Chapter 3
XML Syntax and Parsing Concepts

In this chapter, we cover the rules of XML syntax that are stated or implied in the XML 1.0 Recommendation from the W3C. A considerable amount of XML terminology is introduced, including discussions of parsing, well-formedness, and validation. XML document structure, legal XML Names, and CDATA are also among the topics. The XML 1.0 specification also discusses rules for Document Type Definitions (DTDs), which we present in chapter 4. The material in chapters 3 and 4 is very interrelated.

Elements, Tags, Attributes, and Content

To understand XML syntax, we must first be familiar with several basic terms from HTML (and SGML) terminology. XML syntax, however, differs in some important ways from both HTML and SGML, as we’ll see.

Elements are the essence of document structure. They represent pieces of information and may or may not contain nested elements that represent even more specific information, attributes, and/or textual content. In our employee directory example from chapter 2 (Listing 2-2), some of the elements were Employees, Employee, Name, First, Last, Project, and PhoneNumbers.

Tags are the way elements are indicated or marked up in a document. For each element, there is typically a start tag that begins with < (less than) and ends with > (greater than), and an end tag that begins with </ and ends with >. Some of the start tags in our example were <Employees>, <Employee>, <Name>, and so forth. The corresponding end tags for these elements were </Employees>, </Employee>, and </Name>.

If an element has one or more attributes, they must appear between the < and > delimiters of the start tag. Attributes are qualifying pieces of information that add detail and further define an instance of an element. They are typically details that the language designer feels do not need to be nested elements themselves; the

1. With the exception of something called an empty element, as we will soon discuss.
assumption is that the attributes will generally be accessed less often than the elements that contain them, but this tends to be application dependent. In our employee example, the only element that had an attribute was *Employee*, and the attribute was *sex*, with two kinds of instances:

```xml
<Employee sex="male">
</Employee>
```

or

```xml
<Employee sex="female">
</Employee>
```

Each attribute has a **value**, the quoted text to the right of the equal sign. In the preceding examples, the values of the two instances of the sex attribute are “male” and “female”. Although in this case the value is a single word, values can be any amount of text, enclosed in single or double quotes. HTML permits attributes that do not require values (e.g., the `selected` attribute to denote a default choice in a form, as in `<OPTION selected>`), but this so-called **attribute minimization** is expressly not permitted in XML.

**Content** is whatever an element contains. Sometimes element content is simply text. In other cases, elements contain nested elements; the inner (child) elements are called the content of the outer (parent) element. Content is the data that the element contains. For example, in this fragment:

```xml
<Address>
  <Street>123 Milky Way</Street>
  <City>Columbia</City>
  <State>MD</State>
  <Zip>20777</Zip>
</Address>
```

“123 Milky Way” is the text content of the Street element, “Columbia” is the text content of the City element, and Street, City, State, and Zip are all nested element content of the parent Address element, in other words, “123 Milky Way Columbia MD 20777”. (The space preceding the last three words is due to newlines, as we’ll see.)

Notice that the content of Zip is the text string “20777”. Why do we not say that this is a number or, better yet, an example of some zip code datatype (constrained to either the valid five-digit or five-plus-four-digit ddddd-dddd values for zip codes)? Because there is nothing about the Zip element that conveys its content is numeric! We could, however, denote the element’s datatype explicitly by means of an attribute.

```xml
<Zip type="integer">20777</Zip>
```

2. This is a tremendous oversimplification. For more about this, see “Elements vs. Attributes: Guidelines,” in chapter 4.
We’ll eventually see how an alternative to DTDs called XML Schema makes data typing easier and far more flexible.

Another possibility, called mixed content, was illustrated in chapter 2 in the section “Document-Centric vs. Data-Centric,” in which both text and element content may appear as the content of a parent element. We’ll see how to handle this in chapter 4.

**XML Document Structure**

The XML Recommendation states that an XML document has both logical and physical structure. Physically, it is comprised of storage units called entities, each of which may refer to other entities, similar to the way that include works in the C language. Logically, an XML document consists of declarations, elements, comments, character references, and processing instructions, collectively known as the markup.

Although throughout this book we refer to an “XML document,” it is crucial to understand that XML may not exist as a physical file on disk. XML is sometimes used to convey messages between applications, such as from a Web server to a client. The XML content may be generated on the fly, for example by a Java application that accesses a database. It may be formed by combining pieces of several files, possibly mixed with output from a program. However, in all cases, the basic structure and syntax of XML is invariant.

An XML document consists of three parts, in the order given:

1. An XML declaration (which is technically optional, but recommended in most normal cases)
2. A document type declaration that refers to a DTD (which is optional, but required if you want validation)
3. A body or document instance (which is required)

Collectively, the XML declaration and the document type declaration are called the XML prolog.

**XML Declaration**

The XML declaration is a piece of markup (which may span multiple lines of a file) that identifies this as an XML document. The declaration also indicates whether the document can be validated by referring to an external Document Type Definition (DTD). DTDs are the subject of chapter 4; for now, just think of a DTD as a set of rules that describes the structure of an XML document.
The minimal XML declaration is:

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

*XML is case-sensitive* (more about this in the next subsection), so it’s important that you use lowercase for `xml` and `version`. The quotes around the value of the `version` attribute are required, as are the `?` characters. At the time of this writing, "1.0" is the only acceptable value for the `version` attribute, but this is certain to change when a subsequent version of the XML specification appears.

Do not include a space before the string `xml` or between the question mark and the angle brackets. The strings `<?xml` and `?>` must appear exactly as indicated. The space before the `?>` is optional. No blank lines or space may precede the XML declaration; adding white space here can produce strange error messages.

In most cases, this XML declaration is present. If so, it must be the *very first line* of the document and must not have leading white space. This declaration is technically optional; cases where it may be omitted include when combining XML storage units to create a larger, composite document.

Actually, the formal definition of an XML declaration, according to the XML 1.0 specification is as follows:

```
```

This Extended Backus-Naur Form (EBNF) notation, characteristic of many W3C specifications, means that an XML declaration consists of the literal sequence `<?xml`, followed by the required version information, followed by optional encoding and standalone declarations, followed by an optional amount of white space, and terminating with the literal sequence `?>`. In this notation, a question mark not contained in quotes means that the term that precedes it is optional.

The following declaration means that there is an external DTD on which this document depends. See the next subsection for the DTD that this negative standalone value implies.

```xml
<?xml version="1.0" standalone="no" ?>
```

On the other hand, if your XML document has no associated DTD, the correct XML declaration is:

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

The XML 1.0 Recommendation states: “If there are external markup declarations but there is no standalone document declaration, the value ‘no’ is assumed.”
The optional encoding part of the declaration tells the XML processor (parser) how to interpret the bytes based on a particular character set. The default encoding is UTF-8, which is one of seven character-encoding schemes used by the Unicode standard, also used as the default for Java. In UTF-8, one byte is used to represent the most common characters and three bytes are used for the less common special characters. UTF-8 is an efficient form of Unicode for ASCII-based documents. In fact, UTF-8 is a superset of ASCII.3

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

For Asian languages, however, an encoding of UTF-16 is more appropriate because two bytes are required for each character. It is also possible to specify an ISO character encoding, such as in the following example, which refers to ASCII plus Greek characters. Note, however, that some XML processors may not handle ISO character sets correctly since the specification requires only that they handle UTF-8 and UTF-16.

<?xml version="1.0" encoding="ISO-8859-7" ?>

Both the standalone and encoding information may be supplied:

<?xml version="1.0" standalone="no" encoding="UTF-8" ?>

Is the next example valid?

<?xml version="1.0" encoding='UTF-8' standalone='no'?>

Yes, it is. The order of attributes does not matter. Single and double quotes can be used interchangeably, provided they are of matching kind around any particular attribute value. (Although there is no good reason in this example to use double quotes for version and single quotes for the other, you may need to do so if the attribute value already contains the kind of quotes you prefer.) Finally, the lack of a blank space between 'no' and ?> is not a problem.

Neither of the following XML declarations is valid.

<?XML VERSION="1.0" STANDALONE="no"?>
<?xml version="1.0" standalone="No"?>

The first is invalid because these particular attribute names must be lowercase, as must "xml". The problem with the second declaration is that the value of the standalone attribute must be literally "yes" or "no", not "No". (Do I dare call this a "no No"?)

3. UTF stands for Unicode (or UCS) Transformation Format. UCS is Universal Character Set. Complete information about Unicode is available from http://www.unicode.org/.
Document Type Declaration

The document type declaration follows the XML declaration. The purpose of this declaration is to announce the root element (sometimes called the document element) and to provide the location of the DTD.\(^4\) The general syntax is:

```
<!DOCTYPE RootElement (SYSTEM | PUBLIC)
    ExternalDeclarations?  [InternalDeclarations]? >
```

where `<!DOCTYPE` is a literal string, `RootElement` is whatever you name the outermost element of your hierarchy, followed by either the literal keyword `SYSTEM` or `PUBLIC`. The optional `ExternalDeclarations` portion is typically the relative path or URL to the DTD that describes your document type. (It is really only optional if the entire DTD appears as an `InternalDeclarations`, which is neither likely nor desirable.) If there are `InternalDeclarations`, they must be enclosed in square brackets. In general, you’ll encounter far more cases with `ExternalDeclarations` than `InternalDeclarations`, so let’s ignore the latter for now. They constitute the *internal subset*, which is described in chapter 4.

Let’s start with a simple but common case. In this example, we are indicating that the DTD and the XML document reside in the same directory (i.e., the `ExternalDeclarations` are contained in the file `employees.dtd`) and that the root element is `Employees`:

```
<!DOCTYPE Employees SYSTEM "employees.dtd">
```

Similarly,

```
<!DOCTYPE PriceList SYSTEM "prices.dtd">
```

indicates a root element `PriceList` and the DTD is in the local file: `prices.dtd`.

In the next example, we use normal directory path syntax to indicate a different location for the DTD.

```
<!DOCTYPE Employees SYSTEM "./dtds/employees.dtd">
```

As is often the case, we might want to specify a URL for the DTD since the XML file may not even be on the same host as the DTD. This case also applies when you are using an XML document for message passing or data transmission across servers and still want the validation by referencing a common DTD.

```
<!DOCTYPE Employees SYSTEM "http://somewhere.com/dtds/employees.dtd">
```

4. A Document Type Definition is a set of rules that describe the hierarchical structure of any XML document instance based on that particular DTD. These rules are used to determine whether the document is valid. DTDs are discussed in detail in chapter 4.
Next, we have the case of the PUBLIC identifier. This is used in formal environments to declare that a given DTD is available to the public for shared use. Recall that XML’s true power as a syntax relates to developing languages that permit exchange of structured data between applications and across company boundaries. The syntax is a little different:

```xml
<!DOCTYPE RootElement PUBLIC PublicID URI>
```

The new aspect here is the notion of a PublicID, which is a slightly involved formatted string that identifies the source of the DTD whose path follows as the URI. This is sometimes known as the **Formal Public Identifier (FPI)**.

For example, I was part of a team that developed (Astronomical) Instrument Markup Language (AIML, IML) for NASA Goddard Space Flight Center.⁵ We wanted our DTD to be available to other astronomers. Our document type declaration (with a root element named Instrument) was:

```xml
<!DOCTYPE Instrument PUBLIC
"-//NASA//Instrument Markup Language 0.2//EN"
"http://pioneer.gsfc.nasa.gov/public/iml/iml.dtd">
```

In this case the PublicID is:

```
"-//NASA//Instrument Markup Language 0.2//EN"
```

The URI that locates the DTD is:

```
http://pioneer.gsfc.nasa.gov/public/iml/iml.dtd
```

Let’s decompose the PublicID. The leading hyphen indicates that NASA is not a standards body. If it were, a plus sign would replace the hyphen, except if the standards body were ISO, in which case the string “ISO” would appear. Next we have the name of the organization responsible for the DTD (NASA, in this case), surrounded with double slashes, then a short free-text description of the DTD (“Instrument Markup Language 0.2”), double slashes, and a two-character language identifier (“EN” for English, in this case).

Since the XML prolog is the combination of the XML declaration and the document type declaration, for our NASA example the complete prolog is:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE Instrument PUBLIC
"-//NASA//Instrument Markup Language 0.2//EN"
"http://pioneer.gsfc.nasa.gov/public/iml/iml.dtd">
```

⁵. Thanks to NASA and Commerce One project participants, Julie Breed, Troy Ames, Carl Hostetter, Rick Shafer, Dave Fout, Lisa Koons, Craig Warsaw, Melissa Hess, Ken Wootton, Steve Clark, Randy Wilke, and Lynne Case, among others.
As another example, let’s consider a common case involving DTDs from the W3C, such as those for XHTML 1.0.

```xml
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

W3C is identified as the organization, “DTD XHTML 1.0 Transitional” is the name of the DTD; it is in English; and the actual DTD is located by the URI `http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd`. Similarly, the prolog for XHTML Basic 1.0 is:

```xml
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML Basic 1.0//EN"
"http://www.w3.org/TR/xhtml-basic/xhtml-basic10.dtd">
```

The XHTML Basic 1.0 PublicID is similar but not identical to the XHTML 1.0 case and of course the DTD is different since it’s a different language.

If you noticed that the NASA example uses uppercase for the encoding value UTF-8 and the W3C examples use lowercase, you may have been bothered because that is inconsistent with what we learned about the case-sensitive value for the standalone attribute. The only explanation I can offer is that although element and attribute names are always case-sensitive, attributes values may or may not be. A reasonable guess is that if the possible attribute values are easily enumerated (i.e., “yes” or “no”, or other relatively short lists of choices), then case probably matters.

**NOTE**

DTD-related keywords such as `DOCTYPE`, `PUBLIC`, and `SYSTEM` must be uppercase. XML-related attribute names such as `version`, `encoding`, and `standalone` must be lowercase.

**Document Body**

The **document body**, or instance, is the bulk of the information content of the document. Whereas across multiple instances of a document of a given type (as identified by the `DOCTYPE`) the XML prolog will remain constant, the document body changes with each document instance (in general). This is because the prolog defines (either directly or indirectly) the overall structure while the body contains the real instance-specific data. Comparing this to data structures in computer languages, the DTD referenced in the prolog is analogous to a `struct` in the C language or a `class` definition in Java, and the document body is analogous to a runtime instance of the `struct` or `class`. 
Because the document type declaration specifies the root element, this must be the first element the parser encounters. If any other element but the one identified by the DOCTYPE line appears first, the document is immediately invalid.

Listing 3-1 shows a very simple XHTML 1.0 document. The DOCTYPE is “html” (not “xhtml”), so the document body begins with <html> and ends with </html>.

Listing 3-1  Simple XHTML 1.0 Document with XML Prolog and Document Body

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html
   PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <title>XHTML 1.0</title>
  </head>
  <body>
    <h1>Simple XHTML 1.0 Example</h1>
    <p>See the <a href="http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">DTD</a>.</p>
  </body>
</html>
```

**Markup, Character Data, andParsing**

An XML document contains text characters that fall into two categories: either they are part of the document markup or part of the data content, usually called **character data**, which simply means all text that is not part of the markup. In other words, XML text consists of intermingled character data and markup. Let’s revisit an earlier fragment.

```
<Address>
  <Street>123 Milky Way</Street>
  <City>Columbia</City>
  <State>MD</State>
  <Zip>20777</Zip>
</Address>
```

The character data comprises the four strings “123 Milky Way”, “Columbia”, “MD”, and “20777”; the markup comprises the start and end tags for the five elements Address, Street, City, State, and Zip. Note that this is similar but not identical, to what we previously called content. For example, although each chunk of character data is the content of a particular element, the content of the Address element is all of the child elements. We can think of all the character data belonging to both the element that directly contains it and indirectly to Address. (In fact, in some
XML applications such as XSLT, if we ask for the text content of Address, we’ll get the concatenation of all the individual strings."

The markup itself can be divided into a number of categories, as per section 2.4 of the XML 1.0 specification:

- start tags and end tags (e.g., <Address> and </Address>)
- empty-element tags (e.g., <Divider/>)
- entity references (e.g., &footer; or %otherDTD;)
- character references (e.g., &lt; or &gt;)
- comments (e.g., <!-- whatever -->)
- CDATA section delimiters (e.g., <![CDATA[ insert code here ]]>)
- document type declarations (e.g., <!DOCTYPE ...
- processing instructions (e.g., <?myJavaApp numEmployees="25"
  location="Columbia" ...
- XML declarations (e.g., <?xml version=.... ?>)
- text declarations (e.g., <?xml encoding=.... ?>)
- any white space at the top level (before or after the root element)

We will discuss each of these markup aspects in either this chapter or the next. Note that for all types of markup, there are some delimiters, most but not all of which involve angle brackets.

The specification states that all text that is not markup constitutes the character data of the document. In other words, if you stripped all markup from the document, the remaining content would be the character data. Consider this example:

```xml
<?xml version="1.0" standalone="no" ?>
<!DOCTYPE Message SYSTEM "message.dtd">
<Message mime-type="text/plain">
<!-- This is a trivial example. -->
  <From>The Kenster</From>
  <To>Silly Little Cowgirl</To>
  <Body>
    Hi, there. How is your gardening going?
  </Body>
</Message>
```

The character data when the markup is removed would be:

The Kenster Silly Little Cowgirl Hi, there. How is your gardening going?

In general this is essentially the text between the start and end tags, which we previously called the content of the element, but there is a subtlety related to parsing. Depending on parser details, the newlines after <From> and </To> might be replaced by single spaces, as shown. Alternatively, the newlines might be preserved.

Parsing is the process of splitting up a stream of information into its constituent pieces (often called tokens). In the context of XML, parsing refers to scanning an
XML Syntax Rules

XML document (which need not be a physical file—it can be a data stream) in order to split it into its various markup and character data, and more specifically, into elements and their attributes. XML parsing reveals the structure of the information since the nesting of elements implies a hierarchy. It is possible for an XML document to fail to parse completely if it does not follow the well-formedness rules described in the XML 1.0 Recommendation. A successfully parsed XML document may be either well-formed (at a minimum) or valid, as discussed in detail later in this chapter and the next.

There is a subtlety about processing character data. During the parsing process, if there is markup that contains entity references, the markup will be converted into character data. A typical example from XHTML would be:

```xml
<p>"AT&T is a winning company," said.</p>
```

After the parser substitutes for the entities, the resultant character data is:

"AT&T is a winning company," he said.

After parsing and substituting for special characters, the character data that remains after the substitution is parsed character data, which is referred to as #PCDATA in DTDs and always refers to textual content of elements. Character data that is not parsed is called CDATA in DTDs; this relates exclusively to attribute values.

XML Syntax Rules

In this section, we explain the various syntactical rules of XML. Documents that follow these rules are called well-formed, but not necessarily valid, as we’ll see. If your document breaks any of these rules, it will be rejected by most, if not all, XML parsers.

Well-Formedness

The minimal requirement for an XML document is that it be well-formed, meaning that it adheres to a small number of syntax rules, which are summarized in Table 3-1 and explained in the following sections. However, a document can abide by all these rules and still be invalid. To be valid, a document must both be well-formed and adhere to the constraints imposed by a DTD or XML Schema.

6. See the well-formedness discussion in the XML 1.0 Recommendation, http://www.w3.org/TR/REC-xml#sec-well-formed.
TABLE 3-1 XML Syntax Rules (Well-Formedness Constraints)

- The document must have a consistent, well-defined structure.
- All attribute values must be quoted (single or double quotes).
- White space in content, including line breaks, is significant by default.
- All start tags must have corresponding end tags (exception: empty elements).
- The root element must contain all others, which must nest properly by start/end tag pairing.
- Elements must not overlap; they may be nested, however. (This is also technically true for HTML. Browsers ignore overlapping in HTML, but not in XML.)
- Each element except the root element must have exactly one parent element that contains it.
- Element and attribute names are case-sensitive: Price and PRICE are different elements.
- Keywords such as DOCTYPE and ENTITY must always appear in uppercase; similarly for other DTD keywords such as ELEMENT and ATTLIST.
- Tags without content are called empty elements and must end in "'/s".

Legal XML Name Characters

An XML Name (sometimes called simply a Name) is a token that

- begins with a letter, underscore, or colon (but not other punctuation)
- continues with letters, digits, hyphens, underscores, colons, or full stops [periods], known as name characters.

Names beginning with the string "xml", or any string which would match (('X'|'x')('M'|'m')('L'|'l'))are reserved.

Element and attribute names must be valid XML Names. (Attribute values need not be.) An NMTOKEN (name token) is any mixture of name characters (letters, digits, hyphens, underscores, colons, and periods).

NOTE

The Namespaces in XML Recommendation assigns a meaning to names that contain colon characters. Therefore, authors should not use the colon in XML names except for namespace purposes (e.g., xsl:template).

Listing 3-2 illustrates a number of legal XML Names, followed by three that should be avoided but may or may not be identified as illegal, depending on the XML parser you use, and four that are definitely illegal. (This is file name-tests.xml on the CD; you can try this with your favorite parser, or with one of the ones provided on the CD.)
XML Syntax Rules

Listing 3-2 Legal, Illegal, and Questionable XML Names

```xml
<?xml version = "1.0" standalone = "yes" encoding = "UTF-8"?>
<Test>
<!-- legal -->
<price />
<Price />
<pRice />
<_price />
<subtotal07 />
<discounted-price />
<discounted_price />
<discountedPrice />
<DiscountedPrice />
<kbs:DiscountedPrice />
<xlink:role />
<xsl:apply-templates />

<!-- discouraged -->
<xml-price />
<xml:price />
<discounted:price />

<!-- illegal -->
<7price />
<-price />
<.price />
<discounted price />
</Test>
```

From the legal examples, we see that any mixture of uppercase and lowercase is fine, as are numbers, and the punctuation characters that were in the definition. Since the last three examples in the first group use a colon, they are assumed to be elements in the namespaces identified by the prefixes “kbs”, “xlink”, and “xsl”. Of these, the last two refer to W3C-specified namespaces; xlink:role is an attribute defined by the XLink specification and xsl:apply-templates is an element defined by the XSLT specification. The “kbs” prefix refers to a hypothetical namespace, which I could have declared (but didn’t), since namespaces do not come only from the W3C. (See chapter 5 for a thorough discussion of namespaces.)

The three debatable examples are xml-price, xml:price, and discounted: price. The first two use the reserved letters “xml”; you shouldn’t use them, but most parsers won’t reject them. The discounted:price example uses a colon, which is frowned upon if “discounted” is not meant to be a prefix associated with a declared namespace.

The four illegal cases are much more clear. The first three, 7price, -price, and .price, are illegal because the initial character is not a letter, underscore, or colon. The fourth example is illegal because a space character cannot occur in an XML Name. Most parsers will think this is supposed to be the element named discounted and the attribute named price, minus a required equal sign and value.
Elements and Attributes Are Case-Sensitive

Unlike HTML, which is case insensitive (as is the SGML metalanguage of which HTML is an application), XML is strictly case-sensitive, and so therefore is every application of XML (e.g., XSLT, MathML, SVG and so forth, plus any languages you create). Therefore, the following elements are all unique and are in no way related to one another in XML:

```
price
Price
PRICE
```

The case sensitivity nature of XML often confuses novices. Be sure to remember this when doing string comparisons in code.

The W3C’s Extensible HyperText Markup Language (XHTML) recasts HTML in XML syntax. In XHTML, all elements and attributes have lowercase names, such as:

```
body
h1
img
href
```

Notice that this is not merely a convention; it is an absolute requirement. An XHTML document that contains capital letters in element or attribute names is simply invalid, even though uppercase or mixed-case names such as BODY, Body, or even b0dy would be perfectly acceptable in HTML.

Uppercase Keywords

Since XML is case-sensitive, it should not be surprising that certain special words must appear in a particular case. In general, the keywords that relate to DTDs (e.g., DOCTYPE, ENTITY, CDATA, ELEMENT, ATTLIST, PCDATA, IMPLIED, REQUIRED, and FIXED) must be all uppercase. On the other hand, the various strings used in the XML declaration (e.g., xml, version, standalone, and encoding) must appear in all lowercase.

Case Conventions or Guidelines

When creating your own XML vocabulary, it would be desirable if there were conventions to explain the use of uppercase, lowercase, mixed case, underscores, and hyphens. Unfortunately, no such conventions exist in XML 1.0. It is a good idea to
adopt your own conventions and to apply them consistently, at least across your project, but ideally throughout your entire organization.

For example, for element names I prefer using what is often called CamelCase because the initial letter of each word in a multiword name is uppercase and all others are lowercase, creating humps like a camel’s back. (It’s also sometimes called TitleCase because it resembles the title of a book.) For example:

```xml
<DiscountPrice rate="20%" countryCode="US"/>
```

Note that for attributes, I also use CamelCase, except the first word is always begun with a lowercase letter, as in “countryCode”. In fact, the terms UpperCamelCase (as I use for elements) and lowerCamelCase (as I use for attributes) are often used to make this distinction more clear. One reason that I favor this convention is that in any context (including documentation), it’s easy to distinguish elements from attributes.

It would be just as reasonable, however, to use all uppercase letters for elements, all lowercase for attributes, and a hyphen to separate multipart terms as in the following examples, or even to use all uppercase for elements and attributes.

```xml
<DISCOUNT-PRICE rate="20%" country-code="US"/>
```

As stated earlier, for XHTML, the W3C elected to use all lowercase letters. The most important thing is to pick a convention for your project (or your company) and to be consistent across developers and applications.

We’ve seen UpperCamelCase for elements and lowerCamelCase for attributes in the employee example: Employee with its sex attribute, Address, PhoneNumbers, and so on. The following fragment from the W3C’s SOAP 1.2 Part 2 Adjuncts Working Draft (http://www.w3.org/TR/2001/WD-soap12-part2-20011002/#N4008D) illustrates its use of UpperCamelCase for element names and lowerCamelCase for attributes, as well as for namespace prefixes.

```xml
<env:Body>
  <m:GetLastTradePrice
    env:encodingStyle="http://www.w3.org/2001/09/soap-encoding"
    xmlns:m="http://example.org/2001/06/quotes">
    <m:Symbol>DEF</m:Symbol>
  </m:GetLastTradePrice>
</env:Body>
```

**Root Element Contains All Others**

There must be one root element, also known as the document element, which is the parent of all other elements. That is, all elements are nested within the root element. All descendants of the root, whether immediate children or not, represent the content of the root. Recall that the name of the root element is given in the
DOCTYPE line if a DTD is referenced (either an external or internal one). We also noted that this document element must be the first element the parser encounters (after the XML prolog, which does not contain elements).

A somewhat surprising aspect, at least to this author, is that the XML Recommendation does not preclude a recursive root! In other words, it is possible for a root element to be defined in a DTD as containing itself. Although this is not common, it is worth noting. For example, in NASA’s IML DTD, we allowed that the root element Instrument could contain other Instrument children. (The DTD syntax shown here is formally described in chapter 4.)

<!ELEMENT Instrument (Instrument | Port | CommandProcedureSet)*>

Start and End Tags Must Match

Every start tag must have a corresponding end tag to properly delimit the content of the element the tags represent. The start and end tags are indicated exactly as they are in HTML, with < denoting the beginning of a start tag and <*> indicating the beginning of the end tag. The end delimiter of each tag is >.

<?ElementName>content</ElementName>

Empty Elements

An exception to the rule about start and end tags is the case in which an element has no content. Such empty elements convey information simply by their presence or possibly by their attributes, if any. Examples from XHTML 1.0 include:

<br />
<hr />
<img src="someImage.gif" width="100" height="200" alt="Some Image" />

An empty element begins like a start tag but terminates with the sequence/>. Optional white space may be used before the two terminating characters. This author prefers to include a space to emphasize empty elements. The space before /> is necessary for XHTML 1.0 to be handled correctly by older browser versions. Of course, it’s also possible to specify an empty element by using regular start and end tags, and this is syntactically identical (from the parser’s viewpoint) to the use of empty-element notation.

<img src="someImage.gif" width="100" height="200" alt="Some Image"></img>

Note that just like in HTML (or more appropriately, XHTML), an empty element is often used as a separator, such as <br/> and <hr />, or to indicate by its presence a particular piece of data, or to convey metadata by its attributes. If the term empty element seems strange to you when attributes are involved, just think in terms of the content of the element. There is no content, even when there are attributes, which is why it’s called empty.
Proper Nesting of Start and End Tags

No overlapping of start and end tags from different elements is permitted. Although this might seem like an obvious requirement, HTML as implemented by major browsers is considerably more forgiving and recovers from improper tag overlap. Correct nesting looks like this:

```xml
<OuterElement>
  <InnerElement>inner content</InnerElement>
</OuterElement>
```

An example of improper nesting is:

```xml
<OuterElement>
  <InnerElement>inner content</OuterElement>
</InnerElement>
```

Believe it or not, most browsers recover from this type of error in HTML, but they cannot and will not in XML or any language based on XML syntax. The improper nesting example results in either one or two fatal errors, with a message similar to this (depending on the parser):

```
Fatal error: end tag '</OuterElement>' does not match start tag. Expected '</InnerElement>'
Fatal error: end tag '</InnerElement>' does not match start tag. Expected '</OuterElement>'
```

Parent, Child, Ancestor, Descendant

The notion of the root element and the proper nesting rules leads us to some conclusions and terminology about the hierarchy of elements that are invariant across all XML documents. The terms **ancestor** and **descendant** are not used in the XML 1.0 Recommendation, but they certainly are in the DOM, XSLT, XPath, and so on, which is why they are introduced here:

- An element is a **child** of exactly one **parent**, which is the element that contains it.
- A parent may have more than one child.
- Immediate children and also children of a child are **descendants** of the parent.
- An element is an **ancestor** of all its descendants.
- The root is the ancestor of all elements.
- Every element is a descendant of the root.
- Every element has exactly one parent, except the root, which has no parent.

Attribute Values Must Be Quoted

In HTML (but not in XHTML), we are permitted to be inconsistent in the use of quotation marks to delimit the values of attributes. Generally, single-word values
do not require quotes in HTML. For example, both of these are acceptable and equivalent in HTML:

```html
<IMG SRC=someImage.gif>
<IMG SRC="someImage.gif">
```

In XML (and in XHTML), however, we are not allowed to be so cavalier about quotes. All attribute values must be quoted, even if there are no embedded spaces.

```xml
<img src='someImage.gif' />
<img src='someImage.gif' width='34' height='17'/>
```

Notice that either single or double quotes may be used to delimit the attribute values. Of course, if the attribute value contains double quotes, then you must use single quotes as the delimiter, and vice versa.

```xml
<Book title="Tudor's Guide to Paris" />
<Object width='5.3'' height='7.1'' />
```

White Space Is Significant

**White space** consists of one or more space characters, tabs, carriage returns, line feeds (denoted as `#x20`, `#x9`, `#xD`, and `#xA`, respectively). In the XML 1.0 Recommendation, white space is symbolized in production rules by a capital “S”, with the following definition (See [http://www.w3.org/TR/REC-xml#sec-common-syn](http://www.w3.org/TR/REC-xml#sec-common-syn) and [http://www.w3.org/TR/REC-xml#sec-white-space](http://www.w3.org/TR/REC-xml#sec-white-space)):

```
S ::= (#x20 | #x9 | #xD | #xA)+
```

In contrast to HTML, in which a sequence of white space characters is collapsed into a single white space and in which newlines are ignored, in XML all white space is taken literally. This means that the following two examples are not equivalent:

```xml
<Publication>
  <Published>1992</Published>
  <Publisher>Harmony Books</Publisher>
</Publication>

<Publication>
  <Published>1992</Published>
  <Publisher>Harmony Books</Publisher>
</Publication>
```

By default, XML parsers handle the `Publisher` element differently since in the second example, the string “Harmony Books” contains a newline between the two words. The application that invokes the parser can either consider the white space
Comments

Comments in XML are just like they are in HTML. They begin with the character sequence `<!--` and end with the sequence `-->`. The parser ignores what appears between them, except to verify that the comment is well-formed.

```xml
<Publication>
  <Published>1992</Published>
  <!-- This appears to be the second edition. -->
  <Publisher>Harmony Books</Publisher>
</Publication>
```

In XML, however, there are several restrictions regarding comments:

- Comments cannot contain the double hyphen combination “--” anywhere except as part of the comment’s start and end tags. Thus, this comment is illegal: `<!-- illegal comment -->`
- Comments cannot be nested. This means you need to take care when commenting out a section that already contains comments.
- Comments cannot precede the XML declaration because that part of the prolog must be the very first line in the document.
- Comments are not permitted in a start or end tag. They can appear only between tags (as if they were content) or surrounding tags.
- Comments may be used to cause the parser to ignore blocks of elements, provided that the result, once the commented-out block is effectively removed by the parser, is still well-formed XML.
- Parsers are not required to make comments available to the application, so don’t use them to pass data to an application; use Processing Instructions, discussed next.
- Comments are also permitted in the DTD, as discussed in chapter 4.

Processing Instructions

Processing instructions (often abbreviated as PI) are directives intended for an application other than the XML parser. Unlike comments, parsers are required to pass processing instructions on to the application. The general syntax for a PI is:

```xml
<?targetApplication applicationData ?>
```

Where `targetApplication` is the name (any XML Name) of the application that should receive the instruction, and `applicationData` is any arbitrary string that doesn’t contain the end delimiter. Often `applicationData` consists of name/value pairs that resemble attributes with values, but there is no requirement concerning the format. Aside from the delimiters "<?" and "?>", which must appear exactly
as shown, the only restriction is that there can be no space between the initial question mark and the target. Some examples follow.

<?xml-stylesheet type="text/xsl" href="foo.xsl" ?>
<?MortgageRateHandler rate="7%" period="30 years" ?>
<?javaApp class="MortgageRateHandler" ?>
<?javaApp This is the data for the MortgageRateHandler, folks! ?>
<?acroread file="mortgageRates.pdf" ?>

Processing instructions are not part of the actual structure of the document, so they may appear almost anywhere, except before the XML declaration or in a CDATA section. The parser’s responsibility is merely to pass the PI and its data on to the application. Since the same XML document could be processed by multiple applications, it is entirely possible that some applications will ignore a given PI and just pass it down the chain. In that case, the processing instruction will be acted upon only by the application for which it is intended (has meaning).

Although an XML declaration looks like a processing instruction because it is wrapped in the delimiters "<?" and "?>", it is not considered a PI. It is simply an XML declaration, the one-of-a-kind markup that may or may not be the first line of the document.

The target portion of the processing instruction can be a *notation* (defined in chapter 4). For example:

```xml
<!NOTATION AcrobatReader SYSTEM "/usr/local/bin/acroread">
```

The corresponding PI would be:

```xml
<?AcrobatReader file="Readme.pdf" size="75%" ?>
```

**Entity References**

**Entity references** are markup that the parser replaces with character data. In HTML, there are hundreds of predefined character entities, including the Greek alphabet, math symbols, and the copyright symbol. There are only five predefined entity references in XML, however, as shown in Table 3-2.

**TABLE 3-2 Predefined Entity References**

<table>
<thead>
<tr>
<th>Character</th>
<th>Entity Reference</th>
<th>Decimal Representation</th>
<th>Hexidecimal Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;</code></td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td><code>&amp;</code></td>
<td>&amp;</td>
<td>&amp;</td>
<td>&amp;</td>
</tr>
<tr>
<td>`'&quot;'</td>
<td>&quot;</td>
<td>&quot;</td>
<td>&quot;</td>
</tr>
<tr>
<td><code>'</code></td>
<td>'</td>
<td>'</td>
<td>'</td>
</tr>
</tbody>
</table>
We’ve already seen how entity references can be used as content. They can also appear within attribute values. According to Table 3-2,

```xml
<CD title="Brooks &amp; Dunn’s Greatest Hits" />
```

is equivalent to the decimal representation:

```xml
<CD title="Brooks &amp;#38; Dunn’s Greatest Hits" />
```

and to the hexadecimal representation:

```xml
<CD title="Brooks &amp;#x26; Dunn’s Greatest Hits" />
```

However, the next line is illegal because ampersand (“&”) must be escaped by using either the entity reference or one of its numeric representations:

```xml
<CD title="Brooks & Dunn’s Greatest Hits" />
```

This is because ampersand and less-than are special cases.

---

You are required to use the predefined entities &lt; and &amp; to escape the characters < and & in all cases other than when these characters are used as markup delimiters, or in a comment, a processing instruction, or a CDATA section. In other words, the literal < and & characters can appear only as markup delimiters, or within a comment, a processing instruction, or a CDATA section.

---

Listing 3-3 illustrates the use of all five predefined character entities, several decimal representations of Greek letters, and the three legal variations of the Brooks & Dunn example. If we run this through an XML parser, we can verify that it is well-formed; we did not use the literal ampersand or the literal less-than

---

**Listing 3-3** Examples of Predefined Entities and Greek Letters (predefined-entities.xml)

```xml
<?xml version="1.0" standalone="yes"?>
<Predefined>
  <Test>The hot tip from today’s &lt;StockWatch&gt; column is: &quot;AT&amp;T stock is doing better than Ralph Spoilsports Motors’ stock.&quot;</Test>
  <PS>Now, wasn’t that as easy as &gt;928;? Or &gt;945;’, &gt;946; , &gt;947;?</PS>
  <CD title="Brooks &amp; Dunn’s Greatest Hits"/>
  <CD title="Brooks &amp;#38; Dunn’s Greatest Hits"/>
  <CD title="Brooks &amp;#x26; Dunn’s Greatest Hits"/>
</Predefined>
```
before the word StockWatch. Figure 3-1 shows how this example looks in Internet Explorer, which renders the characters that are represented by the entities. It also confirms that the three Brooks & Dunn variations are equivalent.

HTML (and therefore XHTML) includes three large sets of predefined entities: Latin1, Special, and Symbols. You can pull these definitions into your XML document using external entities, covered in chapter 4. The files containing the entities are:

http://www.w3.org/TR/xhtml1/DTD/xhtml-lat1.ent
http://www.w3.org/TR/xhtml1/DTD/xhtml-special.ent
http://www.w3.org/TR/xhtml1/DTD/xhtml-symbol.ent

**CDATA Sections**

Sometimes it is necessary to indicate that a particular block of text should not be interpreted by the parser. One example is a large number of occurrences of the five predefined entities in a block of text that contains no markup, such as a section of code that needs to test for the numeric less-than or Boolean `&&`. In this case, we want text that would normally be considered markup to be treated simply as literal character data. **CDATA sections** are designated portions of an XML document in which all markup is ignored by the parser and all text is treated as character data instead. The main uses of CDATA sections are:

- To delimit blocks of source code (JavaScript, Java, etc.) embedded in XML
- To embed XML, XHTML, or even HTML examples in an XML document
XML Syntax Rules

The general syntax for a CDATA section is:

```
<![CDATA[
multi-line text block to be treated as character data
]]>
```

No spaces are permitted within the two delimiters "<![CDATA[" and "]>".

Here's a CDATA section used to escape a block of code:

```
<![CDATA[
function doIt()
{
    var foo = 3;
    var bar = 13;
    if (foo < 8 && bar > 8)
        alert("Help!");
    else
        alert("I'm Down");
}
]]>
```

An example of embedded XML in XML follows.

```
<Example>
    <Number>2.4</Number>
    <XMLCode>
        <![CDATA[
<?xml version="1.0" standalone="no" ?>
 <!DOCTYPE Message SYSTEM "message.dtd">
<Message mime-type="text/plain">
    <!-- This is a trivial example. -->
    <From>The Kenster</From>
    <To>Silly Little Cowgirl</To>
    <Body>
        Hi, there. How is your gardening going?
    </Body>
</Message>
]]></XMLCode>
</Example>
```

In contrast to our earlier use of the Message example, the character data is not simply the three lines of content of the From, To, and Body elements. When this example is embedded within a CDATA section, the entire block is character data, which in this case means from the XML declaration to and including the </Message> end tag. In other words, the XML prolog, the comment, the start and end tags, and so on, are no longer markup; in this context, they constitute the character data contained by the CDATA section.
Well-Formed vs. Valid Documents

We learned that if a document follows the XML syntax rules discussed in the previous section, the document is said to be well-formed, the minimal requirement to be an XML document. That is, if a document isn’t well-formed, it can’t even be called XML (excepting XML fragments).

Validity

Even if a document is well-formed, however, it may not be valid. According to the XML specification (see http://www.w3.org/TR/REC-xml#sec-documents and http://www.w3.org/TR/REC-xml#dt-valid),

> A data object is an XML document if it is well-formed, as defined in this specification. A well-formed XML document may in addition be valid if it meets certain further constraints. . . . An XML document is valid if it has an associated document type declaration and if the document complies with the constraints expressed in it.

In other words, since a document type declaration (and therefore a DTD) is optional, only documents that refer to a DTD can be checked for validity. This makes sense because following the well-formedness rules only indicates adherence to basic syntactical constraints; it says nothing about meeting the more stringent requirements of a specific structural model.

To reinforce the difference, let’s take another look at the Employees example from chapter 2. The DTD is repeated here in Listing 3-4 for convenience.

Listing 3-4  Employees DTD (employee.dtd)

```xml
<!ELEMENT Employees ( Employee* ) >
<!ELEMENT Employee ( Name, Title, Projects, Email, PhoneNumbers, Address ) >
<!ATTLIST Employee sex NMTOKEN #REQUIRED >
<!ELEMENT Name ( First, Last ) >
<!ELEMENT First ( #PCDATA ) >
<!ELEMENT Last ( #PCDATA ) >
<!ELEMENT Title ( #PCDATA ) >
<!ELEMENT Projects ( Project+ ) >
<!ELEMENT Project ( #PCDATA ) >
<!ELEMENT Email ( #PCDATA ) >
<!ELEMENT PhoneNumbers ( Home, Office, Cell ) >
<!ELEMENT Home ( #PCDATA ) >
<!ELEMENT Office ( #PCDATA ) >
<!ELEMENT Cell ( #PCDATA ) >
<!ELEMENT Address ( Street, City, State, Zip ) >
<!ELEMENT Street ( #PCDATA ) >
<!ELEMENT City ( #PCDATA ) >
<!ELEMENT State ( #PCDATA ) >
<!ELEMENT Zip ( #PCDATA ) >
```
Well-Formed vs. Valid Documents

Now consider the document in Listing 3-5. Is it valid?

**Listing 3-5** Employee Example 1 (employee-WF.xml)

```xml
<?xml version='1.0' standalone='yes'?>
<Employees>
  <Employee sex="female">
    <Name>
      <First>Betty Jo</First>
      <Last>Bialowsky</Last>
      <!-- aka: Melanie Haber, Audrey Farber, Susan Underhill, Nancy -->
    </Name>
    <Title>Project Leader</Title>
    <Projects>
      <Project>MegaProject</Project>
      <Project>SmallProject</Project>
      <Project>others</Project>
    </Projects>
    <Email>BettyJo.Bialowsky@home.com</Email>
    <PhoneNumbers>
      <Home>555-abc-1235</Home>
      <Office>555-xyz-4321</Office>
      <Cell>555-pqr-1267</Cell>
    </PhoneNumbers>
    <Address>
      <Street>321 Carmel Court</Street>
      <City>Columbia</City>
      <State>MD</State>
      <Zip>20777</Zip>
    </Address>
  </Employee>
</Employees>
```

At first glance, it appears to be valid because it follows the structural rules of the DTD. However, because it does not contain a document type declaration, the parser has no DTD to compare the document instance against in order to determine validity. Therefore, it is **well-formed, but not valid, or at least its validity cannot be determined.**

What about the document in Listing 3-6?

**Listing 3-6** Employee Example 2 (employee-Miss-Elt.xml)

```xml
<?xml version='1.0' standalone='no'?>
<!DOCTYPE Employees SYSTEM "employee.dtd">
<Employees>
  <Employee sex="female">
    <Name>
      <First>Betty Jo</First>
      <Last>Bialowsky</Last>
      <!-- aka: Melanie Haber, Audrey Farber, Susan Underhill, Nancy -->
    </Name>
    <Projects>
      <Project>MegaProject</Project>
    </Projects>
  </Employee>
</Employees>
```
With the inclusion of a reference to a DTD, a validating parser can check this instance. It will conclude, however, that there are two missing elements, namely Title and Cell, so this document is well-formed but invalid. In fact, most parsers diagnose the problem quite clearly. For example, the free parser XML Validator from ElCel Technology (http://www.elcel.com/products/xmlvalid.html) reports:

employee-Miss-Elts.xml [19:20] : Error: premature end to content of element 'PhoneNumbers'. Expecting child element 'Cell'

This indicates, among other things, that the errors are detected on lines 10 and 19. And what about this one in Listing 3-7?

Listing 3-7  Employee Example 3 (employee-BJB.xml)

```xml
<?xml version='1.0' standalone='no'?>
<!DOCTYPE Employees SYSTEM "employee.dtd">
<Employees>
  <Employee sex="female">
    <Name>
      <First>Betty Jo</First>
      <Last>Bialowsky</Last>
      <!-- aka: Melanie Haber, Audrey Farber, Susan Underhill, Nancy -->
    </Name>
    <Title>Project Leader</Title>
    <Projects>
      <Project>MegaProject</Project>
      <Project>SmallProject</Project>
      <Project>others</Project>
    </Projects>
    <Email>BettyJo.Bialowsky@home.com</Email>
    <PhoneNumbers>
      <Home>555-abc-1235</Home>
      <Office>555-xyz-4321</Office>
      <Cell>555-pqr-1267</Cell>
    </PhoneNumbers>
  </Employee>
</Employees>
```
This document is well-formed and valid because it matches the structure defined by the DTD, which is referenced in the document type declaration.

**Well-Formed or Toast?**

Ignoring for a moment the potential importance of validity to data-oriented applications, you might wonder why even when an XML document does not require a DTD (i.e., is standalone), it still must be well-formed. In fact, if a document is not well-formed, it cannot even be called an XML document.

The reason for insisting on well-formedness is to counteract the “browser bloat” syndrome that occurred when the major browser vendors decided they wanted their browser to be able to render the horribly inaccurate HTML developed by graduates (or perhaps flunkies) of the Learn HTML in 2 Days or Less school. Many Web pages contain completely invalid HTML, with improperly nested elements, missing end tags, misspelled element names, missing delimiters, and other aberrations. Browsers such as Netscape Communicator and Internet Explorer do an admirable job of recovering from these errors, but only at the expense of a considerable amount of built-in recovery code.

Fortunately, with XML (and XHTML), parsers do not need to implement recovery code and can therefore stay trim and lightweight. If the parser encounters a well-formedness problem, it should only report the problem to the calling application. It explicitly must not attempt to correct what might be missing, overlapping, or misspelled. Violations of well-formedness constraints are considered fatal errors, according to the XML 1.0 Recommendation. The bottom line here is: either a document is well-formed XML, or it’s toast; that is, it’s not XML.

The extra code necessary to do the HTML-like corrections might not be a significant problem for a desktop PC with lots of memory. It’s more of an issue as XML is fed to handheld PCs and other devices with limited memory and/or processing power.

**Validating and Nonvalidating Parsers**

The differences between validating and nonvalidating parsers are not quite as clear as you might think. According to the XML 1.0 specification (http://www.w3.org/TR/REC-xml#proc-types),
Validating processors must, at user option, report violations of the constraints expressed by the declarations in the DTD, and failures to fulfill the validity constraints given in this specification. To accomplish this, validating XML processors must read and process the entire DTD and all external parsed entities referenced in the document. Non-validating processors are required to check only the document entity, including the entire internal DTD subset, for well-formedness.

In other words, validating parsers must read the entire DTD and check the document against the structural constraints it describes. You might conclude, therefore, that nonvalidating parsers do not need to consult the DTD, but that turns out to be incorrect. Even nonvalidating parsers need to supply default values for attributes and to replace text based on internal entities (discussed in chapter 4).

Although there used to be a class of strictly nonvalidating parsers, they tend to be much less popular of late. Most modern parsers (2000 and beyond) can be run in either validating or nonvalidating mode. Why run in nonvalidating mode when a parser is capable of validation? Because validation can significantly impact performance, especially when long and complex DTDs are involved. Some developers find that while enabling validation during development and test phases is crucial, it’s sometimes beneficial to suppress validation in production systems where document throughput is most valued and the reliability of the data is already known. Consult the documentation of prospective parsers to determine how to toggle this switch, and which is the default mode. For example, the Apache Xerces parser is nonvalidating by default.

Some of the more highly regarded XML parsers include:

- Apache XML Project’s Xerces
- IBM’s XML Parser for Java (xml4j)
- JavaSoft’s XML Parser
- MSXML 4.0 Release: Microsoft XML Core Services component (aka MSXML Parser) and SDK
- Oracle’s XML Parser
- ElCel Technology’s XML Validator

URLs for these parsers and many more can be found on the XML Parsers/Processors list at XMLSoftware.com, http://www.xmlsoftware.com/parsers/.

**Event-Based vs. Tree-Based Parsing**

We will cover tree-based and event-based parsing in some depth when we cover SAX and DOM in chapters 7 and 8, respectively. For now, an overview should be sufficient.
**Event-Based Parsing**

Event-based parsers (SAX) provide a data-centric view of XML. When an element is encountered, the idea is to process it and then forget about it. The event-based parser returns the element, its list of attributes, and the content. This is more efficient for many types of applications, especially searches. It requires less code and less memory since there is no need to build a large tree in memory as you are scanning for a particular element, attribute, and/or content sequence in an XML document.

**Tree-Based Parsing**

On the other hand, tree-based parsers (DOM) provide a document-centric view of XML. In tree-based parsing, an in-memory tree is created for the entire document, which is extremely memory-intensive for large documents. All elements and attributes are available at once, but not until the entire document has been parsed. This technique is useful if you need to navigate around the document and perhaps change various document chunks, which is precisely why it is useful for the Document Object Model (DOM), the aim of which is to manipulate documents via scripting languages or Java.

David Megginson, the main force behind Simple API for XML (SAX), contrasts these two approaches in “Events vs. Trees” on the SAX site ([http://www.saxproject.org/?selected=event](http://www.saxproject.org/?selected=event)). The W3C presents its viewpoint in an item from the DOM FAQ, “What is the relationship between the DOM and SAX?” ([http://www.w3.org/DOM/faq#SAXandDOM](http://www.w3.org/DOM/faq#SAXandDOM)).

**Summary**

This chapter covered XML syntax rules and basic parsing concepts.

- We were introduced to fundamental XML terminology, such as element, attribute, tag, and content.
- XML document structure was discussed, including the XML prolog, consisting of the XML declaration and the document type declaration, both of which are optional but desirable.
- Names of elements, attributes, and many other XML identifiers are required to conform to the definition of an XML Name.
- An XML Name consists of a leading letter, underscore, or colon, followed by name characters (letters, digits, hyphens, underscores, colons, or periods).
- XML is case-sensitive. Although there is no universal convention concerning use of uppercase or lowercase when developing your own language, one recommendation is to use UpperCamelCase for elements and lowerCamelCase for attributes, a convention used in SOAP.
We learned the difference between markup and character data; all text that isn’t markup is character data.

We covered most of the types of markup, including start and end tags, empty element tags, entity references, character references, comments, CDATA sections, document type declarations, processing instructions, and XML declarations.

The minimal requirement for an XML document is that it be well-formed, meaning that it adheres to a number of XML syntax rules.

Although well-formedness is a prerequisite for validity, a document can be valid only if it also conforms to the constraints imposed by a DTD or XML Schema.

More modern parsers can be toggled between two states: validating and non-validating. Validation mode is crucial during development. In a production environment, however, it may be desirable (under certain circumstances) to disable validation for efficiency.

Event-based (e.g., SAX) and tree-based (e.g., DOM) parsing were briefly contrasted.

For Further Exploration

Articles
XML Conformance Update (validation), David Brownell
http://www.xml.com/pub/2000/05/10/conformance/conformance.html

XML.com: Being Too Generous, Leigh Dodds [discusses XML conformance in Internet Explorer 6]
http://www.xml.com/pub/a/2001/09/19/being-too-generous.html

To Validate or Not to Validate: Controversy
http://WDVL.Internet.com/Authoring/Languages/XML/DoingIt/ValidationControversy.html

Resources
XML Checkers and Validation Services
http://WDVL.Internet.com/Software/XML/parsers.html#checking

XML.com: Search for Conformance
http://www.xml.com/search/index.ncsp?sp-q=Conformance&search=search

Software
XML Parsers at XMLSoftware
http://www.xmlsoftware.com/parsers/

XML Editors at XMLSoftware
http://www.xmlsoftware.com/editors/
For Further Exploration

**W3C Specifications**

Extensible Markup Language (XML) 1.0 [W3C XML Recommendation, Second Edition]
http://www.w3.org/TR/REC-xml

Well-Formed XML Documents
http://www.w3.org/TR/REC-xml#sec-well-formed

Validating and Non-Validating Processors
http://www.w3.org/TR/REC-xml#proc-types

XHTML 1.0: The Extensible HyperText Markup Language—A Reformulation of HTML 4 in XML 1.0
http://www.w3.org/TR/xhtml1/