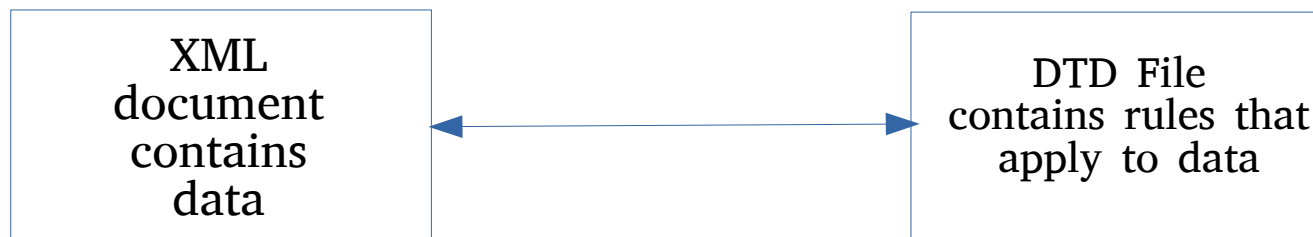


UNIT 2 – Document Type Definition (DTD)

- Introduction to DTD
- Why do we need DTDs?
- Types of DTD
 - External
 - Internal
- Inserting Comments in a DTD
- Element Type Declaration
 - Declaring Elements
 - Elements Content Models
 - Sequence, Occurrences, Choice
 - Empty, Any, Mixed
- Attribute Declaration
 - Declaring Attributes
 - Default for Attributes
 - Attribute Types
- Conditional Sections
- Limitations of DTD

What is DTD ?

- A DTD is a Document Type Definition.
- DTD allows to validate the contents of an XML documents
- A DTD defines the structure and the legal elements and attributes of an XML document.
- DTDs check vocabulary and validity of the structure of XML documents against grammatical rules of appropriate XML language.
- An XML DTD can be either specified inside the document, or it can be kept in a separate document and then linked separately.
- A DTD file has extension .dtd



DTD – Document Type Declaration

Basic syntax of a DTD is as follows –

```
<!DOCTYPE element DTD identifier  
[  
    declaration1  
    declaration2  
    .....  
>
```

In the above syntax,

- A DOCTYPE declaration in an XML document specifies the reference to be included as DTD file.
- The DTD starts with `<!DOCTYPE>` delimiter.
- An element tells the parser to parse the document from the specified root element.
- DTD identifier is an identifier for the document type definition, which may be the path to a file on the system or URL to a file on the internet.
- If the DTD is pointing to external path, it is called External Subset.
- The square brackets [] enclose an optional list of entity declarations called Internal Subset.

Types of DTD

Internal DTD – the contents of DTD are inside an XML document.

EXAMPLE:

```
<?xml version="1.0"?>
<!DOCTYPE book [<!ELEMENT book_name (#PCDATA)>]>
<book>
  <book_name> XML and Related Technologies
</book_name>
</book>
```

External DTD – the contents of DTD are in some external DTD file. The XML document has reference to that DTD file.

EXAMPLE:

```
<?xml version = "1.0" encoding = "UTF-8" standalone = "no" ?>
<!DOCTYPE address SYSTEM "dtd_1.dtd"> [dtd_1.dtd- external DTD file]
<address>
  <name>Atul Kahate</name>
  <company>GLS</company>
  <phone>079-25656565</phone>
</address>
```

Internal DTD

- A DTD is referred to as an internal DTD if elements are declared within the XML files.
- To refer it as internal DTD, standalone attribute in XML declaration must be set to yes.
- This means, the declaration works independent of an external source.

Syntax

`<!DOCTYPE root-element [element-declarations]>`

- root-element is the name of root element
- element-declarations is where you declare the elements.

Rules of Internal DTD

- The document type declaration must appear at the **start** of the document (preceded only by the XML header) – **it is not permitted anywhere else within the document.**
- Similar to the DOCTYPE declaration, the element declarations must start with an exclamation mark.
- The Name in the document type declaration must match the element type of the root element.

External DTD

- A DTD is referred to as an internal DTD if elements are declared within the XML files.
- To refer it as internal DTD, standalone attribute in XML declaration must be set to yes.
- This means, the declaration works independent of an external source.

Syntax

`<!DOCTYPE root-element [element-declarations]>`

root-element is the name of root element

element-declarations is where you declare the elements.

When to use Internal or External DTD

- For simple documents, an internal DTD can be used.
- External DTD allows to define a DTD once, and then refer to it from any number of XML documents. DTD can be reused
- External DTD reduces the size of XML documents.

Types of External DTD

External DTD

```
graph TD; A[External DTD] --> B[System DTD]; A --> C[Public DTD];
```

System DTD

Private DTD used only
in context of
particular environment

Eg:

```
<!DOCTYPE myBook  
SYSTEM  
"mybook.dtd">
```

Public DTD

Used beyond single
XML document. Used
with URL

Inserting Comments in a DTD

- The basic idea is that if you want to comment an element (or an attribute or anything else) in a DTD, you put your comment just before the element.
- The comment is delimited by `<!--` and `-->`, that is, pure XML.

Element Type Declaration

- Elements are backbone of an XML document.
- An element is declared in a DTD by using the Element Type Declaration (using ELEMENT tag)
 - Syntax: `<! ELEMENT <element_name> (#element_type)>`
Eg: `<! ELEMENT book_name (#PCDATA)>`
Eg: `<! ELEMENT book_name (#CDATA)>`
- The element name must be unique within a DTD

- XML documents are read and processed by a specific piece of software called an XML parser.
- When a document is processed by the XML parser, each character in the document is read, or parsed, in order to create a representation of the data.

PCDATA

- PCDATA is parsed, which means that the parser looks at each of the characters and tries to determine their meaning.
- For example,
 - If the parser encounters a < then it knows that the characters that follow represent an element instance.
 - If the parser encounters a /, it knows that it has encountered an end tag.
- Because PCDATA is parsed, it cannot contain <, >, and / characters, as these characters have special meaning in markup.

CDATA

- The term CDATA is used about text data that should not be parsed by the XML parser.
- Characters like "<" and "&" are illegal in XML elements.
- "<" will generate an error because the parser interprets it as the start of a new element.
- "&" will generate an error because the parser interprets it as the start of a character entity.
- Some text, like JavaScript code, contains a lot of "<" or "&" characters. To avoid errors script code can be defined as CDATA.
- Everything inside a CDATA section is ignored by the parser.
- A CDATA section starts with "<![CDATA[" and ends with "]]>":

DTD File

```
<!ELEMENT myBook (book_name)>  
<!ELEMENT book_name (#PCDATA)>
```

Document Type Reference to
internal DTD – book.dtd



XML File

```
<?xml version="1.0"?>  
<!DOCTYPE myBook SYSTEM "book.dtd">  
<myBook>  
    <book_name> XML techs</book_name>  
</myBook>
```

Document Type Reference
to external DTD – book.dtd



Root Element of dtd file
will be “myBook”



Valid and Well Formed Documents

- In the XML world, an XML document can be either well formed or valid or both.

Syntax (Well - Formedness)	Conformance to DTD (Validity)	Result
OK	NOT OK	WELL FORMED, BUT NOT VALID
OK	OK	WELL FORMED AND VALID
NOT OK	OK	NOT WELL FORMED, BUT VALID
NOT OK	NOT OK	NOT WELL FORMED AND NOT VALID

XML document (book.xml)

1. `<?xml version="1.0"?>`
2. `<!-- This XML document refers to book.dtd -->`
3. `<!DOCTYPE myBook SYSTEM "book.dtd">`
4. `<myBook>`
 `<book_name>XML</book_name>`
5. `</myBook>`

DTD file (book.dtd)

1. `<!ELEMENT myBook (author)>`
2. `<!ELEMENT author (#PCDATA)>`

**Case (a) Well-formed: Yes (syntax is correct);
Valid: No (DTD has an author element, but the
XML document has book name)**

XML document (book.xml)

1. `<?xml version="1.0"?>`
2. `<!-- This XML document refers
to book.dtd -->`
3. `<!DOCTYPE myBook SYSTEM "book.
dtd">`
4. `<myBook>`
 `<book_name>XML</book_name>`
5. `</myBook>`

DTD file (book.dtd)

1. `<!ELEMENT myBook (book_name)>`
2. `<!ELEMENT book_name (#PCDATA)>`

Case (b) Well-formed: Yes (syntax is correct)
Valid: Yes (DTD rules are followed by the XML document)

XML document (book.xml)

1. `<?xml version="1.0"?>`
2. `<!-- This XML document refers to book.dtd -->`
3. `<!DOCTYPE myBook SYSTEM "book.dtd">`
4. `<myBook>`
 `<book_name>XML</author>`
5. `</myBook>`

DTD file (book.dtd)

1. `<!ELEMENT myBook (author)>`
2. `<!ELEMENT author (#PCDATA)>`

Case (c) Well-formed: No (syntax is not correct – `book_name` opening tag is closed with an `author` tag); **Valid:** Yes (DTD rules are followed by the XML document)

XML document (book.xml)

1. `<?xml version="1.0"?>`
2. `<!-- This XML document refers to book.dtd -->`
3. `<!DOCTYPE myBook SYSTEM "book.dtd">`
4. `<myBook>`
 `<book_name>XML`
5. `</myBook>`

DTD file (book.dtd)

1. `<!ELEMENT myBook (book_name)>`
2. `<!ELEMENT book_name (#PCDATA)>`

Case (d) Well-formed: No (syntax is not correct, closing tag for book_name is missing); Valid: No (DTD specifies root element as booklist, which is not there in the XML document)

Sequence of Tags in XML Documents

Choices

Choices can be specified by using the pipe (|) character.

This allows us to specify options of the type A OR B. For example, we can specify that the result of an examination can be that the student has passed or failed (but not both), as follows:

```
<!ELEMENT result (pass | fail)>
```

Figure 3.16 shows a complete example. To a guest, we want to offer tea or coffee, but not both!

```
<!ELEMENT guest (name, purpose, beverage)>
```

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

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

```
<!ELEMENT beverage (tea | coffee)>
```

Figure 3.16 Specifying choices

Occurrences

The number of occurrences, or the frequency, of an element can be specified by using the plus (+), asterisk (*), or question mark (?) characters.

If we do not use any of the occurrence symbols (i.e., +, *, or ?), then the element can occur only once. That is, the default frequency of an element is 1.

The significance of these characters is tabulated in Table 3.2.

Character	Meaning
+	The element can occur one or more times
*	The element can occur zero or more times
?	The element can occur zero or one times

Table 3.2 Specifying frequencies of elements

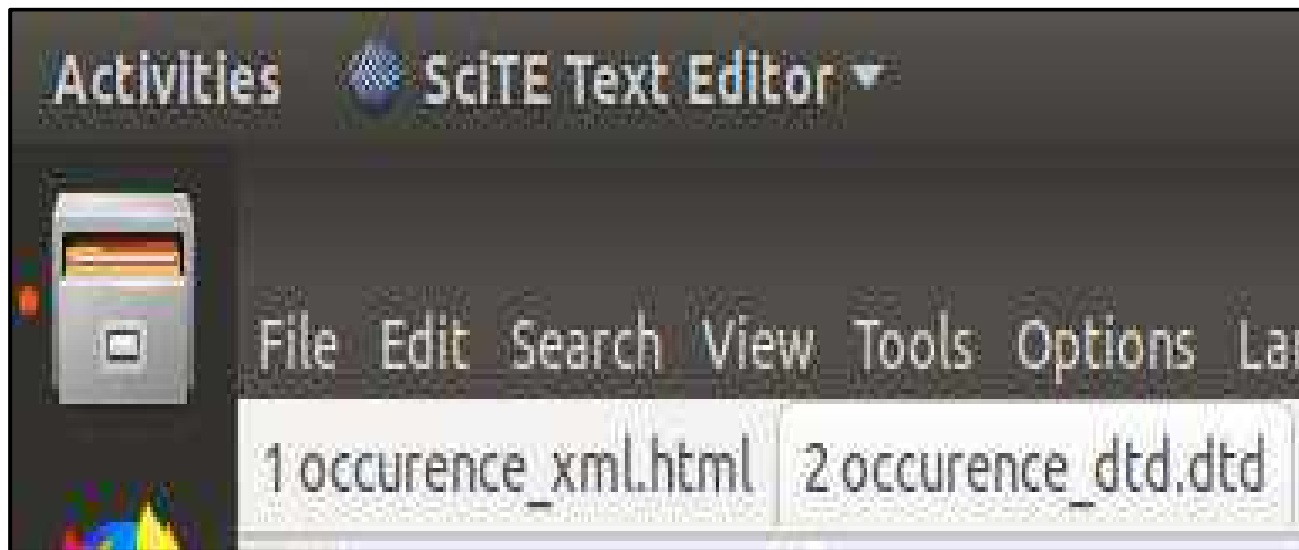
The plus sign (+) indicates that the element must occur at least once. The maximum frequency is infinite.

For example, we can specify that a book must contain one or more chapters as follows:

```
<!ELEMENT book (chapter+) >
```

We can use the same concept to apply to a group of sub-elements. For example, suppose that we want to specify that a book must contain a title, followed by at least one chapter and at least one author. We can use this declaration:

```
<!ELEMENT book (title, (chapter, author) + )>
```



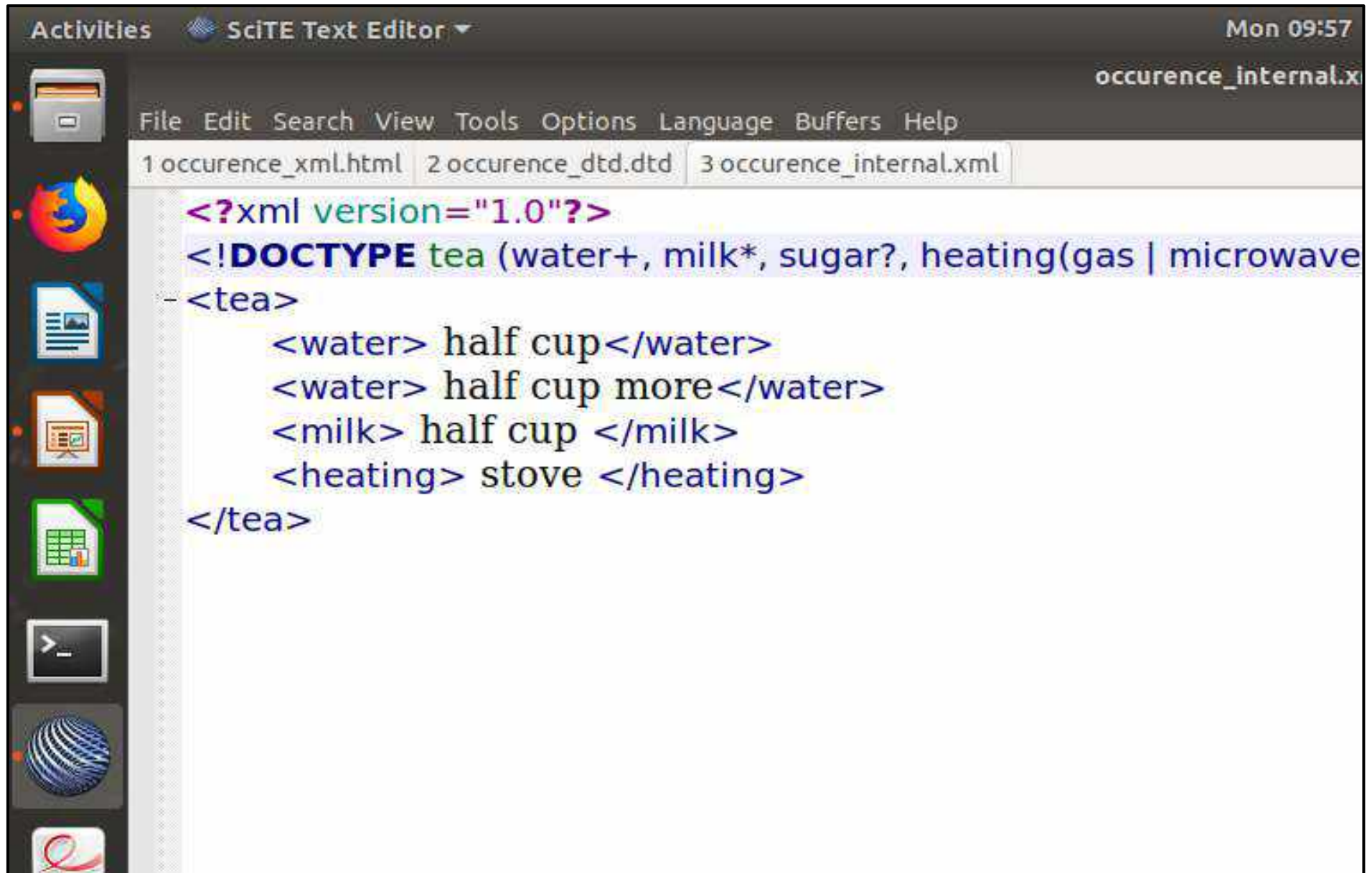
DTD File

XML file

A screenshot of the SciTE Text Editor application with the XML file '1 occurence_xml.html' open. The title bar shows 'Activities' and 'SciTE Text Editor'. The menu bar includes 'File', 'Edit', 'Search', 'View', 'Tools', 'Options', 'Language', and 'Buf'. The file list at the bottom shows two open files: '1 occurence_xml.html' and '2 occurence_dtd.dtd'. The XML content is as follows:

```
<?xml version="1.0"?>
<!DOCTYPE faculty SYSTEM "occurence_dtd.dtd">
<faculty>
  <details>
    <name> prof. ankita </name>
    <course> imscit </course>
    <sem> 3 </sem>
  </details>
  <details>
    <name> prof. purnima </name>
  </details>
  <details>
    <name> prof. riddhi </name>
    <course> imscit </course>
    <course> bca </course>
  </details>
  <details>
    <name> prof. riddhi </name>
    <course> imscit </course>
    <course> bca </course>
    <sem> 3 </sem>
  </details>
</faculty>
```

Internal XML Occurrence Example



```
<?xml version="1.0"?>
<!DOCTYPE tea (water+, milk*, sugar?, heating(gas | microwave
- <tea>
    <water> half cup</water>
    <water> half cup more</water>
    <milk> half cup </milk>
    <heating> stove </heating>
</tea>
```


Empty elements

An empty element is the one that can neither contain any data, nor any sub-elements.

In other words, an empty element is indivisible. It cannot be broken down further. Figure 3.21 shows an example of how to declare an element as empty, and how the corresponding content would appear in an XML document.

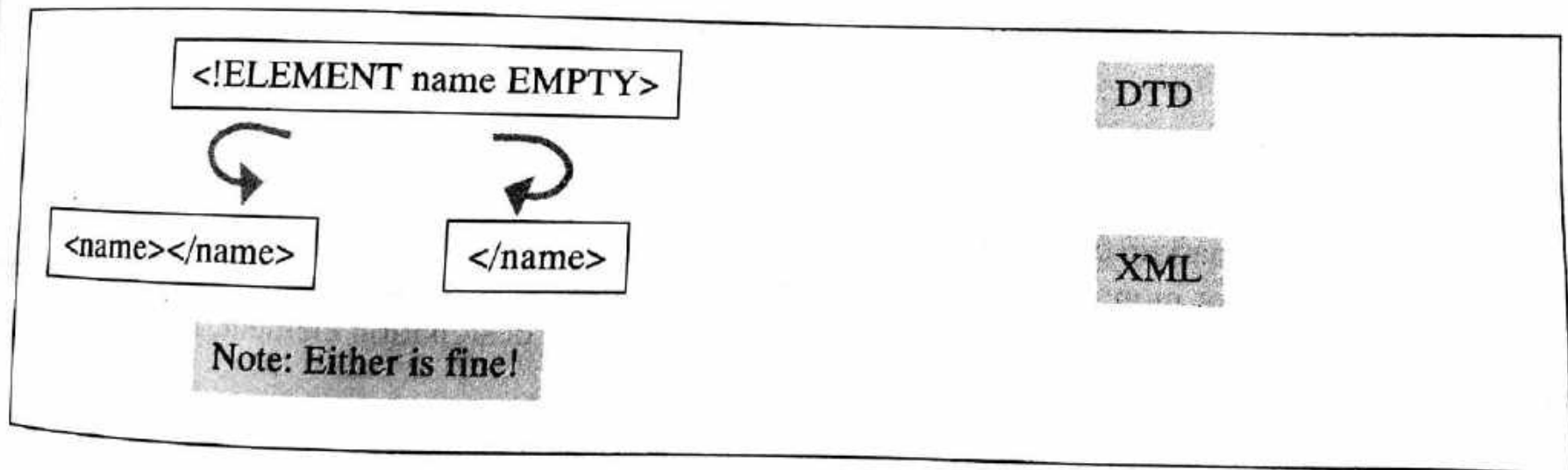


Figure 3.21 Declaring an empty element in a DTD and using it in an XML document

Mixed content

An element can contain either some text or other sub-elements. Such elements are said to have mixed content.

For example, consider the following declaration:

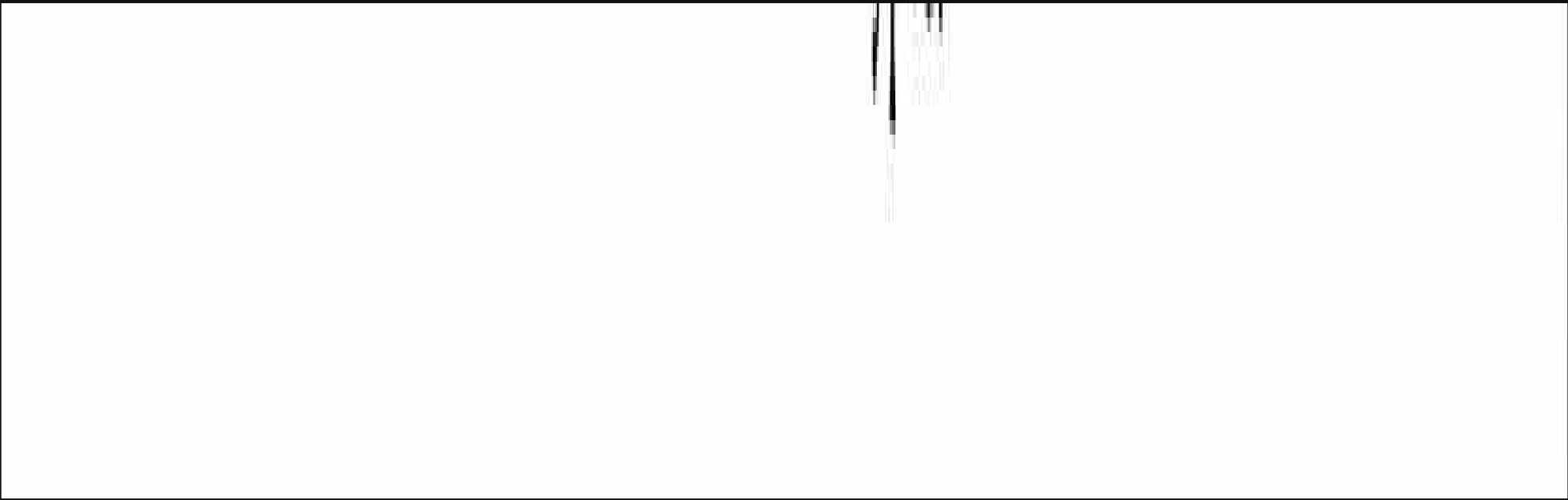
```
<!ELEMENT myBooks (book | #PCDATA) *>
```

This declaration states that an element `myBooks` can contain mixed content. It will allow the `myBooks` element to contain its own text, as well as zero or more sub-elements in the form of the `book` sub-element. An example of this is shown in Figure 3.22.

```
<myBooks>Possessing books is very rich!  
  <book>Computer networks by Tanenbaum</book>  
  <book>Operating systems</book>  
</myBooks>
```

Text content

Sub-element



Attribute Declaration

- Elements describe the markup of XML document.
- Markup are the tags of XML documents.
- Attributes describe the properties of Elements.
- ATTLIST is the keyword to describe attributes for an element

```
<?xml version ="1.0" ?>
<!DOCTYPE email [
  <!ELEMENT email          message) >
  <!ELEMENT message        #PCDATA)>
  <!ATTLIST message    from      CDATA #REQUIRED>
  <!ATTLIST message    to        CDATA #REQUIRED>
  <!ATTLIST message    subject   CDATA #REQUIRED>
]>

<email>
  <message from = "jui" to = "harshu" subject = "where are you?">
    It is time to have food!
  </message>
</email>
```

Figure 3.23 Declaring attributes in a DTD

Defaults for Attributes

- There are three types of default values for attributes.
 - **#IMPLIED** – specifies that if the attribute does not appear in the element, value can be anything
 - **#REQUIRED** – indicates that attributes must compulsory appear inside the elements.
 - **#FIXED** – specifies some constant value of attribute. The XML document cannot have any other value for that attribute

Activities SciTE Text Editor Thu 09:48

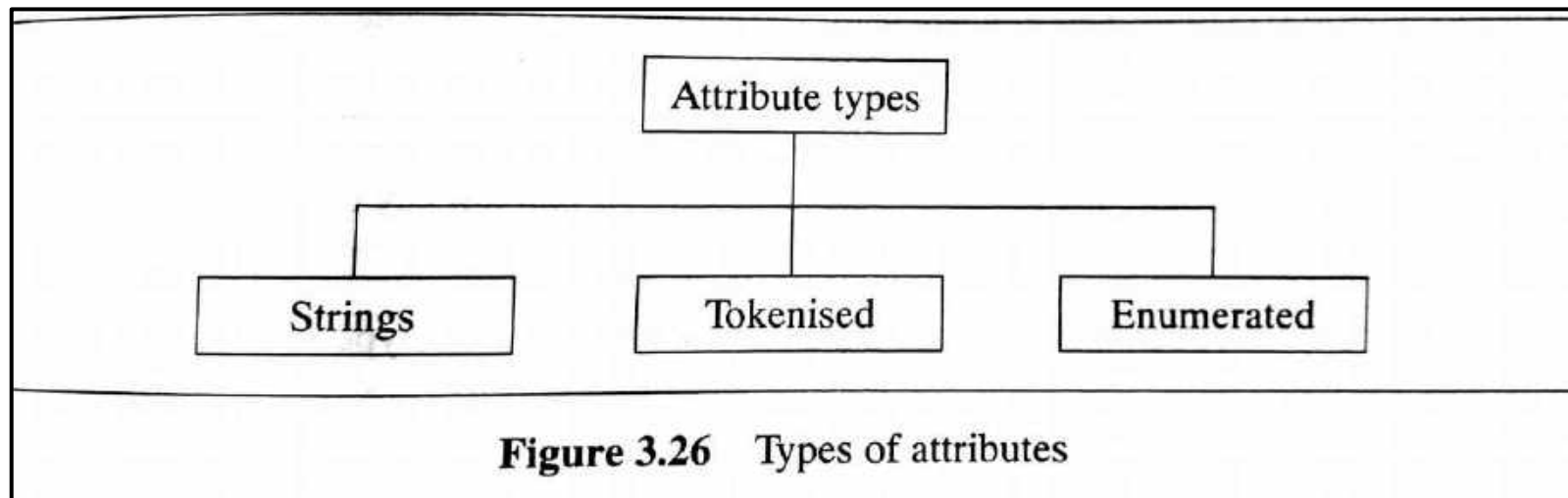
all_attributes_Demo.xml - SciTE

File Edit Search View Tools Options Language Buffers Help

1 all_attributes_Demo.xml

```
<?xml version="1.0"?>
  <!DOCTYPE email[
    <!ELEMENT email (message)>
    <!ELEMENT message (#PCDATA)>
    <!ATTLIST message from CDATA #IMPLIED>
    <!ATTLIST message to CDATA #REQUIRED>
    <!ATTLIST message subject CDATA #FIXED "hello">
  ]>
  <email>
    <message to="abc" from="abc" subject="hello">
      </message>
    </email>
```

Types of Attributes



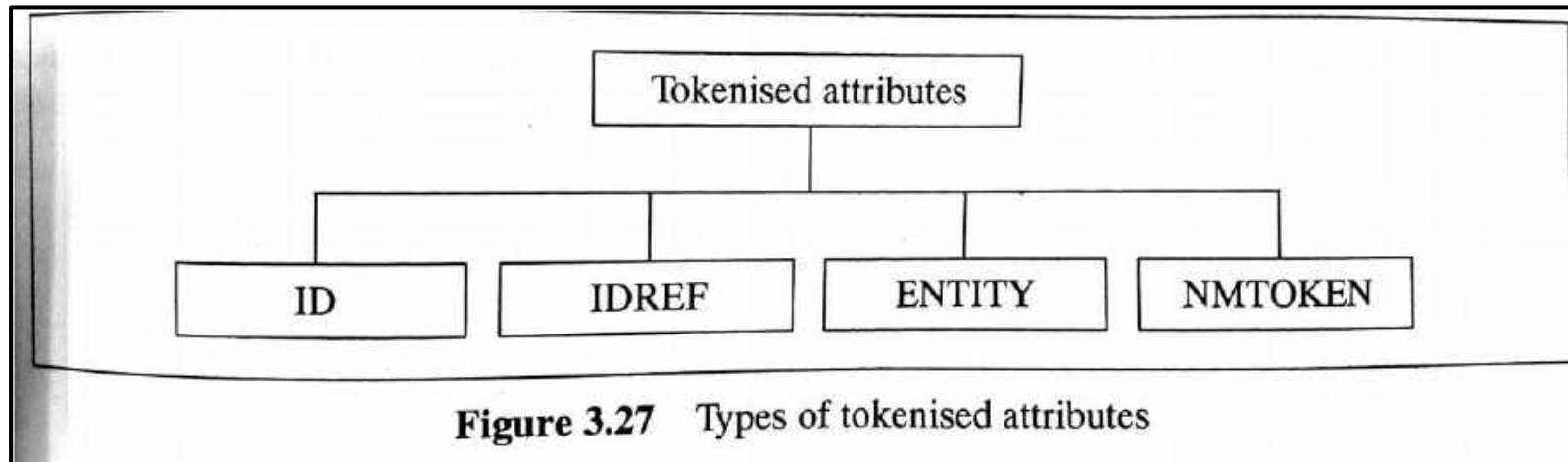
- Can be declared using CDATA keyword

Example:

<!ATTLIST name CDATA>

Tokenised Attributes

- Used to restrict some values to attributes



- **ID:** uniquely identifies an element. It is similar to the concept of *Primary Key*. ID helps to identify an element uniquely.
 - Two or more ID values cannot have the same value.
 - One element (tag) can have only one ID attribute.
- **IDREF:** is used for elements that refers to a specific ID attribute value. It is similar to the concept of *Foreign Key*.
 - If we use IDREF attribute in an XML document, XML parser checks for corresponding valid ID value.
- **IDREFS:** is used for elements that refers to a multiple ID attribute value.
- **NMTOKENS:** refers to attributes that can contain letters, digits, periods, underscores, hyphens and colon as values

ID and IDREF Example

```
<!DOCTYPE details[
  <!ELEMENT details (course+, student+ )>
  <!ELEMENT course (name)>
  <!ATTLIST course code ID #REQUIRED>
  <!ELEMENT name (#PCDATA)>
  <!ELEMENT student (#PCDATA)>
  <!ATTLIST student course_enrolled IDREF #IMPLIED]>
<details>
  <course code="c1">
    <name>Integrated MSc(IT)</name>
  </course>
  <course code="c2">
    <name> BCA</name>
  </course>
  <course code="c3">
    <name> pgdca</name>
  </course>
  <course code="c4"> <name> skdjfkadsh</name></course>
  <student course_enrolled="c1"> abc </student>
  <student course_enrolled="c2"> def</student>
  <student course_enrolled="c3">ghi</student>
  <student course_enrolled="c1">jsdflkdsaf </student>
</details>
```

NMTOKEN Example

```
<?xml version="1.0" standalone="yes"?>
<!DOCTYPE empinfo [
  <!ELEMENT empinfo (employee)+ >
  <!ELEMENT employee (contact)>
  <!ELEMENT contact (#PCDATA)>
  <!ATTLIST contact number NMTOKEN "079-95874568">
  <!ATTLIST contact name NMTOKEN "im_abc">
]>

<empinfo>
  <employee>
    <contact number="0789654125" name="im abc">landline</contact>
  </employee>
</empinfo>
```

As we

type as an element, rather than as an attribute now.

3.5 CONDITIONAL SECTIONS

Programmers often write conditional sections of a code, which execute only if the condition is satisfied. For example, some codes will execute only if the operating system is UNIX, or only if the program is being executed in the debug mode, and so on.

DTDs allow us to write conditional sections, with some basic features. We need them when we want to modify an existing DTD, but do not wish to rewrite it from scratch. Instead, we can use an existing DTD, and modify it to add conditional sections. These sections come into the picture only if the specified condition is satisfied. Otherwise, the original DTD is in effect, as before. The basic syntax of a conditional section in a DTD is shown in Figure 3.41.

```
<![keyword  
[  
    ... conditional DTD declarations ...  
]  
>
```

Figure 3.41 Conditional DTD section syntax

The keyword can be INCLUDE or IGNORE.

- When we specify INCLUDE, the conditional declarations inside this section are considered for validations.
- When we specify IGNORE, the conditional declarations inside this section are not considered for validations.

These can be used only in an external subset (i.e., an external DTD). They cannot be used in the case of an internal subset (i.e., an internal DTD).

The trouble is, unlike a programming language, we do not have an *if-else* or an equivalent construct in the case of a DTD. Therefore, there is no straightforward way to control the inclusion or ignorance of conditional declaration blocks. In other words, we cannot say:

```
if ~
<![INCLUDE ~
]>
```

Therefore, it may seem that these conditional blocks are useless. After all, whatever we write inside an INCLUDE block will always get included, and whatever we write inside an IGNORE block will always get excluded! Figure 3.42 shows this with an example. There is no difference between the declarations (a) and (b).

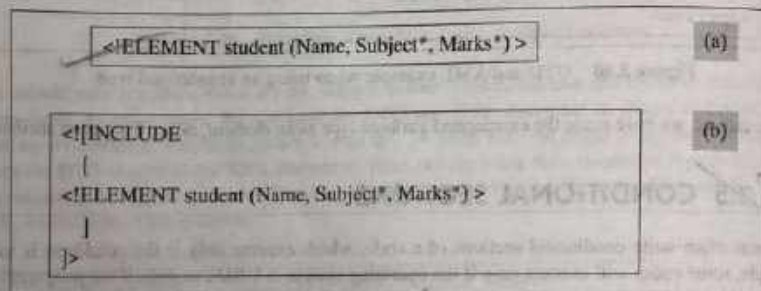


Figure 3.42 Example of INCLUDE

Does it matter whether we use INCLUDE (or IGNORE)? If we use INCLUDE, the declaration would be considered as a part of the DTD. But then, even if we do not use it, it would still be considered anyway! The same is the case for IGNORE, in terms of excluding it.

The solution to this problem is to use parameter entities. This allows our conditional block to be either INCLUDE or IGNORE, without changes. A parameter entity is used only in DTDs. The basic format of a parameter entity is shown in Figure 3.43.

```
<ENTITY % name "substitution text" >
```

Figure 3.43 Parameter entity format

Parameter entities are useful for defining blocks of text or DTD entries that repeat. An example of this is shown in Figure 3.44.

```
<ENTITY % CDATA_Req    "CDATA    #REQUIRED" >
<ENTITY % CDATA_Opt    "CDATA    #IMPLIED" >
<ENTITY % IDREF_Req    "CDATA    #REQUIRED" >
<ENTITY % IDREF_Opt    "CDATA    #IMPLIED" >
```

Figure 3.44 Parameter entity example

We could then use these entity names inside an XML document, as shown in Figure 3.45.

```
<!ATTLIST StudentInfo
  Student_code    #IDREF_Req
  Student_Name    #CDATA_Req
  Student_Division #IDREF_Req
  Student_Address #CDATA_Opt
>
```

Figure 3.45 Usage of parameter entities

These concepts can be used in defining our DTD now. Figure 3.46 shows the example.

```
<ENTITY % useTeachers    "IGNORE">
<ENTITY % useNoTeachers  "INCLUDE">
<![%useTeachers;
|
  <ELEMENT School (Student, Subject*, Marks*, Teacher*) >
  <ELEMENT Teacher (#PCDATA) >
|
]>
<![%useNoTeachers;
|
  <ELEMENT School (Student, Subject*, Marks*) >
|
]>
<ELEMENT Student (#PCDATA) >
<ELEMENT Subject (#PCDATA) >
<ELEMENT Marks (#PCDATA) >
```

Figure 3.46 Defining parameter entities

An XML document that uses the above DTD will automatically be able to use the version that does not have the teacher information. This is clearly because in the current version, the *INCLUDE* is against *useNoTeachers*. But, what about an XML document that is interested in also capturing the teacher information? It will need to modify the above DTD. It will need to swap the *IGNORE* and *INCLUDE* statements, making the DTD look like the one shown in Figure 3.47.

```
<!ENTITY % useTeachers      "INCLUDE">
<!ENTITY % useNoTeachers    "IGNORE">
<![%useTeachers;
|
|  <!ELEMENT School (Student, Subject*, Marks*, Teacher*) >
|  <!ELEMENT Teacher (#PCDATA) >
|
|>
<![%useNoTeachers;
|
|  <!ELEMENT      School (Student, Subject*, Marks*) >
|
|>
<!ELEMENT  Student (#PCDATA) >
<!ELEMENT  Subject (#PCDATA) >
<!ELEMENT  Marks  (#PCDATA) >
```

Figure 3.47 Modifying the DTD to have the opposite effect

Here, we are saying that any XML document that wants the teacher information also to be captured, needs to use the *useTeachers* entity by setting it to *INCLUDE* (instead of the current *IGNORE*). Note that we have now set the *useNoTeachers* entity to *IGNORE*.

But, we have a problem here. We know that one external DTD serves the purpose of many XML documents. That is, one external DTD can be used by many XML documents. In this context, how can we go on swapping the *INCLUDE* and *IGNORE* statements in a DTD on demand? Many XML documents may want the *useTeachers* entity to be set to *INCLUDE*, whereas at the same time, many others may want it to be set to *useNoTeachers*!

This problem is solved as follows. Once we are reasonably sure of our parameter entity definition, we can remove the two *<!ENTITY ...>* statements from our DTD above. Instead, we add them to our respective XML documents. That is:

1. An XML document that is not interested in using teacher information would set the *useNoTeachers* entity to *INCLUDE*, and the *useTeachers* entity to *IGNORE*.

An example of this is shown in Figure 3.48.

```
<?xml version = "1.0" ?>
<!DOCTYPE school SYSTEM "school.dtd"
|
|  <!ENTITY % useTeachers      "IGNORE">
|  <!ENTITY % useNoTeachers    "INCLUDE">
|>
<School>
  <Student> ... </Student>
  <Subject> ... </Subject>
  <Marks> ... </Marks>
</School>
```

Figure 3.48 Example of ignoring the teacher information

2. An XML document that is interested in using teacher information would set the *useTeachers* entity to *INCLUDE*, and the *useNoTeachers* entity to *IGNORE*.

An example of this is shown in Figure 3.49.

```
<?xml version = "1.0" ?>
<!DOCTYPE school SYSTEM "school.dtd"
|
|  <!ENTITY % useTeachers      "INCLUDE">
|  <!ENTITY % useNoTeachers    "IGNORE">
|>
<School>
  <Student> ... </Student>
  <Subject> ... </Subject>
  <Marks> ... </Marks>
  <Teacher> ... </Teacher>
</School>
```

Figure 3.49 Example of considering the teacher information

As a technical aside, even if we retain the entity definitions in our DTD, there is no problem. Internal DTD declarations are always given preference over external DTD declarations. Therefore, even if we have entity declarations both in the internal DTD (i.e., inside the XML document) as well as in the external DTD, the internal would take preference. It would override whatever is specified in the external DTD.

Limitation	Explanation
Non-XML syntax	Although DTDs do have the angled bracket syntax (for example, <code><!ELEMENT ...></code>), this is quite different from the basic XML syntax. For example, a DTD does not have the standard <code><?xml version ...?></code> tag, etc. More specifically, a DTD file is not a valid XML document. This means duplication of validating efforts – one logic for XML, another for DTD.
One DTD per XML	We cannot use multiple DTDs to validate one XML document. We can include only one DTD reference inside an XML document. Although parameter entities make things slightly more flexible, their syntax is quite cryptic.
Weak data typing	DTD defines basic data types. For real-life applications that demand more fine-grained and specific data types, this is not sufficient in many situations.
No inheritance	DTDs are not object-oriented in the sense that they do not allow the designer to create data types and extend them as desired.
Overriding a DTD	An internal DTD can override an external DTD. (This is perhaps the DTD's idea of inheritance!). This allows certain flexibility, but often creates a lot of confusion and leads to clumsy designs.
No DOM support	We shall study later that the Document Object Model (DOM) technology is used to parse, read, and edit XML documents. It cannot be used for DTDs, though.
Table 3.3 Limitations of DTDs	