2 XML and Java

This chapter

- Describes relevant XML standards and technologies
- Classifies XML tools in terms of functionality
- Introduces and demonstrates use of Java XML Pack APIs (JAX)
- Suggests how JAX APIs are best deployed in your architecture

A complex set of closely related XML APIs, each of which is either in specification or development, is the result of a flurry of Java community development activity in the area of XML. These APIs include the JAX family, as well as other popular emerging standards such as JDOM.

This chapter untangles the web of Java APIs for XML, identifying and classifying each in terms of its functionality, intended use, and maturity. Where possible, we provide usage examples for each new API and describe how it might be best used in your J2EE system. We also identify areas in which the APIs overlap and suggest which ones are likely to be combined or eliminated in the future. Subsequent chapters build upon your understanding of these APIs by providing more specific examples of their implementation.

To fully appreciate the capabilities and limitations of the current JAX APIs, section 2.1 provides a brief overview of the state of important XML technologies. These technologies and standards are implemented and used by the JAX APIs, so understanding something about each will speed your mastery of JAX.

2.1 XML and its uses

Before diving into the details of Java's XML API family, a brief refresher on a few important XML concepts is warranted. This section provides such a refresher, as well as an overview of the most important recent developments in XML technology.

XML, the eXstensible Markup Language, is not actually a language in its own right. It is a metalanguage used to construct other languages. XML is used to create structured, self-describing documents that conform to a set of rules created for each specific language. XML provides the basis for a wide variety of industry- and discipline-specific languages. Examples include Mathematical Markup Language (MathML), Electronic Business XML (ebXML), and Voice Markup Language (VXML). This concept is illustrated in figure 2.1.

XML consists of both markup and content. *Markup*, also referred to as *tags*, describes the content represented in the document. This flexible representation of data allows you to easily send and receive data, and transform data from one format to another. The uses of XML are rapidly expanding and are partially the impetus for writing this book. For example, business partners use XML to exchange data with each other in new and easier ways. E-business related information such as pricing, inventory, and transactions are represented in XML and transferred over the Internet using open standards and protocols. There are also many specialized uses of XML, such as the Java Speech Markup Language and the Synchronized Multimedia Integration Language.

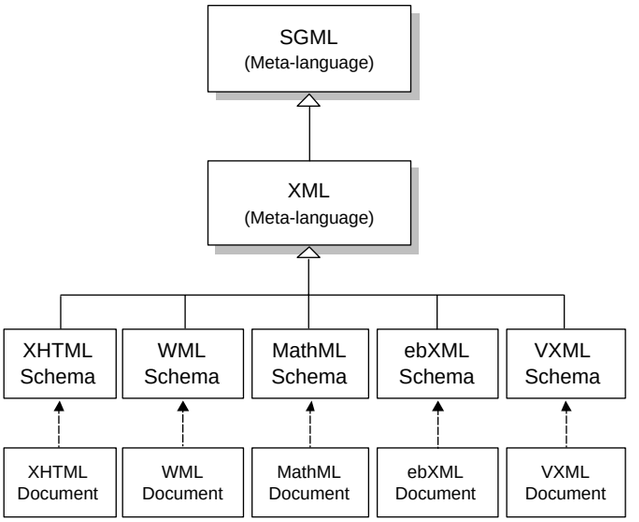

Figure 2.1 XML language hierarchy

Each XML language defines its own *grammar*, a specific set of rules governing the content and structure of documents written in that language. For example, the element price may be valid in an ebXML document but has no meaning in a MathML document. Since each language must fulfill this grammatical requirement, XML provides facilities for generically documenting the correct grammar of any derived language. Any XML parser can validate the structure of any XML document, given the rules of its language.

Using XML as a common base for higher-level languages enables the interchange of data between software components, systems, and enterprises. Parsing and translation tools written to handle any type of XML-based data can be employed to create and manipulate data in a uniform way, regardless of each document's semantic meaning. For example, the same XML parser can be used to read a MathML document and an ebXML document, and the same XML Translator can be used to convert an ebXML purchase order document into a RosettaNet PIP document.

An XML-based infrastructure enables high levels of component reuse and interoperability in your distributed system. It also makes your system interfaces cleaner and more understandable to those who must maintain and extend it. And since XML is an industry standard, it can be deployed widely in your systems without worry about vendor dependence. XML also makes sense from the standpoint of systems integration, as an alternative to distributed object interaction. It allows data-level integration, making the coupling

between your application and other systems much looser and enhancing overall architectural flexibility.

In addition to its uses in messaging and data translation, XML can also be used as a native data storage format in some situations. It is particularly well suited for managing document repositories and hierarchical data. We examine some of the possibilities in this area in chapter 3.

An example XML document

To illustrate the power and flexibility of XML and related technologies, we need a concrete XML example with which to work. We use this simple document throughout the rest of this chapter to illustrate the use of various XML technologies. Most importantly, we use it to demonstrate the use of the JAX APIs in section 2.2.

Listing 2.1 contains an XML *instance document*, a data structure containing information about a specific catalog of products.

Listing 2.1 Product XML document example

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

<product-catalog>
  <product sku="123456" name="The Product">
    <description locale="en_US">
      An excellent product.
    </description>
    <description locale="es_MX">
      Un producto excellente.
    </description>
    <price locale="en_US" unit="USD">
      99.95
    </price>
    <price locale="es_MX" unit="MXP">
      9999.95
    </price>
  </product>
</product-catalog>
```

Defines a product with SKU=123456 and the name "The Product"

❶ Lists descriptions and prices for this product in the U.S. and Mexico

❶ Shows a catalog containing a single product. The product information includes its name, SKU number, description, and price. Note that the document contains multiple price and description nodes, each of which is specific to a locale.

Classifying XML technologies

There are numerous derivative XML standards and technologies currently under development. These are not specific to Java, or any other implementation

language for that matter. They are being developed to make the use of XML easier, more standardized, and more manageable. The widespread adoption of many of them is critical to the success of XML and related standards.

This section provides a brief overview of the most promising specifications in this area. Since it is impossible to provide exhaustive tutorials for each of these in this section, we recommend you visit http://www.zvon.org, a web site with excellent online tutorials for many of these technologies.

2.1.1 XML validation technologies

The rules of an XML language can be captured in either of two distinct ways. When codified into either a document type definition or an XML schema definition, any validating XML parser can enforce the rules of a particular XML dialect generically. This removes a tremendous burden from your application code. In this section, we provide a brief overview of this important feature of XML.

Document type definitions

The first and earliest language definition mechanism is the document type definition (DTD).

DEFINITION A *document type definition* is a text file consisting of a set of rules about the structure and content of XML documents. It lists the valid set of elements that may appear in an XML document, including their order and attributes.

A DTD dictates the hierarchical structure of the document, which is extremely important in validating XML structures. For example, the element `Couch` may be valid within the element `LivingRoom`, but is most likely not valid within the element `BathRoom`. DTDs also define element attributes very specifically, enumerating their possible values and specifying which of them are required or optional.

Listing 2.2 DTD for the product catalog example document

```
<!ELEMENT product-catalog (product+)>

<!ELEMENT product (description+, price+)>
  <!ATTLIST product
    sku ID #REQUIRED
    name CDATA #REQUIRED
  >

<!ELEMENT description (#PCDATA)>
```

Product catalogs must contain one or more products and each product has one or more descriptions and one or more prices

Each product must have a SKU and name attribute

```
  <!ATTLIST description
    locale CDATA #REQUIRED
  >
<!ELEMENT price (#PCDATA)>
  <!ATTLIST price
    locale CDATA #REQUIRED
    unit CDATA #REQUIRED
  >
```

Listing 2.2 contains a DTD to constrain our product catalog example document. For this DTD to be used by a validating XML parser, we could add the DTD in-line to listing 2.1, right after the opening XML processing instruction. We could also store the DTD in a separate file and reference it like this:

```
<!DOCTYPE product-catalog SYSTEM "product-catalog.dtd">
```

Using this statement, a validating XML parser would locate a file named `product-catalog.dtd` in the same directory as the instance document and use its contents to validate the document.

XML Schema definitions

Although a nice first pass at specifying XML languages, the DTD mechanism has numerous limitations that quickly became apparent in enterprise development. One basic and major limitation is that a DTD is not itself a valid XML document. Therefore it must be handled by XML parsing tools in a special way.

More problematic, DTDs are quite limited in their ability to constrain the structure and content of XML documents. They cannot handle namespace conflicts within XML structures or describe complex relationships among documents or elements. DTDs are not modular, and constraints defined for one data element cannot be reused (inherited) by other elements. For these reasons and others, the World Wide Web Consortium (W3C) is working feverishly to replace the DTD mechanism with XML Schema.

DEFINITION An *XML Schema definition (XSD)* is an XML-based grammar declaration for XML documents.

XSD is itself an XML language. Using XSD, data constraints, hierarchical relationships, and element namespaces can be specified more completely than with DTDs. XML Schema allows very precise definition of both simple and complex data types, and allows types to inherit properties from other types.

There are numerous common data types already built into the base XML Schema language as a starting point for building specific languages. Listing 2.3 shows a possible XML Schema definition for our example product catalog document.

Listing 2.3 An XSD for the product catalog document

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

  <xsd:element type="product-catalog"/>

<xsd:complexType name="productCatalog">
  <xsd:element type="productType"
      minOccurs="1"/>
  </xsd:complexType>

  <xsd:complexType name="productType">
   <xsd:element name="description"
        type="xsd:string" minOccurs="1">
      <xsd:attribute name="locale"
        type="xsd:string"/>
    </xsd:element>
    <xsd:element name="price"
        type="xsd:decimal" minOccurs="1">
      <xsd:attribute name="locale"
        type="xsd:string"/>
      <xsd:attribute name="unit"
        type="xsd:string"/>
    </xsd:element>
   <xsd:attribute name="sku"
        type="xsd:decimal"/>
   <xsd:attribute name="name"
        type="xsd:string"/>
  </xsd:complexType>

</xsd:schema>
```

"xsd" namespace defined by XML Schema

Declares the product catalog

Defines catalog type containing one or more product elements

❶ **product type definition**

❶ This XSD defines a complex type called `productType`, which is built upon other primitive data types. The complex type contains attributes and other elements as part of its definition. Just from the simple example, the advantages of using XML Schema over DTDs should be quite apparent to you.

The example XSD in listing 2.3 barely scratches the surface of the intricate structures that you can define using XML Schema. Though we will not focus on validation throughout this book, we strongly encourage you to become proficient at defining schemas. You will need to use them frequently as the use

of XML data in your applications increases. Detailed information on XML Schema can be found at http://www.w3c.org/XML/Schema.

Before leaving the topic of document validation, we note that some parsers do not offer any validation at all, and others only support the DTD approach. Document validation is invaluable during development and testing, but is often turned off in production to enhance system performance. Using validation is also critical when sharing data between enterprises, to ensure both parties are sending and receiving data in a valid format.

2.1.2 XML parsing technologies

Before a document can be validated and used, it must be parsed by XML-aware software. Numerous XML parsers have been developed, including Crimson and Xerces, both from the Apache Software Foundation. You can learn about these parsers at http://xml.apache.org. Both tools are open source and widely used in the industry. Many commercial XML parsers are also available from companies like Oracle and IBM.

DEFINITION An *XML parser* is a software component that can read and (in most cases) validate any XML document. A parser makes data contained in an XML data structure available to the application that needs to use it.

SAX

Most XML parsers can be used in either of two distinct modes, based on the requirements of your application. The first mode is an event-based model called the Simple API for XML (SAX). Using SAX, the parser reads in the XML data source and makes callbacks to its client application whenever it encounters a distinct section of the XML document. For example, a SAX event is fired whenever the end of an XML element has been encountered. The event includes the name of the element that has just ended.

To use SAX, you implement an event handler for the parser to use while parsing an XML document. This event handler is most often a state machine that aggregates data as it is being parsed and handles subdocument data sets independently of one another. The use of SAX is depicted in figure 2.2. SAX is the fastest parsing method for XML, and is appropriate for handling large documents that could not be read into memory all at once.

One of the drawbacks to using SAX is the inability to look forward in the document during parsing. Your SAX handler is a state machine that can only

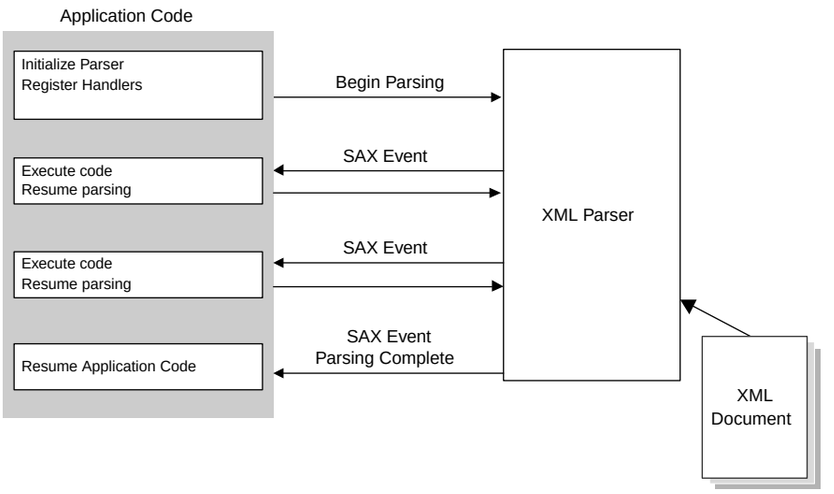

Figure 2.2 Using the SAX API

operate on the portion of the XML document that has already been parsed. Another disadvantage is the lack of predefined relationships between nodes in the document. In order to perform any logic based on the parent or sibling nodes, you must write your own code to track these relationships.

DOM

The other mode of XML parsing is to use the Document Object Model (DOM) instead of SAX. In the DOM model, the parser will read in an entire XML data source and construct a treelike representation of it in memory. Under DOM, a pointer to the entire document is returned to the calling application. The application can then manipulate the document, rearranging nodes, adding and deleting content as needed. The use of DOM is depicted in figure 2.3.

While DOM is generally easier to implement, it is far slower and more resource intensive than SAX. DOM can be used effectively with smaller XML data structures in situations when speed is not of paramount importance to the application. There are some DOM-derivative technologies that permit the use of DOM with large XML documents, which we discuss further in chapter 3.

As you will see in section 2.2, the JAXP API enables the use of either DOM or SAX for parsing XML documents in a parser-independent manner. Deciding which method to use depends on your application's requirements for speed, data manipulation, and the size of the documents upon which it operates.

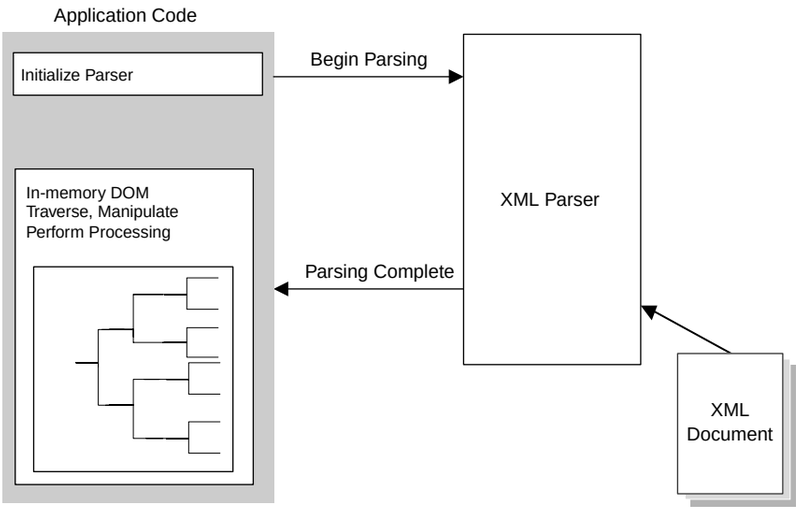

Figure 2.3 Using the DOM API

2.1.3 XML translation technologies

A key advantage of XML over other data formats is the ability to convert an XML data set from one form to another in a generic manner. The technology that enables this translation is the eXtensible Stylesheet Language for Transformations (XSLT).

XSLT

Simply stated, XSLT provides a framework for transforming the structure of an XML document. XSLT combines an input XML document with an XSL stylesheet to produce an output document.

DEFINITION An *XSL stylesheet* is a set of transformation instructions for converting a *source* XML document to a *target* output document.

Figure 2.4 illustrates the XSLT process. Performing XSLT transformations requires an XSLT-compliant processor. The most popular open source XSLT engine for Java is the Apache Software Foundation's Xalan project. Information about Xalan can be found at http://xml.apache.org/xalan-j.

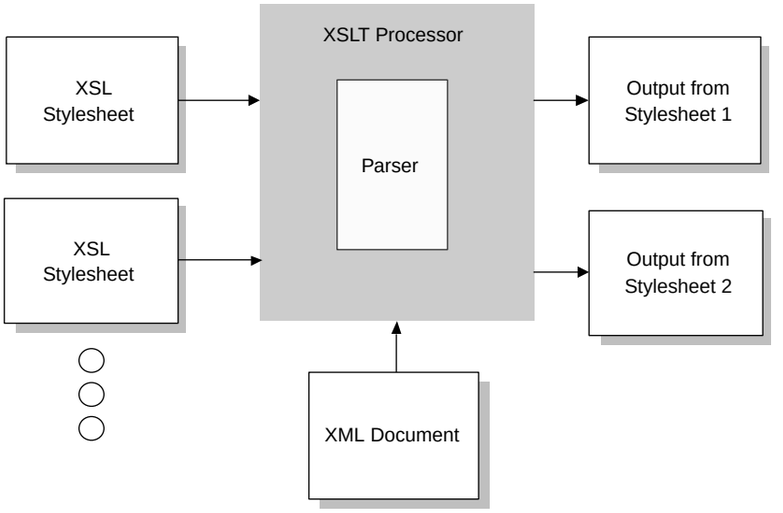

Figure 2.4 XSLT processing overview

An XSLT processor transforms an XML source tree by associating patterns within the source document with XSL stylesheet templates that are to be applied to them. For example, consider the need to transform our product catalog XML document into HTML for rendering purposes. This consists of wrapping the appropriate product data in the XML document with HTML markup. Listing 2.4 shows an XSL stylesheet that would accomplish this task.

Listing 2.4 Translating the product catalog for the Web

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

<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">

<xsl:template match="/">
<html>
  <head><title>My Products</title></head>
  <body>
    <h1>Products Currently For Sale in the U.S.</h1>
  <xsl:for-each select="//product">
    <xsl:value-of select="@name"/> : $
    <xsl:value-of select="./price[@unit='USD']"/> USD
 </xsl:for-each>
```

❶ **Executes for the root element of the source document**

❷ **Prints name and price information**

```
  </body>
</html>
</xsl:template>

</xsl:stylesheet>
```

❶ The `match` attribute is an XPath expression meaning the root XML element. This template is therefore executed against the entire source document.

❷ Each product element in the source document will have its name attribute printed, followed by the string: $, its price in dollars, and the string USD.

XSLT processors can vary in terms of their performance characteristics. Most offer some way to precompile XSL stylesheets to reduce transformation times. As you will see in section 2.2, the JAXP API provides a layer of pluggability for compliant XSLT processors in a manner similar to parsers. This permits the replacement of one XSLT engine with another, faster one as soon as it becomes available.

Details on XSLT can be found at http://www.w3.org/Style/XSL.

Binary transformations for XML

Note that the capabilities of XSLT are not limited to textual transformations. It is often necessary to translate textual data to binary format. A common example is the translation of business data to PDF format for display. For this reason the XSL 1.0 Recommendation also specifies a set of *formatting objects*. Formatting objects are instructions that define the layout and presentation of information. Formatting objects are most useful for print media and design work. Some Java libraries are already available to do the most common types of transformations. See chapter 5 for an example of the most common binary transformation required today, from XML format to PDF.

2.1.4 Messaging technologies

Numerous technologies for transmitting XML-structured data between applications and enterprises are currently under development. This is due to the tremendous potential of XML to bridge the gap between proprietary data formats and messaging protocols. Using XML, companies can develop standard interfaces to their systems and services to which present and future business partners can connect with little development effort. In this section, we provide a brief description of the most promising of these technologies.

SOAP

By far the most promising advances in this area are technologies surrounding the Simple Object Access Protocol (SOAP).

DEFINITION *SOAP* is a messaging specification describing data encoding and packaging rules for XML-based communication.

The SOAP specification describes how XML messages can be created, packaged, and transmitted between systems. It includes a binding (mapping) for the HTTP protocol, meaning that SOAP messages can be transmitted over existing Web systems. Much of SOAP is based upon *XML-RPC*, a specification describing how remote procedure calls can be executed using XML.

SOAP can be implemented in a synchronous (client / server) or asynchronous fashion. The synchronous method (RPC-style) involves a client explicitly requesting some XML data from a SOAP server by sending a SOAP request message. The server returns the requested data to the client in a SOAP response message. This is depicted in figure 2.5.

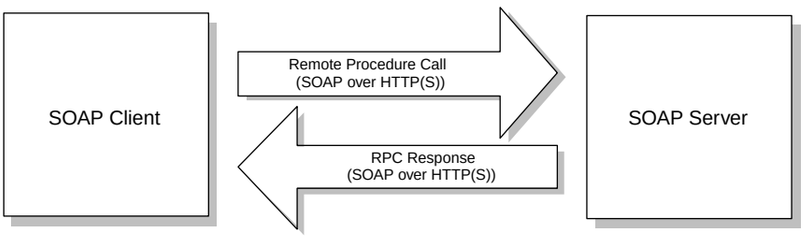

Figure 2.5 RPC-style SOAP messaging

Asynchronous messaging is also fully supported by the SOAP specification. This can be useful in situations where updates to information can be sent and received as they happen. The update event must not require an immediate response, but an asynchronous response might be sent at some point in the future. This response might acknowledge the receipt of the original message and report the status of processing on the receiver side. Asynchronous SOAP is depicted in figure 2.6.

Many J2EE server vendors now support some form of SOAP messaging, via their support of the JAXM API discussed later in this chapter. More information on the SOAP specification is available at http: / / www.w3c.org / TR / SOAP.

Figure 2.6 Message-style SOAP messaging

Web services

Closely related to the development of SOAP is the concept of *web services.* As we alluded to in chapter 1, web services is the catchall phrase for the standardization of distributed business services architecture over the Internet. Web services rely on SOAP clients and servers to transport inter-enterprise messages.

The subjects of XML messaging and web services are quite complex. We take a detailed look at these topics in chapter 4, including examples. In this section, we discuss only the basics of web services and related technologies.

Work is also ongoing to define a standard way to register and locate new web services using distributed service repositories, or search engines. These repositories use XML to describe web services and the companies that provide them. The most promising of these standards to date is the Universal Description, Discovery, and Integration (UDDI) specification. This is due to the broad vendor support UDDI currently enjoys from many companies, including IBM and Microsoft.

UDDI

A consortium of large companies has come together to create a set of standards around the registration and discovery process for web services. The result is UDDI. The goal of UDDI is to enable the online registration and lookup of web services via a publicly available repository, similar in operation to the Domain Name System (DNS) of the Internet. The service registry is referred to as the *green pages* and is defined in an XML Schema. The green pages are syndicated across multiple *operator sites*. Each site provides some level of public information regarding the services. This information is represented as metadata and known as a *tModel*.

One of the challenges when registering a web service is deciding on how it should be classified. A mere alphabetical listing by provider would make it impossible to find a particular type of service. UDDI therefore allows classification of

services by geographic region and standard industry codes, such as NAICS and UN/SPC. Many expect the other services repositories, such as the ebXML Repository, to merge with UDDI in the future, although no one can say for sure.

You can read more about UDDI and related technologies at http://www.uddi.org.

WSDL

The creators of the UDDI directory recognized the need for a standard means for describing web services in the registry. To address this, they created the Web Services Description Language (WSDL). WSDL is an XML language used to generically describe web services. The information contained in each description includes a network address, protocol, and a supported set of operations. We will discuss WSDL in detail and provide examples of it in chapter 4.

2.1.5 Data manipulation and retrieval technologies

Storing and retrieving data in XML format is the subject of much ongoing work with XML. The need for XML storage and retrieval technologies has resulted in the creation of a large number of closely related specifications. In this section, we provide you with a brief overview of these specifications and point you in the direction of more information about each.

XPath

XPath is a language for addressing XML structures that is used by a variety of other XML standards, including XSLT, XPointer, and XQuery. It defines the syntax for creating *expressions*, which are evaluated against an XML document. For example, a forward slash (/) is a simple XPath expression. As you saw in listing 2.4, this expression represents the root node of an XML document.

XPath expressions can represent a node-set, Boolean, number, or string. They can start from the root element or be relative to a specific position in a document. The most common type of XPath expression is a location path, which represents a node-set. For our product catalog document example, the following XPath represents all the product nodes in the catalog:

```
/product
```

XPath has a built-in set of functions that enable you to develop very complex expressions. Although XPath syntax is not a focus of this book, we do explore technologies such as XSLT that use it extensively. Since XPath is so important, we suggest that you become proficient with it as quickly as possible.

You can get more detailed information on XPath at http://www.w3c.org/TR/xpath.

XPointer

XPointer is an even more specific language that builds on XPath. XPointer expressions point to not only a node-set, but to the specific position or range of positions within a node-set that satisfy a particular condition. XPointer functions provide a very robust method for searching through XML data structures. Take, for example, the following node-set:

```
<desc>This chapter provides an overview of the J2EE technologies.</desc>
<desc>This chapter provides an overview of the XML landscape.</desc>
<desc>This chapter is an introduction to distributed systems.</desc>
```

A simple XPointer expression that operates on this node-set is as follows:

```
xpointer(string-range(//desc, 'overview'))
```

This expression returns all nodes with the name `desc` that contain the string `overview`. XPointer expressions can be formed in several ways and can quickly become complex. You can find more information on XPointer at http://www.w3c.org/XML/Linking.

XInclude

XInclude is a mechanism for including XML documents inside other XML documents. This allows us to set up complex relationships among multiple XML documents. It is accomplished by using the `<include>` tag, specifying a location for the document, and indicating whether or not it should be parsed. The `include` tag may be placed anywhere within an XML document. The location may reference a full XML document or may use XPointer notation to reference specific portions of it. The use of XPointer with XInclude makes it easier to include specific XML data and prevents us from having to duplicate data in multiple files.

Adding the following line to an XML document would include a node-set from an external file called `afile.xml` in the current XML document, at the current location:

```
<xi:include href="afile.xml#xpointer( XPath expression )" parse="xml" />
```

Only the nodes matching the specified XPath expression would be included.

More information on XInclude can be found at http://www.w3c.org/TR/xinclude.

XLink

XLink is a technology that facilitates linking resources within separate XML documents. It was created because requirements for linking XML resources require a more robust mechanism than HTML-style hyperlinks can provide. HTML hyperlinks are unidirectional, whereas XLink enables traversal in both directions. XLinks can be either simple or extended. Simple XLinks conform to similar rules as HTML hyperlinks, while extended XLinks feature additional functionality.

The flexibility of XLink enables the creation of extremely complex and robust relationships. The following example uses a simple XLink that establishes a relationship between an order and the customer who placed it.

If this XML document represents a customer:

```
<customer id="0059">
  <name>ABC Company</name>
  <employees>1000-1500</employees>
</customer>
```

This XML document lists orders linked to that customer:

```
<orders>
 <order xlink:type="simple"
  href="customers.xml#//customers/customer/@id[.='0059']"
    title="Customer" show="new">
  <number>12345</number>
  <amount>$500</amount>
 </order>
</orders>
```

Note once again the importance of XPath expressions in enabling this technology. More information on XLink is at http://www.w3c.org/XML/Linking.

XBase

XBase, or XML Base, is a mechanism for specifying a base uniform resource identifier (URI) for XML documents, such that all subsequent references are inferred to be relative to that URI. Despite its simplicity, XBase is extremely handy and allows you to keep individual XLinks to a reasonable length.

The following line describes a base URI using XBase. Any relative URI reference encountered inside the `catalog` element will be resolved using `http://www.manning.com/books` as its base.

```
<catalog xml:base=http://www.manning.com/books/>
         ……
         ……
</catalog>
```

You can learn more about XBase at http://www.w3c.org/TR/xmlbase.

Query languages

As the amount of data being stored in XML has increased, it is not surprising that several query languages have been developed specifically for XML. One of the initial efforts in this area was XQL, the XML Query Language. XQL is a language for querying XML data structures and shares many of its constructs with XPath. Using XQL, queries return a set of nodes from one or more documents. Other query languages include Quilt and XML-QL.

The W3C has recently taken on the daunting task of unifying these specifications under one, standardized query language. The result of this effort is a language is called XQuery. It uses and builds upon XPath syntax. The result of an XML query is either a node-set or a set of primitive values. XQuery is syntactically similar to SQL, with a set of keywords including FOR, LET, WHERE, and RETURN.

The following is a simple XQuery expression that selects all product nodes from `afile.xml`.

```
document( "afile.xml" )//product
```

A slightly more complex XQuery expression selects the `warranty` node for each `product`.

```
FOR $product in  //product
       RETURN $product/warranty
```

XQuery is in its early stages of completion and there are not many products around that fully implement the specification. The latest version of Software AG's Tamino server has some support for XQuery, but a full XQuery engine has yet to be implemented. We discuss XQuery in more detail in chapter 3, within our discussion of XML data persistence. You can get all the details about XQuery at http://www.w3c.org/XML/Query.

2.1.6 Data storage technologies

XML is data, so it should be no surprise that there are a variety of technologies under development for storing native XML data. The range of technologies and products is actually quite large, and it is still unclear which products will emerge as the leaders.

Storing XML on the file system is still very popular, but storing XML in a textual, unparsed format is inefficient and greatly limits its usability. Static documents require reparsing each time they are accessed. An alternative mechanism to storing text files is the Persistent Document Object Model (PDOM). PDOM implements the W3C DOM specification but stores the parsed XML

document in binary format on the file system. In this fashion, it does not need to be reparsed for subsequent access.

PDOM documents may be generated from an existing DOM or through an XML input stream, so the document is not required to be in memory in its entirety at any given time. This is advantageous when dealing with large XML documents. PDOM supports all of the standard operations that you would expect from a data storage component, such as querying (via XQL), inserting, deleting, compressing, and caching. We offer an example of using this technique for data storage in chapter 3. You can learn more about PDOM at http://xml.darmstadt.gmd.de/xql/.

Another alternative to static file system storage is the use of native-XML databases. Databases such as Software AG's Tamino are designed specifically for XML. Unlike relational databases, which store hierarchical XML documents in relational tables, Tamino stores XML in its native format. This gives Tamino a significant performance boost when dealing with XML.

Despite the appearance of native-XML database vendors, traditional database vendors such as Oracle and IBM had no intention of yielding any of the data storage market just because traditional relational databases did not handle XML well initially. The major relational vendors have built extensions for their existing products to accommodate XML as a data type and enable querying functionality. This is advantageous for many companies that rely heavily on RDBMS products and have built up strong skill-sets in those technologies.

Figure 2.7 summarizes your options for XML data storage.

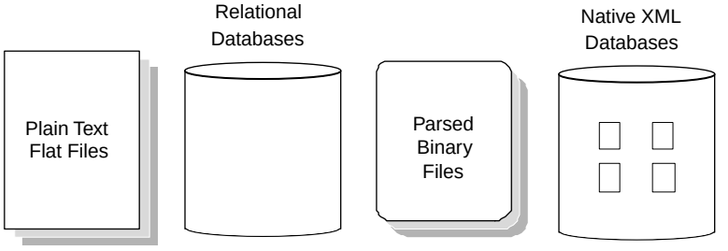

Figure 2.7 XML data storage alternatives

2.2 The Java APIs for XML

The Java development community is actively following and contributing to the specification of many of the XML technologies discussed in section 2.1.

Additionally, Java is often the first language to implement these emerging technologies. This is due largely to the complimentary nature of platform independent code (Java) and data (XML). However, XML API development in Java has historically been disjointed, parallel, and overly complicated. Various groups have implemented XML functionality in Java in different ways and at different times, which led to the proliferation of overlapping, noncompatible APIs.

To address this issue and make developing XML-aware applications in Java simpler, Sun Microsystems is now coordinating Java XML API development via the Java Community Process (JCP). Under this process, the Java development community is standardizing and simplifying the various Java APIs for XML. Most of these efforts have been successful, although a couple of the standard specifications still have overlapping scope or functionality. Nevertheless, XML processing in Java has come a long way in 2000 and 2001.

The Java APIs for XML (JAX) is currently a family of related API specifications. The members of the JAX family are summarized in table 2.1. In this section, we introduce each member of JAX and discuss its current state of maturity. For those JAX members with an existing reference implementation, we also provide usage examples for each.

Table 2.1 The JAX family—Java APIs for XML processing

Java API for XML	JAX acronym	Functional description
Java API for XML parsing	JAXP	Provides implementation-neutral access to XML parsers and XSLT processors.
Java Document Object Model	JDOM	Provides a Java-centric, object-oriented implementation of the DOM framework.
Java API for XML binding	JAXB	Provides a persistent XML mapping for Java object storage as XML.
Long Term JavaBeans Persistence		Similar to JAXB, provides XML serialization for JavaBean components.
Java API for XML messaging	JAXM	Enables the use of SOAP messaging in Java applications, using resource factories in a manner similar to the Java Messaging Service (JMS).
JAX-RPC	JAX-RPC	An XML-RPC implementation API for Java. Similar to JAXM.
Java API for XML repositories	JAXR	Provides implementation-neutral access to XML repositories like ebXML and UDDI.

2.2.1 JAXP

JAXP provides a common interface for creating and using the SAX, DOM, and XSLT APIs in Java. It is implementation- and vendor-neutral. Your applications should use JAXP instead of accessing the underlying APIs directly to enable the replacement of one vendor's implementation with another as desired. As faster or better implementations of the base XML APIs become available, you can upgrade to them simply by exchanging one JAR file for another. This achieves a primary goal in distributed application development: flexibility.

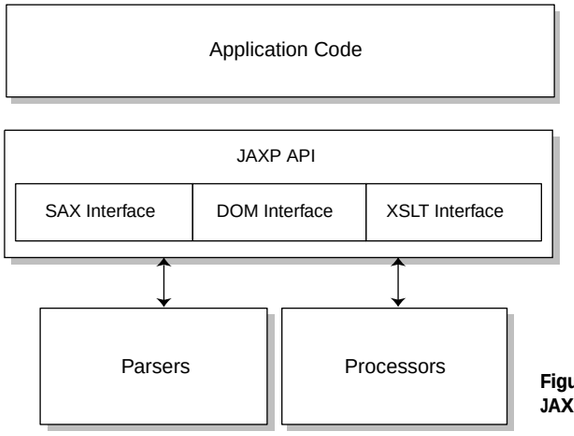

Figure 2.8
JAXP architecture

The JAXP API architecture is depicted in figure 2.8. JAXP enables flexibility by divorcing your application code from the underlying XML APIs. You can use it to parse XML documents using SAX or DOM as the underlying strategy. You can also use it to transform XML via XSLT in a vendor-neutral way.

Table 2.2 The JAXP packages

Package	Description
javax.xml.parsers	Provides a common interface to DOM and SAX parsers.
javax.xml.transform	Provides a common interface to XSLT processors.
org.xml.sax	The generic SAX API for Java
org.w3c.dom	The generic DOM API for Java

The JAXP API consists of four packages, summarized in table 2.2. Of these, the two `javax.xml` packages are of primary interest. The `javax.xml.parsers` package contains the classes and interfaces needed to parse XML documents. The `javax.xml.transform` package defines the interface for XSLT processing.

Configuring JAXP

To use JAXP for parsing, you require a JAXP-compliant XML parser. The JAXP reference implementation uses the Crimson parser mentioned earlier. To do XSLT processing, you also need a compliant XSLT engine. The reference implementation uses Xalan, also mentioned earlier.

When you first access the JAXP parsing classes in your code, the framework initializes itself by taking the following steps:

- It initially checks to see if the system property `javax.xml.parsers.DocumentBuilderFactory` or `javax.xml.parsers.SAXParserFactory` has been set (depending on whether you are requesting the use of SAX or DOM). If you are requesting an XSLT transformation, the system property `javax.xml.transform.TransformerFactory` is checked instead.
- If the appropriate system property has not been set explicitly, the framework searches for a file called `jaxp.properties` in the lib directory of your JRE. Listing 2.5 shows how the contents of this file might appear.
- If the jaxp.properties file is not found, the framework looks for files on the classpath named `/META-INF/services/java.xml.parsers.DocumentBuilderFactory`, `/META-INF/services/SAXParserFactory`, and `/META-INF/services/javax.xml.transform.TransformerFactory`. When found, these files contain the names of the JAXP `DocumentBuilder`, `SAXParserFactory`, and `TransformerFactory` classes, respectively. JAXP-compliant parsers and XSLT processors contain these text files in their jars.
- If a suitable implementation class name cannot be found using the above steps, the platform default is used. Crimson will be invoked for parsing and Xalan for XSLT.

NOTE Statements in the following listing are shown on multiple lines for clarity. In an actual `jaxp.properties` file, each statement should appear as a single line with no spaces between the equals character (=) and the implementation class name.

Listing 2.5 A Sample jaxp.properties file

```
javax.xml.parsers.DocumentBuilderFactory=
  org.apache.crimson.jaxp.DocumentBuilderFactoryImpl
javax.xml.parsers.SAXParserFactory=
  org.apache.crimson.jaxp.SAXParserFactoryImpl
javax.xml..transform.TransformerFactory=
  org.apache.xalan.processor.TransformerFactoryImpl
```

Sets DOM builder, SAX parser, and XSLT processor implementation classes

Since JAXP-compliant parsers and processors already contain the necessary text files to map their implementation classes to the JAXP framework, the easiest way to configure JAXP is to simply place the desired parser and/or processor implementation's JAR file on your classpath, along with the JAXP jar. If, however, you find yourself with two JAXP-compliant APIs on your classpath for some other reason, you should explicitly set the implementation class(es) before using JAXP. Since you would not want to do this in your application code, the properties file approach is probably best.

JAXP is now a part of the J2EE specification, meaning that your J2EE vendor is required to support it. This makes using JAXP an even easier choice over directly using a specific DOM, SAX, or XSLT implementation.

Using JAXP with SAX

The key JAXP classes for use with SAX are listed in table 2.3. Before demonstrating the use of SAX via JAXP, we must digress for a moment on the low level details of SAX parsing. To use SAX with or without JAXP, you must always define one or more event handlers for the parser to use.

DEFINITION A *SAX event handler* is a component that registers itself for callbacks from the parser when SAX events are fired.

The SAX API defines four core event handlers, encapsulated within the `EntityResolver`, `DTDHandler`, `ContentHandler`, and `ErrorHandler` interfaces of the `org.xml.sax` package. The `ContentHandler` is the primary interface that most applications need to implement. It contains callback methods for the `startDocument`, `startElement`, `endElement`, and `endDocument` events. Your application must implement the necessary SAX event interface(s) to define your specific implementation of the event handlers with which you are interested.

Table 2.3 Primary JAXP interfaces to the SAX API

JAXP class or interface	Description
javax.xml.parsers.SAXParserFactory	Locates a SAXParserFactory implementation class and instantiates it. The implementation class in turn provides SAXParser implementations for use by your application code.
javax.xml.parsers.SAXParser	Interface to the underlying SAX parser.
javax.xml.parsers.SAXReader	A class wrapped by the SAXParser that interacts with your SAX event handler(s). It can be obtained from the SAXParser and configured before parsing when necessary.
org.xml.sax.helpers.DefaultHander	A utility class that implements all the SAX event handler interfaces. You can subclass this class to get easy access to all possible SAX events and then override the specific methods in which you have interest.

The other types of event handlers defined in SAX exist to deal with more peripheral tasks in XML parsing. The `EntityResolver` interface enables the mapping of references to external sources such as databases or URLs. The `ErrorHandler` interface is implemented to handle special processing of `SAXExceptions`. Finally, the `DTDHandler` interface is used to capture information about document validation as specified in the document's DTD.

SAX also provides a convenience class called the `org.xml.sax.helpers.DefaultHandler`, which implements all of the event handler interfaces. By extending the `DefaultHandler` class, your component has access to all of the available SAX events.

Now that we understand how SAX works, it is time to put JAXP to work with it. For an example, let us read in our earlier product catalog XML document using SAX events and JAXP. To keep our example short and relevant, we define a SAX event handler class that listens only for the `endElement` event. Each time a product element has been completely read by the SAX parser, we print a message indicating such. The code for this handler is shown in listing 2.6.

Listing 2.6 SAX event handler for product nodes

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

public class ProductEventHandler
  extends DefaultHandler {
```

Extends this class to only handle the `endElement` event

```
  // other event handlers could go here

public void endElement( String namespaceURI,
                        String localName,
                        String qName,
                        Attributes atts )
    throws SAXException {
      // make sure it was a product node
      if (localName.equals("product"))
       System.out.println(
          A product was read from the catalog.);
 }
}
```

Now that we have defined an event handler, we can obtain a SAX parser implementation via JAXP in our application code and pass the handler to it. The handler's `endElement` method will be called once when parsing the example document, since there is only one product node. The code for our JAXP SAX example is given in listing 2.7.

Listing 2.7 Parsing XML with JAXP and SAX

```
import javax.xml.parsers.SAXParserFactory;
import javax.xml.parsers.SAXParser;
import java.io.File;

public class JAXPandSAX  {

  public static void main(String[] args) {

    ProductEventHandler handler
      = new ProductEventHandler();

    try {
      SAXParserFactory factory
        = SAXParserFactory.newInstance();
      SAXParser parser
        = factory.newSAXParser();
      File ourExample
        = new File("product-catalog.xml");

      parser.parse( ourExample, handler);

    } catch (Exception e) {
      System.out.println(e.getMessage());
    }
  }

}
```

Instantiates our event handler

Obtains a SAXParser via JAXP

When the code in listing 2.6 is executed against our product catalog document from listing 2.1, you should see the following output:

```
Product read from the catalog.
```

This statement only prints once, since we have only defined a single product. If there were multiple products defined, this statement would have printed once per product.

Using JAXP with DOM

Using JAXP with DOM is a far less complicated endeavor than with SAX. This is because you do not need to develop an event handler and pass it to the parser. Using DOM, the entire XML document is read into memory and represented as a tree. This allows you to manipulate the entire document at once, and does not require any state-machine logic programming on your part. This convenience comes, of course, at the expense of system resources and speed. The central JAXP classes for working with DOM are summarized in table 2.4.

Table 2.4 Primary JAXP interfaces to the DOM API

JAXP class or interface	Description
javax.xml.parsers.DocumentBuilderFactory	Locates a DocumentBuilderFactory implementation class and instantiates it. The implementation class in turn provides DocumentBuilder implementations.
javax.xml.parsers.DocumentBuilder	Interface to the underlying DOM builder.

Since our product catalog document is very short, there is no danger in reading it in via DOM. The code to do so is given in listing 2.8. You can see that the general steps of obtaining a parser from JAXP and invoking it on a document are the same. The primary difference is the absence of the SAX event handler. Note also that the parser returns a pointer to the DOM in memory after parsing. Using the other DOM API classes in the `org.w3c.dom` package, you could traverse the DOM in your code and visit each product in the catalog. We leave that as an exercise for the reader.

Listing 2.8 Building a DOM with JAXP

```
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;
import org.w3c.dom.Document;
import java.io.File;

public class JAXPandDOM  {
```

Imports the JAXP DOM classes

```
  public static void main(String[] args) {

    try {
   DocumentBuilderFactory factory
        = DocumentBuilderFactory.newInstance();
      DocumentBuilder builder                    | Obtains a DOMBuilder
        = factory.newDocumentBuilder();          | via JAXP
      File ourExample
        = new File("product-catalog.xml");
      Document document                          | Parses the XML and
        = builder.parse( ourExample );           | builds a DOM tree
  } catch (Exception e) {
    System.out.println(e.getMessage());
   }
  }
}
```

Using JAXP with XSLT

JAXP supports XSLT in the same implementation-independent manner as XML parsing. The JAXP interfaces to XSLT are located in the `javax.xml.transform` package. The primary classes and interfaces are summarized in table 2.5. In addition to these top-level interfaces, JAXP includes three subpackages to support the use of SAX, DOM, and I/O streams with XSLT. These packages are summarized in table 2.6.

Table 2.5 Primary JAXP interfaces to the XSLT API

JAXP class or interface	Description
javax.xml.transform.TransformerFactory	Locates a TransformerFactory implementation class and instantiates it.
javax.xml.transform.Transformer	Interface to the underlying XSLT processor.
javax.xml.transform.Source	An interface representing an XML data source to be transformed by the Transformer.
javax.xml.transform.Result	An interface to the output of the Transformer after XSLT processing.

In section 2.1.3, we discussed the XSLT process and saw how our product catalog document could be transformed into HTML via XSLT. Now we examine how that XSLT process can be invoked from your Java code via JAXP. For the sake of clarity and simplicity, we will use the I/O stream helper classes

from the `javax.xml.transform.stream` package to create our `Source` and `Result` objects.

Table 2.6 JAXP helper packages for XSLT

Package name	Description
javax.xml.transform.dom	Contains classes and interfaces for using XSLT with DOM input sources and results.
javax.xml.transform.sax	Contains classes and interfaces for using XSLT with SAX input sources and results.
javax.xml.transform.stream	Contains classes and interfaces for using XSLT with I/O input and output stream sources and results.

The code we need to convert our example document to HTML is shown in listing 2.9. To compile it, you must have the JAXP jar file in your classpath. To run this program, you must have the example product catalog XML document from listing 2.1 saved in a file called `product-catalog.xml`. The stylesheet from listing 2.4 must be saved to a file named `product-catalog-to-html.xsl`. You can either type these files into your favorite editor or download them from the book's web site at http://www.manning.com/Gabrick. You will also need to place a JAXP-compliant XSLT engine (such as Xalan) in your classpath before testing this example.

Listing 2.9 Building a DOM with JAXP

```
import javax.xml.transform.*;
import javax.xml.transform.stream.*;
import java.io.File;

public class JAXPandXSLT {

  public static void main(String[] args) {

   File sourceFile
      = new File("product-catalog.xml");
    File xsltFile
      = new File("product-catalog-to-html.xsl");

    Source xmlSource = new StreamSource(sourceFile);
    Source xsltSource = new StreamSource(xsltFile);
    Result result = new StreamResult(System.out);

    TransformerFactory factory
      = TransformerFactory.newInstance();

try {
```

Imports the JAXP XSLT API

Loads the XML and XSL files

Creates I/O Stream sources and results

Returns an instance of `TransformerFactory`

```
    Transformer transformer
      = factory.newTransformer(xsltSource);

    transformer.transform(xmlSource, result);

    } catch (TransformerConfigurationException tce) {
      System.out.println("No JAXP-compliant XSLT processor found.");
    } catch (TransformerException te) {
      System.out.println("Error while transforming document:");
      te.printStackTrace();
    }
  }
}
```

❶ **Factory returns new `Transformer`**

❷ **Performs transformation**

❶ The `TransformerFactory` implementation then provides its own specific `Transformer` implementation. Note that the transformation rules contained in the XSLT stylesheet are passed to the factory for it to create a `Transformer` object.

❷ This is the call that actually performs the XSLT transformation. Results are streamed to the specified `Result` stream, which is the console in this example.

At first glance, using XSLT via JAXP does not appear to be too complex. This is true for simple transformations, but there are many attributes of the XSLT process that can be configured via the `Transformer` and `TransformerFactory` interfaces. You can also create and register a custom error handler to deal with unexpected events during transformation. See the JAXP documentation for a complete listing of the possibilities. In this book, we concentrate on where and how you would use JAXP in your J2EE code rather than exhaustively exercising this API.

A word of caution

Using XSLT, even via JAXP, is not without its challenges. The biggest barrier to the widespread use of XSLT is currently performance. Performing an XSLT transformation on an XML document is time- and resource-intensive. Some XSLT processors (including Xalan) allow you to precompile the transformation rules contained in your stylesheets to speed throughput. Through the JAXP 1.1 interface, it is not yet possible to access this feature.

Proceed with caution and perform thorough load tests before using XSLT in production. If you need to use XSLT and if performance via JAXP is insufficient, you may consider using a vendor API directly and wrapping it in a utility component using the Faade pattern. You might also look into XSLTC, an XSLT compiler recently donated to the Apache Software Foundation by Sun

Microsystems. It enables you to compile XSLT stylesheets into Java classes called *translets*. More information on XSLTC is available at http://xml.apache.org/xalan-j/xsltc/.

2.2.2 JDOM

The first thing that stands out about this JAX family member is its lack of a JAX acronym. With JAXP now at your disposal, you can write parser-independent XML application code. However, there is another API that can simplify things even further. It is called the Java Document Object Model (JDOM), and has been recently accepted as a formal recommendation under the Java Community Process.

JDOM, created by Jason Hunter and Brett McLaughlin, provides a Java-centric API for working with XML data structures. It was designed specifically for Java and provides an easy-to-use object model already familiar to Java developers. For example, JDOM uses Java collection classes such as `java.util.List` to work with XML data like node-sets. Furthermore, JDOM classes are concrete implementations whereas the DOM classes are abstract. This makes them easy to use and removes your dependence on a specific vendor's DOM implementation, much like JAXP.

The most recent version of JDOM has been retrofitted to use the JAXP API. This means that your use of JDOM does not subvert the JAXP architecture, but builds upon it. When the JDOM builder classes create an XML object, they invoke the JAXP API if available. Otherwise, they rely on a default provider for parsing (Xerces) and a default XSLT processor (Xalan). The JDOM architecture is depicted in figure 2.9.

Table 2.7 lists the central JDOM classes. As you can see, they are named quite intuitively. JDOM documents can be created in memory or built from a stream, a file, or a URL.

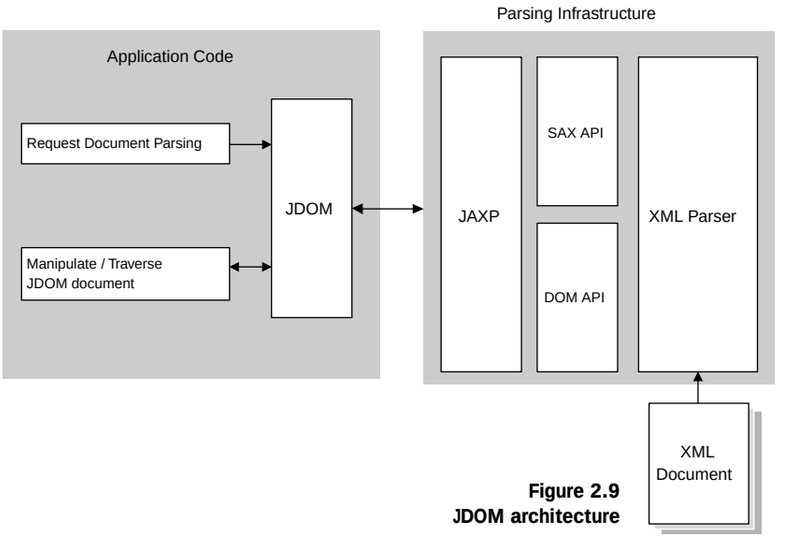

Figure 2.9
JDOM architecture

Table 2.7 Core JDOM classes

Class name	Description
org.jdom.Document	The primary interface to a JDOM document.
org.jdom.Element	An object representation of an XML node.
org.jdom.Attribute	An object representation of an XML node's attribute.
org.jdom.ProcessingInstruction	JDOM contains objects to represent special XML content, including application-specific processing instructions.
org.jdom.input.SAXBuilder	A JDOM builder that uses SAX.
org.jdom.input.DOMBuilder	A JDOM builder that uses DOM.
org.jdom.transform.Source	A JAXP XSLT Source for JDOM Documents. The JDOM is passed to the Transformer as a JAXP SAXSource.
org.jdom.transform.Result	A JAXP XSLT Result for JDOM Documents. Builds a JDOM from a JAXP SAXResult.

To quickly demonstrate how easy JDOM is to use, let us build our product catalog document from scratch, in memory, and then write it to a file. To do so, we simply build a tree of JDOM `Elements` and create a JDOM `Document` from it. The code to make this happen is shown in listing 2.10. When you compile and run this code, you should find a well-formatted version of the XML document shown in listing 2.1 in your current directory.

Listing 2.10 Building a document with JDOM

```
import org.jdom.*;
import org.jdom.output.XMLOutputter;
import java.io.FileOutputStream;

public class JDOMCatalogBuilder {

  public static void main(String[] args) {

    // construct the JDOM elements

    Element rootElement = new Element("product-catalog");
    Element productElement = new Element("product");

    productElement.addAttribute("sku", "123456");
    productElement.addAttribute("name", "The Product");

    Element en_US_descr = new Element("description");
    en_US_descr.addAttribute("locale", "en_US");
    en_US_descr.addContent("An excellent product.");

    Element es_MX_descr = new Element("description");
    es_MX_descr.addAttribute("locale", "es_MX");
    es_MX_descr.addContent("Un producto excellente.");

    Element en_US_price = new Element("price");
    en_US_price.addAttribute("locale", "en_US");
    en_US_price.addAttribute("unit", "USD");
    en_US_price.addContent("99.95");

    Element es_MX_price = new Element("price");
    es_MX_price.addAttribute("locale", "es_MX");
    es_MX_price.addAttribute("unit", "MXP");
    es_MX_price.addContent("9999.95");

    // arrange elements into a DOM tree

    productElement.addContent(en_US_descr);
    productElement.addContent(es_MX_descr);
    productElement.addContent(en_US_price);
    productElement.addContent(es_MX_price);

    rootElement.addContent(productElement);
    Document document = new Document(rootElement);

    // output the DOM to "product-catalog.xml" file
```

Creates element attributes

Adds text to the element

Builds the DOM by adding one element as content to another

Wraps root element and processing instructions

```
    XMLOutputter out = new XMLOutputter("  ", true);
    try {
      FileOutputStream fos = new FileOutputStream("product-catalog.xml");
      out.output(document, fos);
    } catch (Exception e) {
      System.out.println("Exception while outputting JDOM:");
      e.printStackTrace();
    }
  }
}
```

Indents element two spaces and uses newlines

Writes the JDOM representation to a file

Due to its intuitive interface and support for JAXP, you will see JDOM used extensively in remaining chapters. You can find detailed information about JDOM and download the latest version from http://www.jdom.org.

2.2.3 JAXB

The Java API for XML Binding (JAXB) is an effort to define a two-way mapping between Java data objects and XML structures. The goal is to make the persistence of Java objects as XML easy for Java developers. Without JAXB, the process of storing and retrieving (*serializing* and *deserializing*, respectively) Java objects with XML requires the creation and maintenance of cumbersome code to read, parse, and output XML documents. JAXB enables you to work with XML documents as if they were Java objects.

DEFINITION *Serialization* is the process of writing out the state of a running software object to an output stream. These streams typically represent files or TCP data sockets.

The JAXB development process requires the creation of a DTD and a *binding schema*—an XML document that defines the mapping between a Java object and its XML schema. You feed the DTD and binding schema into a *schema compiler* to generate Java source code. The resulting classes, once compiled, handle the details of the XML-Java conversion process. This means that you do not need to explicitly perform SAX or DOM parsing in your application code. Figure 2.10 depicts the JAXB process flow.

Early releases of JAXB show improved performance over SAX and DOM parsers because its classes are lightweight and precompiled. This is a positive sign for the future of JAXB, given the common concerns about performance when using XML.

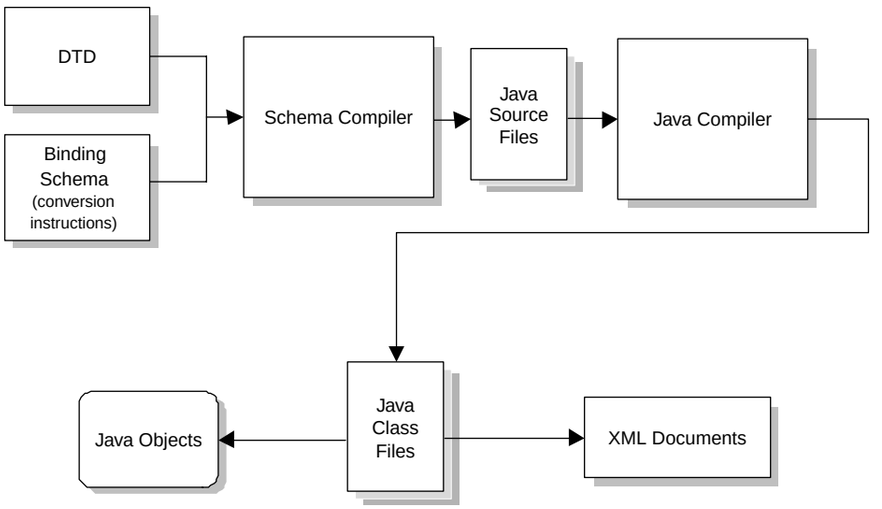

Figure 2.10 JAXB architecture

One tradeoff to consider before using JAXB is a loss of system flexibility, since any change in your XML or object structures requires recompilation of the JAXB classes. This can be inconvenient or impractical for rapidly evolving systems that use JAXB extensively. Each change to the JAXB infrastructure requires regenerating the JAXB bindings and retesting the affected portions of the system.

JAXB manifests other issues in its current implementation that you should explore before using it in your applications. For example, the process by which XML data structures are created from relational tables is overly simplistic and resource intensive. Issues such as these are expected to subside as the specification matures over time. We provide an example of using JAXB in the remainder of this section. More information about the capabilities and limitations of this API are available at http://java.sun.com/xml/jaxb/.

Binding Java objects to XML

To see JAXB in action, we turn once again to our product catalog example from listing 2.1. We previously developed the DTD corresponding to this document, which is shown in listing 2.2. Creating the binding schema is a bit more complicated. We start by creating a new binding schema file called `product-catalog.xjs`. Binding schemas in the early access version of JAXB always have the following root element:

```
<xml-java-binding-schema version="1.0-ea">
```

This element identifies the document as a binding schema. We now define our basic, innermost elements in the product-catalog document:

```
<element name="description" type="class">
  <attribute name="locale"/>
  <content property="description"/>
</element>
```

and

```
<element name="price" type="class">
   <attribute name="locale"/>
   <attribute name="unit"/>
   <content property="price"/>
 </element>
```

The `type` attribute of the `element` node denotes that the elements of `type` description and `price` in the `product-catalog` document are to be treated as individual Java objects. This is necessary because both `description` and `price` have their own attributes as well as content.

The `content` element in each of the above definitions tells the JAXB compiler to create a property for the enclosing class with the specified `name`. The content of the generated Description class will be accessed via the `getDescription` and `setDescription` methods. Likewise, the `Price` class content will be accessed via methods called `getPrice` and `setPrice`.

Having described these basic elements, we can now refer to them in the definition of the `product` element.

```
<element name="product" type="class">
  <content>
    <element-ref name="description"/>
   <element-ref name="price"/>
  </content>
</element>
```

The product element maps to a Java class named Product and will contain two Lists as instance variables. One of these will be a List of `Description` instances. The other will be a `List` of `Price` instances. Notice the use of `element-ref` instead of `element` in the definition of the `description` and `price` nodes. This construct can be used to create complex object structures and to avoid duplication of information in the binding document.

The final element to bind is the root element, `product-catalog`. Its binding is defined as follows:

```
<element name="product-catalog" type="class" root="true">
  <content>
    <element-ref name="product"/>
  </content>
</element>
```

Notice the `root=true` attribute in this binding definition. This attribute identifies `product-catalog` as the root XML element. From this definition, the JAXB compiler will generate a class called ProductCatalog, containing a `List` of `Product` instances. The complete JAXB binding schema for our example is shown in listing 2.11.

Listing 2.11 Complete JAXB binding schema example

```
<xml-java-binding-schema version="1.0-ea">

  <element name="description" type="class">
    <attribute name="locale"/>
    <content property="description"/>
  </element>

  <element name="price" type="class">
    <attribute name="locale"/>
    <attribute name="unit"/>
    <content property="price"/>
  </element>

  <element name="product" type="class">
    <content>
      <element-ref name="description"/>
      <element-ref name="price"/>
    </content>
  </element>

  <element name="product-catalog" type="class" root="true">
    <content>
      <element-ref name="product"/>
    </content>
  </element>

</xml-java-binding-schema>
```

Now that we have a DTD and a binding schema, we are ready to generate our JAXB source code. Make sure you have the JAXB jar files in your classpath and execute the following command:

```
# java com.sun.tools.xjc.Main product-catalog.dtd product-catalog.xjs
```

If all goes well, you will see the following files created in your current directory:

```
Description.java
Price.java
Product.java
ProductCatalog.java
```

You can now compile these classes and begin to use them in your application code.

Using JAXB objects

Using your compiled **JAXB** classes within your application is easy. To read in objects from XML files, you simply point your JAXB objects at the appropriate file and read them in. If you are familiar with the use of `java.io.ObjectInputStream`, the concept is quite similar. Here is some code you can use to read in the product catalog document via JAXB:

```
ProductCatalog catalog = null;
File productCatalogFile = new File("product-catalog.xml");
try {
  FileInputStream fis
    = new FileInputStream(productCatalogFile);
  catalog = ProductCatalog.unmarshal(fis);
} catch (Exception e) {
  // handle
} finally {
 fis.close();
}
```

To reverse the process and save the ProductCatalog instance as XML, you could do the following:

```
try {
  FileOutputStream fos
    = new FileOutputStream(productCatalogFile);
  catalog.marshal(fos);
} catch (Exception e2) {
  // handle
} finally {
  fos.close();
}
```

In the course of application processing, use your JAXB objects just as you would any other object containing instance variables. In many cases, you will need to iterate through the children of a given element instance to find the data you need. For example, to get the U.S. English description for a given Product instance `product`, you would need to do the following:

```
String description = null;
List descriptions = product.getDescription();
ListIterator it = descriptions.listIterator();
```

```
while (it.hasNext()) {
  Description d = (Description) it.next();
  if (d.getLocale().equals(en_US)) {
    description = d.getDescription();
    break;
  }
}
```

This type of iteration is necessary when processing XML data through all APIs, and is not specific to JAXB. It is a necessary part of traversing tree data structures like XML.

We invite you to explore the full capabilities of JAXB at the URL given near the beginning of this section. This can be a very useful API in certain applications, especially those with serious performance demands.

2.2.4 Long Term JavaBeans Persistence

Easily the most poorly named Java XML API, *Long Term JavaBeans Persistence* defines an XML mapping API for JavaBeans components. It is similar in function to JAXB, but leverages the JavaBeans component contract instead of a binding schema to define the mapping from Java to XML. Since JavaBeans must define *get* and *set methods* for each of their publicly accessible properties, it was possible to develop XML-aware components that can serialize JavaBeans to XML without a binding schema. These components use the Java reflection API to inspect a given bean and serialize it to XML in a standard format.

This API has become a part of the Java 2 Standard Edition as of version 1.4. There is no need to download any extra classes and add them to your classpath. The primary interfaces to this API are summarized in table 2.8. These classes behave in a similar fashion to `java.io.ObjectInputStream` and `java.io.ObjectOutputStream`, but use XML instead of a binary format.

Table 2.8 Core Long Term JavaBeans Persistence classes

Class name	Description
java.beans.XMLEncoder	Serializes a JavaBean as XML to an output stream.
java.beans.XMLDecoder	Reads in a JavaBean as XML from an input stream.

Writing a JavaBean to XML

As an example, let us define a simple JavaBean with one property, as follows:

```
public class SimpleJavaBean {
  private String name;
```

```
public SimpleJavaBean(String name) {
    setName(name);
  }

// accessor
  public String getName() { return name; }

// modifier
  public void setName(String name) { this.name = name; }

}
```

As you can see, this bean implements the JavaBeans contract of providing an accessor and modifier for its single property. We can save this bean to an XML file named `simple.xml` using the following code snippet:

```
import java.beans.XMLEncoder;
import java.io.*;

...

XMLEncoder e
  = new XMLEncoder(new BufferedOutputStream(
                   new FileOutputStream("simple.xml")));
e.writeObject(new SimpleJavaBean("Simpleton"));
e.close();
```

The code above creates an `XMLEncoder` on top of a `java.io.BufferedOutputStream` representing the file `simple.xml`. We then pass the `SimpleJavaBean` instance reference to the encoder's `writeObject` method and close the stream. The resulting file contents are as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.0" class="java.beans.XMLDecoder">
  <object class="SimpleJavaBean">
    <void property="name">
      <string>Simpleton</string>
    </void>
  </object>
</java>
```

We will not cover the XML syntax in detail, since you do not need to understand it to use this API. Detailed information about this syntax is available in the specification, should you need it.

Restoring a JavaBean from XML

Reading a previously saved JavaBean back into memory is equally simple. Using our `SimpleJavaBean` example, the bean can be reinstated using the following code:

```
XMLDecoder d
  = new XMLDecoder( new BufferedInputStream(
                      new FileInputStream("simple.xml")));
SimpleJavaBean result = (SimpleJavaBean) d.readObject();
d.close();
```

The `XMLDecoder` knows how to reconstitute any bean saved using the `XMLEncoder` component. This API can be a quick and painless way to export your beans to XML for use by other tools and applications. And remember, you can always transform the bean's XML to another format via XSLT to make it more suitable for import into another environment.

2.2.5 JAXM

The Java API for XML Messaging (JAXM) is an enterprise Java API providing a standard access method and transport mechanism for SOAP messaging in Java. It currently includes support for the SOAP 1.1 and SOAP with Attachments specifications. JAXM supports both synchronous and asynchronous messaging.

The JAXM specification defines the various services that must be provided by a JAXM implementation provider. Using any compliant implementation, the developer is shielded from much of the complexity of the messaging system, but has full access to the services it provides. Figure 2.11 depicts the JAXM architecture.

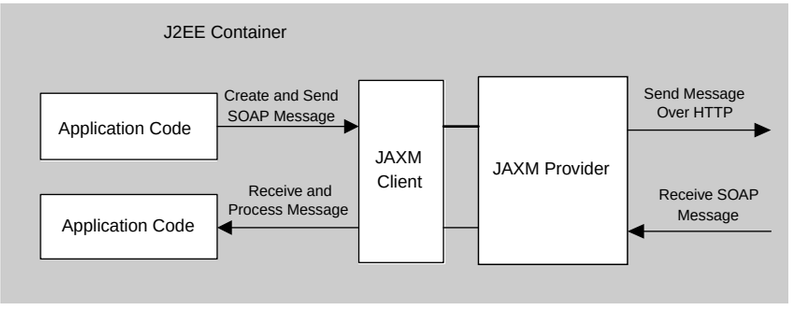

Figure 2.11 JAXM architecture

The two main components of the JAXM architecture are the JAXM Client and Provider. The Client is part of the J2EE Web or EJB container that provides access to JAXM services from within your application. The Provider may be implemented in any number of ways and is responsible for sending and receiving SOAP messages. With the infrastructure in place, sending and receiving SOAP messages can be done exclusively through the JAXM API.

The JAXM API consists of two packages, as summarized in table 2.9. Your components access JAXM services via a `ConnectionFactory` and `Connection` interface, in the same way you would obtain a handle to a message queue in the Java Messaging Service (JMS) architecture. After obtaining a `Connection`, you can use it create a structured SOAP message and send it to a remote host via HTTP(S). JAXM also provides a base Java servlet for you to extend when you need to handle inbound SOAP messages.

Table 2.9 The JAXM API packages

Package name	Description
javax.xml.messaging	Contains the ConnectionFactory and Connection interfaces and supporting objects.
javax.xml.soap	Contains the interface to the SOAP protocol objects, including SOAPEnvelope, SOAPHeader, and SOAPBody

At the time of this writing, JAXM 1.0.1 is available as part of the Java XML Pack and is clearly in the lead of all APIs under development in terms of standardizing the transmission of SOAP messages in Java. Since the creation and consumption of SOAP messages is a complex topic, we defer an example of using JAXM to chapter 4. There we use JAXM to create and access web services in J2EE.

More information about JAXM can be found at http://java.sun.com/xml/jaxm/. Details about the Java XML Pack can be found at http://java.sun.com/xml/javaxmlpack.html.

2.2.6 JAX-RPC

JAX-RPC is a Java-specific means of performing remote procedure calls using XML. JAX-RPC implements the more general XML-RPC mechanism that is the basis of SOAP. Using JAX-RPC, you can expose methods of the beans running in your EJB container to remote Java and non-Java clients. An early access release of the JAX-RPC is now available as part of the Java XML Pack. Up-to-date details about JAX-RPC are at http://java.sun.com/xml/jaxrpc/.

It should be noted that SOAP is fast becoming the preferred method of implementing XML-RPC for web services. Since JAXM already implements the SOAP protocol and has a more mature reference implementation available, the future of the JAX-RPC API remains somewhat uncertain.

2.2.7 JAXR

A critical component to the success of web services is the ability to publish and access information about available services in publicly available registries. Currently, there are several competing standards in the area of web services registries. UDDI and ebXML Registry are currently the two most popular of these standards.

To abstract the differences between registries of different types, an effort is underway to define a single Java API for accessing any type of registry. The planned result is an API called the Java API for XML Registries (JAXR). JAXR will provide a layer of abstraction from the specifics of each registry system, enabling standardized access to web services information from Java.

JAXR is expected to handle everything from executing complex registry queries to submitting and updating your own data to a particular registry system. The primary benefit is that you will have access to heterogeneous registry content without having to code your components to any specific format. Just as JNDI enables dynamic discovery of resources, JAXR will enable dynamic discovery of XML-based registry information. More information on JAXR is available at http://java.sun.com/xml/jaxr/.

The JAXR specification is currently in public review draft, and an early access reference implementation is part of the Java XML Pack. Because of its perceived future importance with regard to web services and the number of parties interested in ensuring its interface is rock solid, this specification is likely to change dramatically before its first official release. We encourage you to stay on top of developments in this API, especially if you plan to produce or consume web services in J2EE.

2.3 Summary

The chapter has been a whirlwind tour of current XML tools and technologies, along with their related Java APIs. Now that you are familiar with the state and direction of both XML and J2EE, we can begin to use them together to enhance your application architecture.

By now, you should be comfortable with viewing XML as a generic metalanguage and understand the relationships between XML, XML parsers, XSLT processors, and XML-based technologies. You should also understand how XML is validated and constrained at high level. Perhaps most importantly, you should see how the various pieces of XML technology fit together to enable a wide

variety of functionality. You will see many of the technologies and APIs discussed in this chapter implemented by the examples in the remaining chapters.

Of all the topics covered in this chapter, web services is by far the hottest topic in business application development today. Chapter 4 contains the details you need to implement and consume web services in J2EE. Chapter 6 provides an end-to-end example of using web services via a case study.