PART I

Oracle and the XML Standards
CHAPTER 1

Introducing XML
Extensible Markup Language (XML) is a meta-markup language, meaning that the language, as specified by the World Wide Web Consortium’s (W3C) XML 1.0 specification, enables users to define their own markup languages to describe and encapsulate data into XML files. These files can then be transformed into HTML (as well as into any other markup language) and displayed within browsers such as Netscape Navigator and Microsoft Internet Explorer, exchanged across the Internet between applications and businesses, or stored in and retrieved from databases. The power of XML comes from its simplicity, its being part of an open standard, and the incorporation of user-defined markup tags that lend semantics to the embedded data.

XML’s origins come from the Standard Generalized Markup Language (SGML)—ratified by the International Standards Organization (ISO) in 1986—on which Hypertext Markup Language (HTML), created in 1990, is based. While SGML is still a widely used standard in the document world, and HTML is still widely used as the basis of millions of web pages on the World Wide Web, XML is rapidly gaining widespread acceptance because of its advantages in data exchange, storage, and description over the existing markup languages. Since the publication of its v1.0 specifications by the W3C in February 1998, XML has been widely seen as the language and data interchange of choice for e-commerce.

What Is an XML Document?

While this book is not meant to be a full XML tutorial, as with any standard, numerous concepts and technical terms need to be explained. Because XML was developed to convey data, a relevant example is a data record of a book listing from a standard database. A complex SQL query could return data in the following format:

```
```

If XML is used as the output form, however, this record now has additional context for each piece of data, as evidenced in the following:

```
<book>
  <title>History of Interviews</title>
  <author>
    <firstname>Juan</firstname>
    <lastname>Smith</lastname>
  </author>
  <publisher>Oracle Press</publisher>
  <publishyear>2003</publishyear>
  <price type="US">10.00</price>
</book>
```

Certain items of note in this example are explored in detail later. Notice that the file has symmetry, and each piece of data has its context enclosing it in the form `<context> ... </context>`. The angle brackets and text inside are called tags, and each set of tags and its enclosed data is called an element. This relationship can be thought of as similar to a column in a database table in which the text of the tag is the column heading and the text between the tags is the data from a
row in that column. In the preceding example, **title** could be the name of the column and **History of Interviews** could be the data in a row.

Notice, too, that several tags contain tags instead of data. This is a significant feature of XML, which permits nesting of data to define relationships better. Returning to the database metaphor, the `<author>` tag could be modeled as a table whose columns were `<firstname>` and `<lastname>`.

In XML terminology, these column tags are referred to as children of the parent `<author>` tag.

Now look at the `<price>` tag and you see that it includes text of the form `name="value"`. These name-value pairs are called attributes, and one or more of these can be included in the start tag of any element. Attributes, however, are not legal in end tags (for example, `</tag name="foo">`). Notice that attribute values **must** be framed by quotes (single or double, as long as the closing and opening quotes are the same) as specified by SGML. HTML is much more permissive in this area.

One final terminology note: the entire XML example is enclosed by `<book> … </book>`. These tags are defined as the root of the document, and only one may exist in any particular document. XML documents that follow these rules of having only one root and properly closing all open tags are considered **well formed**.

XML's basic concepts and terminology are straightforward and are formalized in an open Internet standard. As the W3C XML 1.0 specification states, “XML documents are made up of storage units called *entities*, which contain either parsed data or unparsed data. Parsed data [or PCDATA] is made up of characters, some of which form *character data*, and some of which form *markup*. Markup encodes a description of the document’s storage layout and logical structure.” XML documents have both physical and logical structure. The physical structure of the XML document simply refers to the XML file and the other files that it may import, whereas the *logical structure* of an XML document refers to the prolog and the body of the document.

The XML of the book example represents the body of an XML document, but it is missing important information that helps identify its nature. This information is in the prolog, discussed in the following section.

**The Prolog**

The *prolog* consists of the XML declaration (that is, the version number), a possible language encoding hint, other attributes (name-value pairs), and an optional grammar or data model specified by either an *XML Schema Definition* (XSD) or a *Document Type Definition* (DTD) referred to by a URL. The prolog may also contain the actual XSD or DTD. An example with a reference to an external DTD would look like the following:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE book SYSTEM "book.dtd">```

Note that a line containing `<? ... ?>` is an example of an XML *processing instruction* (PI). In this example, `xml` is the name of the XML PI. In addition, the character set encoding supported in the example is a compressed version of Unicode called *UTF-8*. While XML processors usually detect the encoding from the first 3 bytes in the file, this declaration can be used as a hint to indicate the expected encoding. Finally, the `standalone` attribute refers to whether the processor needs to include or import other external files.

The second line of this prolog refers to a **DOCTYPE**. This is where the declaration of the grammar or data model for this XML document is done. Why is this important? Remember, an
XML file has both physical and logical representations. In some applications, it may be sufficient to process the XML without knowing whether information is missing, but most of the time, an application wants to validate the XML document it receives to confirm everything is there. To do this, the application must know which elements are required, which ones can have children, which ones can have attributes, and so forth. In XML terms, the grammar or data model in this example is referred to as DTD. This DTD can reside within the XML file itself or simply be referred to so that the processor can locate it, as in this example.

The preceding example might look as follows with an XML Schema declaration:

```xml
<?xml version="1.0"/>
<xsd:schema xmlns:xsd=http://www.w3.org/2001/XMLSchema
xmlns:bk="http://www.mypublishsite.com/books">

To begin with, note that the XML Schema declaration has a prefix `xsd:`; which is associated with the XML Schema namespace through the declaration `xmlns:xsd="http://www.w3.org/2001/XMLSchema"`. This prefix is used on the names of the data types defined in the referenced XSD to differentiate them from others using the same name. The `xsd:schema` declaration denotes the beginning of this XML Schema incorporated in this XML document, along with one other declaration, `xmlns:bk="http://www.mypublishsite.com/books"`, which defines the namespace of the prefix `bk:` so as to identify these types as defined by the author of this data model.

Note also that the schema declaration is within the `<book>` tag instead of in the prolog. This is a distinct difference between XSDs and DTDs. Thus, the XML schema declaration is an attribute of the root element of the document and is part of the body, which we discuss next.

The Body

The root element, which contains the remainder of the XML document, follows the prolog and is called the body of the XML document. This part is composed of elements, processing instructions, content, attributes, comments, entity references, and so forth. As previously mentioned, elements must have start tags and corresponding end tags nested in the correct order; otherwise, the XML document is not well-formed, and XML parsers may signal errors because of this. Elements can also have attributes, or name-value pairs, such as `<author firstname="Juan" lastname="Smith">`. Built-in attributes defined by the XML 1.0 specification also exist, such as `xmlns:space="preserve"` to indicate that the whitespace between the elements be considered as data and thus preserved.

Entity references, defined only in DTDs, are similar to macros in that entities are defined once, and references to them, such as `&nameofentity`, can be used in place of their entire definitions. For example, in a DTD, `<!ENTITY Copyright "Copyright 2000 by Smith, Jones, and Doe – All rights reserved">` could be declared, and then `&Copyright` could be used as a shortcut throughout the XML document. An XML parser must recognize entities defined in DTDs, even though the validity check may be turned off and an additional XML Schema is specified. Again, built-in entities also exist as defined by the XML 1.0 specifications, such as those for the ampersand, `&amp;`; apostrophe, `&apos;`; less than, `&lt`; and so forth. Comments are recognized when they are enclosed in the `<!-- -->` construct.

Within the body of the XML document instance, certain element and attribute names may have prefixes, which are XML namespaces identified by Uniform Resource Identifier (URI) references that qualify the names of these elements and attributes and locate resources that could
be on different machines or XML documents. For example, if the declaration `xmlns:bk="http://www.mypublishsite.com/books"` is made in a parent element, the prefix `bk:title` stands for `http://www.mypublishsite.com/books:title`. You can use identical names for either elements or attributes if they are qualified with URIs to differentiate the names. For example, `bk:hello` is called a qualified name; the namespace prefix `bk` is mapped to the URI, `http://www.mypublishsite.com/books`, and the local part is `hello`. Note that URI references can contain characters not allowed in element names; that is why `bk` serves as a substitute for the URI. It is important to mention that the `bk` prefix belongs to the document in which it is declared. Another document declaring the prefix `book` instead of `bk` but referencing the same URI would be considered equivalent when parsed by an XML parser.

Finally, the body may contain character data (CDATA) sections to mark off blocks of text that would otherwise be regarded as markup, comments, entity references, processing instructions, and so forth. The CDATA syntax is

```
<![CDATA[ characters including <, >, /, ?, & not legal anywhere else ]]>
```

These sections are simply skipped by XML parsers as if they were opaque. Later in the book, you will see how you use them to embed SQL statements in XML documents.

Thus, the body of the XML document contains the root element with its schema declarations, child and sibling nodes, elements, attributes, text nodes that represent the textual content of an element or attribute, and CDATA sections.

## Well-Formed XML Documents

As mentioned previously, an XML document is well formed if only one root exists and all start tags have corresponding end tags, with the correct nesting. For example, the following is not well formed:

```
<bookcatalog>
  <book>
    <title>History of Interviews</title>
    <author>
      <firstname>Juan</firstname>
      <lastname>Smith</lastname>
    </author>
    <isbn>99999-99999</isbn>
    <publisher>Oracle Press</publisher>
    <publishyear>2003</publishyear>
    <price type="US">10.00</price>
  </book>
</bookcatalog>
```

The following are the reasons why it is not well formed:

- Two roots exists, `bookcatalog` and `bookcatalog2`.
- The `<title>` tag does not have a correct corresponding end tag, as in `</title>`.
Valid XML Documents

A valid XML document is one that conforms to either a specified DTD or XML Schema, meaning that the elements, attributes, structural relationships, and sequences in the XML document are the same as the ones specified in the DTD or XML Schema. For example, the following XML is valid with respect to the DTD, which follows it:

```xml
<bookcatalog>
  <book>
    <title>History of Interviews</title>
    <author>
      <firstname>Juan</firstname>
      <lastname>Smith</lastname>
    </author>
    <publisher>Oracle Press</publisher>
    <publishyear>2003</publishyear>
    <price type="US">10.00</price>
  </book>
</bookcatalog>
```

The following is the DTD to which the XML document conforms:

```xml
<!-- DTD bookcatalog may have a number of book entries -->
<!DOCTYPE bookcatalog [  
  <!ELEMENT bookcatalog (book)*>  
  <!-- Each book element has a title, 1 or more authors, etc. -->  
  <!ELEMENT title (#PCDATA)>  
  <!ELEMENT author (firstname, lastname)>  
  <!ELEMENT firstname (#PCDATA)>  
  <!ELEMENT lastname (#PCDATA)>  
  <!ELEMENT ISBN (#PCDATA)>  
  <!ELEMENT publisher (#PCDATA)>  
  <!ELEMENT publishyear (#PCDATA)>  
  <!ELEMENT price (#PCDATA)>  
  <!ATTLIST price type (US|CAN|UK|EURO) #REQUIRED>  
]>
```

The **DOCTYPE** declaration of the DTD specifies the root element—in this case, the **bookcatalog** element. An element simply consists of a start tag, for example, `<title>`; all of the text in between, **History of Interviews**; and the corresponding end tag, for example, `</title>`. Only one root element, however, may exist within an XML document. The root element marks the beginning of the document and is considered the parent of all the other elements, which are
nested within its start tag and end tag. For XML documents to be considered valid with respect to this DTD, the root element `bookcatalog` must be the first element to start off the body of the XML document.

Following this are the **element declarations**, which stipulate the child elements that must be nested within the root element `bookcatalog`, the content model for the root element. Note that all the child elements of `bookcatalog` are explicitly called out in its element declaration, and that `author` has a `+` as a suffix. This is an example of the Extended Backus-Naur Format (EBNF) that can be used to describe the content model. The allowed suffixes are

- `?` For 0 or 1 occurrence
- `*` For 0 or more occurrences
- `+` For 1 or more occurrences
- No suffix means 1 and only 1.

Note also the use of `#PCDATA` to declare that the element text must not be marked-up text, and that `price`’s required attribute values are explicitly declared. The difference between `CDATA` and `PCDATA` is that `CDATA` sections are simply skipped by the parser and aren’t checked for well-formedness; hence, they can be viewed as “non-parsed character data.”

Thus a validating XML parser, by parsing the XML document according to the rules specified in this DTD, tries to determine whether the document conforms to the DTD (is valid), meaning that all the required elements, attributes, structural relationships, and sequences are as declared.

### XML Namespaces

Earlier in the chapter, we introduced XML namespaces. This W3C XML standard introduces the following terms with regard to XML namespaces:

- **Local name** Represents the name of the element or attribute without the prefix. In the previous example, `book`, `title`, `author`, `ISBN`, and so forth are considered local names. These are used whenever there is no concern over duplicate tag or attribute names. `Local name` is also used to refer to the name part of a qualified name.

- **Qualified name** Represents the fully prefixed name. For example, as a continuation of the previous examples, `bk:title`, `bk:book`, and so forth are considered qualified names. Qualified names are being used more often because XML Schemas are defining standard types, such as address, customer, purchase order, and so on, and there is a need to differentiate semantics.

- **Namespace prefix** Represents the namespace prefix declared using the special prefix, `xmlns`. The previous example defined one namespace prefix: `bk`. Prefixes are scoped and thus must be unique within the children of the parent element that declared the namespace, but prefixes may be overridden by a new declaration on a descendent element or attribute.

- **Expanded name** Represents the result of applying the namespace defined by the namespace prefix to the qualified name. For example, `bk:booklist` could be expanded to `http://www.mypublishsite.com/books:booklist`. The expanded name is never seen in the XML document itself, but is conceptually important.
Two kinds of namespace attributes exist: prefixed and default. A prefixed namespace attribute is of the form `nsprefix:attr`, where `nsprefix` is the namespace prefix defined previously. Once a prefix has been declared, it can be used to specify a namespace for any elements or attributes in the scope of the element where it was declared. You would, therefore, need to declare global prefixes—that is, prefixes you want to use everywhere in your document—as attributes of the root element.

The default namespace attribute is `xmlns`. `xmlns` has the effect of specifying a default namespace for the entire scope of an element (including the element itself). This default does not apply to the attributes in the subtree, however. For example, consider the following example:

```xml
<booklist xmlns="http://www.osborne.com/books">
  <book isbn="1234-5678-1234">
    <title>Oracle XML Handbook</title>
    <author>Oracle XML Team</author>
  </book>
  <book isbn="24345-564478-1344234">
    <title>The C programming language</title>
    <author>Kernighan and Ritchie</author>
  </book>
</booklist>
```

This root element declaration has the effect of specifying that all the elements under `booklist` (book, title, author) are in the `http://www.osborne.com/books` namespace. The attribute `isbn`, however, is not. Default namespaces can be specified at any level of the document and have the effect of overriding previous declarations. Setting `xmlns=""` has the effect of removing the default namespace declaration for a particular document subtree.

Namespaces complicate the determination of attribute uniqueness. For example, consider the following example:

```xml
<booklist xmlns:dollars="USA" xmlns:pounds="Britain">
    <title>The Code of the Woosters</title>
    <author>P.G. Wodehouse</author>
  </book>
</booklist>
```

The two `price` attributes should be considered different, even though they have the same local name, because their expanded names are different. The following document would not be considered well-formed, however:

```xml
<booklist xmlns:dollars="USA" xmlns:currency="USA">
    <title>The Code of the Woosters</title>
    <author>P.G. Wodehouse</author>
  </book>
</booklist>
```
Here, even though $dollars\cdot price$ and $currency\cdot price$ have different qualified names, they have the same expanded name, which means they are, in fact, the same attribute declared twice on the book element. For a similar reason, only one default namespace is allowed per document.

**XML and the Database**

Databases and XML offer complementary functionality for storing data. Whereas databases store data for efficient retrieval, XML offers an easy information exchange that enables interoperability between applications due to its ability to encapsulate the data with the metadata. Oracle8i, Oracle9i, and, to an even larger extent, Oracle Database 10g enable you to store XML natively and build XML-enabled applications. Storing XML collections in databases enables you to benefit from not only the full power of a SQL engine but also database administration, business intelligence, and recovery tools and procedures, such as Oracle Enterprise Manager, Discoverer, and RMAN. You can use them to enforce rules about data and security, and to block operations that compromise data integrity by embedding rules and logic in a database. Also, converting database tables into XML documents enables you to take advantage of XML’s features while preserving SQL data types, indexes, and enterprise-level scalability. You can present XML documents as HTML pages with XSLT stylesheets, search them using XML-based query languages, or use them as a data-exchange format.

Oracle’s object-relational features have been extended to support hierarchical storage thus enabling you to capture the complex structure of XML data. You can operate and manage XML data on a desired level of granularity which lends itself readily to efficiently construct dynamic XML documents from the stored fragments. You can also store XML documents in a new XMLType data type that supports storing XML as a single document in a virtually unbounded text data type called **Character Large Object (CLOB)**, as data without tags distributed in object-relational tables or as both. You can use Oracle Text to perform searches on XML documents stored in CLOBs.

**NOTE**

A CLOB is one of the Oracle internal Large Objects (LOBs) whose value is composed of character data and can store up to 4GB of data.

Meanwhile, a VARCHAR2 column in a table has a limit of 4000 bytes, and a VARCHAR2 in a PL/SQL variable has a limit of 32767 bytes (32K).

CLOBs can be indexed to search the XML as plain text or as document sections for more precise searches. For example, you can find Oracle WITHIN <title>, ignoring it elsewhere in the document. Oracle Text also provides full-text indexing of documents and the capability to do SQL queries over documents, along with XPATH-like searching. Finally, Oracle’s Advanced Queuing (AQ) now supports XML-based message queuing in the database, supporting both synchronous and asynchronous communications of XML messages defined in the standard **Simple Object Access Protocol (SOAP)** format for both the server and client.

**Database Schema and XML Documents**

XML documents consist of text that conforms to a hierarchy or tree structure specified by a DTD or XML Schema. As distinct from other strictly relational databases, you can easily store this
hierarchical data in an optimal internal form using Oracle’s object-relational tables, which serve
as the foundation for the native XMLType storage. All the existing and future internal applications
can work with the information in the most efficient way possible. When you retrieve information,
for sharing with partners or other applications, you can present the appropriate view of data and
document content specific to the task at hand as integrated XML. These XMLType views enable
you to present data in any number of “logical” combinations, hiding any details of their underlying
physical storage. You can effectively transform the structure of one or more underlying tables into
a more useful or more appropriate structure for the demands of a specific application. When you
link views of information with other views of related information, they quite naturally form “trees”
or “graphs” of related data. When you represent database information as XML, the previous
related views provide the foundation for many different tree-structured XML documents.

Here, we offer a simple example of how a database table would be expressed as an XML DTD:

```xml
<!DOCTYPE table [
  <!ELEMENT table (rows)>
  <!ELEMENT rows (column1, column2, ...)>  
  <!ELEMENT column1 (#PCDATA)>  
  <!ELEMENT column2 (#PCDATA)>  
  ...]>
```

Note, however, that the actual data types in these columns remain unspecified. We can instead
use an XML Schema of the following form:

```xml
<xs:element name="table">
  <xs:sequence>
    <xs:element name="rows">
      <xs:sequence>
        <xs:element name="column1" type="xs:integer" xdb:SQLType="NUMBER"/>  
        <xs:element name="column2" type="xs:string" xdb:SQLType="CLOB"/>
        ...  
      </xs:sequence>
    </xs:element>
  </xs:sequence>
</xs:element>
```

This does provide us with the column data types through the `type` attribute which can
further constrain the data as we will discuss later in the database chapters. However, a database
provides even more capability to express rules than does a DTD or an XML Schema. The
database schema defines type information and constraints—not only simple constraints, such as
permissible value ranges, but also constraints between columns and tables. A database schema
enables you to define relationships or dependencies. For example, your e-commerce business
might receive orders as XML documents. By using a database, you can link customer and order
information, and define a rule about not processing orders for closed accounts. In spite of the
limitations in DTDs or XML Schemas, mapping a database schema to a DTD or an XML Schema
presents the database as a virtual XML document to the tools that need XML documents as input.

### Mapping XML Documents to a Database Schema

When mapping XML documents associated with an XML DTD or an XSD to a database schema
for the purpose of storing XML in Oracle, three basic strategies exist:

- Map the complete XML document as a single, intact object, such as an XMLType CLOB.
Store the data in relational or object-relational tables and create XMLType views over the data to present it as an XML document.

Map XML documents to an Oracle Native XMLType.

You can choose one of the previous approaches, depending on the structure of the XML document and the operations performed by the application. Each of these three approaches is described in turn next. You can also store the XML DTD or Schema in the database to validate the XML documents.

XML Documents Stored As XMLType CLOBs
Storing an intact XML document in an XMLType CLOB is a good strategy if the XML document contains static content that will only be retrieved as a whole or updated by replacing the entire document. Examples include written text such as articles, advertisements, books, legal briefs, and contracts. Applications that use a repository of this nature are known as document-centric and operate on the stored XML outside the database. Storing this kind of document intact within OracleX gives you the advantages of an industry-proven database and its reliability over file system storage. Upon insertion, XML documents are checked and only committed if well formed. Oracle Text can provide both content and path indexes to search, but data retrievals need to be done by processing the whole document. The Oracle XML Developer’s Kit (XDK) provides the functionality to use standards-based interfaces to access, modify, transform, and validate these documents.

XMLType CLOBs can also be used in conjunction with XMLType views and the Native XMLType described in the following sections. In these cases, the XML is a well-formed fragment that is treated as a whole.

XML Documents Stored As XMLType Views
When an application is using XML merely as an encapsulation of its data, it is considered to be data-centric. Typically, the XML document contains elements and attributes that have complex structures, but in reality this structure is simply metadata to convey the actual data of interest. Examples of this kind of document include sales orders, invoices, and airline flight schedules. In this case, there is value to maintaining the storage as SQL data because the actual data types need to be exposed to the application. Oracle Database 10g, with its XMLType object-relational extensions, has the capability to capture the structure of the data in the database present it in an XMLType view while still you can easily update, query, rearrange, and reformat as needed using SQL. The important distinction to remember is that this view cannot convey document order, comments, processing instructions, or the whitespace between elements and attributes that are preserved in the CLOB type.

Using XMLType views is especially useful when you are XML-enabling existing applications or database schemas. This view can serve to abstract the underlying database schema, thereby eliminating the need to modify it to support XML. In fact, since you can have multiple views of differing XML structure over the same database schema, you can directly support a different XML schema for each view without the need to apply XSL to transform the documents to a structure compatible with your database schema.

Applications built against XMLType views have the flexibility of using SQL and SQL-XML interfaces to process within the database and using the XDK to process either in JServer, middle tier, or client.
XML Documents Stored As Native XMLTypes

Finally, you can store an XML document as a Native XMLType, where the underlying storage is dictated and created by its XML schema. This type has the advantages of the other two types together because it stores the XML as SQL data and preserves byte-for-byte fidelity. Creating this native type is as simple as registering its XML schema, which not only creates the underlying database schema but also creates a database resource that can be used for access and updates with the Internet-standard protocols HTTP, FTP, and WebDAV.

Applications that need both a document and data view can make full use of this type, because an extensive array of SQL, PL/SQL, Java, C, and C++ interfaces is available. Inserts, updates, and deletes are simplified due to Native XMLType’s support for query rewrites, thereby eliminating the triggers needed by the XMLType views. The Native XMLType can, however, be used in conjunction with these views to expose differing or subset XML documents. To further support large documents, a “virtual” or “lazy” DOM is provided through the XDK to access only those elements that are needed at any one time.

While the native XMLType exposes broad functionality, you need to remember that the underlying storage is intimately tied to the XML schema that created it. Therefore, it will most likely not be the best choice for applications that need to support multiple schemas or a nontrivial evolving schema.

Summary

Oracle provides you with differing strategies to map XML documents into a database schema. Since XML is an enabling technology and not an application, there is no one strategy that will work optimally for all scenarios. XML is an abstraction born from the need to exchange content or data in an interoperable manner. XML’s cost is the increased overhead to process its structure and tags, thus your correct selection of XML storage model will have a significant performance impact on your application. In later chapters, we will walk you through the process of using each of these strategies with appropriate example applications.