Dr. SNS RAJALAKSHMI COLLEGE OF ARTS & SCIENCE(AUTONOMOUS)

Accredited by NAAC (Cycle III) with ‘A+’ GradeAffiliated to

Bharathiar University

Coimbatore-641049

DEPARTMENT OF COMPUTER APPLICATIONS

III BCA

WEB DEDIGNING-16UCA504

PREPARED BY: Mr.Lalitha

UNIT-V

UNIT-V

What is xml

- **Xml** (eXtensible Markup Language) is a mark up language.
- XML is designed to store and transport data.
- Xml was released in late 90's. it was created to provide an easy to use and store self describing data.
- XML became a W3C Recommendation on February 10, 1998.
- XML is not a replacement for HTML.
- XML is designed to be self-descriptive.
- XML is designed to carry data, not to display data.
- XML tags are not predefined. You must define your own tags.
- XML is platform independent and language independent.

What is mark-up language

A **mark up language** is a modern system for highlight or underline a document.

Students often underline or highlight a passage to revise easily, same in the sense of modern mark up language highlighting or underlining is replaced by tags.

Why xml

Platform Independent and Language Independent: The main benefit of xml is that you can use it to take data from a program like Microsoft SQL, convert it into XML then share that XML with other programs and platforms. You can communicate between two platforms which are generally very difficult. The main thing which makes XML truly powerful is its international acceptance. Many corporation use XML interfaces for databases, programming, office application mobile phones and more. It is due to its platform independent feature.

Features and Advantages of XML

XML is widely used in the era of web development. It is also used to simplify data storage and data sharing.

The main features or advantages of XML are given below.

1) XML separates data from HTML

If you need to display dynamic data in your HTML document, it will take a lot of work to edit the HTML each time the data changes.

With XML, data can be stored in separate XML files. This way you can focus on using HTML/CSS for display and layout, and be sure that changes in the underlying data will not require any changes to the HTML.

With a few lines of JavaScript code, you can read an external XML file and update the data content of your web page.

2) XML simplifies data sharing

In the real world, computer systems and databases contain data in incompatible formats.

XML data is stored in plain text format. This provides a software- and hardware-independent way of storing data.

This makes it much easier to create data that can be shared by different applications.

3) XML simplifies data transport

One of the most time-consuming challenges for developers is to exchange data between incompatible systems over the Internet.

Exchanging data as XML greatly reduces this complexity, since the data can be read by different incompatible applications.

Character and mark up:
Markup text is enclosed in angle brackets (< and >). Character data (sometimes called element content) is the text delimited by the start tag and end tag.
XML stands for **extensible markup language**. A markup language is a set of codes, or tags, that describes the text in a digital document. The most famous markup language is hypertext markup language (HTML), which is used to format Web pages.

XML documents are composed of **markup** and **content**. Using these two fundamental building blocks an XML document can be used to represent a wide variety of data ranging from software GUI files to high end technical publications.

There are six kinds of markup that can occur in an XML document: elements, entity references, comments, processing instructions, marked sections, and document type declarations.

Elements :

These are the most common form of markup. Delimited by angle brackets, most elements identify the nature of the content they surround. Some elements may be empty in which case they have no content. If an element is not empty, it begins with a start-tag, **<element>**, and ends with an end-tag, **</element>.**

Attributes :

These are name-value pairs that occur inside start-tags after the element name. For example **<fontdata classtype="bold">** is a **fontdata** element with the attribute **classtype** having the value bold. In XML, all attribute values must be quoted.

Entity References :

The XML specification reserves the use of certain characters such as < and >. In order to insert these characters into your document as content, there must be an alternative way to represent them. In XML, entities are used to represent these special characters. Entities are also used to refer to often repeated or varying text and to include the content of external files.

Every entity must have a unique name. In order to use an entity, you simply reference it by name. Entity references begin with the ampersand and end with a semicolon.

For example, the **lt** entity inserts a literal **<** into a document. So the string **<element>** can be represented in an XML document as **<element>**.

A special form of entity reference, called a **character reference**, can be used to insert arbitrary Unicode characters into your document. This is a mechanism for inserting characters that cannot be typed directly on your keyboard.

Character references take one of two forms: **decimal references**, ℞, and **hexadecimal references**, ℞. Both of these refer to character number U+211E from Unicode.

Comments :

These begin with **<!--** and end with **-->**. Comments can contain any data except the literal string **--**. You can place comments between markup anywhere in your document.

Comments are not part of the textual content of an XML document and are displayed in Alchemy CATALYST as locked strings.

Processing Instructions :

Commonly referred to as **PI** instructions, they provide an escape hatch used to send raw data to an XML application. Like comments, they are not textually part of the XML document, but the XML processor is required to pass them to an application. Processing instructions have the form: **<?name pidata?>**. The name, called the **PI target**, identifies the **PI** to the application. Applications should process only the targets they recognize and ignore all other PIs.

Any data that follows the PI target is optional, it is for the application that recognizes the target. The names used in PIs may be declared as notations in order to formally identify them. PI names beginning with xml are reserved for XML standardization.

CDATA Sections :

In a document, a **CDATA** section instructs the parser to ignore most markup characters.

Consider a source code listing in an XML document. It might contain characters that the XML parser would ordinarily recognize as markup (< and &, for example). In order to prevent this, a CDATA section can be used.

```
<![CDATA[*p = &q;b = (i <= 3);]]>
```

Between the start of the section, **<![CDATA[** and the end of the section, **]]>**, all character data is passed directly to the application, without interpretation. Elements, entity references, comments, and processing instructions are all unrecognized and the characters that comprise them are passed literally to the application.

XML Namespaces

XML Namespaces provide a method to avoid element name conflicts.

Name Conflicts

In XML, element names are defined by the developer. This often results in a conflict when trying to mix XML documents from different XML applications.

This XML carries HTML table information:

```
<table>
  <tr>
    <td>Apples</td>
```

```
    <td>Bananas</td>
  </tr>
</table>
```

This XML carries information about a table (a piece of furniture):

```
<table>
  <name>African Coffee Table</name>
  <width>80</width>
  <length>120</length>
</table>
```

If these XML fragments were added together, there would be a name conflict. Both contain a <table> element, but the elements have different content and meaning.

A user or an XML application will not know how to handle these differences.

Solving the Name Conflict Using a Prefix

Name conflicts in XML can easily be avoided using a name prefix.

This XML carries information about an HTML table, and a piece of furniture:

```
<h:table>
  <h:tr>
    <h:td>Apples</h:td>
    <h:td>Bananas</h:td>
  </h:tr>
</h:table>

<f:table>
  <f:name>African Coffee Table</f:name>
  <f:width>80</f:width>
  <f:length>120</f:length>
</f:table>
```

In the example above, there will be no conflict because the two <table> elements have different names.

XML Namespaces - The xmlns Attribute

When using prefixes in XML, a **namespace** for the prefix must be defined.

The namespace can be defined by an **xmlns** attribute in the start tag of an element.

The namespace declaration has the following syntax. xmlns:*prefix*="*URI*".

```
<root>

<h:table xmlns:h="http://www.w3.org/TR/html4/">
  <h:tr>
    <h:td>Apples</h:td>
    <h:td>Bananas</h:td>
  </h:tr>
</h:table>

<f:table xmlns:f="https://www.w3schools.com/furniture">
  <f:name>African Coffee Table</f:name>
  <f:width>80</f:width>
  <f:length>120</f:length>
</f:table>

</root>
```

In the example above:

The xmlns attribute in the first <table> element gives the h: prefix a qualified namespace.

The xmlns attribute in the second <table> element gives the f: prefix a qualified namespace.

When a namespace is defined for an element, all child elements with the same prefix are associated with the same namespace.

Namespaces can also be declared in the XML root element:

```
<root xmlns:h="http://www.w3.org/TR/html4/"
xmlns:f="https://www.w3schools.com/furniture">

<h:table>
  <h:tr>
    <h:td>Apples</h:td>
    <h:td>Bananas</h:td>
```

```
  </h:tr>
</h:table>

<f:table>
  <f:name>African Coffee Table</f:name>
  <f:width>80</f:width>
  <f:length>120</f:length>
</f:table>

</root>
```

Note: The namespace URI is not used by the parser to look up information.

The purpose of using an URI is to give the namespace a unique name.

However, companies often use the namespace as a pointer to a web page containing namespace information.

Uniform Resource Identifier (URI)

A **Uniform Resource Identifier** (URI) is a string of characters which identifies an Internet Resource.

The most common URI is the **Uniform Resource Locator** (URL) which identifies an Internet domain address. Another, not so common type of URI is the **Uniform Resource Name** (URN).

Default Namespaces

Defining a default namespace for an element saves us from using prefixes in all the child elements. It has the following syntax:

xmlns="*namespaceURI*"

This XML carries HTML table information:

```
<table xmlns="http://www.w3.org/TR/html4/">
  <tr>
    <td>Apples</td>
    <td>Bananas</td>
  </tr>
</table>
```

This XML carries information about a piece of furniture:

```
<table xmlns="https://www.w3schools.com/furniture">
  <name>African Coffee Table</name>
  <width>80</width>
  <length>120</length>
</table>
```

Namespaces in Real Use

XSLT is a language that can be used to transform XML documents into other formats.

The XML document below, is a document used to transform XML into HTML.

The namespace "http://www.w3.org/1999/XSL/Transform" identifies XSLT elements inside an HTML document:

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

<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<html>
<body>
  <h2>My CD Collection</h2>
  <table border="1">
    <tr>
      <th style="text-align:left">Title</th>
      <th style="text-align:left">Artist</th>
    </tr>
    <xsl:for-each select="catalog/cd">
    <tr>
      <td><xsl:value-of select="title"/></td>
      <td><xsl:value-of select="artist"/></td>
    </tr>
    </xsl:for-each>
  </table>
</body>
</html>
</xsl:template>
```

```
</xsl:stylesheet>
```

XML DTD

An XML document with correct syntax is called "Well Formed".

An XML document validated against a DTD is both "Well Formed" and "Valid".

What is a DTD?

DTD stands for Document Type Definition.

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

Valid XML Documents

A "Valid" XML document is "Well Formed", as well as it conforms to the rules of a DTD:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE note SYSTEM "Note.dtd">
<note>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

The DOCTYPE declaration above contains a reference to a DTD file. The content of the DTD file is shown and explained below.

XML DTD

The purpose of a DTD is to define the structure and the legal elements and attributes of an XML document:

Note.dtd:

```
<!DOCTYPE note
[
<!ELEMENT note (to,from,heading,body)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)>
]>
```

The DTD above is interpreted like this:

- !DOCTYPE note - Defines that the root element of the document is note
- !ELEMENT note - Defines that the note element must contain the elements: "to, from, heading, body"
- !ELEMENT to - Defines the to element to be of type "#PCDATA"
- !ELEMENT from - Defines the from element to be of type "#PCDATA"
- !ELEMENT heading - Defines the heading element to be of type "#PCDATA"
- !ELEMENT body - Defines the body element to be of type "#PCDATA"

Using DTD for Entity Declaration

A DOCTYPE declaration can also be used to define special characters or strings, used in the document:

Example

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

<!DOCTYPE note [
<!ENTITY nbsp " ">
<!ENTITY writer "Writer: Donald Duck.">
<!ENTITY copyright "Copyright: W3Schools.">
]>

<note>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
<footer>&writer; &copyright;</footer>
</note>
```

When to Use a DTD?

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

With a DTD, you can verify that the data you receive from the outside world is valid.

You can also use a DTD to verify your own data.

XML Information Set

XML Information Set (**XML Infoset**) is a W3C specification describing an abstract data model of an XML document in terms of a set of *information items*.[1] The definitions in the XML Information Set specification are meant to be used in *other* specifications that need to refer to the information in a well-formed XML document.

An XML document has an information set if it is well-formed and satisfies the namespace constraints. There is no requirement for an XML document to be valid in order to have an information set.

An information set can contain up to eleven different types of information items:

1. The Document Information Item (always present)
2. Element Information Items
3. Attribute Information Items
4. Processing Instruction Information Items
5. Unexpanded Entity Reference Information Items
6. Character Information Items
7. Comment Information Items
8. The Document Type Declaration Information Item
9. Unparsed Entity Information Items
10. Notation Information Items
11. Namespace Information Items

XML was initially developed without a formal definition of its infoset. This was only formalised by later work beginning in 1999, first published as a separate W3C Working Draft at the end of December that year.[2] Infoset recommendation Second Edition was adopted on 4 February, 2004.[3] If a 2.0 version of the XML standard is ever published, it is likely that this would absorb the Infoset recommendation as an integral part of that standard.

Infoset augmentation:

Infoset augmentation or infoset modification refers to the process of modifying the infoset during schema validation, for example by adding default attributes. The augmented infoset is called the post-schema-validation infoset, or PSVI. [4]

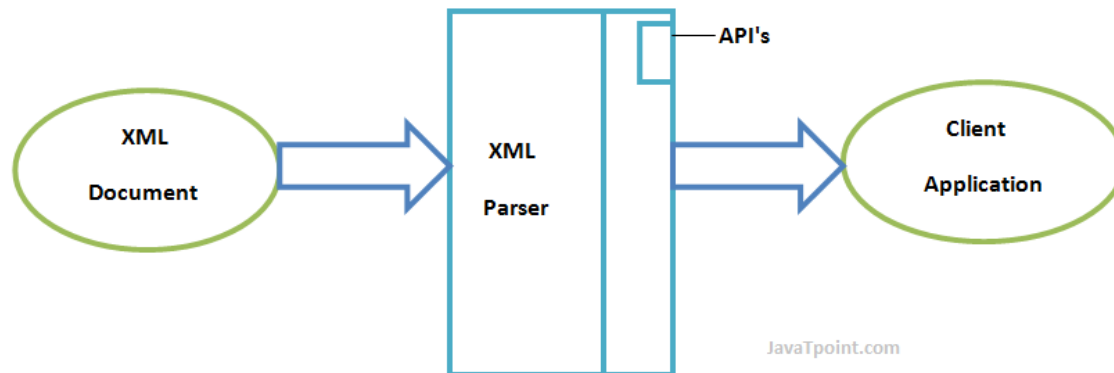
Infoset augmentation is somewhat controversial, with claims that it is a violation of modularity and tends to cause interoperability problems, since applications get different information depending on whether or not validation has been performed. [5]

XML Parsers

An XML parser is a software library or package that provides interfaces for client applications to work with an XML document. The XML Parser is designed to read the XML and create a way for programs to use XML.

XML parser validates the document and check that the document is well formatted.

Let's understand the working of XML parser by the figure given below:



Types of XML Parsers

These are the two main types of XML Parsers:

1. DOM
2. SAX

DOM (Document Object Model)

A DOM document is an object which contains all the information of an XML document. It is composed like a tree structure. The DOM Parser implements a DOM API. This API is very simple to use.

Features of DOM Parser

A DOM Parser creates an internal structure in memory which is a DOM document object and the client applications get information of the original XML document by invoking methods on this document object.

DOM Parser has a tree based structure.

Advantages

1) It supports both read and write operations and the API is very simple to use.

2) It is preferred when random access to widely separated parts of a document is required.

Disadvantages

1) It is memory inefficient. (consumes more memory because the whole XML document needs to loaded into memory).

2) It is comparatively slower than other parsers.

SAX (Simple API for XML)

A SAX Parser implements SAX API. This API is an event based API and less intuitive.

Features of SAX Parser

It does not create any internal structure.

Clients does not know what methods to call, they just overrides the methods of the API and place his own code inside method.

It is an event based parser, it works like an event handler in Java.

Advantages

1) It is simple and memory efficient.

2) It is very fast and works for huge documents.

Disadvantages

1) It is event-based so its API is less intuitive.

2) Clients never know the full information because the data is broken into pieces.

What is an XML Schema?

An XML Schema describes the structure of an XML document.

The XML Schema language is also referred to as XML Schema Definition (XSD).

XSD Example

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:element name="note">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="to" type="xs:string"/>
      <xs:element name="from" type="xs:string"/>
      <xs:element name="heading" type="xs:string"/>
      <xs:element name="body" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

</xs:schema>
```

The purpose of an XML Schema is to define the legal building blocks of an XML document:

- the elements and attributes that can appear in a document
- the number of (and order of) child elements
- data types for elements and attributes
- default and fixed values for elements and attributes

Why Learn XML Schema?

In the XML world, hundreds of standardized XML formats are in daily use.

Many of these XML standards are defined by XML Schemas.

XML Schema is an XML-based (and more powerful) alternative to DTD.

XML Schemas Support Data Types

One of the greatest strength of XML Schemas is the support for data types.

- It is easier to describe allowable document content
- It is easier to validate the correctness of data
- It is easier to define data facets (restrictions on data)
- It is easier to define data patterns (data formats)
- It is easier to convert data between different data types

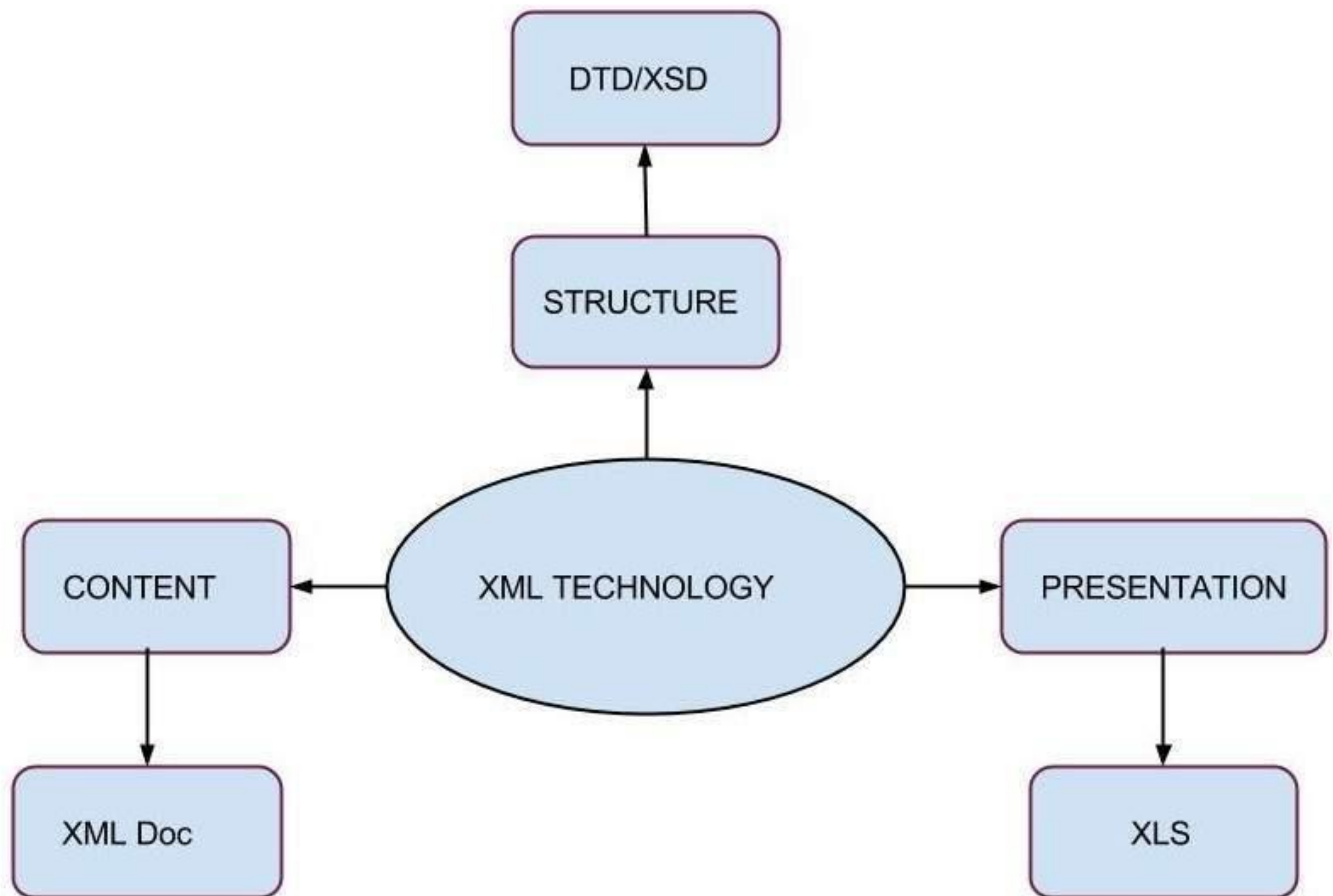
DTD - Overview

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

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**.

We will learn more about these in the chapter DTD Syntax

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.*
- It is not object oriented. Hence, the concept of inheritance cannot be applied on the DTDs.

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**.

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>Tanmay Patil</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>Tanmay Patil</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.*

XML components −

- Element
- Attributes
- Entities

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>. Elements, when used in an XML-DTD, need to be declared which will be discussed in detail in the chapter DTD Elements.

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*.

If attributes are used in an XML DTD then these need to be declared which will be discussed in detail in the chapter DTD Attributes

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: &
- Single quote: '
- Greater than: >
- Less than: <
- Double quote: "

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.

Element Content Types

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

- Empty content
- Element content
- Mixed content
- Any content

Empty Content

This is a special case of element declaration. This element declaration does not 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 />.

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>Tanmay Patil</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*.
*	<!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*.
?	<!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*.
,	<!ELEMENT element-name (child1, child2)>	It gives sequence of child elements separated by comma which must be included in the 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.
  The declaration indicates that the <address> element must have exactly three children - <name>, <company>, and <phone> - and that they must appear in this order. For example −

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

- **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>
```