## 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
6.1 Introduction

In this chapter, we discuss Document Type Definitions (DTDs) that 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 Chapter 5.

Software Engineering Observation 6.1

A transition is underway in the XML community from DTDs to Schema (Chapter 7), which improve upon DTDs. Schema use XML syntax, not EBNF grammar.

6.2 Parsers, Well-formed and Valid XML Documents

Parsers are generally classified as validating or nonvalidating. A validating parser is able to 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 the DTD, but cannot check the document against the DTD for conformity. If the document is syntactically correct, it is well formed.
We will discuss validating and nonvalidating parsers in greater depth in Chapters 8 and 9. In this chapter, we use Microsoft’s XML Validator to check for document conformance to a DTD. XML Validator is available at no charge from

msdn.microsoft.com/downloads/samples/Internet/xml/xml_validator/sample.asp

6.3 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 and 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 internal subset). For example, an internal subset might look like

```xml
<!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 SYSTEM or PUBLIC. For example, the DOCTYPE external subset might look like

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

which points to the myDTD.dtd document. Using 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 2 and 3 when we created HTML documents. The DOCTYPE

```xml
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
    "http://www.w3.org/TR/html4/strict.dtd">
```

uses the PUBLIC keyword to reference the well-known DTD for HTML version 4.01. 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

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

contains declarations from the myDTD.dtd document as well as an internal declaration.
Software Engineering Observation 6.2
The document type declaration’s internal subset plus its external subset form the DTD.

Software Engineering Observation 6.3
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.

6.4 Element Type Declarations

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

```xml
<!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. This 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 6.1
Attempting to use the same element name in multiple element type declarations is an error.

Figure 6.1 lists an XML document that contains a reference to an external DTD in the DOCTYPE. We use Microsoft’s XML Validator to check the document’s conformity against its DTD. [Note: To use XML Validator, Internet Explorer 5 is required. In Chapters 8 and 9, we introduce parsers XML4J and Xerces, which can be used to check a document’s validity against a DTD programmatically. Using Java and one of these parsers provides a platform-independent way to validate XML documents.]

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

```xml
<?xml version = "1.0"?>
<!-- Fig. 6.1: intro.xml -->
<!DOCTYPE myMessage SYSTEM "intro.dtd">
<myMessage>
  <message>Welcome to XML!</message>
</myMessage>
```

Fig. 6.1 XML document declaring its associated DTD.
<!-- Fig. 6.2: intro.dtd -->
<!-- External declarations -->

<!ELEMENT myMessage ( message )>
<!ELEMENT message ( #PCDATA )>

Fig. 6.2 Validation with using an external DTD.
Line 4 of the DTD (Fig. 6.2) 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 only have an element as its content, it is said to have element content. Line 5 declares element `message` whose content is of type `PCDATA`. [Note: Many XML Validator screen captures contain the term `SCHEMA`. The XML Validator is capable of validating an XML document against both DTDs and documents—called `Schemas`—that also define an XML document’s structure. In Chapter 7, we will discuss Schema in Chapter 7 and how they differ from DTDs.]

**Common Programming Error 6.2**

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, it is only well formed—not valid. Figure 6.3 shows the messages generated by Microsoft’s XML Validator when the required `message` element is omitted.

### 6.4.1 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 )>
```

```
<?xml version = "1.0"?>
<!DOCTYPE myMessage SYSTEM "intro.dtd">
<!-- Root element missing child element message -->
<myMessage>
</myMessage>
```

![Fig. 6.3](xmlhtp1_06.fm) Non-valid XML document.
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

```xml
<!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. 6.4).

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

```xml
<!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,

```xml
<!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

```xml
<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.

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

Fig. 6.4 Occurrence indicators.
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 are

```
<library>
  <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 complicated 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>
```
The declaration

```xml
<!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

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

and

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

The declaration

```xml
<!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

```xml
<farm>
    <farmer>Jane Doe</farmer>
    <farmer>John Doe</farmer>
    <cat>Lucy</cat>
    <pig>Bo</pig>
    <chicken>Jill</chicken>
</farm>
```
6.4.2 EMPTY, Mixed Content and ANY

Elements must be further refined by specifying the types of content they contain. In the last 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 non-element content.

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

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

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

```xml
<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

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

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

```xml
<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 6.5 specifies a DTD as an internal subset (lines 6–10) as opposed to an external subset (Fig. 6.1). 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. Document Type Definition (DTD)  

Chapter 6

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 have `PCDATA` only 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 6.6 shows the results of changing the first pipe character in line 7 of Fig. 6.5 to a comma and the result of removing the asterisk. Both of these are illegal DTD syntax.

Common Programming Error 6.3

When declaring mixed content, not listing `PCDATA` as the first item is an error.
Fig. 6.6  Illegal mixed-content element syntax.

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.

**Common Programming Error 6.4**

Child elements of an element declared as type **ANY** must have their own element type declarations.

**Software Engineering Observation 6.4**

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.

### 6.5 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,

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

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 Chapter 5. 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 6.7 demonstrates how to specify attribute declarations for an element. Line 9 declares attributes id and to for element message. Both id and to contain required CDATA. Attribute values are normalized (i.e., consecutive whitespace characters are combined into one whitespace character). We discuss normalization in detail in Section 6.8. Line 13 assigns attribute id the value "445" and assigns attribute to the value "The world".

```xml
<?xml version = "1.0"?>
<!-- Fig. 6.7: intro2.xml -->
<!-- Declaring attributes -->
<!DOCTYPE myMessage [  
   <!ELEMENT myMessage ( message )>  
   <!ELEMENT message ( #PCDATA )>  
   <!ATTLIST message id CDATA #REQUIRED>  
]>  
<myMessage>  
   <message id = "445">  
      Welcome to XML!  
   </message>  
</myMessage>
```

Fig. 6.7 Declaring attributes.
6.5.1 Attribute Defaults ( #REQUIRED, #IMPLIED, #FIXED )

DTDs allow document authors to specify an attribute’s default value using attribute defaults, which we briefly touched upon in the last 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

```xml
<message>number</message>
```

when checked against the DTD attribute list declaration

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

does not conform 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,

```xml
<!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 using the XML document’s data.

6.6 Attribute Types

Attribute types are classified as either strings (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; &gt; etc.) must be used for these characters. Tokenized attributes impose constraints on attribute values—such as which characters are permitted in an attribute name. We discuss tokenized attributes in the next section. Enumerated attributes are the most restrictive of the three types. They can take only one of the values listed in the attribute declaration. We will discuss enumerated attribute types in Section 6.6.2.

6.6.1 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 only allow an attribute to have 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.

**Common Programming Error 6.5**

Using the same value for multiple ID attributes is a logic error—the document validated against the DTD is not valid.
Figure 6.8 lists an XML document that uses ID and IDREF attribute types. Element bookstore consists of element shipping and element book. Each shipping element describes a shipping method.

Line 9 declares attribute shipID as an ID type attribute (i.e., each shipping element has a unique identifier). Lines 24–34 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.

If we assign shippedBy (line 28) the value "s3", an error occurs when we use Microsoft’s Validator (Fig. 6.9). No shipID attribute has a value "s3", which results in a non-valid XML document.

**Common Programming Error 6.6**
Not beginning a type attribute ID’s value with a letter, underscore (_) or a colon (: is an error.

**Common Programming Error 6.7**
Providing more than one ID attribute type for an element is an error.

```xml
<?xml version = "1.0"?>
<!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 = "s1">
    <duration>2 to 4 days</duration>
  </shipping>
  <shipping shipID = "s2">
    <duration>1 day</duration>
  </shipping>
  <book shippedBy = "s2">
    Java How to Program 3rd edition.
  </book>
  <book shippedBy = "s2">
    C How to Program 3rd edition.
  </book>
</bookstore>
```

Fig. 6.8  XML document with ID and IDREF attribute types (part 1 of 2).
32  <book shippedBy = "s1">
33    C++ How to Program 3rd edition.
34  </book>
35  </bookstore>

Fig. 6.8  XML document with ID and IDREF attribute types (part 2 of 2).

Fig. 6.9  Error displayed by XML Validator when an invalid ID is referenced.
Common Programming Error 6.8
Declaring attributes of type \texttt{ID} as \texttt{#FIXED} is an error.

In Chapter 5, we briefly introduced the concept of DTDs and entities. Figure 5.4 (\texttt{lang.xml}) referenced \texttt{lang.dtd}, which contained the values for the entity references \&assoc; and \&text;. External subset \texttt{lang.dtd} contains the two entity declarations

\begin{verbatim}
<!ENTITY assoc "أسّوشِ
  "&#1571;&#1587;&#1617;&#1608;&#1588;&#1616;&#1610;&#1614;&#1578;&#1618;&#1587;">
and

<!ENTITY text "اليونيكو
  
  "&#1575;&#1604;&#1610;&#1606;&#1610;&#1603;&#1608;&#1583;">
\end{verbatim}

for entities \texttt{assoc} and \texttt{text}. A parser replaces the entity references with their values. For example, consider the following entity declaration

\begin{verbatim}
<!ENTITY digits "0123456789">
\end{verbatim}

for \texttt{digits}. This entity might be used as follows

\begin{verbatim}
<useAnEntity>&digits;</useAnEntity>
\end{verbatim}

The entity reference \&digits; is replaced by its value, resulting in

\begin{verbatim}
<useAnEntity>0123456789</useAnEntity>
\end{verbatim}

the value \texttt{0123456789} being placed inside the tags. These entities are called \texttt{general entities}. Related to entities are \texttt{entity attributes}, which indicate that an attribute has an entity for its value. These entity attributes are specified by using tokenized attribute type \texttt{ENTITY}. The primary constraint placed on \texttt{ENTITY} attribute types is that they must refer to \texttt{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 6.10 lists an XML document that demonstrates the use of entities and entity attribute types.

\begin{verbatim}
1  <?xml version = "1.0"?>
2 3  <!-- Fig. 6.10: entityExample.xml -->
4  <!-- ENTITY and ENTITY attribute types -->
5 6  <!DOCTYPE database [ 7  <!NOTATION html SYSTEM "iexplorer"> 8  <!ENTITY city SYSTEM "tour.html" NDATA html>
9  <!ELEMENT database ( company+ )>
10  <!ELEMENT company ( name )>
\end{verbatim}

\textbf{Fig. 6.10} XML document that contains an \texttt{ENTITY} attribute type (part 1 of 2).
<!NOTATION html SYSTEM "iexplorer"/>

Line 7

<!ENTITY city SYSTEM "tour.html" NDATA html>

Line 8

<!ATTLIST company tour ENTITY #REQUIRED>

Line 11

Fig. 6.10  XML document that contains an ENTITY attribute type (part 2 of 2).

<!NOTATION html SYSTEM "iexplorer">

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).

<!ENTITY city SYSTEM "tour.html" NDATA html>

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.

<!ATTLIST company tour ENTITY #REQUIRED>
declares attribute `tour` for element `company`. Attribute `tour` specifies a required `ENTITY` attribute type. Line 16

```xml
<company tour = "city">
```

assigns entity `city` to attribute `tour`. If we replaced line 16 with

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

doctrine fails to conform to the DTD because entity `country` does not exist. Figure 6.11 shows the message generated by XML Validator if `country` is used.

### Common Programming Error 6.9

Not assigning an unparsed external entity to an attribute with attribute type `ENTITY` results in a non-valid 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

```xml
<!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

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

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

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

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

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

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

![Fig. 6.11](image)

**Fig. 6.11** Error generated by XML Validator when a DTD contains a reference to an undefined entity.
An example that does not conform to this is

```xml
<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.

### 6.6.2 Enumerated Attribute Types

In this section, we discuss enumerated attribute types, which 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

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

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

```xml
<!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 this 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,

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

the declaration 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

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

### 6.7 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

```xml
<! [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 6.12.
Lines 4 and 5

```
ENTITY % reject "IGNORE">
ENTITY % accept "INCLUDE">
```
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. [Note: Recall that the DTD is the combination of the internal subset and external subset. Parameter entities may only be placed 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

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

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 6.13 shows the XML document that conforms to the DTD in Fig. 6.12.

---

```
1 <!-- Fig. 6.12: conditional.dtd -->
2 <!-- DTD for conditional section example -->
3
4 <!--ENTITY % reject "IGNORE">
5 <!--ENTITY % accept "INCLUDE">
6
7 <!-- %accept; [ 
8   <!ELEMENT message ( approved, signature )>
9 ]]> 
10
11 <!-- %reject; [ 
12   <!ELEMENT message ( approved, reason, signature )>
13 ]]> 
14
15 <!--ELEMENT approved EMPTY>
16 <!--ATTLIST approved flag ( true | false ) "false">
17
18 <!--ELEMENT reason ( #PCDATA )>
19 <!--ELEMENT signature ( #PCDATA )>
```

---

Fig. 6.12  Conditional sections in a DTD.
Chapter 6  Document Type Definition (DTD)  155

1  <?xml version = "1.0" standalone = "no"?>
2 3  <!-- Fig. 6.13: conditional.xml -->
4  <!-- Using conditional sections -->
5 6  <!DOCTYPE message SYSTEM "conditional.dtd”>
7 8  <message>
9     <approved flag = "true"/>
10    <signature>Chairman</signature>
11  </message>

Fig. 6.13  XML document that conforms to conditional.dtd.

Software Engineering Observation 6.5

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

6.8 Whitespace Characters

In Chapter 5, we briefly discussed whitespace characters and normalization. In this section, we discuss how whitespace characters and normalization relate to DTDs. Whitespace is either preserved or normalized, depending on the context in which it is used.

Figure 6.14 contains a DTD and markup that conforms to the DTD. The output shown is generated by a Java application presented in Chapter 9.

1  <?xml version = "1.0"?>
2

Fig. 6.14  Processing whitespace in an XML document (part 1 of 3).
<!DOCTYPE whitespace [
  <!ELEMENT whitespace ( hasCDATA,
    hasID, hasNMTOKEN, hasEnumeration, hasMixed )>
  <!ELEMENT hasCDATA EMPTY>
  <!ATTLIST hasCDATA cdata CDATA #REQUIRED>
  <!ELEMENT hasID EMPTY>
  <!ATTLIST hasID id ID #REQUIRED>
  <!ELEMENT hasNMTOKEN EMPTY>
  <!ATTLIST hasNMTOKEN nmtoken NMTOKEN #REQUIRED>
  <!ELEMENT hasEnumeration EMPTY>
  <!ATTLIST hasEnumeration enumeration ( true | false ) #REQUIRED>
  <!ELEMENT hasMixed ( #PCDATA | hasCDATA )*>
]>
<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. 6.14  Processing whitespace in an XML document (part 2 of 3).
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 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. [Note: We discuss the ignorable and text portions of the output in Chapter 9.]
6.9 Case Study: Writing a DTD for the Day Planner Application

In this section, we build upon the case study introduced in Chapter 5. Figure 6.15 lists the external subset of the DTD for the day planner XML document, planner.xml. The following document type declaration is inserted into the day planner XML document:

```xml
<!DOCTYPE planner SYSTEM "planner.dtd">
```

```xml
1  <!-- Fig. 6.15: planner.dtd -->
2  <!-- DTD for day planner -->
3  <!ELEMENT planner ( year* )>
4  <!ELEMENT year ( date+ )>
5  <!ATTLIST year value CDATA #REQUIRED>
6  <!ELEMENT date ( note+ )>
7  <!ATTLIST date month CDATA #REQUIRED>
8  <!ATTLIST date day CDATA #REQUIRED>
9  <!ELEMENT note ( #PCDATA )>
10  <!ATTLIST note time CDATA #IMPLIED>
```

Fig. 6.15 DTD for planner.xml.
Software Engineering Observation 6.6

When an attribute’s value is normalized, consecutive carriage returns/line feeds are collapsed into a single carriage return/line feed that is replaced by a space character.

Element **planner** is the root element of the document, which contains any number of optional **year** elements. The declarations (lines 6 and 7) specify that **year** element must contain one or more **date** elements and must contain an attribute **value** that has character data.

Line 9 indicates that a **date** element contains one or more **note** elements. Element **date** is required to have two attributes: **month** and **day**. Element **note** contains parsed character data and an optional attribute **time**.

6.10 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.dtd.com
A repository of DTDs for XML documents.

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

A DTD tutorial.

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

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

msdn.microsoft.com/downloads/samples/Internet/xml/xml_validator/sample.asp
Download page for Microsoft’s XML Validator.

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 is able to 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 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**. 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 (<), greater-than (>), ampersand (&), quote (‘) and double quote (“).

An XML document is a **standalone** XML document if it does not reference an external DTD.

An XML element that can only have another element for content, it is said to have element content.

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. 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 strings (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; &gt; &apos; &quot;) 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 this 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. Attributes values with tokenized attribute types ID, NMTOKEN and enumeration are normalized.

**TERMINOLOGY**

**ANY**
application

**ATTLIST**
statement
attribute
attribute content
attribute declaration
attribute default
attribute list
attribute name
attribute value
asterisk (*)

**CDATA**
character data type
child elements
comma character
conditional section
content specification
classification type
declaration
default value of an attribute

**DOCTYPE**
document type declaration
SELF-REVIEW EXERCISES

6.1 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 (#) characters.

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.
Chapter 6 Document Type Definition (DTD) 163

6.2 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 only take a specific value that has been defined in the DTD.
   g) _______ and _______ 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

6.1 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.

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

EXERCISES

6.3 Create a DTD for Fig. 5.6 (letter.xml).

6.4 Create a DTD (products.dtd) for a retailer with the following specifications: The XML document that conforms to the DTD must contain a list of products and manufacturers. Each product should be represented by a product element and each manufacturer should be represented by a manufacturer element. Each manufacturer has a unique ID. Represent details like name, address, etc., as child elements of a manufacturer element. Each product has attributes such as product code (which is always unique), unit price, etc. Each product is classified into one of four categories: electronics, household, furniture and groceries. Each product should be related to a manufacturer described in the XML document and should be represented as an element. The product can have a model name and description as elements. [Hint: Relate products to manufacturers using the IDREF type attribute.]

6.5 Use the DTD shown in Fig. 6.16 to construct an XML document. Use the declared entities for the authors element. Validate your XML document against the DTD using Microsoft’s XML Validator (or a similar validation program).

```xml
<!-- ex06_15.dtd: Database containing books -->
<!ELEMENT database ( book+ )>
<!ENTITY HD "Harvey Deitel”>
<!ENTITY PD "Paul Deitel”>
```

Fig. 6.16 DTD for a book database (part 1 of 2).
6.6 Write an XML document that declares an address book containing contacts. Each contact has a name and address. An address should contain attributes for street name, state and phone number. The attribute value for state should not contain spaces. For example, "New York" would invalidate the XML document. The attribute value for a phone number must contain hyphens and no spaces (e.g., 978-555-1212). Use entities for states names. [Hint: use NMTOKENs]

6.7 Write an XML document for the DTD created in Exercise 6.6 and validate it.

6.8 Write a DTD for an XML document that stores company profiles. Each company is represented by a company element. Profiles must contain a name element and a tour element. Element tour points to a .jpg image or to an HTML page that relates to the company. Element tour should specify the type of image or document to which it points. [Note: The type of image should be specified by a NOTATION type attribute.]. Also write an XML document and validate it against this DTD.

6.9 For the preceding exercise, write an internal subset of declarations that declare the type attribute as enumerated. Make corresponding changes in the XML document. Observe that the internal subset overrides the external subset when declarations collide.

6.10 Briefly describe each element type declaration:
   a) name ( firstName, middleName?, lastName )
   b) test ( question, answer )* 
   c) discussion ( agenda, ( issues, solutions )* )