Creating Markup with XML

Objectives

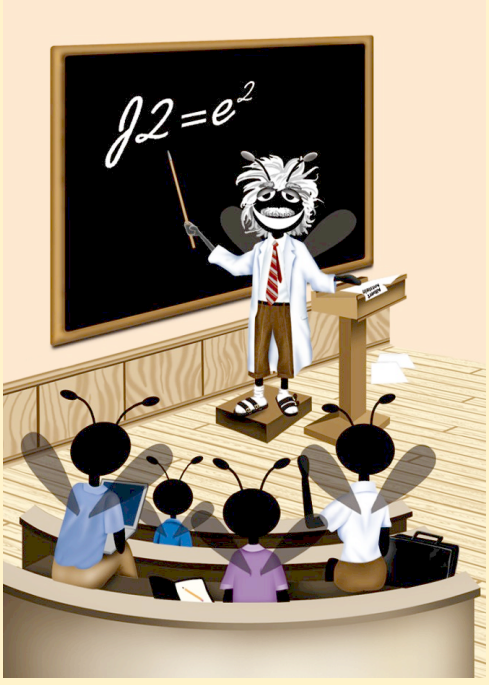
- To create custom markup using XML.
- To understand the concept of an XML parser.
- To use elements and attributes to mark up data.
- To understand the difference between markup text and character data.
- To understand the concept of a well-formed XML document.
- To understand the concept of an XML namespace.
- To be able to use **`CDATA`** sections and processing instructions.

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

A.1 Introduction
A.2 Introduction to XML Markup
A.3 Parsers and Well-Formed XML Documents
A.4 Characters
A.4.1 Characters vs. Markup
A.4.2 White Space, Entity References and Built-In Entities
A.5 CDATA Sections and Processing Instructions
A.6 XML Namespaces
A.7 Internet and World Wide Web Resources

Summary • Terminology • Self-Review Exercises • Answers to Self-Review Exercises

A.1 Introduction

The Extensible Markup Language (XML) is a technology for marking up structured data so that any software with an XML parser can understand and use its content. Data independence, the separation of content from its presentation, is the essential characteristic of XML. XML documents are simply text files that are marked up in a special way, so XML is intelligible to both humans and machines. Any application can conceivably process XML data. This makes XML ideal for data exchange.

Platform independence, the separation of an application from the platform on which it runs, is the essential characteristic of Java. With Java, software developers can write a program once, and it will run on any platform that has an implementation of the Java virtual machine. Java and XML have common goals. Java allows the portability of executable code across platforms. Likewise, XML allows the portability of structured data across applications. When used together, these technologies allow applications and their associated data to work on any computer. Recognizing this fact, software developers across the world are integrating XML into their Java applications to gain Web functionality and interoperability.

In this appendix, we show how to incorporate XML into Java applications. We use a Java application we created, named **`ParserTest`**, to output an XML document's contents. This application is included in the Appendix A examples directory on the CD-ROM that accompanies this book.

A.2 Introduction to XML Markup

In this section, we begin marking up data using XML. Consider a simple XML document (**`first.xml`**) that marks up a message (Fig. A.1). We output the entire XML document to the command line.

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

```
C:\>java -jar ParserTest.jar first.xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- Fig. A.1 : first.xml               -->
<!-- Simple introduction to XML markup -->
<myMessage id="T643070">
   <message>Welcome to XML!</message>
</myMessage>
```

Fig. A.1 Simple XML document containing a message .

The document begins with the optional *XML declaration* in line 1. This declaration identifies the document as an XML document. The ***version*** *information parameter* specifies the version of XML used in the document.

Portability Tip A.1

Although the XML declaration is optional, it should be used to identify the XML version to which the document conforms. 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 A.1

Placing anything, including whitespace (i.e., spaces, tabs and newlines), before an XML declaration is an error.

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. A.1 : first.xml
     Simple introduction to XML markup -->
```

Common Programming Error A.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 are **<myMessage>** and **<message>** (lines 6–7). End tags differ from start tags in that they contain a *forward slash* (**/**) character. Examples of end tags are **</message>** and **</myMessage>** in lines 7–8. XML documents can contain any number of tags.

Good Programming Practice A.1

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

Common Programming Error A.3

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

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. This practice allows document authors to create explicit relationships between data.

Common Programming Error A.4

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

Good Programming Practice A.2

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

Common Programming Error A.5

Attempting to create 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 A.6

XML element names are case sensitive. Using the wrong mixture of case is an error. For example, using the start tag **<message>** *and 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. A.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.

Common Programming Error A.7

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

Notice that the XML declaration output differs from the XML declaration in line 1. 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 A.4.

Portability Tip A.2

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

A.3 Parsers and Well-Formed XML Documents

A software program called an *XML parser* (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 A.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. A.1 is omitted, the error message shown in Fig. A.2 is generated by the parser.

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

```
C:\>java -jar ParserTest.jar error.xml
Exception in thread "main" org.xml.sax.SAXParseException: End of entity
not allowed; an end tag is missing.
   at org.apache.crimson.parser.Parser2.fatal(Parser2.java:3035)
   at org.apache.crimson.parser.Parser2.fatal(Parser2.java:3023)
   at org.apache.crimson.parser.Parser2.content(Parser2.java:1758)
   at org.apache.crimson.parser.Parser2.maybeElement(Parser2.ja-
va:1468)
   at org.apache.crimson.parser.Parser2.parseInternal(Parser2.ja-
va:499)
   at org.apache.crimson.parser.Parser2.parse(Parser2.java:304)
   at org.apache.crimson.parser.XMLReaderImpl.parse(XMLReaderImpl.ja-
va:433)

   at org.apache.crimson.jaxp.DocumentBuilderImpl.parse(DocumentBuild-
erImpl.java:179)
   at javax.xml.parsers.DocumentBuilder.parse(DocumentBuilder.ja-
va:161)
   at ParserTest.main(ParserTest.java:42)
```

Fig. A.2 XML document missing an end tag.

Most XML parsers can be downloaded at no charge. Several Independent Software Vendors (ISVs) have developed XML parsers, which can be found at **www.oasis-open.org/cover/xml.html#xmlparsers**. In this appendix, we will use the reference implementation for the Java API for XML Processing 1.1 (JAXP).

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 containing the XML document's data in memory. 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.

The examples we present use DOM-based parsing. In Appendix C, we provide a detailed discussion of the DOM. We do not discuss SAX-based parsing in these appendices.

A.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**), which encodes the vast majority of the world's commercially viable languages. We discuss Unicode in detail in Appendix I.

A.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*) is 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. A.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.

A.4.2 White Space, 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 characters in a document, including whitespace characters, to the application (e.g., a Java application) using the XML document.

Figure A.3 demonstrates that whitespace characters are passed by the parser to the application using the XML document. In this case, we simply print the data returned by the parser.

```
1  <?xml version = "1.0"?>
2
3  <!-- Fig. A.3 : whitespace.xml          -->
4  <!-- Demonstrating whitespace, entities -->
5  <!-- and empty elements                 -->
6
```

Fig. A.3 Whitespace characters in an XML document (part 1 of 2).

```
 7  <information>
 8
 9     <!-- empty element -->
10     <company name = "Deitel   &amp; Associates, Inc." />
11
12     <!-- start tag contains insignificant whitespace -->
13     <city      >    Sudbury      </city>
14
15
16     <state>Massachusetts</state>
17  </information>
```

```
C:\>java -jar ParserTest.jar whitespace.xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- Fig. A.3 : whitespace.xml          -->
<!-- Demonstrating whitespace, entities -->
<!-- and empty elements                 -->
<information>
   <!-- empty element -->
   <company name="Deitel   &amp; Associates, Inc."/>

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


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

Fig. A.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. Line 2 was considered insignificant by the application as well as the extra space characters in the start tag of line 13. In Appendix B, you will see that whitespace may or may not be significant, depending on the Document Type Definition (DTD) that an XML file uses. We will explore the subtleties of whitespace interpretation in greater detail in Appendix B.

XML element markup consists of a start tag, character data and an end tag. The element of line 10 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   &amp; 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. To use these symbols in character data or in attribute values, *entity references* must be used. Entity references begin with an ampersand (**&**) and end with a *semicolon* (**;**). XML provides entity references (or *built-in entities*) for the ampersand (**&**), left-angle bracket (**<**), right angle bracket (**>**), apostrophe (**'**) and quotation mark (**"**).

Common Programming Error A.8

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

Common Programming Error A.9

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

A.5 CDATA Sections and Processing Instructions

In this section, we discuss parts of an XML document, called ***CDATA** sections*, that can contain text, reserved characters (e.g., **<**) and whitespace characters. Character data in a **CDATA** section are not processed by the XML parser. A common use of a **CDATA** section is for programming code such as JavaScript and C++, which often include the characters **&** and **<**. Figure A.4 presents an XML document that compares text in a **CDATA** section with character data.

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"?>
2
3  <!-- Fig. A.4 : cdata.xml               -->
4  <!-- CDATA section containing C++ code -->
5
6  <book title = "C++ How to Program" edition = "3">
7
8     <sample>
9           // C++ comment
10          if ( this-&gt;getX() &lt; 5 &amp;&amp; value[ 0 ] != 3 )
11             cerr &lt;&lt; this-&gt;displayError();
12    </sample>
13
14    <sample>
15       <![CDATA[
16
17          // C++ comment
18          if ( this->getX() < 5 && value[ 0 ] != 3 )
19             cerr << this->displayError();
20       ]]>
21    </sample>
22
23    C++ How to Program by Deitel &amp; Deitel
24
25    <?button cpp = "sample.cpp" ansi = "yes"?>
26 </book>
```

Fig. A.4 Using a **CDATA** section (part 1 of 2).

```
C:\>java -jar ParserTest.jar cdata.xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- Fig. A.4 : cdata.xml                -->
<!-- CDATA section containing C++ code -->
<book title="C++ How to Program" edition="3">

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

   <sample>
      <![CDATA[

         // C++ comment
         if ( this->getX() < 5 && value[ 0 ] != 3 )
            cerr << this->displayError();
      ]]>
   </sample>

   C++ How to Program by Deitel &amp; Deitel

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

Fig. A.4 Using a **CDATA** section (part 2 of 2).

Common Programming Error A.10

Placing one or more spaces inside the opening ***<![CDATA[*** *or closing* ***]]>*** *is an error.*

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 text **]]>**, because this 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* (*PI*). Processing instructions provide a convenient syntax to allow document authors to embed application-specific data within an XML document. 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 using the XML document.

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**.

Software Engineering Observation A.1

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

A.6 XML Namespaces

Because XML allows document authors to create their own tags, *naming collisions* (i.e., conflicts between two different elements that have the same name) can occur. For example, we may use the element **book** to mark up data about one of our publications. A stamp collector may also create an element **book** to mark up data about a book of stamps. If both of these elements were used in the same document, there would be a naming collision, and it would be difficult to determine which kind of data each element contained. In this section, we discuss a method for preventing collisions called *namespaces*. In Appendix D, we begin using namespaces.

For example,

```
<subject>Math</subject>
```

and

```
<subject>Thrombosis</subject>
```

use a **subject** element to mark up a piece of data. However, in the first case the subject is something one studies in school, whereas in the second case the subject is in the field of medicine. These two **subject** elements can be differentiated using namespaces. For example

```
<school:subject>Math</school:subject>
```

and

```
<medical:subject>Thrombosis</medical:subject>
```

indicate two distinct **subject** elements. Both **school** and **medical** are *namespace prefixes*. Namespace prefixes are prepended to element and attribute names in order to specify the namespace in which the element or attribute can be found. Each namespace prefix is tied to a uniform resource identifier (URI) that uniquely identifies the namespace. Document authors can create their own namespace prefixes, as shown in Fig. A.5 (lines 6–7). Virtually any name may be used for a namespace, except the reserved namespace ***xml***.

```
1  <?xml version = "1.0"?>
2
3  <!-- Fig. A.5 : namespace.xml -->
4  <!-- Namespaces               -->
5
6  <text:directory xmlns:text = "urn:deitel:textInfo"
7     xmlns:image = "urn:deitel:imageInfo">
8
```

Fig. A.5 Demonstrating XML namespaces (part 1 of 2).

```
 9      <text:file filename = "book.xml">
10         <text:description>A book list</text:description>
11      </text: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   </text:directory>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Fig. A.5 : namespace.xml -->
<!-- Namespaces               -->
<text:directory xmlns:text="urn:deitel:textInfo" xmlns:image="urn:dei-
tel:imageInfo">

   <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 width="200" height="100"/>
   </image:file>

</text:directory>
```

Fig. A.5 Demonstrating XML namespaces (part 2 of 2).

In Fig. A.5, two distinct **file** elements are differentiated using namespaces. Lines 6–7 use the XML namespace keyword ***xmlns*** to create two namespace prefixes: **text** and **image**. The values assigned to attributes **xmlns:text** and **xmlns:image** are called *Uniform Resource Identifiers* (*URIs*). By definition, a URI is a series of characters used to differentiate names.

To ensure that a namespace is unique, the document author must provide a unique URI. 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 (e.g., **deitel.com**) used in URLs are guaranteed to be unique. For example, lines 6–7 could have been written as

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

where we use URLs related to the Deitel & Associates, Inc., domain name (**www.deitel.com**). These URLs are never visited by the parser—they only represent a series of characters for differentiating names and nothing more. The URLs need not even exist or be properly formed.

Lines 9–11 use the namespace prefix **text** to describe elements **file** and **description**. Notice that end tags have the namespace prefix **text** applied to them as well. Lines 13–16 apply namespace prefix **image** to elements **file**, **description** and **size**.

```
1  <?xml version = "1.0"?>
2
3  <!-- Fig. A.6 : defaultnamespace.xml -->
4  <!-- Using Default Namespaces        -->
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 -jar ParserTest.jar defaultnamespace.xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- Fig. A.6 : defaultnamespace.xml -->
<!-- Using 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 width="200" height="100"/>
   </image:file>

</directory>
```

Fig. A.6 Using default namespaces.

To eliminate the need to place a namespace prefix in each element, authors may specify a *default namespace* for an element and all of its child elements. Figure A.6 demonstrates the use of default namespaces.

We declare a default namespace using the **`xmlns`** attribute with a URI as its value (line 6). Once this default namespace is in place, child elements that are part of the namespace do not need a namespace prefix. Element **`file`** (line 9) is in the namespace corresponding to the URI **`urn:deitel:textInfo`**. Compare this usage with that in Fig. A.5, where we prefixed the **`file`** and **`description`** elements with the namespace prefix **`text`** (lines 9–11).

The default namespace applies to all elements contained in the **`directory`** element. However, we may use a namespace prefix to specify a different namespace for particular

elements. For example, the **file** element on line 13 uses the prefix **image** to indicate that the element is in the namespace corresponding to the URI **urn:deitel:imageInfo**.

A.7 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, Frequently Asked Questions (FAQs), etc.

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.oasis-open.org/cover
Oasis XML Cover Pages home page is a comprehensive reference for many aspects of XML and its related technologies. The site includes links to news, articles, software and 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
Contains a tutorial that introduces the reader to the major aspects of XML. The tutorial contains many examples.

java.sun.com/xml
Home page of the Sun's JAXP and parser technology.

SUMMARY

- XML is a technology for creating markup languages to describe data of virtually any type in a structured manner.
- XML allows document authors to describe data precisely by creating their own tags. Markup languages can be created using XML for describing almost anything.
- XML documents are commonly stored in text files that end in the extension **.xml**. Any text editor can be used to create an XML document. Many software packages allow data to be saved as XML documents.
- The XML declaration specifies the version to which the document conforms.
- All XML documents must have exactly one root element that contains all of the other elements.
- 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 contents.
- An XML document is considered well formed if it is syntactically correct (i.e., the parser did not report any errors due to missing tags, overlapping tags, etc.). Every XML document must be well formed.

- Parsers may or may not support the Document Object Model (DOM) and/or the Simple API for XML (SAX) for accessing a document's content programmatically by using languages such as Java, Python and C.
- XML documents may contain: carriage return, the line feed and Unicode characters. Unicode is a standard that was released by the Unicode Consortium in 1991 to expand character representation for most of the world's major languages. The American Standard Code for Information Interchange (ASCII) is a subset of Unicode.
- Markup text is enclosed in angle brackets (i.e., **<** and **>**). Character data are the text between a start tag and an end tag. Child elements are considered markup—not character data.
- Spaces, tabs, line feeds and carriage returns are whitespace characters. In an XML document, the parser considers whitespace characters to be either significant (i.e., preserved by the parser) or insignificant (i.e., not preserved by the parser).
- Almost any character may be used in an XML document. However, the characters ampersand (**&**) and left-angle bracket (**<**) are reserved in XML and may not be used in character data, except in **CDATA** sections. Angle brackets are reserved for delimiting markup tags. The ampersand is reserved for delimiting hexadecimal values that refer to a specific Unicode character. These expressions are terminated with a semicolon (**;**) and are called entity references. The apostrophe and double-quote characters are reserved for delimiting attribute values.
- XML provides built-in entities for ampersand (**&**), left-angle bracket (**<**), right-angle bracket (**>**), apostrophe (**'**) and quotation mark (**"**).
- All XML start tags must have a corresponding end tag and all start- and end tags must be properly nested. XML is case sensitive, therefore start tags and end tags must have matching capitalization.
- Elements define a structure. An element may or may not contain content (i.e., child elements or character data). Attributes describe elements. An element may have zero, one or more attributes associated with it. Attributes are nested within the element's start tag. Attribute values are enclosed in quotes—either single or double.
- XML element and attribute names can be of any length and may contain letters, digits, underscores, hyphens and periods; and they must begin with either a letter or an underscore.
- A processing instruction's (PI's) information is passed by the parser to the application using the XML document. Document authors may create their own processing instructions. Almost any name may be used for a PI target except the reserved word **xml** (in any mixture of case). Processing instructions allow document authors to embed application-specific data within an XML document. This data are not intended to be readable by humans, but readable by applications.
- **CDATA** sections may contain text, reserved characters (e.g., **<**), words and whitespace characters. XML parsers do not process the text in **CDATA** sections. **CDATA** sections allow the document author to include data that is not intended to be parsed. **CDATA** sections cannot contain the text **]]>**.
- Because document authors can create their own tags, naming collisions (e.g., 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. Document authors create their own namespaces. Virtually any name may be used for a namespace, except the reserved namespace **xml**.
- A Universal Resource Identifier (URI) is a series of characters used to differentiate names. URIs are used with namespaces.

TERMINOLOGY

<![CDATA[and **]]>** to delimit a **CDATA** section
<? and **?>** to delimit a processing instruction
ampersand (**&**)
angle brackets (**<** and **>**)
apostrophe (**'**)

application
ASCII (American Standard Code for Information Interchange)
attribute
built-in entity
CDATA section
character data
child
child element
comment
container element
content
element
empty element
end tag
entity references
insignificant whitespace character
Java API for XML Parsing (JAXP)
left angle bracket (**<**)
markup language
markup text
namespace
namespace prefix
namespace **xml**
naming collision
node
parser
PI target
PI value
processing instruction (PI)
quotation mark (**"**)
reserved character
reserved keyword
reserved namespace
right angle bracket (**>**)
root element
SAX-based parser
significant whitespace character
Simple API for XML (SAX)
start tag
structured data
tree structure of an XML document
Unicode
Unicode Consortium
Universal Resource Identifier (URI)
XML
XML declaration
XML document
XML namespace
XML parser
XML processor
XML version

SELF-REVIEW EXERCISES

A.1 State whether the following are *true* or *false*. If *false*, explain why.

a) XML is a technology for creating markup languages.
b) XML markup text is delimited by forward and backward slashes (**/** and ****).
c) All XML start tags must have corresponding end tags.
d) Parsers check an XML document's syntax and may support the Document Object Model and/or the Simple API for XML.
e) An XML document is considered well formed if it contains whitespace characters.
f) SAX-based parsers process XML documents and generate events when tags, text, comments, etc., are encountered.
g) When creating new XML tags, document authors must use the set of XML tags provided by the W3C.
h) The pound character (**#**), the dollar sign (**$**), the ampersand (**&**), the greater-than symbol (**>**) and the less-than symbol (**<**) are examples of XML reserved characters.
i) Any text file is automatically considered to be an XML document by a parser.

A.2 Fill in the blanks in each of the following statements:

a) A/An _____ processes an XML document.
b) Valid characters that can be used in an XML document are the carriage return, line feed and _____ characters.
c) An entity reference must be proceeded by a/an _____ character.
d) A/An _____ is delimited by **<?** and **?>**.
e) Text in a/an _____ section is not parsed.

f) An XML document is considered _____ if it is syntactically correct.
g) _____ help document authors prevent element-naming collisions.
h) A/An _____ tag does not contain character data.
i) The built-entity for the ampersand is _____.

A.3 Identify and correct the error(s) in each of the following:

a)
```
<my Tag>This is my custom markup<my Tag>
```
b)
```
<!PI value!>    <!-- a sample processing instruction -->
```
c)
```
<myXML>I know XML!!!</MyXML>
```
d)
```
<CDATA>This is a CDATA section.</CDATA>
```
e)
```
<xml>x < 5 && x > y</xml>  <!-- mark up a Java condition **>
```

ANSWERS TO SELF-REVIEW EXERCISES

A.4 a)True. b) False. In an XML document, markup text is any text delimited by angle brackets (**<** and **>**), with a forward slash being used in the end tag. c) True. d) True. e) False. An XML document is considered well formed if it is parsed successfully. f) True. g) False. When creating new tags, programmers may use any valid name except the reserved word **xml** (in any mixture of case). h) False. XML reserved characters include the ampersand (**&**) and the left angle bracket (**<**), but not the right-angle bracket (**>**), **#** and **$**. i) False. The text file must be parsable by an XML parser. If parsing fails, the document cannot be considered an XML document.

A.5 a) parser. b) Unicode. c) ampersand (**&**). d) processing instruction. e) **CDATA**. f) well formed. g) namespaces. h) empty. i) **&**.

A.6 a) Element name **my tag** contains a space. The forward slash, **/**, is missing in the end tag. The corrected markup is
```
<myTag>This is my custom markup</myTag>
```

b) Incorrect delimiters for a processing instruction. The corrected markup is
```
<?PI value?>    <!-- a sample processing instruction -->
```

c) Incorrect mixture of case in end tag. The corrected markup is
```
<myXML>I know XML!!!</myXML>
```
or
```
<MyXML>I know XML!!!</MyXML>
```

d) Incorrect syntax for a **CDATA** section. The corrected markup is
```
<![CDATA[This is a CDATA section.]]>
```

e) The name **xml** is reserved and cannot be used as an element. The characters **<**, **&** and **>** must be represented using entities. The closing comment delimiter should be two hyphens—not two stars. Corrected markup is
```
<someName>x &lt; 5 &amp;&amp; x &gt; y</someName>
<!-- mark up a Java condition -->
```