

# CHAPTER 3

## XML—the *X* in Ajax

### Chapter Objectives

- Introduce the motivation and importance of XML technologies
- Explain the major technologies for defining XML dialects
- Illustrate how to parse, validate, and process XML documents
- Introduce XSL and XSLT for transforming XML documents

### 3.1 Overview

For two information systems to integrate smoothly, they must either adopt the same data structures or have the ability to interpret their partners' data accurately. This is the data integration challenge, the focus of this chapter.

Different business data types have different logical structures. For example, hospital patients' medical records have a totally different structure from bank transaction records. For efficient processing, each business data type must be represented in some well-defined format. Such data representations are not unique, and the different businesses may represent the same type of business data in different data formats. For example, different hospitals may represent their patients' medical records in different data formats. Such inconsistencies could lead to difficulties in integrating the information systems of the cooperating businesses.

On the other hand, for computers to process and store a type of business data, each information system needs to implement the data format adopted by the business in a particular programming language. Different programming languages may have different data specification mechanisms and data types for specifying the same type of business data. If two cooperating information systems are implemented with different programming languages, they could have difficulties in processing their partners' data. The properties of computer hardware, operating systems, and networking protocols could also complicate such data integration.

*XML*, or *Extensible Markup Language*, is a technology introduced mainly for business data specification and integration. XML is a simplified descendant of *SGML*, or *Standard Generalized Markup Language*. Like XHTML/HTML, XML uses tags and attributes to mark up data. But XML is generic and does not have a specific application domain. It does not limit what tag or attribute names can be used. For different types of business data you may need to define different concrete markup languages to define their data structures. Each of these concrete markup languages needs to follow the general syntax structure of XML but uses a set of tag and attribute names predefined with XML syntax specification mechanisms like *DTD* (*Document Type Definition*) or *XML Schema*, which will be introduced in the following sections. Because of this, people usually say that XML is a metalanguage for defining markup languages in specific application

domains, and these markup languages are called *XML dialects* and can use only predefined tags and attributes. Each XML dialect document is a special case of an XML document and is called an *instance document* of the XML dialect specification. The popular XML dialects include XHTML for specifying web page structures, SOAP (originally standing for *Simple Object Access Protocol* and more recently for *Service-Oriented Architecture Protocol*) for specifying message structures representing remote method invocations, and BPEL (Business Process Execution Language) for specifying business processes.

For a particular type of business data, different information systems may have different specification mechanisms for their logical structure. An XML dialect could be defined and adopted by the cooperating systems and become an intermediate language for data exchange among these systems. For a system to exchange data with its partners, it only needs to have the ability to transform data between its proprietary format and the accepted XML dialect format. Because XML processing functions have been integrated into most operating systems and are freely available, such XML-based data integration is cost-effective.

This chapter first introduces the syntax of basic XML documents. DTD and XML Schema mechanisms are then used to define XML dialects for specifying logical data structures. The XSL (Extensible Stylesheet Language) and XSLT (XSL Transformation) techniques will then be introduced to transform XML documents into other data formats.

## 3.2 XML Documents

An XML document contains an optional *XML declaration* followed by one top-level element, which may contain nested elements and text, as shown by the following example (the contents are in file “dvd.xml”):

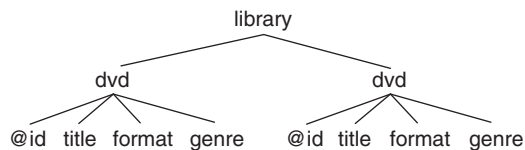
```
<?xml version="1.0" encoding="UTF-8"?>
<!-- This XML document describes a DVD library -->
<library>
  <dvd id="1">
    <title>Gone with the Wind</title>
    <format>Movie</format>
    <genre>Classic</genre >
  </dvd>
```

```

<!--
<dvd id="2">
  <title>Star Trek</title>
  <format>TV Series</format>
  <genre>Science fiction</genre>
</dvd>
</library>

```

This XML document starts with an optional XML declaration. The second line is an example of an *XML comment*, which always starts with `<!--` and ends with `-->`. Such comments can occur anywhere and continue over multiple lines in an XML document, and they are ignored by XML processors. The main content of this example XML document is an element named `library`, which includes two nested elements named `dvd`. Each `dvd` element in turn contains three elements named `title`, `format`, and `genre`. Each `dvd` element also contains an attribute `id`, which specifies a unique ID number. The nesting structure of such an XML document can be described by the following tree that grows downward. Here `library` is called the root, or top-level, element. Prefix `@` is used to indicate that the following name is for an attribute.



### 3.2.1 XML Declaration

If it is used, the optional XML declaration must be the first line of an XML document. It declares the XML version and character encoding of the following XML document. Different versions of the XML specification have different capabilities and features (backward compatible), and in 2008 the latest XML version is 1.1 and the most popular version is 1.0. If an XML document does not have an XML declaration, the XML processors will assume the document is based on some default XML version and character encoding. Because such defaults are not standardized, it is much safer to declare them so that the XML processors will process the XML documents with predictable behavior.

### 3.2.2 Unicode Encoding

XML data are based on *Unicode* (<http://unicode.org>), an industry-standard character coding system designed to support the worldwide interchange,

processing, and display of the written texts of the diverse languages and technical disciplines of the modern world. The Unicode standard assigns unique integers, called *code points*, to characters of most languages, as well as defines methods for storing the integers as byte sequences in a computer. There are three approaches, named UTF-8, UTF-16, and UTF-32, where UTF stands for Unicode Transformation Format. UTF-8 stores each Unicode character as a sequence of one to four 8-bit values (1 byte for the 128 US-ASCII characters, 2 or 3 bytes for most of the remaining characters, 4 bytes for some rarely used characters), and it is the most space-efficient data encoding method if the data is based mainly on the US-ASCII characters, as for English.

### 3.2.3 Tags, Elements, and Attributes

Each *XML element* consists of a *start tag* and an *end tag* with nested elements or text between. The matching start tag and end tag must be based on the same tag name, which is also called the element name. The nested elements or text between the matching start and end tags are called the *value of the element*. The start tag is of form `<tagName>`, and the end tag is of form `</tagName>`. For example, `<format>Movie</format>` is an element, its start tag is `<format>`, its end tag is `</format>`, the element is based on tag name `format`, so it is also called a `format` element. This `format` element has text `Movie` as its value. Any string consisting of a letter followed by an optional sequence of letters or digits and having no variations of “xml” as its prefix is a valid XML tag name. Tag names are case sensitive. An element that is not nested in another element is called a *root*, or *top-level*, element. By specification, an XML document can have exactly one root element.

If an element has no value, like `<tagName></tagName>`, it can be abbreviated into a more concise form, `<tagName/>`.

Elements can be nested. In the preceding example, `title`, `format`, and `genre` elements are nested inside `dvd` elements, which are in turn nested in a `library` element. Elements cannot partially overlap each other. For example, “`<a><b>data</a>data</b>`” contains two partially overlapping `a` and `b` elements and thus is not allowed in a valid XML document. For avoiding partial element overlapping, the element starting first must end last.

The start tag of an element may contain one or more attribute specifications, in the form of a sequence of `attributeName="attributeValue"` separated by white spaces, as in `<dvd id="1">`, where the `dvd` element has attribute

id, with its value being 1. Attribute values must be enclosed in either matching single straight quotes (') or matching double straight quotes ("). Any string consisting of a letter followed by an optional sequence of letters or digits can be a valid attribute name. Although most information of an XML document is in the form of element values, attributes are usually used for specifying short categorizing values for the elements.

### 3.2.4 Using Special Characters

The following five characters are used for identifying XML document structures and thus cannot be used in XML data directly: &, <, >, “, and ‘. If you need to use them as values of XML elements or attributes, you need to use `&amp;` for &, `&lt;` for <, `&gt;` for >, `&quot;` for “, and `&apos;` for ‘. These alternative representations of characters are examples of *entity references*.

As an example, the following XML element is invalid:

```
<Organization>IBM & Microsoft</Organization>
```

whereas the following is valid XML data:

```
<Organization>IBM &amp; Microsoft</Organization>
```

If your keyboard does not allow you to type the characters you want, or if you want to use characters outside the limits of the encoding scheme that you have chosen, you can use a symbolic notation called *entity referencing*. If the character that you need to use has hexadecimal Unicode code point `nnn`, you can use syntax `&#xnnn`; to represent it in XML documents. If the character that you need to use has decimal Unicode code point `nnn`, you can use syntax `&#nnn`; to represent it in XML documents. If you use a special character multiple times in a document, you could define an entity name for it in DTD, which will be covered in Section 3.3.3 of this chapter, for easier referencing. An entity assigns a string name to an entity reference. For example, if your keyboard has no Euro symbol (€), you can type `&#8364`; to represent it in XML documents, where 8364 is the decimal Unicode code point for the Euro symbol. If you need to use the Euro symbol multiple times in a document, you can define an entity name, say, `euro`, through a DTD declaration `<!ENTITY euro "&#8364;">` (more explanation will be available in the section on DTD). Then you can use the more meaningful entity reference `&euro`; in your XML document. In general, if there is an entity name `ccc` for a character, you can replace it with syntax `&ccc`; in XML documents. Entity names “amp”, “lt”, “gt”, “quot” and “apos” are predefined for

&, <, >, “ and ‘, respectively, and you can use them in your XML documents without declaring them with DTD. Table 2.2.1 on page 30 listed the popular HTML entities. The entity numbers in the third column can be used in XML documents too, but only the first five entity names are predefined in XML specifications.

### 3.2.5 Well-Formed XML Documents

A well-formed XML document must conform to the following rules, among others:

- Nonempty elements are delimited by a pair of matching start tags and end tags.
- Empty elements may be in their self-ending tag form, such as <tagName/>.
- All attribute values are enclosed in matching single (‘) or double (“) quotes.
- Elements may be nested but must not partially overlap. Each non-root element must be completely contained in another element.
- The document complies with its declared or default character encoding.

Both the SAX and DOM XML parsers will check whether the input XML document is well formed. If it is not, the parsing process will be terminated with error messages.

## 3.3 Document Type Definition (DTD)

DTD is the first mechanism for defining XML dialects. As a matter of fact, it is part of the XML specification 1.0, so all XML processors must support it. But DTD itself does not follow the general XML syntax. The following is an example DTD declaration for the earlier DVD XML document example (the contents are in example file dvd.dtd):

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT library (dvd+)>
<!ELEMENT dvd (title, format, type)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT format (#PCDATA)>
<!ELEMENT genre (#PCDATA)>
<!ATTLIST dvd id CDATA #REQUIRED>
```

Because DTD is part of the XML specification, you should start its declarations with the XML declaration for XML version and character encoding.

### 3.3.1 Declaring Elements

To declare a new XML element or tag name (element type) in an XML dialect, use the following syntax:

```
<!ELEMENT elementName (elementContent)>
```

#### 3.3.1.1 Empty Elements

Empty elements are declared with the keyword `EMPTY` in parentheses:

```
<!ELEMENT elementName (EMPTY)>
```

As an example, `<!ELEMENT br (EMPTY)>` declares that `br` is an empty element.

#### 3.3.1.2 Elements with Text or Generic Data

Elements with text or generic data are declared with the data type in parentheses in one of the following forms:

```
<!ELEMENT elementName (#CDATA)>
```

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

```
<!ELEMENT elementName (ANY)>
```

`#CDATA` means that the element contains character data that is not supposed to be parsed by a parser for markups like entity references or nested elements.

`#PCDATA` means that the element contains data that *is* going to be parsed by a parser for markups, including entity references, but not for nested elements.

The keyword `ANY` declares an element with any content as its value, including text, entity references, and nested elements. Any element nested in this element must also be declared. `ANY` is used mainly during the development stage of an XML dialect; it should not be used in the final XML dialect specification.

As an example, `<!ELEMENT index (#PCDATA)>` declares a new `index` element type so that the XML parsers will further identify markups, excluding nested elements, in values of this type of element.

#### 3.3.1.3 Elements with Children (Sequences)

An element with one or more nested child elements as its value are defined with the names of the child elements in parentheses:

```
<!ELEMENT elementName (childElementNames)>
```

where `childElementNames` is a sequence of child element names separated by commas. These children must appear in the same sequence in XML documents adopting this DTD declaration. In a full declaration, the child elements must also be declared, and the children can also have children.

As an example, `<!ELEMENT index (term, pages)>` declares an element type named `index` whose value contains a `term` element and a `pages` element in the same order.

As another example, `<!ELEMENT footnote (message)>` declares an element type named `footnote` that can only contain exactly one `message` element as its value.

For declaring zero or more occurrences of the same element as the value of a new element type, use syntax

```
<!ELEMENT elementName (childName*)>
```

Here symbol `*` indicates that the previous element should occur zero or more times, a notation originally adopted by *regular expressions*.

For example, `<!ELEMENT footnote (message*)>` declares that elements of type `footnote` should contain zero or more occurrences of `message` elements.

If you change the symbol `*` to symbol `+` in the preceding syntax, then elements of the new element type should have one or more occurrences of the child element.

For declaring zero or one occurrence of an element as the value of a new element type, use syntax

```
<!ELEMENT elementName (childName?)>
```

Here the `?` symbol declares that the previous element can occur zero or one time, also a notation originated from regular expressions.

For example, `<!ELEMENT footnote (message?)>` declares that a `footnote` element should contain either elements or one `message` element as its value.

If an element can contain alternative elements, you can use the pipe symbol, `|`, to separate the alternatives. For example, DTD declaration

```
<!ELEMENT section (section1 | section2)?>
```

specifies that a `section` element contains either a `section1` element or a `section2` element, but not both.

### 3.3.1.4. Declaring Mixed Content

For an example, look at declaration

```
<!ELEMENT email (to+,from,header,message*,#PCDATA)>
```

The preceding example declares that an `email` element must contain in the same order at least one `to` child element, exactly one `from` child element, exactly one `header` element, zero or more `message` elements, and some other parsed character data as well.

### 3.3.2 Declaring Attributes

In DTD, XML element attributes are declared with an `ATTLIST` declaration. An attribute declaration has the following syntax:

```
<!ATTLIST elementName attributeName attributeType defaultValue>
```

As you can see from the preceding syntax, the `ATTLIST` declaration specifies the element that can have the attribute, the name of the attribute, the type of the attribute, and the default attribute value.

The attribute-type can have values including the following:

Value	Explanation
CDATA	The value is character data.
(eval1 eval2 ...)	The value must be one of the enumerated values.
ID	The value is a unique ID.
IDREF	The value is the ID value of another element.
ENTITY	The value is an entity.

The attribute default-value can have the following values:

Value	Explanation
Default-value	The attribute is optional and has this default value.
#REQUIRED	The attribute value must be included in the element.
#IMPLIED	The attribute is optional.
#FIXED value	The attribute value is fixed to the one specified.

DTD example:

```
<!ELEMENT circle EMPTY>
<!ATTLIST circle radius CDATA "1">
```

XML example:

```
<circle radius="10"></circle> <circle/>
```

In the preceding example, the element `circle` is defined to be an empty element with the attribute `radius` of type `CDATA`. The `radius` attribute has a default value of 1. The first `circle` element has radius 10, and the second `circle` element has the default radius 1.

If you want to make an attribute optional but you do not want to provide a default value for it, you can use the special value `#IMPLIED`. In the preceding example, if you change the attribute declaration to

```
<!ATTLIST circle radius CDATA #IMPLIED>
```

then the second `circle` element will have no `radius` value.

On the other hand, if you change the preceding attribute declaration to

```
<!ATTLIST circle radius CDATA #REQUIRED>
```

then the second `circle` element is not valid and will be rejected by XML validating parsers because it misses a required value for its `radius` attribute.

If you change the preceding attribute declaration to

```
<!ATTLIST circle radius CDATA #FIXED "10">
```

then all `circle` elements must specify 10 as their `radius` value, and the second `circle` element is not valid and will be rejected by XML validating parsers because it misses the required value 10 for its `radius` attribute.

The following line declares a type attribute for `circle` elements

```
<!ATTLIST circle type (solid|outline) "solid">
```

which can take on either `solid` or `outline` as its value. If a `circle` element does not have a `type` attribute value specified, it would have the default type value `solid`.

### 3.3.3 Declaring Entity Names

An entity name can be declared as a nickname or shortcut for a character or a string. It is used mainly to represent special characters that must be specified with Unicode or long strings that repeat multiple times in XML documents.

To declare an entity name, use the following syntax:

```
<!ENTITY entityName "entityValue">
```

where `entityName` can be any string consisting of a letter followed by an optional sequence of letters or digits. The following two declarations define “euro” as an entity name for the Euro symbol (€) and “cs” as an entity name for string “Computer Science”.

```
<!ENTITY euro "&#8364;">  
<!ENTITY cs "Computer Science">
```

XML documents can use syntax `&entityName;` in their text to represent the character or string associated with `entityName`. For example, if an XML document includes the preceding two DTD declarations, then `&euro;` and `&cs;` in its text will be read by XML parsers as the same as € and Computer Science. Table 2.2.1 on page 30 listed the popular HTML entities. The entity numbers in the third column can be used in XML documents too, but only the first five entity names are predefined in XML specifications.

### 3.3.4 Associating DTD Declarations with XML Documents

To specify that an XML document is an instance of an XML dialect specified by a set of DTD declarations, you can either include the set of DTD declarations inside the XML document, which is less useful but convenient for teaching purposes, or save the DTD declarations in a separate DTD file and link the XML document to it, which is common practice.

If the DTD declarations are to be included in your XML document, they should be wrapped in a `DOCTYPE` definition with the following syntax

```
<!DOCTYPE rootElementTag [DTD-Declarations]>
```

and the `DOCTYPE` definition should be between the XML declaration and the root element of an XML document.

For example, file `dvd_embedded_dtd.xml` has the following contents:

```
<?xml version="1.0" encoding="UTF-8"?>  
<!DOCTYPE library [  
<!ELEMENT library (dvd+)>  
<!ELEMENT dvd (title, format, genre)>  
<!ELEMENT title (#PCDATA)>
```

```

<!ELEMENT format (#PCDATA)>
<!ELEMENT genre (#PCDATA)>
<!ATTLIST dvd id CDATA #REQUIRED>
]>
<library>
  <dvd id="1">
    <title>Gone with the Wind</title>
    <format>Movie</format>
    <genre>Classic</genre>
  </dvd>
  <dvd id="2">
    <title>Star Trek</title>
    <format>TV Series</format>
    <genre>Science fiction</genre>
  </dvd>
</library>

```

To link a DTD declaration file to an XML document, use the following DOCTYPE definition between the XML declaration and the root element:

```
<!DOCTYPE rootElementTag SYSTEM DTD-URL>
```

where DTD-URL can be either a file system path for the DTD file on the local file system or a URL for the DTD file deployed on the Internet.

The following content of file `dvd_dtd.xml` shows how it links to the DTD file `dvd.dtd` next to it in the local file system.

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE library SYSTEM "dvd.dtd">
<library>
  <dvd id="1">
    <title>Gone with the Wind</title>
    <format>Movie</format>
    <genre>Classic</genre>
  </dvd>
  <dvd id="2">
    <title>Star Trek</title>
    <format>TV Series</format>
    <genre>Science fiction</genre>
  </dvd>
</library>

```

## 3.4 XML Schema

Although DTD is part of the XML specification and supported by all XML processors, it is weak in its expressiveness for defining complex data structures. *XML Schema* (<http://www.w3.org/XML/Schema>) is an alternative industry standard for defining XML dialects. XML Schema itself is an XML dialect; thus, it can take advantage of many existing XML techniques and processors. It also has a much more detailed way to define what the data can and cannot contain, and it promotes declaration reuse so that common declarations can be factored out and referenced by multiple element or attribute declarations.

The following is an example XML Schema declaration for the earlier XML DVD dialect, and file `dvd.xsd` contains this declaration.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="library">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="dvd" minOccurs="0"
          maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="title"
                type="xs:string"/>
              <xs:element name="format"
                type="xs:string"/>
              <xs:element name="genre"
                type="xs:string"/>
            </xs:sequence>
            <xs:attribute name="id"
              type="xs:integer" use="required"/>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

### 3.4.1 XML Namespace

For convenient usage, XML element and attribute names should be short and meaningful. Therefore, XML dialects declared by different

people or companies tend to adopt the same names. If an XML document uses elements from more than one of these dialects, then naming conflicts may happen. XML namespace was introduced to avoid XML name conflicts.

A set of XML elements, attributes, and data types can be associated with a *namespace*, which could be any *unique string*. To help ensure uniqueness, namespaces are normally related to the declaring company or institution's URL. For example, "http://www.w3.org/2001/XMLSchema" is the namespace for the 2001 version of XML Schema, and "http://www.w3.org/1999/xhtml" is the namespace for *XHTML 1.0 Transitional*, as specified in all the XHTML examples in the previous chapter (refer to Section 2.2.2). String "http://csis.pace.edu" could be another example namespace for XML Schema declared by Pace University's School of Computer Science and Information Systems. Although namespaces normally look like URLs, they do not have to. There are usually no web resources corresponding to namespaces.

To specify XML elements, attributes, or data types of a namespace in an XML document, they are supposed to be qualified by their namespace. Because namespaces are normally long to be unique, *namespace prefixes*, which are normally one to four characters long, could be declared to represent the full namespaces in an XML document. Each XML document can choose its own namespace prefixes. In the earlier example, "xs" is an XML prefix representing namespace "http://www.w3.org/2001/XMLSchema". The association between a namespace prefix and a namespace is specified in the start tag of the root element in the form of attribute specification, except that the namespace prefix has its own prefix, `xmlns:`. For example, to specify that "xs" is the namespace prefix for namespace "http://www.w3.org/2001/XMLSchema", the following two lines in the example XML document are used:

```
<xs:schema
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
```

In an XML document, if an element is qualified by its namespace prefix, as `xs:element` for element `element` declared in namespace "http://www.w3.org/2001/XMLSchema", its attributes and nested elements by default belong to the same namespace.

If an XML document uses several namespaces, but most of the elements and attributes use the same namespace, you can use attribute `xmlns` to declare the default namespace in the start tag of the root element so that

those elements, attributes, or data types not qualified by namespace prefixes will be assumed to belong to this default namespace. As an example, if an XML document has the following attribute declaration in the start tag of its root element,

```
xmlns="http://csis.pace.edu"
```

then all unqualified elements and attributes in this document are supposed to belong to namespace “http://csis.pace.edu”.

If an XML Schema document declares an XML dialect belonging to a particular namespace, its root element should contain a `targetNamespace` attribute to specify the target namespace for elements, attributes, and data types declared in this dialect. As an example, if an XML Schema document’s root element includes attribute `targetNamespace`, like

```
<xs:schema targetNamespace="http://csis.pace.edu" . . . . .>
```

then all elements, attributes, and data types declared in this document belong to namespace “http://csis.pace.edu”. Example file `dvd-ns.xsd` contains the same contents as file `dvd.xsd`, but it declares all elements and attributes under namespace “http://csis.pace.edu”. Without such a `targetNamespace` attribute in the XML Schema root element, the XML dialect does not belong to any namespace, as in the previous example.

All XML Schema declarations for elements and data types immediately nested in the root schema element are called *global declarations*. Normally, the declarations of attributes are nested inside the declarations of their elements, and the declarations of the nested elements are nested inside the declaration of their hosting element. The proper usage of global declarations can promote declaration reuse, as you will see soon.

In the following examples, namespace prefix `xs` is assumed for the XML Schema namespace.

### 3.4.2 Declaring Simple Elements and Attributes

A *simple element* is an XML element that can contain only text based on simple data types defined in XML Schema specification (including `string`, `decimal`, `integer`, `positiveInteger`, `boolean`, `date`, `time`, `anyType`), those derived from such simple data types, or user custom types. It cannot contain any other elements or attributes. The following are some examples.

To declare element `color` that can take on any string value, use

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

As a result, element `<color>blue</color>` will have value “blue”, and element `<color/>` will have no value.

To declare element `color` that can take on any string value, with “red” to be its default value, use

```
<xs:element name="color" type="xs:string"
  default="red"/>
```

As a result, element `<color>blue</color>` will have value “blue”, and element `<color/>` will have the default value “red”.

To declare element `color` that can take on only the fixed string value, “red”, use

```
<xs:element name="color" type="xs:string"
  fixed="red"/>
```

As a result, element `<color>red</color>` will be correct, element `<color>blue</color>` will be invalid, and element `<color/>` will have the fixed (default) value “red”.

Although simple elements cannot have attributes, the syntax for declaring attributes in XML Schema is similar to that for simple elements. You just need to change `xs:element` to `xs:attribute` in the preceding examples. For example,

```
<xs:attribute name="lang" type="xs:string"
  default="EN"/>
```

declares that `lang` is an attribute of type `xs:string`, and its default value is `EN`. Such attribute declarations are always embedded in the declarations of complex elements to which they belong.

Attributes are optional by default. You can use attribute element’s `use` attribute to specify that the declared attribute is required for its hosting element. For example, if the earlier attribute `lang` does not have a default value but it must be specified for its hosting element, you can use the following declaration:

```
<xs:attribute name="lang" type="xs:string"
  use="required"/>
```

### 3.4.3 Declaring Complex Elements

A *complex element* is an XML element that contains other elements and/or attributes. There are four kinds of complex elements:

- Empty elements
- Elements that contain only other elements
- Elements that contain only customized simple types or attributes
- Elements that contain both other elements and text

To declare that `product` is an empty element type with optional integer-typed attribute `pid`, you can use the following:

```
<xs:element name="product">
  <xs:complexType>
    <xs:attribute name="pid" type="xs:integer"/>
  </xs:complexType>
</xs:element>
```

Example `product` elements include `<product/>` and `<product pid="1">`.

The following example declares that an `employee` element's value is a sequence of two nested elements: a `firstName` element followed by a `lastName` element, both of type `string`.

```
<xs:element name="employee">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="firstName"
        type="xs:string"/>
      <xs:element name="lastName"
        type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

The following is an example `employee` element:

```
<employee>
  <firstName>Tom</firstName>
  <lastName>Sawyer</lastName>
</employee>
```

Such nested element declarations have two problems. First, the width of paper or computer display will make deep element nesting hard to declare and read. Second, what if you also need to declare a `manager` element that

also contains a sequence of `firstName` and `lastName` elements? The type declaration for the `employee` and `manager` elements would be duplicated.

Fortunately, you can use global declarations and XML Schema element's type attribute to resolve the preceding two problems. The following is the previous `employee` element declaration in global declaration format as well as the declaration of a new `manager` element type.

```
<xs:element name="employee" type="fullName"/>
<xs:element name="manager" type="fullName"/>
<xs:complexType name="fullName">
  <xs:sequence>
    <xs:element name="firstName" type="xs:string"/>
    <xs:element name="lastName" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
```

The following example declares a `complexType` element, `shoeSize`. The content is defined as an integer value, and the `shoeSize` element also contains an attribute named `country`:

```
<xs:element name="shoeSize">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base="xs:integer">
        <xs:attribute name="country"
          type="xs:string"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>
```

An example `shoeSize` element is `<shoeSize country="france">35</shoeSize>`.

A mixed complex type element can contain attributes, elements, and text. You use attribute `mixed="true"` of the `complexType` element to specify that the value is a mixture of elements and text. The following declaration is for a `letter` element that can have a mixture of elements and text as its value:

```
<xs:element name="letter">
  <xs:complexType mixed="true">
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="orderID"
        type="xs:positiveInteger"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

```

        <xs:element name="shipDate" type="xs:date"/>
    </xs:sequence>
</xs:complexType>
</xs:element>

```

The following is an example letter element:

```

<letter>
Dear Mr.<name>John Smith</name>,
Your order <orderID>1032</orderID>
will be shipped on <shipDate>2008-09-23</shipDate>.
</letter>

```

### 3.4.4 Controlling Element Order and Repetition

For elements that contain other elements, the application of the sequence element (sequence, all, and choice are called order indicators of XML Schema) enforces an order of the nested elements, as for the previous employee element in which the nested lastName element must follow the firstName element. If you need to stipulate that each nested element can occur exactly once but in any order, you can replace the sequence element with the all element, as in the following modified employee example:

```

<xs:element name="employee">
  <xs:complexType>
    <xs:all>
      <xs:element name="firstName"
        type="xs:string"/>
      <xs:element name="lastName"
        type="xs:string"/>
    </xs:all>
  </xs:complexType>
</xs:element>

```

If you need to specify that an employee element need contain either a firstName element or a lastName element, but not both, you can replace the all element with the choice element in the previous example.

*Occurrence indicators* are used to define how many times an element can be used. XML Schema has two occurrence indicators, maxOccurs and minOccurs; both are attributes of XML Schema element element.

Attribute maxOccurs specifies up to how many times that its hosting element can occur at that location. It takes on a nonnegative integer or “unbounded” as the upper limit. Its default value is unbounded (unlimited).

Attribute `minOccurs` specifies at least how many times that its hosting element should occur at that location. It takes on a nonnegative integer as the lower limit. Its default value is 1.

As an example, the following declaration specifies that the `dvd` element can occur zero or an unlimited number of times.

```
<xs:element name="dvd" minOccurs="0"
  maxOccurs="unbounded">
```

### 3.4.5 Referencing XML Schema Specification in an XML Document

Not like DTD declarations, XML Schema declarations are always put in files separated from their instance document files. When you create an XML document, you may want to declare that the document is an instance of an XML dialect specified by an XML Schema file. The method of such association depends on whether the XML Schema declaration uses target namespaces.

#### 3.4.5.1 Specifying an XML Schema without Target Namespace

Assume that an XML dialect is specified with an XML Schema file `schemaFile.xsd` without using a target namespace, and the Schema file has URL `schemaFileURL`, which is either a local file system path like “`schemaFile.xsd`” or a web URL like “`http://csis.pace.edu/schemaFile.xsd`”. The instance documents of this dialect can be associated with its XML Schema declaration with the following structure, where `rootTag` is the name of a root element, `xsi` is defined as the namespace prefix for *XML Schema Instance*, and the latter includes a `noNamespaceSchemaLocation` attribute for specifying the location of the XML Schema file that does not use a target namespace.

```
<rootTag
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-[SYMBOLCHARACTER]
  instance"
  xsi:noNamespaceSchemaLocation="schemaFileURL"
>
```

#### 3.4.5.2 Specifying an XML Schema with Namespace

Assume that an XML dialect is specified with an XML Schema file `schemaFile.xsd` using target namespace `namespaceString` (say, `http://csis.pace.edu`), and the Schema file has URL `schemaFileURL`, which is either a

local file system path like “schemaFile.xsd” or a web URL like “http://csis.pace.edu/schemaFile.xsd”. The instance documents of this dialect can be associated with its XML Schema declaration with the following structure, where `rootTag` is the name of a root element, `xsi` is defined as the namespace prefix for XML Schema Instance, and the latter includes a `schemaLocation` attribute for specifying the location of the XML Schema file that uses a target namespace.

```
<rootTag
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-[SYMBOLCHARACTER]
  instance"
  xsi:schemaLocation=
    "namespaceString schemaFileURL"
>
```

### 3.5 XML Parsing and Validation with SAX and DOM

Most XML applications need to read in an XML document, analyze its data structure, and activate events when some language features are found. SAX (Simple API for XML) and DOM (Document Object Model) are two types of popular XML parsers for parsing and processing XML documents. SAX works as a pipeline. It reads in the input XML document sequentially and fires events when it detects the start or end of language features like elements and attributes. An application adopting a SAX parser needs to write an event handler class that has a processing method for each interested event type, and the methods are invoked by the SAX parser when corresponding types of events are fired. Because the XML document does not need to be stored completely in computer memory, SAX is efficient for some types of applications that do not need to search information backward in an XML document.

On the other hand, a DOM parser builds a complete tree data structure in the computer memory so it can be more convenient for detailed document analysis and language transformation. Even though DOM parsers use more computer memory, DOM is the main type of XML parser that is used with the Ajax technique.

Both SAX and DOM can work in validation mode. As part of the parsing process, they can check whether the input XML document is well formed. Furthermore, if the parser is fed both the XML dialect specification in DTD

or XML Schema as well as an XML document, the parser can check whether the XML document is an instance of the XML dialect.

Because SAX is not used on the client in Ajax, this book will not discuss SAX further. DOM parsing will be discussed in the next chapter.

## 3.6 XML Transformation with XSLT

As intermediate language representation of business data, XML instance documents must be transformable into other XML dialects or into XHTML documents for customized web presentation.

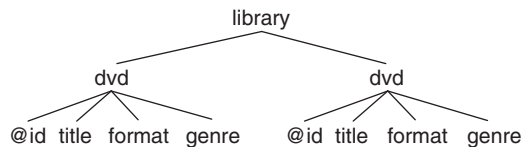
The World Wide Web Consortium (W3C) specified XSL (Extensible Stylesheet Language) as the standard language for writing stylesheets to transform XML documents among different dialects or into other languages. XSL stylesheets themselves are pure XML documents, so they can be processed by standard XML tools. XSL includes three components: XSLT (XSL Transformation) as an XML dialect for specifying XML transformation rules or stylesheets, XPath as a standard notation system for specifying subsets of elements in an XML document, and XSL-FO for formatting XML documents. This section briefly introduces XPath and XSLT. Most recent web browsers support XPath and XSLT, and so do Sun's recent JDK (Java SE Development Kit) versions.

Most examples in this section are based on file `dvd.xml` with the following contents:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- This XML document describes a DVD library -->
<library>
  <dvd id="1">
    <title>Gone with the Wind</title>
    <format>Movie</format>
    <genre>Classic</ genre >
  </dvd>
  <dvd id="2">
    <title>Star Trek</title>
    <format>TV Series</format>
    <genre>Science fiction</genre>
  </dvd>
</library>
```

### 3.6.1 Identifying XML Nodes with XPath

Before you can specify transformation rules for an XML dialect, you need to be able to specify subsets of XML elements that will be transformed based on some rules. You can visualize all components in an XML document, including the elements, attributes, and text, as a graph of nodes. And you can describe an XML document as an upside-down tree in which a node is connected to another node under it if the latter is immediately nested in the former or is a parameter or text value of the former. This is basically a DOM tree for representing an XML document in computer memory. The parameter names have symbol @ as their prefix in such a tree. The sibling nodes are ordered as they appear in the XML document. As an example, the contents of file `dvd.xml` can be described by the following tree.



XPath uses *path expressions* to select nodes in an XML document. The node is selected by following a path similar to file system paths. There are two ways to specify a path expression for the location of a set of nodes: absolute and relative. An *absolute location path* starts with a slash, /, and has the general form of

`/step/step/...`

whereas a *relative location path* does not start with a slash and has the general form of

`step/step/...`

In both cases, the path expression is evaluated from left to right, and each step is evaluated in the current node set to refine it. For an absolute location path, the current node set before the first step is the empty set, and the first step will identify the root element. For a relative location path, the current node set for the first step is defined by its context environment, which is normally another path expression. Each step has the following general form (items in square brackets are optional):

`[axisName::]nodeTest[predicate]`

where the optional axis name specifies the tree relationship between the selected nodes and the current node, the node test identifies a node type within an axis, and zero or more predicates are for further refining the selected node set.

Let us first look at some simpler path expressions in which the axis names and predicates are not used. Here the most useful path expressions include the following:

Expression	Description
nodeName	Selects all child nodes of the named node
/	Selects from the root node
//	Selects nodes in the document from the current node that match the selection, no matter where they are
.	Selects the current node
..	Selects the parent of the current node
@	Selects attributes
text()	Selects the text value of the current element
*	Selects any element nodes
@*	Selects any attribute node
node()	Selects any node of any kind (elements, attributes, ...)

Relative to the previous XML document `dvd.xml`, path expression `library` selects all the `library` elements in the current node set; `/library` selects the root element `library`; `library/dvd` selects all `dvd` elements that are children of `library` elements in the current node set; `//dvd` selects all `dvd` elements, no matter where they are in the document (no matter how many levels in which they are nested in other elements) relative to the current node set; `library//title` selects all `title` elements that are descendants of the `library` elements in the current node set, no matter where they are under the `library` elements; `//@id` selects all attributes that are named “id” relative to the current node set; and `/library/dvd/title/text()` selects the text values of all the `title` elements of the `dvd` elements.

Predicates in square brackets can be used to further narrow the subset of chosen nodes. For example, `/library/dvd[1]` selects the first `dvd` child element of `library` (IE5 and later use `[0]` for the first child); `/library/dvd[last()]` selects the last `dvd` child element of `library`; `/library/dvd[last()-1]` selects the last `dvd` child element next to the last of `library`; `/library/dvd[position()<3]` selects the first two `dvd` child elements of `library`; `//dvd[@id]` selects all `dvd` elements that have an `id` attribute; `//dvd[@id='2']` selects the `dvd` element that has an `id` attribute with value 2; `/library/dvd[genre='Classic']` selects all `dvd` child elements of `library` that have “Classic” as their `genre` value; and `/library/dvd[genre='Classic']/title` selects all `title` elements of `dvd` elements of `library` that have “Classic” as their `genre` value. Path expression predicates can use many popular binary operators in the same meaning as they are used in programming languages, including `+`, `-`, `*`, `div` (division), `=` (equal), `!=` (not equal), `<`, `<=`, `>`, `>=`, or (logical disjunction), and (logical conjunction), and `mod` (modulus).

You can use XPath wildcard expressions `*`, `@*`, and `node()` to select unknown XML elements. For example, for the previous XML document `dvd.xml`, `/library/*` selects all the child nodes of the `library` element; `//*` selects all elements in the document; and `//dvd[@*]` selects all `dvd` elements that have any attribute.

Several path expressions can also be combined by the disjunctive operator `|` for logical OR. For example, `//title | //genre` selects all `title` and `genre` elements in the previous document.

XPath also defines a set of *XPath axes* for specifying node subsets relative to the current node in a particular direction in the XML document’s tree representation. The following table lists the popular XPath axis names and their meanings.

Axis Name	Result
ancestor	Selects all ancestors (parent, grandparent, etc.) of the current node
ancestor-or-self	Selects all ancestors (parent, grandparent, etc.) of the current node and the current node itself
attribute	Selects all attributes of the current node
child	Selects all children of the current node
descendant	Selects all descendants (children, grandchildren, etc.) of the current node

Axis Name	Result
descendant-or-self	Selects all descendants (children, grandchildren, etc.) of the current node and the current node itself
following	Selects everything in the document after the end tag of the current node
following-sibling	Selects all siblings after the current node
namespace	Selects all namespace nodes of the current node
parent	Selects the parent of the current node
preceding	Selects everything in the document that is before the start tag of the current node
preceding-sibling	Selects all siblings before the current node
self	Selects the current node

As examples relative to the XML document `dvd.xml`, `child::dvd` selects all `dvd` nodes that are children of the current node; `attribute::id` selects the `id` attribute of the current node; `child::*` selects all children of the current node; `attribute::*` selects all attributes of the current node; `child::text()` selects all text child nodes of the current node; `child::node()` selects all child nodes of the current node; `descendant::dvd` selects all `dvd` descendants of the current node; `ancestor::dvd` selects all `dvd` ancestors of the current node; and `child::* / child::title` selects all `title` grandchildren of the current node.

### 3.6.2 Transforming XML Documents into XHTML Documents

XSLT is the major component of XSL, and it allows you to use the XML syntax to transform the instance documents of a particular XML dialect into those of another XML dialect or into other document types such as PDF. One of the popular functions of XSLT is to transform XML documents into HTML for web-based presentation, as shown in the examples in this section.

XSLT is based on DOM tree representation in computer memory. A common way to describe the transformation process is to say that XSLT transforms an XML source tree into an XML result tree. In the transformation process, XSLT uses XPath expressions to define parts of the source document that should match one or more predefined templates. When a match is found, XSLT will transform the matching part of the source document into the result document.

XSLT is an XML dialect that is declared under namespace “http://www.w3.org/1999/XSL/Transform”. Its root element is `stylesheet` or `transform`, and its current version is 1.0. The following is the contents of file `dvdToHTML.xsl`, which can transform XML document `dvd.xml` into an HTML file.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="html" version="4.0"/>
  <xsl:template match="/">
    <html>
      <head>
        <title>DVD Library Listing</title>
        <link rel="stylesheet" type="text/css"
          href="style.css"/>
      </head>
      <body>
        <table border="1">
          <tr>
            <th>Title</th>
            <th>Format</th>
            <th>Genre</th>
          </tr>
          <xsl:for-each select="/library/dvd">
            <xsl:sort select="genre"/>
            <tr>
              <td>
                <xsl:value-of select="title"/>
              </td>
              <td>
                <xsl:value-of select="format"/>
              </td>
              <td>
                <xsl:value-of select="genre"/>
              </td>
            </tr>
          </xsl:for-each>
        </table>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

The root element `stylesheet` declares a namespace prefix “`xsl`” for XSL namespace “`http://www.w3.org/1999/XSL/Transform`”. This root element could also be `transform`. The fourth line’s `xsl:output` element specifies that the output file of this transformation should follow the specification of HTML 4.0. Each `xsl:template` element specifies a transformation rule: if the document contains nodes satisfying the XPath expression specified by the `xsl:template`’s `match` attribute, then they should be transformed based on the value of this `xsl:template` element. Because this particular `match` attribute has value “`/`” selecting the root element of the input XML document, the rule applies to the entire XML document. The `template` element’s body (element value) dumps out an HTML template linked to an external CSS file named `style.css`. After generating the HTML table headers, the XSLT template uses an `xsl:for-each` element to loop through the `dvd` elements selected by the `xsl:for-each` element’s `select` attribute. In the loop body, the selected `dvd` elements are first sorted based on their `genre` value. Then the `xsl:value-of` elements are used to retrieve the values of the elements selected by their `select` attributes.

To use a web browser to transform the earlier file `dvd.xml` with this XSLT file `dvdToHTML.xsl` into HTML, you can add the following line after the XML declaration:

```
<?xml-stylesheet type="text/xsl" href="dvdToHTML.xsl"?>
```

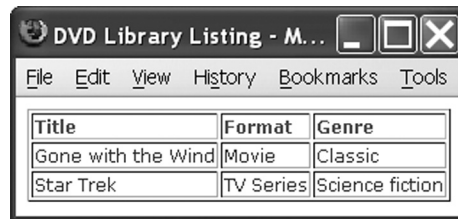
The resultant XML file is `dvd_XSLT.xml`, and its entire contents is shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl"
  href="dvdToHTML.xsl"?>
<library>
  <dvd id="1">
    <title>Gone with the Wind</title>
    <format>Movie</format>
    <genre>Classic</genre>
  </dvd>
  <dvd id="2">
    <title>Star Trek</title>
    <format>TV Series</format>
    <genre>Science fiction</genre>
  </dvd>
</library>
```

The following CSS file `style.css` is used for formatting the generated HTML file:

```
body, td
{
    font-weight: normal;
    font-size: 12px;
    color: purple;
    font-family: Verdana, Arial, sans-serif;
}
th {
    font-weight: bold;
    font-size: 12px;
    color: green;
    font-family: Verdana, Arial, sans-serif;
    text-align: left;
}
```

The following screen capture shows the web browser presentation of the HTML file generated by this XSLT transformation.



Element `xs1:value-of` can also be used to retrieve the value of attributes. For example, to retrieve the value of attribute `id` of the first `dvd` element, you can use

```
<xs1:value-of select="/library/dvd[1]/@id"/>
```

### 3.7 Summary

XML technologies are at the core of supporting platform- and language-independent system and data integration across networks. They support the portable approaches of defining customized languages for describing business data structures, parsing and validating business data, and transforming business data among various forms.

### 3.8 Self-Review Questions

1. XML is used mainly for specifying business data in a platform- and programming language-independent way.
  - a. True
  - b. False
2. An XML document can contain multiple root elements.
  - a. True
  - b. False
3. An XML dialect is a special XML document type that uses a predefined set of tag and attribute names and follows a predefined set of syntax rules, and it is for specifying the data structure of a particular type of document.
  - a. True
  - b. False
4. DTD and XML Schema are the main mechanisms for declaring XML dialects.
  - a. True
  - b. False
5. XML Schema is more expressive than DTD in declaring XML dialects.
  - a. True
  - b. False

6. An XML instance document can be claimed valid without referring to its dialect specification in DTD or XML Schema.
  - a. True
  - b. False
7. The value of an attribute must be inside a pair of double quotes or a pair of single quotes.
  - a. True
  - b. False
8. A namespace is for avoiding naming conflicts for element or attribute names so that several XML dialects can be used in one XML instance document.
  - a. True
  - b. False
9. If a namespace string is in the form of a URL, then there must be a corresponding web resource deployed at that URL.
  - a. True
  - b. False
10. SAX can always parse large XML documents more efficiently.
  - a. True
  - b. False
11. Each template element in an XSL document functions like a transformation rule.
  - a. True
  - b. False
12. XML processing tools are part of the latest web browsers and Sun Java JDKs.
  - a. True
  - b. False

### Keys to the Self-Review Questions

1. a 2. b 3. a 4. a 5. a 6. b 7. a 8. a 9. b 10. b 11. a 12. a

### 3.9 Exercises

1. What are the main functions of XML in today's IT systems?
2. What kinds of XML documents are well formed?
3. What kinds of XML documents are valid?
4. Why are namespaces important in XML technologies?
5. What are the similarities and differences between SAX and DOM parsers?
6. List some XML features that can be specified with XML Schema but not with DTD.
7. What are the major differences between CSS and XSL stylesheets?
8. What is the function of XPath in XSLT?

### 3.10 Programming Exercises

1. Declare an XML dialect for specifying a subset of student course registration information with DTD.
2. Declare an XML dialect for specifying a subset of student course registration information with XML Schema.
3. Write an XSLT document to transform the instance documents of the previous XML dialect into HTML through a web browser.

### 3.11 References

1. Michael Morrison. *Sams Teach Yourself XML in 24 Hours, Second Edition*, Sams, 2002. ISBN 0-672-32213-7.
2. Paul Whitehead, Ernest Friedman-Hill, and Emily Vander Veer. *Java and XML*, Wiley, 2002. ISBN 0-7645-3683-4.
3. W3C. *Extensible Markup Language (XML) 1.0 (Fourth Edition)*. <http://www.w3.org/TR/xml/>
4. W3C. *XML Schema*. <http://www.w3.org/XML/Schema/>
5. W3C. *The Extensible Stylesheet Language Family (XSL)*. <http://www.w3.org/Style/XSL/>
6. SAX. <http://www.saxproject.org/>

7. W3C. *Document Object Model (DOM)*. <http://www.w3.org/DOM/>
8. *XML Tutorial*. <http://www.w3schools.com/xml/>
9. Sun Microsystems. *Simple API for XML*.  
<http://java.sun.com/j2ee/1.4/docs/tutorial/doc/JAXPSAX.html>
10. *XML DOM Tutorial*. <http://www.w3schools.com/dom/>
11. *XSLT Tutorial*. <http://www.w3schools.com/xsl/>
12. *XPath Tutorial*. <http://www.w3schools.com/xpath/>