CHAPTER 1

XML Concepts

This book is targeted at programmers who need to develop solutions using XML. Being a programmer myself, I know that theory without practical examples and applications can be tedious, and you probably want to get straight to real-world examples. You're in luck, because this book is full of working examples — but not in this chapter. Some theory is necessary so that you have a fundamental understanding of XML. I'll keep the theory of XML and related technologies to a minimum as I progress through the chapters, but we do need to cover some of the basics up front.

This chapter provides readers who are new to XML with an overview and history of XML, its purposes, and comparisons against previous and alternative integration technologies, and ends with an overview of the next XML version, XML 1.1. The rest of the chapters in this part of the book will use real-world examples to describe XML basic formats, the structure of well-formed XML documents, and XML validation against DTDs and Schemas. The chapters on XSL Transformations and XSL Formatting Objects will illustrate the transformation and formatting of XML data using XSLT via working examples. This part of the book will be finished with examples of parsing XML documents, as well as specific examples of XML parsing using Simple API for XML (SAX) and Document Object Model (DOM).

What Is XML?

XML stands for *Extensible Markup Language,* and it is used to describe documents and data in a standardized, text-based format that can be easily transported via standard Internet protocols. XML, like HTML, is based on the granddaddy of all markup languages, Standard Generalized Markup Language (SGML).

SGML is remarkable not just because it's the inspiration and basis for all modern markup languages, but also because of the fact that SGML was created in 1974 as part of an IBM document-sharing project, and officially became an

In This Chapter

International Organization for Standardization (ISO) standard in 1986, long before the Internet or anything like it was operational. The ISO standard documentation for SGML (ISO 8879:1986) can be purchased online at `http://www.iso.org`.

The first popular adaptation of SGML was HTML, which was developed as part of a project to provide a common language for sharing technical documents. The advent of the Internet facilitated the document exchange method, but not the display of the document. The markup language that was developed to standardize the display format of the documents was called *Hypertext Markup Language,* or *HTML*, which provides a standardized way of describing document layout and display, and is an integral part of every Web browser and Website.

Although SGML was a good format for document sharing, and HTML was a good language for describing the *layout* of the documents in a standardized way, there was no standardized way to describe and share *data* that was stored in the document. For example, an HTML page might have a body that contains a listing of today's closing prices of a share of every company in the Fortune 500. This data can be displayed using HTML in a myriad of ways. Prices can be bold if they have moved up or down by 10 percent, and prices that are up from yesterday's closing price can be displayed in green, with prices that are down displayed in red. The information can be formatted in a table, and alternating rows of the table can be in different colors.

However, once the data is taken from its original source and rendered as HTML in a browser, the values of the data only have value as part of the markup language on that page. They are no longer individual pieces of data, but are now simply pieces of "content" wedged between elements and attributes that specify how to display that content. For example, if a Web developer wanted to extract the top ten price movers from the daily closing prices displayed on the Web page, there was no standardized way to locate the top ten values and isolate them from the others, and relate the prices to the associated Fortune 500 Company.

Note that I say that there was no *standardized* way to do this; this did not stop developers from trying. Many a Web developer in the mid- to late-1990s, including myself, devised very elaborate and clever ways of scraping the data they needed from between HTML tags, mostly by eyeballing the page and the HTML source code, then coding routines in various languages to read, parse, and locate the required values in the page. For example, a developer may read the HTML source code of the stock price page and discover that the prices were located in the only table on the HTML page. With this knowledge, code could be developed in the developer's choice of language to locate the table in the page, extract the values nested in the table, calculate the top price movers for the day based on values in the third column in the table, and relate the company name in the first column of the table with the top ten values.

However, it's fair to say that this approach represented a maintenance nightmare for developers. For example, if the original Web page developers suddenly decided to add a table before the stock price table on the page, or add an additional column to the table, or nest one table in another, it was back to the drawing board for the developer who was scraping the data from the HTML page, starting over to find the values in the page, extract the values into meaningful data, and so on. Most developers who struggled with this inefficient method of data exchange on the Web were looking for better ways to share data while still using the Web as a data delivery mechanism.

But this is only one example of many to explain the need for a tag-based markup language that could describe data more effectively than HTML. With the explosion of the Web, the need for a universal format that could function as a lowest common denominator for data exchange while still using the very popular and standardized HTTP delivery methods of the Internet was growing.

In 1998 the World Wide Web Consortium (W3C) met this need by combining the basic features that separate data from format in SGML with extension of the HTML tag formats that were adapted for the Web and came up with the first Extensible Markup Language (XML) Recommendation. The three pillars of XML are Extensibility, Structure, and Validity.

Extensibility

XML does a great job of describing structured data as text, and the format is open to extension. This means that any data that can be described as text and that can be nested in XML tags will be generally accepted as XML. Extensions to the language need only follow the basic XML syntax and can otherwise take XML wherever the developer would like to go. The only limits are imposed on the data by the data itself, via syntax rules and self-imposed format directives via data validation, which I will get into in the next chapter.

Structure

The structure of XML is usually complex and hard for human eyes to follow, but it's important to remember that it's not designed for us to read. XML parsers and other types of tools that are designed to work with XML easily digest XML, even in its most complex forms. Also, XML was designed to be an open data exchange format, not a compact one—XML representations of data are usually much larger than their original formats. In other words, XML was not designed to solve disk space or bandwidth issues, even though text-based XML formats do compress very well using regular data compression and transport tools.

It's also important to remember that XML data syntax, while extensible, is rigidly enforced compared to HTML formats. I will get into the specifics of formatting rules a little later in this chapter, and will show examples in the next chapter.

Validity

Aside from the mandatory syntax requirements that make up an XML document, data represented by XML can optionally be validated for structure and content, based on two separate data validation standards. The original XML data validation standard is called Data Type Definition (DTD), and the more recent evolution of XML data validation is the XML Schema standard. I will be covering data validation using DTDs and Schemas a little later in this chapter, and showing working examples of data validation in the next chapter.

What Is XML Not?

With all the hype that continues to surround XML and derivative technologies such as XSL and Web Services, it's probably as important to review what XML is not as it is to review what XML is.

While XML facilitates data integration by providing a transport with which to send and receive data in a common format, XML is not data integration. It's simply the glue that holds data integration solutions together with a multi-platform "lowest common denominator" for data transportation. XML cannot make queries against a data source or read data into a repository by itself. Similarly, data cannot be formatted as XML without additional tools or programming languages that specifically generate XML data from other types of data. Also, data cannot be parsed into destination data formats without a parser or other type of application that converts data from XML to a compatible destination format.

It's also important to point out that XML is not HTML. XML may look like HTML, based on the similarities of the tags and the general format of the data, but that's where the similarity ends. While HTML is designed to describe display characteristics of data on a Web page to browsers, XML is designed to represent data structures. XML data can be transformed into HTML using Extensible Style Sheet Transformations (XSLT). XML can also be parsed and formatted as HTML in an application. XML can also be part of an XML page using XML data islands. I'll discuss XSLT transformations, XML parsing, and data islands in much more detail later in the book.

XML Standards and the World Wide Web Consortium

The World Wide Web Consortium (W3C) is where developers will find most of the specifications for standards that are used in the XML world. W3C specifications are referred to as "Recommendations" because the final stage in the W3C development process may not necessarily produce a specification, depending on the nature of the W3C Working Group that is producing the final product, but for all intents and purposes, most of the final products are specifications.

W3C specifications on the Recommendation track progress through five stages: Working Draft, Last Call Working Draft, Candidate Recommendation, Proposed Recommendation, and Recommendation, which is the final stop for a specific version of a specification such as XML.

W3C Working Groups produce Recommendations, and anyone can join the W3C and a Working Group. More information on joining the W3C can be found at `http://www.w3.org/Consortium/Prospectus/Joining`. Currently, W3C Working Groups are working hard at producing the latest recommendations for XML and related technologies such as XHTML, Xlink, XML Base, XML Encryption, XML Key Management, XML Query, XML Schema, XML Signature, Xpath, Xpointer, XSL, and XSLT.

XML Elements and Attributes

Because XML is designed to describe data and documents, the W3C XML Recommendation, which can be found buried in the links at `http://www.w3.org/XML`, is very strict about a small core of format requirements that make the difference between a text document containing a bunch of tags and an actual XML document. XML documents that meet W3C XML document formatting recommendations are described as being *well-formed* XML documents. Well-formed XML documents can contain elements, attributes, and text.

Elements

Elements look like this and always have an opening and closing tag:

```
<element></element>
```

There are a few basic rules for XML document elements. Element names can contain letters, numbers, hyphens, underscores, periods, and colons when namespaces are used (more on namespaces later). Element names cannot contain spaces;

underscores are usually used to replace spaces. Element names can start with a letter, underscore, or colon, but cannot start with other non-alphabetic characters or a number, or the letters *xml*.

Aside from the basic rules, it's important to think about using hyphens or periods in element names. They may be considered part of well-formed XML documents, but other systems that will use the data in the element name such as relational database systems often have trouble working with hyphens or periods in data identifiers, often mistaking them for something other than part of the name.

Attributes

Attributes contain values that are associated with an element and are always part of an element's opening tag:

```
<element attribute="value"></element>
```

The basic rules and guidelines for elements apply to attributes as well, with a few additions. The attribute name must follow an element name, then an equals sign (=), then the attribute value, in single or double quotes. The attribute value can contain quotes, and if it does, one type of quote must be used in the value, and another around the value.

Text

Text is located between the opening and closing tags of an element, and usually represents the actual data associated with the elements and attributes that surround the text:

```
<element attribute="value">text</element>
```

Text is not constrained by the same syntax rules of elements and attributes, so virtually any text can be stored between XML document elements. Note that while the value is limited to text, the format of the text can be specified as another type of data by the elements and attributes in the XML document.

Empty elements

Last but not least, elements with no attributes or text can also be represented in an XML document like this:

```
<element/>
```

This format is usually added to XML documents to accommodate a predefined data structure. I'll be covering ways to specify an XML data structure a little later in this chapter.

XML Document Structure

Although elements, attributes, and text are very important for XML documents, these design objects alone do not make up a well-formed XML document without being arranged under certain structural and syntax rules. Let's examine the structure of the very simple well-formed XML 1.0 document in Listing 1-1.

Listing 1-1: **A Very Simple XML Document**

```
<?xml version="1.0" encoding="UTF-8"?>
<rootelement>
   <firstelement position="1">
      <level1 children="0">This is level 1 of the nested
       elements</level1>
   </firstelement>
   <secondelement position="2">
      <level1 children="1">
         <level2>This is level 2 of the nested
         elements</level2>
      </level1>
   </secondelement>
</rootelement>
```

Most XML documents start with an `<?xml?>` element at the top of the page. This is called an XML document declaration. An XML document declaration is an optional element that is useful to determine the version of XML and the encoding type of the source data. It is not a required element for an XML document to be well formed in the W3C XML 1.0 specification. This is the most common XML document declaration:

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

There are two attributes contained in this XML declaration that are commonly seen but not often explained. The XML version is used to determine what version of the W3C XML recommendation that the document adheres to. XML parsers use this information to apply version-specific syntax rules to the XML document.

Data Source Encoding

Data source encoding is one of the most important features for XML documents. Most developers based in the United States or other English-speaking countries are familiar with ASCII text only, and have not commonly tested the capacity of ASCII's 128-member character set. However, with the advent of the Internet, HTML and

especially XML developers have been forced to examine the limitations of ASCII, and have worked with Unicode in HTML documents, even if they didn't know that they were (HTML code generators usually add the Unicode directives to HTML pages).

Because the XML Recommendation was developed by the W3C, an international organization which has offices at the Massachusetts Institute of Technology (MIT) in the United States, the European Research Consortium for Informatics and Mathematics (ERCIM) in France, and Keio University in Japan, Unicode was chosen as the standard text format to accommodate the world's languages, instead of just English. Most developers are used to seeing UTF-8 or sometimes UTF-16 in the encoding attribute of an XML document, but this is just the tip of the iceberg.

UTF stands for *Universal Character Set Transformation Format,* and the number 8 or 16 refers to the number of bits that the character is stored in. Each 8- or 16-bit container represents the value of the character in bits as well as the identity of each character and its numeric value. UTF-8 is the most common form of XML encoding; in fact, an XML document that does not specify an encoding type must adhere to either UTF-8 or UTF-16 to be considered a well-formed XML 1.0 document. Using UTF-8, UTF-16, and the newer UTF-32, XML editors, generators and parsers can identify and work with all major world languages and alphabets, including non-Latin alphabets such as Middle Eastern and Asian alphabets, scripts, and languages. This includes punctuation, non-Arabic numbers, math symbols, accents, and so on.

Unicode is managed and developed by a non-profit group called the Unicode Consortium. For more information on encoding and a listing of encoding types for XML, the Unicode consortium and the W3C has published a joint report, available at the Unicode Consortium site:
`http://www.unicode.org/unicode/reports/tr20`.

Aside from UTF declarations for XML document encoding, any ISO registered charset name that is registered by the Internet Assigned Numbers Authority (IANA) is an acceptable substitute. For example, an XML 1.0 document encoded in Macedonian would look like this in the XML declaration:

```
<?xml version="1.0" encoding="JUS_I.B1.003-mac"?>
```

A list of currently registered names can be found at
`http://www.iana.org/assignments/character-sets`.

Element and Attribute Structure

Under the optional XML declarations, every XML document contains a single-value root element, represented in this case by the `rootelement` element:

```
<rootelement>
```

Other elements and text values can be nested under the root element, but the root element must be first in the list and unique in the document. This can be compared to a computer hard drive, which contains one root directory, with files and/or subdirectories under the root directory.

Next in the sample XML document are the nested elements, attributes, and text, as illustrated by the nested `firstelement` under the root element in our example:

```
<firstelement position="1">
   <level1 children="0">This is level 1 of the nested
    elements</level1>
</firstelement>
```

The `firstelement` has an attribute called `position` with a value of `1`. The `position` attribute provides additional data related to `firstelement`. In this case it indicates that the original sorting position of the first element in the XML document is `1`. If the XML document data is altered and the order of the elements is rearranged as part of that alteration, the position element may be useful for reordering the element, or could be changed when the document is altered to reflect a new position of the element in the XML document, regardless of the element name. In general, attributes are great for adding more information and descriptions to the values of elements, and the text associated with elements, as shown in the previous example.

Nested under the `firstelement` element is the `level1` element, which contains an attribute, called `children`. The element name is used to describe the nesting level in the XML document, and the value of the `children` attribute is used to describe how many more levels of nesting are contained under the `level1` element, in this case, no more nested levels (`0`). The phrase `This is level 1 of the nested elements` represents a text data value that is part of the `level1` element. Text data contains values associated with a tag.

The second element under the root element is called `secondelement` and is a variation of the `firstelement` element. Let's compare the `firstelement` and `secondelement` elements to get a better sense of the structure of the document:

```
<secondelement position="2">
   <level1 children="1">
      <level2>This is level 2 of the nested
       elements</level2>
   </level1>
</secondelement>
```

Like the `firstelement`, the `secondelement` has an attribute called `position`, this time with a value of `2`. Nested under the `secondelement` element is another `level1` element. The existence of this element illustrates the fact that well-formed XML documents can have more than one instance of the same element name. The only exception to this is the root element, which must be unique.

Also, like the `firstelement` element, the `level1` element also has an attribute called `children`. The `level1` element is again used to describe the nesting level in the XML document, and the attribute is used to describe how many more levels of nesting are contained under the `level1` element. In this case, the children attribute indicates that there is one more nesting level (`1`) inside the `level1` element. The phrase `This is level 2 of the nested elements` inside the `level2` element represents text data for the `level2` element.

Last but not least, to finish the XML document, the `rootelement` tag is closed:

```
</rootelement>
```

XML Document Syntax

Another important aspect of a well-formed XML document is the document syntax. XML represents data and not content or layout like other markup languages such as HTML. Data has very strict structure and format rules. XML also has very strict rules about the syntax used to represent that data. Developers who are used to coding with the somewhat forgiving syntax of HTML will have some adjustments to make when dealing with XML syntax.

For starters, XML element names must start and end with the same case. This is not well-formed XML:

```
<level2>This is level 2 of the nested elements</Level2>
```

The tag name started with `<level2>` must be closed with `</level2>`, not `</Level2>`, to be considered well-formed XML.

Quotes must be used on all attribute names. Something like this will not be considered well-formed XML:

```
<secondelement position=2>
```

Attributes must be formatted with single or double quotes to be considered well-formed XML:

```
<secondelement position="2">
```

Comments should always follow the SGML comment tag format:

```
<!--Comment tags should always follow this format when in XML
documents-->
```

Element tags must always be closed. HTML and other forms of markup are somewhat forgiving, and can often be left open or improperly nested without affecting the content or display of a page. XML parsers and other tools that read and manipulate XML documents are far less forgiving about structure and syntax than browsers.

XML Namespaces

Namespaces are a method for separating and identifying duplicate XML element names in an XML document. Namespaces can also be used as identifiers to describe data types and other information. Namespace declarations can be compared to defining a short variable name for a long variable (such as pi=3.14159....) in programming languages. In XML, the variable assignment is defined by an attribute declaration. The variable name is the attribute name, and the variable value is the attribute value. In order to identify namespace declarations versus other types of attribute declarations, a reserved `xmlns:` prefix is used when declaring a namespace name and value. The attribute name after the `xmlns:` prefix identifies the name for the defined namespace. The value of the attribute provides the unique identifier for the namespace. Once the namespace is declared, the namespace name can be used as a prefix in element names.

Listing 1-2 shows the very simple XML document I reviewed in Listing 1-1, this time with some namespaces to differentiate between nested elements.

Listing 1-2: **A Very Simple XML Document with Namespaces**

```
<?xml version="1.0" encoding="UTF-8"?>
<rootelement>
   <firstelement
    xmlns:fe="http://www.benztech.com/schemas/verybasic"
    position="1">
      <fe:level1 children="0">This is level 1 of the nested
       elements</fe:level1>
   </firstelement>
   <secondelement
    xmlns:se="http://www.benztech.com/schemas/verybasic"
    position="2">
      <se:level1 children="1">
         <se:level2>This is level 2 of the nested
          elements</se:level2>
      </se:level1>
   </secondelement>
</rootelement>
```

In this example, I am using two namespaces as identifiers to differentiate two `level1` elements in the same document. The `xmlns:` attribute declares the namespace for an XML document or a portion of an XML document. The attribute can be placed in the root element of the document, or in any other nested element.

In our example, the namespace name for the `firstelement` element is `fe`, and the namespace name for the secondelement is `se`. Both use the same URL as the value for the namespace. Often the URL in the namespace resolves to a Web page that

provides documentation about the namespace, such as information about the data encoding types identified in the namespace. However, in this case, we are just using the namespace to defined prefixed for unique identification of duplicate element names. The URL does resolve to an actual document, but is just used as a placeholder for the namespace name declarations.

Although the namespace declaration value does not need to be a URL or resolve to an actual URL destination, it is a good idea to use a URL anyway, and to choose a URL that could resolve to an actual destination, just in case developers want to add documentation for the namespace to the URL in the future.

When to use namespaces

Namespaces are optional components of basic XML documents. However, namespace declarations are recommended if your XML documents have any current or future potential of being shared with other XML documents that may share the same element names. Also, newer XML-based technologies such as XML Schemas, SOAP, and WSDL make heavy use of XML namespaces to identify data encoding types and important elements of their structure. I'll be showing many more examples of namespaces being used in context to identify elements for XML document data encoding, identification, and description as the examples progress through the book.

XML Data Validation

As I've shown you so far in this chapter, there are very strict rules for the basic structure and syntax of well-formed XML documents. There are also several formats within the boundaries of well-formed XML syntax that provide standardized ways of representing specific types of data.

For example, NewsML offers a standard format for packaging news information in XML. NewsML defines what the element name should be that contains the title, publication date, headline, article text, and other parts of a news item. NewsML also defines how these elements should be arranged, and which elements are optional. NewsML documents are well-formed XML, and they also conform to NewsML specifications.

The validity of an XML document is determined by a Document Type Definition (DTD) or an XML Schema. There are several formats for data validation to choose from. A good listing for XML validation formats can be found at `http://www.oasis-open.org/cover/schemas.html`. However, the most common and officially W3C sanctioned formats are the Document Type Definition (DTD) and the W3C Schema, which I will focus on in this chapter.

XML documents are compared to rules that are specified in a DTD or schema. A well-formed XML document that meets all of the requirements of one or more specifications is called a *valid* XML Document.

XML documents do not validate themselves. XML validation takes place when a document is parsed. Most of today's parsers have validation built-in to the core functionality, and usually support W3C Schema and DTD validation, and may support other types of validation, depending on the parser. In addition, defining a variable or calling a different class in the parser can often disable validation by ignoring DTD and/or Schema directives in the XML document. Parsers or parser classes that don't support validation are called *nonvalidating parsers,* and parsers or classes that support validation are called *validating parsers.*

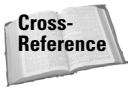
For example, the NewsML specification is defined and managed by the International Press Telecommunications Council (IPTC). The IPTC has published a DTD that can be used by news providers to validate NewsML news items (Reuters and other news providers have NewsML-compatible news feeds). If a member of the press wants to produce NewsML formatted news items, they can download the DTD from the IPTC Website at `http://www.iptc.org`. Once the DTD is downloaded, XML developers can validate their NewsML output against the DTD using a validating parser.

Listing 1-3 shows the same simple XML document in Listing 1-1, but this time there is a DTD and a Schema reference in the document.

Listing 1-3: **A Very Simple XML Document with a Schema and DTD Reference**

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE rootelement SYSTEM "verysimplexml.dtd">
<rootelement xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:noNamespaceSchemaLocation="verysimplexml.xsd">
   <firstelement position="1">
      <level1 children="0">This is level 1 of the nested
       elements</level1>
   </firstelement>
   <secondelement position="2">
      <level1 children="1">
         <level2>This is level 2 of the nested
          elements</level2>
      </level1>
   </secondelement>
</rootelement>
```

It is not common to see both DTD and Schema references in a single document that verify the same structural rules, but it's a good example of the fact that you can combine Schema and DTD references in a single document. References to a DTD and a schema can occur when an XML document is made up of two or more source documents. The DTD and schema references maintain all of the structure rules that were present in the original document. Dual references can also be used when illegal XML characters are represented in an XML document by entity references. I'll describe entity references in more detail later in this chapter.



The following section of this chapter is intended to give you an introductory overview of DTDs and W3C Schemas. For more detail on XML document validation with real-world examples, please see Chapter 3.

Validating XML documents with DTDs

Document Type Definition (DTD) is the original way to validate XML document structure and enforce specific formatting of select text, and probably still the most prevalent. Although the posting of the XML declaration at the top of the DTD would lead one to believe that this is an XML document, DTDs are in fact non-well-formed XML documents. This is because they follow DTD syntax rules rather than XML document syntax. In Listing 1-3, the reference is to the DTD located in the first element under the XML document declaration:

```
<!DOCTYPE rootelement SYSTEM "verysimplexml.dtd">
```

Listing 1-4 shows the `verysimplexml.dtd` file that is referred to in the XML document in Listing 1-3.

Listing 1-4: **Contents of the verysimplexml.dtd File**

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT rootelement (firstelement, secondelement)>
<!ELEMENT firstelement (level1)>
<!ATTLIST firstelement
   position CDATA #REQUIRED
>
<!ELEMENT level1 (#PCDATA | level2)*>
<!ATTLIST level1
   children (0 | 1) #REQUIRED
>
<!ATTLIST secondelement
   position CDATA #REQUIRED
>
<!ELEMENT level2 (#PCDATA)>
<!ELEMENT secondelement (level1)>
```

Let's go through this DTD line by line to get to know DTD structure. The first line is an XML document declaration, which tells parsers the version of XML and the encoding type for the document. The next line specifies that valid XML documents must contain a `firstelement` and the `secondelement`, which have to be present under the `rootelement`, and have to be in the order listed:

```
<!ELEMENT rootelement (firstelement, secondelement)>
```

Next, the DTD describes the `firstelement`. The `firstelement` must have a `level1` element nested directly under the `firstelement`.:

```
<!ELEMENT firstelement (level1)>
```

Next, the DTD specifies an attribute for the `firstelement`. The `ATTLIST` declaration tells us that valid XML documents need a position attribute for each instance of the `firstelement` (`#REQUIRED`), and that it is regular character data (`CDATA`):

```
<!ATTLIST firstelement
   position CDATA #REQUIRED
>
```

The next element declaration tells us that the `level1` element can contain one of two things. The `|` is equivalent to an `or` in a DTD. The `level1` element can contain another nested element called `level2`, or a value of parsed character data (`PCDATA`):

```
<!ELEMENT level1 (#PCDATA | level2)*>
```

The next `ATTLIST` declaration tells us that `level1` can have one of two values, `0` or `1`:

```
<!ATTLIST level1
   children (0 | 1) #REQUIRED
>
```

The `ATTLIST` declaration for `secondlement` tells us that valid XML documents need a position attribute for each instance of the `secondelement` (`#REQUIRED`), and that it is regular character data (`CDATA`):

```
<!ATTLIST secondelement
   position CDATA #REQUIRED
>
```

Following the nesting deeper into the document, a declaration for the `level2` element is defined. The `level2` element declaration simply states that the element must contain a value of parsed character data (`PCDATA`):

```
<!ELEMENT level2 (#PCDATA)>
```

Last but not least, the `secondelement` is defined, along with a mandatory `level1` element nested underneath it:

```
<!ELEMENT secondelement (level1)>
```

As you can see from the last few lines of this DTD, the element and attribute declarations do not have to be in the same order as the element and attributes that they represent. It is up to the parser to reassemble the DTD into something that defines the relationship of all the elements and enforces all the rules contained in each line of the DTD.

Validating XML documents with Schemas

The W3C Schema is the officially sanctioned Schema definition. Unlike DTDs, the format of W3C Schemas follows the rules of well-formed XML documents. The Schema also allows for much more granular control over the data that is being described. Because of the XML format and the detailed format controls, Schemas tend to be very complex and often much longer than the XML documents that they are describing. Paradoxically, Schemas are often much more easy for developers to read and follow, due to the less cryptic nature of the references in Schemas versus DTDs.

References to schemas are defined by creating an instance of the `XMLSchema-instance` namespace. Here is the Schema declaration in the XML document in Listing 1-3:

```
<rootelement xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:noNamespaceSchemaLocation="verysimplexml.xsd">
```

In this case, the namespace declaration reference to `http://www.w3.org/2001/XMLSchema-instance` resolves to an actual document at that location, which is a brief description of the way that the W3C Schema should be referenced. The `noNamespaceSchemaLocation` value tells us that there is no predefined namespace for the Schema. This means that all of the elements in the XML document should be validated against the schema specified. The location of the Schema I am using is verysimplexml.xsd. Because there is no path defined, the file containing the schema should be located in the same directory as the XML file to be validated by the Schema.

You can also define the schema location, and map it to a specific namespace by using the `schemaLocation` attribute declaration instead of `noNamespace SchemaLocation`. If you do so, you have to declare a namespace that matches the `schemaLocation` attribute value. The declaration must be made before you reference the schema in a `schemaLocation` attribute assignment. Here's an example of a `schemaLocation` assignment in a root element of an XML document:

```
<rootelement
xmlns:fe="http://www.benztech.com/schemas/verybasic"
xsi:schemaLocation="http://www.benztech.com/schemas/verybasic
">
```

Listing 1-5 shows the `verysimplexml.xsd` file that is referred to in the XML document in Listing 1-3.

Listing 1-5: **Contents of the verysimplexml.xsd File**

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
   <xs:element name="firstelement">
      <xs:complexType>
         <xs:sequence>
            <xs:element ref="level1"/>
         </xs:sequence>
         <xs:attribute name="position" type="xs:boolean"
          use="required"/>
      </xs:complexType>
   </xs:element>
   <xs:element name="level1">
      <xs:complexType mixed="true">
         <xs:choice minOccurs="0" maxOccurs="unbounded">
            <xs:element ref="level2"/>
         </xs:choice>
         <xs:attribute name="children" use="required">
            <xs:simpleType>
               <xs:restriction base="xs:NMTOKEN">
                  <xs:enumeration value="0"/>
                  <xs:enumeration value="1"/>
               </xs:restriction>
            </xs:simpleType>
         </xs:attribute>
      </xs:complexType>
   </xs:element>
   <xs:element name="level2" type="xs:string"/>
   <xs:element name="rootelement">
      <xs:complexType>
         <xs:sequence>
            <xs:element ref="firstelement"/>
            <xs:element ref="secondelement"/>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
   <xs:element name="secondelement">
      <xs:complexType>
         <xs:sequence>
            <xs:element ref="level1"/>
         </xs:sequence>
         <xs:attribute name="position" type="xs:byte"
          use="required"/>
      </xs:complexType>
   </xs:element>
</xs:schema>
```

I'll go through this code line by line to introduce readers to the W3C Schema XSD format. After the declaration, the next line refers to the `xs` namespace for XML Schemas. The reference URL, `http://www.w3.org/2001/XMLSchema`, actually resolves to the W3C Website and provides documentation for Schemas, as well as reference materials for data types and Schema namespace formatting.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
```

The first element definition describes the `firstelement` as a complex data type, that the element contains one nested element called `level1`, and an attribute called `position`, and that the attribute is required.

```
<xs:element name="firstelement">
   <xs:complexType>
      <xs:sequence>
         <xs:element ref="level1"/>
      </xs:sequence>
      <xs:attribute name="position" type="xs:boolean"
       use="required"/>
   </xs:complexType>
</xs:element>
```

The next element describes the `level1` element, that it is an optional element (`minOccurs="0"`), and that the `level1` element can occur an unlimited number of times in the document (`maxOccurs="unbounded"`). Nested in the `level1` element is a reference to the `level2` element, just as it is in the document. Next, the `children` attribute is specified as required, and defined as a simple Schema data type called `NMTOKEN` value for the base attribute, which is, for the purposes of this schema, a string. The children string must be one of two predefined values, `"0"` and `"1"`, as defined by the enumeration values nested inside of the restriction element.

```
<xs:element name="level1">
   <xs:complexType mixed="true">
      <xs:choice minOccurs="0" maxOccurs="unbounded">
         <xs:element ref="level2"/>
      </xs:choice>
      <xs:attribute name="children" use="required">
         <xs:simpleType>
            <xs:restriction base="xs:NMTOKEN">
               <xs:enumeration value="0"/>
               <xs:enumeration value="1"/>
            </xs:restriction>
         </xs:simpleType>
      </xs:attribute>
   </xs:complexType>
</xs:element>
```

Because the `level2` element has no attributes or nested elements, it can be described in one line and referred to as a nested element in the `level1` element via the `ref=` reference:

```
    <xs:element name="level2" type="xs:string"/>
<xs:element ref="level2"/>
```

As with the DTD example, the element and attribute declarations in a W3C Schema do not have to be in the same order as the element and attributes that they represent in an XML document. Like the DTD, it is up to the parser to reassemble the Schema into something that defines the relationship of all the elements and enforces all the rules contained in each line of the Schema, regardless of the order. The next element in this Schema example is the `rootelement`. The `rootelement` must have a `firstelement` and a `secondelement` nested under it to be considered a valid XML document when using this Schema. The previous definitions for the `firstelement`, `secondelement`, and all the nested elements underneath them are defined earlier in the Schema.

```
<xs:element name="rootelement">
   <xs:complexType>
      <xs:sequence>
         <xs:element ref="firstelement"/>
         <xs:element ref="secondelement"/>
      </xs:sequence>
   </xs:complexType>
</xs:element>
```

The schema defines the `secondelement`, which must contain a nested `level1` element, and have an attribute named `position`, this time a `byte` value.

```
<xs:element name="secondelement">
   <xs:complexType>
      <xs:sequence>
         <xs:element ref="level1"/>
      </xs:sequence>
      <xs:attribute name="position" type="xs:byte"
       use="required"/>
   </xs:complexType>
</xs:element>
```

Finally, the closing of the schema tag indicates the end of the schema.

```
</xs:schema>
```

Special Characters and Entity References

The W3C XML Recommendation also supports supplements to the default encoding. Special characters in a well-formed XML document can be referenced via a declared entity, Unicode, or hex character reference. Entity references must start with an ampersand (`&`), Unicode character references start with an ampersand and a pound sign (`&#`), and hexadecimal character references start with an ampersand, pound sign, and an x (`&#x`). All entity, Unicode, and hexadecimal references end with a semicolon (`;`).

Listing 1-6 shows a simple XML document that uses Entity, Unicode, and Hex references to generate a copyright symbol (`(c)`) and a registered trademark symbol (`(r)`) in an XML document.

Listing 1-6: **Entity, Unicode, and Hex Character References in an XML Document**

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE rootelement SYSTEM "specialcharacters.dtd">
<rootelement>
   <entityreferences>&copy; &reg;</entityreferences>
   <unicodereferences>&#169; &#174;</unicodereferences>
   <hexreferences>&#xA9; &#xAE;</hexreferences>
</rootelement>
```

The values in the `unicodereferences` and `hexreferences` elements are the Unicode and hex values that represent the symbols. Both follow the character reference rules outlined earlier. The addition of a DTD is necessary for the entity references in the `entityreferences` element. The values for the entity references must be defined outside of the XML document. Listing 1-7 shows the specialcharacters.dtd file, including the entity definitions for `&copy` and `&reg`. This very basic DTD defines the structure of the document, and also defines two entity references and their values. I've created a Hex and a Unicode reference to illustrate that entity references in XML documents can refer to either format. The first `ENTITY` tag in the DTD defines the copy reference as the hex character reference `%A9`. The value follows XML rules for formatting a hex character reference, which makes the hex value `"&#xA9;"`. The second `ENTITY` tag refers to the Unicode character `174`, formatted as `"&#174;"` according to XML document Unicode character reference rules.

Listing 1-7: The specialcharacters.dtd File with Entity Definitions for &copy and &reg

```
<?xml version="1.0" encoding="UTF-8"?>
<!ENTITY copy "&#xA9;">
<!ENTITY reg "&#174;">
<!ELEMENT rootelement (entityreferences, unicodereferences,
hexreferences)>
<!ELEMENT entityreferences (#PCDATA)>
<!ELEMENT hexreferences (#PCDATA)>
<!ELEMENT unicodereferences (#PCDATA)>
```

Listing 1-8 shows the output from the XML document, with the resolved character references. This is what the document looks like when the character and entity references are rendered by a Microsoft Internet Explorer 6 browser.

Listing 1-8: MSIE Rendered Character and Entity References Using the specialcharacters.dtd File

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE rootelement (View Source for full doctype...)>
<rootelement>
  <entityreferences>(c) (r)</entityreferences>
  <unicodereferences>(c) (r)</unicodereferences>
  <hexreferences>(c) (r)</hexreferences>
</rootelement>
```

Using entity references as variables

Entity references can also be used as variables and combined with other entity references in a DTD, which is a handy way of standardizing certain declarations and other unalterable components of an XML document. For example, an entity reference called `copyline` can be created in a DTD like this:

```
<!ENTITY copy "&#xA9;">
<!ENTITY copyline "&copy; Benz Technologies, Inc, all rights
reserved;">
```

When a reference to the `&copyline;` entity is made in an XML document, the output would look like this:

```
(c) Benz Technologies, Inc, all rights reserved
```

Using this technique ensures that XML document validation imposes a standard format for certain important pieces of text in an XML document, as well as the structure of the document.

Reserved character references

All of the character reference formats defined earlier include an ampersand. So how do you represent an ampersand in XML documents? To accommodate ampersands and four other special characters that are part of the XML core syntax special reserved character references are defined. Less than and greater than symbols (which are used to define XML elements), and quotes (which are used to define attribute values) are supported via special predefined character substitutions without any Entity, Unicode, or Hex references needed. An ampersand (`&`) is used at the beginning and a semicolon (`;`) is placed at the end of the reference. Table 1-1 shows the reserved character entity and its reference.

Table 1-1
Reserved Character Entities and References

Entity Reference	*Special Character*
&amp;	ampersand (&)
&apos	apostrophe or single quote (')
&gt;	greater-than (>)
&lt;	less-than (<)
&quot;	double quote (")

XML 1.1

XML 1.1 represents an incremental development of the W3C XML recommendation. The new recommendation is actually split into two significant recommendations, XML 1.1 and XML Namespaces 1.1. Most of the new features are "behind the scenes" enhancements, which will have little or no effect on most XML applications. For example, the new way of handling line-endings in XML 1.1 documents will probably affect developers who are coding XML 1.1 parsers or XML 1.1 development

tools. They will probably not, however, significantly affect developers who are using XML 1.1 parsers or development tools to develop XML applications.

XML 1.1 new features

New character sets accommodation for evolving Unicode specifications form the base of new features for XML 1.1. Since the first W3C XML document recommendation was released in 1998, Unicode has expanded to accommodate much more of the alphabets and characters of the world. This was addressed to some extent in the second edition of the XML Recommendation in 2000, but the newer recommendation goes beyond the second edition to redefine what a well-formed document is, based on new Unicode standards.

Defining XML 1.1 documents

The version number in the optional XML declaration defines XML 1.1 documents, like this:

```
<?xml version="1.1">
```

Any document that does not specifically state the XML document version as 1.1 is treated as an XML 1.0 document. XML 1.1 documents are backward compatible with XML 1.0 documents. There is an exception: Some new Unicode characters that XML 1.1 processors recognize as part of well-formed element, attribute, and namespace names are not accepted by XML 1.0 document syntax rules. These characters could already be used in XML 1.0 text and attribute values. XML 1.1 officially adds these characters and character sets into structural items of XML documents — element names, attribute names, and namespaces.

XML 1.1 character sets

A more inclusive philosophy is the basis of XML 1.1. This is a reaction to the evolution of Unicode specifications, which has outpaced XML recommendation updates. Instead of the XML 1.0 approach of defining which characters cannot be included within XML documents and considering markup with undefined characters as not well formed, XML 1.1 instead defines which characters can specifically not be included in well-formed XML documents and considers any undefined characters as part of well-formed XML. This makes it easier to accommodate developing Unicode specifications. This rule applies to all XML markup, including elements, attributes, and namespaces. XML 1.0 documents will be limited to the character set defined in Unicode 2.0, and XML 1.1 documents theoretically should handle any Unicode from 2.0 to the current 3.2 and beyond.

New characters and the new philosophy will be supported by the requirement of normalization in XML 1.1 document parsed entities. This means that XML 1.1 processors that generate data will have to conform to the W3C Character Model for the

World Wide Web 1.0 (CHARMOD), currently at the "Working Draft" stage of the W3C Recommendation process, and XML 1.1. Next, the character data should be resolved into one of five formats: Cdata, CharData, content, name, or nmtoken. Parsers will have to verify normalization based on the same character model.

XML 1.1 line-end characters

Another feature of XML 1.1 is the capability to handle line-end characters generated in IBM mainframe file formats, which has been a long-standing issue between XML documents generated and shared across ASCII and EBCDIC-based platforms. XML 1.1 parsers are required to recognize and accept EBCDIC line-end characters (#x85) and the Unicode line separator (#x2028). These values should be converted to one of the XML 1.0 ASCII line-end characters-—linefeed (decimal 10, #xA), or carriage return (decimal 13, #xD).

The place that most XML developers may see and/or use the XML 1.1 line-end and character set rules will be when including hard-coded values in character or entity references. For example, if you want to hard-code a carriage return in an XML 1.0 document, the following hex character reference can be used:

```
<?xml version="1.0" encoding="UTF-8"?>
<LineEndExample>An example of a hard coded&#xD;new
line</LineEndExample>
```

The results look like this when parsed:

```
<?xml version="1.0" encoding="UTF-8"?>
<LineEndExample>An example of a hard coded
new line</LineEndExample>
```

In XML 1.1, you could also hard-code an EBCDIC value to be used on IBM mainframe systems. When parsed on non-IBM mainframe systems, the line end should be replaced with an XML 1.0 ASCII value.

```
<?xml version="1.1" encoding="UTF-8"?>
<LineEndExample>An example of a hard coded&#x85;IBM new
line</LineEndExample>
```

These results look like this when parsed:

```
<?xml version="1.1" encoding="UTF-8"?>
<LineEndExample>An example of a hard coded
IBM new line</LineEndExample>
```

Namespaces for XML 1.1The essential difference between the XML Namespaces 1.0 and 1.1 recommendations is the ability to "undeclare" a previously defined name-space declaration and its associated prefix. As with XML 1.1 updates, this is a

change that will mostly affect XML parser and development tool developers, rather than the average XML application developer.

Being able to "undeclare" a namespace provides a more flexible and efficient way of managing and reusing namespaces and their prefixes. Namespaces are applicable to any nested elements above the namespace declaration. Being able to remove a prefix and/or re-declare it in another part of a large XML document has benefits in parser performance. It also provides an out for a document that may have the same namespace prefix defined to different namespaces.

Namespaces for XML 1.1 is a separate document at the W3C but is closely linked to the XML 1.1 Recommendation. XML 1.0 documents use XML Namespace 1.0 rules, and XML 1.1 documents use XML Namespace 1.1 recommendation rules.

XML 1.1 references

More information on XML 1.1 can be found at `http://www.w3.org/TR/2002/CR-xml11-20021015/`, the namespaces for XML 1.1 working draft can be found at `http://www.w3.org/TR/2002/WD-xml-names11-20020905/`, and the CHARMOD working draft can be found at `http://www.w3.org/TR/charmod`. Also, all of the links are located on the W3C XML core Working Group page at `http://www.w3.org/XML/Activity#core-wg`.

Summary

In this chapter, I've kept the examples to a minimum to illustrate the basics of technologies that make up the XML world. The concepts introduced here will be extended with real-world examples throughout the rest of the book.

I've also introduced you to the real changes in the XML 1.1 Recommendation. These changes will affect parsers and generators and those who develop them the most. XML 1.1 parsers will probably contain normalizing and non-normalizing parser classes for conversions of line endings and character sets, just as most XML 1.0 parsers contain validating and non-validating parser classes.

- ✦ An introduction to XML
- ✦ XML structure
- ✦ Working with well-formed XML documents
- ✦ Validating XML documents
- ✦ Character and entity references
- ✦ Changes in XML 1.1 and XML Namespaces 1.1

In the next few chapters, I'll dive much deeper into XML documents, and the components that make up well-formed XML documents, by showing some real-world examples of documents, how they are generated, how they can be combined, and how namespaces can track element parts of combined documents.

✦ ✦ ✦