the node is a result from an evaluation of the expression, with respect to some context. A node is an additional attribute to the `xsl:template` construct, which enables an element to be processed many times in different ways.

The steps required for transformation follow these logical steps:

1. Obtain a transformation factory class used for instantiating a transformer class. This step is not much different from the previous SAX or DOM factory classes.
2. Once the factory class is instantiated, create a new transformer class by passing it the stylesheet for formatting the output.
3. Use the transformer class for transforming the data by specifying the XML input source and the output source where the results will be sent.

The XSL stylesheet is defined in a separate file with extension `*.xsl`. This stylesheet is passed when the transformer class is instantiated. Once the transformer class is obtained, the transformation can take place. We input the input source and output source when calling the transform method.

The following sections describe the JAXP-specific XSLT processing steps. These steps consist of the following:

- Getting a factory and transformer class
- Transforming the XML

### ***Getting the Factory and Transformer Class***

Working with XSLT is similar to working with the parsers. The class responsible for performing the transformation is called the `Transform` class, and it can only be obtained through a factory class (see Figure 8.9). The factory class is called `TransformerFactory` and is instantiated by calling its static

method `newInstance()`. Once the factory class is instantiated, the `newTransformer()` method is called to get an instance of the transformer. The `newTransformer()` method takes a `StreamSource` as a parameter, which is the XSL template that will be applied to the XML input.

The steps for instantiating the class are as follows:

1. As with SAX and DOM, use the factory class for instantiating a transformer implementation class:

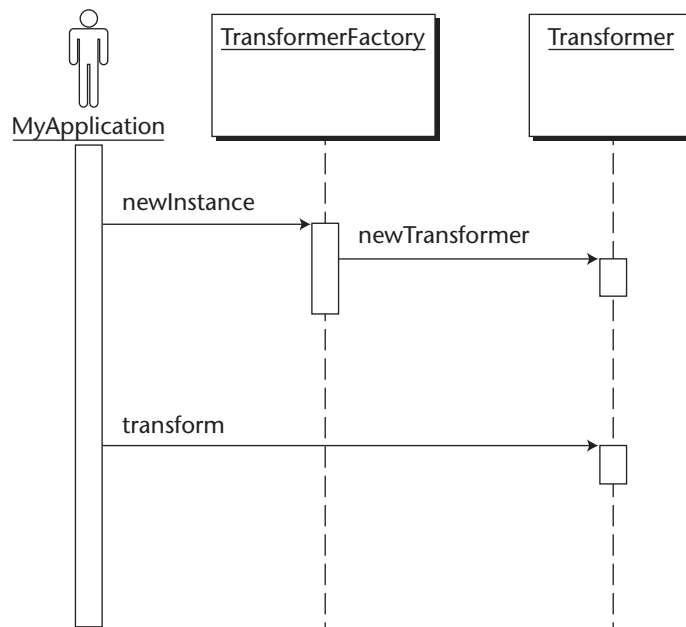
```
TransformerFactory factory =  
    TransformerFactory.newInstance();
```

2. Use the transformer class for applying the stylesheet to the input XML data. When getting a new instance, the stylesheet is passed as a parameter to the `newTransformer()` method:

```
Transformer transformer =  
    factory.newTransformer(new StreamSource("order.xml"));
```

3. The transformer then calls the transform method to invoke the transformation process. The parameters required in the transform method are input stream and output result:

```
transformer.transform(new StreamSource("PurchaseOrder.xml"),  
    new StreamResult(System.out));
```



**Figure 8.9** Sequence diagram showing a JAXP transformation with XSLT.

## 376 Chapter 8

---

The factory implementation class can be supplied as a Java system property using the following syntax:

```
javax.xml.transform.TransformerFactory=org.apache.xalan.processor
.TransformerFactoryImpl
```

Many options can be set on the factory class that affect the instance of the transformer. Some of these options are vendor-specific, such as attributes that can be passed to the transformer. Another more JAXP-related option would be the setting for the `javax.xml.transform.ErrorListener` interface for catching transformation-related errors. The `ErrorListener` interface defines a list of methods:

```
public void warning (TransformerException exception)
    throws TransformerException;
public void error (TransformerException exception)
    throws TransformerException;
public void fatalError (TransformerException exception)
    throws TransformerException;
```

To use the `ErrorListener` interface, an implementation must be provided that implements the various error levels. This implementation of `ErrorListener` then is passed to the factory with a setter method called `setErrorListener()`:

```
factory.setErrorListener(new MyErrorListener());
```

Another useful option is to set the URI to handle URIs found in XML in order for them to be properly handled (for example, constructs such as `xsl:import` or `xsl:output`). The `URIResolver` interface defines one method:

```
public Source resolve (String href, String base) throws
TransformerException;
```

The implementation of the `URIResolver` interface can be set on the factory class with a method called `setURIResolver()`.

### ***Transforming XML***

Transforming XML consists of creating a new transformer instance by passing it an XSL template. The template defines the logic that needs to be applied to the XML input. In our example, a purchase order (`PurchaseOrder.xml`) is sent to the ACME Web Service retailer who processes the

---

## XML Processing and Data Binding with Java APIs 377

---

purchase order to determine what items were ordered. Based on the type of item ordered, the retailer creates new orders (`SWOrder.xml` and `HWOrder.xml`) that are sent to the appropriate warehouse. The following is a snippet of the code that instantiates the transformer:

```
TransformerFactory factory = TransformerFactory.newInstance();
factory.setErrorListener(new MyErrorListener());
Transformer transformer =
    factory.newTransformer(new StreamSource("order.xsl"));
```

The instantiation of a new transformer takes in a source stream representing the stylesheet for transforming XML. The source stream is the mechanism responsible for locating the stylesheet. The specification provides the implementation of the following classes as input sources:

**javax.xml.transform.stream.StreamSource.** Reads the input from I/O devices. The constructor accepts `InputStream`, `Reader`, and `String`.

**javax.xml.transform.dom.DOMSource.** Reads the input from a DOM tree (`org.w3c.dom.Node`).

**javax.xml.transform.sax.SAXSource.** Reads the input from SAX input source events (`org.xml.sax.InputSource` or `org.xml.sax.XMLReader`).

In the case where multiple stylesheets are applied to a single or multiple XML input, multiple transformers need to be instantiated for each stylesheet. Different types can be provided as input streams, such as URL, XML streams, DOM trees, SAX events, or custom data types. As for the output, it contains just as many possibilities with various combinations. The developer does not have to provide the same type of output as the input. It can be a variety of combinations that make this process very flexible and extensible. The following are some of the output classes provided by the specification:

**javax.xml.transform.stream.StreamResult.** Writes to an I/O device. The writer, `OutputStream`, is one of the available parameters.

**javax.xml.transform.dom.DOMResult.** Writes to a Document object (`org.w3c.dom.Document`).

**javax.xml.transform.sax.SAXResult.** Writes using `ContentHandler` callback methods.

## 378 Chapter 8

### *XSLT Sample Code*

Our sample scenario will use the `PurchaseOrder` XML input and will produce two resulting outputs. The resulting outputs will be orders that are sent to two different warehouses. Each warehouse specializes in different types of products: software or hardware. The idea behind this service is to provide a transformation capability to take the original order and split it into two different orders, depending upon the product's type. We will generate software and hardware XML orders that will be sent to the respective warehouse. Listing 8.6 is the `PurchaseOrder.xml` file.

```
<?xml version="1.0"?>
<PurchaseOrder>
  <Header>
    <PurchaseOrderNumber>2123536673005</PurchaseOrderNumber>
    <Date>02/22/2002</Date>
    <BuyerNumber>0002232</BuyerNumber>
  </Header>
  <!-- type indicates whether the item is a Software or Hardware
       product -->
  <Order>
    <LineItem type='SW'>
      <ProductNumber>221112</ProductNumber>
      <Quantity>250</Quantity>
    </LineItem>
    <LineItem type='HW'>
      <ProductNumber>343432</ProductNumber>
      <Quantity>12</Quantity>
    </LineItem>
    <LineItem type='HW'>
      <ProductNumber>210020</ProductNumber>
      <Quantity>145</Quantity>
    </LineItem>
    <LineItem type='SW'>
      <ProductNumber>33333</ProductNumber>
      <Quantity>145</Quantity>
    </LineItem>
  </Order>
</PurchaseOrder>
```

**Listing 8.6** `PurchaseOrder.xml`.

The following are the two XSL stylesheets for the software and hardware orders. They take in the `PurchaseOrder` and transform it to `SoftwareOrder.xml` and `HardwareOrder.xml`. In order to produce two types of outputs, the sample client uses two different stylesheets for producing two different outputs.

Listing 8.7 is the `hw-order.xsl` stylesheet.

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
<xsl:output method="xml" indent="no"/>
<!-- Header is needed in all warehouses -->
<xsl:template match="PurchaseOrder">
  <HardwareOrder>
    <xsl:apply-templates select="*" />
  </HardwareOrder>
</xsl:template>
<xsl:template match="Header">
  <ShippingInfo>
    <xsl:apply-templates select="ShippingAddress" />
  </ShippingInfo>
</xsl:template>

<xsl:template match="Order">
  <OrderInfo>
    <xsl:apply-templates select="*" />
  </OrderInfo>
</xsl:template>

<!-- Software and Hardware orders go to different warehouse locations. -
->
<xsl:template match="LineItem">
  <xsl:if test="@type='HW'">
    <ProductNo>
      <xsl:value-of select="ProductNumber" />
    </ProductNo>
    <ProductQty>
      <xsl:value-of select="Quantity" />
    </ProductQty>
  </xsl:if>
</xsl:template>
</xsl:stylesheet>
```

**Listing 8.7** `hw-order.xsl` stylesheet.

Listing 8.8 shows the `sw-order.xsl` stylesheet.

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
  <xsl:output method="xml" indent="no"/>
```

**Listing 8.8** `sw-order.xsl`. (*continues*)

**380 Chapter 8**

```
<!-- Header is needed in all warehouses -->
<xsl:template match="PurchaseOrder">
  <SoftwareOrder>
    <xsl:apply-templates select="*" />
  </SoftwareOrder>
</xsl:template>
<xsl:template match="Header">
  <ShippingInfo>
    <xsl:apply-templates select="ShippingAddress" />
  </ShippingInfo>
</xsl:template>
<xsl:template match="Order">
  <OrderInfo>
    <xsl:apply-templates select="*" />
  </OrderInfo>
</xsl:template>
<!-- Software and Hardware orders go to different warehouse locations. -->
<xsl:template match="LineItem">
  <xsl:if test="@type='SW'">
    <ProductNo>
      <xsl:value-of select="ProductNumber" />
    </ProductNo>
    <ProductQty>
      <xsl:value-of select="Quantity" />
    </ProductQty>
  </xsl:if>
</xsl:template>
</xsl:stylesheet>
```

**Listing 8.8** sw-order.xsl. (*continued*)

Listing 8.9 is a sample test class `MyXSLTExample.java` that performs the transformation and generates the output to the console for demonstration.

```
package jws.ch08.xslt;

import javax.xml.parsers.*;

import org.w3c.dom.*;
import org.xml.sax.*;
import java.io.*;
import javax.xml.transform.*;
import javax.xml.transform.stream.*;
```

**Listing 8.9** MyXSLTExample.java.

```
import javax.xml.transform.dom.*;

class MyXSLTExample {

    public MyXSLTExample () {}
    public static void main (String [] args) {

        System.out.println("Transforming .... ");
        try {
            TransformerFactory tFactory = TransformerFactory.newInstance();
            tFactory.setErrorListener(new MyErrorListener());
            if (tFactory.getFeature(DOMSource.FEATURE) &&
                tFactory.getFeature(DOMResult.FEATURE)) {
                //Instantiate a DocumentBuilderFactory.
                DocumentBuilderFactory dFactory =
                    DocumentBuilderFactory.newInstance();
                // And setNamespaceAware, which is required when parsing xsl files
                dFactory.setNamespaceAware(true);
                //Use the DocumentBuilderFactory to create a DocumentBuilder.
                DocumentBuilder dBuilder = dFactory.newDocumentBuilder();
                //Use the DocumentBuilder to parse the Software XSL stylesheet.
                Document xslSWDoc = dBuilder.parse("sw-order.xsl");
                //Use the DocumentBuilder to parse the Hardware XSL stylesheet.
                Document xslHWDoc = dBuilder.parse("hw-order.xsl");
                // Use the DOM Document to define a DOMSource object.
                DOMSource xslDomSWSource = new DOMSource(xslSWDoc);
                // Use the DOM Document to define a DOMSource object.
                DOMSource xslDomHWSource = new DOMSource(xslHWDoc);
                // Set the systemId: note this is actually a URL, not a local
                // filename
                xslDomSWSource.setSystemId("sw-order.xsl");
                // Set the systemId: note this is actually a URL, not a local
                // filename
                xslDomHWSource.setSystemId("hw-order.xsl");

                // Process the stylesheet DOMSource and generate a Transformer.
                Transformer swTransformer =
                    tFactory.newTransformer(xslDomSWSource);

                // Process the stylesheet DOMSource and generate a Transformer.
                Transformer hwTransformer =
                    tFactory.newTransformer(xslDomHWSource);

                //Use the DocumentBuilder to parse the XML input.
                Document xmlDoc = dBuilder.parse("PurchaseOrder.xml");

                // Use the DOM Document to define a DOMSource object.
                DOMSource xmlDocSource = new DOMSource(xmlDoc);
```

**Listing 8.9** MyXSLTExample.java. (*continues*)

**382 Chapter 8**

```

// Set the base URI for the DOMSource so any relative URIs it
// contains can be resolved.
xmlDomSource.setSystemId("PurchaseOrder.xml");

// write to System.out
System.out.println("\n--- Software Order ---\n");
swTransformer.transform(xmlDomSource,
    new StreamResult(System.out));
System.out.println("\n--- Hardware Order ---\n");
hwTransformer.transform(xmlDomSource,
    new StreamResult(System.out));
} else {
    System.out.println("Transformer does not support DOM source and
result!");
}
} catch (TransformerConfigurationException tce) {
    System.out.println("TransformerConfigurationException occurred :
"+tce);
} catch (ParserConfigurationException pce) {
    System.out.println("ParserConfigurationException occurred : "+pce);
} catch (TransformerException te) {
    System.out.println("TransformerException occurred : "+te);
} catch (SAXException se) {
    System.out.println("SAXException occurred : "+se);
} catch (IOException ioe) {
    System.out.println("IOException occurred : "+ioe);
}
}
}

```

**Listing 8.9** MyXSLTExample.java. (continued)

The output of the XSLT transformation sample is shown in Listing 8.10. It uses two distinct XSL stylesheets, `sw-order.xsl` and `hw-order.xsl`, to separate the software and hardware orders, respectively. The output shows a shipping address in both the orders and the Line Items.

```

----- Software Order -----

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

```

**Listing 8.10** Output of an XSLT transformation sample.

```
<ShippingInfo>
  233 St-John Blvd
  Boston

  MA
  03054
  USA
</ShippingInfo>
<OrderInfo>
  <ProductNo>221112</ProductNo>
  <ProductQty>250</ProductQty>
  <ProductNo>33333</ProductNo>
  <ProductQty>145</ProductQty>
</OrderInfo>
</SoftwareOrder>

----- Hardware Order -----
<?xml version="1.0" encoding="UTF-8"?>
<HardwareOrder>
  <ShippingInfo>
    233 St-John Blvd
    Boston
    MA
    03054
    USA
  </ShippingInfo>
  <OrderInfo>
    <ProductNo>343432</ProductNo>
    <ProductQty>12</ProductQty>
    <ProductNo>210020</ProductNo>
    <ProductQty>145</ProductQty>
  </OrderInfo>
</HardwareOrder>
```

**Listing 8.10** Output of an XSLT transformation sample. (*continued*)

## Threading

The JAXP specification does not mandate that the processor implementations provide thread safety. The common practice is to instantiate one processor instance per thread to avoid any synchronization issues. In the case where one instance is used by many threads, the application developer must provide the correct synchronized blocks so that the execution does not corrupt the data or produce unexpected results.

## Java Architecture for XML Binding (JAXB)

The XML binding technology provides application developers with a way to generate Java objects based on XML definitions and XML definitions based on Java objects. The Java Architecture for XML Binding (JAXB), formerly known as JCP project Adelard, is a high-level specification that defines an abstraction for binding semantics via classes and interfaces. XML data binding provides a way for applications to work with objects rather than complex data trees.

Converting XML to Java can be achieved by using parsing (with JAXP) and then constructing Java objects. The idea behind using a standard such as JAXB is to provide ease of use and performance enhancement. When developers are faced with the design and implementation of such solutions, in many cases they find themselves locked to their homegrown solutions with unnecessary complexities. This is potentially a maintenance nightmare where applications do not scale very well due to poor design decisions. A binding specification aims at taking all that away from the developer by implementing the various techniques in the parser where the classes are generated. It provides a standardized way to validate documents by defining the XML schema and data type mapping definitions to enforce correctness.

The current reference implementation of the JAXB 1.0 specification from Sun is limited to using DTDs as the schema language. This is a limitation that has a significant impact on today's industry requirements especially for Web services. This limitation is solved by XML schema support in the next version of JAXB. In this chapter, we will take a look at a JAXB-like Java/XML data binding framework named CASTOR. CASTOR is an open source project from Exolab ([castor.exolab.org](http://castor.exolab.org)). It provides a Java/XML data binding framework with W3C XML schema support. CASTOR is available for download from the Exolab Web site at <http://castor.exolab.org/>.

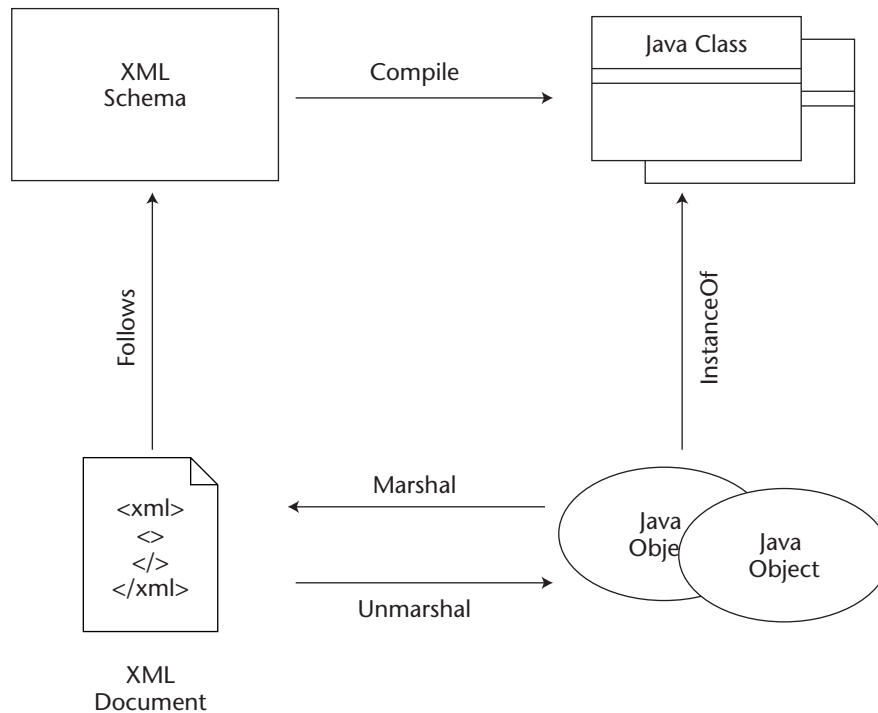
XML binding using CASTOR is done with the following steps:

1. Create a W3C XML schema defining the data structure, data types, and semantics of the XML data instance used in the application. In a Java class, it corresponds to a class variable or a property using accessor methods.
2. Using a CASTOR source code generator, transform the XML Schema to bind Java objects.

- Using the CASTOR implementation enables the binding Java objects to be transformed into XML instances (marshalling), and the XML instances then can be transformed into binding Java objects (unmarshalling) dynamically.

The binding runtime APIs then are used for converting XML data into Java objects and vice versa (see Figure 8.10). The Java objects then can be operated on as regular Java objects; creating and appending objects or deleting them is performed using standard Java semantics. This makes the work transparent of the XML hierarchy and much easier to work with.

- CASTOR provides a code generator, `org.exolab.castor.builder.SourceGenerator`, for generating Java binding objects from an XML Schema.
- CASTOR provides a full-fledged Java object and XML attribute mapping, which enables you to cast the data in the XML format to the proper Java type and vice-versa. The complete list of supported Java object types and XML Schema attributes are available at <http://castor.exolab.org/xml-mapping.html>.



**Figure 8.10** XML binding life cycle.

## 386 Chapter 8

The following section will walk you through an example taking a modified version of our `PurchaseOrder.xml` and using binding techniques to generate domain objects representing data stored in the XML input. The data-binding runtime used for this will be the CASTOR implementation, which provides support for an XML Schema unlike a JAXB reference implementation, which is limited to DTDs. Due to this limitation (as of this book's writing), the specification is going through a major revision. It will soon provide a much richer feature set aligned more with the current needs.

### Data Binding Generation

In order to create Java classes, an XML data schema must be created. For our example, we will use the following Purchase Order XML data file (see Listing 8.11) and generate a validation W3C XML Schema.

```
<?xml version="1.0"?>
<PurchaseOrder>
  <Header>
    <PurchaseOrderNumber>2123536673005</PurchaseOrderNumber>
    <Date>02/22/2002</Date>
    <BuyerNumber>0002232</BuyerNumber>
    <BuyerName>John Doe</BuyerName>
    <BuyerAddress>
      <Street>233 St-John Blvd</Street>
      <City>Boston</City>
      <State>MA</State>
      <Zip>03054</Zip>
      <Country>USA</Country>
    </BuyerAddress>
    <ShippingAddress>
      <Street>233 St-John Blvd</Street>
      <City>Boston</City>
      <State>MA</State>
      <Zip>03054</Zip>
      <Country>USA</Country>
    </ShippingAddress>
  </Header>
</PurchaseOrder>
```

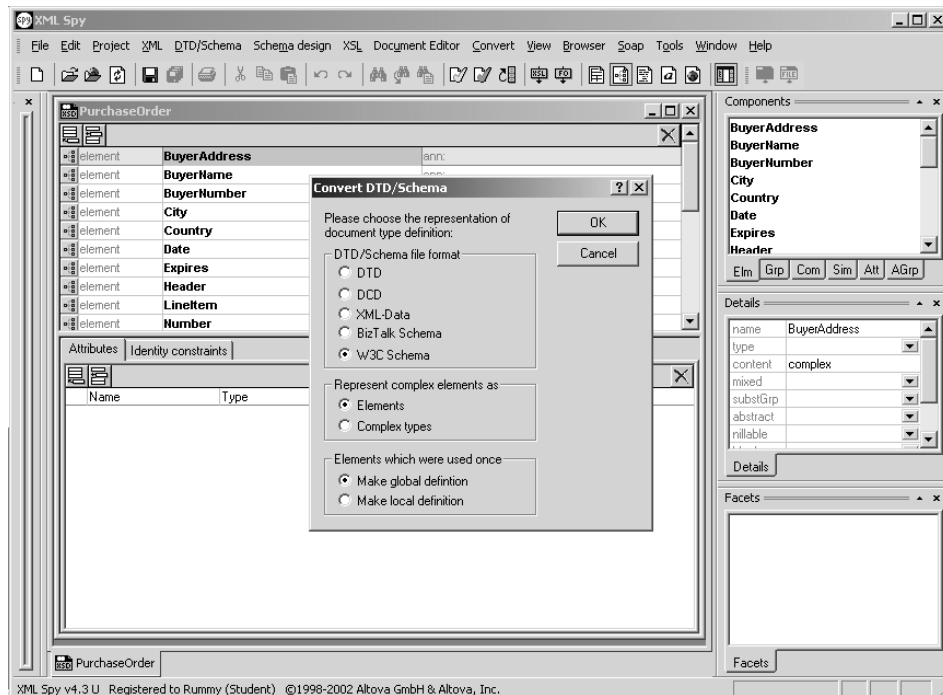
**Listing 8.11** PurchaseOrder.xml file.

```
<PaymentInfo>
  <Type>Visa</Type>
  <Number>0323235664664564</Number>
  <Expires>02/2004</Expires>
  <Owner>John Doe</Owner>
</PaymentInfo>
</Header>
<!-- type indicates whether the item is a Software or Hardware product
-->
<Order>
  <LineItem type='SW'>
    <ProductNumber>221112</ProductNumber>
    <Quantity>250</Quantity>
  </LineItem>
  <LineItem type='HW'>
    <ProductNumber>343432</ProductNumber>
    <Quantity>12</Quantity>
  </LineItem>
  <LineItem type='HW'>
    <ProductNumber>210020</ProductNumber>
    <Quantity>145</Quantity>
  </LineItem>
  <LineItem type='SW'>
    <ProductNumber>33333</ProductNumber>
    <Quantity>145</Quantity>
  </LineItem>
</Order>
</PurchaseOrder>
```

**Listing 8.11** PurchaseOrder.xml file. (continued)

Once the XML input is defined, we can create an XML Schema file to be used for generating the supporting data binding classes. This can be accomplished manually or by using the functionality provided by an XML-based IDE (for example, XML Spy, Breeze XML, Oracle XDK, or Jbuilder). Figure 8.11 is a screenshot of the XML Spy 4.2 screen that enables us to generate the XML Schema for our input file.

## 388 Chapter 8



**Figure 8.11** XML Schema generation using XML Spy 4.2.

Listing 8.12 shows the purchase order `PurchaseOrder.xsd` XML Schema generated from the `PurchaseOrder.xml` input.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
  <xs:element name="BuyerAddress">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Street"/>
        <xs:element ref="City"/>
        <xs:element ref="State"/>
        <xs:element ref="Zip"/>
        <xs:element ref="Country"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

**Listing 8.12** 'PurchaseOrder.xsd' XML Schema generated from the `PurchaseOrder.xml` input.



**390 Chapter 8**

```
<xs:element ref="Number" />
<xs:element ref="Expires" />
<xs:element ref="Owner" />
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="ProductNumber">
  <xs:simpleType>
    <xs:restriction base="xs:int">
    </xs:restriction>
  </xs:simpleType>
</xs:element>
<xs:element name="PurchaseOrder">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="Header" />
      <xs:element ref="Order" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="PurchaseOrderNumber" type="xs:long" />
<xs:element name="Quantity">
  <xs:simpleType>
    <xs:restriction base="xs:short">
    </xs:restriction>
  </xs:simpleType>
</xs:element>
<xs:element name="ShippingAddress">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="Street" />
      <xs:element ref="City" />
      <xs:element ref="State" />

      <xs:element ref="Zip" />
      <xs:element ref="Country" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="State" type="xs:string" />
<xs:element name="Street" type="xs:string" />
<xs:element name="Type" type="xs:string" />
<xs:element name="Zip" type="xs:short" />
</xs:schema>
```

**Listing 8.12** 'PurchaseOrder.xsd' XML Schema generated from the Purchase Order.xml input. (*continued*)

## XML Processing and Data Binding with Java APIs 391

Now that we have an XML Schema definition, we can generate a set of supporting Java classes. The following is an example:

```
java org.exolab.castor.builder.SourceGenerator -i PurchaseOrder.xsd
-package jws.ch08.castor
```

This code will generate a set of Java binding object source files and Java mapping descriptors from the XML Schema, `PurchaseOrder.xsd`, and it will place them in the `jws.ch08.castor.*` package. This package contains the accessor methods, which also include the 'marshal' and 'unmarshal' methods required for transforming an XML instance to Java objects and vice-versa.

The source code generator also provides the ability to use the following types of collections when generating source code:

- **Java 1.1 (default).** `java.util.Vector`.
- **Java 1.2.** Use the option types `-j2`. The collection type is `java.util.Collection`.

The generated classes reflect the definition of the XML Schema. After running the schema compiler (`org.exolab.castor.builder.SourceGenerator`), we get the following classes:

- `PurchaseOrder.java`
- `Header.java`
- `BuyerAddress.java`
- `ShippingAddress.java`
- `PaymentInfo.java`
- `Order.java`
- `LineItem.java`

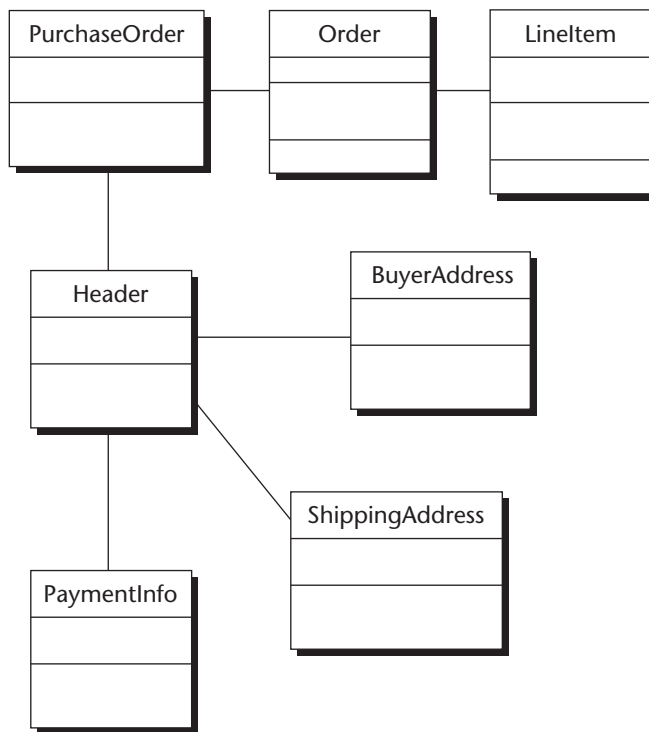
These classes contain the getter and setter methods for all of the attributes defined in the schema. In addition to these accessor methods, there are callback methods that contain the logic to create XML from Java or Java from XML. These terms are referred to as marshalling and unmarshalling, respectively. The idea here is to work with the Java classes and not XML. The `PurchaseOrder` class contains references to `Header` and `Order` classes. The `Order` class is composed of one or many `LineItems`, which compose the `Order`. The `Header` contains information about the buyer, such as buyer's address (`BuyerAddress`), shipping address (`ShippingAddress`), and

**392 Chapter 8**

payment information (PaymentInfo). Figure 8.12 is a class diagram that shows the relationship between all of the classes involved in building a PurchaseOrder.

The following is a list of the methods generated for PurchaseOrder.java. The first part of the class consists of the accessor methods for the Header and Order objects.

```
public class PurchaseOrder implements java.io.Serializable {  
    private Header _header;  
    private Order _order;  
  
    public PurchaseOrder()  
    public Header getHeader()  
    public Order getOrder()  
    public void setHeader(Header header)  
    public void setOrder(Order order)
```



**Figure 8.12** Purchase order class diagram.

The rest of the methods are callbacks for binding specific tasks:

```
public boolean isValid()
public void marshal(java.io.Writer out)
    throws MarshalException, ValidationException
public void marshal(org.xml.sax.DocumentHandler handler)
    throws MarshalException, ValidationException

public static PurchaseOrder unmarshal(java.io.Reader reader)
    throws MarshalException, ValidationException

public void validate()throws ValidationException
```

## Marshalling XML

When the JAXB provider (CASTOR) is used, the Java object can be dynamically transformed to an XML file. This transformation is based on the binding mapping of the XML Schema, which determines how the given object's property has to be transformed to an XML element (for example, the element or attribute). This process is called marshalling. The following is an extract from the client test program that makes a call to the `PurchaseOrder` `marshal` method to read an XML input and to transform it to an object representation.

```
PurchaseOrder purchaseOrder;
purchaseOrder= PurchaseOrder.marshal(new
    FileWriter("UpdatePurchaseOrder.xml"));
```

Now that the object is created, we can add or remove components. If we want to add a new `LineItem` object to the graph, it is very simple:

1. Create a `LineItem` object.
2. Set the attributes (`ProductNumber` or `Quantity`).
3. Get a reference to the `Order` object stored in `PurchaseOrder`.
4. Add the `LineItem` to the `Order` by calling its setter method.
5. Add the `Order` back to the `PurchaseOrder` and voila!
6. The following code shows how to add a `LineItem` to the purchase order:

```
/* create the new line Item */
LineItem item = new LineItem();
```

## 394 Chapter 8

---

```
item.setProductNumber("123456");
item.setQuantity("200");
/* SW or HW */
item.setType("SW");

/* get a reference to order object */
Order order = PurchaseOrder.getOrder();
/* set the new line Item */
Order.setLineItem(item);
/* set the updated order */
PurchaseOrder.setOrder(order);
```

To remove an Object from the graph is just as easy. Suppose we want to remove the `LineItem` with a product number of "33333". Here are the steps to do so:

1. Create a `LineItem` object.
2. Set the attributes (`ProductNumber` or `Quantity`).
3. Get a reference to the `Order` object stored in `PurchaseOrder`.
4. Call `removeLineItem()` passing the `LineItem` that we want to delete.
5. The following is the code that removes the `LineItem` from the purchase order:

```
/*Create the line Item to remove */
LineItem item = new LineItem();
item.setProductNumber("33333");
item.setQuantity("145");
item.setType("SW");

/* get order from purchase order */
Order order = PurchaseOrder.getOrder();

/* remove the line Item from list */
Order.removeLineItem(item);

/* set the updated order */
PurchaseOrder.setOrder(order);
```

We can now save the new changes back to the same or different file. This process is called unmarshalling and is examined in the following section.

## Unmarshalling Java

The XML instance created by an application can be dynamically transformed to a Java object based on the mapping XML schema, which determines how a given XML element/attribute is transformed into the Java object model. This is performed by the Java introspection to determine the function form `getXxxYyy()` or `setXxxYyy(<type> z)`. The accessor then is associated with an XML element/attribute named 'xxx-yyy', which is based on the mapping XML schema. This process is referred to as unmarshalling.

In order to unmarshal the XML data into a Java class graph, we must use the callback method provided in the `PurchaseOrder` class:

```
PurchaseOrder purchaseOrder;  
purchaseOrder = PurchaseOrder.unmarshal(new  
    FileReader("PurchaseOrder.xml"));
```

If we take the example where a new `LineItem` was added and then unmarshaled, the `PurchaseOrder` that we would get is shown in the following output:

```
<Order>  
  <LineItem type='SW'>  
    <ProductNumber>221112</ProductNumber>  
    <Quantity>250</Quantity>  
  </LineItem>  
  <LineItem type='HW'>  
    <ProductNumber>343432</ProductNumber>  
    <Quantity>12</Quantity>  
  </LineItem>  
  <LineItem type='HW'>  
    <ProductNumber>210020</ProductNumber>  
    <Quantity>145</Quantity>  
  </LineItem>  
  <LineItem type='SW'>  
    <ProductNumber>33333</ProductNumber>  
    <Quantity>145</Quantity>  
  </LineItem>  
  <LineItem type='SW'>  
    <ProductNumber>123456</ProductNumber>  
    <Quantity>200</Quantity>  
  </LineItem>  
</Order>
```

## 396 Chapter 8

### Other Callback Methods

Additional methods are generated by the binding compiler to provide further functionality. Validation is something that occurs before the data objects are written to the output source (that is, the XML file). The binding compiler generates `validate()` and `isValid()` methods. The `isValid()` method is a wrapper of `validate` but returns `true` or `false`. The `validate` method itself throws a `ValidateException` if the new data graph does not conform to the XML schema.

This covers the surface of XML binding and should be sufficient for performing most of the binding operations. The most challenging part in binding is to generate a correct XML Schema definition. Once this is achieved, the rest is magic. Most of the operation is handled by the runtime of the binding provider. This is a significant improvement over doing it manually, where fewer method calls are needed, thus resulting in a much cleaner code. The code in the following section is the complete list of code used in this section.

### Sample Code for XML Binding

The source code shown in Listing 8.13 uses the `PurchaseOrder.xml` and `PurchaseOrder.xsd` files used in previous examples.

The sample data-binding client code unmarshals the XML input source into a Java object representation. The code then traverses the `Object` and prints out all the `LineItem` objects of the `Order`. Finally, it adds a new `LineItem` object to the vector and marshals the object back to an XML format, saving it as `MyPurchaseOrder.xml`.

```
package jws.ch08.castor;

import java.io.*;
import org.exolab.castor.xml.ClassDescriptorResolver; import
org.exolab.castor.xml.Unmarshaller;
import org.exolab.castor.xml.Marshaller;
import org.exolab.castor.xml.MarshalException;
import org.exolab.castor.xml.util.ClassDescriptorResolverImpl;

import java.beans.PropertyChangeEvent;
import java.beans.PropertyChangeListener;
import org.exolab.castor.types.Duration;
```

**Listing 8.13** ProductOrderClient.java.

---

**XML Processing and Data Binding with Java APIs 397**

---

```
import org.exolab.castor.types.Date;
import org.exolab.castor.types.Time;

public class PurchaseOrderClient implements PropertyChangeListener {

    public void propertyChange(PropertyChangeEvent event) {
        System.out.println("PropertyChange: " + event.getPropertyName());
    } //-- propertyChange

    public static void main(String[] args) {
        try {
            System.out.println("Unmarshalling Purchase Order");
            PurchaseOrder purchaseOrder = null;
            purchaseOrder = PurchaseOrder.unmarshal(new
                FileReader("PurchaseOrder.xml"));
            System.out.println();
            System.out.println("unmarshalled...performing tests...");
            System.out.println();
            System.out.println("Getting Header ...");
            Header header = purchaseOrder.getHeader();
            System.out.println("Getting Buyer Address ...");
            BuyerAddress buyerAddress = header.getBuyerAddress();
            System.out.println("Getting Shipping Address ...");
            ShippingAddress shipAddress = hader.getShippingAddress();
            System.out.println("Getting Order ...");
            Order order = purchaseOrder.getOrder();

            ListItem [] itemList = order.getListItem();
            for (int i=0; i<itemList.length; i++) {
                //-- Display unmarshalled address to the screen
                System.out.println("Purchase Order - Unmarshalling from XML to
                JavaObject");
                System.out.println("-----");
                System.out.println();

                System.out.println("Order Type");
                System.out.println(" " + itemList[i].getType());

                System.out.println("Product Number");
                System.out.println(" "+

                itemList[i].getProductNumber());

                System.out.println("Quantity:");
                System.out.println(" " + itemList[i].getQuantity());
            }
        }
    }
}
```

**Listing 8.13** ProductOrderClient.java. (*continues*)

**398 Chapter 8**

```
    }

    System.out.println("==Unmarshalling complete==");
    System.out.println("Add a new Item ...");
    LineItem newItem = new LineItem();
    newItem.setProductNumber(98234);
    newItem.setQuantity((short)666);
    newItem.setType("HW");

    System.out.println("Set item in order");
    int index = order.getLineItemCount();
    System.out.println("Count before:"+index);
    order.addLineItem(newItem);

    System.out.println("Count before:"+
        order.getLineItemCount());

    System.out.println("Set order in purchase order");
    purchaseOrder.setOrder(order);

    System.out.println("\n\n=Marshalling PrescriberMessage Java object
to XML =");
    purchaseOrder.marshal(new FileWriter("MyPurchaseOrder.xml"));

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

**Listing 8.13** ProductOrderClient.java. (*continued*)

The client generates the output shown in Listing 8.14. This code displays the contents of an Order and adds a new Order saving it to MyPurchaseOrder.xml.

```
Getting Header ...
Getting Buyer Address ...
Getting Shipping Address ...
Getting Order ...
Purchase Order - Unmarshalling from XML to JavaObject
-----
Order Type
    SW
Product Number
    221112
```

**Listing 8.14** A sample binding output.

```
Quantity:
    250
Purchase Order - Unmarshalling from XML to JsonObject
-----
Order Type
    HW
Product Number
    343432
Quantity:
    12
Purchase Order - Unmarshalling from XML to JsonObject
-----
Order Type
    HW
Product Number
    210020
Quantity:
    145
Purchase Order - Unmarshalling from XML to JsonObject
-----
Order Type
    SW
Product Number
    33333
Quantity:
    145
=====UnMarshalling complete=====
Add a new Item ...
Set item in order
Count before adding the item: 4
Count after adding item : 5
Set order in purchase order
```

**Listing 8.14** A sample binding output. (*continued*)

## Summary

---

You now should have a better understanding of Java APIs for XML processing and binding. These APIs provide the functionality that is essential in developing Java Web services.

In this chapter, we addressed Java APIs for XML processing and binding. We covered such varied topics as XML and XSL basics, JAXP and its APIs, how to process XML data with SAX and DOM, how to parse and transform XML documents using the JAXP API, and data binding between Java and XML using CASTOR.

