Document Type Definition (DTD)

Objectives

To understand what a DTD is
To be able to write DTDs
To be able to declare elements and attributes in a DTD
To understand the difference between general entities and parameter entities
To be able to use conditional sections with entities
To be able to use NOTATIONS
To understand how a whitespace is to be processed

Outline

Introduction
Document Tvpe Definitions (DTDs) define an XML document's structure (e.g., what elements, attributes, etc. are permitted in the document). An XML document is not required to have a corresponding DTD. However, DTDs are often recommended to ensure document conformity,

especially in *business-to-business (B2B) transactions,* where XML documents are exchanged. DTDs specify an XML document's structure and are themselves defined using *EBNF (Extended Backus-Naur Form)* grammar-not the XML syntax.

Observation:

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

Parsers, Well-formed and Valid XML Documents

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

Document Type Declaration

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

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

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

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

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

which points to the **myDTD.dtd** document. Using the **PUBLIC** keyword indicates that the DTD is widely used (e.g., the DTD for HTML documents). The DTD may be made available in well-known locations for more efficient downloading. The **DOCTYPE**

<!DOCTYPE HTML PUBLIC "-//w3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">

uses the **PUBLIC** keyword to reference the well-known DTD for HTML version 4.01. XML parsers that do not have a local copy of the DTD may use the URL provided to download the DTD to perform validation.

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

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

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

Observation:

- *The document type declaration internal subset plus its external subset form the DTD.*

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

Element **Type** Declarations

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

<!ELEMENT myElement (#PCDATA)>

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

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

Example 1 lists an XML document that contains a reference to an external DTD in the **I)OCTYPE.** Microsoft's XML Validator will be used to check the document's conformity against its DTD. To use XML Validator, Internet Explorer 5 is required. Parsers XML4J and

Xerces can be used to check a document's validity against a DTD programmatically. Using Java and one of these parsers provides a platform-independent way to validate XML documents.

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

Example 1. XML document declaring associated DTD.

The document type declaration is named **myMessage-the** name of the root element. The element **myMessage** contains a single child element named **message**.

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

Example 2. Validation with using an external DTD

The DTD declares element **myMessage.** Notice that the content specification contains the name **message.** This indicates that element **myMessage** contains exactly one child element named **message.** Because **myMessage** can only have an element as its content, it is said to have *element content.* Element **message** whose content is of type **PCDATA.** The XML Validator is capable of validating an XML document against both DTDs and Schemas.

Error:

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

If an XML document's structure is inconsistent with its corresponding DTD but is syntactically correct, it is only well formed-not valid. The XML Validator will generate error messages in such cases.

Sequences, pipe characters, occurrence indicators

DTDs allow the document author to define the order and frequency of child elements. The *comma* (,) - called a *sequence* - specifies the order in which the elements must occur. For example,
<!ELEMENT classroom (**teacher, student**)**>**

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

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

<!ELEMENT dessert (iceCream | pastry)>

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

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

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

Example 4 Occurrence indicators.

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

<!ELEMENT album (song+)>

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

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

<!ELEMENT album (title, (songtitle, duration)+)>

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

```
<album>
<title>XML Classical Hits</title>
<songTitle>XML overture</songTitle>
<duration>lo</duration>
<songTitle>XML Symphony 1.0</songTitle>
<duration>54</duration>
</album>
```

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

<!ELEMENT library (book*)>

indicates that clement library contains any number of book elements, including the possibility of none at all. Markup examples that conform to this are:

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

and

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

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

<!ELEMENT seat (person?)>

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

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

and

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

EMPTY, Mixed Content and **ANY**

Elements must be further refined by specifying the types of content they contain. The element content introduced were indicating that an element can contain one or more child elements as its content. *Content specification types* describe non-element content.

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

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

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

```
<oven/>
```

in an XML document conforming to this declaration.

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

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

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

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

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

Example 5 specifies a DTD as an internal subset as opposed to an external subset (example 1). In the prolog there is **standalone** attribute with a value of **yes.** An XML document is **standalone** if it does not reference an external subset. This DTD defines three elements: one that contains mixed content and two that contain parsed character data.

```
<?xml version = "1.0" standalone = "yes"?>
<!DOCTYPE format [
<!ELEMENT format ( #PCDATA | bold | italic )*>
<!ELEMENT bold ( #PCDATA )>
<!ELEMENT italic ( #PCDATA )>
 ]>
<format>
```

```
This is a simple formatted sentence.
<bold>l have tried bold.</bold>
<italic>l have tried italiC.</italic>
  Now what?
</format>
```

Example 5 A mixed-content element.

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

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

Error:

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

Observation:

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

Attribute Declarations

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

<!ELEMENT x EMPTY>
<!ATTLIST x y CDATA #REQUIRED>
declares **EMPTY** element **x.** The attribute declaration specifies that **y** is an attribute of **x.** Keyword **CDATA** indicates that **y** can contain any character text except for the <, >, &, ' and " characters. Note that the **CDATA** keyword in an attribute declaration has a different meaning than the **CDATA** section in an XML document. Recall that in a **CDATA** section all characters

are legal except the]] > end tag. Keyword **#REQUIRED** specifies that the attribute must be provided for element **x.**

Example 7 demonstrates how to specify attribute declarations for an element. Attributes **id** and **to are** declared for element **message.** Both **id** and **to** contain required **CDATA.** Attribute values are normalized (i.e., consecutive whitespace characters are combined into one whitespace character). Attribute **id is** assigned the value **"451",** and attribute **to** is assigned the value **"The world".**

```
<?xml version = "1.0"?>
<!DOCTYPE myMessage [
<!ELEMENT myMessage ( message )>
<!ELEMENT message ( #PCDATA )>
<!ATTLIST message id CDATA #REQUIRED>
 ]>
<myMessage>
<message id = "445">
  Welcome to XML!
</message>
</myMessage>
```

Example 7. Declaring attributes.

Attribute Defaults (#REQUIRED, #IMPLIED, #FIXED)

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

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

<message>number</message>

when checked against the DTD attribute list declaration

<!ATTLIST message number CDATA #REQUIRED>

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

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

<!ATTLIST address zip #FIXED "02115">

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

Attribute Types

Attribute types are classified as either *strings* **(CDATA),** *tokenized or enumerated.* String attribute types do not impose any constraints on attribute values-other than disallowing the <, >, &, ' and " characters. Entity references (e.g., **&lt;, &gt;,** etc.) must be used for these characters. *Tokenized attributes* impose constraints on attribute values-such as which characters are permitted in an attribute name. *Enumerated attributes* are the most restrictive of the three types. They can take only one of the values listed in the attribute declaration.

Tokenized Attribute Type (ID, IDREF, ENTITY, NMTOKEN)

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

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

Example 8 lists an XML document that uses **ID** and **IDREF** attribute types. Element **bookstore** consists of element **shipping** and element **book.** Each **shipping** element describes a shipping method.

Attribute **shipID** is declared as an **ID** type attribute (i.e., each **shipping** element has a unique identifier). **Book** elements is declared with attribute **shippedBy** of type **IDREF.** Attribute **shippedBy** points to one of the **ship ping** elements by matching its **shipID** attribute.

If to assign **shippedBy** the value **"s3",** an error occurs when to use Microsoft's Validator. No **shipID** attribute has a value **"s3"**, which results in a non-valid XML document.

```
<?xml version = "1.0"?>
<!DOCTYPE bookstore [
<!ELEMENT bookstore (shipping+, book+ )>
<!ELEMENT shipping ( duration )>
<!ATTLIST shipping shipID ID #REQUIRED>
<!ELEMENT book ( #PCDATA )>
<!ATTLIST  book shippedBy IDREF #IMPLIED>
<!ELEMENT duration ( #PCDATA )>
 ]>
<bookstore>
<shipping shipID = "s1">
  <duration>2 to 4 days</duration>
</shipping>
<shipping shipID = "s2">
  <duration>l day</duration>
</shipping>
<book shippedBy = "s2">
  Java How to Program 3rd edition.
</book>
<book shippedBy = "s2"
  C How to Program 3rd edition.
</book>
<book shippedBy = "s1"
  C++ How to Program 3rd edition.
</book>
</bookstore>
```

Example 8. XML document with ID and IDREF attribute types

The file **(lang.xml)** referenced **lang.dtd,** which contained the values for the entity references **&assoc;** and **&text;.** External subset **lang. dtd** contains the two *entity declarations*
<!ENTITY assoc
"&#1571;&#1587;&#1617;&#1608;&#1588;&#1616;&#1610;&#1614;&#1578;&#1618;&#
#1587;11>

and

<!ENTITY text "&#1575;&#1604;&#1610;&#1608;&#1606;&#1610;&#1603;&#1608;&#1583;;">

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

<!ENTITY digits "0123456789">

for **digits.** This entity might be used as follows

<useAnEntity>&digits;</useAnEntity>

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

<useAnEntity>0123456789</useAnEntity>

the value **0123456789** being placed inside the tags. These entities are called *general entities.*

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

Example 9 lists an XML document that demonstrates the use of entities and entity attribute types.

```
<?xml version = "1.0"?>
<!DOCTYPE database [
<!NOTATION html SYSTEM "iexplorer">
<!ENTITY city SYSTEM "tour.html" NDATA html>
<!ELEMENT database ( company+ )>
<!ELEMENT company ( name )>
<!ATTLIST company tour ENTITY #REQUIRED>
<!ELEMENT name ( #PCDATA )>
 ]>
<database>
<company tour = "city">
```

```
<name>Deitel &amp; Associates, Inc.</name>
</company>
</database>
```

Example 9 XMLdocument containing an ENTITY attribute type

<!NOTATION html SYSTEM "iexplorer">

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

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

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

<!ATTLIST company tour ENTITY #REQUIRED>

declares attribute **tour** for element **company.** Attribute **tour** specifies a required **ENTITY** attribute type.

<company tour = "city">

assigns entity **city** to attribute **tour.** Replacing this with

<company tour = "country">

the document fails to conform to the DTD because entity **country** does not exist. Validator if **country** is used.

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

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

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

<directory file = "animations graph1 graph2">

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

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

<!ATTLIST sportsClub phone NMTOKEN #REQUIRED>

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

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

An example that does not conform to this is

<sportsClub phone = "555 555 4902">

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

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

Enumerated Attribute Types

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

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

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

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

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

NOTATION is also an enumerated attribute type. For example,

<!ATTLIST book reference NOTATION (JAVA | C) "C">
the declaration indicates that **reference** must be assigned either **JAVA** or **C.** If a value is not assigned, **C** is specified as the default. The notation for **C** might be declared as

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

Conditional Sections

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

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

directs the parser to include the declaration of element name.

Similarly, the conditional section

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

directs the parser to exclude the declaration of element **message.**

Conditional sections are often used with entities, as demonstrated in example 10.

```
<!ENTITY % reject "IGNORE">
<!ENTITY % accept "INCLUDE">
<![ %accept;   [
<!ELEMENT message ( approved, signature )>
 ] ] >
<![ %reject;   [
<!ELEMENT message ( approved, reason, signature )>
 ] ] >
<!ELEMENT approved EMPTY>
<!ATTLIST approved flag ( true I false ) "false">
<!ELEMENT reason ( #PCDATA )>
<!ELEMENT signature ( #PCDATA )>
```

Example 10. Conditional sections in a DTD

```
<?xml version = "1.0" standalone = "no"?>
<!DOCTYPE message SYSTEM "conditional.dtd">
```

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

Example 11 XMLdocument that conforms to conditional.dtd.

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

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

The entities **accept** and **reject,** which represent the strings **INCLUDE** and **IGNORE,** are used respectively. Notice that the parameter entity references are preceded by %, where as normal entity references are preceded by &.

<![**%accept; [**

represents the beginning tag of an **IGNORE** section (the value of the **accept** entity is **IGNORE),** while another line represents the start tag of an **INCLUDE** section. By changing the values of the entities, we can easily choose which **message** element declaration to allow. Example 11 shows the XML document that conforms to the DTD in example 11.

Observation:

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

Whitespace Characters

Whitespace characters and normalization are discussed how they relate to DTDs. Whitespace is either preserved or normalized, depending on the context in which it is used. Example 12 contains a DTD and markup that conforms to the DTD. The output shown is generated by a Java application.

```
<?xml version = "1.0"?>
<!DOCTYPE whitespace    [
<!ELEMENT whitespace ( hasCDATA,
  hasID, hasNMTOKEN, hasEnumeration, hasMixed )>
<!ELEMENT hasCDATA EMPTY>
<!ATTLIST hasCDATA cdata CDATA #REQUIRED>
<!ELEMENT hasID EMPTY>
<!ATTLIST hasID id ID #REQUIRED>
<!ELEMENT hasNMTOKEN EMPTY>
<!ATTLIST hasNMTOKEN rmtoken NMTOKEN
#REQUIRED>
<!ELEMENT hasEnumeration EMPTY>
<!ATTLIST hasEnuxneration enumeration ( true | false )
#REQUIRED>
<!ELEMENT hasMixed (#PCDATA | hasCDATA)*>
] >
<whitespace>
<hasCDATA cdata = "simple cdata"/>
<hasID id = "i20"/>
<hasNMTOKEN nmtoken = "hello"/>
<hasEnumeration enumeration = "true"/>
<hasMixed>
   This is text.
<hasCDATA cdata = " simple  cdata"/>
   This is some additional text.
</hasMixed>
</whitespace>
```

Example 12 Processing whitespace in an XML document

<hasCDATA cdata = " **simple cdata"/>**

assigns a value containing multiple whitespace characters to attribute **cdata.** Attribute **cdata** is required and must contain **CDATA.** As mentioned earlier, **CDATA** can contain almost any text, including whitespace. As the output illustrates, spaces in **CDATA** are preserved and passed on to the application using the XML document.

Attribute **id** is declared with tokenized attribute type **ID.** Because this is not **CDATA,** it is normalized and the leading whitespace characters are removed. Similarly, values that contain leading whitespace are assigned to attributes **nmtoken** and **enumeration** which are declared in the DTD as an **NMTOKEN** and an enumeration, respectively. Both these attributes are normalized by the parser.

Internet and World Wide Web Resources

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

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

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

wdvl.internet.com/Authoring/Languages/XML/Tutorials/Intro/index3.html
A DTD tutorial.

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

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

msdn.microsoft.com/downloads/samples/Internet/xml/xml-validator/ sample.asp
Download page for Microsoft's XML Validator.

www.networking.ibm.com/xml/xmlvalidatorForm.html
IBM's DOMit XML Validator.

SUMMARY

1. Document Type Definitions (DTDs) define an XML document's structure (e.g., what elements, attributes, etc. are permitted in the XML document). An XML document is not required to have a corresponding DTD. DTDs use EBNF (Extended Backus-Naur Form) grammar.
2. Parsers are generally classified as validating or nonvalidating. A validating parser is able to read the DTD and *determine whether or not the XML document conforms to it. If the document* conforms to the DTD, it is referred to as valid. If the document fails to conform to the DTD but is syntactically correct, it is well formed but not valid. By definition, a valid document is well formed.
3. A nonvalidating parser is able to read a DTD, but cannot check the document against the DTD for conformity. If the document is syntactically correct, it is well formed.

4. DTDs are introduced into XML documents by using the document type declaration (i.e., **DOCTYPE).** The document type declaration can point to declarations that are outside the XML document (called the external subset) or can contain the declaration inside the document (called internal subset).
5. External subsets physically exist in a different file that typically ends with the **.dtd** extension, although this file extension is not required. External Subsets are specified using keyword **SYSTEM.** Both the internal and external subset may be specified at the same time.
6. Elements are the primary building block used in XML documents and are declared in a DTD with element type declarations **(ELEMENTs).**
7. The element name that follows **ELEMENT** is often called a generic identifier. The set of parentheses that follow the element name specify the element's allowed content and is called the content specification.
8. Keyword **PCDATA** specifies that the element must contain parsable character data-that is, any text except the characters less-than (<), greater-than (>), ampersand (&), quote (') and double quote (").
9. An XML document is a **standalone** XML document if it does not reference an external DTD.
10. **An** XML element that can only have another element for content, it is said to have element content.
11. DTDs allow the document author to define the order and frequency of child elements. The comma (,) - called a sequence - specifies the order in which the elements must occur. Choices are specified using the pipe (|) character. The content specification may contain any number of pipe character separated choises.
12. An element's frequency (i.e., number of occurrences) is specified by using either the plus sign (+), asterisk (*) or question mark (?) occurrence indicator.
13. The frequency of an element group (i.e., two or more elements that occur in some combinaition) is specified by enclosing the element names inside the content specification followed by an occurrence indicator.
14. Elements can be further refined by describing the content types they may contain. Content specification types (e.g., **EMPTY,** mixed content, **ANY,** etc.) describe nonelement content.
15. An element can be declared as having mixed content (i.e., a combination of elements and **PCDATA).** The comma (,), plus sign (+) and question mark (?) occurrence indicators cannot be used with mixed content elements.
16. An element declared as type **ANY** can contain any content including **PCDATA,** elements, or a cornbination of elements and **PCDATA.** Elements with **ANY** content can also be empty elements.
17. An attribute list for an element is declared using the **ATTLIST** element type declaration.
18. Attribute values are normalized (i.e., consecutive whitespace characters are combined into one whitespace character).
19. DTDs allow document authors to specify an attribute's default value using attribute defaults. Keywords **#IMPLIED, #REQUIRED** and **#FIXED** are attribute defaults.
20. Keyword **#IMPLIED** specifies that if the attribute does not appear in the element, then the application using the XML document can apply whatever value (if any) it chooses.
21. Keyword **#REQUIRED** indicates that the attribute must appeair in the element. The XML document is not valid if the attribute is missing.
22. An attribute declaration with default value **#FIXED** specifies that the attribute value is constant and cannot be different in the XML document.

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