**DTD**

**DTD – Web Services:**

DTD stands for Document Type Definition.

It is the document which defines the structure of an XML document.

**As we have two types of XML elements:**

* 1. Simple elements and
  2. Compound elements

We need to represent in DTD, my xml contains these elements which are of either simple or compound. So, we need to understand the syntax of how to declare a simple element of an xml in dtd as well as compound element.

**Syntax for Declaring Simple Element of an XML**

**<!Element elementname (#PCDATA)>**

Here we declared that my xml should contain an element whose name is

elementname which contains parse able character data.

**Syntax for Compound element of an XML**

**<!Element elementname (sub-elem1, sub-elem2…)**

Here we declared I have an element “elementname” which contains sub elements

under it sub-elem1, sub-elem2 etc.

**For example the sample DTD document for an xml is shown below.**

**XML Document**

*<?xml version=”1.0” encoding=”utf-8”?>*

*<purchaseOrder>*

*<orderItems>*

*<item>*

*<itemCode>IC323</itemCode>*

*<quantity>24</quantity>*

*</item>*

*<item>*

*<itemCode>IC324</itemCode>*

*<quantity>abc</quantity>*

*</item>*

*</orderItems>*

*<purchaseOrder>*

**For the above xml the DTD looks as shown below.**

*<?xml version=”1.0” encoding=”utf-8”?>*

*<!Element purchaseOrder (orderItems)>*

*<!Element orderItems (item+)>*

*<!Element item (itemCode, quantity)>*

*<!Element itemCode (#PCDATA)>*

*<!Element quantity (#PCDATA)>*

**Occurrences of an Element under another element**

In the above xml, if you observe the orderItems element can contain any number of

item elements in it, but at least one item element must be there for a purchaseOrder. This is called **occurrence of an element** under another element, to indicate this we use three symbols.

**? –**

Represents the sub element under a parent element can appear zero or onetime (0/1).

**+ -**

indicates the sub element must appear at least once and can repeat any

number of times (1 – N)

**\* -**

indicates the sub element is optional and can repeat any number of times (0 - N)

**You will mark the occurrence of an element under another element as follows**

***<!Element elementname (sub-elem1 (?/+/\*), sub-elem2(?/+/\*)>***

Leaving any element without any symbol indicates it is mandatory and at max can repeat only once.

**Declaring attribute for an element Syntaxes:**

***<!ATTLIST elementname attributename (val1|val2) “default value”>***

***<!ATTLIST elementname attributename #FIXED “value”>***

***<!ATTLIST elementname attributename #REQUIRED>***

***<!ATTLIST elementname attributename #IMPLIED>***

As shown above to declare an attribute you need to use the tag ATTLIST and elementname stands for which element you want to declare the attribute and attributename stands for what is the attribute you want to have in that element.

**Val1|Val2 :**

This indicates the element attribute can carry possible values as either val1 or val2.

**#FIXED :**  
 Indicates only it can carry a fixed value you mentioned in quote besides it.

**#REQUIRED :**

In this case you are not specifying any value but the presence of that attribute is mandatory

**#IMPLIED :**

If the attribute is not mandatory and doesn’t have any default value then you should declare it as implied

**Drawback with DTD’s**

DTD are not type safe, which means when we declare simple elements we indicate it should contain data of type (#PCDATA).

#PCDATA means **parsable character data** means any data that is computer represented format. So, it indicates an element can contain any type of data irrespective of whether it is int or float or string. You cannot impose stating my element should contain int type data or float. This is the limitation with DTD documents.

**Web Sites :**

XML **Document Type Declaration**, commonly known as DTD, is a way to describe precisely the XML language. DTD’s checks the validity of structure and vocabulary of an XML document against the grammatical rules of the appropriate XML language.

A DTD defines the structure and the legal elements and attributes of an XML document.

**Why Use a DTD?**

With a DTD, independent groups of people can agree on a standard DTD for interchanging data.

An application can use a DTD to verify that XML data is valid.

**An XML document can be defined as**:

**Well-formed**:

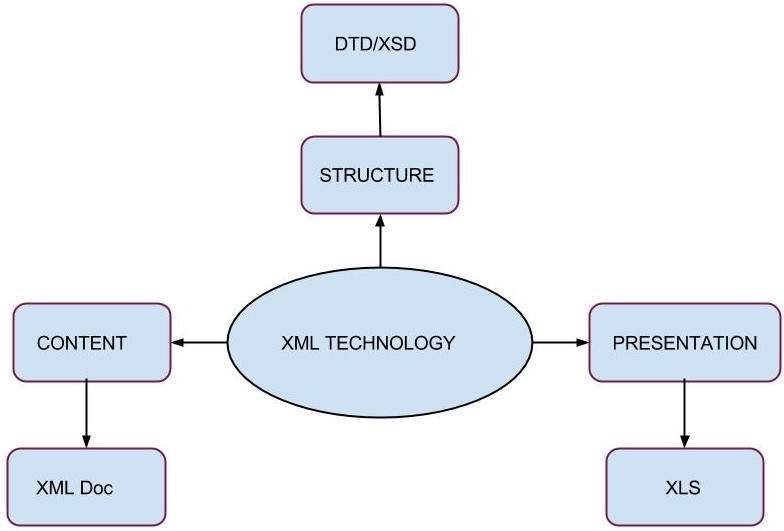
If the XML document adheres to all the general XML rules such as tags must be properly nested, opening and closing tags must be balanced, and empty tags must end with '/>', then it is called as ***well-formed***.

**OR**

**Valid**:

An XML document said to be valid when it is not only *well-formed*, but it also conforms to available DTD that specifies which tags it uses, what attributes those tags can contain, and which tags can occur inside other tags, among other properties.

**The following diagram represents that a DTD is used to structure the XML document:**



**Types**

DTD can be classified on its declaration basis in the XML document, such as:

* **Internal DTD**
* **External DTD**

When a DTD is declared within the file it is called **Internal DTD** and if it is declared in a separate file it is called **External DTD**.

**An Internal DTD Declaration**

If the DTD is declared inside the XML file, it must be wrapped inside the <!DOCTYPE> definition:

**XML document with an internal DTD**

*<?xml version="1.0"?>  
<!DOCTYPE note [  
<!ELEMENT note (to,from,heading,body)>  
<!ELEMENT to (#PCDATA)>  
<!ELEMENT from (#PCDATA)>  
<!ELEMENT heading (#PCDATA)>  
<!ELEMENT body (#PCDATA)>  
]>  
<note>  
<to>Tove</to>  
<from>Jani</from>  
<heading>Reminder</heading>  
<body>Don't forget me this weekend</body>  
</note>*

## **An External DTD Declaration**

If the DTD is declared in an external file, the <!DOCTYPE> definition must contain a reference to the DTD file:

### **XML document with a reference to an external DTD**

*<?xml version="1.0"?>****<!DOCTYPE note SYSTEM "note.dtd">****<note>  
  <to>Tove</to>  
  <from>Jani</from>  
  <heading>Reminder</heading>  
  <body>Don't forget me this weekend!</body>  
</note>*

**And here is the file "note.dtd", which contains the DTD:**

*<!ELEMENT note (to,from,heading,body)>  
<!ELEMENT to (#PCDATA)>  
<!ELEMENT from (#PCDATA)>  
<!ELEMENT heading (#PCDATA)>  
<!ELEMENT body (#PCDATA)>*

**Features :**

**Following are some important points that a DTD describes:**

* The elements that can appear in an XML document.
* The order in which they can appear.
* Optional and mandatory elements.
* Element attributes and whether they are optional or mandatory.
* Whether attributes can have default values.

**Advantages of using DTD**

* **Documentation -**

You can define your own format for the XML files. Looking at this document a user/developer can understand the structure of the data.

* **Validation –**

It gives a way to check the validity of XML files by checking whether the elements appear in the right order, mandatory elements and attributes are in place, the elements and attributes have not been inserted in an incorrect way, and so on.

**Disadvantages of using DTD**

* It does not support the **namespaces**. Namespace is a mechanism by which element and attribute names can be assigned to groups. However, in a DTD namespaces have to be defined within the DTD, which violates the purpose of using namespaces.
* It supports only the text string data type i.e. it is just used for **structural validation**. It does not support the data validation.

It doesn’t check whether the data we entered is a numeric value or character value.

***<name>Srini</name> and <name>123</name>***

* For DTD’s the above two are valid only.
* It is not object oriented. Hence, the concept of inheritance cannot be applied on the DTDs.
* Limited possibilities to express the cardinality for elements.

**DTD – Syntax**

An XML DTD can be either specified inside the document, or it can be kept in a separate document and then the document can be linked to the DTD document to use it.

**Syntax**

**Basic syntax of a DTD is as follows**:

***<!DOCTYPE element DTD identifier***

***[***

***declaration1***

***declaration2***

***........***

***]>***

**In the above syntax**

* **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**.

**DTD - XML Building Blocks**

## **The Building Blocks of XML Documents**

Seen from a DTD point of view, all XML documents are made up by the following building blocks:

* *Elements*
* *Attributes*
* *Entities*
* *PCDATA*
* *CDATA*

## **Elements**

Elements are the **main building blocks** of both XML and HTML documents.

**Examples** **of HTML** elements are "body" and "table".

**Examples** **of XML** elements could be "note" and "message".

Elements can contain text, other elements, or be empty.

**Examples of empty HTML** elements are "hr", "br" and "img".

**Examples**:

*<****body****>some text</****body****>  
<****message****>some text</****message****>*

## **Attributes**

Attributes provide **extra information about elements**.

Attributes are always placed inside the opening tag of an element. Attributes always come in name/value pairs.

**The following "img" element has additional information about a source file:**

***<img src="computer.gif" />***

The name of the element is "img". The name of the attribute is "src". The value of the attribute is "computer.gif".

Since the element itself is empty it is closed by a " /".

## **Entities**

Some characters have a special meaning in XML, like the less than sign (<) that defines the start of an XML tag.

**Most of you know the HTML entity:** "&nbsp;”.

This "**no-breaking-space**" entity is used in HTML to insert an extra space in a document. Entities are expanded when a document is parsed by an XML parser.

**The following entities are predefined in XML:**

|  |  |  |
| --- | --- | --- |
| **Entity References** |  | **Character** |
| &lt; |  | < |
| &gt; |  | > |
| &amp; |  | & |
| &quot; |  | " |
| &apos; |  | ' |

## **PCDATA**

**PCDATA means parsed character data.**

Think of character data as the text found between the start tag and the end tag of an XML element.

**PCDATA is text that WILL be parsed by a parser**. **The text will be examined by the parser for entities and markup**.

Tags inside the text will be treated as markup and entities will be expanded.

However, parsed character data should not contain any &, <, or > characters; these need to be represented by the &amp; &lt; and &gt; entities, respectively.

## **CDATA**

**CDATA means character data.**

**CDATA is text that will NOT be parsed by a parser**.

Tags inside the text will NOT be treated as markup and entities will not be expanded

**Internal DTD**

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

**Syntax:**

**The syntax of internal DTD is as shown:**

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

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

**Example**

Following is a simple example of internal DTD:

*<?xml version="1.0" encoding="UTF-8****" standalone="yes"*** *?>*

*<!DOCTYPE address [*

*<!ELEMENT address (name,company,phone)>*

*<!ELEMENT name (#PCDATA)>*

*<!ELEMENT company (#PCDATA)>*

*<!ELEMENT phone (#PCDATA)>*

*]>*

*<address>*

*<name>Srinivas B</name>*

*<company>TutorialsPoint</company>*

*<phone>(011) 123-4567</phone>*

*</address>*

**Let us go through the above code:**

**Start Declaration**-

Begin the XML declaration with following statement

***<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>***

**DTD** -

Immediately after the XML header, the *document type declaration* follows, commonly referred to as the DOCTYPE:

***<!DOCTYPE address [***

The DOCTYPE declaration has an exclamation mark (!) at the start of the element name. **The DOCTYPE informs the parser that a DTD is associated with this XML document.**

**DTD Body**-

The DOCTYPE declaration is followed by body of the DTD, where you declare elements, attributes, entities, and notations:

*<!ELEMENT address (name,company,phone)>*

*<!ELEMENT name (#PCDATA)>*

*<!ELEMENT company (#PCDATA)>*

*<!ELEMENT phone\_no (#PCDATA)>*

Several elements are declared here that make up the vocabulary of the <name> document. <!ELEMENT name (#PCDATA)> defines the element *name* to be of type "#PCDATA".

Here #**PCDATA means parse-able text data**.

**End Declaration** –

Finally, the declaration section of the DTD is closed using a closing bracket and a closing angle bracket (]>). This effectively ends the definition, and thereafter, the XML document follows immediately.

**Rules**

* 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**

In external DTD elements are declared outside the XML file. They are accessed by specifying the system attributes which may be either the legal *.dtd* file or a valid URL.

To reference it as external DTD, ***standalone* attribute in the XML declaration must be set as no.** This means, declaration includes information from the external source.

**Syntax**

**Following is the syntax for external DTD:**

***<!DOCTYPE root-element SYSTEM "file-name">***

Where *file-name* is the file with *.dtd* extension.

**Example**

**The following example shows external DTD usage:**

*<?xml version="1.0" encoding="UTF-8" standalone="no" ?>*

*<!DOCTYPE address SYSTEM "address.dtd">*

*<address>*

*<name>Srinivas B</name>*

*<company>TutorialsPoint</company>*

*<phone>(011) 123-4567</phone>*

*</address>*

The content of the DTD file **address.dtd** are as shown:

*<!ELEMENT address (name,company,phone)>*

*<!ELEMENT name (#PCDATA)>*

*<!ELEMENT company (#PCDATA)>*

*<!ELEMENT phone (#PCDATA)>*

**Types**

You can refer to an external DTD by either using **system identifiers** or **public identifiers**.

**System Identifiers**

A system identifier enables you to specify the location of an external file containing DTD declarations. Syntax is as follows:

***<!DOCTYPE name SYSTEM "address.dtd" [...]>***

As you can see it contains keyword SYSTEM and a URI reference pointing to the location of the document.

**Public Identifiers**

Public identifiers provide a mechanism to locate DTD resources and are written as below:

***<!DOCTYPE name PUBLIC "-//Beginning XML//DTD Address Example//EN">***

As you can see, it begins with keyword PUBLIC, followed by a specialized identifier. Public identifiers are used to identify an entry in a catalog. Public identifiers can follow any format; however, a commonly used format is called ***Formal Public Identifiers, or FPIs****.*

**DTD – Components**

A DTD will basically contain declarations of the following XML components:

* *Element*
* *Attributes*
* *Entities*

1. **Elements**

XML elements can be defined as building blocks of an XML document. Elements can behave as a container to hold text, elements, attributes, media objects or mix of all.

Each XML document contains one or more elements, the boundaries of which are either delimited by start-tags and end-tags, or empty elements.

**Example**

Below is a simple example of XML elements

***<name>Tutorials Point</name>***

As you can see we have defined a <name> tag. There's a text between start and end tag of <name>.

1. **Attributes**

Attributes are part of the XML elements. An element can have any number of unique attributes. Attributes give more information about the XML element or more precisely it defines a property of the element. An XML attribute is always a *name-value* pair.

**Example**

Below is a simple example of XML attributes:

***<img src="flower.jpg"/>***

Here *img* is the element name whereas *src* is an attribute name and *flower.jpg* is a value given for the attribute *src*.

1. **Entities**

Entities are placeholders in XML. These can be declared in the document prolog or in a DTD.

**Entities can be primarily categorized as:**

* Built-in entities
* Character entities
* General entities
* Parameter entities

**There are five built-in entities that play in well-formed XML, they are:**

* ampersand: &amp;
* Single quote: &apos;
* Greater than: &gt;
* Less than: &lt;
* Double quote: &quot;

**DTD – Elements**

XML elements can be defined as building blocks of an XML document. Elements can behave as a container to hold text, elements, attributes, media objects or mix of all.

A DTD element is declared with an ELEMENT declaration. When an XML file is validated by DTD, parser initially checks for the root element and then the child elements are validated.

**Syntax**

All DTD element declarations have this general form:

***<!ELEMENT elementname (content)>***

* ***ELEMENT*** declaration is used to indicate the parser that you are about to define an element.
* ***elementname*** is the element name (also called the ***generic identifier***) that you are defining.
* ***content*** defines what content (if any) can go within the element.

**DTD Element Rules**

All data contained in an element must follow a set rule. As stated previously, the rule is the definition to which the element's data content must conform. There are two basic types of rules that elements must fall into.

The first type of rule deals with **content**. The second type of rule deals with **structure**.

##### Content Rules

The content rules for .elements deal with the actual data that defined elements may contain. These rules include the ANY rule, the EMPTY rule, and the #PCDATA rule.

###### **The ANY Rule**

An element may be defined. using the ANY rule. This rule is just what it sounds like: The element may contain other elements and/or normal character data (just about anything as long as it is well formed). An element using the ANY rule would appear as follows:

***<!ELEMENT elementname ANY>***

The drawback to this rule is that it is so wide open that it defeats the purpose of validation. A DTD that defines all its elements using the ANY rule will always be valid as long as the XML is well formed. This really precludes any effective validation.

#### **XML Fragments Using the ANY Rule**

*<elementname>*

*This is valid content*

*</elementname>*

*<elementname>*

*<anotherelement>*

*This is more valid content*

*</anotherelement>*

*This is still valid content*

*</elementname>*

*<elementname>*

*<emptyelement />*

*<yetanotherelement>*

*This is still valid content!*

*</yetanotherelement>*

*Here is more valid content*

*</elementname>*

You should see from this listing why it is not always a great idea to use the ANY rule.

All three fragments containing the element elementname are valid. There is, in effect, no validation for this element. Use of the ANY rule should probably be limited to instances where the XML data will be freeform text or other types of data that will be highly variable and have difficulty conforming to a set structure.

###### ***The* EMPTY *Rule***

This rule is the exact opposite of the ANY rule. An element that is defined with this rule will contain no data. However, an element with the EMPTY rule could still contain attributes (more on attributes in a bit).

The following element is an example of the EMPTY rule:

**<!ELEMENT elementname EMPTY>**

This concept is seen a lot in HTML. There are many tags such as the break tag (<br />) and the paragraph tag (<p />) that follow this rule. Neither one of these tags contains any data, but both are very important in HTML documents. The best example of an empty tag used in HTML is the image tag (<img>). Even though the image tag does not contain any data, it does have attributes that describe the location and display of an image for a Web browser.

In XML, the EMPTY rule might be used to define empty elements that contain diagnostic information for the processing of data. Empty elements could also be created to hold metadata describing the contents of the XML document for indexing purposes. Empty elements could even be used to provide clues for applications that will render the data for viewing (such as an empty "gender" tag, which designates an XML record as "male" or "female"—male records could be rendered in blue, and female records could be rendered in pink) .

###### **The #PCDATA Rule**

The #PCDATA rule indicates that parsed character data will be contained in the element. Parsed character data is data that may contain normal markup and will be interpreted and parsed by any XML parser accessing the document.

**The following element demonstrates the #PCDATA rule:**

<!ELEMENT elementname (#PCDATA)>

**An element in an XML document that adheres to the #PCDATA rule might appear as follows:**

*<data>*

*This is some parsed character data*

*</data>*

It is possible in an element using the #PCDATA rule to use the CDATA keyword to prevent the character data from being parsed.

#### **CDATA**

*<sample>*

*<data>*

*<![CDATA[<tag>This will not be parsed</tag>]]>*

*</data>*

*</sample>*

All the data between <![CDATA[ and ]]> will be ignored by the parser and treated as normal characters (markup ignored).

##### Structure Rules

Whereas the content rules. deal with the actual content of the data contained in defined elements, structure rules deal with how that data may be organized. There are two types of structure rules we will look at here. The first is the "element only" rule. The second rule is the "mixed" rule.

###### **The "Element Only" Rule**

The "**element only**" rule. specifies that only elements may appear as children of the current element. The child element sequences should be separated by commas and listed in the order they should appear. If there are to be options for which elements will appear, the listed elements should be separated by the pipe symbol (|). The following element definition demonstrates the "element only" rule:

<!ELEMENT elementname (element1, element2, element3)>

You can see here that a list of elements is expected to appear as child elements of elementname when the referencing XML document is parsed. All these child elements must be present and in the specified order. Here is how an element that is listing a series of options will appear:

<!ELEMENT elementname (element1 | element2)>

**The element defined here will have a single child element:**

either element1 or element2.

###### ***The "Mixed" Rule***

The "mixed" rule is used to help define elements that may have both character data (#PCDATA) and child elements in the data they contain. A list of options or a sequential list will be enclosed by parentheses. Options will be separated by the pipe symbol (|), whereas sequential lists will be separated by commas. The following element is an example of the "mixed" rule:

<!ELEMENT elementname (#PCDATA | childelement1 | childelement2)\*>

In this example, the element may contain a mixture of character data and child elements. The pipe symbol is used here to indicate that there is a choice between #PCDATA and each of the child elements. However, the asterisk symbol (\*) is added here to indicate that each of the items within the parentheses may appear zero or more times (we will cover the use of element symbols in the next section). This can be useful for describing data sets that have optional values. Consider the following element definition:

**NOTE**

The asterisk symbol used in these examples indicates that an item may occur zero or more times. Element symbols are covered in detail in Table 3.1.

<!ELEMENT Son (#PCDATA | Name | Age)\*>

This definition defines an element, Son, for which there may be character data, elements, or both. A man might have a son, but he might not. If there is no son, then normal character data (such as "N/A") could be used to describe this condition. Alternatively, the man might have an adopted son and would like to indicate this. Consider the XML fragments shown in Listing 3.9 in relation to the definition for the element Son.

#### **The "Mixed" Rule**

*<Son>*

*N/A*

*</Son>*

*<Son>*

*Adopted Son*

*<Name>Bobby</Name>*

*<Age>12</Age>*

*</Son>*

The first fragment contains only character data. The second fragment contains a mixture of character data and the two defined child elements. Both fragments conform to the definition and are valid.

#### **Element Symbols**

In addition to the normal rules that apply to element definitions, element symbols can be used to control the occurrence of data. Table 3.1 shows the symbols that are available for use in DTDs.

#### **Table : Element Symbols**

|  |  |
| --- | --- |
| **Symbol** | **Definition** |
| **Asterisk (\*)** | The data will appear zero or more times (0, 1, 2, ...). Here's an example: <!ELEMENT children (name\*)> In this example, the element children could have zero or more occurrences of the child element name. This type of rule would be useful on a form asking a person about his or her children. It is possible that the person could have no children or many children. |
| **Comma (,)** | Provides separation of elements in a sequence. Here's an example: <!ELEMENT address (street, city, state, zip)> -In this example, the element address will have four child elements: street, city, state, and zip. Each of the child elements must appear in the defined order in the XML document. |
| **Parentheses [( )]** | The parentheses are used to contain the rule for an element. Parentheses may also be used to group a sequence, subsequence, or a set of alternatives in a rule. Here's an example: <!ELEMENT address (street, city, (state | province), zip)> In this example, the parentheses enclose a sequence. Additionally, a subsequence is nested within the sequence by a second set of parentheses. The subsequence indicates that there will be either a state or a province element in that spot in the main sequence. |
| **Pipe (|)** | Separates choices in a set of options. Here's an example: <!ELEMENT dessert (cake | pie)> The element dessert will have one child element: either cake or pie. |
| **Plus sign (+)** | Signifies that the data must appear one or more times (1, 2, 3, ...). Here's an example: <!ELEMENT appliances (refrigerator+)> The appliances element will have one or more refrigerator child elements. This assumes that every household has at least one refrigerator. |
| **Question mark (?)** | Data will appear either zero times or one time in the element. Here's an example: <!ELEMENT employment (company?)> The element employment will have either zero occurrences or one occurrence of the child element company. |
| **No symbol** | When no symbol is used (other than parentheses), this signifies that the data must appear once in the XML file. Here's an example: <!ELEMENT contact (name)> The element contact will have one child element: name. |

Element symbols can be added to element definitions for another level of control over the XML documents that are being validated against it.

#### **Limited Use of Symbols**

*<!ELEMENT contactlist (contact) >*

*<!ELEMENT contact (name, age, sex, address, city, state, zip, children) >*

*<!ELEMENT name (#PCDATA) >*

*<!ELEMENT age (#PCDATA) >*

*<!ELEMENT sex (#PCDATA) >*

*<!ELEMENT address (#PCDATA) >*

*<!ELEMENT city (#PCDATA) >*

*<!ELEMENT state (#PCDATA) >*

*<!ELEMENT zip (#PCDATA) >*

*<!ELEMENT children (child) >*

*<!ELEMENT child (childname, childage, childsex) >*

*<!ELEMENT childname (#PCDATA) >*

*<!ELEMENT childage (#PCDATA) >*

*<!ELEMENT childsex (#PCDATA) >*

A contact record for a contactlist file is being laid out. It is very straight forward and includes the basic address information you would expect to see in this type of file. Information on the contact's children is also included. This looks like a well-laid-out, easy-to-use file format. However, there are several problems. What if you are not sure about a contact's address? What if the contact does not have children? What if the user is a lady and you are afraid to ask her age? The way that this DTD is laid out, it will be very difficult for a referencing XML document to be deemed valid if any of this information is unknown.

Using element symbols, you can create a more flexible DTD that will take into account the possibility that you might not always know all of a contact's personal information.

**Take a look at a similar DTD laid out below.**

#### **Broader Use of Symbols**

*<!ELEMENT contactlist (contact+) >*

*<!ELEMENT contact (name, age?, sex, address?, city?, state?, zip?, children?) >*

*<!ELEMENT name (#PCDATA) >*

*<!ELEMENT age (#PCDATA) >*

*<!ELEMENT sex (#PCDATA) >*

*<!ELEMENT address (#PCDATA) >*

*<!ELEMENT city (#PCDATA) >*

*<!ELEMENT state (#PCDATA) >*

*<!ELEMENT zip (#PCDATA) >*

*<!ELEMENT children (child\*) >*

*<!ELEMENT child (childname, childage?, childsex) >*

*<!ELEMENT childname (#PCDATA) >*

*<!ELEMENT childage (#PCDATA) >*

*<!ELEMENT childsex (#PCDATA) >*

There is still a single root element, contactlist, which will contain one or more instances (+) of the element contact. Under each contact element is a list of child elements that make up the description of the contact record.

It is assumed here that the name and sex of the contact will be known. However, the definition indicates that there will be zero or one occurrence (?) of the age, address, city, state, zip, and children elements. These elements are set for zero or one occurrence because the definition is taking into account that this information might not be known.

Looking further down the listing, you see that the children element is marked to have zero or more instances (\*) of the child element. This is because a person might have no children or many children (or we might not know how many children the person has). Under the child element, it is assumed that childname and childsex information will be known (if there is at least one child element). However, the childage element is marked as zero or one (?), just in case it is unknown how old the child is.

**Element Content Types**

Content of elements declaration in a DTD can be categorized as below:

* Empty content
* Element content
* Mixed content
* Any content

1. **Empty Content**

This is a special case of element declaration.

This element declaration doesn’t contain any content. These are declared with the keyword **EMPTY**.

**Syntax**

Following is the syntax for empty element declaration:

***<!ELEMENT elementname EMPTY >***

**In the above syntax:**

**ELEMENT** is the element declaration of category *EMPTY*

**elementname** is the name of empty element.

**Example**   
Following is a simple example demonstrating empty element declaration:

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

***<!DOCTYPE hr[***

***<!ELEMENT address EMPTY>***

***]>***

***<address />***

In this example *address* is declared as an empty element. The markup for *address* element would appear as <address />.

1. **Element Content**

In element declaration with element content, the content would be allowable elements within parentheses. We can also include more than one element.

**Syntax**

Following is a syntax of element declaration with element content:

***<!ELEMENT elementname (child1, child2...)>***

* **ELEMENT** is the element declaration tag
* **elementname** is the name of the element.
* *child1, child2..* are the elements and each element must have its own definition within the DTD.

**Example**

Below example demonstrates a simple example for element declaration with element content:

***<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>***

***<!DOCTYPE address [***

***<!ELEMENT address (name,company,phone)>***

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

***<!ELEMENT company (#PCDATA)>***

***<!ELEMENT phone (#PCDATA)>***

***]>***

***<address>***

***<name>Srinivas B</name>***

***<company>TutorialsPoint</company>***

***<phone>(011) 123-4567</phone>***

***</address>***

In the above example, *address* is the parent element and *name*, *company* and *phone\_no* are its child elements.

**List of Operators and Syntax Rules**

Below table shows the list of operators and syntax rules which can be applied in defining child elements:

|  |  |  |  |
| --- | --- | --- | --- |
| ***Operator*** | ***Syntax*** | ***Description*** | ***Example*** |
| ***+*** | ***<!ELEMENT element-name (child1+)>*** | *It indicates that child element can occur* ***one or more times*** *inside parent element.* | *<!ELEMENT address (name+)>*  *Child element name can occur one or more times inside the element name address.* ***[1 or more]*** |
| ***\**** | ***<!ELEMENT element-name (child1\*)>*** | *It indicates that child element can occur* ***zero or more times*** *inside parent element.* | *<!ELEMENT address (name\*)>*  *Child element name can occur zero or more times inside the element name address.* ***[0 or more]*** |
| ***?*** | *<!ELEMENT element-name (child1?)>* | *It indicates that child element can occur* ***zero or one time*** *inside parent element.* | *<!ELEMENT address (name?)>*  *Child element name can occur zero or one time inside the element name address.* ***[0 or 1]*** |
| ***,*** | *<!ELEMENT element-name (child1, child2)>* | *It gives sequence of child elements separated by comma which must be included in the element-name.* | *<!ELEMENT address (name, company)>*  *Sequence of child elements name, company, which must occur in the same order inside the element name address.* |
| ***|*** | *<!ELEMENT element-name (child1 | child2)>* | *It allows making choices in the child element.* | *<!ELEMENT address (name | company)>*  It allows you to choose either of child elements i.e. name or company, which must occur in inside the element name address. |

**Rules**

We need to follow certain rules if there is more than one element content:

* **Sequences –**

Often the elements within DTD documents must appear in a distinct order. If this is the case, you define the content using a sequence.

**For example:**

***<!ELEMENT address (name,company,phone)>***

The declaration indicates that the <address> element must have exactly three children - <name>, <company>, and <phone> - and that they must appear in this order.

* **Choices** -

Suppose you need to allow one element or another, but not both. In such cases you must use the pipe (|) character. The pipe functions as an exclusive OR.

**For example:**

***<!ELEMENT address (mobile | landline)>***

* **Mixed Element Content**

This is the combination of (#PCDATA) and children elements. PCDATA stands for parsed character data, that is, text that is not markup. Within mixed content models, text can appear by itself or it can be interspersed between elements. The rules for mixed content models are similar to the element content as discussed in the previous section.

**Syntax**

Following is a generic syntax for mixed element content:

***<!ELEMENT elementname (#PCDATA|child1|child2)\*>***

* **ELEMENT** is the element declaration tag.
* **elementname** is the name of the element.
* **PCDATA** is the text that is not markup. #PCDATA must come first in the mixed content declaration.
* *child1, child2..* are the elements and each element must have its own definition within the DTD.
* The operator (\*) must follow the mixed content declaration if children elements are included
* The (#PCDATA) and children element declarations must be separated by the (|) operator.

**Example**

Following is a simple example demonstrating the mixed content element declaration in a DTD.

***<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>***

***<!DOCTYPE address [***

***<!ELEMENT address (#PCDATA|name)\*>***

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

***]>***

***<address>***

***Here's a bit of text mixed up with the child element.***

***<name>Tanmay Patil</name>***

***</address>***

**Example:**

*<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>*

*<!DOCTYPE address [*

*<!ELEMENT address (#PCDATA|name)\*>*

*<!ELEMENT name (#PCDATA)>*

*]>*

*<address>*

*Here's a bit of text mixed up with the child element.*

*<name>Srinivas</name>*

*<name>Srini</name>*

*Oh is it*

*<name></name>*

*Ok I got it*

*</address>*

**ANY Element Content**

You can declare an element using the ANY keyword in the content. It is most often referred to as mixed category element. ANY is useful when you have yet to decide the allowable contents of the element.

**Syntax**

Following is the syntax for declaring elements with ANY content:

***<!ELEMENT elementname ANY>***

Here, the ANY keyword indicates that text (PCDATA) and/or any elements declared within the DTD can be used within the content of the <elementname> element. They can be used in any order any number of times. However, the ANY keyword does not allow you to include elements that are not declared within the DTD.

**Example**

Following is a simple example demonstrating the element declaration with ANY content:

***<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>***

***<!DOCTYPE address [***

***<!ELEMENT address ANY>***

***]>***

***<address>***

***Here's a bit of sample text***

***We can write any number of lines here. But we cannot create a new tag.***

***</address>***

**Summary:**

In a DTD, elements are declared with an ELEMENT declaration.

## **Declaring Elements**

In a DTD, XML elements are declared with the following syntax:

***<!ELEMENT element-name category>  
or  
<!ELEMENT element-name (element-content)>***

## **Empty Elements**

Empty elements are declared with the category keyword EMPTY:

***<!ELEMENT element-name EMPTY>***

***Example:  
 <!ELEMENT br EMPTY>***  
  
**XML example**:  
<br />

## **Elements with Parsed Character Data**

Elements with only parsed character data are declared with #PCDATA inside parentheses:

***<!ELEMENT element-name (#PCDATA)>***

***Example:  
<!ELEMENT from (#PCDATA)>***

## **Elements with any Contents**

Elements declared with the category keyword ANY, can contain any combination of parsable data:

***<!ELEMENT element-name ANY>  
Example:  
<!ELEMENT note ANY>***

## **Elements with Children (sequences)**

Elements with one or more children are declared with the name of the children elements inside parentheses:

***<!ELEMENT element-name (child1)>  
or  
<!ELEMENT element-name (child1,child2,...)>*  
Example:  
*<!ELEMENT note (to,from,heading,body)>***

When children are declared in a sequence separated by commas, the children must appear in the same sequence in the document. In a full declaration, the children must also be declared, and the children can also have children. The full declaration of the "note" element is:

***<!ELEMENT note (to,from,heading,body)>  
<!ELEMENT to (#PCDATA)>  
<!ELEMENT from (#PCDATA)>  
<!ELEMENT heading (#PCDATA)>  
<!ELEMENT body (#PCDATA)>***

## **Declaring Only One Occurrence of an Element**

***<!ELEMENT element-name (child-name)>***

***Example:  
 <!ELEMENT note (message)>***

The example above declares that the child element "message" must occur once, and only once inside the "note" element.

## **Declaring Minimum One Occurrence of an Element**

***<!ELEMENT element-name (child-name+)>***

***Example:  
 <!ELEMENT note (message+)>***

The + sign in the example above declares that the child element "message" must occur one or more times inside the "note" element.

## **Declaring Zero or More Occurrences of an Element**

***<!ELEMENT element-name (child-name\*)>***

***Example:  
 <!ELEMENT note (message\*)>***

The \* sign in the example above declares that the child element "message" can occur zero or more times inside the "note" element.

## **Declaring Zero or One Occurrences of an Element**

***<!ELEMENT element-name (child-name?)>***

***Example:  
 <!ELEMENT note (message?)>***

The ? Sign in the example above declares that the child element "message" can occur zero or one time inside the "note" element.

## **Declaring either/or Content**

***<!ELEMENT note (to,from,header,(message|body))>***

The example above declares that the "note" element must contain a "to" element, a "from" element, a "header" element, and either a "message" or a "body" element.

## **Declaring Mixed Content**

***<!ELEMENT note (#PCDATA|to|from|header|message)\*>***

The example above declares that the "note" element can contain zero or more occurrences of parsed character data, "to", "from", "header", or "message" elements.

**DTD – Attributes**

Attribute gives more information about an element or more precisely it defines a property of an element. An XML attribute is always in the form of a name-value pair. An element can have any number of unique attributes.

Attribute declaration is very much similar to element declarations in many ways except one; instead of declaring allowable content for elements, you declare a list of allowable attributes for each element. These lists are called ATTLIST declaration.

**Syntax**

Basic syntax of DTD attributes declaration is as follows:

***<!ATTLIST element-name attribute-name attribute-type attribute-value>***

**In the above syntax**

* The DTD attributes start with <!ATTLIST keyword if the element contains the attribute.
* **element-name** specifies the name of the element to which the attribute applies.
* **attribute-name** specifies the name of the attribute which is included with the element-name.
* **attribute-type** defines the type of attributes. We will discuss more on this in the following sections.
* **attribute-value** takes a fixed value that the attributes must define. We will discuss more on this in the following sections.

#### **ATTLIST Declaration**

*<!ATTLIST name*

*sex CDATA #REQUIRED*

*age CDATA #IMPLIED*

*race CDATA #IMPLIED >*

Here an attribute list is declared.

The name element is being referenced by the declaration. Three attributes are defined; sex, age, and race.

The three attributes are character data (CDATA). Only one of the attributes, sex, is required (#REQUIRED). The other two attributes, age and race, are optional (#IMPLIED).

**An XML element using the attribute list declared here would appear as follows:**

<name sex="male" age="30" race="Caucasian">Michael Qualls</name>

The name element contains the value "Michael Qualls". It also has three attributes of Michael Qualls: sex, age, and race. The attributes are all character data (CDATA). However, attributes actually have 10 possible data types.

**Example**

Below is a simple example for attribute declaration in DTD:

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

***<!DOCTYPE address [***

***<!ELEMENT address ( name )>***

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

***<!ATTLIST name id CDATA #REQUIRED>***

***]>***

***<address>***

***<name id="123">Srinivas B</name>***

***</address>***

**Let us go through the above code:**

* Begin with the XML declaration with the following statement:

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

* Immediately following the XML header is the document type declaration, commonly referred to as the DOCTYPE:

***<!DOCTYPE address [***

The DOCTYPE informs the parser that a DTD is associated with this XML document. The DOCTYPE declaration has an exclamation mark (!) at the start of the element name.

**Following is the body of DTD. Here we have declared element and attribute:**

***<!ELEMENT address ( name )>***

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

* Attribute *id* for the element *name* is defined as:

***<!ATTLIST name id CDATA #REQUIRED>***

Here attribute type is *CDATA* and its value is *#REQUIRED*.

**Rules of Attribute Declaration**

* All attributes used in an XML document must be declared in the Document Type Definition (DTD) using an Attribute-List Declaration
* Attributes may only appear in start or empty tags.
* The keyword ATTLIST must be in upper case
* No duplicate attribute names will be allowed within the attribute list for a given element.

**Attribute Types**

When declaring attributes, you can specify how the processor should handle the data that appears in the value. We can categorize attribute types in three main categories:

* *String type*
* *Tokenized types*
* *Enumerated types*

**Following table provides a summary of the different attribute types:**

Before going over a more detailed example of using attributes in your DTDs, let's first review the 10 valid types of attributes that may be used in a DTD.

| **Type** | **Definition** |
| --- | --- |
| **CDATA** | Characterdata only. The attribute will contain no markup. Here's an example: **<ATTLIST box height CDATA "0">**. In this example, an attribute, height, has been defined for the element box. This attribute will contain character data and have a default value of "0". |
| **ENTITY** | The name of an unparsed general entity that is declared in the DTD but refers to some external data (such as an image file). Here's an example: <!ATTLIST img src ENTITY #REQUIRED> The src attribute is an ENTITY type that refers to some external image file. |
| **ENTITIES** | This is the same as the ENTITY type but represents multiple values listed in sequential order, separated by whitespace. Here's an example: <!ATTLIST imgs srcs ENTITIES #REQUIRED> The value of the imgs element using the srcs attribute would be something like img1.gif img2.gif img3.gif. This is simply a list of image files separated by whitespace. |
| **ID** | An attribute that uniquely identifies the element. The value for this type of attribute must be unique within the XML document. Each element may only have a single ID attribute, and the value of the ID attribute must be a valid XML name, meaning that it may not start with a numeric digit (which precludes the use of a simple numbering system for IDs). Here's an example: <!ATTLIST cog serial ID #REQUIRED> Each cog element in the XML document will have a required attribute, serial, that uniquely identifies it. |
| **IDREF** | This is the value of an ID attribute of another element in the document. It's used to establish a relationship with other tags when there is not necessarily a parent/child relationship. Here's an example: <!ATTLIST person cousin IDREF #IMPLIED> Each person element could have a cousin attribute that references the value of the ID attribute of another element. |
| **IDREFS** | This is the same as IDREF; however, it represents multiple values listed in sequential order, separated by whitespace. Here's an example: <!ATTLIST person cousins IDREFS #IMPLIED> Each person element could have a cousins attribute that contains references to the values of multiple ID attributes of other elements. |
| **NMTOKEN** | Restricts the value of the attribute to a valid XML name. Here's an example: <!ATTLIST address country NMTOKEN "usa"> Each address element will have a country attribute with a default value of "usa". |
| **NMTOKENS** | This is the same as NMTOKENS; however, it represents multiple values listed in sequential order, separated by whitespace. Here's an example: <!ATTLIST region states NMTOKENS "KS OK" > Each region element will have a states attribute with a default value of "KS OK". |
| **NOTATION** | This type refers to the name of a notation declared in the DTD (more on notations later). It is used to identify the format of non-XML data. An example would be using the NOTATION type to refer to an external application that will interact with the document. Here's an example: <!ATTLIST music play NOTATION "mplayer2.exe "> In this example, the element music has an attribute, play, that will hold the name of a notation that determines the type of music player to use. The default value (notation) is "mplayer2.exe ". |
| **Enumerated** | This type is not an actual keyword the way the other types are. It is actually a listing of possible values for the attribute separated by pipe symbols (|). Here's an example: <!ATTLIST college grad (1|0) "1"> The element college has an attribute, grad, that will have a value of either "1" or "0" (with the default value being "1"). |

You saw during the coverage of the 10 valid attribute types that we used two preset default behavior settings: #REQUIRED and #IMPLIED. There are four different default types that may be used in an attribute definition, as detailed in Table 3.3.

**Attribute Value Declaration**

Within each attribute declaration, you must specify how the value will appear in the document. You can specify if an attribute:

* Can have a default value
* Can have a fixed value
* is required
* is implied

**Default Values:**

It contains the default value. The values can be enclosed in single quotes(') or double quotes(").

**Syntax**   
**Following is the syntax of value:**

***<!ATTLIST element-name attribute-name attribute-type "default-value">***

Where *default-value* is the attribute value defined.

**Example**

**Following is a simple example of attribute declaration with default value:**

**Example1:**

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

*<!DOCTYPE address [*

*<!ELEMENT address ( name )>*

*<!ELEMENT name ( #PCDATA )>*

*<!ATTLIST name id CDATA "0">*

*]>*

*<address>*

*<name id="123">Tanmay Patil</name>*

*</address>*

***Example 2:***

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

*<!DOCTYPE address [*

*<!ELEMENT address ( name )>*

*<!ELEMENT name ( #PCDATA )>*

*<!ATTLIST name id CDATA "0">*

*]>*

*<address>*

*<name>Tanmay Patil</name>*

*</address>*

***Example 3:***

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

*<!DOCTYPE address [*

*<!ELEMENT address ( name )>*

*<!ELEMENT name ( #PCDATA )>*

*<!ATTLIST name id CDATA "0">*

*]>*

*<address>*

*<name id="ABC">Tanmay Patil</name>*

*</address>*

In this example we have *name* element with attribute *id* whose default value is *0*. The default value is been enclosed within the double quotes.

**There are four different default types that may be used in an attribute definition:**

#### **Default Value Types**

| **Type** | **Definition** |
| --- | --- |
| **#REQUIRED** | Indicates that the value of the attribute must be specified. Here's an example <!ATTLIST season year CDATA #REQUIRED > In this example, the element season has a character data attribute, year, that is required. |
| **#IMPLIED** | Indicates that the value of the attribute is optional. Here's an example: <!ATTLIST field size CDATA #IMPLIED > In this case, each field element may have a size attribute, but it is not required. |
| **#FIXED** | Indicates that the attribute is optional, but if it is present, it must have a specified set value that cannot be changed. Here's an example: <!ATTLIST bcc hidden #FIXED "true" > Each bcc element has an attribute, hidden, that has a fixed value of "true". |
| **Default** | This is not an actual default behavior type. The value of the default is supplied in the DTD. Here's an example: <!ATTLIST children number CDATA "0"> This represents that the children element has a number attribute with a default value of "0". |

**FIXED Values**

#FIXED keyword followed by the fixed value is used when you want to specify that the attribute value is constant and cannot be changed. A common use of fixed attributes is specifying version numbers.

**Syntax**

**Following is the syntax of fixed values:**

*<!ATTLIST element-name attribute-name attribute-type #FIXED "value" >*

Where #FIXED is an attribute value defined.

**Example**

Following is a simple example of attribute declaration with FIXED value:

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

***<!DOCTYPE address [***

***<!ELEMENT address (company)\*>***

***<!ELEMENT company (#PCDATA)>***

***<!ATTLIST company name NMTOKEN #FIXED "tutorialspoint">***

***]>***

***<address>***

***<company name="tutorialspoint">we are a free online teaching faculty</company>***

***</address>***

In this example, we have used the keyword #FIXED where it indicates that the value "tutorialspoint" is the only value for the attribute *name* of element <company>. If we try to change the attribute value then it gives an error.

**Following is an invalid DTD:**

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

***<!DOCTYPE address [***

***<!ELEMENT address (company)\*>***

***<!ELEMENT company (#PCDATA)>***

***<!ATTLIST company name NMTOKEN #FIXED "tutorialspoint">***

***]>***

***<address>***

***<company name="abc">we are a free online teaching faculty</company>***

***</address>***

**REQUIRED values**

Whenever you want specify that an attribute is required, use #REQUIRED keyword.

**Syntax**

Following is the syntax of #REQUIRED:

*<!ATTLIST element-name attribute-name attribute-type #REQUIRED>*

Where #REQUIRED is an attribute type defined.

**Example**

**Following is a simple example of DTD attribute declaration with #REQUIRED keyword:**

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

***<!DOCTYPE address [***

***<!ELEMENT address ( name )>***

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

***<!ATTLIST name id CDATA #REQUIRED>***

***]>***

***<address>***

***<name id="123">Tanmay Patil</name>***

***</address>***

In this example, we have used #REQUIRED keyword to specify that the attribute *id* must be provided for the element-name *name*

**IMPLIED Values**

When declaring attributes you must always specify a value declaration. If the attribute you are declaring has no default value, has no fixed value, and is not required, then you must declare that the attribute as *implied*. Keyword #IMPLIED is used to specify an attribute as *implied*.

**Syntax**

**Following is the syntax of #IMPLIED:**

***<!ATTLIST element-name attribute-name attribute-type #IMPLIED>***

Where #IMPLIED is an attribute type defined.

**Example**

Following is a simple example of #IMPLIED

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

***<!DOCTYPE address [***

***<!ELEMENT address ( name )>***

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

***<!ATTLIST name id CDATA #IMPLIED>***

***]>***

***<address>***

***<name />***

***</address>***

**Example**:

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

**<!DOCTYPE address [**

**<!ELEMENT address ( name )>**

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

**<!ATTLIST name id CDATA #IMPLIED>**

**]>**

**<address>**

**<name id="1">Srini</name>**

**</address>**

**Example**:

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

**<!DOCTYPE address [**

**<!ELEMENT address ( name )>**

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

**<!ATTLIST name id CDATA #IMPLIED>**

**]>**

**<address>**

**<name>Srini</name>**

**</address>**

In this example we have used the keyword #IMPLIED as we do not want to specify any attributes to be included in element *name*. It is optional.

**Summary**:

In a DTD, attributes are declared with an ATTLIST declaration.

**Declaring Attributes**

An attribute declaration has the following syntax:

***<!ATTLIST element-name attribute-name attribute-type attribute-value>***

***DTD example:  
<!ATTLIST payment type CDATA "check">***  
**XML example:**  
*<payment type="check" />*

The **attribute-type** can be one of the following:

|  |  |
| --- | --- |
| Type | Description |
| CDATA | The value is character data |
| (*en1*|*en2*|..) | The value must be one from an enumerated list |
| ID | The value is a unique id |
| IDREF | The value is the id of another element |
| IDREFS | The value is a list of other ids |
| NMTOKEN | The value is a valid XML name |
| NMTOKENS | The value is a list of valid XML names |
| ENTITY | The value is an entity |
| ENTITIES | The value is a list of entities |
| NOTATION | The value is a name of a notation |
| xml: | The value is a predefined xml value |

**The attribute-value can be one of the following:**

|  |  |
| --- | --- |
| **Value** | **Explanation** |
| *value* | The default value of the attribute |
| #REQUIRED | The attribute is required |
| #IMPLIED | The attribute is optional |
| #FIXED *value* | The attribute value is fixed |

## **A Default Attribute Value**

**DTD:  
*<!ELEMENT square EMPTY>  
<!ATTLIST square width CDATA "0">*  
Valid XML:  
<square width="100" />**

In the example above, the "square" element is defined to be an empty element with a "width" attribute of  type CDATA. If no width is specified, it has a default value of 0.

## **#REQUIRED**

### **Syntax**

*<!ATTLIST element-name attribute-name attribute-type #REQUIRED>*

### **Example DTD:** <!ATTLIST person number CDATA #REQUIRED> **Valid XML:** <person number="5677" /> **Invalid XML:** <person />

Use the #REQUIRED keyword if you don't have an option for a default value, but still want to force the attribute to be present.

## **#IMPLIED**

### **Syntax**

<!ATTLIST element-name attribute-name attribute-type #IMPLIED>

### **Example DTD:** <!ATTLIST contact fax CDATA #IMPLIED> **Valid XML:** <contact fax="555-667788" /> **Valid XML**: <contact />

Use the #IMPLIED keyword, if you don't want to force the author to include an attribute, and you don't have an option for a default value.

## **#FIXED**

### **Syntax**

<!ATTLIST element-name attribute-name attribute-type #FIXED "value">

### **Example DTD:** <!ATTLIST sender company CDATA #FIXED "Microsoft"> **Valid XML**: <sender company="Microsoft" /> **Invalid XML**: <sender company="W3Schools" />

Use the #FIXED keyword when you want an attribute to have a fixed value without allowing the author to change it. If an author includes another value, the XML parser will return an error.

## **Enumerated Attribute Values**

### **Syntax**

<!ATTLIST element-name attribute-name (en1|en2|..) default-value>

### **Example DTD:** <!ATTLIST payment type (check|cash) "cash"> **XML example**: <payment type="check" /> **or** <payment type="cash" />

Use enumerated attribute values when you want the attribute value to be one of a fixed set of legal values.

**XML Elements vs. Attributes**

In XML, there are no rules about when to use attributes, and when to use child elements.

## **Use of Elements vs. Attributes**

Data can be stored in child elements or in attributes.

**Take a look at these examples:**

*<person sex="female">  
  <firstname>Anna</firstname>  
  <lastname>Smith</lastname>  
</person>   
<person>  
  <sex>female</sex>  
  <firstname>Anna</firstname>  
  <lastname>Smith</lastname>  
</person>*

In the first example sex is an attribute. In the last, sex is a child element. Both examples provide the same information.

There are no rules about when to use attributes, and when to use child elements. My experience is that attributes are handy in HTML, but in XML you should try to avoid them. Use child elements if the information feels like data.

## **My Favorite Way**

**I like to store data in child elements.**

The following three XML documents contain exactly the same information:

**A date attribute is used in the first example**:

*<note date="12/11/2002">  
  <to>Tove</to>  
  <from>Jani</from>  
  <heading>Reminder</heading>  
  <body>Don't forget me this weekend!</body>  
</note>*

**A date element is used in the second example:**

*<note>  
  <date>12/11/2002</date>  
  <to>Tove</to>  
  <from>Jani</from>  
  <heading>Reminder</heading>  
  <body>Don't forget me this weekend!</body>  
</note>*

An expanded date element is used in the third: (THIS IS MY FAVORITE):

*<note>  
  <date>  
    <day>12</day>  
    <month>11</month>  
    <year>2002</year>  
  </date>  
  <to>Tove</to>  
  <from>Jani</from>  
  <heading>Reminder</heading>  
  <body>Don't forget me this weekend!</body>  
</note>*

## **Avoid using attributes?**

**Some of the problems with attributes are**:

* attributes cannot contain multiple values (child elements can)
* attributes are not easily expandable (for future changes)
* attributes cannot describe structures (child elements can)
* attributes are more difficult to manipulate by program code
* attribute values are not easy to test against a DTD

If you use attributes as containers for data, you end up with documents that are difficult to read and maintain. Try to use **elements** to describe data. Use attributes only to provide information that is not relevant to the data.

**Don't end up like this (this is not how XML should be used):**

*<note day="12" month="11" year="2002"  
 to="Tove" from="Jani" heading="Reminder"  
 body="Don't forget me this weekend!">  
</note>*

## **An Exception to my Attribute Rule**

Rules always have exceptions.

**My rule about attributes has one exception:**

Sometimes I assign ID references to elements. These ID references can be used to access XML elements in much the same way as the NAME or ID attributes in HTML. This example demonstrates this:

*<messages>  
 <note id="p501">  
  <to>Tove</to>  
  <from>Jani</from>  
  <heading>Reminder</heading>  
  <body>Don't forget me this weekend!</body>  
 </note>  
 <note id="p502">  
  <to>Jani</to>  
  <from>Tove</from>  
  <heading>Re: Reminder</heading>  
  <body>I will not!</body>  
 </note>  
</messages>*

The ID in these examples is just a counter, or a unique identifier, to identify the different notes in the XML file, and not a part of the note data.

What I am trying to say here is that metadata (data about data) should be stored as attributes, and that data itself should be stored as elements.

**DTD – Entities**

Entities in DTDs are storage units. They can also be considered placeholders. Entities are special markups that contain content for insertion into the XML document. Usually this will be some type of information that is bulky or repetitive.

Entities make this type of information more easily handled because the DTD author can use them to indicate where the information should be inserted in the XML document. This is much better than having to retype the same information over and over.

An entity's content could be well-formed XML, normal text, binary data, a database record, and so on. **The main purpose of an entity is to hold content, and there is virtually no limit on the type of content an entity can hold.**

**The general syntax of an entity is as follows:**

*<!ENTITY entityname [SYSTEM | PUBLIC] entitycontent>*

* ENTITY is the tag name that specifies that this definition will be for an entity.
* entityname is the name by which the entity will be referred in the XML document.
* entitycontent is the actual contents of the entity—the data for which the entity is serving as a placeholder.
* SYSTEM and PUBLIC are optional keywords. Either one can be added to the definition of an entity to indicate that the entity refers to external content.

**NOTE**

The keyword SYSTEM or PUBLIC used in an entity declaration will always be indicative of the contents of the entity being contained in an external file. Think of this as something like a pointer in C and C++. The entity is used as a reference to an external source of data.

**NOTE**

Entity declarations do not follow the same "top-down" rule that element definitions do. They may be listed anywhere in the body of the DTD. However, it is good practice to list them first in the DTD as they may be referenced later in the document.

Entities may either point to internal data or external data. Internal entities represent data that is contained completely within the DTD. External entities point to content in another location via a URL. External data could be anything from normal parsed text in another file, to a graphics or audio file, to an Excel spreadsheet. The type of data to which an external entity can refer is virtually unlimited.

An entity is referenced in an XML document by inserting the name of the entity prefixed by & and suffixed by ;. When referenced in this manner, the content of the entity will be placed into the XML document when the document is parsed and validated.

**Let's take a look at an example of how this works.**

**Using Internal Entities**

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

*<!DOCTYPE library [*

*<!ENTITY cpy "Copyright 2000">*

*<!ELEMENT library (book+)>*

*<!ELEMENT book (title,author,copyright)>*

*<!ELEMENT title (#PCDATA)>*

*<!ELEMENT author (#PCDATA)>*

*<!ELEMENT copyright (#PCDATA)>*

*]>*

*<library>*

*<book>*

*<title>How to Win Friends</title>*

*<author>Joe Charisma</author>*

*<copyright>&cpy;</copyright>*

*</book>*

*<book>*

*<title>Make Money Fast</title>*

*<author>Jimmy QuickBuck</author>*

*<copyright>&cpy;</copyright>*

*</book>*

*</library>*

In the DTD, an entity called cpy is declared that contains the content "Copyright 2000". In the copyright element of the XML document, this entity is referenced by using &cpy;. When this document is parsed, &cpy; will be replaced with "Copyright 2000" in each instance in which it is used. Using the entity &cpy; saves the XML document author from having to type in "Copyright 2000" over and over. These are a fairly simple example, but imagine if the entity contained a string of data that was several hundred characters long. It is much more convenient (and easier on the fingers) to be able to reference a three- or four-character entity in an XML document than to type in all that content.

#### **Predefined Entities**

There are five predefined entities, as shown in Table 3.4. These entities do not have to be declared in the DTD. When an XML parser encounters these entities (unless they are contained in a CDATA section), they will automatically be replaced with the content they represent.

#### **Table Predefined Entities**

| **Entity** | **Content** |
| --- | --- |
| &amp; | & |
| &lt; | < |
| &gt; | > |
| &quot; | " |
| &apos; | ' |

#### **Using Predefined Entities**

*<icecream>*

*<flavor>Cherry Garcia</flavor>*

*<vendor>Ben &amp; Jerry's</vendor>*

*</icecream>*

In this listing, the ampersand in "Ben & Jerry's" is replaced with the predefined entity for an ampersand (&amp;) .

#### **External Entities**

External entities are used to reference external content. As stated previously, external entities get their content by referencing it via a URL placed in the entitycontent portion of the entity declaration. Either the SYSTEM keyword or the PUBLIC keyword is used here to let the XML parser know that the content is external.

XML is incredibly flexible. External entities can contain references to almost any type of data—even other XML documents. One well-formed XML document can contain another well-formed XML document through the use of an external entity reference. Taking this a step further, it can be easily extrapolated that a single XML document can be made up of references to many small XML documents. When the document is parsed, the XML parser will gather all the small XML documents, merging them into a whole. The end-user application will only see one document and never know the difference. One useful way to apply the principle of combining XML documents through the use of external entities would be in an employee-tracking application, like the one shown below.

#### **Using External Entities**

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

*<!DOCTYPE employees [*

*<!ENTITY bob SYSTEM "http://srvr/emps/bob.xml">*

*<!ENTITY nancy SYSTEM "http://srvr/emps/nancy.xml">*

*<!ELEMENT employees (clerk)>*

*<!ELEMENT clerk (#PCDATA)>*

*]>*

*<employees>*

*<clerk>&bob;</clerk>*

*<clerk>&nancy;</clerk>*

*</employees>*

In this listing, two external entity references are used to refer to XML documents outside the current document that contain the employee data on "bob" (bob.xml) and "nancy" (nancy.xml). The SYSTEM keyword is used here to let the XML parser know that this is external content. In order to insert the external content into the XML document, the entities &bob; and &nancy; are used. It is useful to be able to contain the employee information in a separate file and "import" it using an entity reference. This is because this same information could be easily referenced by other XML documents, such as an employee directory and a payroll application. Defining logical units of data and separating them into multiple documents, as in this example, makes the data more extensible and reduces the need to reproduce redundant data from document to document.

**CAUTION**

Use prejudice when splitting up your XML data into multiple documents. Splitting up employee records into 100 different XML documents so that they will have increased extensibility across multiple applications might be a good idea. Taking the orders table from your order tracking database and splitting it into 100,000 documents would be a horrible idea. External entities are parsed at runtime. Could you imagine parsing thousands of entities that point to XML documents at runtime? Applications would suddenly be forced to search through 100,000 separate documents to find what they need instead of a single indexed table. Performance would be destroyed. So, keep in mind that although the approach mentioned here does have very applicable uses, it should not represent an overall data storage solution.

#### **Non-Text External Entities and Notations**

Some external entities will contain non-text data, such as an image file. We do not want the XML parser to attempt to parse these types of files. In order to stop the XML parser, we use the NDATA keyword.

**Take a look at the following declaration:**

<!ENTITY myimage SYSTEM "myimage.gif" NDATA gif>

The NDATA keyword is used to alert the parser that the entity content should be sent unparsed to the output document.

The final part of the declaration, gif, is a reference to a notation. A *notation* is a special declaration that identifies the format of non-text external data so that the XML application will know how handle the data. Any time an external reference to non-text data is used, a notation identifying the data must be included and referenced. Notations are declared in the body of the DTD and have the following syntax:

**<!NOTATION notationname [SYSTEM | PUBLIC ] dataformat>**

* ENTITY is the tag name that specifies that this definition will be for an entity.
* notationname is the name by which the notation will be referred in the XML document.
* SYSTEM is a keyword that is added to the definition of the notation to indicate that the format of external data is being defined. You could also use the keyword PUBLIC here instead of SYSTEM. However, using PUBLIC requires you to provide a URL to the data format definition.
* dataformat is a reference to a MIME type, ISO standard, or some other location that can provide a definition of the data being referenced.

**NOTE**

Notation declarations do not follow the same "top-down" rule that element definitions do. They may be listed anywhere in the body of the DTD. However, it is good practice to list them after the entity that references them in order to increase clarity.

#### **Using External Non-Text Entities**

*<!NOTATION gif SYSTEM "image/gif" >*

*<!ENTITY employeephoto SYSTEM "images/employees/MichaelQ.gif" NDATA gif >*

*<!ELEMENT employee (name, sex, title, years) >*

*<!ATTLIST employee pic ENTITY #IMPLIED >*

*...*

*<employee pic="employeephoto">*

*...*

*</employee>*

In this example, an ENTITY type of attribute, pic, is defined for the element employee. In the XML document, the pic attribute is given the value employeephoto, which is an external entity that serves as a placeholder for the GIF file MichaelQ.gif. In order to aid the application process and display the GIF file, the external entity (using the NDATA keyword) references the notation gif, which points to the MIME type for GIF files.

#### **Parameter Entities**

The final type of entity we will look at is the parameter entity, which is very similar to the internal entity. The main difference between an internal entity and a parameter entity is that a parameter entity may only be referenced inside the DTD. Parameter entities are in effect entities specifically for DTDs.

Parameter entities can be useful when you have to use a lot of repetitive or lengthy text in a DTD. Use the following syntax for parameter entities:

**<!ENTITY % entityname entitycontent>**

The syntax for a parameter entity is almost identical to the syntax for a normal, internal entity. However, notice that in the syntax, after the declaration, there is a space, a percent sign, and another space before entityname. This alerts the XML parser that this is a parameter entity and will be used only in the DTD. These types of entities, when referenced, should begin with % and end with ;.

**Following shows an example of this.**

#### **Using Parameter Entities**

*<!ENTITY % pc "(#PCDATA)">*

*<!ELEMENT name %pc;>*

*<!ELEMENT age %pc;>*

*<!ELEMENT weight %pc;>*

In this listing, pc is used as a parameter entity to reference (#PCDATA). All entities in the DTD that hold parsed character data use the entity reference %pc;. This saves the DTD author from having to type #PCDATA over and over. This particular example is somewhat trivial, but you can see where this can be extrapolated out to a situation where you have a long character string that you do not want to have to retype.

We are almost finished. Having covered the use of element, attribute, and entity declarations in DTDs, we have just a few more loose ends to tie up. In the next section, we will look at the use of the IGNORE and INCLUDE directives. Then we will discuss the use of comments in DTDs. In the final part of the chapter, we will look at the future of DTDs, some possible shortcomings of DTDs, and a possible alternative for DTD validation.

**Revisiting - Entities:**

Entities are used to define shortcuts to special characters within the XML documents. Entities can be primarily of four types:

* Built-in entities
* Character entities
* General entities
* Parameter entities

**Entity Declaration Syntax**

In general, entities can be declared ***internally*** or ***externally***. Let us understand each of these and their syntax as follows:

**Internal Entity**

If an entity is declared within a DTD it is called as internal entity.

**Syntax**

Following is the syntax for internal entity declaration:

***<!ENTITY entity\_name "entity\_value">***

**In the above syntax**:

* **entity\_name** is the name of entity followed by its value within the double quotes or single quote.
* **entity\_value** holds the value for the entity name.
* The entity value of the Internal Entity is de-referenced by adding prefix **&** to the entity name *i.e. &entity\_name.*

**EXAMPLE**

Following is a simple example for internal entity declaration:

***<?xml version="1.0" encoding="UTF-8" standalone="yes"?>***

***<!DOCTYPE address [***

***<!ELEMENT address (#PCDATA)>***

***<!ENTITY name "Tanmay patil">***

***<!ENTITY company "TutorialsPoint">***

***<!ENTITY phone\_no "(011) 123-4567">***

***]>***

***<address>***

***&name;***

***&company;***

***&phone\_no;***

***</address>***

**Example**:

***<?xml version="1.0" encoding="UTF-8" standalone="yes"?>***

***<!DOCTYPE address [***

***<!ELEMENT address (#PCDATA)>***

***<!ENTITY name "Tanmay patil">***

***<!ENTITY company "TutorialsPoint">***

***<!ENTITY phone\_no "(011) 123-4567">***

***]>***

***<address>***

***&abc;***

***</address>***

***Error****: Entity abc is undefined*

**Example**:

***<?xml version="1.0" encoding="UTF-8" standalone="yes"?>***

***<!DOCTYPE address [***

***<!ELEMENT address (#PCDATA)>***

***<!ENTITY name "Tanmay patil">***

***<!ENTITY company "TutorialsPoint">***

***<!ENTITY phone\_no "(011) 123-4567">***

***]>***

***<address>***

***&name;***

***</address>***

In the above example, the respective entity names ***name*, *company* and *phone\_no*** are replaced by their values in the XML document. The entity values are de-referenced by adding prefix **&** to the entity name.

Save this file as **sample.xml** and open it in any browser, you will notice that the entity values for *name*, *company*, *phone\_no* are replaced respectively.

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>

<!DOCTYPE address [

<!ELEMENT address (#PCDATA)>

<!ENTITY name "Tanmay patil">

<!ENTITY company "TutorialsPoint">

<!ENTITY phone\_no "(011) 123-4567">

]>

<address>

&name;

</address>

**IN BROWSER:**

<address> Tanmay patil </address>

**External Entity**

If an entity is declared outside a DTD it is called as external entity. You can refer to an external Entity by either using system identifiers or public identifiers.

**Syntax**

Following is the syntax for External Entity declaration:

**<!ENTITY name SYSTEM "URI/URL">**

In the above syntax:

* **name** is the name of entity.
* **SYSTEM** is the keyword.
* **URI/URL** is the address of the external source enclosed within the double or single quotes.

**Types**

You can refer to an external DTD by either using:

* **System Identifiers –**

A system identifier enables you to specify the location of an external file containing DTD declarations. Syntax is as follows:

*<!DOCTYPE name SYSTEM "address.dtd" [...]>*

As you can see it contains keyword SYSTEM and a URI reference pointing to the document's location.

* **Public Identifiers -**

Public identifiers provide a mechanism to locate DTD resources and are written as below:

*<!DOCTYPE name PUBLIC "-//Beginning XML//DTD Address Example//EN">*

As you can see, it begins with keyword PUBLIC, followed by a specialized identifier. Public identifiers are used to identify an entry in a catalog. Public identifiers can follow any format; however, a commonly used format is called *Formal Public Identifiers, or FPIs.*

**Example**

Let us understand the external entity with the following example:

***<?xml version="1.0" encoding="UTF-8" standalone="yes"?>***

***<!DOCTYPE address SYSTEM "address.dtd">***

***<address>***

***<name>Srinivas B</name>***

***<company>TutorialsPoint</company>***

***<phone>(011) 123-4567</phone>***

***</address>***

Below is the content of the DTD file *address.dtd*:

***<!ELEMENT address (name, company, phone)>***

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

***<!ELEMENT company (#PCDATA)>***

***<!ELEMENT phone (#PCDATA)>***

**Built-in entities**

All XML parsers must support built-in entities. In general, you can use these entity references anywhere. You can also use normal text within the XML document, such as in element contents and attribute values.

**There are five built-in entities that play their role in well-formed XML, they are:**

* **ampersand**: &amp;
* **Single quote**: &apos;
* **Greater than**: &gt;
* **Less than**: &lt;
* **Double quote**: &quot;

**Example**

Following example demonstrates the built-in entity declaration:

<?xml version="1.0"?>

<note>

<description>I'm a technical writer & programmer</description>

</note>

As you can see here the &amp; character is replaced by & whenever the processor encounters this.

**IN BROWSER:**

<note>

<description>I'm a technical writer &amp; programmer</description>

</note>

**IN XML SPY:**

<note>

<description>I'm a technical writer &amp; programmer</description>

</note>

**Character entities**

Character Entities are used to name some of the entities which are symbolic representation of information i.e characters that are difficult or impossible to type can be substituted by Character Entities.

**Example**

Following example demonstrates the character entity declaration:

*<?xml version="1.0" encoding="UTF-8" standalone="yes"?>*

*<!DOCTYPE author[*

*<!ENTITY writer SYSTEM "http://www.w3schools.com/entities.dtd">*

*<!ENTITY copyright SYSTEM "http://www.w3schools.com/entities.dtd">*

*]>*

*<author>&writer;&copyright;</author>*

You will notice here we have used **&#169;** as value for copyright character. Save this file as *sample.xml* and open it in your browser and you will see that copyright is replaced by the character ©.

**The above code appears in browser as:**

*<?xml version="1.0" encoding="UTF-8" standalone="true"?>*

*<!DOCTYPE author>*

*<author>Tanmay patil©</author>*

**General entities**

General entities must be declared within the DTD before they can be used within an XML document. Instead of representing only a single character, general entities can represent characters, paragraphs, and even entire documents.

**Syntax**

To declare a general entity, use a declaration of this general form in your DTD:

*<!ENTITY ename "text">*

**Example**

Following example demonstrates the general entity declaration:

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

*<!DOCTYPE note [*

*<!ENTITY source-text "tutorialspoint">*

*]>*

*<note>*

*&source-text;*

*</note>*

Whenever an XML parser encounters a reference to *source-text* entity, it will supply the replacement text to the application at the point of the reference.

**Parameter entities**

The purpose of a parameter entity is to enable you to create reusable sections of replacement text.

**Syntax**

Following is the syntax for parameter entity declaration:

*<!ENTITY % ename "entity\_value">*

* *entity\_value* is any character that is not an '&', '%' or ' " '.

**Example**

Following example demonstrates the parameter entity declaration. Suppose you have element declarations as below:

*<!ELEMENT residence (name, street, pincode, city, phone)>*

*<!ELEMENT apartment (name, street, pincode, city, phone)>*

*<!ELEMENT office (name, street, pincode, city, phone)>*

*<!ELEMENT shop (name, street, pincode, city, phone)>*

Now suppose you want to add additional element *country*, then you need to add it to all four declarations. Hence we can go for a parameter entity reference. Now using parameter entity reference the above example will be :

*<!ENTITY % area "name, street, pincode, city">*

*<!ENTITY % contact "phone">*

Parameter entities are dereferenced in the same way as a general entity reference, only with a percent sign instead of an ampersand:

*<!ELEMENT residence (%area;, %contact;)>*

*<!ELEMENT apartment (%area;, %contact;)>*

*<!ELEMENT office (%area;, %contact;)>*

*<!ELEMENT shop (%area;, %contact;)>*

When the parser reads these declarations, it substitutes the entity's replacement text for the entity reference.

**More DTD Directives**

The keywords are INCLUDE and IGNORE, and they do just what their names suggest—they indicate pieces of markup that should either be included in the validation process or ignored.

#### **The IGNORE Keyword**

When developing or updating a DTD, you may need to comment out parts of the DTD that are not yet reflected in the XML documents that use the DTD. You could use a normal comment directive (which will be covered in the next section), or you can use an IGNORE directive. The syntax for IGNORE is shown in Listing 3.19.

#### **Using IGNORE Directives**

*<![ IGNORE*

*This is the part of the DTD ignored*

*]]>*

You can choose to ignore elements, entities, or attributes. However, you must ignore entire declarations. You may not attempt to ignore a part of a declaration.

**For example, the following would be invalid:**

*<!ELEMENT Employee <![ IGNORE (#PCDATA) ]]> (Name, Address, Phone) >*

In this example, the DTD author has attempted to ignore the rule #PCDATA in the middle of an element declaration. This is invalid and would trigger an error.

#### **The INCLUDE Keyword**

The INCLUDE directive marks declarations to be included in the document. It might seem interesting that this keyword exists at all because not using an INCLUDE directive is the same as using it! In the absence of the INCLUDE directive, all declarations (unless they are commented out or enclosed in an IGNORE directive) will be included anyway.

The syntax for INCLUDE, is very similar to the syntax for the IGNORE directive.

#### **Using INCLUDE Directives**

*<![ INCLUDE*

*This is the part of the DTD included*

*]]>*

The INCLUDE directive follows the same basic rules as the IGNORE directive. It may enclose entire declarations but not pieces of declarations. The INCLUDE directive can be useful when you're in the process of developing a new DTD or adding to an existing DTD. Sections of the DTD can be toggled between the INCLUDE directive and the IGNORE directive in order to make it clear which sections are currently being used and which are not. This can make the process of developing a new DTD easier, because you are able to quickly "turn on" or "turn off" different sections of the DTD.

**NOTE**

If an INCLUDE directive is enclosed by an IGNORE directive, the INCLUDE directive and its declarations will be ignored.

#### **Comments Within a DTD**

Comments can also be added to DTDs. Comments within a DTD are just like comments in HTML and take the following syntax:

<!-- Everything between the opening tag and closing tag is a comment -->

As in HTML, comments in a DTD may not be nested. Comments may, however, span multiple lines. Generally comments in a DTD are used to demarcate different sections of the DTD or to help human readers understand different abbreviations used in the declarations. Comments will be ignored by the XML parser during processing.

**Following shows how to insert comments into a DTD.**

#### **Using Comments**

*<!-- This is a comment -->*

*<!ELEMENT rootelement (element1, element2)>*

*<!ELEMENT element1 (#PCDATA)>*

*<!-- This is another comment -->*

*<!ELEMENT element2 (#PCDATA)>*

*<!-- This is a comment*

*that spans multiple lines -->*

Comments provide a useful way to explain the meaning of different elements, attribute lists, and entities within the DTD. They can also be used to demarcate the beginning and end of different sections in the DTD.

The DTD is a powerful tool for defining rules for XML documents to follow. DTDs have had and will continue to have an important place in the XML world for some time to come. However, DTDs are not perfect. As XML has expanded beyond a simple document markup language, these limitations have become more apparent. XML is quickly becoming the language of choice for describing more abstract types of data. DTDs are hard-pressed to keep up. We will now take a look at some of the drawbacks to DTDs and what future alternatives will be available.

**DTD - Validation**

We use DTD to describe precisely the XML document. DTDs check the validity of structure and vocabulary of an XML document against the grammatical rules of the appropriate XML language. Now to check the validity of DTD, following procedures can be used:

* **Using XML DTD validation tools –**

You can use some IDEs such as XML Spy (not free) and XMLStarlet(opensource) can be used to validate XML files against DTD document.

* **Using XML DTD on-line validators –**

W3C Markup Validation Service is designed to validate Web documents. Use the online validator to check the validaty of your XML DTD here

* **Write your own XML validators with XML DTD validation API –**

Newer versions of JDK (above 1.4) support XML DTD validation API. You can write your own validator code to check the validity of XML DTD validation.

**DTD Drawbacks and Alternatives**

As newer uses for XML come into being, the needs for validation expand. XML is being used to describe the data structure of video files, audio files, and Braille devices, among other things—not to mention the ever-growing plethora of alternative data devices such as cellular phones, handheld computers, televisions, and even appliances. There are several drawbacks that limit the ability of DTDs to meet these growing and changing validation needs.

First and foremost, DTDs are composed of non-XML syntax. Given that one of the central tenets of XML is that it be totally extensible, it may not seem to make a lot of sense that this is the case for DTDs. However, you must consider that XML is a child of SGML, and in SGML, DTDs are the method used to validate documents. Therefore, XML inherited DTDs from its parent. Although DTDs are effective at defining the structure for document markup, as XML evolves, the fact that DTDS are not formed of XML syntax and are non-extensible becomes constraining.

Additionally, there can only be a single DTD per document. It is true that there can be internal and external subsets of DTDs, but there can only be a single DTD referenced. In the modern programming world, we are used to being able to draw the programming constructs we use from different modules or classes. If we applied this idea to DTDs, we might expect to be able to use a DTD for customers, a separate DTD for inventory, and a separate DTD for orders. However, this is not the case. All aspects of an XML document must be within a single DTD. This limitation is similar to what programmers faced back in the days of monolithic applications before object-oriented programming became a normal standard for application development. This leads into the next limitation.

DTDS are not object oriented. There is no inheritance in DTDs. As programmers, we have gotten used to describing new objects based on the characteristics of existing objects. One classic example is having Porsche, Ford, and Chevrolet classes that inherent some characteristics from a base car class. DTDs have no capability to do this.

DTDs do not support namespaces very well. For a namespace to be used, the entire namespace must be defined within the DTD. If there are more than one namespace, each of them must be defined within the DTD. This totally defeats the purpose of namespaces—being able to define multiple namespaces from many different external sources.

Additionally, DTDs have weak data typing and no support for the XML DOM. DTDs basically have one data type: the text string. There are a few restraints, such as the element rules and attribute types covered in this chapter, but these are pretty weak considering the types of applications for which XML is now being used (especially in e-commerce). The Document Object Model has become a powerful tool to manipulate XML data; however, the DTD is totally cut off from the reach of the DOM.

Finally, and possibly most important from a security standpoint, is the ability of the internal DTD subset to override the external DTD subset. An company could spend a great deal of time and effort crafting a DTD to validate the XML data in its e-commerce transactions only to have the settings in the DTD overridden by the internally defined elements of a DTD. The implications on this from a transaction security standpoint should be fairly clear.

So, what is to be done about the DTD? The DTD is still an effective mechanism for validating XML documents and will be so for a long time to come. It just does not "scale" to meet the needs of the expanding XML world. At the time of this writing, the W3C organization has just recently finished the final touches on the recommendation for the XML Schema, which is a new validation mechanism for XML that corrects all the shortcomings of DTDs. XML Schema is a powerful and important technology for the future of XML. The next chapter of this book will be devoted to covering the XML Schema.

## **TV Schedule DTD**

*<!DOCTYPE TVSCHEDULE [  
  
< !ELEMENT TVSCHEDULE (CHANNEL+)>  
< !ELEMENT CHANNEL (BANNER,DAY+)>  
< !ELEMENT BANNER (#PCDATA)>  
< !ELEMENT DAY (DATE,(HOLIDAY|PROGRAMSLOT+)+)>  
< !ELEMENT HOLIDAY (#PCDATA)>  
< !ELEMENT DATE (#PCDATA)>  
< !ELEMENT PROGRAMSLOT (TIME,TITLE,DESCRIPTION?)>  
< !ELEMENT TIME (#PCDATA)>  
< !ELEMENT TITLE (#PCDATA)>   
< !ELEMENT DESCRIPTION (#PCDATA)>  
  
< !ATTLIST TVSCHEDULE NAME CDATA #REQUIRED>  
< !ATTLIST CHANNEL CHAN CDATA #REQUIRED>  
< !ATTLIST PROGRAMSLOT VTR CDATA #IMPLIED>  
< !ATTLIST TITLE RATING CDATA #IMPLIED>  
< !ATTLIST TITLE LANGUAGE CDATA #IMPLIED>  
]>*

## **Newspaper Article DTD**

*<!DOCTYPE NEWSPAPER [  
  
< !ELEMENT NEWSPAPER (ARTICLE+)>  
< !ELEMENT ARTICLE (HEADLINE,BYLINE,LEAD,BODY,NOTES)>  
< !ELEMENT HEADLINE (#PCDATA)>  
< !ELEMENT BYLINE (#PCDATA)>  
< !ELEMENT LEAD (#PCDATA)>  
< !ELEMENT BODY (#PCDATA)>  
< !ELEMENT NOTES (#PCDATA)>  
  
< !ATTLIST ARTICLE AUTHOR CDATA #REQUIRED>  
< !ATTLIST ARTICLE EDITOR CDATA #IMPLIED>  
< !ATTLIST ARTICLE DATE CDATA #IMPLIED>  
< !ATTLIST ARTICLE EDITION CDATA #IMPLIED>  
  
< !ENTITY NEWSPAPER "Vervet Logic Times">  
< !ENTITY PUBLISHER "Vervet Logic Press">  
< !ENTITY COPYRIGHT "Copyright 1998 Vervet Logic Press">  
  
]>*

## **Product Catalog DTD**

*<!DOCTYPE CATALOG [  
  
< !ENTITY AUTHOR "John Doe">  
< !ENTITY COMPANY "JD Power Tools, Inc.">  
< !ENTITY EMAIL "jd@jd-tools.com">  
  
< !ELEMENT CATALOG (PRODUCT+)>  
  
< !ELEMENT PRODUCT  
(SPECIFICATIONS+,OPTIONS?,PRICE+,NOTES?)>  
< !ATTLIST PRODUCT  
NAME CDATA #IMPLIED  
CATEGORY (HandTool|Table|Shop-Professional) "HandTool"  
PARTNUM CDATA #IMPLIED  
PLANT (Pittsburgh|Milwaukee|Chicago) "Chicago"  
INVENTORY (InStock|Backordered|Discontinued) "InStock">  
  
< !ELEMENT SPECIFICATIONS (#PCDATA)>  
< !ATTLIST SPECIFICATIONS  
WEIGHT CDATA #IMPLIED  
POWER CDATA #IMPLIED>  
  
< !ELEMENT OPTIONS (#PCDATA)>  
< !ATTLIST OPTIONS  
FINISH (Metal|Polished|Matte) "Matte"  
ADAPTER (Included|Optional|NotApplicable) "Included"  
CASE (HardShell|Soft|NotApplicable) "HardShell">  
  
< !ELEMENT PRICE (#PCDATA)>  
< !ATTLIST PRICE  
MSRP CDATA #IMPLIED  
WHOLESALE CDATA #IMPLIED  
STREET CDATA #IMPLIED  
SHIPPING CDATA #IMPLIED>  
  
< !ELEMENT NOTES (#PCDATA)>  
  
]>*

**Question: 1**

**What is DTD?**

**Answer**:

DTD is abbreviated as Document Type Definition and it is defined to build legal building blocks of an XML document. It defines the XML document structure with elements and attributes.

**Question: 2**

**Give the main reasons to use DTD?**

**Answer:**

Some main reasons to use DTD are given Below:

* Using DTD XML files can write its description into its format.
* DTD is a most acceptable standard using DTD different group members can interchange data each other.
* Also, using DTD we can check the validity of our own data and the data that comes from different groups (other persons).

**Question: 3**

**What is Description of DTD**

**Answer:**

**< !DOCTYPE employee :**

It defines that the root element of the document is employee.  
**< !ELEMENT employee:**

It defines that the employee element contains 3 elements "firstname, lastname and email".

**< !ELEMENT firstname:**

It defines that the firstname element is #PCDATA typed. (parse-able data type).

**< !ELEMENT lastname:**

It defines that the lastname element is #PCDATA typed. (parse-able data type).

**< !ELEMENT email:**

It defines that the email element is #PCDATA typed. (parse-able data type).

**Question: 4**

**what are XML DTD with entity declaration**

**Answer:**

A doctype declaration can also define special strings that can be used in the XML file.  
An entity has three parts:

* An ampersand (&)
* An entity name
* A semicolon (;)

**Syntax to declare entity:**

* **< !ENTITY entity-name "entity-value">**Let's see a code to define the ENTITY in doctype declaration.  
  author.xml

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

< !DOCTYPE author [

< !ELEMENT author (#PCDATA)>

< !ENTITY sj "Sonoo Jaiswal">

]>

&sj;

**Question: 5**

**what is Purpose of DTD**

**Answer:**

Its main purpose is to define the structure of an XML document. It contains a list of legal elements and define the structure with the help of them.

**Question: 6**

**What is Simple Element?**

**Answer:**

A simple element contains only text and following are the kinds of Simple Element:

* No attributes
* Doesn’t contain other elements
* It cannot be empty

**Question: 7**

**What is Complex Element?**

**Answer:**

A complex element contains other elements or attributes and following are kinds of Complex Elements:

* It has empty elements
* It contain other elements
* It contain only text
* It contain both other elements and text

**Question: 8**

**What are DTD Attributes**

**Answer:**

**An attribute declaration has the following syntax:**< !ATTLIST element-name attribute-name attribute-type attribute-value>

**DTD example:**

< !ATTLIST payment type CDATA "check">

**XML example:**

|  |  |
| --- | --- |
| **Type** | **Description** |
| **CDATA** | The value is character data |
| **(en1|en2|..)** | The value must be one from an enumerated list |
| **ID** | The value is a unique id |
| **IDREF** | The value is the id of another element |
| **IDREFS** | The value is a list of other ids |
| **NMTOKEN** | The value is a valid XML name |
| **NMTOKENS** | The value is a list of valid XML names |
| **ENTITY** | The value is an entity |
| **ENTITIES** | The value is a list of entities |
| **NOTATION** | The value is a name of a notation |
| **xml:** | The value is a predefined xml value |

**Question: 9**

**What are the Enumerated Attribute Values**

**Answer:**

**Syntax**  
< !ATTLIST element-name attribute-name (en1|en2|..) default-value>  
**Example**  
**DTD:**< !ATTLIST payment type (check|cash) "cash">

**XML example:**  
< payment type="check" />  
or  
< payment type="cash" />

Use enumerated attribute values when you want the attribute value to be one of a fixed set of legal values.

**Question: 10**

**what are the Building Blocks of dtd?**

**Answer:**

The Building Blocks of XML Documents  
Seen from a DTD point of view, all XML documents are made up by the following building blocks:

* Elements
* Attributes
* Entities
* PCDATA
* CDATA

**How to perform validation in DTD ?**

We can perform validation using DTD in XML with internet explorer 5 and its higher version.

**To validate XML document using XML parser :**

We can perform validation using XML parser in XML document.

You can find out error code, error text, or line that cause error by accessing the **parseError** object.

We can use load() method to validate files and use loadXML() method to validate strings.

**Example**:

*var xmlDoc = new ActiveXObject("Microsoft.XMLDOM");*

*xmlDoc.async="false";*

*xmlDoc.validateOnParse="true";*

*xmlDoc.load("note\_dtd\_error.xml");*

*document.write("<br />Error Code: ");*

*document.write(xmlDoc.parseError.errorCode);*

*document.write("<br />Error Reason: ");*

*document.write(xmlDoc.parseError.reason);*

*document.write("<br />Error Line: ");*

*document.write(xmlDoc.parseError.line);*

**To turnoff validation :**

We can turnoff validation in XML document by using an code in XML Parser like: validateOnParse="false".

**Example**:

*var xmlDoc = new ActiveXObject("Microsoft.XMLDOM");*

*xmlDoc.async="false";*

*xmlDoc.validateOnParse="false";*

*xmlDoc.load("document\_dtd\_error.xml");*

*document.write("<br />Error Code: ");*

*document.write(xmlDoc.parseError.errorCode);*

*document.write("<br />Error Reason: ");*

*document.write(xmlDoc.parseError.reason);*

*document.write("<br />Error Line: ");*

*document.write(xmlDoc.parseError.line);*

**How you define entities in DTD?**

We can say that entities are nothing other than variable. We use entities to define short name of big standard text and also use to define special character. Entity consist of these three terms &(ampersand), entitiy name and ;(semicolon).

**To declare entity internally :**

**Syntax**:

<!ENTITY entity-name "entity-value">

**Example**:

**DTD Example:**

*<!ENTITY writer "Micky Mouse">*

*<!ENTITY copyright "Copyright WaltDisney">*

**XML example:**

*<author>&writer;&copyright;</author>*

**To declare entity externally :**

**Syntax:**

*<!ENTITY entity\_name SYSTEM "URI/URL">*

**Example**:

**A DTD Example**:

*<!ENTITY writer SYSTEM "http://www.r4r.co.in/entities.dtd">*

*<!ENTITY copyright SYSTEM "http://www.r4r.co.in/entities.dtd">*

**A XML example:**

*<author>&writer;&copyright;</author>*

*How we can store data in both child elements or attribute? Explain it.*

XML doesn't have specific rules about where we use child elements and where we use attributes. So, we can place them as per our requirement.

In support of that statement, I have given you example. Which definitely suggest you where we use child elements and attributes?

**Example**:

*<person sex="male">*

*<firstname>Sumit</firstname>*

*<middlename>kumar</middlename>*

*<lastname>Garg</lastname>*

*</person>*

**In this example we use sex as an attribute and attribute value is "male".**

*<person>*

*<sex>male</sex>*

*<firstname>Sumit</firstname>*

*<middlename>Kumar</middlename>*

*<lastname>Garg</lastname>*

*</person>*

**In this example sex is a child element and element value is "male".**

 But I advise you to use attributes in HTML and use child elements in case of XML.

**[How you define Attributes in DTD?](http://r4r.co.in/answer/index.php?id=4610&option=DTD%20SUB)**

We declare attribute in a DTD by using a keyword called **ATTLIST**.

Now. I show you how to define an attribute in a DTD.

**Syntax**:

**<!ATTLIST element-name attribute-name attribute-type default-value>**

 We declare attribute in DTD like that:

**<!ATTLIST status type CDATA "true">**

 We declare attribute in XML like that:

**<status type="true" />**

**Default Attribute Value :**

You can also set the default attribute value like that:

**DTD:**

<!ELEMENT rectangle EMPTY>

<!ATTLIST rectangle width CDATA "10">

**Valid XML:**

<rectangle width="50" />

This example set default value of rectangle is 10.

**# Required :**

We use # Required keyword if you don't want to set default value of an attribute.

**Syntax:**

**<!ATTLIST element\_name attribute\_name attribute\_type #REQUIRED>**

**Example**:

**DTD:**

**<!ATTLIST SSN number CDATA #REQUIRED>**

**Valid XML:**

<SSN number="0007" />

**Invalid XML:**

**<SSN />**

  Where, SSN stands for Social Security Number.

**#IMPLIED :**

Use this when don't want user to include attribute and also don't want to set default value.

**Syntax**

<!ATTLIST element-name attribute-name attribute-type #IMPLIED>

**Example:**

**DTD:**

<!ATTLIST contact fax CDATA #IMPLIED>

**Valid XML:**

<contact fax="091-123456" />

**Valid XML:**

<contact />

**#FIXED :**

If you want that attribute value not to be changed by the users in future. Than we can perform that task with the help of FIXED keyword.

**Syntax:**

<!ATTLIST element\_name attribute\_name attribute\_type #FIXED "value">

**Example:**

**DTD:**

<!ATTLIST indian company CDATA #FIXED "Microsoft">

**Valid XML:**

<indian company="R4R" />

**Invalid XML:**

<indian company="abc" />

**Enumerated Attribute Values :**

Use this when we want to use some fixed values.

**Syntax:**

<!ATTLIST element\_name attribute\_name (en1|en2|..) default\_value>

**Example:**

**DTD:**

<!ATTLIST payment type (check|DD|cash) "cash">

**XML example:**

<payment type="check" />

       or

<payment type="DD" />

       or

<payment type="cash" />

[**How you define building blocks in an XML document?**](http://r4r.co.in/answer/index.php?id=4596&option=DTD%20SUB)

 I have given you some building blocks that are used for making an XML and HTML document. The list are:

1. Elements

2. Attributes

3. Entities

4. CDATA

5. PCDATA

 Main building block of an XML and HTML document are elements.

1. **Elements :**

Elements may contain text, other elements or empty. Some empty HTML elements are "hr","img" and "br".

 Some HTML elements are "head", "body" and "title" etc.

  Some XML elements are "main","from" and "to" etc.

**Example**:

<title>write here some text</title>

<body>write here some text</body>

<message>write here some text</message>

<to>write here some text</to>

2. **Attributes :**

Attributes is used to give us more information about elements. We use attribute inside the open tag of element.

**Example**:

<img src="R4Rlogo.gif" />

 In this example name of element and attribute is "img" and "src" respectively.And the value of this src attribute is "R4Rlogo.gif".

3. **Entities :**

Entities are those special characters that we used in XML. We use this special character to perform some specific task.

Some entities that we used in XML are:

**Some entity references are**:

&lt;

&gt;

&apos;

&amp;

&quot;

**Some characters are:**

<

>

'

"

&

4. **PCDATA :**

Firstly I told you PCDATA is stands for parsed character data.

We can define PCDATA as PCDATA is an text that is parsed through a parser. And we can analyze PCDATA text through the parser used for entities and markup.

**5. CDATA :**

CDATA stands for character data. We can't parse the CDATA text through parser. Means that tags used inside CDATA text will not handle

CDATA text as markup and also we can't expend the entities.

[**Give the main reasons to use DTD?**](http://r4r.co.in/answer/index.php?id=4595&option=DTD%20SUB)

**Some main reasons to use DTD are given Below:**

1. Using DTD XML files can write its description into its format.

2. DTD is an most acceptable standard using DTD different group members can interchange data each other.

3. Also, using DTD we can check the validity of our own data and the data that comes from different groups(other persons).

[**How to declare DTD?**](http://r4r.co.in/answer/index.php?id=4591&option=DTD%20SUB)

DTD has two types of declarations internal and external. I have given you brief information of internal and external DTD.

1. **Internal DTD Declaration :**

I have given you a syntax using them you can declare internal DTD in our XML file.

**Syntax**:

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

**Example**:

Now I used internal DTD in XML file like that,

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

*<!DOCTYPE note [*

*<!ELEMENT note (to,from,heading,body)>*

*<!ELEMENT to      (#PCDATA)>*

*<!ELEMENT from    (#PCDATA)>*

*<!ELEMENT heading (#PCDATA)>*

*<!ELEMENT body    (#PCDATA)>*

*]>*

*<note>*

*<to>Abhi</to>*

*<from>Sudd</from>*

*<heading>Message</heading>*

*<body>Recently R4R given a VoiceBox facility using them you can chat with R4R</body>*

*</note>*

[**What is DTD?**](http://r4r.co.in/answer/index.php?id=4590&option=DTD%20SUB)

DTD is stands for Document Type Definition. We use DTD in XML document to define the legal building blocks.

 Using DTD we can also define the structure of a document with name of legal attributes and elements.

**Note:**

An internal DTD is good to use for rules that only apply to that specific document. If a document has both internal and external DTD subsets, the internal rules override the external rules in cases where the same item is defined in both subsets.

**Note:**

Imagine a scenario where data is being exchanged in an XML format between multiple organizations. The integrity of business-to-business data is vital for the smooth functioning of commerce. There needs to be a way to ensure that the structure of the XML data does not change from organization to organization (thus rendering the data corrupt and useless). A DTD can ensure this.

An extra advantage of using DTDs in this situation is that a single DTD could be referenced by all the organization's applications. The defined structure of the data would be in a centralized resource, which means that any changes to the data structure definition would only need to be implemented in one place. All the applications that referenced the DTD would automatically use the new, updated structure.

**NOTE**

The Document Type Declaration should not be confused with the Document Type Definition. These are two exclusive items. Also confusing is the acronym DTD, which is only ever used in reference to the Document Type Definition. The Document Type Declaration is the area of the XML document after the XML declaration that begins with <!DOCTYPE and ends with ]>. It actually encompasses the Document Type Definition. The Document Type Definition will be contained within an opening bracket ([) and a closing bracket (]).

**NOTE**

The keyword SYSTEM used in a Document Type Declaration will always be indicative of the Document Type Definition being contained in an external file.

**Note:**

**There may be one Document Type Declaration per XML document. The syntax is as follows:**

**<!DOCTYPE rootelement SYSTEM | PUBLIC DTDlocation [ internalDTDelements ] >**

* The exclamation mark (!) is used to signify the beginning of the declaration.
* DOCTYPE is the keyword used to denote this as a Document Type Definition.
* rootelement is the name of the root element or document element of the XML document.
* SYSTEM and PUBLIC are keywords used to designate that the DTD is contained in an external document. Although the use of these keywords is optional, to reference an external DTD you would have to use one or the other. The SYSTEM keyword is used in tandem with a URL to locate the DTD. The PUBLIC keyword specifies some public location that will usually be some application-specific resource reference.
* internalDTDelements are internal DTD declarations. These declarations will always be placed within opening ([) and closing (]) brackets.

**NOTE**

This book typically uses the more common SYSTEM keyword when referencing external DTDs.

It is possible for a Document Type Declaration to contain both an external DTD subset and an internal DTD subset. In this situation, the internal declarations take precedence over the external ones. In other words, if both the external and internal DTDs define a rule for the same element, the rule of the internal element will be the one used.

#### **contactlist.dtd**

*<!ELEMENT contactlist (fullname, address, phone, email) >*

*<!ELEMENT fullname (#PCDATA)>*

*<!ELEMENT address (addressline1, addressline2)>*

*<!ELEMENT addressline1 (#PCDATA)>*

*<!ELEMENT addressline2 (#PCDATA)>*

*<!ELEMENT phone (#PCDATA)>*

*<!ELEMENT email (#PCDATA)>*

#### **contactlist.xml**

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

*<!DOCTYPE contactlist SYSTEM "contactlist.dtd">*

*<contactlist>*

*<fullname>Bobby Soninlaw</fullname>*

*<address>*

*<addressline1>101 South Street</addressline1>*

*<addressline2>Apartment #2</addressline2>*

*</address>*

*<phone>(405) 555-1234</phone>*

*<email>bs@mail.com</email>*

*</contactlist>*

**Zippy Human Resources: XML for Employee Records, Part I**

Now that you have seen how elements are defined in a DTD, you have enough tools to follow along with a mini case study that shows how a company could use XML in its Human Resources department.

The Human Resources department for a small but growing company, Zippy Delivery Service, has decided that in order to make their employee data flexible across all the applications used by the company, the employee data should be stored in XML.

**The Zippy Human Resources department's first task is to decide on the fields to be included in the XML structure:**

* *Employee Name*
* *Position*
* *Age*
* *Sex*
* *Race*
* *Marital Status*
* *Address Line 1*
* *Address Line 2*
* *City*
* *State*
* *Zip Code*
* *Phone Number*
* *E-Mail Address*

After determining which elements are needed, they decide to put together a DTD in order to ensure that the structure of the employee records in the XML data file never changes.

Additionally, the decision is made that multiple employee records should be stored in a single file. Because this is the case, they need to declare a document (root) element to hold employee records and a parent element for the elements making up each individual employee record. The Human Resources department also realizes that some of the data might not be applicable to all employees. Therefore, they need to use element symbols to account for varying occurrences of data. They've come up with the following DTD structure as the first draft:

**Employees1.dtd**

*<!ELEMENT employees (employee+) >*

*<!ELEMENT employee (name, position, age, sex, race, m\_status,*

*address1,address2?, city, state, zip, phone?, email?) >*

*<!ELEMENT name (#PCDATA) >*

*<!ELEMENT position (#PCDATA) >*

*<!ELEMENT age (#PCDATA) >*

*<!ELEMENT sex (#PCDATA) >*

*<!ELEMENT race (#PCDATA) >*

*<!ELEMENT m\_status (#PCDATA) >*

*<!ELEMENT address1 (#PCDATA) >*

*<!ELEMENT address2 (#PCDATA) >*

*<!ELEMENT city (#PCDATA) >*

*<!ELEMENT state (#PCDATA) >*

*<!ELEMENT zip (#PCDATA) >*

*<!ELEMENT phone (#PCDATA) >*

*<!ELEMENT email (#PCDATA) >*

The Human Resources department has decided that the document element employees are required to have one or more (+) child elements (employee). The employee element would be the container element for each individual employee's data.

Out of the elements comprising the employee data, the Human Resources department knows that not all employees have a second line to their street address. Also, some employees do not have home telephone numbers or e-mail addresses.

Therefore, the elements address2, phone, and email are marked to appear zero or one time in each record (?).

The new DTD structure is saved in a file named employees1.

**The first several employee records are then entered into an XML document, called Employees1.xml:**

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

*<!DOCTYPE employees SYSTEM "employees1.dtd">*

*<employees>*

*<employee>*

*<name>Bob Jones</name>*

*<position>Dispatcher</position>*

*<age>37</age>*

*<sex>Male</sex>*

*<race>African American</race>*

*<m\_status>Married</m\_status>*

*<address1>202 Carolina St.</address1>*

*<city>Oklahoma City</city>*

*<state>OK</state>*

*<zip>73114</zip>*

*<phone>4055554321</phone>*

*<email>bobjones@mail.com</email>*

*</employee>*

*<employee>*

*<name>Mary Parks</name>*

*<position>Delivery Person</position>*

*<age>19</age>*

*<sex>Female</sex>*

*<race>Caucasian</race>*

*<m\_status>Single</m\_status>*

*<address1>1015 Empire Blvd.</address1>*

*<address2>Apt. D3</address2>*

*<city>Oklahoma City</city>*

*<state>OK</state>*

*<zip>73107</zip>*

*<phone>4055559876</phone>*

*<email>maryparks@mail.com</email>*

*</employee>*

*<employee>*

*<name>Jimmy Griffin</name>*

*<position>Delivery Person</position>*

*<age>23</age>*

*<sex>Male</sex>*

*<race>African American</race>*

*<m\_status>Single</m\_status>*

*<address1>1720 Maple St.</address1>*

*<city>Oklahoma City</city>*

*<state>OK</state>*

*<zip>73107</zip>*

*<phone>4055556633</phone>*

*</employee>*

*</employees>*

The XML document Employees1 initially has three employee records entered into it. The Document Type Declaration is entered after the XML declaration and before the document element, employees, and it uses the SYSTEM keyword to denote that it is referencing the DTD, employees1.dtd, externally.

The Human Resources department at Zippy Delivery Service feels that they are off to a good start. They have defined a DTD, employees1.dtd, for their XML data structure and have created an XML document, Employees1.xml (containing three employee records), that is valid according to the DTD. However, you'll find out during the course of this chapter that the Human Resources department's DTD can be improved.

**Zippy Human Resources: XML for Employee Records, Part II**

This is the second part of our mini case study on the use of XML in the Human Resources department at Zippy Delivery Service. You saw in Part I that the Human Resources department was able to put together a DTD (Employees1. dtd) and an XML document with some employee records (Employees1.xml). The DTD was referenced from the XML file for purposes of validation.

Upon review of their DTD, the members of the Human Resources department have decided that they are not quite satisfied. They feel that they have two types of information about each employee: personal information and contact information. They've decided that the personal information would be better stored as attributes of the employee name element rather than as individual elements. Additionally, they've decided that they need an ID type of attribute for each employee element in order to be able to quickly search the XML document. The DTD, therefore, has been amended as follows:

*<!ELEMENT employees (employee+) >*

*<!ELEMENT employee (name, position, address1, address2?, city, state,zip, phone?, email?) >*

*<!ATTLIST employee serial ID #REQUIRED >*

*<!ELEMENT name (#PCDATA) >*

*<!ATTLIST name*

*age CDATA #REQUIRED*

*sex CDATA #REQUIRED*

*race CDATA #IMPLIED*

*m\_status CDATA #REQUIRED >*

*<!ELEMENT position (#PCDATA) >*

*<!ELEMENT address1 (#PCDATA) >*

*<!ELEMENT address2 (#PCDATA) >*

*<!ELEMENT city (#PCDATA) >*

*<!ELEMENT state (#PCDATA) >*

*<!ELEMENT zip (#PCDATA) >*

*<!ELEMENT phone (#PCDATA) >*

*<!ELEMENT email (#PCDATA) >*

You can see that a new ID attribute, serial, has been added for the employee element. The serial attribute is marked as required (#REQUIRED). The age, sex, race, and m\_status elements have been removed and changed to attributes of the name element. Each of these attributes is character data (CDATA). Also, the race attribute has been deemed optional (#IMPLIED).

**The XML document has also been updated to reflect the new requirements of the changed DTD:**

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

*<!DOCTYPE employees SYSTEM "employees2.dtd">*

*<employees>*

*<employee serial="emp1">*

*<name age="37" sex="Male" race="African*

*American" m\_status="Married">*

*Bob Jones*

*</name>*

*<position>Dispatcher</position>*

*<address1>202 Carolina St.</address1>*

*<city>Oklahoma City</city>*

*<state>OK</state>*

*<zip>73114</zip>*

*<phone>4055554321</phone>*

*<email>bobjones@mail.com</email>*

*</employee>*

*<employee serial="emp2">*

*<name age="19" sex="Female" race="Caucasian"*

*m\_status="Single">*

*Mary Parks*

*</name>*

*<position>Delivery Person</position>*

*<address1>1015 Empire Blvd.</address1>*

*<address2>Apt. D3</address2>*

*<city>Oklahoma City</city>*

*<state>OK</state>*

*<zip>73107</zip>*

*<phone>4055559876</phone>*

*<email>maryparks@mail.com</email>*

*</employee>*

*<employee serial="emp3">*

*<name age="23" sex="Male" race="African*

*American" m\_status="Single">*

*Jimmy Griffin*

*</name>*

*<position>Delivery Person</position>*

*<address1>1720 Maple St.</address1>*

*<city>Oklahoma City</city>*

*<state>OK</state>*

*<zip>73107</zip>*

*<phone>4055556633</phone>*

*</employee>*

*</employees>*

In order for the XML document to remain valid according to the DTD, a serial attribute has been added for each employee element. Each serial attribute is set to a unique value. The age, sex, race, and m\_status elements have been removed and added as attributes of the name element.

The Zippy Human Resources department now feels that they are getting pretty close to having the DTD and XML structure they need in order to have an effective solution for storing their employee records. However, as you'll see in Part III, there is still a bit more tweaking that can be done with the addition of entities.

**Zippy Human Resources: XML for Employee Records, Part III**

This is the final part of the mini case study on the use of XML in the Human Resources department at Zippy Delivery Service. In Part II, the Human Resources department decided to change the structure of their DTD by moving the employees' personal data into attributes. This created a separation between personal data and contact data (which remained stored in elements).

At this point, the Human Resources department felt pretty satisfied with their work. Now, however, there are just a couple more minor areas where they feel the DTD (Employees2.dtd) could be improved. They've decided that they need to add several entities in order to speed the entry process for new records and to cut down on having to retype redundant information. First, they've added an entity for "Delivery Person". This makes sense to them because all but a few of the employees of Zippy Delivery Service are delivery people, and this will save them from having to type it over and over. The second entity they've decided to add is a parameter entity to give them a shortcut for entering #PCDATA type elements.

**Here's the updated DTD (you can download Employees3.dtd from the Sams Web site):**

*<!ENTITY dp "Delivery Person">*

*<!ENTITY % pc "#PCDATA">*

*<!ELEMENT employees (employee+) >*

*<!ELEMENT employee (name, position, address1, address2?, city, state,*

*zip, phone?, email?) >*

*<!ATTLIST employee serial ID #REQUIRED >*

*<!ELEMENT name (%pc;) >*

*<!ATTLIST name*

*age CDATA #REQUIRED*

*sex CDATA #REQUIRED*

*race CDATA #IMPLIED*

*m\_status CDATA #REQUIRED >*

*<!ELEMENT position (%pc;) >*

*<!ELEMENT address1 (%pc;) >*

*<!ELEMENT address2 (%pc;) >*

*<!ELEMENT city (%pc;) >*

*<!ELEMENT state (%pc;) >*

*<!ELEMENT zip (%pc;) >*

*<!ELEMENT phone (%pc;) >*

*<!ELEMENT email (%pc;) >*

In the new DTD, the entity dp is declared first. This entity is used to insert the value "Delivery Person" into the XML document when it is referenced. Next, the entity pc is declared. This is a parameter entity that holds the value "#PCDATA" for insertion into the DTD when referenced.

The XML document Employees2.xml has been updated to reflect the addition of the dp entity (the whole XML document is not listed because only a few lines actually changed; data not shown here should be assumed to be the same as in Parts I and II of this case study). Here's the code for Employees3.xml.

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

*<!DOCTYPE employees SYSTEM "employees3.dtd">*

*<employees>*

*<employee serial="emp1">*

*<name age="37" sex="Male" race="African*

*American" m\_status="Married">*

*Bob Jones*

*</name>*

*<position>Dispatcher</position>*

*...*

*</employee>*

*<employee serial="emp2">*

*<name age="19" sex="Female" race="Caucasian"*

*m\_status="Single">*

*Mary Parks*

*</name>*

*<position>&dp;</position>*

*...*

*</employee>*

*<employee serial="emp3">*

*<name age="23" sex="Male" race="African*

*American" m\_status="Single">*

*Jimmy Griffin*

*</name>*

*<position>&dp;</position>*

*...*

*</employee>*

*</employees>*

For the first employee, Bob Jones, the dp entity was not used for his position value because he is the company's dispatcher. However, for Mary Parks and Jimmy Griffin, the entity reference &dp; was inserted as the value for their position elements because they are both delivery people. This entity reference would also be used for any new employees added to the XML document that are delivery people.

The DTD for Zippy Deliver Service's Human Resources department is now complete. The DTD contains all the information required. It takes account for information that might not be applicable. The employees' personal and contact information has been logically separated between attributes and elements. Also, entities have been added to serve as timesaving devices for future additions to the XML document. The Zippy Human Resource department has built a DTD that will serve to validate their XML employee records effectively and efficiently.

**We covered the following items in relation to using DTDs:**

* A DTD may be internal to the XML document or external and referenced by the XML document.
* A DTD is attached to an XML document through a Document Type Declaration. A Document Type Declaration appears after the XML declaration and before the root element of the XML document. The Document Type Declaration may include a reference to an external DTD, encompass an internal DTD, or both.
* XML elements are declared and defined within the DTD. Elements are parsed from the top down, and elements in the XML document should appear in the same order they appear in the DTD. Element declarations have a specific set of rules and symbols that may be used in their definitions.
* XML attributes are declared and defined within the DTD. Attributes are not processed in a top-down fashion, but it is good programming practice to insert them after the element they reference. Attribute declarations have a specific set of types that may be used in their definitions.
* Entities are used in DTD as storage spaces or placeholders for data. Normally entities are used to store repetitive or bulky data for easy reference. There are four types of entities: internal, predefined, external, and parameter. Notations are used as references to help define the format of the external data.
* The IGNORE directive is used to indicate blocks of declarations that should not be included when the document is processed.
* The INCLUDE directive is used to indicate blocks of declarations that should be included when the document is processed. This directive is totally unnecessary to the successful processing of a DTD.
* Comments may be included in DTDs. Comments in DTDs are used in exactly the same way they are used in HTML.
* The DTD has several drawbacks that limit its scalability with respect to new and future XML applications. The XML Schema is the new recommendation from the W3C organization for XML validation.

**DTD Examples:**

**Description**

An XML document is valid if it has an associated document type definition and if the document complies with the constraints expressed in it. The document type definition must appear before the first element in the document. The name following the word DOCTYPE in the document type definition must match the name of root element.

**DTD**

A document can contain only the root element tutorial which can contain some text. :

*<!ELEMENT tutorial (#PCDATA)>*

**Valid documents**

A valid document containing some text.:

*<!DOCTYPE tutorial SYSTEM "tutorial.dtd">*

*<tutorial>This is an XML document</tutorial>*

**This document is also valid.:**

*<!DOCTYPE tutorial SYSTEM "tutorial.dtd">*

*<tutorial/>*

**Documents with errors**

**Root element does not conform to the given DTD.:**

*<!DOCTYPE tutorial SYSTEM "tutorial.dtd">*

*<text>This is an XML document</text>*

=====================================

**Description**

An element type has element content when elements of that type must contain only child elements (no character data), optionally separated by white space.

**DTD**

The root element XXX must contain precisely one element AAA followed by one element BBB. Elements AAA and BBB can contain some text but no other elements:

*<!ELEMENT XXX (AAA , BBB)>*

*<!ELEMENT AAA (#PCDATA)>*

*<!ELEMENT BBB (#PCDATA)>*

**Valid documents**

**A valid document containing some text:**

*<!DOCTYPE XXX SYSTEM "tutorial.dtd">*

*<XXX>*

*<AAA>Start</AAA>*

*<BBB>End</BBB>*

*</XXX>*

**This document is also valid:**

*<!DOCTYPE XXX SYSTEM "tutorial.dtd">*

*<XXX> <AAA/> <BBB/> </XXX>*

**Documents with errors**

**Element BBB is missing:**

*<!DOCTYPE XXX SYSTEM "tutorial.dtd">*

*<XXX> <AAA/> \_\_\_ </XXX>*

**Element BBB must follow element AAA:**

*<!DOCTYPE XXX SYSTEM "tutorial.dtd">*

*<XXX> <BBB/> <AAA/> </XXX>*

**Root element XXX can contain only one element BBB:**

*<!DOCTYPE XXX SYSTEM "tutorial.dtd">*

*<XXX> <AAA/> <BBB/> <BBB/> </XXX>*

**Root element XXX must not contain any text.:**

*<!DOCTYPE XXX SYSTEM "tutorial.dtd">*

*<XXX> Elements: <AAA/> <BBB/> </XXX>*

=======================================

**Description**

If an element name in DTD is followed by the star [\*], this element can occur zero, once or several times.

**DTD**

The root element XXX can contain zero or more elements AAA followed by precisely one element BBB. Element BBB must be always present.:

*<!ELEMENT XXX (AAA\* , BBB)>*

*<!ELEMENT AAA (#PCDATA)>*

*<!ELEMENT BBB (#PCDATA)>*

**Valid documents**

**A valid document:**

*<!DOCTYPE XXX SYSTEM "tutorial.dtd">*

*<XXX> <AAA/> <BBB/> </XXX>*

**Another valid document.**

The element AAA is not mandatory.:

*<!DOCTYPE XXX SYSTEM "tutorial.dtd">*

*<XXX> <BBB/> </XXX>*

**Several AAA elements can occur inside the document:**

*<!DOCTYPE XXX SYSTEM "tutorial.dtd">*

*<XXX> <AAA/> <AAA/> <AAA/> <AAA/> <AAA/> <AAA/> <AAA/> <BBB/> </XXX>*

**Documents with errors**

**Element BBB is missing:**

*<!DOCTYPE XXX SYSTEM "tutorial.dtd">*

*<XXX> \_\_\_ </XXX>*

**Element BBB must follow element AAA:**

*<!DOCTYPE XXX SYSTEM "tutorial.dtd">*

*<XXX> <BBB/> <AAA/> </XXX>*

**Element AAA must not follow element BBB:**

*<!DOCTYPE XXX SYSTEM "tutorial.dtd">*

*<XXX> <AAA/> <AAA/> <AAA/> <AAA/> <BBB/> <AAA/> <AAA/> </XXX>*

*===================================*

**Description**

If an element name in DTD is followed by the plus [+], this element can occur once or several times.

**DTD**

The root element XXX must contain one or several elements AAA followed by precisely one element BBB. Element BBB must be always present.:

*<!ELEMENT XXX (AAA+ , BBB)>*

*<!ELEMENT AAA (#PCDATA)>*

*<!ELEMENT BBB (#PCDATA)>*

**Valid documents**

**A valid document:**

*<!DOCTYPE XXX SYSTEM "tutorial.dtd">*

*<XXX> <AAA/> <BBB/> </XXX>*

**Several AAA elements can occur inside the document:**

*<!DOCTYPE XXX SYSTEM "tutorial.dtd">*

*<XXX> <AAA/> <AAA/> <AAA/> <AAA/> <AAA/> <AAA/> <AAA/> <BBB/> </XXX>*

**Documents with errors**

**Elements AAA and BBB are missing:**

*<!DOCTYPE XXX SYSTEM "tutorial.dtd">*

*<XXX> \_\_\_ \_\_\_ </XXX>*

**At least one element AAA must be present.:**

*<!DOCTYPE XXX SYSTEM "tutorial.dtd">*

*<XXX> \_\_\_<BBB/> </XXX>*

**Element BBB must follow element AAA:**

*<!DOCTYPE XXX SYSTEM "tutorial.dtd">*

*<XXX> <BBB/> <AAA/> </XXX>*

**Element AAA must not follow element BBB:**

*<!DOCTYPE XXX SYSTEM "tutorial.dtd">*

*<XXX> <AAA/> <AAA/> <AAA/> <AAA/> <BBB/> <AAA/> <AAA/> </XXX>*

=========================================

**Description**

If an element name in DTD is followed by the question mark [?], this element can occur zero or one times.

**DTD**

The root element XXX can contain one element AAA followed by precisely one element BBB. Element BBB must be always present.:

*<!ELEMENT XXX (AAA? , BBB)>*

*<!ELEMENT AAA (#PCDATA)>*

*<!ELEMENT BBB (#PCDATA)>*

**Valid documents**

**A valid document:**

*<!DOCTYPE XXX SYSTEM "tutorial.dtd">*

*<XXX> <AAA/> <BBB/> </XXX>*

**Element AAA is not mandatory:**

*<!DOCTYPE XXX SYSTEM "tutorial.dtd">*

*<XXX> <BBB/> </XXX>*

**Documents with errors**

**Element BBB is missing:**

*<!DOCTYPE XXX SYSTEM "tutorial.dtd">*

*<XXX> \_\_\_ </XXX>*

**Maximaly one AAA element can occur inside the document:**

*<!DOCTYPE XXX SYSTEM "tutorial.dtd">*

*<XXX> <AAA/> <AAA/> <AAA/> <AAA/> <AAA/> <AAA/> <AAA/> <BBB/> </XXX>*

**Element BBB must follow element AAA:**

*<!DOCTYPE XXX SYSTEM "tutorial.dtd">*

*<XXX> <BBB/> <AAA/> </XXX>*

===============================================

**Description**

This example uses a combination of [ + \* ?]

**DTD**

The root element XXX can contain one element AAA followed by one or more elements BBB. Element AAA can contain one element CCC and several elements DDD. Element BBB must contain precisely one element CCC and one element DDD:

*<!ELEMENT XXX (AAA? , BBB+)>*

*<!ELEMENT AAA (CCC? , DDD\*)>*

*<!ELEMENT BBB (CCC , DDD)>*

*<!ELEMENT CCC (#PCDATA)>*

*<!ELEMENT DDD (#PCDATA)>*

**Valid documents**

**A valid document:**

*<!DOCTYPE XXX SYSTEM "tutorial.dtd">*

*<XXX>*

*<AAA>*

*<CCC/><DDD/>*

*</AAA>*

*<BBB>*

*<CCC/><DDD/>*

*</BBB>*

*</XXX>*

**Elements in AAA are not mandatory:**

*<!DOCTYPE XXX SYSTEM "tutorial.dtd">*

*<XXX>*

*<AAA/>*

*<BBB>*

*<CCC/><DDD/>*

*</BBB>*

*</XXX>*

**Element AAA can be omitted:**

*<!DOCTYPE XXX SYSTEM "tutorial.dtd">*

*<XXX>*

*<BBB>*

*<CCC/><DDD/>*

*</BBB>*

*</XXX>*

**Documents with errors**

**Element BBB must contain elements CCC and DDD:**

*<!DOCTYPE XXX SYSTEM "tutorial.dtd">*

*<XXX>*

*<AAA/>*

*<BBB/>*

*</XXX>*

**Element AAA can contain maximaly one element CCC:**

*<!DOCTYPE XXX SYSTEM "tutorial.dtd">*

*<XXX>*

*<AAA>*

*<CCC/><CCC/>*

*<DDD/><DDD/>*

*</AAA>*

*<BBB>*

*<CCC/><DDD/>*

*</BBB>*

*</XXX>*

*=======================================*

**Description**

With character [ | ] you can select one from several elements.

**DTD**

The root element XXX must contain one element AAA followed by one element BBB. Element AAA must contain one element CCC followed by element DDD. Element BBB must contain either one element CCC or one element DDD:

*<!ELEMENT XXX (AAA , BBB)>*

*<!ELEMENT AAA (CCC , DDD)>*

*<!ELEMENT BBB (CCC | DDD)>*

*<!ELEMENT CCC (#PCDATA)>*

*<!ELEMENT DDD (#PCDATA)>*

**Valid documents**

**A valid document:**

*<!DOCTYPE XXX SYSTEM "tutorial.dtd">*

*<XXX>*

*<AAA>*

*<CCC/> <DDD/>*

*</AAA>*

*<BBB>*

*<CCC/>*

*</BBB>*

*</XXX>*

**Another valid document:**

*<!DOCTYPE XXX SYSTEM "tutorial.dtd">*

*<XXX>*

*<AAA>*

*<CCC/> <DDD/>*

*</AAA>*

*<BBB>*

*<DDD/>*

*</BBB>*

*</XXX>*

**Documents with errors**

**The element BBB can contain either element CCC or DDD but not both:**

*<!DOCTYPE XXX SYSTEM "tutorial.dtd">*

*<XXX>*

*<AAA>*

*<CCC/> <DDD/>*

*</AAA>*

*<BBB>*

*<CCC/> <DDD/>*

*</BBB>*

*</XXX>*

**The element BBB can contain either element CCC or DDD but not both:**

*<!DOCTYPE XXX SYSTEM "tutorial.dtd">*

*<XXX>*

*<AAA>*

*<CCC/> <DDD/>*

*</AAA>*

*<BBB>*

*<DDD/> <CCC/>*

*</BBB>*

*</XXX>*

====================================================

**Description**

Text can be interspersed with elements.

**DTD**

The element AAA can contain either BBB or CCC. On the other hand the element BBB can contain any combination of text and CCC elements.:

*<!ELEMENT XXX (AAA+ , BBB+)>*

*<!ELEMENT AAA (BBB | CCC )>*

*<!ELEMENT BBB (#PCDATA | CCC )\*>*

*<!ELEMENT CCC (#PCDATA)>*

**Valid documents**

**A valid document exploring several possibilities:**

*<!DOCTYPE XXX SYSTEM "tutorial.dtd">*

*<XXX>*

*<AAA>*

*<CCC>Precisely one element.</CCC>*

*</AAA>*

*<AAA>*

*<BBB>*

*<CCC/>*

*<CCC/>*

*<CCC/>*

*</BBB>*

*</AAA>*

*<BBB/>*

*<BBB>*

*This is <CCC/> a combination <CCC/> of <CCC> CCC elements </CCC> and text <CCC/>.*

*</BBB>*

*<BBB>*

*Text only.*

*</BBB>*

*</XXX>*

**Documents with errors**

**The element AAA cannot contain any text.:**

*<!DOCTYPE XXX SYSTEM "tutorial.dtd">*

*<XXX>*

*<AAA>*

*Element: <CCC/>*

*</AAA>*

*<BBB>*

*Element: <CCC/>*

*</BBB>*

*</XXX>*

================================================

**Description**

Attributes are used to associate name-value pairs with elements. Attribute specifications may appear only within start-tags and empty-element tags. The declaration starts with ATTLIST then follows the name of the element the attributes belong to and then follows the definition of the individual attributes.

**DTD**

An attribute of CDATA type can contain any character if it conforms to well formedness constraints. The required attribute must be always present, the implied attribute is optional:

*<!ELEMENT attributes (#PCDATA)>*

*<!ATTLIST attributes*

*aaa CDATA #REQUIRED*

*bbb CDATA #IMPLIED>*

**Valid documents**

**CDATA attribute can contain any character conforming to well-formedness constraints:**

*<!DOCTYPE attributes SYSTEM "tutorial.dtd">*

*<attributes aaa="#d1" bbb="\*~\*">*

*Text*

*</attributes>*

**The order of attributes is not important:**

*<!DOCTYPE attributes SYSTEM "tutorial.dtd">*

*<attributes bbb="$25" aaa="13%">*

*Text*

*</attributes>*

**The bbb attribute can be omitted as it is implied:**

*<!DOCTYPE attributes SYSTEM "tutorial.dtd">*

*<attributes aaa="#d1" />*

**Documents with errors**

The aaa attribute is required. Therefore it must be always present.:

*<!DOCTYPE attributes SYSTEM "tutorial.dtd">*

*<attributes \_\_\_ bbb="X24"/>*

================================================

**Description**

An attribute of CDATA type can contain any character if it conforms to well formedness constraints. NMTOKEN type can contain only letters, digits and point [ . ] , hyphen [ - ], underline [ \_ ] and colon [ : ] . NMTOKENS can contain the same characters as NMTOKEN plus whitespaces. White space consists of one or more space characters, carriage returns, line feeds, or tabs.

**DTD**

**The attributes bbb and ccc must be always present, the attribute aaa is optional:**

*<!ELEMENT attributes (#PCDATA)>*

*<!ATTLIST attributes*

*aaa CDATA #IMPLIED*

*bbb NMTOKEN #REQUIRED*

*ccc NMTOKENS #REQUIRED>*

**Valid documents**

**All required attributes are present and their values are of the correct type:**

*<!DOCTYPE attributes SYSTEM "tutorial.dtd">*

*<attributes aaa="#d1" bbb="a1:12" ccc=" 3.4 div -4"/>*

**All required attributes are present and their values are of the correct type:**

*<!DOCTYPE attributes SYSTEM "tutorial.dtd">*

*<attributes bbb="a1:12" ccc="3.4 div -4"/>*

**Documents with errors**

**The character # is not permitted in attributes of type NMTOKEN and NMTOKENS:**

*<!DOCTYPE attributes SYSTEM "tutorial.dtd">*

*<attributes aaa="#d1" bbb="#d1" ccc="#d1"/>*

**The space character is forbiden in attributes of NMTOKEN type:**

*<!DOCTYPE attributes SYSTEM "tutorial.dtd">*

*<attributes bbb="A B C" ccc="A B C"/>*

================================================

**Description**

The value of an attribute of ID type can contain only characters permitted for NMTOKEN and must start with a letter. No element type may have more than one ID attribute specified. The value of an ID attribute must be unique between all values of all ID attributes.

**DTD**

**The attributes id, code and X uniquely determine their element:**

*<!ELEMENT XXX (AAA+ , BBB+ , CCC+)>*

*<!ELEMENT AAA (#PCDATA)>*

*<!ELEMENT BBB (#PCDATA)>*

*<!ELEMENT CCC (#PCDATA)>*

*<!ATTLIST AAA*

*id ID #REQUIRED>*

*<!ATTLIST BBB*

*code ID #IMPLIED*

*list NMTOKEN #IMPLIED>*

*<!ATTLIST CCC*

*X ID #REQUIRED*

*Y NMTOKEN #IMPLIED>*

**Valid documents**

**All ID values are unique :**

*<!DOCTYPE XXX SYSTEM "tutorial.dtd">*

*<XXX>*

*<AAA id="a1"/>*

*<AAA id="a2"/>*

*<AAA id="a3"/>*

*<BBB code="QWQ-123-14-6" list="14:5"/>*

*<CCC X="zero" Y="16" />*

*</XXX>*

**The attributes list and Y are of type NMTOKEN not ID. They can have therefore the same value as ID attributes or to have the same value in several elements :**

*<!DOCTYPE XXX SYSTEM "tutorial.dtd">*

*<XXX>*

*<AAA id="L12"/>*

*<BBB code="QW" list="L12"/>*

*<CCC X="x-0" Y="QW" />*

*<CCC X="x-1" Y="QW" />*

*</XXX>*

**Documents with errors**

The ID attribute must not start with a number or contain a character not permitted in NMTOKEN:

*<!DOCTYPE XXX SYSTEM "tutorial.dtd">*

*<XXX>*

*<AAA id="L12"/>*

*<BBB code="#QW" list="L12"/>*

*<CCC X="12" Y="QW" />*

*</XXX>*

**The ID attribute must have a unique value:**

*<!DOCTYPE XXX SYSTEM "tutorial.dtd">*

*<XXX>*

*<AAA id="L12"/>*

*<BBB code="QW" list="L12"/>*

*<CCC X="ZA" Y="QW" />*

*<CCC X="ZA" Y="QW" />*

*</XXX>*

**The ID attribute must have a unique value. Both id and X are of type ID:**

*<!DOCTYPE XXX SYSTEM "tutorial.dtd">*

*<XXX>*

*<AAA id="L12"/>*

*<BBB code="QW" list="L12"/>*

*<CCC X="L12" Y="QW" />*

*</XXX>*

==============================================

**Description**

The value of IDREF attribute must match the value of some ID attribute in the document. The value of IDREFS attribute can contain several references to elements with ID attribute separated with whitespaces.

**DTD**

The attributes id and mark uniquely determine their element. The attributes ref refer to these elements:

*<!ELEMENT XXX (AAA+ , BBB+, CCC+, DDD+)>*

*<!ELEMENT AAA (#PCDATA)>*

*<!ELEMENT BBB (#PCDATA)>*

*<!ELEMENT CCC (#PCDATA)>*

*<!ELEMENT DDD (#PCDATA)>*

*<!ATTLIST AAA*

*mark ID #REQUIRED>*

*<!ATTLIST BBB*

*id ID #REQUIRED>*

*<!ATTLIST CCC*

*ref IDREF #REQUIRED>*

*<!ATTLIST DDD*

*ref IDREFS #REQUIRED>*

**Valid documents**

All ID values are unique and all IDREF and IDREFS point to elements with relevant IDs:

*<!DOCTYPE XXX SYSTEM "tutorial.dtd">*

*<XXX>*

*<AAA mark="a1"/>*

*<AAA mark="a2"/>*

*<AAA mark="a3"/>*

*<BBB id="b001" />*

*<CCC ref="a3" />*

*<DDD ref="a1 b001 a2" />*

*</XXX>*

**Documents with errors**

There are no ID attributes with value a3 or b001:

*<!DOCTYPE XXX SYSTEM "tutorial.dtd">*

*<XXX>*

*<AAA mark="a1"/>*

*<AAA mark="a2"/>*

*<BBB id="b01" />*

*<CCC ref="a3" />*

*<DDD ref="a1 b001 a2" />*

*</XXX>*

**The attribute ref in element CCC is of type IDREF. It can contain only one reference:**

*<!DOCTYPE XXX SYSTEM "tutorial.dtd">*

*<XXX>*

*<AAA mark="a1"/>*

*<AAA mark="a2"/>*

*<AAA mark="a3"/>*

*<BBB id="b001" />*

*<CCC ref="a1 b001 a2" />*

*<DDD ref="a1 b001 a2" />*

*</XXX>*

================================================

**Description**

Permitted attribute values can be defined in DTD.

**DTD**

**This DTD precisely states permitted values:**

*<!ELEMENT XXX (AAA+, BBB+)>*

*<!ELEMENT AAA (#PCDATA)>*

*<!ELEMENT BBB (#PCDATA)>*

*<!ATTLIST AAA*

*true ( yes | no ) #REQUIRED>*

*<!ATTLIST BBB*

*month (1|2|3|4|5|6|7|8|9|10|11|12) #IMPLIED>*

**Valid documents**

**All values are given in DTD:**

*<!DOCTYPE XXX SYSTEM "tutorial.dtd">*

*<XXX>*

*<AAA true="yes"/>*

*<AAA true="no"/>*

*<AAA true="yes"/>*

*<BBB month="8" />*

*<BBB month="2" />*

*<BBB month="12" />*

*</XXX>*

**Documents with errors**

The attribute true cannot have the value "maybe", the attribute month cannot have the value "16" :

*<!DOCTYPE XXX SYSTEM "tutorial.dtd">*

*<XXX>*

*<AAA true="yes"/>*

*<AAA true="no"/>*

*<AAA true="maybe"/>*

*<BBB month="8" />*

*<BBB month="2" />*

*<BBB month="16" />*

*</XXX>*

===============================================

**Description**

If an attribute is implied, a default value can be provided for the case when the attribute is not used.

**DTD**

Both attributes are implied. Their default value is given.:

*<!ELEMENT XXX (AAA+, BBB+)>*

*<!ELEMENT AAA (#PCDATA)>*

*<!ELEMENT BBB (#PCDATA)>*

*<!ATTLIST AAA*

*true ( yes | no ) "yes">*

*<!ATTLIST BBB*

*month NMTOKEN "1">*

**Valid documents**

The values of true are yes, no and yes. The values of month are 8, 2 and 1.:

*<!DOCTYPE XXX SYSTEM "tutorial.dtd">*

*<XXX>*

*<AAA true="yes"/>*

*<AAA true="no"/>*

*<AAA/>*

*<BBB month="8" />*

*<BBB month="2" />*

*<BBB/>*

*</XXX>*

=============================================

**Description**

An element can be defined EMPTY. In such a case it can contain only attributes but no text.

**DTD**

**The AAA elements can contain only attributes but no text:**

*<!ELEMENT XXX (AAA+)>*

*<!ELEMENT AAA EMPTY>*

*<!ATTLIST AAA*

*true ( yes | no ) "yes">*

**Valid documents**

**Both these forms are allowed. In the second case the ending tag must immediately follow the starting one:**

*<!DOCTYPE XXX SYSTEM "tutorial.dtd">*

*<XXX>*

*<AAA true="yes"/>*

*<AAA true="no"></AAA>*

*</XXX>*

**Documents with errors**

**The AAA element must not contain any text and the starting tag must be immediately closed:**

*<!DOCTYPE XXX SYSTEM "tutorial.dtd">*

*<XXX>*

*<AAA true="yes"/>*

*<AAA true="no"></AAA>*

*<AAA> </AAA>*

*<AAA>Hello!</AAA>*

*</XXX>*

=============================================