B

Document Type Definition (DTD)

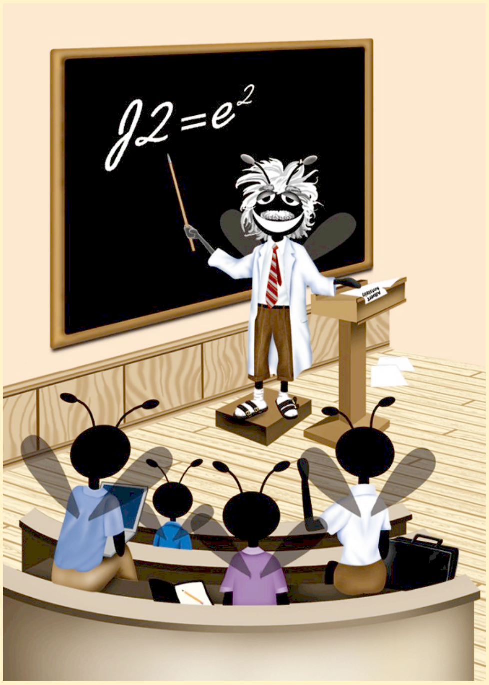
Objectives

- To understand what a DTD is.
- To be able to write DTDs.
- To be able to declare elements and attributes in a DTD.
- To understand the difference between general entities and parameter entities.
- To be able to use conditional sections with entities.
- To be able to use **`NOTATION`**s.
- To understand how an XML document's whitespace is processed.

To whom nothing is given, of him can nothing be required.
Henry Fielding

Like everything metaphysical, the harmony between thought and reality is to be found in the grammar of the language.
Ludwig Wittgenstein

Grammar, which knows how to control even kings.
Molière



Outline

B.1 Introduction

In this appendix, we discuss *Document Type Definitions* (*DTDs*), which define an XML document's structure (e.g., what elements, attributes, etc. are permitted in the document). An XML document is not required to have a corresponding DTD. However, DTDs are often recommended to ensure document conformity, especially in business-to-business (B2B) transactions, where XML documents are exchanged. DTDs specify an XML document's structure and are themselves defined using *EBNF* (*Extended Backus-Naur Form*) grammar—not the XML syntax introduced in Appendix A.

B.2 Parsers, Well-Formed and Valid XML Documents

Parsers are generally classified as *validating* or *nonvalidating*. A validating parser is able to read a DTD and determine whether the XML document conforms to it. If the document conforms to the DTD, it is referred to as *valid*. If the document fails to conform to the DTD but is syntactically correct, it is well formed, but not valid. By definition, a valid document is well formed.

A nonvalidating parser is able to read the DTD, but cannot check the document against the DTD for conformity. If the document is syntactically correct, it is well formed.

In this appendix, we use a Java program we created to check a document conformance. This program, named **`Validator.jar`**, is located in the Appendix B examples directory. **`Validator.jar`** uses the reference implementation for the Java API for XML Processing 1.1, which requires **`crimson.jar`** and **`jaxp.jar`**.

B.3 Document Type Declaration

DTDs are introduced into XML documents using the *document type declaration* (i.e., ***DOCTYPE***). A document type declaration is placed in the XML document's *prolog* (i.e., all lines preceding the root element), begins with **<!DOCTYPE** and ends with **>**. The document type declaration can point to declarations that are outside the XML document (called the *external subset*) or can contain the declaration inside the document (called the *internal subset*). For example, an internal subset might look like

```
<!DOCTYPE myMessage [
   <!ELEMENT myMessage ( #PCDATA )>
]>
```

The first **myMessage** is the name of the document type declaration. Anything inside the *square brackets* (**[]**) constitutes the internal subset. As we will see momentarily, **ELEMENT** and **#PCDATA** are used in "element declarations."

External subsets physically exist in a different file that typically ends with the ***.dtd*** *extension*, although this file extension is not required. External subsets are specified using either keyword the keyword ***SYSTEM*** or the keyword ***PUBLIC***. For example, the **DOCTYPE** external subset might look like

```
<!DOCTYPE myMessage SYSTEM "myDTD.dtd">
```

which points to the **myDTD.dtd** document. The **PUBLIC** keyword indicates that the DTD is widely used (e.g., the DTD for HTML documents). The DTD may be made available in well-known locations for more efficient downloading. We used such a DTD in Chapters 9 and 10 when we created XHTML documents. The **DOCTYPE**

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

uses the **PUBLIC** keyword to reference the well-known DTD for XHTML version 1.0. XML parsers that do not have a local copy of the DTD may use the URL provided to download the DTD to perform validation.

Both the internal and external subset may be specified at the same time. For example, the **DOCTYPE**

```
<!DOCTYPE myMessage SYSTEM "myDTD.dtd" [
   <!ELEMENT myElement ( #PCDATA )>
]>
```

contains declarations from the **myDTD.dtd** document, as well as an internal declaration.

Software Engineering Observation B.1

The document type declaration's internal subset plus its external subset form the DTD.

Software Engineering Observation B.2

The internal subset is visible only within the document in which it resides. Other external documents cannot be validated against it. DTDs that are used by many documents should be placed in the external subset.

B.4 Element Type Declarations

Elements are the primary building blocks used in XML documents and are declared in a DTD with ***element*** *type declarations* (***ELEMENT****s*). For example, to declare element **myMessage**, we might write

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

The element name (e.g., **myElement**) that follows **ELEMENT** is often called a *generic identifier*. The set of parentheses that follow the element name specify the element's allowed content and is called the *content specification*. Keyword **PCDATA** specifies that the element must contain *parsable character data*. These data will be parsed by the XML parser, therefore any markup text (i.e., **<**, **>**, **&**, etc.) will be treated as markup. We will discuss the content specification in detail momentarily.

Common Programming Error B.1

Attempting to use the same element name in multiple element type declarations is an error.

Figure B.1 lists an XML document that contains a reference to an external DTD in the **DOCTYPE**. We use **Validator.jar** to check the document's conformity against its DTD.

The document type declaration (line 6) specifies the name of the root element as **MyMessage**. The element **myMessage** (lines 8–10) contains a single child element named **message** (line 9).

Line 3 of the DTD (Fig. B.2) declares element **myMessage**. Notice that the content specification contains the name **message**. This indicates that element **myMessage** contains exactly one child element named **message**. Because **myMessage** can have only an element as its content, it is said to have *element content*. Line 4, declares element **message** whose content is of type **PCDATA**.

Common Programming Error B.2

Having a root element name other than the name specified in the document type declaration is an error.

If an XML document's structure is inconsistent with its corresponding DTD, but is syntactically correct, the document is only well formed—not valid. Figure B.3 shows the messages generated when the required **message** element is omitted.

```
 1  <?xml version = "1.0"?>
 2
 3  <!-- Fig. B.1: welcome.xml    -->
 4  <!-- Using an external subset -->
 5
 6  <!DOCTYPE myMessage SYSTEM "welcome.dtd">
 7
 8  <myMessage>
 9     <message>Welcome to XML!</message>
10  </myMessage>
```

Fig. B.1 XML document declaring its associated DTD.

```
<!-- Fig. B.2: welcome.dtd -->
<!-- External declarations -->
<!ELEMENT myMessage ( message )>
<!ELEMENT message ( #PCDATA )>
```

```
C:\>java -jar Validator.jar welcome.xml
Document is valid.
```

Fig. B.2 Validation by using an external DTD.

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

<!-- Fig. B.3 : welcome-invalid.xml    -->
<!-- well-formed, but invalid document -->

<!DOCTYPE myMessage SYSTEM "welcome.dtd">

<!-- Root element missing child element message -->
<myMessage>
</myMessage>
```

```
C:\>java -jar Validator.jar welcome-invalid.xml
error: Element "myMessage" requires additional elements.
```

Fig. B.3 Invalid XML document.

B.4.1 Sequences, Pipe Characters and Occurrence Indicators

DTDs allow the document author to define the order and frequency of child elements. The *comma* (**,**)—called a *sequence*—specifies the order in which the elements must occur. For example,

```
<!ELEMENT classroom ( teacher, student )>
```

specifies that element **`classroom`** must contain exactly one **`teacher`** element followed by exactly one **`student`** element. The content specification can contain any number of items in sequence.

Similarly, choices are specified using the *pipe character* (**|**), as in

```
<!ELEMENT dessert ( iceCream | pastry )>
```

which specifies that element **`dessert`** must contain either one **`iceCream`** element or one **`pastry`** element, but not both. The content specification may contain any number of pipe character-separated choices.

An element's frequency (i.e., number of occurrences) is specified by using either the *plus sign* (**+**), *asterisk* (*****) or *question mark* (**?**) *occurrence indicator* (Fig. B.4).

Occurrence Indicator	Description
Plus sign (**+**)	An element can appear any number of times, but must appear at least once (i.e., the element appears one or more times).
Asterisk (*****)	An element is optional, and if used, the element can appear any number of times (i.e., the element appears zero or more times).
Question mark (**?**)	An element is optional, and if used, the element can appear only once (i.e., the element appears zero or one times).

Fig. B.4 Occurrence indicators.

A plus sign indicates one or more occurrences. For example,

```
<!ELEMENT album ( song+ )>
```

specifies that element **album** contains one or more **song** elements.

The frequency of an *element group* (i.e., two or more elements that occur in some combination) is specified by enclosing the element names inside the content specification with parentheses, followed by either the plus sign, asterisk or question mark. For example,

```
<!ELEMENT album ( title, ( songTitle, duration )+ )>
```

indicates that element **album** contains one **title** element followed by any number of **songTitle/duration** element groups. At least one **songTitle/duration** group must follow **title**, and in each of these element groups, the **songTitle** must precede the **duration**. An example of markup that conforms to this is

```
<album>
   <title>XML Classical Hits</title>

   <songTitle>XML Overture</songTitle>
   <duration>10</duration>

   <songTitle>XML Symphony 1.0</songTitle>
   <duration>54</duration>
</album>
```

which contains one **title** element followed by two **songTitle/duration** groups. The asterisk (*****) character indicates an optional element that, if used, can occur any number of times. For example,

```
<!ELEMENT library ( book* )>
```

indicates that element **library** contains any number of **book** elements, including the possibility of none at all. Markup examples that conform to this mark up are

```
<library>
   <book>The Wealth of Nations</book>
   <book>The Iliad</book>
   <book>The Jungle</book>
</library>
```

and

```
<library></library>
```

Optional elements that, if used, may occur only once are followed by a question mark (**?**). For example,

```
<!ELEMENT seat ( person? )>
```

indicates that element **seat** contains at most one **person** element. Examples of markup that conform to this are

```
<seat>
   <person>Jane Doe</person>
</seat>
```

and

```
<seat></seat>
```

Now we consider three more element type declarations and provide a declaration for each. The declaration

```
<!ELEMENT class ( number, ( instructor | assistant+ ),
                  ( credit | noCredit ) )>
```

specifies that a **class** element must contain a **number** element, either one **instructor** element or any number of **assistant** elements and either one **credit** element or one **noCredit** element. Markup examples that conform to this are

```
<class>
   <number>123</number>
   <instructor>Dr. Harvey Deitel</instructor>
   <credit>4</credit>
</class>
```

and

```
<class>
   <number>456</number>
   <assistant>Tem Nieto</assistant>
   <assistant>Paul Deitel</assistant>
   <credit>3</credit>
</class>
```

The declaration

```
<!ELEMENT donutBox ( jelly?, lemon*,
   ( ( creme | sugar )+ | glazed ) )>
```

specifies that element **donutBox** can have zero or one **jelly** elements, followed by zero or more **lemon** elements, followed by one or more **creme** or **sugar** elements or exactly one **glazed** element. Markup examples that conform to this are

```
<donutBox>
   <jelly>grape</jelly>
   <lemon>half-sour</lemon>
   <lemon>sour</lemon>
   <lemon>half-sour</lemon>
   <glazed>chocolate</glazed>
</donutBox>
```

and

```
<donutBox>
   <sugar>semi-sweet</sugar>
   <creme>whipped</creme>
   <sugar>sweet</sugar>
</donutBox>
```

The declaration

```
<!ELEMENT farm ( farmer+, ( dog* | cat? ), pig*,
   ( goat | cow )?,( chicken+ | duck* ) )>
```

indicates that element **farm** can have one or more **farmer** elements, any number of optional **dog** elements or an optional **cat** element, any number of optional **pig** elements, an optional **goat** or **cow** element and one or more **chicken** elements or any number of optional **duck** elements. Examples of markup that conform to this are

```
<farm>
   <farmer>Jane Doe</farmer>
   <farmer>John Doe</farmer>
   <cat>Lucy</cat>
   <pig>Bo</pig>
   <chicken>Jill</chicken>
</farm>
```

and

```
<farm>
   <farmer>Red Green</farmer>
   <duck>Billy</duck>
   <duck>Sue</duck>
</farm>
```

B.4.2 EMPTY, Mixed Content and ANY

Elements must be further refined by specifying the types of content they contain. In the previous section, we introduced element content, indicating that an element can contain one or more child elements as its content. In this section, we introduce *content specification types* for describing nonelement content.

In addition to element content, three other types of content exist: ***EMPTY***, *mixed content* and ***ANY***. Keyword **EMPTY** declares empty elements, which do not contain character data or child elements. For example,

```
<!ELEMENT oven EMPTY>
```

declares element **oven** to be an empty element. The markup for an **oven** element would appear as

```
<oven/>
```

or

```
<oven></oven>
```

in an XML document conforming to this declaration.

An element can also be declared as having mixed content. Such elements may contain any combination of elements and **PCDATA**. For example, the declaration

```
<!ELEMENT myMessage ( #PCDATA | message )*>
```

indicates that element **myMessage** contains mixed content. Markup conforming to this declaration might look like

```
<myMessage>Here is some text, some
   <message>other text</message>and
   <message>even more text</message>.
</myMessage>
```

Element **myMessage** contains two **message** elements and three instances of character data. Because of the *****, element **myMessage** could have contained nothing.

Figure B.5 specifies the DTD as an internal subset (lines 6–10). In the prolog (line 1), we use the ***standalone*** attribute with a value of **yes**. An XML document is *standalone* if it does not reference an external subset. This DTD defines three elements: one that contains mixed content and two that contain parsed character data.

```
1  <?xml version = "1.0" standalone = "yes"?>
2
3  <!-- Fig. B.5 : mixed.xml         -->
4  <!-- Mixed content type elements -->
5
6  <!DOCTYPE format [
7     <!ELEMENT format ( #PCDATA | bold | italic )*>
8     <!ELEMENT bold ( #PCDATA )>
9     <!ELEMENT italic ( #PCDATA )>
10 ]>
11
12 <format>
13    Book catalog entry:
14    <bold>XML</bold>
15    <italic>XML How to Program</italic>
16    This book carefully explains XML-based systems development.
17 </format>
```

```
C:\>java -jar Validator.jar mixed.xml
Document is valid.
```

Fig. B.5 Example of a mixed-content element.

Line 7 declares element **format** as a mixed content element. According to the declaration, the **format** element may contain either parsed character data (**PCDATA**), element **bold** or element **italic**. The asterisk indicates that the content can occur zero or more times. Lines 8 and 9 specify that **bold** and **italic** elements only have **PCDATA** for their content specification—they cannot contain child elements. Despite the fact that elements with **PCDATA** content specification cannot contain child elements, they are still considered to have mixed content. The comma (**,**), plus sign (**+**) and question mark (**?**) occurrence indicators cannot be used with mixed-content elements that contain only **PCDATA**.

Figure B.6 shows the results of changing the first pipe character in line 7 of Fig. B.5 to a comma and the result of removing the asterisk. Both of these are illegal DTD syntax.

Common Programming Error B.3

When declaring mixed content, not listing ***PCDATA*** *as the first item is an error.*

An element declared as type **ANY** can contain any content, including **PCDATA**, elements or a combination of elements and **PCDATA**. Elements with **ANY** content can also be empty elements.

Software Engineering Observation B.3

Elements with ***ANY*** *content are commonly used in the early stages of DTD development. Document authors typically replace* ***ANY*** *content with more specific content as the DTD evolves.*

B.5 Attribute Declarations

In this section, we discuss *attribute declarations*. An attribute declaration specifies an *attribute list* for an element by using the ***ATTLIST*** *attribute list declaration*. An element can have any number of attributes. For example,

```
<!ELEMENT x EMPTY>
<!ATTLIST x y CDATA #REQUIRED>
```

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

<!-- Fig. B.6 : invalid-mixed.xml -->
<!-- Mixed content type elements   -->

<!DOCTYPE format [
   <!ELEMENT format ( #PCDATA | bold, italic )>
   <!ELEMENT bold ( #PCDATA )>
   <!ELEMENT italic ( #PCDATA )>
]>

<format>
   Book catalog entry:
   <bold>XML</bold>
   <italic>XML How to Program</italic>
   This book carefully explains XML-based systems development.
</format>
```

Fig. B.6 Changing a pipe character to a comma in a DTD (part 1 of 2).

```
C:>java -jar Validator.jar invalid-mixed.xml
fatal error: Mixed content model for "format" must end with ")*", not
",".
```

Fig. B.6 Changing a pipe character to a comma in a DTD (part 2 of 2).

declares **EMPTY** element **x**. The attribute declaration specifies that **y** is an attribute of **x**. Keyword **CDATA** indicates that **y** can contain any character text except for the **<**, **>**, **&**, **'** and **"** characters. Note that the **CDATA** keyword in an attribute declaration has a different meaning than the **CDATA** section in an XML document we introduced in Appendix A. Recall that in a **CDATA** section all characters are legal except the **]]>** end tag. *Keyword* **#REQUIRED** specifies that the attribute must be provided for element **x**. We will say more about other keywords momentarily.

Figure B.7 demonstrates how to specify attribute declarations for an element. Line 9 declares attribute **id** for element **message**. Attribute **id** contains required **CDATA**. Attribute values are normalized (i.e., consecutive whitespace characters are combined into one whitespace character). We discuss normalization in detail in Section B.8. Line 13 assigns attribute **id** the value **"6343070"**.

DTDs allow document authors to specify an attribute's default value using *attribute defaults*, which we briefly touched upon in the previous section. Keywords ***#IMPLIED***, ***#REQUIRED*** and ***#FIXED*** are attribute defaults. Keyword **#IMPLIED** specifies that if the attribute does not appear in the element, then the application using the XML document can use whatever value (if any) it chooses.

Keyword **#REQUIRED** indicates that the attribute must appear in the element. The XML document is not valid if the attribute is missing. For example, the markup

```
<message>XML and DTDs</message>
```

```
1   <?xml version = "1.0"?>
2
3   <!-- Fig. B.7: welcome2.xml -->
4   <!-- Declaring attributes   -->
5
6   <!DOCTYPE myMessage [
7      <!ELEMENT myMessage ( message )>
8      <!ELEMENT message ( #PCDATA )>
9      <!ATTLIST message id CDATA #REQUIRED>
10  ]>
11
12  <myMessage>
13
14     <message id = "6343070">
15        Welcome to XML!
16     </message>
17
18  </myMessage>
```

Fig. B.7 Declaring attributes (part 1 of 2).

```
C:\>java -jar Validator.jar welcome2.xml
Document is valid.
```

Fig. B.7 Declaring attributes (part 2 of 2).

when checked against the DTD attribute list declaration

```
<!ATTLIST message number CDATA #REQUIRED>
```

does not conform to it because attribute **number** is missing from element **message**.

An attribute declaration with default value **#FIXED** specifies that the attribute value is constant and cannot be different in the XML document. For example,

```
<!ATTLIST address zip #FIXED "02115">
```

indicates that the value **"02115"** is the only value attribute **zip** can have. The XML document is not valid if attribute **zip** contains a value different from **"02115"**. If element **address** does not contain attribute **zip**, the default value **"02115"** is passed to the application that is using the XML document's data.

B.6 Attribute Types

Attribute types are classified as either *string* (**CDATA**), *tokenized* or *enumerated*. *String attribute* types do not impose any constraints on attribute values, other than disallowing the **<** and **&** characters. Entity references (e.g., **&lt;**, **&amp;**, etc.) must be used for these characters. *Tokenized attribute types* impose constraints on attribute values, such as which characters are permitted in an attribute name. We discuss tokenized attribute types in the next section. *Enumerated attribute types* are the most restrictive of the three types. They can take only one of the values listed in the attribute declaration. We discuss enumerated attribute types in Section B.6.2.

B.6.1 Tokenized Attribute Type (ID, IDREF, ENTITY, NMTOKEN)

Tokenized attribute types allow a DTD author to restrict the values used for attributes. For example, an author may want to have a unique ID for each element or allow an attribute to have only one or two different values. Four different tokenized attribute types exist: **ID**, **IDREF**, **ENTITY** and **NMTOKEN**.

Tokenized attribute type **ID** uniquely identifies an element. Attributes with type **IDREF** point to elements with an **ID** attribute. A validating parser verifies that every **ID** attribute type referenced by **IDREF** is in the XML document.

Figure B.8 lists an XML document that uses **ID** and **IDREF** attribute types. Element **bookstore** consists of element **shipping** and element **book**. Each **shipping** element describes who shipped the book and how long it will take for the book to arrive.

Line 9 declares attribute **shipID** as an **ID** type attribute (i.e., each **shipping** element has a unique identifier). Lines 27–37 declare **book** elements with attribute **shippedBy** (line 11) of type **IDREF**. Attribute **shippedBy** points to one of the **shipping** elements by matching its **shipID** attribute.

Common Programming Error B.4

*Using the same value for multiple **`ID`** attributes is a logic error: The document validated against the DTD is not valid.*

The DTD contains an *entity declaration* for each of the entities **`isbnXML`**, **`isbnJava`** and **`isbnCPP`**. The parser replaces the entity references with their values. These entities are called *general entities*.

Figure B.9 is a variation of Fig. B.8 that assigns **`shippedBy`** (line 32) the value **`"bug"`**. No **`shipID`** attribute has a value **`"bug"`**, which results in a invalid XML document.

```
1  <?xml version = "1.0"?>
2
3  <!-- Fig. B.8: IDExample.xml                        -->
4  <!-- Example for ID and IDREF values of attributes -->
5
6  <!DOCTYPE bookstore [
7     <!ELEMENT bookstore ( shipping+, book+ )>
8     <!ELEMENT shipping ( duration )>
9     <!ATTLIST shipping shipID ID #REQUIRED>
10    <!ELEMENT book ( #PCDATA )>
11    <!ATTLIST book shippedBy IDREF #IMPLIED>
12    <!ELEMENT duration ( #PCDATA )>
13    <!ENTITY isbnXML "0-13-028417-3">
14    <!ENTITY isbnJava "0-13-034151-7">
15    <!ENTITY isbnCPP "0-13-0895717-3">
16 ]>
17
18 <bookstore>
19    <shipping shipID = "bug2bug">
20       <duration>2 to 4 days</duration>
21    </shipping>
22
23    <shipping shipID = "Deitel">
24       <duration>1 day</duration>
25    </shipping>
26
27    <book shippedBy = "Deitel" isbn = "&isbnJava;">
28       Java How to Program 4th edition.
29    </book>
30
31    <book shippedBy = "Deitel" isbn = "&isbnXML;">
32       XML How to Program.
33    </book>
34
35    <book shippedBy = "bug2bug" isbn = "&isbnCPP;">
36       C++ How to Program 3rd edition.
37    </book>
38 </bookstore>
```

```
C:\>java -jar Validator.jar IDExample.xml
Document is valid.
```

Fig. B.8 XML document with **`ID`** and **`IDREF`** attribute types (part 1 of 2).

```
C:\>java -jar ParserTest.jar idexample.xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- Fig. B.8: IDExample.xml                        -->
<!-- Example for ID and IDREF values of attributes -->
<bookstore>
   <shipping shipID="bug2bug">
      <duration>2 to 4 days</duration>
   </shipping>

   <shipping shipID="Deitel">
      <duration>1 day</duration>
   </shipping>

   <book shippedBy="Deitel" isbn="0-13-034151-7">
      Java How to Program 4th edition.
   </book>

   <book shippedBy="Deitel" isbn="0-13-028417-3">
      XML How to Program.
   </book>

   <book shippedBy="bug2bug" isbn="0-13-0895717-3">
      C++ How to Program 3rd edition.
   </book>
</bookstore>
```

Fig. B.8 XML document with **ID** and **IDREF** attribute types (part 2 of 2).

Common Programming Error B.5

*Not beginning a type attribute **ID**'s value with a letter, an underscore (_) or a colon (**:**) is an error.*

Common Programming Error B.6

*Providing more than one **ID** attribute type for an element is an error.*

Common Programming Error B.7

*Declaring attributes of type **ID** as **#FIXED** is an error.*

Related to entities are *entity attributes,* which indicate that an attribute has an entity for its value. Entity attributes are specified by using tokenized attribute type ***ENTITY***. The primary constraint placed on **ENTITY** attribute types is that they must refer to *external unparsed entities*. An external unparsed entity is defined in the external subset of a DTD and consists of character data that will not be parsed by the XML parser.

Figure B.10 lists an XML document that demonstrates the use of entities and entity attribute types.

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

```

Fig. B.9 Error displayed when an invalid **ID** is referenced (part 1 of 2).

```
<!-- Fig. B.9: invalid-IDExample.xml                -->
<!-- Example for ID and IDREF values of attributes -->

<!DOCTYPE bookstore [
   <!ELEMENT bookstore ( shipping+, book+ )>
   <!ELEMENT shipping ( duration )>
   <!ATTLIST shipping shipID ID #REQUIRED>
   <!ELEMENT book ( #PCDATA )>
   <!ATTLIST book shippedBy IDREF #IMPLIED>
   <!ELEMENT duration ( #PCDATA )>
]>

<bookstore>
   <shipping shipID = "bug2bug">
      <duration>2 to 4 days</duration>
   </shipping>

   <shipping shipID = "Deitel">
      <duration>1 day</duration>
   </shipping>

   <book shippedBy = "Deitel">
      Java How to Program 4th edition.
   </book>

   <book shippedBy = "Deitel">
      C How to Program 3rd edition.
   </book>

   <book shippedBy = "bug">
      C++ How to Program 3rd edition.
   </book>
</bookstore>
```

```
C:\>java -jar Validator.jar invalid-IDExample.xml
error: No element has an ID attribute with value "bug".
```

Fig. B.9 Error displayed when an invalid **ID** is referenced (part 2 of 2).

Line 7 declares a *notation* named **html** that refers to a **SYSTEM** identifier named **"iexplorer"**. Notations provide information that an application using the XML document can use to handle unparsed entities. For example, the application using this document may choose to open Internet Explorer and load the document **tour.html** (line 8).

Line 8 declares an entity named **city** that refers to an external document (**tour.html**). *Keyword* ***NDATA*** indicates that the content of this external entity is not XML. The name of the notation (e.g., **html**) that handles this unparsed entity is placed to the right of **NDATA**.

Line 11 declares attribute **tour** for element **company**. Attribute **tour** specifies a required **ENTITY** attribute type. Line 16 assigns entity **city** to attribute **tour**. If we replaced line 16 with

```
<company tour = "country">
```

the document fails to conform to the DTD because entity **country** does not exist. Figure B.11 shows the error message generated when the above replacement is made.

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

<!-- Fig. B.10: entityExample.xml        -->
<!-- ENTITY and ENTITY attribute types   -->

<!DOCTYPE database [
   <!NOTATION xhtml SYSTEM "iexplorer">
   <!ENTITY city SYSTEM "tour.html" NDATA xhtml>
   <!ELEMENT database ( company+ )>
   <!ELEMENT company ( name )>
   <!ATTLIST company tour ENTITY #REQUIRED>
   <!ELEMENT name ( #PCDATA )>
]>

<database>
   <company tour = "city">
      <name>Deitel &amp; Associates, Inc.</name>
   </company>
</database>
```

```
C:\>java -jar Validator.jar entityexample.xml
Document is valid.
```

Fig. B.10 XML document that contains an **ENTITY** attribute type.

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

<!-- Fig. B.11: invalid-entityExample.xml -->
<!-- ENTITY and ENTITY attribute types    -->

<!DOCTYPE database [
   <!NOTATION xhtml SYSTEM "iexplorer">
   <!ENTITY city SYSTEM "tour.html" NDATA xhtml>
   <!ELEMENT database ( company+ )>
   <!ELEMENT company ( name )>
   <!ATTLIST company tour ENTITY #REQUIRED>
   <!ELEMENT name ( #PCDATA )>
]>

<database>
   <company tour = "country">
      <name>Deitel &amp; Associates, Inc.</name>
   </company>
</database>
```

Fig. B.11 Error generated when a DTD contains a reference to an undefined entity (part 1 of 2).

```
C:\>java -jar Validator.jar invalid-entityexample.xml
error: Attribute value "country" does not name an unparsed entity.
```

Fig. B.11 Error generated when a DTD contains a reference to an undefined entity (part 2 of 2).

Common Programming Error B.8

Not assigning an unparsed external entity to an attribute with attribute type **`ENTITY`** *results in an invalid XML document.*

Attribute type **`ENTITIES`** may also be used in a DTD to indicate that an attribute has multiple entities for its value. Each entity is separated by a space. For example

```
<!ATTLIST directory file ENTITIES #REQUIRED>
```

specifies that attribute **`file`** is required to contain multiple entities. An example of markup that conforms to this might look like

```
<directory file = "animations graph1 graph2">
```

where **`animations`**, **`graph1`** and **`graph2`** are entities declared in a DTD.

A more restrictive attribute type is **`NMTOKEN`** (*name token*), whose value consists of letters, digits, periods, underscores, hyphens and colon characters. For example, consider the declaration

```
<!ATTLIST sportsClub phone NMTOKEN #REQUIRED>
```

which indicates **`sportsClub`** contains a required **`NMTOKEN`** **`phone`** attribute. An example of markup that conforms to this is

```
<sportsClub phone = "555-111-2222">
```

An example that does not conform to this is

```
<sportsClub phone = "555 555 4902">
```

because spaces are not allowed in an **`NMTOKEN`** attribute.

Similarly, when an **`NMTOKENS`** attribute type is declared, the attribute may contain multiple string tokens separated by spaces.

B.6.2 Enumerated Attribute Types

Enumerated attribute types declare a list of possible values an attribute can have. The attribute must be assigned a value from this list to conform to the DTD. Enumerated type values are separated by pipe characters (**|**). For example, the declaration

```
<!ATTLIST person gender ( M | F ) "F">
```

contains an enumerated attribute type declaration that allows attribute **`gender`** to have either the value **`M`** or the value **`F`**. A default value of **`"F"`** is specified to the right of the element attribute type. Alternatively, a declaration such as

```
<!ATTLIST person gender ( M | F ) #IMPLIED>
```

does not provide a default value for **gender**. This type of declaration might be used to validate a marked-up mailing list that contains first names, last names, addresses, etc. The application that uses such a mailing list may want to precede each name by either Mr., Ms. or Mrs. However, some first names are gender neutral (e.g., Chris, Sam, etc.), and the application may not know the **person**'s gender. In this case, the application has the flexibility to process the name in a gender-neutral way.

NOTATION is also an enumerated attribute type. For example, the declaration

```
<!ATTLIST book reference NOTATION ( JAVA | C ) "C">
```

indicates that **reference** must be assigned either **JAVA** or **C**. If a value is not assigned, **C** is specified as the default. The notation for **C** might be declared as

```
<!NOTATION C SYSTEM
     "http://www.deitel.com/books/2000/chtp3/chtp3_toc.htm">
```

B.7 Conditional Sections

DTDs provide the ability to include or exclude declarations using conditional sections. Keyword **INCLUDE** specifies that declarations are included, while keyword **IGNORE** specifies that declarations are excluded. For example, the conditional section

```
<![INCLUDE[
<!ELEMENT name ( #PCDATA )>
]]>
```

directs the parser to include the declaration of element **name**.

Similarly, the conditional section

```
<![IGNORE[
<!ELEMENT message ( #PCDATA )>
]]>
```

directs the parser to exclude the declaration of element **message**. Conditional sections are often used with entities, as demonstrated in Fig. B.12.

```
<!-- Fig. B.12: conditional.dtd          -->
<!-- DTD for conditional section example -->

<!ENTITY % reject "IGNORE">
<!ENTITY % accept "INCLUDE">

<![ %accept; [
   <!ELEMENT message ( approved, signature )>
]]>

<![ %reject; [
   <!ELEMENT message ( approved, reason, signature )>
]]>

<!ELEMENT approved EMPTY>
<!ATTLIST approved flag ( true | false ) "false">
```

Fig. B.12 Conditional sections in a DTD (part 1 of 2).

```
17
18  <!ELEMENT reason ( #PCDATA )>
19  <!ELEMENT signature ( #PCDATA )>
```

Fig. B.12 Conditional sections in a DTD (part 2 of 2).

Lines 4–5 declare entities **reject** and **accept**, with the values **IGNORE** and **INCLUDE**, respectively. Because each of these entities is preceded by a *percent* (**%**) *character*, they can be used only inside the DTD in which they are declared. These types of entities—called *parameter entities*—allow document authors to create entities specific to a DTD—not an XML document. Recall that the DTD is the combination of the internal subset and external subset. Parameter entities may be placed only in the external subset.

Lines 7–13 use the entities **accept** and **reject**, which represent the strings **INCLUDE** and **IGNORE**, respectively. Notice that the parameter entity references are preceded by **%**, whereas normal entity references are preceded by **&**. Line 7 represents the beginning tag of an **IGNORE** section (the value of the **accept** entity is **IGNORE**), while line 11 represents the start tag of an **INCLUDE** section. By changing the values of the entities, we can easily choose which **message** element declaration to allow.

Figure B.13 shows the XML document that conforms to the DTD in Fig. B.12.

Software Engineering Observation B.4

Parameter entities allow document authors to use entity names in DTDs without conflicting with entities names used in an XML document.

B.8 Whitespace Characters

In Appendix A, we briefly discussed whitespace characters. In this section, we discuss how whitespace characters relate to DTDs. Depending on the application, insignificant whitespace characters may be collapsed into a single whitespace character or even removed entirely. This process is called *normalization.* Whitespace is either preserved or normalized, depending on the context in which it is used.

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

<!-- Fig. B.13: conditional.xml -->
<!-- Using conditional sections -->

<!DOCTYPE message SYSTEM "conditional.dtd">

<message>
   <approved flag = "true" />
   <signature>Chairman</signature>
</message>
```

```
C:\>java -jar Validator.jar conditional.xml
Document is valid.
```

Fig. B.13 XML document that conforms to **conditional.dtd**.

Figure B.14 contains a DTD and markup that conforms to the DTD. Line 28 assigns a value containing multiple whitespace characters to attribute **cdata**. Attribute **cdata** (declared in line 11) is required and must contain **CDATA**. As mentioned earlier, **CDATA** can contain almost any text, including whitespace. As the output illustrates, spaces in **CDATA** are preserved and passed on to the application that is using the XML document.

Line 30 assigns a value to attribute **id** that contains leading whitespace. Attribute **id** is declared on line 14 with tokenized attribute type **ID**. Because this is not **CDATA**, it is normalized and the leading whitespace characters are removed. Similarly, lines 32 and 34 assign values that contain leading whitespace to attributes **nmtoken** and **enumeration**—which are declared in the DTD as an **NMTOKEN** and an enumeration, respectively. Both these attributes are normalized by the parser.

```
1   <?xml version = "1.0"?>
2
3   <!-- Fig. B.14 : whitespace.xml       -->
4   <!-- Demonstrating whitespace parsing -->
5
6   <!DOCTYPE whitespace [
7      <!ELEMENT whitespace ( hasCDATA,
8         hasID, hasNMTOKEN, hasEnumeration, hasMixed )>
9
10     <!ELEMENT hasCDATA EMPTY>
11     <!ATTLIST hasCDATA cdata CDATA #REQUIRED>
12
13     <!ELEMENT hasID EMPTY>
14     <!ATTLIST hasID id ID #REQUIRED>
15
16     <!ELEMENT hasNMTOKEN EMPTY>
17     <!ATTLIST hasNMTOKEN nmtoken NMTOKEN #REQUIRED>
18
19     <!ELEMENT hasEnumeration EMPTY>
20     <!ATTLIST hasEnumeration enumeration ( true | false )
21              #REQUIRED>
22
23     <!ELEMENT hasMixed ( #PCDATA | hasCDATA )*>
24  ]>
25
26  <whitespace>
27
28     <hasCDATA cdata = "  simple cdata  "/>
29
30     <hasID id = "  i20"/>
31
32     <hasNMTOKEN nmtoken = "   hello"/>
33
34     <hasEnumeration enumeration = "   true"/>
35
36     <hasMixed>
37        This is text.
38        <hasCDATA cdata = " simple    cdata"/>
```

Fig. B.14 Processing whitespace in an XML document (part 1 of 2).

```
         This is some additional text.
      </hasMixed>

   </whitespace>
```

```
C:\Documents and Settings\Administrator\My Documents\advanced java xml
appendice
s\appBexamples>java -jar Validator.jar whitespace.xml
Document is valid.
```

```
C:\>java -jar ParserTest.jar ..\appBexamples\whitespace.xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- Fig. B.14 : whitespace.xml       -->
<!-- Demonstrating whitespace parsing -->
<whitespace>

   <hasCDATA cdata="  simple cdata  "/>

   <hasID id="i20"/>

   <hasNMTOKEN nmtoken="hello"/>

   <hasEnumeration enumeration="true"/>

   <hasMixed>
      This is text.
      <hasCDATA cdata=" simple    cdata"/>
      This is some additional text.
   </hasMixed>

</whitespace>
```

Fig. B.14 Processing whitespace in an XML document (part 2 of 2).

B.9 Internet and World Wide Web Resources

`www.wdvl.com/Authoring/HTML/Validation/DTD.html`
Contains a description of the historical uses of DTDs, including a description of SGML and the HTML DTD.

`www.xml101.com/dtd`
Contains tutorials and explanations on creating DTDs.

`www.w3schools.com/dtd`
Contains DTD tutorials and examples.

`wdvl.internet.com/Authoring/Languages/XML/Tutorials/Intro/index3.html`
A DTD tutorial for Web developers.

`www.schema.net`
A DTD repository with XML links and resources.

`www.networking.ibm.com/xml/XmlValidatorForm.html`
IBM’s DOMit XML Validator.

SUMMARY

- Document Type Definitions (DTDs) define an XML document's structure (e.g., what elements, attributes, etc. are permitted in the XML document). An XML document is not required to have a corresponding DTD. DTDs use EBNF (Extended Backus-Naur Form) grammar.
- Parsers are generally classified as validating or nonvalidating. A validating parser can read the DTD and determine whether or not the XML document conforms to it. If the document conforms to the DTD, it is referred to as valid. If the document fails to conform to the DTD but is syntactically correct, it is well formed but not valid. By definition, a valid document is well formed.
- A nonvalidating parser is able to read a DTD, but cannot check the document against the DTD for conformity. If the document is syntactically correct, it is well formed.
- DTDs are introduced into XML documents by using the document type declaration (i.e., **DOCTYPE**). The document type declaration can point to declarations that are outside the XML document (called the external subset) or can contain the declaration inside the document (called the internal subset).
- External subsets physically exist in a different file that typically ends with the **.dtd** extension, although this file extension is not required. External subsets are specified using keyword **SYSTEM** or **PUBLIC** Both the internal and external subset may be specified at the same time.
- Elements are the primary building block used in XML documents and are declared in a DTD with element type declarations (**ELEMENT**s).
- The element name that follows **ELEMENT** is often called a generic identifier. The set of parentheses that follow the element name specify the element's allowed content and is called the content specification.
- Keyword **PCDATA** specifies that the element must contain parsable character data—that is, any text except the characters less than (**<**) and ampersand (**&**).
- An XML document is a **standalone** XML document if it does not reference an external DTD.
- An XML element that can have only another element for content is said to have element content.
- DTDs allow document authors to define the order and frequency of child elements. The comma (**,**)—called a sequence—specifies the order in which the elements must occur. Choices are specified using the pipe character (**|**). The content specification may contain any number of pipe-character-separated choices.
- An element's frequency (i.e., number of occurrences) is specified by using either the plus sign (**+**), asterisk (*****) or question mark (**?**) occurrence indicator.
- The frequency of an element group (i.e., two or more elements that occur in some combination) is specified by enclosing the element names inside the content specification, followed by an occurrence indicator.
- Elements can be further refined by describing the content types they may contain. Content specification types (e.g., **EMPTY**, mixed content, **ANY**, etc.) describe nonelement content.
- An element can be declared as having mixed content (i.e., a combination of elements and **PCDATA**). The comma (**,**), plus sign (**+**) and question mark (**?**) occurrence indicators cannot be used with mixed content elements.
- An element declared as type **ANY** can contain any content including **PCDATA**, elements, or a combination of elements and **PCDATA**. Elements with **ANY** content can also be empty elements.
- An attribute list for an element is declared using the **ATTLIST** element type declaration.
- Attribute values are normalized (i.e., consecutive whitespace characters are combined into one whitespace character).

- DTDs allow document authors to specify an attribute's default value using attribute defaults. Keywords **#IMPLIED**, **#REQUIRED** and **#FIXED** are attribute defaults.
- Keyword **#IMPLIED** specifies that if the attribute does not appear in the element, then the application using the XML document can apply whatever value (if any) it chooses.
- Keyword **#REQUIRED** indicates that the attribute must appear in the element. The XML document is not valid if the attribute is missing.
- An attribute declaration with default value **#FIXED** specifies that the attribute value is constant and cannot be different in the XML document.
- Attribute types are classified as either string (**CDATA**), tokenized or enumerated. String attribute types do not impose any constraints on attribute values, other than disallowing the **<** and **&** characters. Entity references (e.g., **&lt;**, **&amp;**, etc.) must be used for these characters. Tokenized attributes impose constraints on attribute values, such as which characters are permitted in an attribute name. Enumerated attributes are the most restrictive of the three types. They can take only one of the values listed in the attribute declaration.
- Four different tokenized attribute types exist: **ID**, **IDREF**, **ENTITY** and **NMTOKEN**. Tokenized attribute type **ID** uniquely identifies an element. Attributes with type **IDREF** point to elements with an **ID** attribute. A validating parser verifies that every **ID** attribute type referenced by **IDREF** is in the XML document.
- Entity attributes indicate that an attribute has an entity for its value and are specified using tokenized attribute type **ENTITY**. The primary constraint placed on **ENTITY** attribute types is that they must refer to external unparsed entities.
- Attribute type **ENTITIES** may also be used in a DTD to indicate that an attribute has multiple entities for its value. Each entity is separated by a space.
- A more restrictive attribute type is attribute type **NMTOKEN** (name token), whose value consists of letters, digits, periods, underscores, hyphens and colon characters.
- Attribute type **NMTOKENS** may contain multiple string tokens separated by spaces.
- Enumerated attribute types declare a list of possible values an attribute can have. The attribute must be assigned a value from this list to conform to the DTD. Enumerated type values are separated by pipe characters (|).
- **NOTATION** is also an enumerated attribute type. Notations provide information that an application using the XML document can use to handle unparsed entities.
- Keyword **NDATA** indicates that the content of an external entity is not XML. The name of the notation that handles this unparsed entity is placed to the right of **NDATA**.
- DTDs provide the ability to include or exclude declarations using conditional sections. Keyword **INCLUDE** specifies that declarations are included, while keyword **IGNORE** specifies that declarations are excluded. Conditional sections are often used with entities.
- Parameter entities are preceded by percent (**%**) characters and can be used only inside the DTD in which they are declared. Parameter entities allow document authors to create entities specific to a DTD—not an XML document.
- Whitespace is either preserved or normalized, depending on the context in which it is used. Spaces in **CDATA** are preserved. Attribute values with tokenized attribute types **ID**, **NMTOKEN** and enumeration are normalized.

TERMINOLOGY

#FIXED
#IMPLIED
#PCDATA
#REQUIRED

.dtd extension
ANY
asterisk (*****)
ATTLIST statement
attribute content
attribute declaration
attribute default
attribute list
attribute name
attribute value
CDATA
character data type
child element
comma character
conditional section
content specification
content specification type
declaration
default value of an attribute
DOCTYPE (document type declaration)
document type declaration
double quote (**"**)
DTD (Document Type Definition)
EBNF (Extended Backus-Naur Form) grammar
element
element content
element name
ELEMENT statement
element type declaration (**!ELEMENT**)
EMPTY
empty element
ENTITIES
entity attribute
ENTITY tokenized attribute type
enumerated attribute type
Extended Backus-Naur Form (EBNF) grammar
external subset
external unparsed entity
fixed value
general entity
generic identifier
hyphen (**-**)
ID tokenized attribute type
IDREF tokenized attribute type
IGNORE
INCLUDE
internal subset
mixed content
mixed-content element
mixed-content type
NDATA
NMTOKEN tokenized attribute type (name token)
nonvalid document
nonvalidating parser
normalization
NOTATION
notation type
occurrence indicator
optional element
parameter entity
parsed character data
parser
percent sign (**%**)
period
pipe character (**|**)
plus sign (**+**)
question mark (**?**)
quote (**'**)
sequence (**,**)
standalone XML document
string attribute type
string token
structural definition
syntax
SYSTEM
text
tokenized attribute type
type
valid document
validating parser
validation
well-formed document
whitespace character

SELF-REVIEW EXERCISES

B.1 State whether the following are *true* or *false*. If the answer is *false*, explain why.

a) The document type declaration, **DOCTYPE**, introduces DTDs in XML documents.
b) External DTDs are specified by using the keyword **EXTERNAL**.
c) A DTD can contain either internal or external subsets of declarations, but not both.
d) Child elements are declared in parentheses, inside an element type declaration.
e) An element that appears any number of times is followed by an exclamation point (**!**).

f) A mixed-content element can contain text as well as other declared elements.
g) An attribute declared as type **`CDATA`** can contain all characters except for the asterisk (**`*`**) and pound sign (**`#`**).
h) Each element attribute of type **`ID`** must have a unique value.
i) Enumerated attribute types are the most restrictive category of attribute types.
j) An enumerated attribute type requires a default value.

B.2 Fill in the blanks in each of the following statements:
a) The set of document type declarations inside an XML document is called the __________.
b) Elements are declared with the __________ type declaration.
c) Keyword __________ indicates that an element contains parsable character data.
d) In an element type declaration, the pipe character (**`|`**) indicates that the element can contain __________ of the elements indicated.
e) Attributes are declared by using the __________ type.
f) Keyword __________ specifies that the attribute can take only a specific value that has been defined in the DTD.
g) **`ID`**, **`IDREF`**, __________ and **`NMTOKEN`** are all types of tokenized attributes.
h) The **`%`** character is used to declare a/an __________.
i) DTD is an acronym for __________.
j) Conditional sections of DTDs are often used with __________.

ANSWERS TO SELF-REVIEW EXERCISES

B.1 a) True. b) False. External DTDs are specified using keyword **`SYSTEM`**. c) False. A DTD contains both the internal and external subsets. d) True. e) False. An element that appears one or zero times is specified by a question mark (**`?`**). f) True. g) False. An attribute declared as type **`CDATA`** can contain all characters except for ampersand (**`&`**), less than (**`<`**), greater than (**`>`**), quote (**`'`**) and double quotes (**`"`**). h) True. i) True. j) False. A default value is not required.

B.2 a) internal subset. b) **`ELEMENT`**. c) **`PCDATA`**. d) one. e) **`ATTLIST`**. f) **`#FIXED`**. g) **`ENTITY`**. h) parameter entity. i) Document Type Definition. j) entities.