
Creating Markup with XML

Objectives

- To learn how to use XML to create custom markup.
- To understand the concept of an XML parser.
- To use elements and attributes to mark up data.
- To understand the difference between markup text and character data.
- To use **CDATA** sections and processing instructions.
- To understand the concept of an XML namespace.

The chief merit of language is clearness, and we know that nothing detracts so much from this as do unfamiliar terms.
Galen

Every country has its own language, yet the subjects of which the untutored soul speaks are the same everywhere.
Tertullian

The historian, essentially, wants more documents than he can really use; the dramatist only wants more liberties than he can really take.
Henry James

Entities should not be multiplied unnecessarily.
William of Occam

Outline

- 2.1 Introduction
- 2.2 Introduction to XML Markup
- 2.3 Parsers and Well-Formed XML Documents
- 2.4 Characters
 - 2.4.1 Characters vs. Markup
 - 2.4.2 Whitespace, Entity References and Built-In Entities
- 2.5 CDATA Sections and Processing Instructions
- 2.6 XML Namespaces
- 2.7 Summary
- 2.8 Internet and World Wide Web Resources

2.1 Introduction

Web services are based on the Extensible Markup Language (XML), a technology for structuring data. Java and XML share a similar goal: Interoperability. Java's platform independence, the separation of an application from the platform on which it runs, allows executable code to run in any programming environment for which there is a Java Virtual Machine (JVM). XML's data independence, the separation of content from its presentation, allows structured data to be portable across applications. When used together, Java and XML allow applications and their associated data to be deployed on a wide range of platforms; this interoperability and portability are crucial to Web services.

In this chapter, we demonstrate how to output an XML document's contents, using a Java application we created, named **ParserTest**.¹ This application is included with the book's examples at www.deitel.com.

2.2 Introduction to XML Markup

In this section, we begin to use XML to mark up data. Consider a simple XML document (**first.xml**) that marks up a message (Fig. 2.1). We output the entire XML document to the command line.

```
1 <?xml version = "1.0" encoding = "UTF-8"?>
2
3 <!-- Fig. 2.1 : first.xml -->
4 <!-- Simple introduction to XML markup -->
5
6 <myMessage id = "643070">
7   <message>Welcome to XML!</message>
8 </myMessage>
```

Fig. 2.1 Simple XML document containing a message. (Part 1 of 2.)

1. Before running the examples in this chapter, execute the batch file (**jws1_xml.bat**) provided with the book's examples.

```
c:\>java -classpath %CLASSPATH% com.deitel.jws1.xml.ParserTest
chapter2/fig02_1/first.xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- Fig. 2.1 : first.xml -->
<!-- Simple introduction to XML markup -->
<myMessage id="643070">
  <message>Welcome to XML!</message>
</myMessage>
```

Fig. 2.1 Simple XML document containing a message. (Part 2 of 2.)

The document begins with the optional *XML declaration* (line 1), which identifies the document as an XML document. The **version** information parameter specifies the version of XML used in the document. Currently, there is only one version of XML—version 1.0.

The optional **encoding** declaration specifies the method used to represent characters electronically. **UTF-8** is a character encoding typically used for Latin-alphabet characters (e.g., English) that can be stored in one byte. When present, this declaration allows authors to specify a character encoding explicitly. When omitted, either UTF-8 or *UTF-16* (a format for encoding and storing characters in two bytes) is the default. We discuss character encoding in Section 2.4.



Portability Tip 2.1

The **encoding** declaration allows XML documents to be authored in a wide variety of human languages.



Portability Tip 2.2

Although the XML declaration is optional, it should be included to identify the version of XML used in the document. Otherwise, in the future, a document without an XML declaration might be assumed to conform to the latest version of XML. Errors or other serious problems may result.



Common Programming Error 2.1

Placing anything, including whitespace (i.e., spaces, tabs, carriage returns and line feeds), before an XML declaration is an error.



Good Programming Practice 2.1

By convention, XML documents use the file extension **.xml**.

Lines 3–4 are comments, which begin with **<!--** and end with **-->**. Comments can be placed almost anywhere in an XML document and can span multiple lines. For example, we could have written lines 3–4 as

```
<!-- Fig. 2.1 : first.xml
Simple introduction to XML markup -->
```



Common Programming Error 2.2

Placing **--** between **<!--** and **-->** is an error.

In XML, data are marked up using *tags*, which are names enclosed in *angle brackets* (`< >`). Tags are used in pairs to delimit the beginning and end of markup. A tag that begins markup is called a *start tag*, and a tag that terminates markup is called an *end tag*. Examples of start tags include `<myMessage>` and `<message>` (lines 6–7). End tags differ from start tags in that the former contain a *forward slash* (`/`) character. Examples of end tags include `</message>` and `</myMessage>` in lines 7–8.

Individual units of markup (i.e., everything from a start tag to an end tag, inclusive) are called *elements*, which are the most fundamental building blocks of an XML document. XML documents contain exactly one element—called a *root element* (e.g., `myMessage` in lines 6–8)—that contains all other elements in the document. Elements are embedded or nested within each other to form hierarchies—with the root element at the top of the hierarchy. Nesting allows document authors to create explicit relationships between data. XML documents can contain any number of elements.



Common Programming Error 2.3

Improperly nesting XML tags is an error. For example, `<x><y>hello</x></y>` is an error; the nested `<y>` tag must end before the `</x>` tag.



Good Programming Practice 2.2

When creating an XML document, add whitespace to emphasize the document's hierarchical structure. This practice makes documents more readable to humans.



Common Programming Error 2.4

Providing more than one root element in an XML document is an error.

Elements, such as the root element, that contain other elements are called *parent elements*. Elements nested within a parent element are called *children*. Parent elements can have any number of children, but an individual child element can have only one parent. As we will see momentarily, it is possible for an element to be both a parent element and a child element. Element `message` is an example of a child element, and element `myMessage` is an example of a parent element.



Common Programming Error 2.5

XML is case sensitive. Using the wrong mixture of case is an error. For example, pairing the start tag `<message>` with the end tag `</Message>` is an error.

In addition to being placed between tags, data can be placed in *attributes*, which are name–value pairs in start tags. Elements can have any number of attributes. In Fig. 2.1, attribute `id` is assigned the value `"643070"`. XML element and attribute names can be of any length and may contain letters, digits, underscores, hyphens and periods; they must begin with a letter or an underscore.



Good Programming Practice 2.3

XML elements and attribute names should be meaningful. For example, use `<address>` instead of `<adr>`.



Common Programming Error 2.6

Using spaces in an XML element name or attribute name is an error.



Common Programming Error 2.7

Not placing an attribute's value in either single or double quotes is an error.

2.3 Parsers and Well-Formed XML Documents

A software program called an *XML parser*² (or an *XML processor*) is required to process an XML document. XML parsers read the XML document, check its syntax, report any errors and allow programmatic access to the document's contents. An XML document is considered *well formed* if it is syntactically correct (i.e., errors are not reported by the parser when the document is processed). Figure 2.1 is an example of a well-formed XML document.

If an XML document is not well formed, the parser reports errors. For example, if the end tag (line 8) in Fig. 2.1 is omitted, the error message shown in Fig. 2.2 is generated by the parser.

Parsers can support the *Document Object Model (DOM)* and/or the *Simple API for XML (SAX)* for accessing a document's content programmatically, using languages such as Java™, Python and C. A DOM-based parser builds a tree structure in memory that contains the XML document's data. A SAX-based parser processes the document and generates *events* (i.e., notifications to the application) when tags, text, comments, etc., are encountered. These events return data from the XML document. Software programs can “listen” for the events to obtain data from the XML document.

```

1  <?xml version = "1.0" encoding = "UTF-8"?>
2
3  <!-- Fig. 2.2 : error.xml          -->
4  <!-- XML document missing an end tag -->
5
6  <myMessage id = "643070">
7      <message>Welcome to XML!</message>

```

```

c:\>java -classpath %CLASSPATH% com.deitel.jws1.xml.ParserTest
chapter2/fig02_2/error.xml
[Fatal Error] error.xml:24:4: XML document structures must start and
end within the same entity.
Exception in thread "main" org.xml.sax.SAXParseException: XML document
structures must start and end within the same entity.
    at org.apache.xerces.parsers.DOMParser.parse(
DOMParser.java:235)
    at org.apache.xerces.jaxp.DocumentBuilderImpl.parse(
DocumentBuilderImpl.java:201)
    at javax.xml.parsers.DocumentBuilder.parse(Unknown Source)
    at com.deitel.jws1.xml.ParserTest.main(ParserTest.java:43)

```

Fig. 2.2 XML document missing an end tag.

2. In this chapter, we use the reference implementation for the Java API for XML Processing 1.2 (JAXP), which is part of the Java Web Services Pack 1.0.

The examples we present use DOM-based parsing.³ In Chapter 7, we provide a detailed discussion of the DOM. We do not discuss SAX-based parsing in this book.

2.4 Characters

In this section, we discuss the collection of characters—called a *character set*—permitted in an XML document. XML documents may contain carriage returns, line feeds and *Unicode*[®] characters. Unicode is a character set created by the *Unicode Consortium* (www.unicode.org). The Unicode Consortium encodes the vast majority of the world's commercially viable languages.

2.4.1 Characters vs. Markup

Once a parser determines that all characters in a document are legal, it must differentiate between markup text and character data. Markup text is enclosed in angle brackets (< and >). Character data (sometimes called *element content*) are the *text* delimited by the start tag and end tag. Child elements are considered markup—not character data. Lines 1, 3–4 and 6–8 in Fig. 2.1 contain markup text. In line 7, the tags <message> and </message> are the markup text, and the text **Welcome to XML!** is character data.

2.4.2 Whitespace, Entity References and Built-In Entities

Spaces, tabs, line feeds and carriage returns are characters commonly called *whitespace characters*. An XML parser is required to pass all character data in a document, including whitespace characters, to the application (e.g., a Java application) consuming the XML document's data.

Figure 2.3 demonstrates that whitespace characters are passed by the parser to the application consuming the XML document's data. In this case, we simply print the data returned by the parser.

```

1  <?xml version = "1.0" encoding = "UTF-8"?>
2
3  <!-- Fig. 2.3 : whitespace.xml          -->
4  <!-- Demonstrating whitespace, entities -->
5  <!-- and empty elements                -->
6
7  <information>
8
9      <!-- empty element whose attribute value -->
10     <!-- contains significant whitespace    -->
11     <company name = "Deitel  & Associates, Inc." />
12
13     <!-- start tag contains insignificant whitespace -->
14     <city      >  Sudbury      </city>
15

```

Fig. 2.3 Whitespace characters in an XML document. (Part 1 of 2.)

3. Some DOM-based parsers may internally use SAX-based parsing to create the initial tree structure.

```

16
17     <state>Massachusetts</state>
18 </information>

```

```

c:\>java -classpath %CLASSPATH% com.deitel.jws1.xml.ParserTest
chapter2/fig02_3/whitespace.xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- Fig. 2.3 : whitespace.xml -->
<!-- Demonstrating whitespace, entities -->
<!-- and empty elements -->
<information>

    <!-- empty element whose attribute value -->
    <!-- contains significant whitespace -->
    <company name="Deitel  & Associates, Inc."/>

    <!-- start tag contains insignificant whitespace -->
    <city>  Sudbury  </city>

    <state>Massachusetts</state>
</information>

```

Fig. 2.3 Whitespace characters in an XML document. (Part 2 of 2.)

A parser can inform an application as to whether individual whitespace characters are *significant* (i.e., need to be preserved) or *insignificant* (i.e., need not be preserved). The output window illustrates that the majority of whitespace characters in the document are considered significant. Lines 2 and 6 were considered insignificant by the application, as were the extra space characters in the start tag on line 14. Lines 8, 12 and 15–16 were considered significant, because they are part of the character data. We will explore the subtleties of whitespace interpretation in greater detail in Chapter 6.

The element in line 11 is called an *empty element*, because it does not contain character data between its start and end tags. The forward-slash character closes the tag. Alternatively, this empty element can be written as

```
<company name = "Deitel  & Associates, Inc."></company>
```

Both forms are equivalent.

Almost any character can be used in an XML document, but the characters *ampersand* (&) and *left angle bracket* (<) are reserved in XML and may not be used in character data or in attribute values. To use these symbols in character data or in attribute values, *entity references* must be used. Entity references are special names that begin with an ampersand (&) and end with a *semicolon* (;). XML provides entity references (or *built-in entities*) for the ampersand (&#amp;#38;), left angle bracket (<), right angle bracket (>), apostrophe (') and quotation-mark (") characters. XML parsers replace these entity references with their corresponding characters.



Common Programming Error 2.8

Using the left-angle bracket (<) in character data or in attribute values is an error.



Common Programming Error 2.9

Using the ampersand (&)—other than in an entity reference—in character data or in attribute values is an error.

2.5 CDATA Sections and Processing Instructions

In this section, we introduce **CDATA sections**, which can contain text, reserved characters (e.g., <) and whitespace characters, and *processing instructions*, which allow document authors to embed application-specific data within an XML document.

Character data in a **CDATA** section are not processed by the XML parser. A common use of a **CDATA** section is for delimiting programming code (e.g., Java, JavaScript and C++), which often includes characters such as & and <. Character data in a **CDATA** section are passed to the application consuming the XML document's data. Figure 2.4 presents an XML document that compares text in a **CDATA** section with character data.



Common Programming Error 2.10

Placing one or more spaces inside the string `<![CDATA[` or the string `]]>` is an error.

The first **sample** element (lines 8–12) contains C++ code as character data. Each occurrence of <, > and & is replaced by an entity reference. Lines 15–20 use a **CDATA** section to indicate a block of text that the parser should not treat as character data or markup. **CDATA** sections begin with `<![CDATA[` and terminate with `]]>`. Notice that the < and & characters (lines 18–19) do not need to be replaced by entity references.

```

1  <?xml version = "1.0" encoding = "UTF-8"?>
2
3  <!-- Fig. 2.4 : cdata.xml          -->
4  <!-- CDATA section containing C++ code -->
5
6  <book title = "Russ Tick&apos;s C++ Programming" edition = "8">
7
8      <sample>
9          // sample code
10         if ( this->getX() &lt; 5 &amp;&amp; value[ 0 ] != 3 )
11             cerr &lt;&lt; this->displayError();
12     </sample>
13
14     <sample>
15         <![CDATA[
16
17             // sample code
18             if ( this->getX() < 5 && value[ 0 ] != 3 )
19                 cerr << this->displayError();
20         ]]>
21     </sample>
22
23     Russ Tick&apos;s C++ programming
24
```

Fig. 2.4 Using a **CDATA** section. (Part 1 of 2.)

```

25     <?button cpp = "sample.cpp" ansi = "yes"?>
26 </book>

```

```

c:\>java -classpath %CLASSPATH% com.deitel.jws1.xml.ParserTest
chapter2/fig02_4/cdata.xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- Fig. 2.4 : cdata.xml -->
<!-- CDATA section containing C++ code -->
<book edition="8" title="Russ Tick's C++ Programming">

    <sample>
        // Sample code
        if ( this->getX() &lt; 5 &amp;&amp; value[ 0 ] != 3 )
            cerr &lt;&lt; this->displayError();
    </sample>

    <sample>
        <![CDATA[

            // Sample code
            if ( this->getX() < 5 && value[ 0 ] != 3 )
                cerr << this->displayError();

        ]]>
    </sample>

    Russ Tick's C++ programming

    <?button cpp = "sample.cpp" ansi = "yes"?>
</book>

```

Fig. 2.4 Using a **CDATA** section. (Part 2 of 2.)

Because a **CDATA** section is not parsed, it can contain almost any text, including characters normally reserved for XML syntax, such as **<** and **&**. However, **CDATA** sections cannot contain the string **]]>**, because this string is used to terminate a **CDATA** section. For example,

```

<![CDATA[
    The following characters cause an error: ]]>
]]>

```

is an error.

Line 25 is an example of a processing instruction (or *PI*). Processing instructions have no effect on a document if the application processing the document does not use them. The information contained in a PI is passed to the application that is consuming the XML document's data.

Processing instructions are delimited by **<?** and **?>** and consist of a *PI target* and a *PI value*. Almost any name may be used for a PI target, except the reserved word **xml** (in any mixture of case). In the current example, the PI target is named **button**, and the PI value is **cpp = "sample.cpp" ansi = "yes"**. This PI might be used by an application to create a button that, when clicked, displays the entire code listing for a file named

sample.cpp. The **ansi = "yes"** portion of the PI value might be used to indicate that the C++ code is standard C++ compliant.



Software Engineering Observation 2.1

Processing instructions provide a means for programmers to insert application-specific information into an XML document without affecting the document's portability.

2.6 XML Namespaces

Object-oriented programming languages, such as Java and standard C++, provide class libraries that group their identifiers (i.e., classes, methods, etc.) into packages and namespaces. These packages and namespaces prevent *naming collisions* between programmer-defined identifiers, third-party identifiers and class-library identifiers. For example, a class named **Transaction** might represent a monetary transaction between two large companies; however, a bank might use class **Transaction** to represent a monetary transaction with a Federal Reserve Bank. A naming collision would occur if these two classes were used in the same Java application without their fully qualified package names.

Like Java and standard C++, XML supports *namespaces*,⁴ which provide a means of uniquely identifying XML elements. Because document authors create their own markup languages or *vocabularies*, namespaces are needed to group a vocabulary's elements logically.

Elements are qualified with *namespace prefixes*, which identify the namespaces to which elements belong. For example,

```
<deitel:book>Java Web Services</deitel:book>
```

qualifies element **book** with the namespace prefix **deitel**. This indicates that element **book** is part of namespace **deitel**.



Common Programming Error 2.11

Creating a namespace prefix named **xml** in any mixture of case is an error.

The markup in Fig. 2.5 demonstrates the use of namespaces. This XML document contains two **file** elements that are differentiated by using namespaces.

```

1  <?xml version = "1.0" encoding = "UTF-8"?>
2
3  <!-- Fig. 2.5: namespace.xml -->
4  <!-- Demonstrating namespaces -->
5
6  <text:directory xmlns:text = "urn:deitel:textInfo"
7    xmlns:image = "urn:deitel:imageInfo">
8
9    <text:file filename = "book.xml">
10      <text:description>A book list</text:description>
11    </text:file>
12
```

Fig. 2.5 XML namespaces demonstration. (Part 1 of 2.)

4. XML Namespaces is a W3C Recommendation (www.w3.org/TR/REC-xml-names).

```

13     <image:file filename = "funny.jpg">
14         <image:description>A funny picture</image:description>
15         <image:size width = "200" height = "100" />
16     </image:file>
17
18 </text:directory>

```

```

c:\>java -classpath %CLASSPATH% com.deitel.jws1.xml.ParserTest
chapter2/fig02_5/namespace.xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- Fig. 2.5: namespace.xml -->
<!-- Demonstrating namespaces -->
<text:directory xmlns:image="urn:deitel:imageInfo" xmlns:text="urn:de-
itel:textInfo">

    <text:file filename="book.xml">
        <text:description>A book list</text:description>
    </text:file>

    <image:file filename="funny.jpg">
        <image:description>A funny picture</image:description>
        <image:size height="100" width="200"/>
    </image:file>

</text:directory>

```

Fig. 2.5 XML namespaces demonstration. (Part 2 of 2.)

Lines 6–7 use attribute **xmlns** to declare two namespace prefixes: **text** and **image**. Each namespace prefix is bound (or mapped) to a series of characters called a *uniform resource identifier (URI)* that uniquely identifies the namespace. Document authors create their own namespace prefixes and URIs.

To ensure that namespaces are unique, document authors must provide unique URIs. Here, we use the text **urn:deitel:textInfo** and **urn:deitel:imageInfo** as URIs. A common practice is to use *Universal Resource Locators (URLs)* for URIs, because the domain names (such as **www.deitel.com**) used in URLs are guaranteed to be unique. For example, lines 6–7 could have been written as

```

<text:directory xmlns:text =
    "http://www.deitel.com/xmlns-text"
    xmlns:image = "http://www.deitel.com/xmlns-image">

```

These URLs relate to the Deitel & Associates, Inc., domain name. The parser never visits these URLs—they simply represent a series of characters used to differentiate names. The URLs need not refer to actual Web pages or be formed properly.

Lines 9–11 qualify elements **file** and **description** with namespace prefix **text**. Lines 13–16 qualify elements **file**, **description** and **size** with namespace prefix **image**. Notice that namespace prefixes are applied to end tags as well.

To eliminate the need to qualify every element of a particular vocabulary with a namespace prefix, document authors can specify a *default namespace* (i.e., a namespace

that does not have a namespace prefix bound to it). Figure 2.6 demonstrates the creation and use of default namespaces.

Line 6 declares a default namespace, using attribute **xmlns** with a URI as its value. Once this default namespace is declared, child elements belonging to the namespace are not qualified by namespace prefixes. Element **file** (line 9–11) is in the namespace corresponding to the URI **urn:deitel:textInfo**. Compare this example with Fig. 2.5, where we qualified **file** and **description** with **text** (lines 9–11).

The default namespace applies to the **directory** element and all elements that are not qualified with namespace prefixes. However, namespace prefixes can be used to qualify other elements as belonging to different namespaces. For example, the **file** element in line 13 is qualified with **image** to indicate that it is in the namespace corresponding to the URI **urn:deitel:imageInfo**, rather than the default namespace.

```

1  <?xml version = "1.0" encoding = "UTF-8"?>
2
3  <!-- Fig. 2.6: defaultnamespace.xml -->
4  <!-- Using a default namespace      -->
5
6  <directory xmlns = "urn:deitel:textInfo"
7    xmlns:image = "urn:deitel:imageInfo">
8
9    <file filename = "book.xml">
10      <description>A book list</description>
11    </file>
12
13    <image:file filename = "funny.jpg">
14      <image:description>A funny picture</image:description>
15      <image:size width = "200" height = "100" />
16    </image:file>
17
18  </directory>

```

```

c:\>java -classpath %CLASSPATH% com.deitel.jws1.xml.ParserTest
chapter2/fig02_6/defaultnamespace.xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- Fig. 2.6: defaultnamespace.xml -->
<!-- Using a default namespaces      -->
<directory xmlns="urn:deitel:textInfo" xmlns:image="urn:deitel:image-
Info">

  <file filename="book.xml">
    <description>A book list</description>
  </file>

  <image:file filename="funny.jpg">
    <image:description>A funny picture</image:description>
    <image:size height="100" width="200"/>
  </image:file>

</directory>

```

Fig. 2.6 Default namespace demonstration.



Software Engineering Observation 2.2

Attributes need not be qualified with namespace prefixes, because they always are associated with elements. Attributes that are not qualified with a namespace prefix do not belong to any namespace (i.e., a default namespace or the element's namespace).

In this chapter, we have introduced XML. In Chapter 5, we introduce Document Type Definition files (DTDs), which describe an XML document's structure. DTDs allow programmers to confirm that the XML documents being processed are complete and use the correct markup.

2.7 Summary

XML is a technology for creating vocabularies to describe data of virtually any type in a structured manner. To process an XML document, a software program called an XML parser is required. The XML parser reads the XML document, checks its syntax, reports any errors and allows access to the document's data. An XML document is considered well formed if it is syntactically correct (i.e., if the parser did not report any errors due to missing tags, overlapping tags, etc.). Every XML document must be well formed. XML documents may contain carriage-return, line-feed and Unicode characters.

All XML start tags must have a corresponding end tag, and all start and end tags must be properly nested. Elements define a structure. An element may contain content (i.e., child elements and/or character data). An element may have zero, one or more attributes associated with it that describe the element.

CDATA sections allow the document author to include data that are not intended to be parsed and may contain text, reserved characters (e.g., `<`) and whitespace characters. Processing instructions allow document authors to embed application-specific data within an XML document.

Because document authors can create their own vocabularies, naming collisions (i.e., conflicts that arise when document authors use the same names for elements) can occur. Namespaces provide a means for document authors to prevent naming collisions.

2.8 Internet and World Wide Web Resources

www.w3.org/XML

Worldwide Web Consortium Extensible Markup Language home page. Contains links to related XML technologies, recommended books, a time line for publications, developer discussions, translations, software, etc.

www.w3.org/Addressing

Worldwide Web Consortium addressing home page. Contains information on URIs and links to other resources.

www.xml.com

This is one of the most popular XML sites on the Web. It has resources and links relating to all aspects of XML, including articles, news, seminar information, tools and Frequently Asked Questions (FAQs).

www.xml.org

"The XML Industry Portal" is another popular XML site that includes links to many different XML resources, such as news, FAQs and descriptions of XML-derived markup languages.

www.coverpages.org

Oasis XML Cover Pages home page is a comprehensive XML reference. The site includes links to news, articles, software and information about events.

html.about.com/compute/html/cs/xmlandjava/index.htm

This site contains articles about XML and Java and is updated regularly.

www.w3schools.com/xml

This site contains a tutorial that introduces the reader to XML. The tutorial contains many examples.

java.sun.com/xml/jaxp

Home page of Sun's JAXP and parser technology.

xml.apache.org

Home page of Apache Software Foundation. This site contains news, links to Apache projects, Foundation information and downloads.