4

Java API for XML Processing

THE Java API for XML Processing (JAXP) is for processing XML data using applications written in the Java programming language. JAXP leverages the parser standards SAX (Simple API for XML Parsing) and DOM (Document Object Model) so that you can choose to parse your data as a stream of events or to build an object representation of it. JAXP also supports the XSLT (XML Stylesheet Language Transformations) standard, giving you control over the presentation of the data and enabling you to convert the data to other XML documents or to other formats, such as HTML. JAXP also provides namespace support, allowing you to work with DTDs that might otherwise have naming conflicts.

Designed to be flexible, JAXP allows you to use any XML-compliant parser from within your application. It does this with what is called a “pluggability layer”, which allows you to plug in an implementation of the SAX or DOM APIs. The pluggability layer also allows you to plug in an XSL processor, letting you control how your XML data is displayed.

The JAXP APIs

The main JAXP APIs are defined in the `javax.xml.parsers` package. That package contains two vendor-neutral factory classes: `SAXParserFactory` and

`DocumentBuilderFactory` that give you a `SAXParser` and a `DocumentBuilder`, respectively. The `DocumentBuilder`, in turn, creates DOM-compliant `Document` object.

The factory APIs give you the ability to plug in an XML implementation offered by another vendor without changing your source code. The implementation you get depends on the setting of the `javax.xml.parsers.SAXParserFactory` and `javax.xml.parsers.DocumentBuilderFactory` system properties. The default values (unless overridden at runtime) point to Sun's implementation.

The remainder of this section shows how the different JAXP APIs work when you write an application.

An Overview of the Packages

The SAX and DOM APIs are defined by XML-DEV group and by the W3C, respectively. The libraries that define those APIs are:

`javax.xml.parsers`
: The JAXP APIs, which provide a common interface for different vendors' SAX and DOM parsers.

`org.w3c.dom`
: Defines the `Document` class (a DOM), as well as classes for all of the components of a DOM.

`org.xml.sax`
: Defines the basic SAX APIs.

`javax.xml.transform`
: Defines the XSLT APIs that let you transform XML into other forms.

The "Simple API" for XML (SAX) is the event-driven, serial-access mechanism that does element-by-element processing. The API for this level reads and writes XML to a data repository or the Web. For server-side and high-performance apps, you will want to fully understand this level. But for many applications, a minimal understanding will suffice.

The DOM API is generally an easier API to use. It provides a relatively familiar tree structure of objects. You can use the DOM API to manipulate the hierarchy of application objects it encapsulates. The DOM API is ideal for interactive applications because the entire object model is present in memory, where it can be accessed and manipulated by the user.

On the other hand, constructing the DOM requires reading the entire XML structure and holding the object tree in memory, so it is much more CPU and memory

intensive. For that reason, the SAX API will tend to be preferred for server-side applications and data filters that do not require an in-memory representation of the data.

Finally, the XSLT APIs defined in `javax.xml.transform` let you write XML data to a file or convert it into other forms. And, as you'll see in the XSLT section, of this tutorial, you can even use it in conjunction with the SAX APIs to convert legacy data to XML.

The Simple API for XML (SAX) APIs

The basic outline of the SAX parsing APIs are shown at right. To start the process, an instance of the `SAXParserFactory` class is used to generate an instance of the parser.

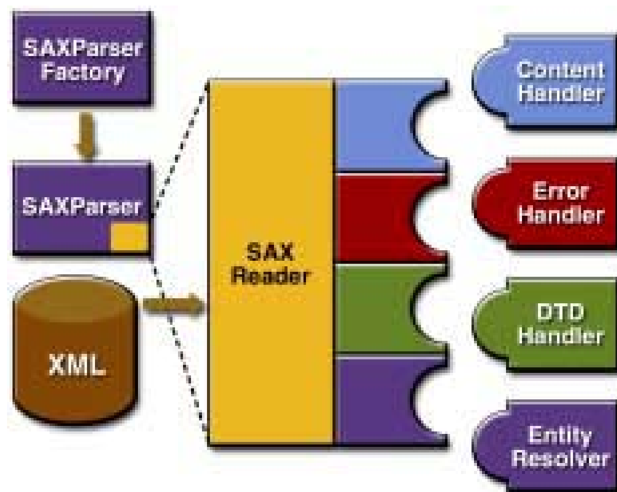

Figure 4–1 SAX APIs

The parser wraps a `SAXReader` object. When the parser's `parse()` method is invoked, the reader invokes one of several callback methods implemented in the application. Those methods are defined by the interfaces `ContentHandler`, `ErrorHandler`, `DTDHandler`, and `EntityResolver`.

Here is a summary of the key SAX APIs:

`SAXParserFactory`
: A `SAXParserFactory` object creates an instance of the parser determined by the system property, `javax.xml.parsers.SAXParserFactory`.

`SAXParser`
: The `SAXParser` interface defines several kinds of `parse()` methods. In general, you pass an XML data source and a `DefaultHandler` object to the parser, which processes the XML and invokes the appropriate methods in the handler object.

`SAXReader`
: The `SAXParser` wraps a `SAXReader`. Typically, you don't care about that, but every once in a while you need to get hold of it using `SAXParser`'s `getXMLReader()`, so you can configure it. It is the `SAXReader` which carries on the conversation with the SAX event handlers you define.

`DefaultHandler`
: Not shown in the diagram, a `DefaultHandler` implements the `ContentHandler`, `ErrorHandler`, `DTDHandler`, and `EntityResolver` interfaces (with null methods), so you can override only the ones you're interested in.

`ContentHandler`
: Methods like `startDocument`, `endDocument`, `startElement`, and `endElement` are invoked when an XML tag is recognized. This interface also defines methods `characters` and `processingInstruction`, which are invoked when the parser encounters the text in an XML element or an inline processing instruction, respectively.

`ErrorHandler`
: Methods `error`, `fatalError`, and `warning` are invoked in response to various parsing errors. The default error handler throws an exception for fatal errors and ignores other errors (including validation errors). That's one reason you need to know something about the SAX parser, even if you are using the DOM. Sometimes, the application may be able to recover from a validation error. Other times, it may need to generate an exception. To ensure the correct handling, you'll need to supply your own error handler to the parser.

`DTDHandler`
: Defines methods you will generally never be called upon to use. Used when processing a DTD to recognize and act on declarations for an *unparsed entity.*

`EntityResolver`
: The `resolveEntity` method is invoked when the parser must identify data identified by a URI. In most cases, a URI is simply a URL, which specifies the location of a document, but in some cases the document may be identified by a URN—a *public identifier*, or name, that is unique in the Web space.

The public identifier may be specified in addition to the URL. The `EntityResolver` can then use the public identifier instead of the URL to find the document, for example to access a local copy of the document if one exists.

A typical application implements most of the `ContentHandler` methods, at a minimum. Since the default implementations of the interfaces ignore all inputs except for fatal errors, a robust implementation may want to implement the `ErrorHandler` methods, as well.

The SAX Packages

The SAX parser is defined in the following packages listed in Table 4–1.

Table 4–1 SAX Packages

Package	Description
`org.xml.sax`	Defines the SAX interfaces. The name `org.xml` is the package prefix that was settled on by the group that defined the SAX API.
`org.xml.sax.ext`	Defines SAX extensions that are used when doing more sophisticated SAX processing, for example, to process a document type definitions (DTD) or to see the detailed syntax for a file.
`org.xml.sax.helpers`	Contains helper classes that make it easier to use SAX—for example, by defining a default handler that has null-methods for all of the interfaces, so you only need to override the ones you actually want to implement.
`javax.xml.parsers`	Defines the `SAXParserFactory` class which returns the `SAXParser`. Also defines exception classes for reporting errors.

The Document Object Model (DOM) APIs

Figure 4–2 shows the JAXP APIs in action:

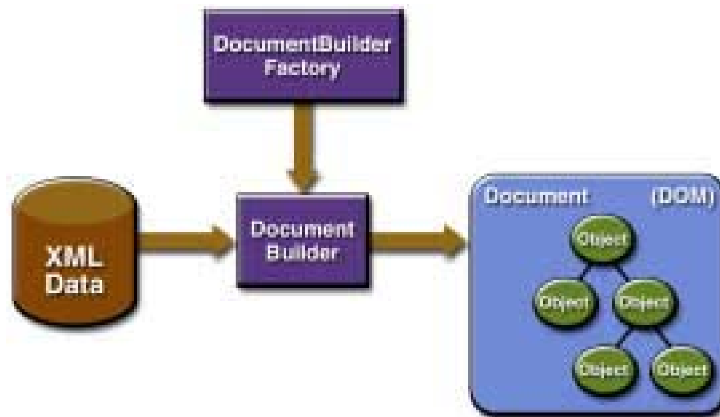

Figure 4–2 DOM APIs

You use the `javax.xml.parsers.DocumentBuilderFactory` class to get a DocumentBuilder instance, and use that to produce a `Document` (a DOM) that conforms to the DOM specification. The builder you get, in fact, is determined by the System property, `javax.xml.parsers.DocumentBuilderFactory`, which selects the factory implementation that is used to produce the builder. (The platform's default value can be overridden from the command line.)

You can also use the `DocumentBuilder newDocument()` method to create an empty `Document` that implements the `org.w3c.dom.Document` interface. Alternatively, you can use one of the builder's parse methods to create a `Document` from existing XML data. The result is a DOM tree like that shown in the diagram.

Note: Although they are called objects, the entries in the DOM tree are actually fairly low-level data structures. For example, under every *element node* (which corresponds to an XML element) there is a *text node* which contains the name of the element tag! This issue will be explored at length in the DOM section of the tutorial, but users who are expecting objects are usually surprised to find that invoking the `text()` method on an element object returns nothing! For a truly object-oriented tree, see the JDOM API at `http://www.jdom.org`.

The DOM Packages

The Document Object Model implementation is defined in the packages listed in Table 4–2.:

Table 4–2 DOM Packages

Package	Description
`org.w3c.dom`	Defines the DOM programming interfaces for XML (and, optionally, HTML) documents, as specified by the W3C.
`javax.xml.parsers`	Defines the `DocumentBuilderFactory` class and the `DocumentBuilder` class, which returns an object that implements the W3C Document interface. The factory that is used to create the builder is determined by the `javax.xml.parsers` system property, which can be set from the command line or overridden when invoking the `new Instance` method. This package also defines the `ParserConfigurationException` class for reporting errors.

The XML Stylesheet Language for Transformation (XSLT) APIs

Figure 4–3 shows the XSLT APIs in action.

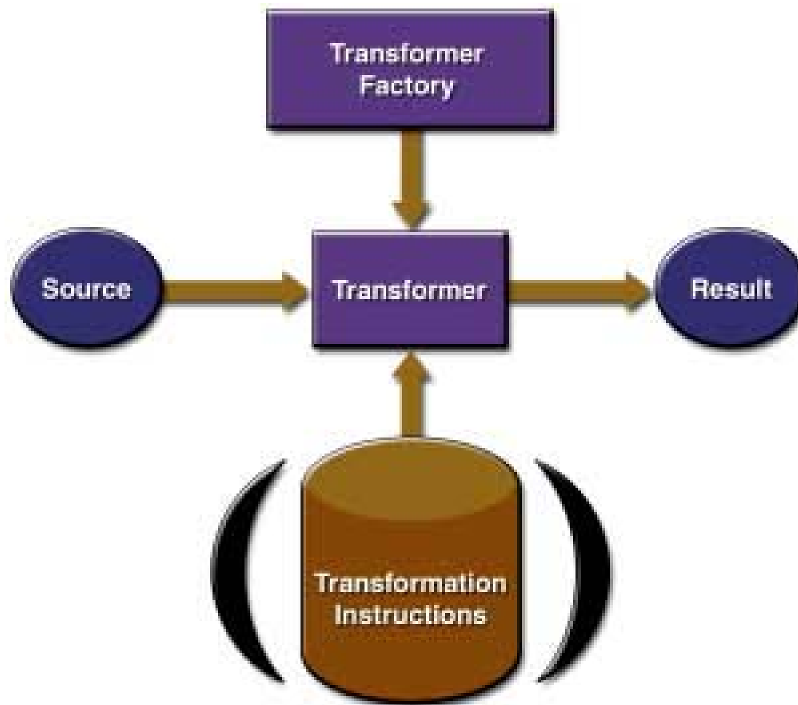

Figure 4–3 XSLT APIs

A `TransformerFactory` object is instantiated, and used to create a `Transformer`. The source object is the input to the transformation process. A source object can be created from SAX reader, from a DOM, or from an input stream.

Similarly, the result object is the result of the transformation process. That object can be a SAX event handler, a DOM, or an output stream.

When the transformer is created, it may be created from a set of transformation instructions, in which case the specified transformations are carried out. If it is created without any specific instructions, then the transformer object simply copies the source to the result.

The XSLT Packages

The XSLT APIs are defined in the following packages:

Table 4–3 XSLT Packages

Package	Description
`javax.xml.transform`	Defines the `TransformerFactory` and `Transformer` classes, which you use to get a object capable of doing transformations. After creating a transformer object, you invoke its `transform()` method, providing it with an input (source) and output (result).
`javax.xml.transform.dom`	Classes to create input (source) and output (result) objects from a DOM.
`javax.xml.transform.sax`	Classes to create input (source) from a SAX parser and output (result) objects from a SAX event handler.
`javax.xml.transform.stream`	Classes to create input (source) and output (result) objects from an I/O stream.

Compiling and Running the Programs

In the J2EE 1.4 Application Server, the JAXP libraries are distributed in the directory `<J2EE_HOME>/lib/endorsed`. To run the sample programs, you'll need to used the Java 2 platform's "endorsed standards" mechanism to access those libraries. For details, see Compiling and Running the Program (page 139).

Where Do You Go from Here?

At this point, you have enough information to begin picking your own way through the JAXP libraries. Your next step from here depends on what you want to accomplish. You might want to go to:

Simple API for XML (page 127)

If the data structures have already been determined, and you are writing a server application or an XML filter that needs to do fast processing.

Document Object Model (page 187)

If you need to build an object tree from XML data so you can manipulate it in an application, or convert an in-memory tree of objects to XML. This part of the tutorial ends with a section on namespaces.

XML Stylesheet Language for Transformations (page 261)

If you need to transform XML tags into some other form, if you want to generate XML output, or (in combination with the SAX API) if you want to convert legacy data structures to XML.