

## **UNIT I - Introduction to Web**

**Introduction to Web: Internet and World Wide Web, Domain name service, Protocols: HTTP,FTP, SMTP. HTML5 concepts, CSS3, Anatomy of web page. XML: Document type Definition, XML schemas, Document object model, XSLT, DOM and SAX Approaches.**

**Introduction to Web: Internet and World Wide Web**

### **1.1 Introduction to Internet**

**The Internet is a global communication system that links together thousands of individual networks. It allows exchange of information between two or more computers on a network.**

**Thus the internet helps in transfer of messages through mail, chat, video & audio conference, etc.**

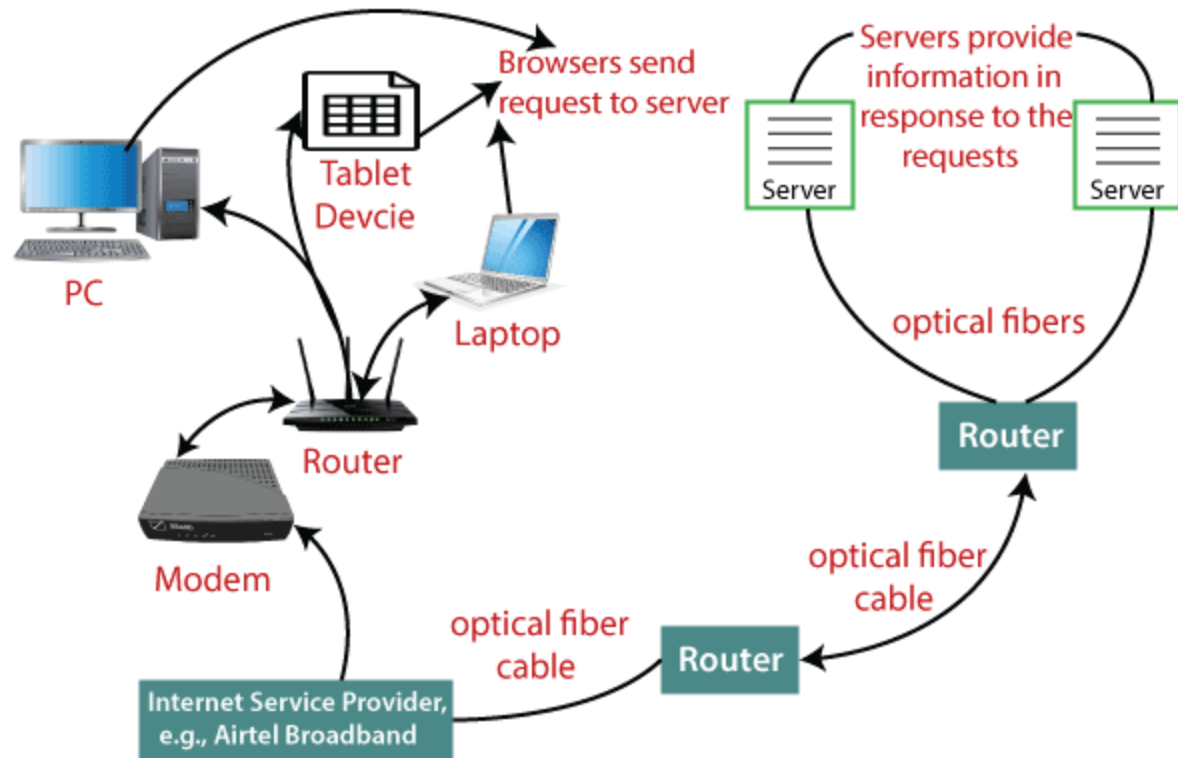
**It has become mandatory for day-to-day activities: bills payment, online shopping and surfing, tutoring, working, communicating with peers, etc.**

**Internet was evolved in 1969, under the project called ARPANET (Advanced Research Projects Agency Network) to connect computers at different universities and U.S. defence**

**It uses the standard internet protocol suite (TCP/IP) to connect billions of computer users worldwide.**

**It is set up by using cables such as optical fibers and other wireless and networking technologies.**

**The Internet is the fastest means of sending or exchanging information and data between computers across the world.**



## Advantages of the Internet:

- **Instant Messaging:** You can send messages or communicate to anyone using internet, such as email, voice chat, video conferencing, etc.
- **Get directions:** Using GPS technology, you can get directions to almost every place in a city, country, etc. You can find restaurants, malls, or any other service near your location.
- **Online Shopping:** It allows you to shop online such as you can buy clothes, shoes, book movie tickets, railway tickets, flight tickets, and more.
- **Pay Bills:** You can pay your bills online, such as electricity bills, gas bills, college fees, etc.
- **Online Banking:** It allows you to use internet banking in which you can check your balance, receive or transfer money, get a statement, request cheque-book, etc.
- **Online Selling:** You can sell your products or services online. It helps you reach more customers and thus increases your sales and profit.

- **Work from Home:** In case you need to work from home, you can do it using a system with internet access. Today, many companies allow their employees to work from home.
- **Entertainment:** You can listen to online music, watch videos or movies, play online games.
- **Cloud computing:** It enables you to connect your computers and internet-enabled devices to cloud services such as cloud storage, cloud computing, etc.
- **Career building:** You can search for jobs online on different job portals and send you CV through email if required.

## **1.2 World Wide Web(WWW)**

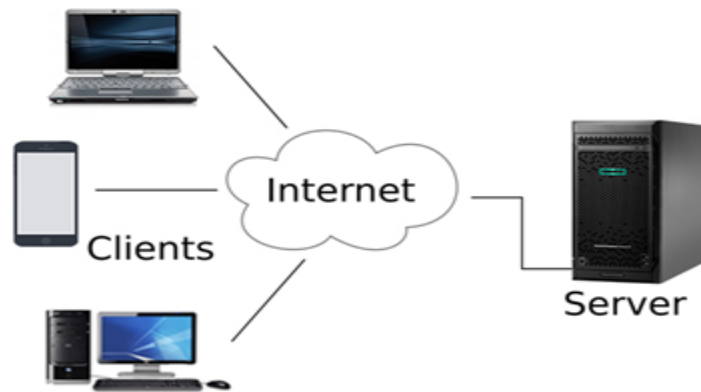
The World Wide Web(WWW) is an information sharing model that allows accessing information over the medium of the Internet.

It is the collection of electronic documents that are linked together. These electronic documents are known as 'Web Pages'. A collection of related WebPages is known as a 'Web Site'.

A Website resides on Server computers that are located around the world. Information on the WWW is always accessible, from anywhere in the world.

World Wide Web, which is also known as a Web, is a collection of websites or web pages stored in web servers and connected to local computers through the internet. These websites contain text pages, digital images, audios, videos, etc.

The basic architecture is characterized by a Web Browser that displays information content and Web Server that transfer's information to the client.



### Working of WWW:

The World Wide Web is based on several different technologies:

- ❖ Web browsers,
- ❖ Hypertext Markup Language (HTML),
- ❖ Uniform Resource Locator (URL)
- ❖ Hypertext Transfer Protocol (HTTP).

☐ A Web browser is used to access web pages. Web browsers can be defined as programs which display text, data, pictures, animation and video on the Internet.

☐ HTML: Hyper Text Markup Language for creating and editing document content.

☐ URL: Uniform Resource Locator for locating resources on the Internet.

☐ HTTP: HyperText Transfer Protocol to transfer the data.

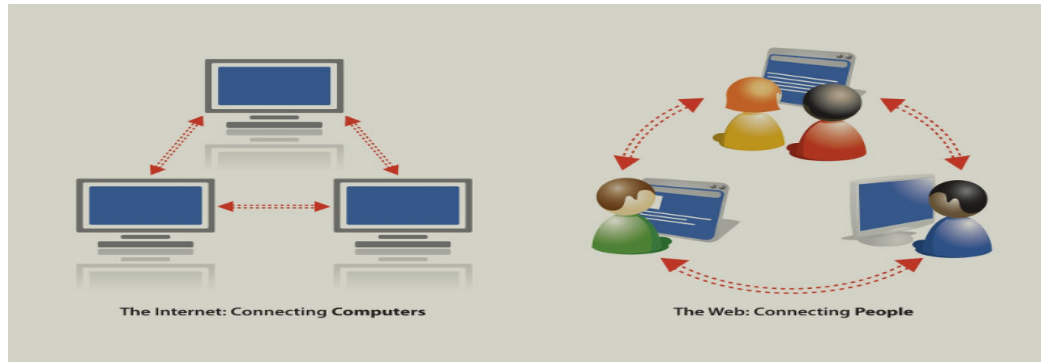
- The Web was invented in 1991 by Tim Berners-Lee, while consulting at CERN (European Organization for Nuclear Research) in Switzerland.
- The Web is a distributed information system.
- The Web contains multimedia.
- Information on the Web is connected by hyperlinks.

### Features of WWW:

- HyperText Information System.
- Cross-Platform.
- Distributed.
- Open Standards and Open Source.

- Uses Web Browsers to provide a single interface for many services.
- Dynamic, Interactive and Evolving.

### Difference between Internet and WWW



## INTERNET VERSUS WORLD WIDE WEB

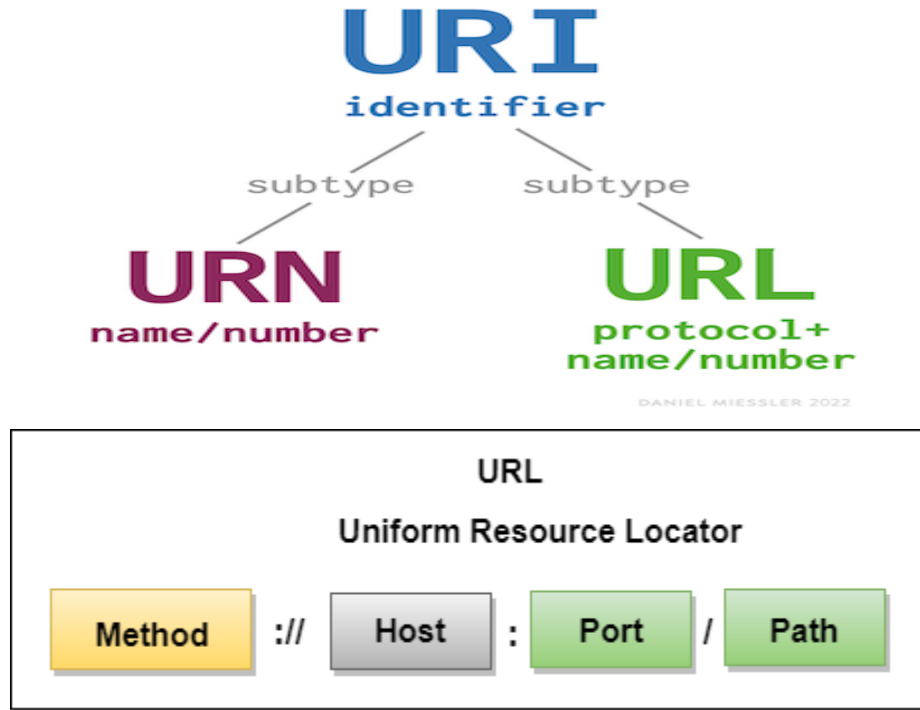
INTERNET	WORLD WIDE WEB
A global system of interconnected computer networks that use the TCP/IP protocol to link devices worldwide	Online content that is formatted in HTML and accessed via HTTP protocol
A massive interconnection of computer networks around the world	Service provided by the internet
Uses Transmission Control Protocol/Internet Protocol (TCP/IP)	Uses Hyper Text Transfer Protocol (HTTP)
	Visit <a href="http://www.PEDIAA.com">www.PEDIAA.com</a>

### Difference between Web 1.0 ,Web 2.0 & Web 3.0

Web 1.0	Web 2.0	Web 3.0
---------	---------	---------

<b>Created in 1989 by Tim Berners-Lee</b>	<b>Term coined by Tim O'Reilly in 2004</b>	<b>Modern usage with blockchain defined by Gavin Wood, co-founder of <u>Ethereum</u>, in 2014.</b>
<b>Static website content</b>	<b>Dynamic content and user input</b>	<b>Semantic content that can benefit from AI</b>
<b>Information delivery</b>	<b>Social networks</b>	<b>Metaverse worlds</b>
<b>Centralized infrastructure</b>	<b>Cloud utility infrastructure that is still largely centralized</b>	<b>Decentralized, edge computing and peer-to-peer</b>
<b>Relational database-driven content and application delivery</b>		<b>Blockchain-based distributed services</b>

## **Difference Between URL, URI and URN**



**URI stands for Uniform Resource Identifier.**

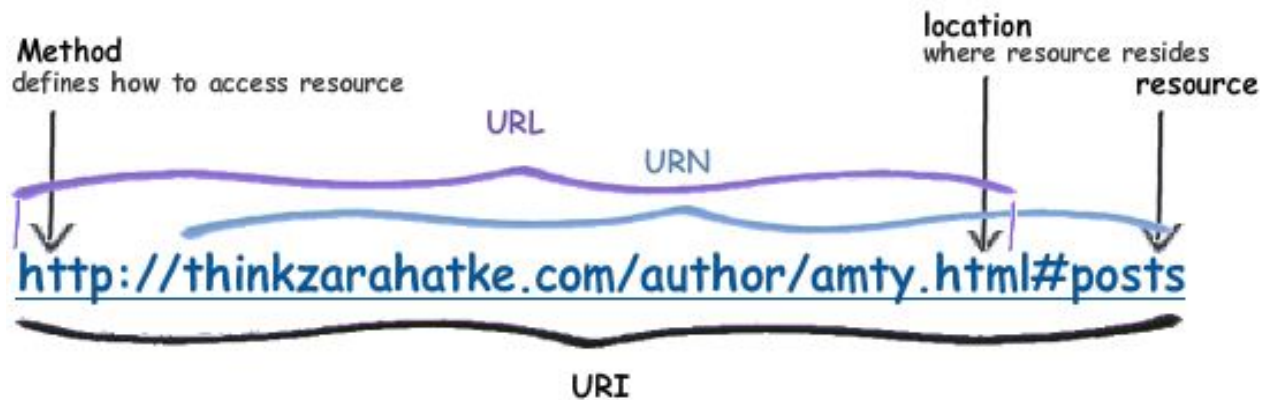
- URI is a sequence of characters used to identify resource location or a name or both over the World Wide Web.
- A URI can be further classified as a locator, a name, or both.
- Syntax of URI: Starts with a scheme followed by a colon character, and then by a scheme-specific part.
- The most popular URI schemes are HTTP, HTTPS, and FTP.

**URL stands for Uniform Resource Location.**

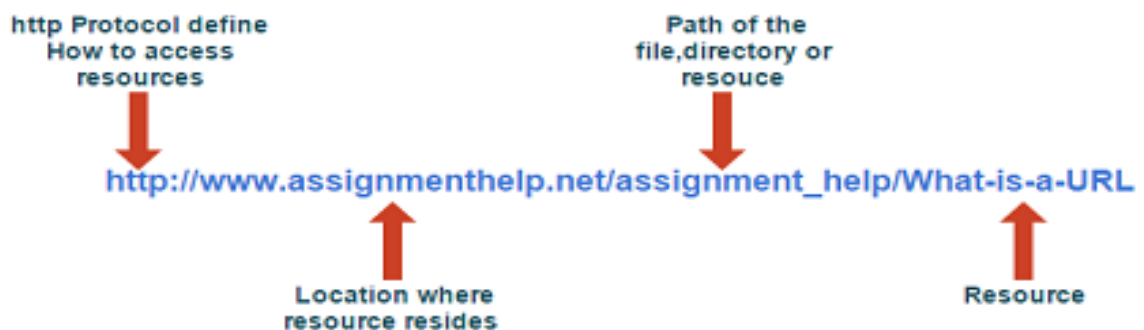
- URL is a subset of URI that describes the network address or location where the source is available.
- URL begins with the name of the protocol to be used for accessing the resource and then a specific resource location.

## URN stands for Uniform Resource Name.

It is a URI that uses a URN scheme. “urn” scheme: It is followed by a namespace identifier, followed by a colon, followed by namespace specific string



## Difference between URI, URL and URN



**URI:** `http://www.assignmenthelp.net/assignment_help/What-is-a-URL`

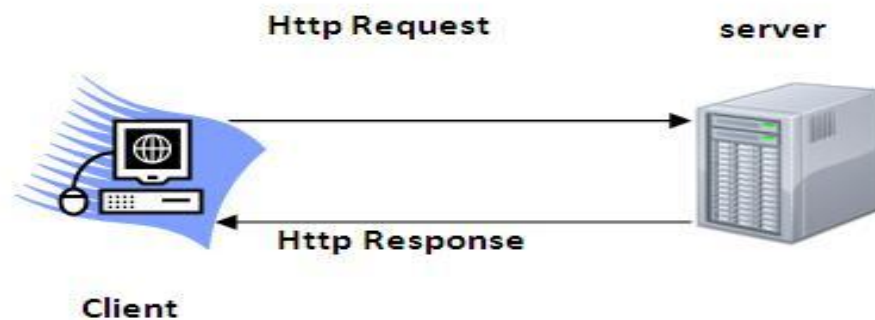
**URL:** `http://www.assignmenthelp.net/assignment_help`

**URN:** `www.assignmenthelp.net/assignment_help/What-is-a-URL`



# HTTP (Hyper Text Transfer Protocol)

- The Hypertext Transfer Protocol (HTTP) is application-level protocol for collaborative, distributed, hypermedia information systems.
- It is the data communication protocol used to establish communication between client and server.
- It is a protocol used to access the data on the World Wide Web (www).
- The HTTP protocol can be used to transfer the data in the form of plain text, hypertext, audio, video, and so on.
- HTTP is used to carry the data in the form of MIME-like format.(Multipurpose Internet Mail Extension)
- It is the data communication protocol used to establish communication between client and server.



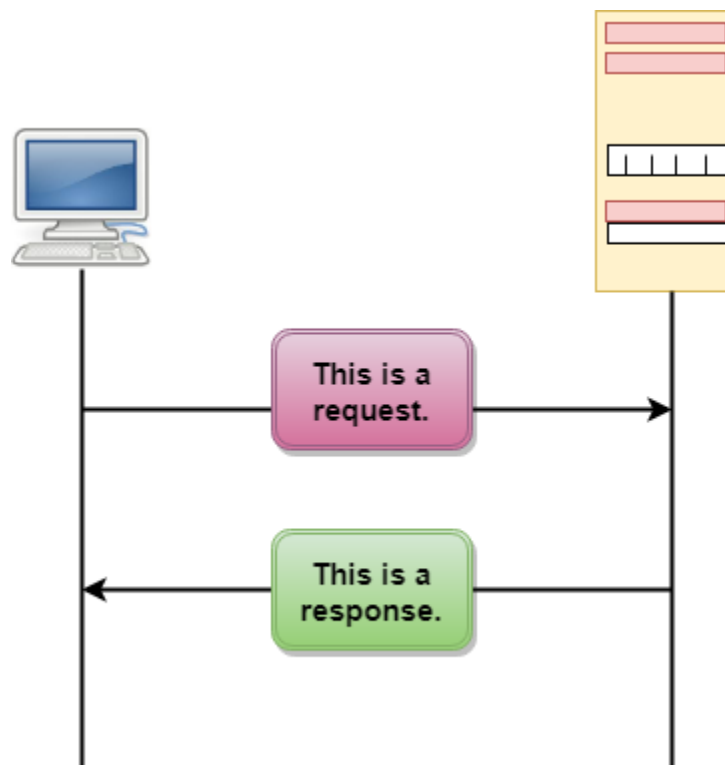
## The Basic Characteristics of HTTP

- ❖ It is the protocol that allows web servers and browsers to exchange data over the web.
- ❖ It is a request response protocol.
- ❖ It uses the reliable TCP connections by default on TCP port 80.
- ❖ It is stateless means each request is considered as the new request.

## Basic Features of HTTP

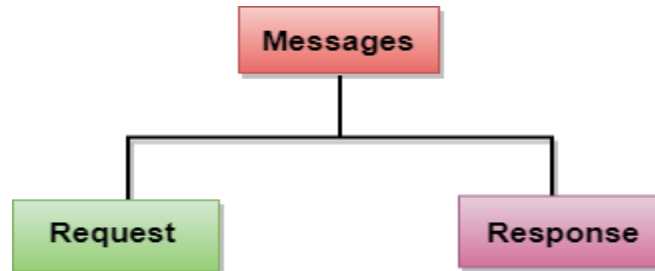
- ❖ **HTTP is media independent:** It specifies that any type of media content can be sent by HTTP as long as both the server and the client can handle the data content.

- ❖ **HTTP is connectionless:** The HTTP client, i.e., a browser initiates an HTTP request and after a request is made, the client waits for the response. The server processes the request and sends a response back after which client disconnect the connection. So client and server knows about each other during current request and response only.
- ❖ **HTTP is stateless:** The client and server are aware of each other during a current request only. Afterwards, both of them forget each other. Due to the stateless nature of protocol, neither the client nor the server can retain the information about different request across the web pages.

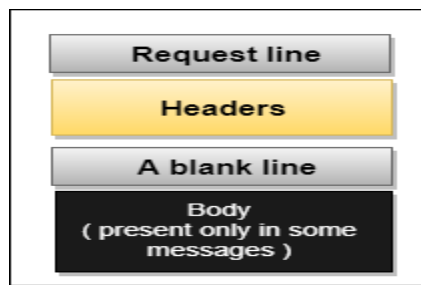


# Messages

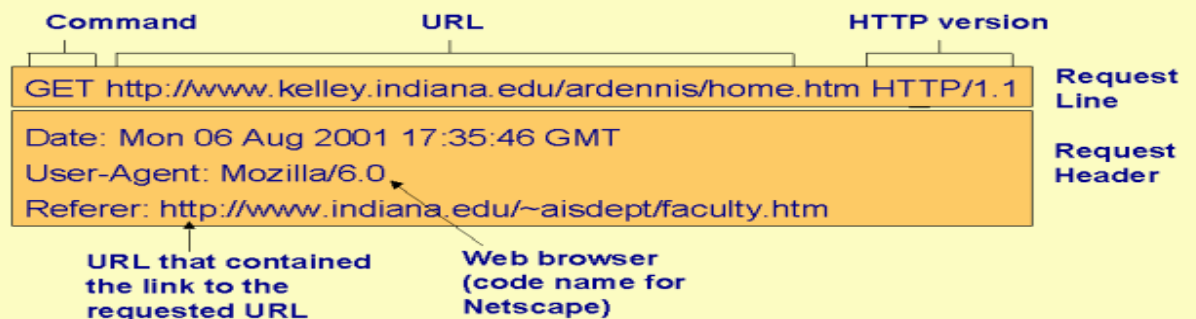
HTTP messages are of two types: request and response.



**Request Message:** The request message is sent by the client that consists of a request line, headers, and body.



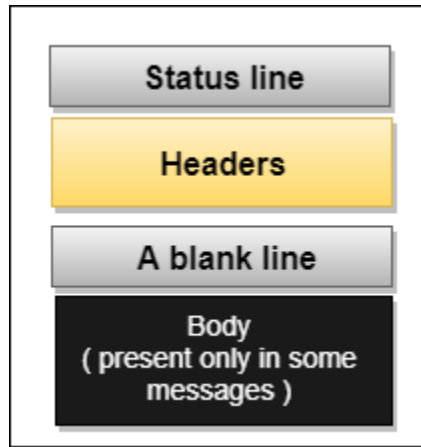
## Example of an HTTP Request



**Note that this HTTP Request message has no "Body" part.**

GMT – Greenwich Mean Time

**Response Message:** The response message is sent by the server to the client that consists of a status line, headers, and sometimes a body.



HTTP/1.1 200 OK

Date: Sun, 28 Aug 2017 08:56:53 GMT

Server: Apache/2.4.27 (Linux)

Last-Modified: Fri, 20 Jan 2017 07:16:26 GMT

ETag: "10000000565a5-2c-3e94b66c2e680"

Accept-Ranges: bytes

Content-Length: 44

Connection: close

Content-Type: text/html

X-Pad: avoid browser bug

<html><body><h1>It works!</h1></body></html>

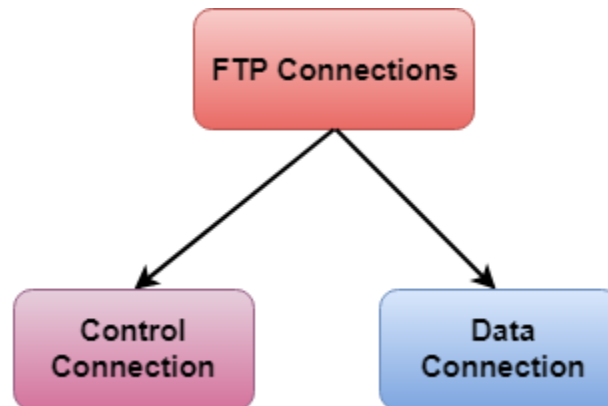
# FTP(File Transfer Protocol)

- **FTP stands for File transfer protocol.**
- **File Transfer Protocol (FTP) is an application layer protocol that is used to transfer the files between the local devices (PC, smartphone, etc.) to a server. It transfers both text and binary files over the Internet.**
- **FTP is a standard internet protocol provided by TCP/IP used for transmitting the files from one host to another.**
- **The first feature of FTP was developed by **Abhay Bhushan** in 1971.**
- **It is mainly used for transferring the web page files on the internet.**
- **It is also used for downloading the files to the computer from other servers.**

## Objectives of FTP

- **It provides the sharing of files.**
- **It is used to connect remote computers.**
- **It transfers the data more reliably and efficiently.**

FTP opens two connections between the computers



One connection is used for data connection, and another connection is used for the control connection.

### Control Connection

- ❖ A Control Connection is established on **Port number 21**. It is the primary connection and is used to send commands back and forth between the client and the server.
- ❖ It is used for sending the control information like user identification, password, and remote directory, etc., once the control connection is established.

## **Data Connection**

**Data Connection is initiated on Port number 20.**

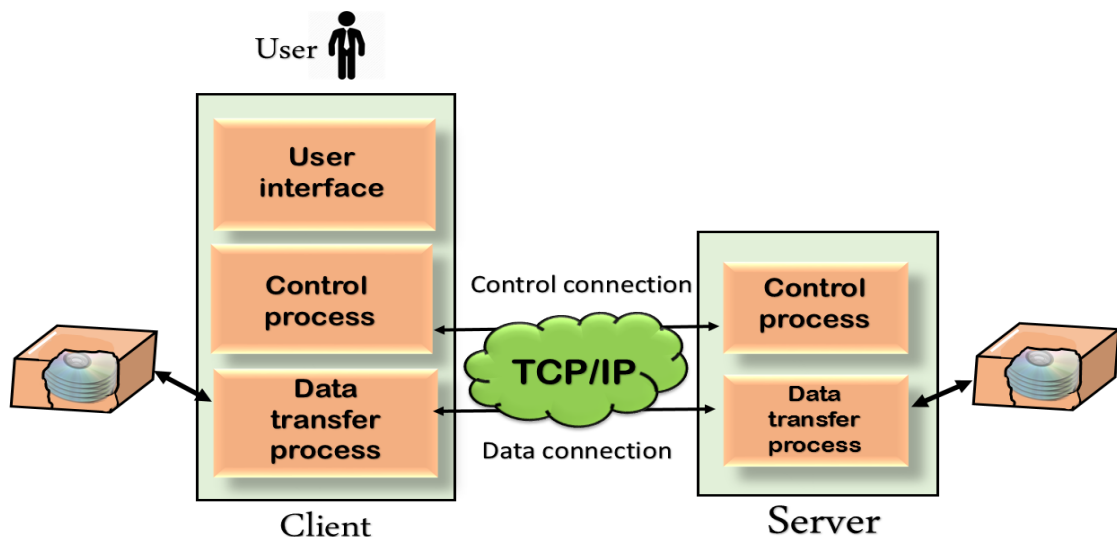
**Using the established Control Connection, the client and server will create a separate Data Connection to transfer the requested data.**

**The Data Connection stays open until the transfer is complete  
Data Connections are closed by either the client or the server, depending on which party is sending the information.**

**When a client is retrieving data from a server, the server will close the connection once all the data has been transferred.**

**When the client is transferring data to the server, the client will terminate the connection after all the data has been transferred.**

## Mechanism of FTP



- The FTP client has three components: **The user interface, control process, and data transfer process.**

**The server has two components: the server control process and the server data transfer process.**

FTP transfers files in three different modes –

- **Stream mode** – Here, the FTP handles the data as a string of bytes without separating boundaries.
- **Block mode** – In the block mode, the FTP decomposes the entire data into different blocks of data.
- **Compressed mode** – In this mode, the FTP uses the Lempel-Ziv algorithm to compress the data.



# SMTP

- **SMTP stands for Simple Mail Transfer Protocol.**
- **SMTP is a set of communication guidelines that allow software to transmit an electronic mail over the internet is called Simple Mail Transfer Protocol.**
- **SMTP (Simple Mail Transfer Protocol) is a **TCP/IP** protocol used in sending and receiving e-mail.**
- **It is a program used for sending messages to other computer users based on e-mail addresses.**
- **It can send a single message to one or more recipients.**
- **Sending message can include text, voice, video or graphics.**
- **The main purpose of SMTP is used to set up communication rules between servers.**

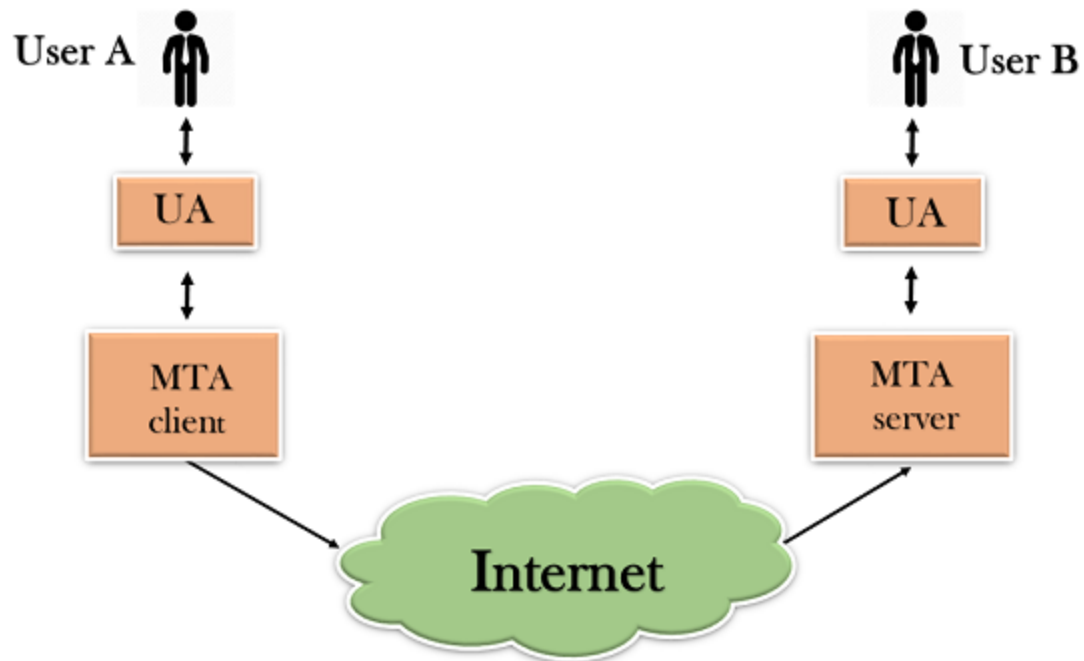
## Components of SMTP



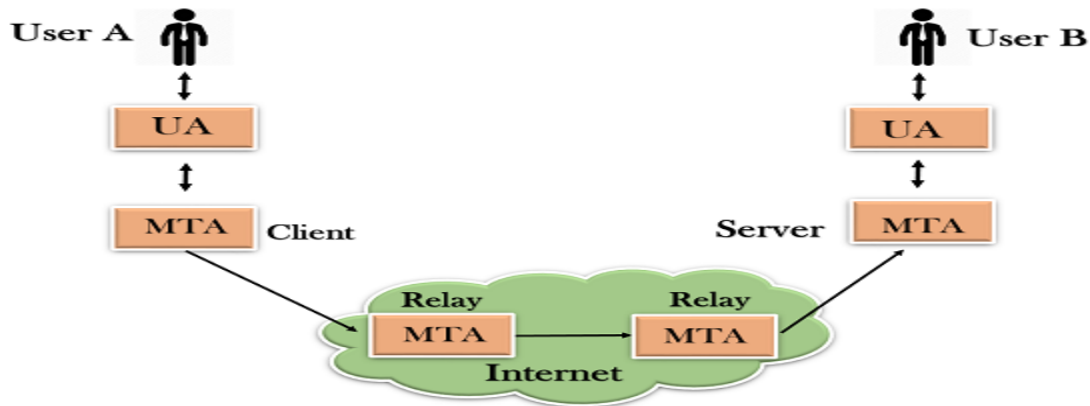
Two components such as user **agent (UA)** and **mail transfer agent (MTA)**.

The user agent (UA) prepares the message, creates the envelope and then puts the message in the envelope.

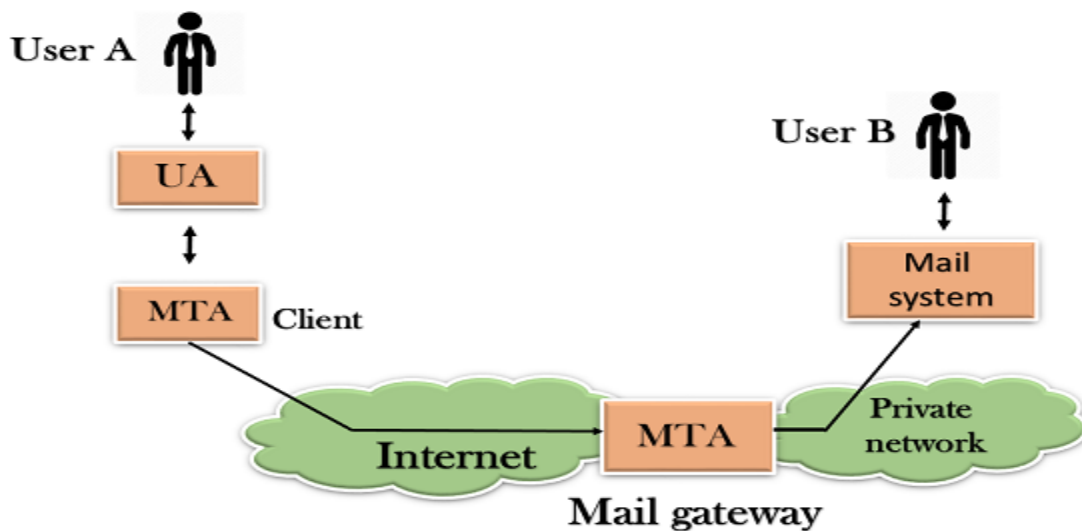
The mail transfer agent (MTA) transfers this mail across the internet.



**SMTP allows a more complex system by adding a relaying system. Instead of just having one MTA at sending side and one at the receiving side, more MTAs can be added, acting either as a client or server to relay the email.**



The relaying system without TCP/IP protocol can also be used to send the emails to users, and this is achieved by the use of the mail gateway. The mail gateway is a relay MTA that can be used to receive an email.





**DTD**

- Document Type Definition
- The DTD defines the structure of the xml document and how content is nested.
- An XML document is valid only when it is well-formed and confirms to the DTD (or XML schema) defined for it.
- DTD defines the grammar rules for forming an XML document.

# Ways of linking DTD with XML

- Internal
  - Including DTD in the same file as XML file
- External
  - Creating another file for DTD and linking it with XML file
- If both are provided, then if there are similar declarations, internal DTD takes preference.



# Linking external DTD with XML

- 2 ways to associate DTD with XML
  - **<!DOCTYPE root SYSTEM location>**
    - SYSTEM is used to explicitly specify the location of the DTD.
    - Example: **<!DOCTYPE note SYSTEM "file:///c:/x.dtd" >**
  - **<!DOCTYPE root PUBLIC identifier location>**
    - PUBLIC is used if the DTD is a standard and is shared by many organizations

# Linking external DTD with XML

- The identifier is a name mapped to the actual location of DTD from where everybody shares the dtd.
- If the dtd is not accessible or available then like SYSTEM command dtd is obtained from location.
- Example
- `<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN" "http://java.sun.com/dtd/web-app_2_3.dtd">`

# Basic DTD declarations

- `<!ELEMENT ...>`
- `<!ATTLIST ...>`
- `<!ENTITY ...>`
- `<!NOTATION ...>`

# ELEMENT Declaration

- **Text Only:**
  - Specifies that this element can contain content that is text
  - `<!ELEMENT name (#PCDATA)>`
  - Example: `<!ELEMENT fname (#PCDATA)>`
  - VALID XML : `<fname>ravi</fname>`
  - INVALID XML:  
`<fname><nickname>ravi</nickname></fname>`

# ELEMENT Declaration

- **Element Only:**
  - Specifies that this element can contain elements as specified by the tag
  - `<!ELEMENT name (child1) >`
  - Example: `<!ELEMENT custname (fname)>`
  - VALID XML :  
`<custname><fname>ravi</fname>`  
`</custname>`
  - INVALID XML: `<custname>ravinath</custname>`

# Order of elements

- **Sequence list:**
  - ‘,’ separated list
  - The child elements must appear in the specified order
  - Example: `<!ELEMENT custname (fname,lname)>`
  - VALID XML :  
`<custname><fname>ravi</fname>  
<lname>nath</lname></custname>`
  - INVALID XML: `<custname><lname>ravi</lname>  
<fname>nath</fname></custname>`

# Order of elements

- **Choice list:**
  - ' | ' separated list
  - The child elements can appear any order
  - Example:
    - `<!ELEMENT custname (fname | lname)>`
  - VALID XML :  
`<custname><lname>ravi</lname>  
<fname>nath</fname></custname>`
  - INVALID XML:  
`<custname><lname>ravi</lname>  
</custname>`

# Element Declaration

## 3. Mixed content:

- Specifies that this element can contain mixture of elements and text as specified.
- Mixed content must have
  - #PCDATA declared as first item
  - Items separated by |
  - content model must be defined as zero or more('\*').
- Example: `<!ELEMENT name #PCDATA|child1|child2)>`



```
<?xml version="1.0" ?>
<!DOCTYPE person[
<!ELEMENT person (address)+>
<!ELEMENT address (#PCDATA|city|state)*>
<!ELEMENT city (#PCDATA)>
<!ELEMENT state (#PCDATA)> ]>
<person>
  <address>Mr. Mohan Lal
    172 Veera Apts., MG Road
    <city>Cochin</city>
    <state>Kerala</state>
  </address>
</person>
```

# Element Declaration

## 4. Anything:

- Specifies that this element can contain any well-formed xml data
- `<!ELEMENT name ANY>`
- Example: `<!ELEMENT address ANY>`
- Valid XML:  

```
<address>  
    Mr. Mohan Lal  
    172 Veera Apts., MG Road  
    <city>Cochin</city>  
    <state>Kerala</state>  
  </address>  
</address>
```

# Element Declaration

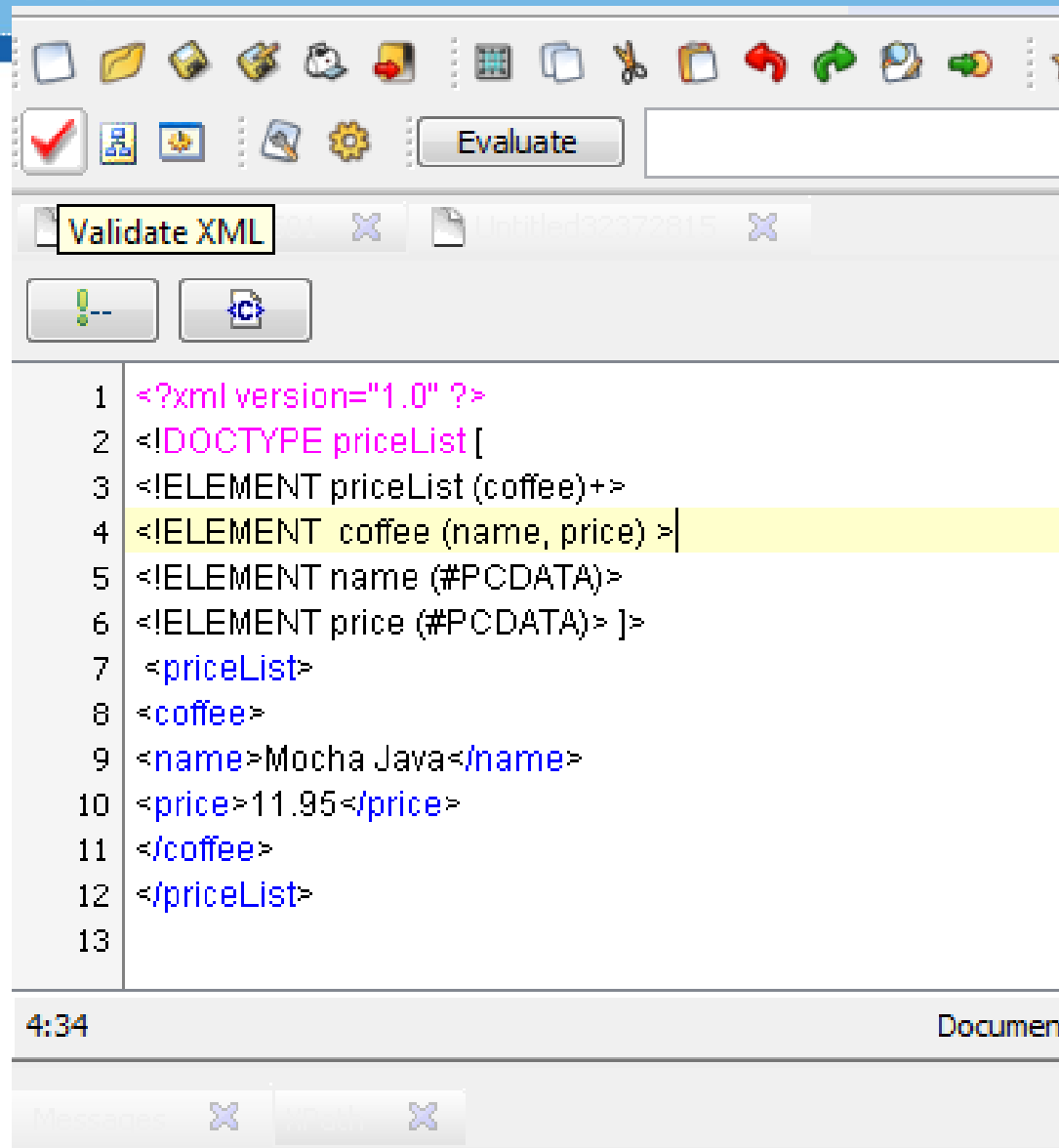
## 5. Empty:

- Specifies that the element will be empty. That is it cannot contain anything
- `<!ELEMENT name EMPTY>`
- Example: `<!ELEMENT leftmargin EMPTY>`
- Valid XML:  
`<leftmargin/>`  
`<leftmargin></leftmargin>`
- Invalid XML:  
`<leftmargin>xxx</leftmargin>`

# Element Declaration

- **Cardinality**
  - none: the absence of cardinality indicates one and only one
  - \* → 0 or more
  - + → 1 or more
  - ? → 0 or 1
- Example:  
`<!ELEMENT letter (to+,subject?,body,from)>`

# Example: Embedded DTD with XML



Document is valid!

# Attribute Declaration

- **<!ATTLIST elementName**  
    **attrName type attDefault value >**
- **elementName** , **attrName** are compulsory
- **type** specifies the type of the value the attribute can hold
- **attDefault** specifies whether an attributes presence is required or not. Also says how the parser must handle the attributes absence.
- Value specifies the default value

# Attribute Types

- **CDATA**: text data (string)
- **ID**: valid xml name which is unique for each identifier for each instance of the current element.
- **IDREF**: a reference to the ID type
- **IDREFS**: List of **IDREFs** separated by comma
- **NMTOKEN**: text data that can contain the characters limited to letters, digits, underscores, colons, periods and dashes
- **NMTOKENS**: a comma-separated list of **NMTOKEN** items

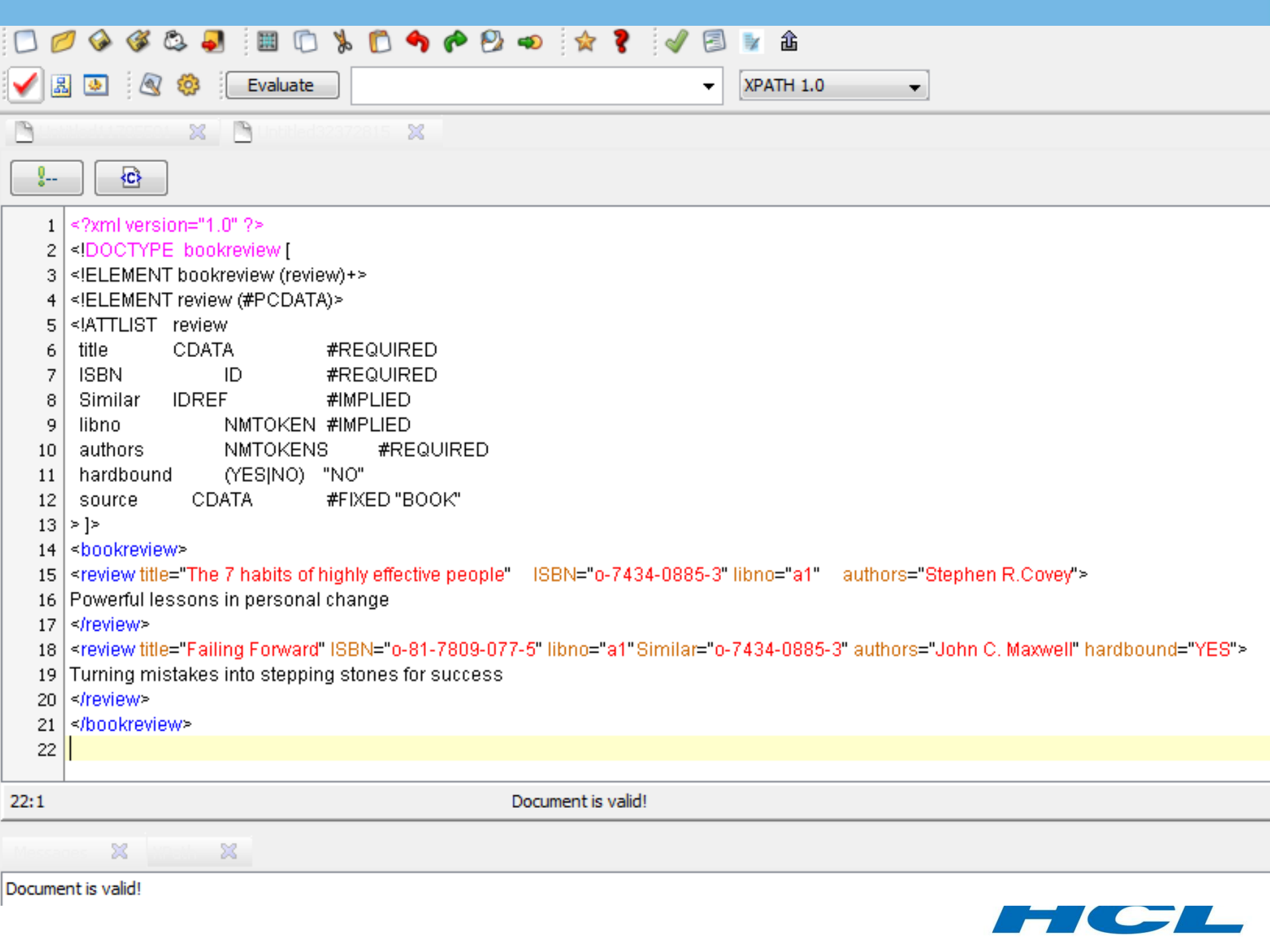
# Attribute Types

- **ENTITY**: name of the predefined entity.
- **ENTITIES**: a list of ENTITY names separated by white space chars
- **NOTATION**: used to map a reference to a *notation-type* declaration section that is declared else where in the DTD.
- Enumerated value/attribute choice list: a list of values that attribute value can have



# Attributes Default

- **#REQUIRED**: the attribute is required.
- **#IMPLIED**: the attribute is optional.
- **#FIXED *with default value***: the attribute must always have the default value
- **Default value**: the attribute is optional. If it is present, then it should be of the attribute type. If it is not present then parser will take the default value.



```

1 <?xml version="1.0" ?>
2 <!DOCTYPE bookreview [
3 <!ELEMENT bookreview (review)+>
4 <!ELEMENT review (#PCDATA)>
5 <!ATTLIST review
6 title      CDATA          #REQUIRED
7 ISBN       ID             #REQUIRED
8 Similar    IDREF          #IMPLIED
9 libno      NMTOKEN        #IMPLIED
10 authors    NMTOKENS       #REQUIRED
11 hardbound  (YES|NO)       "NO"
12 source     CDATA          #FIXED "BOOK"
13 ]>
14 <bookreview>
15 <review title="The 7 habits of highly effective people" ISBN="0-7434-0885-3" libno="a1" authors="Stephen R.Covey">
16 Powerful lessons in personal change
17 </review>
18 <review title="Failing Forward" ISBN="0-81-7809-077-5" libno="a1" Similar="0-7434-0885-3" authors="John C. Maxwell" hardbound="YES">
19 Turning mistakes into stepping stones for success
20 </review>
21 </bookreview>
22

```

22:1

Document is valid!

Document is valid!

# Notation

- Notation is used to map a reference to a notation-type declaration section that is declared else where in the DTD.
- The notation is more important for the application outside the parser.
- Example:

```
<!NOTATION gif SYSTEM  
  'http://mysite.com/GIF_viewer.exe'>  
<!ATTLIST PIC viewer NOTATION (gif) #IMPLIED>  

```

# Entities

- Entity/Entities are replaceable contents which helps reduced re-type or reassign of the same content
- All the entities except predefined entities need to be declared.
- Entities can be classified in two ways:
  - Depending on where it is used
  - Depending on how it is parsed

# Classification 1

- Two types of Entities:
- **Parameter Entity**: Entity reference within used within the DTD.
- **General Entity**: Entity reference within used within the XML document.
- It is an error to put a parameter reference in the xml document. But it is not an error to put an entity reference in DTD *in defining the value of another entity*. But the reference will not be 'resolved' until it is used in the document.

# Entity declaration

- General Entity

- Declaration:

- `<!ENTITY GenEntity "sometext">`

- Referencing in XML or DTD:

- `&GenEntity;`

- Parameter Entity

- Declaration:

- `<!ENTITY % ParEntity "sometext">`

- Referencing DTD:

- `%ParEntity;`

# Example- Entity declaration

## DTD

cprice.dtd

```
<!ELEMENT priceList (coffee)>
<!ENTITY dcode "01">
<!ENTITY % any "(#PCDATA)">
<!ELEMENT coffee (name, price)>
<!ELEMENT name %any; >
<!ATTLIST name Code (01|02) "&dcode;">
<!ELEMENT price %any;>
<!ATTLIST price Val CDATA #REQUIRED>
```

```
<!DOCTYPE priceList SYSTEM  
"cprice.dtd">  
<priceList>  
  <coffee>  
    <name>&dcode ;</name>  
    <price Val="11.95">Rs.11.95</price>  
  </coffee>  
</priceList>
```



# Another classification

- Parsed Entities: well-formed content which is parsed
- The example in the previous slide is an example for the parsed entity.
- Unparsed Entities: non-XML data
- Unparsed entities depend on the notation declaration to identify them so that the application processing the XML document knows what kind of entity is being used and what to do with it.

# Example: Parsed Entity

In DTD

```
<!ENTITY rights "All rights reserved">
```

```
<!ENTITY book "Designed by WW. &rights; ">
```

In XML:

```
&book;
```

# Example: Unparsed Entity

In DTD:

```
<!NOTATION gif SYSTEM
    'http://mysite.com/GIF_viewer.exe'>

<!ENTITY picref SYSTEM 'http://mysite/img.gif' NDATA
    gif>

<!ENTITY poster EMPTY>

<!ATTLIST poster src ENTITY #IMPLIED>
```

In XML:

```
<poster src="picref"/>
```

**XML**

---

# XML Overview

- XML stands for **E**xtensible **M**arkup **L**anguage. It is a text-based markup language derived from Standard Generalized Markup Language (SGML).
- XML tags identify the data and are used to store and organize the data, rather than specifying how to display it like HTML tags, which are used to display the data.
- XML is not going to replace HTML in the near future, but it introduces new possibilities by adopting many successful features of HTML.

# Characteristics of XML

- **XML is extensible:** XML allows you to create your own self-descriptive tags, or language, that suits your application.
- **XML carries the data, does not present it:** XML allows you to store the data irrespective of how it will be presented.
- **XML is a public standard:** XML was developed by an organization called the World Wide Web Consortium (W3C) and is available as an open standard

# XML Usage

- XML can work behind the scene to simplify the creation of HTML documents for large web sites.
- XML can be used to exchange the information between organizations and systems.
- XML can be used for offloading and reloading of databases.
- XML can be used to store and arrange the data, which can customize your data handling needs.
- XML can easily be merged with style sheets to create almost any desired output.
- Virtually, any type of data can be expressed as an XML document

# What is Markup?

- XML is a markup language that defines set of rules for encoding documents in a format that is both human-readable and machine-readable.
- *So what exactly is a markup language?*
- Markup is information added to a document that enhances its meaning in certain ways, in that it identifies the parts and how they relate to each other. More specifically, a markup language is a set of symbols that can be placed in the text of a document to demarcate and label the parts of that document.



# Example

Following example shows how XML markup looks, when embedded in a piece of text:

```
<message>  
  <text>Hello, world!</text>  
</message>
```

This snippet includes the markup symbols, or the tags such as `<message>...</message>` and `<text>...</text>`. The tags `<message>` and `</message>` mark the start and the end of the XML code fragment. The tags `<text>` and `</text>` surround the text Hello, world!.

# XML Syntax

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

**<contact-info>**

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

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

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

**</contact-info>**

Two kinds of information in the above example:

- The markup, like *<contact-info>* and
- The text, or the character data, *Tutorials Point* and *(040) 123-4567*.

# XML Declaration

The XML document can optionally have an XML declaration. It is written as below:

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

Where *version* is the XML version and *encoding* specifies the character encoding used in the document.

# Syntax Rules for XML declaration

- The XML declaration is case sensitive and must begin with "<?xml>" where "xml" is written in lower-case.
- If document contains XML declaration, then it strictly needs to be the first statement of the XML document.
- The XML declaration strictly needs be the first statement in the XML document.
- HTTP protocol can override the value of *encoding* that you put in the XML declaration.

# Tags and Elements

- An XML file is structured by several XML-elements, also called XML-nodes or XMLtags.
- XML-elements' names are enclosed by triangular brackets < > as shown below:

`<element>`

## Syntax Rules for Tags and Elements

**Element Syntax:** Each XML-element needs to be closed either with start or with end elements as shown below:

`<element>...</element>`

or in simple-cases, just this way:

`<element/>`

# Nesting of elements

- An XML-element can contain multiple XML-elements as its children, but the children elements must not overlap. i.e., an end tag of an element must have the same name as that of the most recent unmatched start tag.

Following example shows incorrect nested tags:

```
<?xml version="1.0"?>  
<contact-info>  
  <company>TutorialsPoint  
  <contact-info>  
  </company>
```

Following example shows correct nested tags:

```
<?xml version="1.0"?>  
<contact-info>  
  <company>TutorialsPoint</company>  
</contact-info>
```

# Root element

- An XML document can have only one root element. For example, following is not a correct XML document, because both the x and y elements occur at the top level without a root element:

```
<x>...</x>
```

```
<y>...</y>
```

- The following example shows a correctly formed XML document:

```
<root>
```

```
<x>...</x>
```

```
<y>...</y>
```

```
</root>
```

- **Case sensitivity:** The names of XML-elements are case-sensitive. That means the name of the start and the end elements need to be exactly in the same case.

For example, **<contact-info>** is different from **<Contact-Info>**.

# Attributes

An **attribute** specifies a single property for the element, using a name/value pair. An XML-element can have one or more attributes. For example:

```
<a href="http://www.tutorialspoint.com/">Tutorialspoint!</a>
```

Here, *href* is the attribute name and *http://www.tutorialspoint.com/* is attribute value.

## Syntax Rules for XML Attributes

- Attribute names in XML (unlike HTML) are case sensitive. That is, *HREF* and *href* are considered two different XML attributes.
- Same attribute cannot have two values in a syntax. The following example shows incorrect syntax because the attribute *b* is specified twice:

```
<a b="x" c="y" b="z">....</a>
```

- Attribute names are defined without quotation marks, whereas attribute values must always appear in quotation marks. Following example demonstrates incorrect xml

syntax:

```
<a b=x>....</a>
```



# XML References

- *References* usually allow you to add or include additional text or markup in an XML document. References always begin with the symbol "&", which is a reserved character and end with the symbol ";". XML has two types of references:
- **Entity References:** An entity reference contains a name between the start and the end delimiters. For example **&amp;**; where *amp* is *name*. The *name* refers to a predefined string of text and/or markup.
- **Character References:** These contain references, such as **&#65;**, contains a hash mark (“#”) followed by a number. The number always refers to the Unicode code of a character. In this case, 65 refers to alphabet "A".

# XML Text

- The names of XML-elements and XML-attributes are case-sensitive, which means the name of start and end elements need to be written in the same case.
- To avoid character encoding problems, all XML files should be saved as Unicode UTF- 8 or UTF-16 files.
- Whitespace characters like blanks, tabs and line-breaks between XML-elements and between the XML-attributes will be ignored.
- Some characters are reserved by the XML syntax itself. Hence, they cannot be used directly. To use them, some replacement-entities are used, which are listed below

# Contd..

not allowed character	replacement-entity	character description
<	&lt;	less than
>	&gt;	greater than
&	&amp;	ampersand
'	&apos;	apostrophe
"	&quot;	quotation mark

# XML Documents

An XML *document* is a basic unit of XML information composed of elements and other markup in an orderly package. An XML *document* can contains wide variety of data.

For example, database of numbers, numbers representing molecular structure or a mathematical equation.

## **XML Document example**

:

```
<?xml version="1.0"?>
<contact-info>
  <name>Tanmay Patil</name>
  <company>TutorialsPoint</company>
  <phone>(011) 123-4567</phone>
</contact-info>
```

# Contd..

The following image depicts Document Prolog Section



The **document prolog** comes at the top of the document, before the root element.

This section contains:

- XML declaration
- Document type declaration

# Document Elements Section

**Document Elements** are the building blocks of XML. These divide the document into a hierarchy of sections, each serving a specific purpose. You can separate a document into multiple sections so that they can be rendered differently, or used by a search engine. The elements can be containers, with a combination of text and other elements.

# XML Declaration

XML declaration contains details that prepare an XML processor to parse the XML document. It is optional, but when it is

Following syntax shows XML declaration:

```
<?xml
```

```
version="version_number"
```

```
encoding="encoding_declaration"
```

```
standalone="standalone_status"
```

```
?
```

# Contd..

Each parameter consists of a parameter name, an equals sign (=), and parameter value inside a quote. Following table shows the above syntax in detail:

Parameter	Parameter_value	Parameter_description
Version	1.0	Specifies the version of the XML standard used.
Encoding	UTF-8, UTF-16, ISO-10646-UCS-2, ISO-10646-UCS-4, ISO-8859-1 to ISO-8859-9, ISO-2022-JP, Shift JIS, EUC-JP	It defines the character encoding used in the document. UTF-8 is the default encoding used.
Standalone	yes or no.	It informs the parser whether the document relies on the information from an external source, such as external document type definition (DTD), for its content. The default value is set to <i>no</i> . Setting it to <i>yes</i> tells the processor there are no external declarations required for parsing the document.



# Rules

- An XML declaration should abide with the following rules:
- If the XML declaration is present in the XML, it must be placed as the first line in the XML document.
- If the XML declaration is included, it must contain version number attribute.
- The Parameter names and values are case-sensitive.
- The names are always in lower case.
- The order of placing the parameters is important. The correct order is: *version, encoding and standalone*.
- Either single or double quotes may be used.
- The XML declaration has no closing tag i.e. `</?xml>`

# XML Declaration Examples

- XML declaration with no parameters:

```
<?xml >
```

- XML declaration with version definition:

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

- XML declaration with all parameters defined:

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

- XML declaration with all parameters defined in single quotes:

```
<?xml version='1.0' encoding='iso-8859-1'  
standalone='no' ?>
```

# XML Tags

XML tags form the foundation of XML. They define the scope of an element in the XML. They can also be used to insert comments, declare settings required for parsing the environment and to insert special instructions.

## Start Tag

- The beginning of every non-empty XML element is marked by a start-tag. An example of start-tag is:

`<address>`

## End Tag

Every element that has a start tag should end with an end-tag. An example of endtag is:

`</address>`

- Note that the end tags include a solidus ("/") before the name of an element.

# Contd..

## Empty Tag

The text that appears between start-tag and end-tag is called *content*. An element which has no content is termed as **empty**. An **empty** element can be represented in two ways as below:

(1) A start-tag immediately followed by an end-tag as shown below:

```
<hr></hr>
```

(2) A complete empty-element tag is as shown below:

```
<hr />
```

Empty-element tags may be used for any element which has no content.

# XML Tags Rules

## Rule 1

XML tags are case-sensitive. Following line of code is an example of wrong syntax `</Address>`, because of the case difference in two tags, which is treated as erroneous **Syntax in XML.**

```
<address>This is wrong syntax</Address>
```

Following code shows a correct way, where we use the same case to name the start and the end tag.

```
<address>This is correct syntax</address>
```

## Rule 2

XML tags must be closed in an appropriate order, i.e., an XML tag opened inside another element must be closed before the outer element is closed. For example:

```
<outer_element>
```

```
<internal_element>
```

This tag is closed before the outer\_element

```
</internal_element>
```

```
</outer_element>
```

# XML Elements

XML elements can be defined as building blocks of an XML. Elements can behave as containers to hold text, elements, attributes, media objects or all of these.

Each XML document contains one or more elements, the scope of which are either delimited by start and end tags, or for empty elements, by an empty-element tag.

## Syntax

```
<element-name attribute1 attribute2>  
....content  
</element-name>
```

where

**element-name** is the name of the element. The *name* its case in the start and end tags must match.

**attribute1, attribute2** are attributes of the element separated by white spaces. An attribute defines a property of the element. It associates a name with a value, which is a string of characters. An attribute is written as:

name = "value"

The *name* is followed by an = sign and a string *value* inside double(" ") or single(' ') quotes.

# Empty Element

An empty element (element with no content) has following syntax:

```
<name attribute1 attribute2.../>
```

Example of an XML document using various XML element:

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

```
<contact-info>
```

```
<address category="residence">
```

```
<name>Tanmay Patil</name>
```

```
<company>TutorialsPoint</company>
```

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

```
<address/>
```

```
</contact-info>
```

# XML Elements Rules

Following rules are required to be followed for XML elements:

- An element *name* can contain any alphanumeric characters. The only punctuation marks allowed in names are the hyphen (-), under-score (\_) and period (.).
- Names are case sensitive. For example, Address, address, and ADDRESS are different names.
- Start and end tags of an element must be identical.
- An element, which is a container, can contain text or elements as seen in the above example.



# XML Attributes

Attributes are part of the XML elements. An element can have multiple unique attributes. Attribute gives more information about XML elements. To be more precise, they define properties of elements. An XML attribute

## Syntax

An XML attribute has following syntax:

```
<element-name attribute1 attribute2 >  
....content..  
< /element-name>
```

where *attribute1* and *attribute2* has the following form:

name = "value"

The *value* has to be in double (" ") or single (' ') quotes. Here, *attribute1* and *attribute2* are unique attribute labels.

Attributes are used to add a unique label to an element, place the label in a category, add a Boolean flag, or otherwise associate it with some string of data. Following example demonstrates the use of attributes: is always a *name-value* pair.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE garden [
<!ELEMENT garden (plants)*>
<!ELEMENT plants (#PCDATA)>
<!ATTLIST plants category CDATA #REQUIRED>
]>
<garden>
<plants category="flowers" />
<plants category="shrubs">
</plants>
</garden>
```

Attributes are used to distinguish among elements of the same name. When you do not want to create a new element for every situation. Hence, use of an attribute can add a little more detail in differentiating two or more similar elements.

In the above example we have categorized the plants by including attribute *category* and assigning different values to each of the elements. Hence we have two categories of *plants*, one *flowers* and other *color*. Hence we have two plant elements with different attributes.

# Attribute Types

Attribute Type	Description
StringType	It takes any literal string as a value. CDATA is a StringType. CDATA is character data. This means, any string of non-markup characters is a legal part of the attribute.
TokenizedType	<p>This is more constrained type. The TokenizedType attributes are <b>ID</b>: It is used to specify the element as unique.</p> <p><b>IDREF</b>: It is used to reference an ID that has been named for another element.</p> <p><b>IDREFS</b>: It is used to reference all IDs of an element.</p> <p><b>ENTITY</b>: It indicates that the attribute will represent an external entity in the document.</p> <p><b>ENTITIES</b>: It indicates that the attribute will represent external entities in the document.</p> <p><b>NMTOKEN</b>: It is similar to CDATA with restrictions on what data can be part of the attribute.</p> <p><b>NMTOKENS</b>: It is similar to CDATA with restrictions on what data can be part of the attribute.</p>
EnumeratedType	<p>This has a list of predefined values in its declaration. out of which, it must assign one value. There are two types of enumerated attribute:</p> <p><b>NotationType</b>: It declares that an element will be referenced to a NOTATION declared somewhere else in the XML document.</p> <p><b>Enumeration</b>: Enumeration allows you to define a specific list of values that the attribute value must match.</p>

# Element Attribute Rules

- An attribute name must not appear more than once in the same start-tag or empty-element tag.
- An attribute must be declared in the Document Type Definition (DTD) using an Attribute-List Declaration.
- Attribute values must not contain direct or indirect entity references to external entities.
- The replacement text of any entity referred to directly or indirectly in an attribute value must not contain either less than sign <

# XML Comments

XML comments are similar to HTML comments. The comments are added as notes or lines for understanding the purpose of an XML code. Comments can be used to include related links, information and terms. They are visible only in the source code; not in the XML code. Comments may appear anywhere in XML code.

## Syntax

```
<!-------Your comment----->
```

A comment starts with `<!--` and ends with `-->`. You can add textual notes as comments between the characters. You must not nest one comment inside the other.

# Example

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

```
<!--Students grades are uploaded by months---->
```

```
<class_list>
```

```
<student>
```

```
<name>Tanmay</name>
```

```
<grade>A</grade>
```

```
</student>
```

```
</class_list>
```

# XML Character Entities

- Before we understand the Character Entities, let us first understand what an XML entity is.
- As put by W3 Consortium the definition of entity is as follows:
- *“The document entity serves as the root of the entity tree and a starting-point for an XML processor.”*
- This means, entities are the placeholders in XML. These can be declared in the document prolog or in a DTD. Both, the HTML and the XML, have some symbols reserved for their use, which cannot be used as content in XML code. For example, < and > signs are used for opening and closing XML tags. To display these special characters, the character entities are used.

# Contd..

There are few special characters or symbols which are not available to be typed directly from keyboard.

Character Entities can be used to display those symbols/special characters also.

## **Types of Character Entities**

- Predefined Character Entities
- Numbered Character Entities
- Named Character Entities



# Predefined Character Entities

- They are introduced to avoid the ambiguity while using some symbols. For example, an ambiguity is observed when less than (<) or greater than (>) symbol is used with the angle tag (<>). Character entities are basically used to delimit tags in XML.
- Following is a list of pre-defined character entities from XML specification. These can be used to express characters without ambiguity.
- Ampersand: &amp;
- Single quote: &apos;
- Greater than: &gt;
- Less than: &lt;
- Double quote: &quot;

# Numeric Character Entities

The numeric reference is used to refer to a character entity. Numeric reference can either be in decimal or hexadecimal format. As there are thousands of numeric references available, these are a bit hard to remember. Numeric reference refers to the character by its number in the Unicode character set.

- General syntax for decimal numeric reference is:

`&# decimal number ;`

- General syntax for hexadecimal numeric reference is:

`&#x Hexadecimal number ;`

# Contd..

Entity name	Character	Decimal reference	Hexadecimal reference
quot	"	&#34;	&#x22;
amp	&	&#38;	&#x26;
apos	'	&#39;	&#x27;
lt	<	&#60;	&#x3C;
gt	>	&#62;	&#x3E;

# Named Character Entity

As it is hard to remember the numeric characters, the most preferred type of character entity is the named character entity. Here, each entity is identified with a name.

- For example
  - 'Acute' represents capital Á character with acute accent.
  - 'ugrave' represents the small ù with grave accent.

# XML CDATA Sections

- The term CDATA means, Character Data. CDATA are defined as blocks of text that are not parsed by the parser, but are otherwise recognized as markup.
- The predefined entities such as `&lt;`, `&gt;`, and `&amp;`; require typing and are generally difficult to read in the markup. In such cases, CDATA section can be used. By using CDATA section, you are commanding the parser that the particular section of the document contains no markup and should be treated as regular text.

# Syntax

```
<![CDATA[  
    characters with markup  
]]>
```

The above syntax is composed of three sections:

**CDATA Start section** - CDATA begins with the nine-character delimiter **<![CDATA[**

**CDATA End section** - CDATA section ends with **]]>** delimiter.

**CData section** - Characters between these two enclosures are interpreted as characters, and not as markup. This section may contain markup characters (<, >, and &), but they are ignored by the XML processor.

# Example

- The following markup code shows example of CDATA. Here, each character written inside the CDATA section is ignored by the parser.

```
<script>
<![CDATA[
    <message> Welcome to TutorialsPoint </message>
]] >
</script >
```

**In the above syntax, everything between <message> and </message> is treated as character data and not as markup.**

# CDATA Rules

The given rules are required to be followed for XML CDATA:

- CDATA cannot contain the string "]]>" anywhere in the XML document.
- Nesting is not allowed in CDATA section.



# XML Whitespace

Whitespace is a collection of spaces, tabs, and newlines. They are generally used to make a document more readable.

XML document contain two types of white spaces

- (a) Significant Whitespace and
- (b) Insignificant Whitespace.

## Significant Whitespace

A significant Whitespace occurs within the element which contain text and markup present together. For example:

`<name>TanmayPatil</name>`

and

`<name>Tanmay Patil</name>`

The above two elements are different because of the space between **Tanmay** and **Patil**. Any program reading this element in an XML file is obliged to maintain the distinction.

# Insignificant Whitespace

Insignificant whitespace means the space where only element content is allowed. For example:

```
<address.category="residence">
```

or

```
<address....category="..residence">
```

The above two examples are same. Here, the space is represented by dots (.). In the above example, the space between *address* and *category* is insignificant.

A special attribute named **xml:space** may be attached to an element. This indicates that whitespace should not be removed for that element by the application. You can set this attribute to **default** or **preserve** as shown in the example below:

```
<!ATTLIST address xml:space (default|preserve) 'preserve'>
```

Where:

The value **default** signals that the default whitespace processing modes of an application are acceptable for this element;

The value **preserve** indicates the application to preserve all the whitespaces.

# XML Processing

- *Processing instructions (PIs) allow documents to contain instructions for applications. PIs are not part of the character data of the document, but MUST be passed through to the application.”*
- Processing instructions (PIs) can be used to pass information to applications. PIs can appear anywhere in the document outside the markup. They can appear in the prolog, including the document type definition (DTD), in textual content, or after the document.

# Syntax

Following is the syntax of PI:

**<?target instructions?>**

Where:

**target** - identifies the application to which the instruction is directed.

**instruction** - it is a character that describes the information for the application to process.

A PI starts with a special tag <? and ends with ?>. Processing of the contents ends immediately after the string ?> is encountered.

# Example

PIs are rarely used. They are mostly used to link XML document to a style sheet.

Following is an example:

```
<?xml-stylesheet href="tutorialspointstyle.css" type="text/css"?>
```

Here, the *target* is an xmlstylesheet.

*href="tutorialspointstyle.css"* and *type="text/css"* are *data* or *instructions* that the target application will use at the time of processing the given XML document.

In this case, a browser recognizes the target by indicating that the XML should be transformed before being shown; the first attribute states that the type of the transform is XSL and the second attribute points to its location.

# Processing Instructions Rules

A PI can contain any data except the combination `?>`, which is interpreted as the closing delimiter.

Here are two examples of valid PIs:

**`<?welcome to pg=10 of Web Technology?>`**

**`<?welcome?>`**

# XML Encoding

- **Encoding** is the process of converting unicode characters into their equivalent binary representation. When the XML processor reads an XML document, it encodes the document depending on the type of encoding. Hence, we need to specify the type of encoding in the XML declaration.

There are mainly two types of encoding:

- **UTF-8**
- **UTF-16**
- UTF stands for *UCS Transformation Format*, and UCS itself means Universal Character Set. The number 8 or 16 refers to the number of bits used to represent a character.
- They are either 8(one byte) or 16(two bytes). For the documents without encoding information, UTF-8 is set by default.

# Syntax

Encoding type is included in the prolog section of the XML document.

Syntax for UTF-8 encoding is as below:

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

Syntax for UTF-16 encoding:

```
<?xml version="1.0" encoding="UTF-16" standalone="no" ?>
```



# XML Validation

**Validation** is a process by which an XML document is validated. An XML document is said to be valid if its contents match with the elements, attributes and associated document type declaration (DTD), and if the document complies with the constraints expressed in it. Validation is dealt in two ways by the XML parser. They are:

- Well-formed XML document
- Valid XML document

# Well-formed XML document

An XML document is said to be **well-formed** if it adheres to the following rules:

- Non DTD XML files must use the predefined character entities for **amp(&)**, **apos(single quote)**, **gt(>)**, **lt(<)**, **quot(double quote)**.
- It must follow the ordering of the tag. i.e., the inner tag must be closed before closing the outer tag.
- Each of its opening tags must have a closing tag or it must be a self ending tag.(<title>....</title> or <title/>).
- It must have only one attribute in a start tag, which needs to be quoted.
- **amp(&)**, **apos(single quote)**, **gt(>)**, **lt(<)**, **quot(double quote)** entities other than these must be declared.

# Example of well-formed XML document

```
<?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>
```

Above example is said to be well-formed as:

- It defines the type of document. Here, the document type is **element** type.
- It includes a root element named as **address**.
- Each of the child elements among name, company and phone is enclosed in its self-explanatory tag.
- Order of the tags is maintained.

# Valid XML document

- If an XML document is well-formed and has an associated Document Type Declaration (DTD), then it is said to be a valid XML document.
- The XML Document Type Declaration, commonly known as DTD, is a way to describe XML language precisely. DTDs check vocabulary and validity of the structure of XML documents against grammatical rules of appropriate XML language.
- An XML DTD can be either specified inside the document, or it can be kept in a separate document and then linked separately.

# XML Schema

Xml Schema Documents

# Drawbacks of DTD

- Syntax is not XML and so XML parsers cannot parse them into component parts very easily.
- Weak Data types.
- Not modular so cannot be reusable.
- Not easily extensible. (No inheritance in DTD)

# XML Schema

- XML schema is used to describe structure of the data in XML document and it overcomes several disadvantages of DTD.
- Supports
  - More data types
  - Is XML document
  - Extensible
  - Modular
- The XML Schema language is also referred to as XML Schema Definition (XSD).

# Numeric types

- `integer` Examples: 45
- `nonPositiveInteger`, `negativeInteger`, `nonNegativeInteger`, `positiveInteger`
- `byte` (8 bits), `short` (16 bits), `int` (32 bits), `long` (64 bits)
- `unsignedLong`, `unsignedInt`, `unsignedShort`, `unsignedByte`
- `number` (18 decimal places) Examples:  
1.1, .8, 9.5, 6, 8.7E3
- `float` (32-bit)
- `double` (64-bit) Examples: -INF, INF, 8.7E3, 1.1, NaN
- `decimal` for all numeric types



# String types

- **string** (no whitespace normalization is done) Examples: **hello world**
- **normalizedString** (whitespace replaced by #x20)
- **token** (Whitespaces are normalized)
- **language** (accept all the language codes standardized by RFC 1766)  
Examples: **us-en, en**
- **NMTOKEN** (a single name token composed of characters allowed in an XML name) Examples: **"ABC", "2010-01-15"**
- **NMTOKENS** (whitespace-separated lists of NMTOKEN components)
- **Name** (similar to NMTOKEN with the additional restriction that the values must start either with a letter or the characters ":" or "\_".)

# String types

- **NCName** ( noncolonized name)
- **ID** (same as **NCName** and here must not be any duplicate values in a document)
- **IDREF** (it must match an ID defined in the same document)
- **IDREFS** (whitespace-separated list of **NCName** values)
- **ENTITY** (same as **NCName**, provided for compatibility)
- **ENTITIES** (value spaces of ENTITIES are the whitespace-separated lists )

# Date Types

- `dateTime (yyyy-mm-ddThh:mm:ss)` Examples: 2010-01-10T16:30:34
- `date(yyyy-mm-dd)` Same as `datetime` with an optional time zone (Examples: 2010-01-11, 2010-01-11+12:00 .
- `duration (PnYnMnDTnHnMnS)` Each part (except the leading "P") is optional. Example: P2Y3M8DT100S
- `time (hh:mm:ss)` Example: 6:30:34
- `gYearMonth` Examples: 2010-10, 2010-10+02:00, or 2010-10Z)
- `gYear` Examples 2010, 2010+02:00 or 2010Z
- `gDay (---DD)`, `gMonthDay( --MM-DD )`, `gMonth(--MM or --MM--)`

# Other types

- **boolean** (Legal values for boolean are true, false, 1 (which indicates true), and 0 (which indicates false)).
- **base64Binary** (Base64-encoded binary data)
- **hexBinary** (hexadecimal-encoded binary data)
- **anyURI** Example: `http://www.xyz.com/home.html`

# Declaring elements

- Simple form:
  - `<xsd:element name="IssueDate" />`
- With type specified:
  - `<xsd:element name="IssueDate" type="xsd:date" />`
- With number of times element can occur:
  - `<xsd:element name="IssueDate" type="xsd:date" minOccurs="0" maxOccurs="1" />`

→ In XML:

**`<IssueDate>2007-05-25</IssueDate>`**

# minOccurs, maxOccurs

- **minOccurs, maxOccurs** can have any non-negative integer.
- **maxOccurs** can take a special value → unbounded : which means that the element must occur at least **minOccurs** times but there is no upper limit on how many times it can occur.
- If **minOccurs** is specified and **maxOccurs** is omitted, the value of **maxOccurs** = **minOccurs**.

# <xs:schema>

- Root element of all XML Schema document.
- Example :<xs:schema  
xmlns:xs="http://www.w3.org/2001/XMLSchema"  
elementFormDefault="qualified">
- Could also have targetNamespace and xmlns attributes
- An XML document can reference a Schema through xsi:schemaLocation attribute of the root tag.

# Simple atomic type

XML:

```
<name>Mocha Java</name>
```

```
<price>11.95</price>
```

XSD:

```
<xs:element name="name" type="xs:string"/>
```

```
<xs:element name="price" type="xs:double"/>
```

Default Values: 

```
<xs:element name="size" type="xs:int"
    default="12"/>
```

Fixed Values: 

```
<xs:element name="size" type="xs:int"
    fixed="12"/>
```

In the same way attributes can also be defined by 

```
<xs:attribute>
```



# Restricting values

## Restriction Specifiers

Constraints	Description
enumeration	Defines a list of acceptable values
fractionDigits	Specifies the maximum number of decimal places allowed. Must be equal to or greater than zero
length	Specifies the exact number of characters or list items allowed. Must be equal to or greater than zero
maxExclusive	Specifies the upper bounds for numeric values (the value must be less than this value)
maxInclusive	Specifies the upper bounds for numeric values (the value must be less than or equal to this value)
maxLength	Specifies the maximum number of characters or list items allowed. Must be equal to or greater than zero
minExclusive	Specifies the lower bounds for numeric values (the value must be greater than this value)
minInclusive	Specifies the lower bounds for numeric values (the value must be greater than or equal to this value)
minLength	Specifies the minimum number of characters or list items allowed. Must be equal to or greater than zero
pattern	Defines the exact sequence of characters that are acceptable
totalDigits	Specifies the exact number of digits allowed. Must be greater than zero
whiteSpace	Specifies how white space (line feeds, tabs, spaces, and carriage returns) is handled

# Examples

- ```
<xs:element name="shirtSize">  
  <xs:simpleType>  
    <xs:restriction base="xs:int">  
      <xs:enumeration value="36"/>  
      <xs:enumeration value="40"/>  
      <xs:enumeration value="42"/>  
      <xs:enumeration value="44"/>  
    </xs:restriction>  
  </xs:simpleType>  
</xs:element>
```

Restrict the sizes to 36 40 42 44

- ```
<xs:element name="empage">  
  <xs:simpleType>  
    <xs:restriction base="xs:int">
```

Employment Age between 20 and 60

- `<xs:element name="initials">`

- `<xs:simpleType>`

- `<xs:restriction base="xs:string">`

- `<xs:pattern value="[a-zA-Z][a-zA-Z][a-zA-Z]" />`

- `</xs:restriction>`

- `</xs:simpleType>`

- `</xs:element>`

Country code with three letters only  
either in upper or lower case

- `<xs:element name="letter">`

- `<xs:simpleType>`

- `<xs:restriction base="xs:integer">`

- `<xs:pattern value="([0-9])*" />`

- `</xs:restriction>`

- `</xs:simpleType>`

- `</xs:element>`

Any integer, no value is also  
acceptable

•<xs:element name="letter">

Any integer

<xs:simpleType>

<xs:restriction base="xs:integer">

<xs:pattern value="([0-9])+"/>

</xs:restriction>

</xs:simpleType>

</xs:element>

•<xs:element name="gender">

<xs:simpleType>

<xs:restriction base="xs:string">

<xs:pattern value="male|female"/>

</xs:restriction>

</xs:simpleType>

</xs:element>

•<xs:element name="password">

<xs:simpleType>

<xs:restriction base="xs:string">

<xs:pattern value="[a-zA-Z0-9]{8}"/>

</xs:restriction>

</xs:simpleType>

</xs:element>

Password consisting of 8 chars containing letters or digits

- ```
<xs:element name="password">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:length value="8"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

- ```
<xs:element name="date">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:minLength value="5"/>
      <xs:maxLength value="8"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

# List type

- List type can be created from an existing list of strings or integers separated by spaces specified as simple type.

- Example:

XML: `<sizes>23 32 36 40 42 44</sizes>`

XSD:

```
<xsd:simpleType name="sizes">
```

```
<xsd:list itemType="shirtSize" />
```

```
</xsd:simpleType>
```

where **shirtSize** may be defined as

Enumerated restricted values (1st example of

“Restricting values” )

# Union Types

- Union Types enable element content or attribute values to be of a type provided from a union of different simple and list types.

XML:

```
<sizes>20 23 32 36 40</sizes>
```

```
<sizes>S M L XL XXL</sizes>
```

```
<xsd:simpleType name="sizesUnion">
```


```
<xsd:union memberTypes="sizes letter"/>
```

```
</xsd:simpleType>
```

Where **sizes** is defined as in previous slide and **letter** may be defined similarly

# User-defined types/complex

```
<xsd:complexType name="Address">  
  <xsd:sequence>  
    <xsd:element name="Street" type="xs:string"/>  
    <xsd:element name="City" type="xs:string" />  
    <xsd:element name="State" type="xs:string" />  
    <xsd:element name="Country" type="xs:string" />  
    <xsd:element name="PIN" type="xs:pin" />  
  </xsd:sequence>  
</xsd:complexType>
```



Can you define the XSD for pin?

Elements must appear in the order in which they are declared



# Using complex types

- `<xsd:element name="MailAddress" type="Address" />`

- `<xsd:element name="BillingAdd"`

In XML  
`type="Address"/>`  
`<MailAddress>`

`<Street>Gangarams Book Bureau, M.G  
Road</Street>`

`<City>Bangalore</City>`

`<State>Karnataka</State>`

`<Country>India</Country>`

`<PIN>560001</PIN></MailAddress>`

# Sequence Content Type

- An element content can be defined as containing a sequence of other elements.

```
<xsd:element name="MailAddress">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="Street"/>
      <xsd:element name="City"/>
      <xsd:element name="State"/>
      <xsd:element name="Country"/>
      <xsd:element name="PIN"/>
    </xsd:sequence>
  </xsd:complexType> </xsd:element>
```

The `<xsd:sequence>` specifies that the child elements must appear in a specific order

# <xsd:anyType>

- Any

```
<xsd:element name="XXX" type="xsd:anyType" />
```

# Complex Empty Elements

XML:

```
<Line width="10">
```

XSD:

```
<xs:element name="Line">  
  <xs:complexType>  
    <xs:complexContent>  
      <xs:restriction base="xs:integer">  
        <xs:attribute name="width"  
type="xs:positiveInteger"/>  
      </xs:restriction>  
    </xs:complexContent>  
  </xs:complexType>  
</xs:element>
```

# Complete Example

## XML

```
1  <?xml version="1.0"?>
2  <letter>
3    <from>Malini</from>
4    <to>Manju</to>
5    <cc>Mrinal</cc>
6    <bcc/>
7    <subject>Reminder</subject>
8    <body>Let us meet tomorrow</body>
9  </letter>
10
```

Schema for this ...

```
1 <?xml version="1.0"?>
2 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
3   <xs:element name="letter">
4     <xs:complexType>
5       <xs:sequence>
6         <xs:element name="from" type="xs:string"/>
7         <xs:element name="to" type="xs:string"/>
8         <xs:element name="cc" type="xs:string" minOccurs="0"/>
9         <xs:element name="bcc" type="xs:string" minOccurs="0"/>
10        <xs:element name="subject" type="xs:string"/>
11        <xs:element name="body" type="xs:string"/>
12      </xs:sequence>
13    </xs:complexType>
14  </xs:element>
15 </xs:schema>
```

XSD

# Question?

- Create a complex type for FullName consisting of FirstName, MiddleName and LastName where MiddleName is optional.

# ref

- An element which is defined elsewhere in the schema document can be used elsewhere by using **ref** attribute.
- For instance consider element called 'phone' defined as below

```
<xsd:element name="phone">
  <xsd:complexType>
    <xsd:element name="mobile"
      type="unsignedLong" minOccurs="0"
      maxOccurs="1" />
    <xsd:element name="res"
      type="unsignedLong" minOccurs="0"
      maxOccurs="1" />
  </xsd:complexType>
</xsd:element>
```



- This element can be referenced anywhere in the schema document like shown below:

```
<xsd:element name="addressBook">  
  <xsd:complexType>  
    <xsd:element name="name" />  
    <xsd:element ref="phone" />  
  <xsd:complexType>  
</xsd:element>
```

# Declaring Attributes

- Simple Attribute

```
<xsd:attribute name="href" />
```

- Attribute with data type

```
<xsd:attribute name="border"  
type="boolean" />
```

- Attribute with `use` → `required`, `optional`, `prohibited`

```
<xsd:attribute name="width"  
use="optional" />
```

- Attribute with `fixed` or `default` content

```
<xsd:attribute name="paid"  
use="required" fixed="True" />
```

```
<xsd:attribute name="paid" use="optional"  
default="True" />
```

```
1  <?xml version="1.0"?>
2  <books>
3      <book id="1">
4          <isbn>0-7434-0885-3</isbn>
5
6          <title>The 7 habits of highly effective people </title>
7              <authors>
8                  <author>Stephen R.Covey</author>
9              </authors>
10         </book>
11
12         <book id="2">
13             <isbn>978-0394719409</isbn>
14             <title>The Hidden Injuries of Class</title>
15             <authors>
16                 <author>Richard Sennett</author>
17                 <author>Jonathan Cob</author>
18             </authors>
19         </book>
20 </books>
```

XSD for this →

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
3   <xs:element name="books">
4     <xs:complexType>
5       <xs:sequence>
6         <xs:element maxOccurs="unbounded" ref="book"/>
7       </xs:sequence>
8     </xs:complexType>
9   </xs:element>
10  <xs:element name="book">
11    <xs:complexType>
12      <xs:sequence>
13        <xs:element ref="isbn"/>
14        <xs:element ref="title"/>
15        <xs:element ref="authors"/>
16      </xs:sequence>
17      <xs:attribute name="id" type="xs:integer"/>
18    </xs:complexType>
19  </xs:element>
20  <xs:element name="isbn" type="xs:NMTOKEN"/>
21  <xs:element name="title" type="xs:string"/>
22  <xs:element name="authors">
23    <xs:complexType>
24      <xs:sequence>
25        <xs:element maxOccurs="unbounded" ref="author"/>
26      </xs:sequence>
27    </xs:complexType>
28  </xs:element>
29  <xs:element name="author" type="xs:string"/>
30 </xs:schema>
```

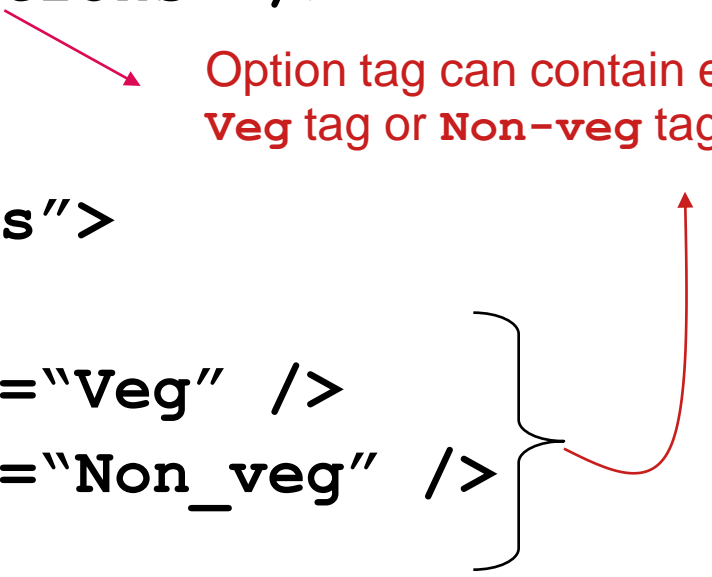
# Elements in any order

- `<xs:all>` specifies that
  - the child elements can appear in any order
  - each child element must occur only once
- `<xs:element name="employee">`
  - `<xs:complexType>`
    - `<xs:all>`
      - `<xs:element name="fname"`  
`type="xs:string"/>`
      - `<xs:element name="lname"`  
`type="xs:string"/>`

# Choice Group

```
<xsd:element name="Dinner">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="FullName" />
      <xsd:element ref="Options" />
    </xsd:sequence>
  </xsd:element>
  <xsd:group name="Options">
    <xsd:choice>
      <xsd:element name="Veg" />
      <xsd:element name="Non_veg" />
    </xsd:choice>
  </xsd:group>
</xsd:element>
```

Option tag can contain either Veg tag or Non-veg tag



# Extending Complex Types

- A complex type can be extended to include another element thus creating new types.
- This is a way by which we can reuse existing structure to build larger structures.
- Example:

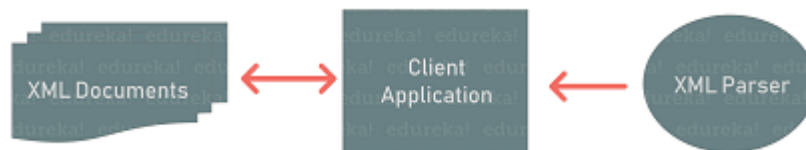
```
<xsd:element name="ContactDetails">  
  <xsd:complexType>  
    <xsd:complexContent>  
      <xsd:extension base="Address">  
        <xsd:attribute name="entrydt"  
          type="xsd:date" /> </xsd:extension>  
      </xsd:complexContent>  
    </xsd:complexType></xsd:element>
```

# XML Parser

*XML, eXtensible Markup Language* is a markup language that defines a set of rules for encoding documents in a format that is readable. **XML Parsing** refers to going through an XML document in order to access or modify data. An **XML Parser** provides the required functionality to access or modify data in an XML document.

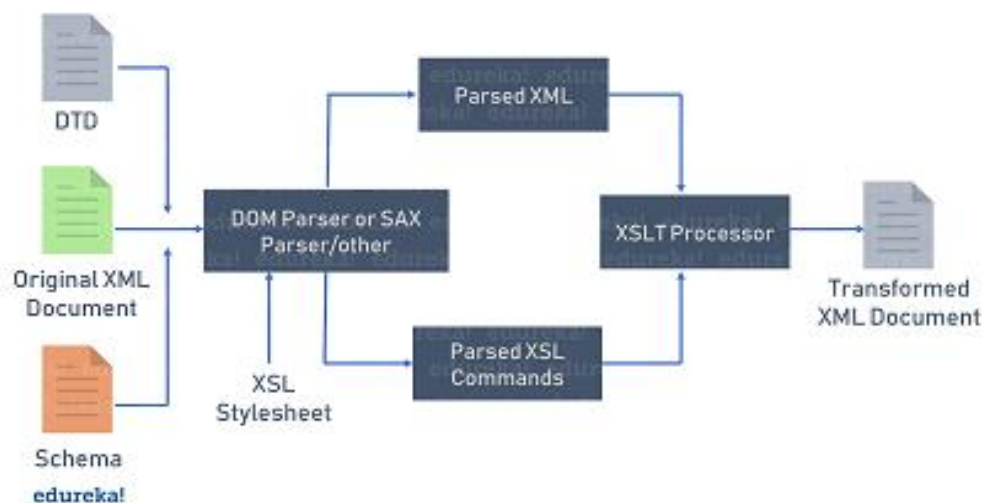
## What is XML Parser?

The **XML parser** is a software library or a package that provides an interface for client applications to work with XML documents. It checks for proper format of the XML document and also validates the XML documents.



## JAVA XML Parser

The fundamental component of XML development is XML parsing. XML parsing for Java is a standalone XML component that parses an XML document (and at times also a standalone DTD or XML Schema) so that user program can process it. The figure below shows an XML document as input to the XML Parser for Java.



- An XML document is sent as input to the XML Parser for Java
- The DOM or SAX parser interface parses the XML document
- The parsed XML is then transferred to the application for further processing

The XML Parser for Java might also include an integrated XSL Transformation (XSLT) Processor for transforming XML data using XSL stylesheets. Using the XSLT Processor, you can easily transform XML documents from XML to XML, XML to HTML, or virtually to any other text-based format.

Java provides a lot of options to parse XML documents. Some of the commonly used java XML parsers are:



1. DOM Parser
2. SAX Parser
3. StAX Parser
4. JAXB

**DOM Parser** – DOM is an acronym for Document Object Model. Unlike SAX parser DOM parser loads the complete XML file into memory and creates a tree structure where each node in the tree represents a component of XML file. With DOM parser you can create nodes, remove nodes, change their contents and traverse the node hierarchy. DOM provides maximum flexibility while working with XML files but it comes with a cost of potentially large memory footprint and significant processor requirements in case of large XML files

**SAX Parser** – SAX is an acronym for Simple API for XML. SAX Parser parses the XML file line by line and triggers events when it encounters opening tag, closing tag or character data in XML file. This is why SAX parser is called an event-based parser

**StAX Parser** – StAX is an acronym for Streaming API for XML. Stream-based parsers are very useful when your application has memory limitations. For example, a cellphone running Java Micro Edition. Similarly, if your application needs to process several requests simultaneously, for example an application server, StAX parser should be used.

**Stream-based parsing can further be classified as:**

**Pull Parsing** – In pull parsing, client application calls for methods on an XML parsing library when it needs to interact with an XML infoset. In other words, client only gets XML data when it explicitly asks for it.

**Push Parsing** – In push parsing, it is the XML parser which pushes XML data to the client, when it encounters elements in an XML infoset. In other words, parser sends the data to application irrespective of the application being ready to use it or not.

**Comparison between SAX, DOM and StAX parser:**

The table below summarizes the features of SAX, DOM and StAX parser

Feature	StAX	SAX	DOM
API Type	Pull, streaming	Push, streaming	In memory tree
Ease of Use	High	Medium	High
XPath Capability	No	No	Yes
CPU and Memory Efficiency	Good	Good	Varies
Forward Only	Yes	Yes	No
Read XML	Yes	Yes	Yes
Write XML	Yes	No	Yes
Create, Read, Update, Delete	No	No	Yes

## Java XML Parser – DOM

*DOM* stands for *Document Object Model*. DOM Parser is the easiest java XML parser to implement and learn. It parses an entire XML document, loads it into memory and constructs a tree representation of the document.

**new.xml**

```
<?xml version="1.0"?>
<shops>
  <supermarket>
    <sid>201</sid>
    <sname>sparc</sname>
    <product> grocery</product>
    <branch> two</branch>
    <location> new york</location>
  </supermarket>
  <supermarket>
    <sid>540</sid>
    <sname> big basket</sname>
    <product> grocery</product>
    <branch> seven</branch>
    <location>India</location>
  </supermarket>
  <supermarket>
    <sid>301</sid>
    <sname>Wallmart</sname>
    <product> grocery</product>
    <branch> fifteen</branch>
    <location> France</location>
  </supermarket>
</shops>
```

# How to Parse XML File using DOM parser

```
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;
import org.w3c.dom.Document;
import org.w3c.dom.NodeList;
import org.w3c.dom.Node;
import org.w3c.dom.Element;
import java.io.File;

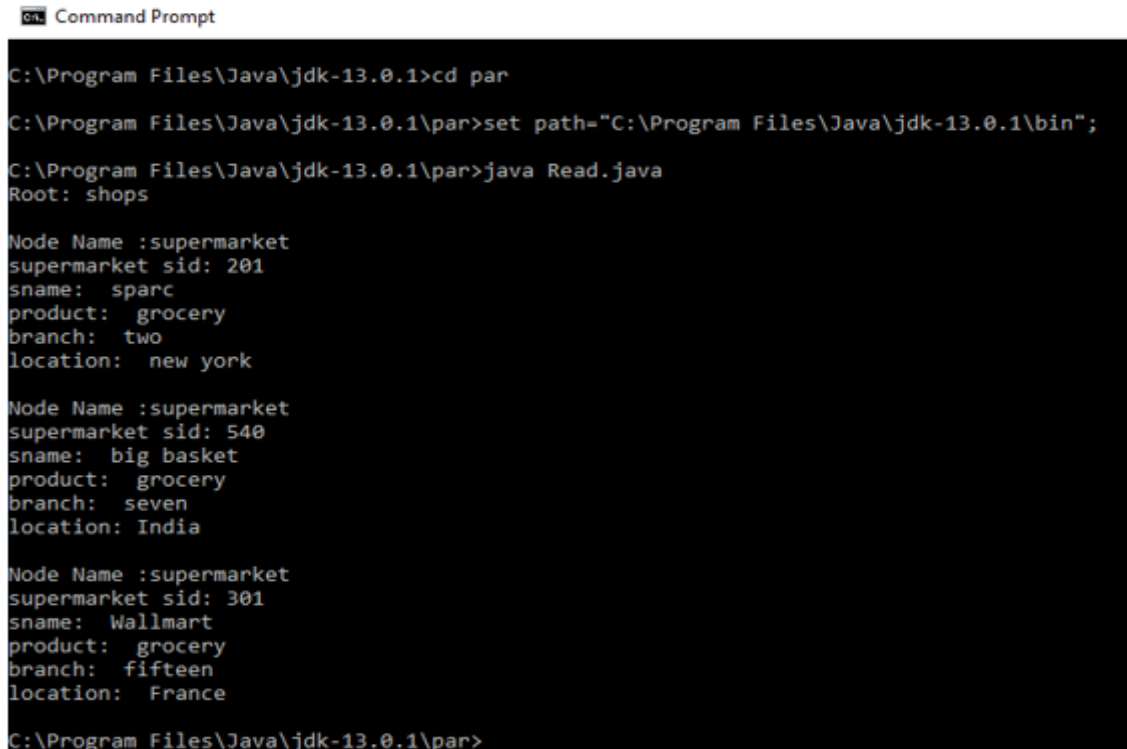
public class Read
{
    public static void main(String argv[])
    {
        try
        {
            File file = new File("C:\\Program Files\\Java\\jdk-13.0.1\\par\\new.xml");
            DocumentBuilderFactory docb = DocumentBuilderFactory.newInstance();
            DocumentBuilder dbu = docb.newDocumentBuilder();
            Document dt = dbu.parse(file);
            dt.getDocumentElement().normalize();
            System.out.println("Root: " + dt.getDocumentElement().getNodeName());
            NodeList nd = dt.getElementsByTagName("supermarket");
            for (int i = 0; i<nd.getLength(); i++)
            {
                Node node = nd.item(i);
                System.out.println("\nNodeName : " + node.getNodeName());
                if (node.getNodeType() == Node.ELEMENT_NODE)
                {
                    Element el = (Element) node;
                    System.out.println("supermarket sid: " + el.getElementsByTagName("sid").item(0).getTextContent());
                    System.out.println("sname: " + el.getElementsByTagName("sname").item(0).getTextContent());
                    System.out.println("product: " + el.getElementsByTagName("product").item(0).getTextContent());
                    System.out.println("branch: " + el.getElementsByTagName("branch").item(0).getTextContent());
                    System.out.println("location: " + el.getElementsByTagName("location").item(0).getTextContent());
                }
            }
        }
    }
}
```

```

    } }
}
catch (Exception e)
{
e.printStackTrace();
} }
}

```

## Output:



```

C:\Program Files\Java\jdk-13.0.1>cd par
C:\Program Files\Java\jdk-13.0.1\par>set path="C:\Program Files\Java\jdk-13.0.1\bin";
C:\Program Files\Java\jdk-13.0.1\par>java Read.java
Root: shops

Node Name :supermarket
supermarket sid: 201
sname:  sparco
product:  grocery
branch:  two
location:  new york

Node Name :supermarket
supermarket sid: 540
sname:  big basket
product:  grocery
branch:  seven
location:  India

Node Name :supermarket
supermarket sid: 301
sname:  Wallmart
product:  grocery
branch:  fifteen
location:  France
C:\Program Files\Java\jdk-13.0.1\par>

```

## Advantage and Disadvantages of DOM Parser

### Advantages

- API is very simple to use
- It supports both read and write operations
- Preferred when random access to widely separated parts of a document is required

### Disadvantages

- It is memory inefficient. As the file size increases, its performance deteriorates and consumes more memory
- Comparatively slower than other XML parsers available in Java

# How to Create XML File using DOM parser

```
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.transform.OutputKeys;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.transform.TransformerException;

public class CreateXMLDOM
{
    public static void main(String[] args)
    {
        // Defines a factory API that enables applications to obtain a parser that produces DOM object trees from XML documents.
        DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();

        // Defines the API to obtain DOM Document instances from an XML document.
        DocumentBuilder db;

        try
        {
            db = dbf.newDocumentBuilder();

            // The Document interface represents the entire HTML or XML document.
            Document doc = db.newDocument();

            // The Element interface represents an element in an HTML or XML document.
            Element element = doc.createElementNS("https://crunchify.com/CrunchifyCreateXMLDOM", "Companies");

            // Adds the node newChild to the end of the list of children of this node.
            // If the newChild is already in the tree, it is first removed.
            doc.appendChild(element);

            // append child elements to root element
            element.appendChild(getCompany(doc, "1", "Paypal", "Payment", "1000"));
        }
    }
}
```

```
element.appendChild(getCompany(doc, "2", "Amazon", "Shopping", "2000"));
element.appendChild(getCompany(doc, "3", "Google", "Search", "3000"));
element.appendChild(getCompany(doc, "4", "Crunchify", "Java Tutorials", "10"));
```

**// output DOM XML to console**

**// An instance of this abstract class can transform a source tree into a result tree.**

```
Transformer trans = TransformerFactory.newInstance().newTransformer();
trans.setOutputProperty(OutputKeys.INDENT, "yes");
```

**// Acts as a holder for a transformation Source tree in the form of a Document Object Model (DOM) tree.**

```
DOMSource source = new DOMSource(doc);
```

**// Acts as an holder for a transformation result, which may be XML, plain Text, HTML, or some other form of markup.**

```
StreamResult console = new StreamResult(System.out);
trans.transform(source, console);
System.out.println("\nTutorial by Crunchify. XML DOM Created Successfully..");
}
catch (TransformerException | ParserConfigurationException e)
{
e.printStackTrace();
}
}
```

**// The Node interface is the primary datatype for the entire Document Object Model.**

**// It represents a single node in the document tree.**

```
private static Node getCompany(Document doc, String id, String name, String age, String role)
{
Element ele = doc.createElement("Company");
ele.setAttribute("id", id);
ele.appendChild(getCompanyElements(doc, ele, "Name", name));
ele.appendChild(getCompanyElements(doc, ele, "Type", age));
ele.appendChild(getCompanyElements(doc,ele, "Employees", role));
return ele;
}
```

**// Utility method to create text node**

```
private static Node getCompanyElements(Document doc, Element element, String name, String value)
{
Element node = doc.createElement(name);
node.appendChild(doc.createTextNode(value));
}
```

```
return node;
}
}
```

### Output:

```
F:\shankar\jj>javac CreateXMLDOM.java

F:\shankar\jj>java CreateXMLDOM
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<Companies xmlns="https://crunchify.com/CrunchifyCreateXMLDOM">
  <Company id="1">
    <Name>Paypal</Name>
    <Type>Payment</Type>
    <Employees>1000</Employees>
  </Company>
  <Company id="2">
    <Name>Amazon</Name>
    <Type>Shopping</Type>
    <Employees>2000</Employees>
  </Company>
  <Company id="3">
    <Name>Google</Name>
    <Type>Search</Type>
    <Employees>3000</Employees>
  </Company>
  <Company id="4">
    <Name>Crunchify</Name>
    <Type>Java Tutorials</Type>
    <Employees>10</Employees>
  </Company>
</Companies>

XML DOM Created Successfully..
```

## ***SAX Parser***

SAX Is Simple API for XML and meant has Push Parser also considered to be stream-oriented XML Parser. it is used in case of high- performance applications like where the XML file is too large and comes with the community- based standard and requires less memory. The main task is to read the XML file and creates an event to do call function or uses call back routines. The working of this parser is just like Event handler part of the java. it is necessary to register the handlers to parse the document for handling different events. SAX Parser uses three methods `startElement()` , `endElement()` , `characters()`.

- **startElement():** Is used to identify the element, start element is identified.
- **endElement():** To stop the Supermarket tag in the example.
- **character():** Is used to identify the character in the node

### **new.xml**

```
<?xml version="1.0"?>
<shops>
<supermarket>
<sid>201</sid>
<sname>sparc</sname>
<product> grocery</product>
<branch> two</branch>
<location> new york</location>
</supermarket>
<supermarket>
<sid>540</sid>
<sname> big basket</sname>
<product> grocery</product>
<branch> seven</branch>
<location>India</location>
</supermarket>
<supermarket>
<sid>301</sid>
<sname>Wallmart</sname>
<product> grocery</product>
<branch> fifteen</branch>
<location> France</location>
```



</supermarket>

</shops>

## Program

```
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;
import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.DefaultHandler;
public class Rsax
{
public static void main(String args[])
{
try
{
SAXParserFactory factory = SAXParserFactory.newInstance();
SAXParser saxParser = factory.newSAXParser();
DefaultHandler handler = new DefaultHandler()
{
booleansid = false;
booleansname = false;
boolean product = false;
boolean branch = false;
boolean location = false;
public void startElement( String sg, String pp,String q, Attributes a) throws SAXException
{
System.out.println("Beginning Node :" + q);
if(q.equalsIgnoreCase("SID"))
{
sid=true;
}
if (q.equalsIgnoreCase("SNAME"))
{
sname = true;
}
if (q.equalsIgnoreCase("PRODUCT"))
{
product = true;
}
if (q.equalsIgnoreCase("BRANCH"))
```

```

{
branch = true;
}
if (q.equalsIgnoreCase("LOCATION"))
{
location = true;
}
}
public void endElement(String u, String l, String qNa) throws SAXException
{
System.out.println("LAsT Node:" + qNa);
}
public void characters(char chr[], int st, int len) throws SAXException
{
if (sid)
{
System.out.println("SID : " + new String(chr, st, len));
sid = false;
}
if (sname)
{
System.out.println("Shop Name: " + new String(chr, st, len));
sname = false;
}
if (product)
{
System.out.println("Available Product: " + new String(chr, st, len));
product = false;
}
if (branch)
{
System.out.println("No.of Branches: " + new String(chr, st, len));
branch = false;
}
if (location)
{
System.out.println("Address : " + new String(chr, st, len));
location = false;
}
}
};
saxParser.parse("C:\\Program Files\\Java\\jdk-13.0.1\\par\\new.xml", handler);
}
catch (Exception ex)

```

```
{  
ex.printStackTrace();  
}  
}  
}
```

Output:

```
Command Prompt  
C:\Program Files\Java\jdk-13.0.1\par>java Rsax.java  
Beginning Node :shops  
Beginning Node :supermarket  
Beginning Node :sid  
SID : 201  
LAsT Node:sid  
Beginning Node :sname  
Shop Name:  sparc  
LAsT Node:sname  
Beginning Node :product  
Available Product:  grocery  
LAsT Node:product  
Beginning Node :branch  
No.of Branches:  two  
LAsT Node:branch  
Beginning Node :location  
Address :  new york  
LAsT Node:location  
LAsT Node:supermarket  
Beginning Node :supermarket  
Beginning Node :sid  
SID : 540  
LAsT Node:sid  
Beginning Node :sname  
Shop Name:  big basket  
LAsT Node:sname  
Beginning Node :product  
Available Product:  grocery  
LAsT Node:product
```