**XML**

***“Trying to wrap your brain around XML is sort of like trying to put an octopus in a bottle. Every time you think you have it under control, a new tentacle shows up. XML has many tentacles, reaching out in all directions.”-*** (Dick Baldwin)

**XML** stands for **extensible markup language**.

**XML** is the basic building block of Web services. All the Web services technologies specified by the **WS-I Basic Profile 1.0** are built on XML and the W3C XML Schema Language.

* **Markup language** is the language using which you can build other languages like, **HTML, XML**.
* XML is defined and governed by **W3Org**.
* The first and final version of XML is **XML 1.0**.
* XML is the document which represents data. Unlike C, C++, Java etc.
* XML is not a programming language.
* It is the de facto standard for exchanging information between computer systems.
* When we represent data in an xml document it would be more structured and has well defined semantics attached to it.
* “**Semantics**” here represents, what a particular data field is representing or stands for, so that any person reading the xml document would be able to interpret in the same manner.
* Another significant feature of XML is, once written it is portable across the platforms (Windows, Linux etc.).
* No changes are required to carry the data across platforms.

***XML is not a single technology, but a group of related technologies that continually adds new members.***

*XML is a lingua-franca that simplifies business-to-business transactions on the web.*

Every language has keywords. If you take example as C, it has keywords like (if, for, while, do, break, continue etc.) but when it comes to **XML there are no keywords or reserved words**. What you write will become the element of that XML document.

***Vendor independence in the data-formatting context#***

***"Other successful Internet technologies let people run their systems without having to take into account another company's own computer systems, notably:***

***TCP/IP for networking, Java for programming, Web browsers for content delivery. XML fills the data formatting piece of the puzzle.****”*

***"These technologies do not create dependencies. It means you can build solutions that are completely agnostic about the platforms and software that you***

***use.****“*

* ***Phipps, IBM's chief XML and Java evangelist***

**Note:**

**What is SGML?**

SGML is an ISO standard (ISO 8879:1986) which provides a formal notation for the definition of generalized markup languages. SGML is not a language in itself. Rather, it is a meta-language that is used to define other languages.

**SGML: the three parts**

An SGML document is really the combination of three parts. Let's refer to the parts as files (but they don't have to be separate physical files).

One file contains the content of the document (words, pictures, etc.). This is the part that the author wants to expose to the client.

A second file is the DTD that defines the accepted syntax. A third file is a *stylesheet* that establishes how the content that conforms to the DTD is to be rendered on the output device.

This is how the author wants the material to be presented to the client.

**HTML versus SGML**

HTML implements some of the concepts derived from SGML but in effect the DTD and the Style Sheet are hard-coded into the browser software.

Because each browser manufacturer has some flexibility in implementing the intended style, the same document will sometimes look different when rendered with two different browsers. This is a (wanted) shortcoming of HTML.

Web page designers are constantly faced with the problem of designing work around to compensate for the deficiencies in some versions of some browsers being used to view the page.

**SGML – HTML**

What the world needs now is...

What the Web community needs is an approach where a standard browser is simply a rendering engine that validates a document according to a given DTD and renders it according to a given stylesheet.

A package deal

The combination of the document, the DTD, and the stylesheet would constitute a package delivered by a server to the browser.

The author of the document would provide the DTD and the stylesheet in addition to the data to be rendered. Then the author could be more confident that it would be rendered properly, especially for complex data.

**SGML – HTML – XML**

The two extremes With HTML, the DTD and the stylesheet are essentially

hard-coded into the browser.

With SGML, the processor requires both a DTD and a stylesheet.

XML, the middle ground With XML, the DTD is optional but the stylesheet (or some processing mechanism that substitutes for a stylesheet) is required.

**Characteristics of XML:**

There are three important characteristics of XML that make it useful in a variety of systems and solutions:

* **XML is extensible:**

XML allows you to create your own self-descriptive tags, or language, that suits your application.

* **XML carries the data, does not present it:**

XML allows you to store the data irrespective of how it will be presented.

* **XML is a public standard:**

XML was developed by an organization called the World Wide Web Consortium (W3C) and is available as an open standard.

**An XML document must have a root tag.**

**An XML document is an information unit that can be seen in two ways:**

As a linear sequence of characters that contain characters data and markup.

As an abstract data structure that is a tree of nodes.

**Why XML ?**

XML is a text-based markup language that is fast becoming the **standard for data interchange on the Web**. As with HTML, you identify data using tags

(Identifiers enclosed in angle brackets, like this: <...>). Collectively, the tags are known as "**markup**".

But unlike HTML, XML tags *identify* the data, rather than specifying how to display it. Where an HTML tag says something like "display this data in bold font"

(<b>...</b>), an XML tag acts like a field name in your program. It puts a label on a piece of data that identifies it .

Example:

<message>...</message>).

**Note:**

Since identifying the data gives you some sense of what *means* (how to interpret it, what you should do with it), XML is sometimes described as a mechanism for specifying the *semantics* (meaning) of the data.

In the same way that you define the field names for a data structure, you are free to use any XML tags that make sense for a given application. Naturally, though, for multiple applications to use the same XML data, they have to agree on the tag names they intend to use.

**Here is an example of some XML data you might use for a messaging application:**

***<message>***

***<to>****you@yourAddress.com****</to>***

***<from>****me@myAddress.com****</from>***

***<subject>****XML Is Really Cool****</subject>***

***<text>***

*How many ways is XML cool? Let me count the ways...*

***</text>***

***</message>***

**Note:**

The content of the <to>tag is entirely contained within the scope of the <message>..</message> tag. It is this ability for one tag to contain others that gives XML its ability to represent hierarchical data structures.

**XML Usage :**

**A short list of XML usage says it all**:

* XML can work behind the scene to simplify the creation of HTML documents for large web sites.
* XML can be used to exchange the information between organizations and systems.
* XML can be used for offloading and reloading of databases.
* XML can be used to store and arrange the data, which can customize your data handling needs.
* XML can easily be merged with style sheets to create almost any desired output.
* Virtually, any type of data can be expressed as an XML document.

**What is Markup?**

XML is a markup language that defines set of rules for encoding documents in a format that is both human-readable and machine-readable. So *what exactly is a markup language?*

Markup is information added to a document that enhances its meaning in certain ways, in that it identifies the parts and how they relate to each other. More specifically, a markup language is a set of symbols that can be placed in the text of a document to demarcate and label the parts of that document.

**Following example shows how XML markup looks, when embedded in a piece of text:**

*<message>*

*<text>Hello, world!</text>*

*</message>*

This snippet includes the markup symbols, or the tags such as <message>...</message> and <text>... </text>.

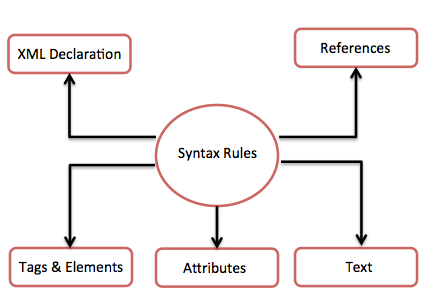
The tags <message> and </message> mark the start and the end of the XML code fragment. The tags <text> and </text> surround the text Hello, world!.

**Is XML a Programming Language?**

A programming **language consists of grammar rules and its own vocabulary** which is used to create computer programs. These programs instructs computer to perform specific tasks.

XML does not qualify to be a programming language as it does not perform any computation or algorithms. It is usually stored in a simple text file and is processed by special software that is capable of interpreting XML.

**The following diagram depicts the syntax rules to write different types of markup and text in an XML document.**



Let us see each component of the above diagram in detail:

## XML Declaration

The XML document can optionally have an XML declaration (**PROLOG**). It is written as below:

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

Where *version* is the XML version and *encoding* specifies the character encoding used in the document.

**The minimal prolog contains a declaration that identifies the document as an**

**XML document, like this:**

<?xml version="1.0"?>

**The declaration may also contain additional information, like this:**

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

**The XML declaration is essentially the same as the HTML header, <html>, except that it uses <?..?> And it may contain the following attributes:**

**Version**

Identifies the version of the XML markup language used in the data. This attribute is not optional.

**Encoding**

Identifies the character set used to encode the data. "ISO-8859-1" is "Latin-1" the Western European and English language character set. (The default is compressed Unicode: UTF-8.)

**Standalone**

Tells whether or not this document references an external entity or an external data type specification (see below). If there are no external references, then "yes" is appropriate

The prolog can also contain definitions of entities (items that are inserted when you reference them from within the document) and specifications that tell which tags are valid in the document, both declared in a Document Type Definition (DTD) that can be defined directly within the prolog, as well as with pointers to external specification files. But those are the subject of later tutorials.

**Note:** The declaration is actually optional. But it's a good idea to include it whenever you create an XML file. The declaration should have the version number, at a minimum, and ideally the encoding as well. That standard simplifies things, if the XML standard is extended in the future, and if the data ever needs to be localized for different geographical regions.

Everything that comes after the XML prolog constitutes the document's *content*.

### Syntax Rules for XML declaration:

* The XML declaration is **case sensitive** and must begin with "<?xml>" where "xml" is written in lower-case.
* If document contains XML declaration, then it strictly needs to be the first statement of the XML document.
* The XML declaration strictly needs be the first statement in the XML document.
* An HTTP protocol can override the value of *encoding* that you put in the XML declaration.

**Why Is XML Important?**

There are a number of reasons for XML's surging acceptance. This section lists a few of the most prominent.

**Plain Text**

Since XML is not a binary format, you can create and edit files with anything from a standard text editor to a visual development environment. That makes it easy to debug your programs, and makes it useful for storing small amounts of data. At the other end of the spectrum, an XML front end to a database makes it possible to efficiently store large amounts of XML data as well. So XML provides scalability for anything from small configuration files to a company-wide data repository.

**Data Identification**

XML tells you what kind of data you have, not how to display it. Because the markup tags identify the information and break up the data into parts, an email program can process it, a search program can look for messages sent to particular people, and an address book can extract the address information from the rest of the message. In short, because the different parts of the information have been identified, they can be used in different ways by different applications.

**Stylability**

When display is important, the stylesheet standard, XSL, lets you dictate how to portray the data.

**For example, the stylesheet for:**

*<to>you@yourAddress.com</to>*

**Can say:**

*1. Start a new line.*

*2. Display "To:" in bold, followed by a space*

*3. Display the destination data.*

This produces:

***To:*** *you@yourAddress*

Of course, you could have done the same thing in HTML, but you wouldn't be able to process the data with search programs and address-extraction programs and the like. More importantly, since XML is inherently style-free, you can use a completely different stylesheet to produce output in postscript, TEX, PDF, or some new format that hasn't even been invented yet. That flexibility amounts to what one author described as "futureproofing" your information. The XML documents you author today can be used in future document-delivery systems that haven't even been imagined yet.

**Inline Reusabiliy**

One of the nicer aspects of XML documents is that they can be composed from separate entities. You can do that with HTML, but only by linking to other documents. Unlike HTML, XML entities can be included "in line" in a document. The included sections look like a normal part of the document -- you can search the whole document at one time or download it in one piece. That lets you modularize your documents without resorting to links. You can single-source a section so that an edit to it is reflected everywhere the section is used, and yet a document composed from such pieces looks for all the world like a one-piece document.

**Linkability**

Thanks to HTML, the ability to define links between documents is now regarded as a necessity. This initiative lets you define two-way links, multiple-target links,

"expanding" links (where clicking a link causes the targeted information to appear inline), and links between two existing documents that are defined in a third.

**Easily Processed**

As mentioned earlier, regular and consistent notation makes it easier to build a program to process XML data. For example, in HTML a <dt> tag can be delimited by </dt>, another <dt>, <dd>, or </dl>. That makes for some difficult programming. But in XML, the <dt> tag must always have a </dt> terminator, or else it will be defined as a <dt/> tag. That restriction is a critical part of the constraints that make an XML document well-formed. (Otherwise, the XML parser won't be able to read the data.) And since XML is a vendor-neutral standard, you can choose among several XML parsers, any one of which takes the work out of processing XML data.

**Hierarchical**

Finally, XML documents benefit from their hierarchical structure. Hierarchical document structures are, in general, faster to access because you can drill down to the part you need, like stepping through a table of contents. They are also easier to rearrange, because each piece is delimited. In a document, for example, you could move a heading to a new location and drag everything under it along with the heading, instead of having to page down to make a selection, cut, and then paste the selection into a new location.

**How Can You Use XML?**

**There are several basic ways to make use of XML:**

* **Traditional data processing**, where XML encodes the data for a program to process.
* **Document-driven programming**, where XML documents are containers that build interfaces and applications from existing components.
* **Archiving** -- the foundation for document-driven programming, where the customized version of a component is saved (archived) so it can be used later.
* **Binding**, where the DTD or schema that defines an XML data structure is used to automatically generate a significant portion of the application that will eventually process that data

**Traditional Data Processing**

XML is fast becoming the data representation of choice for the Web. It's terrific when used in conjunction with network-centric Java-platform programs that send and retrieve information. So a client/server application, for example, could transmit XML-encoded data back and forth between the client and the server.

In the future, XML is potentially the answer for data interchange in all sorts of transactions, as long as both sides agree on the markup to use.

For example, should an email program expect to see tags named <FIRST> and <LAST>, or <FIRSTNAME> and <LASTNAME>?

The need for common standards will generate a lot of industry-specific standardization efforts in the years ahead. In the meantime, mechanisms that let you "translate" the tags in an XML document will be important. Such mechanisms include projects like the RDF initiative, which defines "meta tags", and the XSL specification, which lets you translate XML tags into other XML tags.

**Document-Driven Programming (DDP)**

The newest approach to using XML is to construct a document that describes how an application page should look. The document, rather than simply being displayed, consists of references to user interface components and business-logic components that are "hooked together" to create an application on the fly.

Of course, it makes sense to utilize the Java platform for such components. Both Java BeansTM for interfaces and Enterprise Java BeansTM for business logic can be used to construct such applications. Although none of the efforts undertaken so far are ready for commercial use, much preliminary work has already been done.

**Binding**

Once you have defined the structure of XML data using either a DTD or the one of the schema standards, a large part of the processing you need to do have already been defined.

For example, if the schema says that the text data in a <date> element must follow one of the recognized date formats, then one aspect of the validation criteria for the data has been defined -- it only remains to write the code. Although a DTD specification cannot go the same level of detail, a DTD (like a schema) provides a grammar that tells which data structures can occur, in what sequences. That specification tells you how to write the high level code that processes the data elements.

But when the data structure (and possibly format) is fully specified, the code you need to process it can just as easily be generated automatically. That process is known as *binding* -- creating classes that recognize and process different data elements by processing the specification that defines those elements. As time goes on, you should find that you are using the data specification to generate significant chunks of code, so you can focus on the programming that is unique to your application.

**Archiving**

The Holy Grail of programming is the construction of reusable, modular components. Ideally, you'd like to take them off the shelf, customize them, and plug them together to construct an application, with a bare minimum of additional coding and additional compilation.

The basic mechanism for saving information is called *archiving*. You archive a component by writing it to an output stream in a form that you can reuse later. You can then read it in and instantiate it using its saved parameters.

For example, if you saved a table component, its parameters might be the number of rows and columns to display.

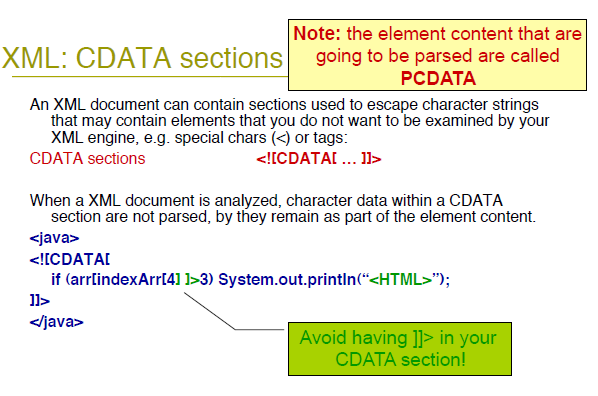
Archived components can also be shuffled around the Web and used in a variety of ways. When components are archived in binary form, however, there are some limitations on the kinds of changes you can make to the underlying classes if you want to retain compatibility with previously saved versions. If you could modify the archived version to reflect the change, that would solve the problem. But that's hard to do with a binary object.

Such considerations have prompted a number of investigations into using XML for archiving. But if an object's state were archived in text form using XML, then anything and everything in it could be changed as easily as you can say, "search and replace".

XML's text-based format could also make it easier to transfer objects between applications written in different languages. For all of these reasons, XML-based archiving is likely to become an important force in the not-too-distant future.

**Summary**

XML is pretty simple, and very flexible. It has many uses yet to be discovered -- we are just beginning to scratch the surface of its potential. It is the foundation for a great many standards yet to come, providing a common language that different computer systems can use to exchange data with one another. As each industry-group comes up with standards for what they want to say, computers will begin to link to each other in ways previously unimaginable.



## Tags and Elements

An XML file is structured by several XML-elements also called **XML-nodes** or **XML-tags**.

XML-elements' names are enclosed by triangular brackets < > as shown below:

*<element>*

### Syntax Rules for Tags and Elements

**Element Syntax:**

Each XML-element needs to be closed either with start or with end elements as shown below:

*<element>....</element>*

or in simple-cases, just this way:

*<element/>*

**Nesting of elements:**

An XML-element can contain multiple XML-elements as its children, but the children elements must not overlap. i.e., an end tag of an element must have the same name as that of the most recent unmatched start tag.

**Following example shows incorrect nested tags:**

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

*<contact-info>*

*<company>TutorialsPoint*

*<contact-info>*

*</company>*

**Following example shows correct nested tags:**

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

***<contact-info>***

***<company>TutorialsPoint</company>***

***<contact-info>***

**Root element:**

An XML document can have only one root element.

For example, following is not a correct XML document, because both the x and y elements occur at the top level without a root element:

***<x>...</x>***

***<y>...</y>***

**The following example shows a correctly formed XML document:**

***<root>***

***<x>...</x>***

***<y>...</y>***

***</root>***

**Case sensitivity:**

The names of XML-elements are case-sensitive. That means the name of the start and the end elements need to be exactly in the same case.

For example **<contact-info>** is different from **<Contact-Info>**.

## Attributes

An **attribute** specifies a single property for the element, using a name/value pair. An XML-element can have one or more attributes.

**For example:**

*<a href="http://www.tutorialspoint.com/">Tutorialspoint!</a>*

Here *href* is the attribute name and *http://www.tutorialspoint.com/* is attribute value.

### Syntax Rules for XML Attributes

* Attribute names in XML (unlike HTML) are case sensitive. That is, *HREF* and *href* are considered two different XML attributes.
* **Same attribute cannot have two values in syntax**. The following example shows incorrect syntax because the attribute *b* is specified twice:

***<a b="x" c="y" b="z">....</a>***

* Attribute names are defined without quotation marks, whereas attribute values must always appear in quotation marks. Following example demonstrates incorrect xml syntax:

***<a b=x>....</a>***

In the above syntax, the attribute value is not defined in quotation marks.

* As in HTML, the attribute name is followed by an equal sign and the attribute value, and multiple attributes are separated by spaces. Unlike HTML, however, in XML commas between attributes are not ignored -- if present, they generate an error.

**Valid:**

*<test>*

*<tag attribute1="a" attribute2="b" attribute3="c">TAGS</tag>*

*</test>*

**Invalid:**

*<test>*

*<tag attribute1="a", attribute2="b", attribute3="c">TAGS</tag>*

*</test>*

**Error in xmlspy:**

This file is not well-formed: Whitespaces (Blank, Tab, CR, LF) expected.

## XML References

*References* usually allow you to add or include additional text or markup in an XML document. References always begin with the symbol **"&"**, which is a reserved character and end with the symbol **";"**.

**XML has two types of references:**

**Entity References:**

An entity reference contains a name between the start and the end delimiters.

**Example:**

**&amp;**

Where *amp* is *name*.

The *name* refers to a predefined string of text and/or markup.

**Character References:**

These contain references, such as **&#65;**, contains a hash mark (“#”) followed by a number. The number always refers to the Unicode code of a character. In this case, 65 refer to alphabet "A".

## XML Text

* The names of XML-elements and XML-attributes are case-sensitive, which means the name of start and end elements need to be written in the same case.
* To avoid character encoding problems, all XML files should be saved as **Unicode UTF-8 or UTF-16** files.
* **Whitespace characters** like blanks, tabs and line-breaks between XML-elements and between the XML-attributes will be **ignored**.
* Some characters are reserved by the XML syntax itself. Hence, they cannot be used directly.

**To use them, some replacement-entities are used, which are listed below:**

|  |  |  |
| --- | --- | --- |
| **not allowed character** | **replacement-entity** | **character description** |
| < | &lt; | less than |
| > | &gt; | greater than |
| & | &amp; | ampersand |
| ' | &apos; | apostrophe |
| " | &quot; | quotation mark |

**XML Primer**

The eXtensible Markup Language (XML) is a meta-language—a language for defining other languages—defined by a specification. It's not a product you can buy. As a specification, XML defines the rules for creating XML markup languages.

An XML markup language defines a set of tags that are used to organize and describe text. Tags are usually paired; together, a start tag, an end tag, and everything between them are called an **element**.

**Example**

You could save the addresses of your friends, family members, and business associates in a text file using XML as shown below.

**XML Address Document**

|  |
| --- |
| *<?xml version="1.0" encoding="UTF-8" ?>*  *<addresses>*  *<address category="friend">*  *<name>Sreekanth</name>*  *<street>Center of the city</street>*  *<city>Hyderabad</city>*  *<state>TS</state>*  *<zip>500087</zip>*  *</address>*  *<address category="business">*  *<name>Amazon.com</name>*  *<street>madhapur</street>*  *<city>Hyderabad</city>*  *<state>TS</state>*  *<zip>500081</zip>*  *</address>*  *</addresses>* |

Even if you don't know anything about XML, the above sample xml is easy to understand. It's easy to figure out which data is the street information and which is the city, and where each address begins and ends. Compare this to other text formats like comma-delimited or tab-delimited data. To understand the contents of the text you are reading, you need to look at a separate document that describes the organization of the data. With XML along with the data semantics about the data is attached along with it which makes it easy of use.

XML documents are composed of Unicode text (usually UTF-8), so people as well as software can understand them. In other words, you can open an XML document and read it in any text. Because XML's syntactical rules are strict, however, you can also parse and manipulate it with a variety of software tools.

As said earlier XML is not a programming language, so it doesn’t have any pre-defined set of reserve words or key words. There are pre-defined set of elements in XML. It is the language with which you will build up your own mark-up.

In proper XML-speak, the term XML application means the use of XML; it doesn't mean "computer program." The term XML application is synonymous with XML markup language.

The relationship of XML to a specific XML markup language, like the Address Book Markup Language or MathML, is analogous to the relationship of the Java programming language to a program or code library (package) written in that language. The Java language specification defines the legal syntax of the programming language, but developers can create any Java program or package they want as long as it adheres to the Java language syntax. The XML specification defines the legal syntax of every XML markup language, but developers can create any XML markup language they want as long as it adheres to XML syntax. While useful, this analogy should not be misunderstood: XML is not a programming language like Java, C++, or VisualBasic.NET. XML only defines the syntax of elements used in text—it is not software and isn't compiled, interpreted, or executed. It's just plain text.

An XML parser is a utility that can read and analyze an XML document. In most cases an XML parser is combined with a parser API (such as SAX2 or DOM 2) that allows a developer to interact with the XML document while it's being parsed, or after.

**XML Document Instance**

An XML document can be saved or transferred over a network.

A Web page written in XHTML (a variant of HTML), which is a text file, is an XML document. Similarly, a SOAP message, which is generated and exchanged over a network, is an XML document. A business might choose to store address information as an XML document. In this case the text file might look like below:

**An XML Address Document Instance**

|  |
| --- |
| *<?xml version="1.0" encoding="UTF-8" ?>*  *<address >*  *<name>Sreekanth</name>*  *<street>Center of the city</street>*  *<city>Hyderabad</city>*  *<state>TS</state>*  *<zip>500087</zip>*  *</address>* |

The above example is called an **XML document instance**, which means it represents one possible set of data for a particular markup language. It might be saved as a file or sent over the Internet as the payload of a SOAP message.

If you were to create another XML document with the same tags but different contents (like a different street or Zip code) it would be considered a different XML document instance.

**Anatomy of an XML Document**

An XML document is made up of declarations, elements, attributes, text data, comments, and other components. This section examines an XML document instance in detail and explains its most important components.

1. **XML Declaration**

An XML document may start with an XML declaration, but it's not required.

An XML declaration declares the version of XML used to define the document (there is only one version at this time, version 1.0). It may also indicate the character encoding used to store or transfer the document, and whether the document is standalone or not.

The following snippet shows the XML declaration in bold.

|  |
| --- |
| ***<?xml version="1.0" encoding="UTF-8" ?>***  *<address>*  *<name>Amazon.com</name>*  *<street>1516 2nd Ave</street>*  *<city>Seattle</city>*  *<state>WA</state>*  *<zip>90952</zip>*  *</address>* |

1. **XML Element**

Element in an XML is written in angular braces for e.g. <beans>. In XML there are two types of elements as follows

1. **Start Element/Opening Tag:**

Start element is the element which is written in <elementname> indicating the start of a block.

1. **End Element/End Tag:**

End element is the element which is written in </elementname> indicating the end of a block.

As everything is written in terms of start and end elements, XML is said to be more structured in nature. An XML Element may contain content or may contain other elements under it. So, XML elements which contain other elements in it are called as **Compound elements or XML Containers**.

XML markup languages organize data hierarchically, in a tree structure, where each branch of the tree is called an **element** and is delimited by a pair of tags. All elements are named and have a start tag and an end tag. A start tag looks like <tagname> and an end tag looks like </tagname>. The tagname is a label that usually describes the information contained by the element. Between the start and end tags, an element may contain text or other elements, which themselves may contain text or more elements. The following is an example, based on an XML instance of the Address Markup Language, which I'll call **Address Markup for short.**

|  |
| --- |
| *<?xml version="1.0" encoding="UTF-8" ?>*  *<address>*  *<name>Amazon.com</name>*  *<street>1516 2nd Ave</street>*  *<city>Seattle</city>*  *<state>WA</state>*  *<zip>90952</zip>*  *</address>* |

There are six elements in this example (address, name, street, city, state, and zip).

The address element uses the start tag <address> and the end tag </address>, and contains the other five elements. The address element, because it contains all the other elements, is referred to as the root element.

Each XML document must have one root element, and that element must contain all the other elements and text, except the XML declaration, comments, and certain processing instructions. The other elements (name, street, city, state, zip) all contain text.

**According to the WS-I Basic Profile 1.0, XML documents used in Web services must use either UTF-8 or UTF-16 encoding.**

This limitation simplifies things for Web service vendors and makes interoperability easier, because there is only one character encoding standard to worry about, Unicode.

UTF-8 and UTF-16 encoding allows you to use characters from English, Chinese, French, German, Japanese, and many other languages.

* An element name must always begin with a letter or underscore, but can contain pretty much any Unicode character you like, including underscores, letters, digits, hyphens, and periods.
* Some characters may not be used: /, <, >, ?, ", @, &, and others.
* Also, an element name must never start with the string xml, as this is reserved by the XML 1.0 specification.

**Empty tags:**

As long as you follow XML's rules you may name elements anything and your elements may contain any combination of valid text and other elements. Elements do not have to contain any data at all. It's perfectly acceptable to use an empty-element tag, a single tag of the form <tagname/>, which is interpreted as a pair of start and end tags with no content (<tagname></tagname>).

Empty-element tags are typically used when an element has no data, when it acts like flag, or when it’s pertinent data is contained in its attributes (attributes are described in the next section).

One really big difference between XML and HTML is that an XML document is always constrained to be well formed. There are several rules that determine when a document is well-formed, but one of the most important is that every tag has a closing tag. So, in XML, the </to>tag is not optional. The <to>element is never terminated by any tag other than </to>.

**Note:** Another important aspect of a well-formed document is that all tags are completely nested. So you can have <message>..<to>..</to>..</message>, but never <message>..<to>..</message>..</to>.

Sometimes, though, it makes sense to have a tag that stands by itself. For example, you might want to add a "flag" tag that marks message as important. A tag like that doesn't enclose any content, so it's known as an "empty tag". You can create an empty tag by ending it with /> instead of >.

**For example, the following message contains such a tag:**

<message to="you@yourAddress.com" from="me@myAddress.com"

subject="XML Is Really Cool">

**<flag/>**

<text>

**Note:** The empty tag saves you from having to code <flag></flag> in order to have a well-formed document. You can control which tags are allowed to be empty by creating a DTD/XSD. If there is no DTD, then the document can contain any kinds of tags you want, as long as the document is well-formed.

**XML Attribute**

If we want to have supplementary information attach to an element, instead of having it as content or another element, we can write it as an Attribute of the element.

**Example*:***

|  |
| --- |
| ***<book type=”entertainment”>***  ***<isbn>isbn1001</isbn>***  ***</book>*** |

In the above example “**book” is an element** which contains one attributes type which acts as a supplementary information. **Isbn is the sub-element** of the book element.

An element may have one or more attributes. You use an attribute to supplement the data contained by an element, to provide information about it not captured by its contents. For example, we could describe the kind of address in an XML address document by declaring a category attribute as

**Using Attributes in XML**

|  |
| --- |
| *<?xml version="1.0" encoding="UTF-8" ?>*  *<address* ***category="business****" >*  *<name>Amazon.com</name>*  *<street>1516 2nd Ave</street>*  *<city>Seattle</city>*  *<state>WA</state>*  *<zip>90952</zip>*  *</address>* |

Each attribute is a name-value pair.

The value must be in single or double quotes. You can define any number of attributes for an element, but a particular attribute may occur only once in a single element. Attributes cannot be nested like elements. Attribute names have the same restrictions as element names. Attributes must be declared in the start tag and never the end tag of an element. In many cases, empty-element tags (discussed in previous section) are used when the attributes contain all the data. For example, we could add an empty phone element to the XML address document

**Using the Empty-Element Tag in XML**

|  |
| --- |
| *<?xml version="1.0" encoding="UTF-8" ?>*  *<address category="business" >*  *<name>Amazon.com</name>*  *<street>1516 2nd Ave</street>*  *<city>Seattle</city>*  *<state>WA</state>*  *<zip>90952</zip>*  *<phone countrycode="01" areacode="715" number="55529482" ext="341" />*  *</address>* |

Using attributes instead of nested elements is considered a matter of style, rather than convention. There are no "standard" design conventions for using attributes or elements.

**Comments**

You can add comments to an XML document just as you can add comments to a Java program. A comment is considered documentation about the XML document and is not part of the data it describes. Comments are placed between a <!-- designator and a --> designator, as in HTML: <!-- comment goes here -->. As an example we can comment our XML address document as shown.

**Using Comments in XML**

|  |
| --- |
| *<?xml version="1.0" encoding="UTF-8" ?>*  ***<!-- This document contains address information -->***  *<address category="business" >*  *<name>Amazon.com</name>*  *<street>1516 2nd Ave</street>*  *<city>Seattle</city>*  *<state>WA</state>*  *<zip>90952</zip>*  *</address>* |

**CDATA Section**

An element may contain other elements, text, or a mixture of both.

When an element contains text, you have to be careful about which characters you use because certain characters have special meaning in XML. Using quotes (single or double), less-than and greater-than signs (< and >), the ampersand (&), and other special characters in the contents of an element will confuse parsers, which consider these characters to be special parsing symbols.

To avoid parsing problems you can use escape characters like &gt for greater-than or &amp for ampersand, but this technique can become cumbersome.

**A CDATA section allows you to mark a section of text as literal so that it will not be parsed for tags and symbols, but will instead be considered just a string of characters.**

**For example:**

If you want to put HTML in an XML document, but you don't want it parsed, you can embed it in a CDATA section.

**In example below the address document contains a note in HTML format.**

**Using a CDATA Section in XML**

|  |
| --- |
| *<?xml version="1.0" encoding="UTF-8" ?>*  ***<!-- This document contains address information -->***  *<address category="business" >*  *<name>Amazon.com</name>*  *<street>1516 2nd Ave</street>*  *<city>Seattle</city>*  *<state>WA</state>*  *<zip>90952</zip>*  ***<note>***  ***<![CDATA[***  ***<html>***  ***<body>***  ***<p> Last time I contacted <b>Amazon.com</b> I spoke to ...***  ***</body>***  ***</html>***  ***]]>***  ***</note>***  *</address>* |

**CDATA Sections take the form <![CDATA[ text goes here ]]>** .

If we include the HTML in the note element **without embedding** it in a CDATA section, XML processors will parse it as Address Markup, instead of treating it as ordinary text, causing two kinds of problems:

**First**, HTML's syntax isn't as strict as XML's so parsing problems are likely. **Second**, the HTML is not actually part of Address Markup; it's simply a part of the text contained by the note element, and we want it treated as literal text.

XML (eXtensible Markup Language) is used to organize documents and business data. XML files can be stored or transmitted between two applications on a network.

Basically, they are just plain text documents that contain special tags that label different parts of a document or fields of data.

**For example**, the XML document in Listing 1-1 contains Amazon.com's mailing address.

**Well-formness**

As how any programming language has syntax in writing its code, Well-formness of XML document talks about how to write an XML document. Well-formness indicates the readability nature of an XML document. In other way if an XML document is said to be well-formed then it is readable in nature.

**Following are the rules that describe the Well-formness of an XML Document:**

1) **Every XML document must start with PROLOG:**

Prolog stands for processing instruction, and typically used for understanding about the version of XML used and the data encoding used.

***Example*:** ***<?xml version=”1.0” encoding=”utf-8”?>***

1. **Root Element:**

XML document must contain a root element, and should be the only one root element. All the other elements should be the children of root element.

3) **Level constraint:**

Every start element must have an end element and the level at which you open a start element, the same level you need to close your end element as well.

***Example:***

|  |
| --- |
| *<?xml version=”1.0” encoding=”utf-8”?>*  *<student>*  *<info>*  *<rollno>42</rollno>*  *<name>John</info>* ***(-- info is closed in-correctly --)***  *</name>*  *<student>* |

If any XML is said to follow the above defined rules then it is termed as well-formed.

**XML Usage**

**An XML document is used in two scenarios**

1. **Used for transferring information**:

As said earlier XML is a document representing data and is used for carrying information between two computer systems in an Interoperable manner.

1. **Configurations**:

In J2EE world every component/resource you develop like Servlet or EJB, it has to be deployed into a Web Server.

**Example:**

In a Servlet application, the path with which the servlet has to be accessible should be specified to the Servlet container. So that it can map the incoming request to the Servlet. This is done in a web.xml.

When we provide the configuration information in a XML file the main advantage is, the same xml document can be used in different platforms without changing anything.

**Validatity**

Every XML in-order to parse (read) should be well-formed in nature. As said earlier well-formness of an XML document indicates whether it is readable or not, it doesn’t talks about whether the data contained in it is valid or not.

Validity of the XML document would be defined by the application which is going to process your XML document.

**Let’s consider a scenario as follows.**

You want to get driving license. In order to get a **driving license** you need to follow certain process like filling the RTA forms and signing them and submitting to the **RTA department**.

Instead of this can you write your own format of letter requesting the driving license from an RTA department, which seems to be not relevant because driving license is something that would be issued by the RTA department. So, the whole and sole authority of defining what should a person has to provide data to get a driving license will lies in the hands of RTA department rather than you.

In the same way when an application is going to process your xml document, the authority of defining what should be there as part of that xml is lies in the hands of the application which is going to process your document.

**For example let’s consider the below xml fragment:**

***Example: PurchaseOrder xml document***

|  |
| --- |
| *<?xml version=”1.0” encoding=”utf-8”?>*  *<purchaseOrder>*  *<orderItems>*  *<item>*  *<itemCode>IC323</itemCode>*  *<quantity>24</quantity>*  *</item>*  *<item>*  *<itemCode>IC324</itemCode>*  *<quantity>abc</quantity>*  *</item>*  *</orderItems>*  *<purchaseOrder>* |

In the above xml even though it confirms to all the well-formness rules, it cannot be used for business transaction, as the 2nd <quantity> element carries the data as “abc” which doesn’t makes any sense.

**“So, in order to check for data validity we need to define the validation criteria of an XML document in either a DTD or XSD document”.**

**These are the basic standards you need to be familiar with. They come up in pretty much any discussion of XML.**

**SAX**

**Simple API for XML**

This API was actually a product of collaboration on the XML-DEV mailing list, rather than a product of the W3C. It's included here because it has the same "**final**" characteristics as a W3C recommendation.

You can also think of this standard as the "**Serial Access**" protocol for XML. This is the fast-to-execute mechanism you would use to read and write XML data in a server, for example. This is also called an “**Event-Driven Protocol”**, because the technique is to register your handler with a SAX parser, after which the parser invokes your callback methods whenever it sees a new XML tag (or encounters an error, or wants to tell you anything else).

**DOM**

**Document Object Model**

The Document Object Model protocol converts an XML document into a collection of objects in your program. You can then manipulate the object model in any way that makes sense. This mechanism is also known as the "**random access**" protocol, because you can visit any part of the data at any time. You can then modify the data, remove it, or insert new data.

**DTD**

**Document Type Definition**

The DTD specification is actually part of the XML specification, rather than a separate entity. On the other hand, it is optional - - you can write an XML document without it. And there are a number of schema proposals that offer more flexible alternatives. So it is treated here as though it were a separate specification.

A DTD specifies the kinds of tags that can be included in your XML document, and the valid arrangements of those tags. You can use the DTD to make sure you don't create an invalid XML structure. You can also use it to make sure that the XML structure you are reading (or that got sent over the net) is indeed valid.

Unfortunately, it is difficult to specify a DTD for a complex document in such a way that it prevents all invalid combinations and allows all the valid ones. So constructing a DTD is something of an art. The DTD can exist at the front of the document, as part of the prolog. It can also exist as a separate entity, or it can be split between the document prolog and one or more additional entities.

However, while the DTD mechanism was the first method defined for specifying valid document structure, it was not the last. Several newer schema specifications have been devised.

**Namespaces**

The namespace standard lets you write an XML document that uses two or more sets of XML tags in modular fashion.

Suppose for example that you created an XML-based parts list that uses XML descriptions of parts supplied by other manufacturers (online!). The "price" data supplied by the subcomponents would be amounts you want to total up, while the "price" data for the structure as a whole would be something you want to display.

The namespace specification defines mechanisms for qualifying the names so as to eliminate ambiguity. That lets you write programs that use information from other sources and do the right things with it.

**XSL**

**Extensible Stylesheet Language**

The XML standard specifies how to identify data, not how to display it. HTML, on the other hand, told how things should be displayed without identifying what they were.

The XSL standard has two parts, XSLT (the transformation standard, described next) and XSL-FO (the part that covers *formatting objects*, also known as *flow objects*). XSL-FO gives you the ability to define multiple areas on a page and then link them together. When a text stream is directed at the collection, it fills the first area and then "flows" into the second when the first area is filled. Such objects are used by newsletters, catalogs, and periodical publications.

**XSLT (+XPATH)**

**Extensible Stylesheet Language for Transformations**

The XSLT transformation standard is essentially a translation mechanism that lets you specify what to convert an XML tag into so that it can be displayed -- for example, in HTML. Different XSL formats can then be used to display the same data in different ways, for different uses. (The XPATH standard is an addressing mechanism that you use when constructing transformation instructions, in order to specify the parts of the XML structure you want to transform.)

**RELAX**

**Regular Language description for XML**

Simpler than XML Structure Schema, RELAX uses XML syntax to express the structure relationships that are present in a DTD, and adds the XML Datatype Schema mechanisms, as well. Includes a DTD to RELAX converter.

**SOX**

**Schema for Object-oriented XML**

SOX is a schema proposal that includes extensible data types, namespaces, and embedded documentation.

**TREX**

**Tree Regular Expressions for XM**

A means of expressing validation criteria by describing a *pattern* for the structure and content of an XML document. Includes

a RELAX to TREX converter.

**Schematron**

**Schema for Object-oriented XML**

An assertion-based schema mechanism that allows for sophisticated validation.

**Linking and Presentation Standards**

Arguably the two greatest benefits provided by HTML were the ability to link between documents, and the ability to create simple formatted documents (and, eventually, very complex formatted documents).

**The following standards aim at preserving the benefits of HTML in the XML arena, and to adding additional functionality, as well.**

**XML Linking**

These specifications provide a variety of powerful linking mechanisms, and are sure to have a big impact on how XML documents are used.

**XLink:**

The XLink protocol is a proposed specification to handle links between XML documents. This specification allows for some pretty sophisticated linking, including two-way links, links to multiple documents, "expanding" links that insert the linked information into your document rather than replacing your document with a new page, links between two documents that are created in a third, independent document, and indirect links (so you can point to an "address book" rather than directly to the target document -- updating the address book then automatically changes any links that use it).

**XML Base:**

This standard defines an attribute for XML documents that defines a "base" address, that is used when evaluating a relative address specified in the document. (So, for example, a simple file name would be found in the base-address directory.)

**XPointer:**

In general, the XLink specification targets a document or document-segment using its ID. The XPointer specification defines mechanisms for "addressing into the internal structures of XML documents", without requiring the author of the document to have defined an ID for that segment. To quote the spec, it provides for "reference to elements, character strings, and other parts of XML documents, whether or not they bear an explicit ID attribute".

**XHTML**

The XHTML specification is a way of making XML documents that look and act like HTML documents. Since an XML document can contain any tags you care to define, why not define a set of tags that look like HTML? That's the thinking behind the XHTML specification, at any rate. The result of this specification is a document that can be displayed in browsers and also treated as XML data. The data may not be quite as identifiable as "pure" XML, but it will be a heck of a lot easier to manipulate than standard HTML, because XML specifies a good deal more regularity and consistency.

For example, every tag in a well-formed XML document must either have an end-tag associated with it or it must end in />.

So you might see <p>...</p>, or you might see <p/>, but you will never see <p> standing by itself. The upshot of that requirement is that you never have to program for the weird kinds of cases you see in HTML where, for example, a <dt> tag might be terminated by </DT>, by another <DT>, by <dd>, or by </dl>. That makes it a lot easier to write code!

**RDF**

**Resource Description Framework**

RDF is a proposed standard for defining data about data. Used in conjunction with the XHTML specification, for example, or with HTML pages, RDF could be used to describe the content of the pages. For example, if your browser stored your ID information as FIRSTNAME, LASTNAME, and EMAIL, an RDF description could make it possible to transfer data to an application that wanted NAME and EMAILADDRESS. Just think: One day you may not need to type your name and address at every web site you visit!

**RDF Schema**

The RDF Schema proposal allows the specification of consistency rules and additional information that describe how the statements in a Resource Description Framework (RDF) should be interpreted.

**XTM**

**XML Topic Maps**

In many ways a simpler, more readily usable knowledge-representation than RDF, the topic maps standard is one worth watching. So far, RDF is the W3C standard for knowledge representation, but topic maps could possibly become the "developer's choice" among knowledge representation standards.

**Standards That Build on XML**

The following standards and proposals build on XML. Since XML is basically a language-definition tool, these specifications use it to define standardized languages for specialized purposes.

**Extended Document Standards**

These standards define mechanisms for producing extremely complex documents -- books, journals, magazines, and the like -- using XML.

**SMIL**

**Synchronized Multimedia Integration Language**

SMIL is a W3C recommendation that covers audio, video, and animations. It also addresses the difficult issue of synchronizing the playback of such elements.

**MathML**

**Mathematical Markup Language**

MathML is a W3C recommendation that deals with the representation of mathematical formulas.

**SVG**

**Scalable Vector Graphics**

SVG is a W3C working draft that covers the representation of vector graphic images. (Vector graphic images that are built from commands that say things like "draw a line (square, circle) from point x,y to point m,n" rather than encoding the image as a series of bits. Such images are more easily scalable, although they typically require more processing time to render.)

**DrawML**

**Drawing Meta Language**

DrawML is a W3C note that covers 2D images for technical illustrations. It also addresses the problem of updating and refining such images.

**eCommerce Standards**

These standards are aimed at using XML in the world of business-to-business (B2B) and business-to-consumer (B2C) commerce.

**ICE**

**Information and Content Exchange**

ICE is a protocol for use by content syndicators and their subscribers. It focuses on "automating content exchange and reuse, both in traditional publishing contexts and in business-to-business relationships".

**ebXML**

**Electronic Business with XML**

This standard aims at creating a modular electronic business framework using XML. It is the product of a joint initiative by the United Nations (UN/CEFACT) and the Organization for the Advancement of Structured Information Systems (OASIS).

**cxml**

**Commerce XML**

cxml is a RosettaNet (www.rosettanet.org) standard for setting up interactive online catalogs for different buyers, where the pricing and product offerings are company specific. Includes mechanisms to handle purchase orders, change orders, status updates, and shipping notifications.

**CBL**

**Common Business Library**

CBL is a library of element and attribute definitions maintained by CommerceNet ([www.commerce.net](http://www.commerce.net)).

**An Overview of the APIs**

This gives you a map so you can find your way around JAXP and the associated XML APIs. The first step is to understand where JAXP fits in with respect to the major Java APIs for XML:

**JAXP: Java API for XML Parsing**

This API is the subject of the present tutorial. It provides a common interface for creating and using the standard SAX, DOM, and XSLT APIs in Java, regardless of which vendor's implementation is actually being used..

**JAXB: Java Architecture for XML Binding**

This standard defines a mechanism for writing out Java objects as XML *marshalling*) and for creating Java objects from such structures (*unmarshalling*). (You compile a class description to create the Java classes, and use those classes in your application.)

**JDOM: Java DOM**

The standard DOM is a very simple data structure that intermixes text nodes, element nodes, processing instruction nodes, CDATA nodes, entity references, and several other kinds of nodes. That makes it difficult to work with in practice, because you are always sifting through collections of nodes, discarding the ones you don't need into order to process the ones you are interested in. JDOM, on the other hand, creates a tree of *objects* from an XML structure. The resulting tree is much easier to use, and it can be created from an XML structure without a compilation step.

For more information on JDOM, visit http://www.jdom.org. For information on the Java Community Process (JCP) standards effort for JDOM, see JSR 102.

**DOM4J**

Although it is not on the JCP standards track, DOM4J is an open-source, object-oriented alternative to DOM that is in many ways ahead of JDOM in terms of implemented features. As such, it represents an excellent alternative for Java developers who need to manipulate XML-based data.

For more information on DOM4J, see http://www.dom4j.org.

**JAXM: Java API for XML Messaging**

The JAXM API defines a mechanism for exchanging *asynchronous* XML-based messages between applications.("Asynchronous" means "send it and forget it".)

**JAX-RPC: Java API for XML-based Remote Process Communications**

The JAX-RPC API defines a mechanism for exchanging *synchronous* XML-based messages between applications. ("Synchronous" means "send a message and wait for the reply".)

**JAXR: Java API for XML Registries**

The JAXR API provides a mechanism for publishing available services in an external registry, and for consulting the registry to find those services.

**The JAXP APIs**

Now that you know where JAXP fits into the big picture, the remainder of this page discusses 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 the reference implementation.

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

**An Overview of the Packages**

As discussed in the previous section, 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.

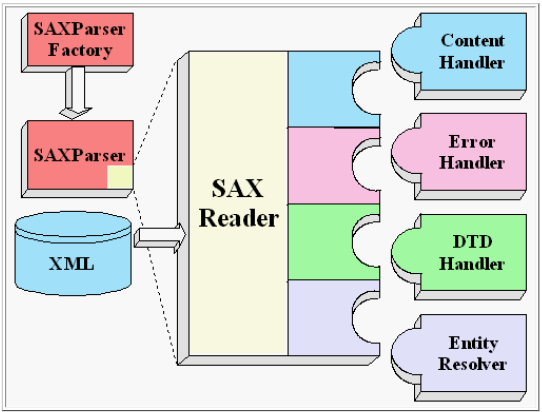
On the other hand, constructing the DOM requires reading the entire XML structure and holding the object tree in memory, so it is much more CPU and memory intensive. For that reason, the SAX API will tend to be preferred for server-side applications and data filters that do not require an in-memory representation 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 is shown at right. To start the process, an instance of the SAXParserFactory classed is used to generate an instance of the parser.

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**

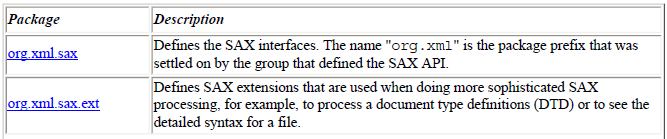
The resolveEntity method is invoked when the parser must identify data identified by a URI. In most cases, a URI is simply a URL, which specifies the location of a document, but in some cases the document may be identified by a URN -- a *public identifier*, or name, that is unique in the web space. The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

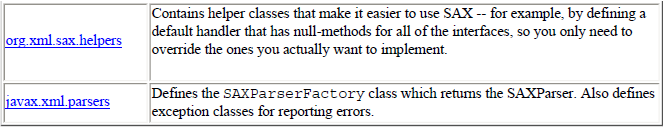
A typical application implements most of the ContentHandler methods, at a minimum. Since the default implementations of the interfaces ignore all inputs except for fatal errors, a robust implementation may want to implement the ErrorHandler methods, as well.

**The SAX Packages**

The SAX parser is defined in the following packages.

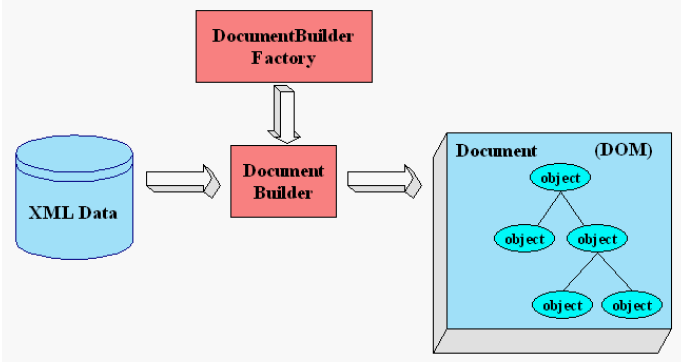
***Package Description***





**The Document Object Model (DOM) APIs**

The diagram below shows the JAXP APIs in action:



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.

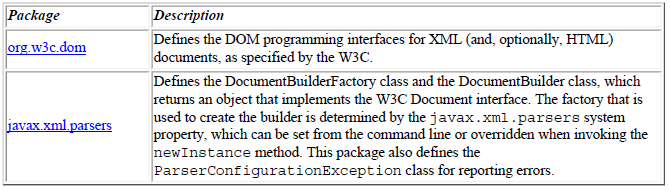
**Note:**

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.

**The DOM Packages**

The Document Object Model implementation is defined in the following packages:



**The XML Style Sheet Translation (XSLT) APIs**

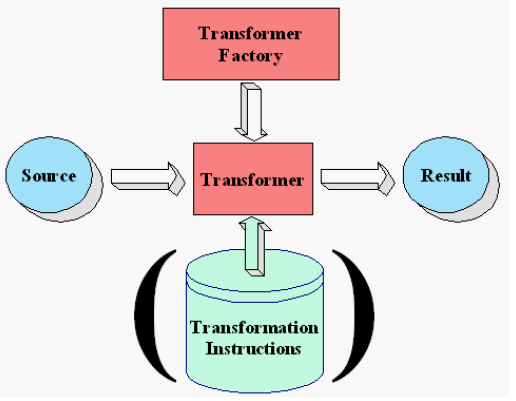
A TransformerFactory object is instantiated, and used to create a Transformer. The source object is the input to the transformation process. A source object can be created from SAX reader, from a DOM, or from an input stream.

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.

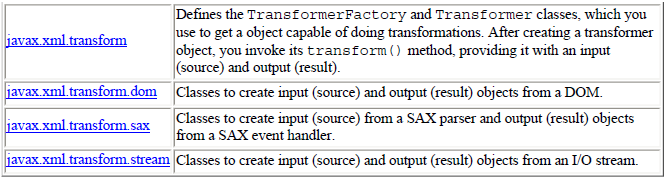
When the transformer is created, it may be created from a set of transformation instructions, in which case the specified transformations are carried out. If it is created without any specific instructions, then the transformer object simply copies the source to the result.

**The XSLT Packages**

The XSLT APIs are defined in the following packages:

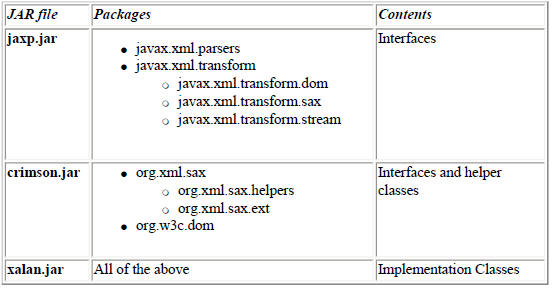






**Overview of the JAR Files**

Here are the jar files that make up the JAXP bundles, along with the interfaces and classes they contain.



**Note:**

When defining the classpath, specify the jar files in the order shown here: jaxp.jar, crimson.jar, xalan.jar.

**Forced Choices between Elements and Attributes:**

Sometimes, the choice between an attribute and an element is forced on you by the nature of attributes and elements. Let's look at a few of those considerations:

**The data contains substructures**

In this case, the data item must be modeled as an *element*. It can't be modeled as an attribute, because attributes take only simple strings. So if the title can contain emphasized text like this:

The <em>Best</em> Choice, then the title must be an element.

**The data contains multiple lines**

Here, it also makes sense to use an *element*. Attributes need to be simple, short strings or else they become unreadable, if not unusable.

**The data changes frequently**

When the data will be frequently modified, especially by the end user, then it makes sense to model it as an *element*. XML-aware editors tend to make it very easy to find and modify element data. Attributes can be somewhat harder to get to, and therefore somewhat more difficult to modify.

**The data is a small, simple string that rarely if ever changes**

This is data that can be modeled as an *attribute*. However, just because you *can* does not mean that you should. Check the "Stylistic Choices" section below, to be sure.

**The data is confined to a small number of fixed choices**

Here is one time when it really makes sense to use an *attribute*. Using the DTD, the attribute can be prevented from taking on any value that is not in the preapproved list. An XML-aware editor can even provide those choices in a drop-down list. Note, though, that the gain in validity restriction comes at a cost in extensibility. The author of the XML document cannot use any value that is not part of the DTD. If another value becomes useful in the future, the DTD will have to be modified before the document author can make use of it.

**Stylistic Choices**

As often as not, the choices are not as cut and dried as those shown above. When the choice is not forced, you need a sense of "style" to guide your thinking. The question to answer, then, is what makes good XML style, and why.

Defining a sense of style for XML is, unfortunately, as nebulous a business as defining "style" when it comes to art or music. There are a few ways to approach it, however. The goal of this section is to give you some useful thoughts on the subject of "XML style".

**Visibility**

The first heuristic for thinking about XML elements and attributes uses the concept of *visibility*. If the data is intended to be shown -- to be displayed to some end user -- then it should be modeled as an element. On the other hand, if the information guides XML processing but is never displayed, then it may be better to model it as an attribute. For example, in order-entry data for shoes, shoe size would definitely be an element. On the other hand, a manufacturer's code number would be reasonably modeled as an attribute.

**Consumer / Provider**

Another way of thinking about the visibility heuristic is to ask who is the consumer and/or provider of the information. The shoe size is entered by a human sales clerk, so it's an element.

The manufacturer's code number for a given shoe model, on the other hand, may be wired into the application or stored in a database, so that would be an attribute. (If it were entered by the clerk, though, it should perhaps be an element.) You can also think in terms of whom or what is processing the information. Things can get a bit murky at that end of the process, however. If the information "consumers" are order-filling clerks, will they need to see the manufacturer's code number? Or, if an order-filling program is doing all the processing, which data items should be elements in that case? Such philosophical distinctions leave a lot of room for differences in style.

**Container vs. Contents**

Another way of thinking about elements and attributes is to think of an element as a *container*. To reason by analogy, the *contents* of the container (water or milk) correspond to XML data modeled as elements. On the other hand, *characteristics* of the container (blue or white, pitcher or can) correspond to XML data modeled as attributes. Good XML style will, in some consistent way, separate each container's contents from its characteristics.

To show these heuristics at work: In a slideshow the type of the slide (executive or technical) is best modeled as an attribute. It is a characteristic of the slide that lets it be selected or rejected for a particular audience. The title of the slide, on the other hand, is part of its contents. The visibility heuristic is also satisfied here. When the slide is displayed, the title is shown but the type of the slide isn't. Finally, in this example, the consumer of the title information is the presentation audience, while the consumer of the type information is the presentation program.

**Normalizing Data**

In the SAX tutorial, the section Defining Attributes and Entities in the DTD shows how to create an external entity that you can reference in an XML document. Such an entity has all the advantages of a modularized routine -- changing that one copy affects every document that references it. The process of eliminating redundancies is known as *normalizing*, so defining entities is one good way to normalize your data.

In an HTML file, the only way to achieve that kind of modularity is with HTML links -- but of course the document is then fragmented, rather than whole. XML entities, on the other hand, suffer no such fragmentation. The entity The result is a DOM tree like that shown in the diagram reference acts like a macro -- the entity's contents are expanded in place, producing a whole document, rather than a fragmented one. And when the entity is defined in an external file, multiple documents can reference it.

The considerations for defining an entity reference, then, are pretty much the same as those you would apply to modularize program code:

1. Whenever you find yourself writing the same thing more than once, think entity.

That lets you write it one place and reference it multiple places.

2. If the information is likely to change, especially if it is used in more than one place, definitely think in terms of defining an entity. An example is defining productName as an entity so that you can easily change the documents when the product name changes.

3. If the entity will never be referenced anywhere except in the current file, define it in the local\_subset of the document's DTD, much as you would define a method or inner class in a program.

4. If the entity will be referenced from multiple documents, define it as an external entity, the same way that would define any generally usable class as an external class.

External entities produce modular XML that is smaller, easier to update and maintain. They can also make the resulting document somewhat more difficult to visualize, much as a good OO design can be easy to change, once you understand it, but harder to wrap your head around at first.

You can also go overboard with entities. At an extreme, you could make an entity reference for the word "the" -- it wouldn't buy you much, but you could do it.

**Note:**

The larger an entity is, the less likely it is that changing it will have unintended effects.

When you define an external entity that covers a whole section on installation instructions, for example, making changes to the section is unlikely to make any of the documents that depend on it come out wrong. Small inline substitutions can be more problematic, though.

For example, if productName is defined as an entity, the name change can be to a different part of speech, and that can kill you! Suppose the product name is something like "HtmlEdit". That's a verb. So you write, "You can HtmlEdit your file...". Then, when the official name is decided, it's "Killer". After substitution, that becomes "You can Killer your file...". Argh. Still, even if such simple substitutions can sometimes get you in trouble, they can also save a lot of work. To be totally safe, though, you could set up entities named productNoun, productVerb, productAdj, and productAdverb!

**Normalizing DTDs**

Just as you can normalize your XML document, you can also normalize your DTD declarations by factoring out common pieces and referencing them with a parameter entity. This process is described in the SAX tutorial in Defining Parameter Entities. Factoring out the DTDs (also known as modularizing or normalizing) gives the same advantages and disadvantages as normalized XML -- easier to change, somewhat more difficult to follow.

You can also set up conditionalized DTDs, as described in the SAX tutorial section Conditional Sections. If the number and size of the conditional sections is small relative to the size of the DTD as a whole, that can let you "single source" a DTD that you can use for multiple purposes. If the number of conditional sections gets large, though, the result can be a complex document that is difficult to edit.

**XML: namespaces**

**How do you avoid tag conflicts?**

Since you can define your own tags, if you reuse XML files from other authors you might find tag conflicts.

**These can be avoided by declaring a namespace as an attribute of the root element:**

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

**Processing XML Documents**

Although XML is just plain text, and can be accessed using a common text editor, it's usually read and manipulated by software applications and not by people using text editors.

A software application that reads and manipulates XML documents will use an XML parser.

In general, parsers read a stream of data (usually a file or network stream) and break it down into functional units that can then be processed by a software application.

An XML parser can read an XML document and parse its contents according to the XML syntax. Parsers usually provide a programming API that allows developers to access elements, attributes, text, and other constructs in XML documents.

**There are basically two standard kinds of XML parser APIs:**

* SAX and
* DOM.

SAX (Simple API for XML) was the first standard XML parser API and is very popular. Although several individuals created it,

David Brownell currently maintains SAX2, the latest version, as an open development project at SourceForge.org.

SAX2 parsers are available in many programming languages including Java.

SAX2 is based on an event model. As the SAX2 parser reads an XML document, starting at the beginning, it fires off events every time it encounters a new element, attribute, piece of text, or other component. SAX2 parsers are generally very fast because they read an XML document sequentially and report on the markup as its encountered.

DOM (Document Object Model) was developed after SAX2 and maintained by the W3C.

DOM level 2 (DOM 2) is the current version, but there is a DOM level 3 in the works.

DOM 2 parsers are also available for many programming languages, including Java. DOM 2 presents the programmer with a generic, object-oriented model of an XML document.

Elements, attributes, and text values are represented as objects organized into a hierarchical tree structure that reflects the hierarchy of the XML document being processed.

DOM 2 allows an application to navigate the tree structure, modify elements and attributes, and generate new XML documents in memory. It's a very powerful and flexible programming model, but it's also slow compared to SAX2, and consumes a lot more memory.

In addition to providing a programming model for reading and manipulating XML documents, the parser's primary responsibility is checking that documents are well formed; that is, that their elements, attributes, and other constructs conform to the syntax prescribed by the XML 1.0 specification.

**For example**, an element without an end tag, or with an attribute name that contains invalid characters, will result in a syntax error. A parser may also, optionally, enforce validity of an XML document. An XML document may be well formed, but invalid because it is not organized according to its schema.

**Two popular Java parser libraries:**

Crimson and

Xerces-J, include both SAX2 and DOM 2, so you can pick the API that better meets your needs.

Crimson is a part of the Java 2 platform (JDK 1.4), which means it's available to you automatically. Xerces, which some people feel is better, is maintained by the Apache Software Foundation. You must download it as a JAR file and place it in your classpath (or ext directory) before you can use it. Either parser library is fine for most cases, but Xerces supports W3C XML Schema validation while Crimson doesn't.

JAXP (Java API for XML Processing), which is part of the J2EE platform, is not a parser. It's a set of factory classes and wrappers for DOM 2 and SAX2 parsers.

JAXP eliminates this portability problem by providing a consistent programming model for instantiating and configuring DOM 2 and SAX2 parsers.

JAXP can be used with Crimson or Xerces-J. JAXP is a standard Java extension library, so using it will help keep your J2EE applications portable.

Other non-standard XML APIs are also available to Java developers, including JDOM, dom4j, and XOM. These APIs are tree-based like DOM 2, and although they are non-standard, they tend to provide simpler programming models than DOM 2. JDOM and dom4j are actually built on top of DOM 2 implementations, wrapping DOM 2 with their own object-oriented programming model. JDOM and dom4j can both be used with either Xerces-J or Crimson.

If ease of use is important, you may want to use one of these non-standard parser libraries, but if J2EE portability is more important, stick with JAXP, DOM 2, and SAX2.

**XML Namespaces**

An XML namespace provides a qualified name for an XML element or attribute, the same way that a Java package provides a qualified name for a Java class.

In most Java programs, classes are imported from other packages (java.io, javax.xml, and the rest). When the Java program is compiled, every operation performed on every object or class is validated against the class definition in the appropriate package.

If Java didn't have package names, the classes in the Java core libraries (I/O, AWT, JDBC, etc.) would all be lumped together with developer-defined classes. Java package names allow us to separate Java classes into distinct namespaces, which improves organization and access control, and helps us avoid name conflicts (collisions).

XML namespaces are similar to Java packages, and serve the same purposes; an XML namespace provides a kind of package name for individual elements and attributes.

**An Example of Using Namespaces**

Creating XML documents based on multiple markup languages is often desirable. For example, suppose we are building a billing and inventory control system for a company called Durga-jobs. We can define a standard markup language for address information, the Address Markup Language, to be used whenever an XML document needs to contain address information. An instance of Address Markup is shown below.

**An Instance of the Address Markup Language**

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

*<address category="business" >*

*<name>Amazon.com</name>*

*<street>1516 2nd Ave</street>*

*<city>Seattle</city>*

*<state>WA</state>*

*<zip>90952</zip>*

*</address>*

Address Markup is used in Address Book Markup (nested in the addresses element) defined, but it will also be reused in about half of Durga-jobs Books' other XML markup languages (types of XML documents): Invoice, Purchase Order, Shipping, Marketing, and others.

Address Markup has its own schema, defined using either DTD (Document Type Definition) or the W3C XML Schema Language, which dictates how its elements are organized. Every time we use address information in an XML document, it should be validated against Address Markup's schema.

**For example, the address information is included in the PurchaseOrder XML document.**

**The PurchaseOrder Document Using the Address Markup Langauge**

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

*<purchaseOrder orderDate="2003-09-22" >*

*<accountName>Amazon.com</accountName> <accountNumber>923</accountNumber>*

*<address>*

*<name>AMAZON.COM</name>*

*<street>1850 Mercer Drive</street>*

*<city>Lexington</city>*

*<state>KY</state>*

*<zip>40511</zip>*

*</address>*

*<book>*

*<title>J2EE Web Services</title>*

*<quantity>300</quantity>*

*<wholesale-price>29.99</wholesale-price>*

*</book>*

*<total>8997.00</total>*

*</purchaseOrder>*

If the purchase-order document has its own schema (defined by the Purchase Order Markup Language) and the address information has its own schema (defined by the Address Markup Language), how do we indicate that the address element should conform to the Address Markup Language, while the rest of the elements conform to the Purchase Order Markup Language? We use namespaces.

We can state that the address elements conform to Address Markup by declaring the namespace of Address Markup in the address element. We can do the same thing for the purchase order elements by declaring, in the purchaseOrder element, that they conform to the Purchase Order Markup. Below example illustrates.

**Declaring Namespaces in XML**

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

*<purchaseOrder orderDate="2003-09-22" xmlns="http://www.Durga-jobs.com/jwsbook/PO">*

*<accountName>Amazon.com</accountName> <accountNumber>923</accountNumber>*

*<address xmlns="http://www.Durga-jobs.com/jwsbook/ADDR">*

*<name>AMAZON.COM</name>*

*<street>1850 Mercer Drive</street>*

*<city>Lexington</city>*

*<state>KY</state>*

*<zip>40511</zip>*

*</address>*

*<book>*

*<title>J2EE Web Services</title>*

*<quantity>300</quantity>*

*<wholesale-price>29.99</wholesale-price>*

*</book>*

*<total>8997.00</total>*

*</purchaseOrder>*

The xmlns attribute declares a specific XML namespace in the form xmlns="someURI". The value of an xmlns attribute is a URI reference, which must conform to the URI specification (RFC2396) defined by the IETF (Internet Engineering Task Force).

URIs (Uniform Resource Identifiers) can take many different forms; the most common is the URL (Universal Resource Locator) .

**For example**, the URLs for both namespaces start with [**http://www.Durga-jobs.com/jwsbook**](http://www.Durga-jobs.com/jwsbook),

The final part of the URL (/PO or /ADDR in the example) completes the URL to create a unique identifier for each namespace.

In standard HTTP URLs are used, which may or may not point to an actual document or resource. It's important to remember that the URI used for the XML namespace should be unique to that markup language, but it doesn't have to point to an actual resource or document.

**Default Namespaces, Prefixes, and Qualified Names**

The xmlns declarations made above defined the default namespace for the element and all its descendants. The scope of a default namespace applies only to the element and its descendants, so the xmlns used in the address element applies only to the address, name, street, city, state, and zip elements.

The default xmlns declared in the purchaseOrder element applies to all the elements except the address elements, because the address element overrides the default namespace of the purchaseOrder element to define its own default namespace.

Using default XML namespaces can get tricky, especially when elements are interleaved or when a lot of markup languages are used in the same document.

To simplify things, XML Namespaces defines a shorthand notation for associating elements and attributes with namespaces. You can assign an XML namespace to a prefix, then use that prefix to fully qualify each element name.

The code assigns the prefix "addr:" to the http://www.Durga-jobs.com/jwsbook/ADDR namespace and the prefix "po:" to the http://www.Durga-jobs.com/jwsbook/PO namespace, then uses one prefix or the other to qualify each element.

**Declaring and Using Namespaces Prefixes in XML**

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

*<po:purchaseOrder orderDate="2003-09-22" xmlns:po="http://www.Durga-jobs.com/jwsbook/PO" xmlns:addr="http://www.Durga-jobs.com/jwsbook/ADDR">*

*<po:accountName>Amazon.com</po:accountName> <po:accountNumber>923</po:accountNumber>*

*<addr:address>*

*<addr:name>AMAZON.COM</addr:name>*

*<addr:street>1850 Mercer Drive</addr:street>*

*<addr:city>Lexington</addr:city>*

*<addr:state>KY</addr:state>*

*<addr:zip>40511</addr:zip>*

*</addr:address>*

*<po:book>*

*<po:title>J2EE Web Services</po:title>*

*<po:quantity>300</po:quantity>*

*<po:wholesale-price>29.99</po:wholesale-price>*

*</po:book>*

*<po:total>8997.00</po:total>*

*</po:purchaseOrder>*

The elements prefixed with addr: belong to the http://www.Durga-jobs.com/jwsbook/ADDR namespace and the elements prefixed with po: belong to the http://www.Durga-jobs.com/jwsbook/PurchaseOrder namespace. It's not necessary to qualify every element with a namespace prefix. You can rely on default namespaces to determine the namespaces of all elements not explicitly prefixed, as follows.

**Combining Default Namespaces and Namespaces Prefixes in XML**

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

*<purchaseOrder orderDate="2003-09-22" xmlns="http://www.Durga-jobs.com/jwsbook/PO" xmlns:addr="http://www.Durga-jobs.com/jwsbook/ADDR">*

*<accountName>Amazon.com</accountName> <accountNumber>923</accountNumber>*

*<addr:address>*

*<addr:name>AMAZON.COM</addr:name>*

*<addr:street>1850 Mercer Drive</addr:street>*

*<addr:city>Lexington</addr:city>*

*<addr:state>KY</addr:state>*

*<addr:zip>40511</addr:zip>*

*</addr:address>*

*<book>*

*<title>J2EE Web Services</title>*

*<quantity>300</quantity>*

*<wholesale-price>29.99</wholesale-price>*

*</book>*

*<total>8997.00</total>*

*</purchaseOrder>*

In this example the namespace for the entire document is declared to be http://www.Durga-jobs.com/jwsbook/PO—it's the default for all of the children of the root element, purchaseOrder. Any element that doesn't have a prefix is, by default, a member of <http://www.Durga-jobs.com/jwsbook/PO>.

When a namespace prefix is applied to an element, however, it overrides the default namespace. In, the "addr:" prefix is assigned to the address elements, which makes http://www.Durga-jobs.com/jwsbook/ADDR the namespace of the address, name, street, city, state, and zip elements.

The way you use prefixes with namespaces can depend on how the document is defined by its schema. The schema may determine whether you need to prefix all the elements, or just the parent elements, and whether default namespace declarations apply to unprefixed elements. In XML-speak, a prefix combined with an element name is called a QName, which stands for "qualified name." Conceptually, a QName like addr:address can be dereferenced to http://www.Durga-jobs.com/jwsbook/ADDR:address—but this is not done in practice. A QName has two parts, the XML namespace and the local name.

**For example**, the QName of the street element declared is composed of the http://www.Durga-jobs.com/jwsbook/ADDR XML namespace and the street local name.

XML namespaces based on URLs tend to be universally unique, which makes it easy for parsers and software applications to distinguish between instances of different markup languages within the same document. Namespaces help avoid name collisions, where two elements from different markups share a common local name. For example, a WSDL document can use Durga-jobs's postal address element as well as the SOAP-binding address element in the same document. Although both elements are named address, they belong to different namespaces with different QNames, so there is no name conflict.

**Avoiding Element Name Collisions by Using XML Namespaces** Code View:

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

*<definitions name="Address-Update" targetNamespace="http://www.durga-jobs.org/jwsbook/Address-Update" xmlns:tns="http://www.durga-jobs.org/jwsbook/Address-Update" xmlns:addr="http://www.Durga-jobs.com/jwsbook/ADDR" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns="http://schemas.xmlsoap.org/wsdl/">*

*...*

***<!-- message elements describe the paramters and return values -->***

*<message name="AddressMessage">*

*<part name="address" element="addr:address" />*

*</message>*

***...***

***<!-- service tells us the Internet address of a Web service -->***

*<service name="AddressUpdateService">*

*<documentation>Update a customers mailing address</documentation>*

*<port name="AddressUpdate\_Port" binding="tns:AddressUpdate\_Binding">*

*<soap:address location="http://www.durga-jobs.org/jwsbook/BookPrice" />*

*</port>*

*</service>*

*</definitions>*

XML parsers and other tools can use XML namespaces to process, sort, and search XML elements in a document according to their QNames. This allows reusable code modules to be invoked for specific namespaces.

**For example**, you can create a custom Java tool to map an instance of Address Markup to a relational database. It will be invoked only for address elements that belong to the Address Markup namespace, http://www.Durga-jobs.org/addr, and not for address elements of any other namespace. XML namespaces also allow for a great versioning system.

If the Address Markup changes, we can assign the new version its own namespace, such as http://www.Durga-jobs.org/ADDR-2, so it can be distinguished from its predecessor. We can support both the old and new versions of the Address Markup Language simultaneously, because the parser can uniquely identify each version by its namespace. Each version has its own markup for validation and perhaps its own code modules.

**The W3C XML Schema Language**

SOAP, WSDL, and UDDI are markup languages defined using the W3C XML Schema Language, so understanding the latter is critical to understanding J2EE Web Services. This chapter will provide you with a good understanding of both W3C XML Schema Language basics and, optionally, advanced concepts, so that you are ready to learn about SOAP, WSDL, and the UDDI standards covered later. Throughout this chapter the term XML schema will be used to refer to the W3C XML Schema Language as a technology, while the word schema by itself will refer to a specific XML schema document.