XML

Hand Book

Dr. John T Mesia Dhas

Dr. T. S. Shiny Angel

The Palm Series

XML

Hand Book

Dr. John T Mesia Dhas

Dr. T. S. Shiny Angel

The Palm Series

Title: XML Hand Book

Author: Dr. John T Mesia Dhas, Dr. T. S. Shiny Angel

Publisher: Self-published by Dr. John T Mesia Dhas

Address of Publisher: No-1, MGR Street, Charles Nagar, Pattabiram

Chennai – 600072

India

Email: jtmdhasres@gmail.com

Printer: The Palm

Mogappair West

Chennai -600037

India

ISBN:

CHAPTER 1

INTRODUCTION

1.1 BACKGROUND

- XML stands for **E**xtensible **M**arkup **L**anguage and is a text-based markup language derived from Standard Generalized Markup Language (SGML). Although XML derived from SGML, XML has simplified the process of defining and using this metadata
- Extensible Markup Language (XML) is fast becoming a standard for data exchange in the next generation's Internet applications. XML allows user-defined tags that make XML document handling more flexible
- XML stores data in plain text format. This provides a software and hardware independent way of storing, transporting, and sharing data.
- XML is a syntax for expressing structured data in a text format
- XML is not a language on its own. Instead, XML is used to build markup languages.
- XML was designed to store and transport data, but not to display data.
- XML was designed to be both human- and machine-readable.
- XML is a markup language much like HTML. But it is not a replacement for HTML
- XML was designed to be self-descriptive
- XML is platform independent and language independent
- XML is a W3C Recommendation
- XML was originally called Web SGML, later renamed the Extensible Markup Language(XML)

1.2 CHARACTERISTICS

There are three important characteristics of XML that make it useful in a variety of systems and solutions:

- **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.

1.3 XML USAGE

An XML can be:

- Used to exchange the information between organizations and systems.
- Used for offloading and reloading of databases.
- Used to store and arrange the data, which can customize your data handling needs.
- used extensively as Configuration files in J2EE architectures and .Net architecture (***web.config*** – in Asp.Net, ***web.xml*** in Tomcat web server)
- Often used to separate data from presentation
- Easily merged with style sheets to create almost any desired output.
- Virtually, any type of data can be expressed as an XML document.
- In Visual Studio.Net, software installation is made easy by using xml. Software installation is simply done using xcopy. No more use of Windows registry
- XML is complement to HTML
- XML is alternate to relational databases
- XML is universal standard EDI format(Electronic Data Interchange format)
- Used for B2B transactions on the Web:
  - Electronic business orders (ebXML)
  - Financial Exchange (IFX)
  - Messaging exchange (SOAP)

1.4 INDUSTRIES USING XML

For exchanging data between organization in the Internet, there are thousands of XML formats exist, for many different industries, to describe day-to-day data transactions:

- Stocks and Shares
- Financial transactions
- Medical data
- Mathematical data
- Scientific measurements
- News information
- Weather services

1.5 ADVANTAGES OF XML OVER SGML

Although SGML provided a solution for exchanging data in a structured, standardized manner, it was suitable to exchange data between different branches of same organization that is scattered across different geographical locations, but it was inappropriate for direct application on the Internet.

The difference between SGML and XML are listed below:

SGML	XML
Used for exchanging data between branches within the same organization, but not suitable for exchanging information in the Internet	It is universal format for exchanging data between organization in the Internet
many SGML implementations require some DTD for processing	XML permits well-formed documents to be parsed without the need for a DTD
Syntax of SGML much complex than XML	Syntax of XML is much simpler and more permissive in its syntax than SGML
SGML Stands for Standard Generalized Markup Language	XML stands for Extensible Markup Language
XML documents should be readable with SGML parsers	Whereas, some SGML might produce errors in XML parsers

Other difference includes:

- XML is a subset of SGML
- XML is simpler compared to SGML
- A list of SGML declarations have been removed in XML
- Some constructs that are allowed in SGML are no longer permitted in XML
- Some SGML entities are no longer allowed in XML
- Some comment practices in SGML have also been disallowed in XML

1.6 ADVANTAGES OF XML OVER HTML

HTML was a pure-Internet approach for displaying and presenting information in a platform- independent manner, but it was wholly inadequate for representing data structures.
The difference between HTML and XML are listed below:

HTML	XML
HTML stands for Hypertext Markup Language	XML stands for Extensible Markup Language
HTML is intended for consumption by humans	XML is meant for both machine and human Consumption
It has no capability to represent metadata, provide validation, or support even the basic needs of e-business	It has capability to represent metadata, provide validation or support even the basic needs of e-business
Html is a presentation language	XML is neither a programming language nor a presentation language, but is data definition language

HTML	XML
HTML was designed to display data – with focus on how data should looks	XML was designed to carry data - with focus on what data is
HTML is used for designing a web-page to be rendered on the client side browsers, in a platform independent manner.	XML is used basically to transport data between the application and the database. It is universal standard for data exchange in the next generation's Internet applications
HTML tags are predefined, but you cannot define your own tag	XML tags are not predefined. You must define your own tags
HTML is case insensitive.	XML is case sensitive.
HTML is a markup language itself.	XML provides a framework for defining markup languages
HTML is not strict if the user does not use the closing tags	XML makes it mandatory for the user the close each tag that has been used
HTML does not preserve white space	XML preserves white space
Empty tag need not ends with /	Slash(/)required in empty tags
Attribute value need not enclosed in quotation	Attribute value must be enclosed In quotation
HTML is static	XML is dynamic

An XML-enabled version of HTML is known as XHTML, in which html tags should follows all XML syntax rules

1.7 ADVANTAGES OF XML OVER EDI

- Although, adaption of EDI is fairly widespread among larger-sized businesses, the cost of EDI implementation and ongoing maintenance can be measured in the billions in aggregate. Millions of dollars in transactions occur on a daily basis using EDI- mediated messages. X12/EDI will be fairly slow to adopt a new standard, which would necessitate new processing technology, mapping software, and back-end integration
- Compared to EDI and other electronic commerce and data-interchange standards, XML offers serious cost savings and efficiency enhancements that make implementation of XML good for the bottom line.
- The following components are used in electronic commerce systems for document exchange: document creation tools, processing components, validity checking, data mapping, back-end integration, access to a communications backbone, security, etc.
- XML greatly simplifies many of these steps. XML's built-in validity checking, low-cost parsers(DOM,SAX) and processing tools, Extensible Style sheet Language (XSLT) based mapping, and use of the Internet bring down much of the e-commerce chain cost
- Another drawback of EDI is that they don't support easily support the needs for internationalization and localization. It is difficult to represent information contained

in a non-Latin alphabet

- whereas, XML syntax allows for international characters that follow the Unicode standard to be included as content in any XML element

The difference between EDI and XML are listed below:

EDI	XML
EDI is a well-established technology for automating order processing and document interchange between computer applications.	XML is an emerging standard designed to simplify Web-based e-commerce transactions between computer applications.
EDI enables highly secure document exchanges.	XML documents typically need to encrypted to maintain security levels.
EDI documents are typically in compressed, machine-only readable form.	XML is an open human-readable, text format.
EDI documents are typically sent private and relatively expensive value- added networks (VANs).	XML documents are typically sent via the Internet - i.e. a relatively low-cost public network.
EDI traditionally requires customized mapping of each new trading partner's document format.	XML is designed to require one customized mapping per industry grouping, so most companies will be able to work to one format and use XML
EDI typically requires dedicated servers that cost from US$10,000 and up.	XML requires a reliable PC with an Internet connection.
EDI can involve high on-going transaction based costs keeping up the connection to the EDI network and keeping the servers up and running.	XML in Internet-based has low ongoing flat-rate costs using existing Internet connections and relatively low-cost Web Servers.
EDI-based transactions account for the bulk of value of goods and services exchanged electronically.	XML processes relatively low transaction values.
EDI is estimated to be limited to 300,000 companies worldwide and about 20% of their suppliers because of operational costs and complexity.	XML appears to have no upper limit in terms of numbers of users.
EDI was traditionally built from the ground up in semi-isolation without being able to share resources with other programs.	XML is being developed in a world of shared software development populated by many low-cost tools and open source projects

1.8 ADVANTAGES OF XML OVER DATABASES AND FLAT FILES

XML	Flat Files and Database
- XML is a structured document format that includes not only the data but also metadata that describes that data's content and context	- Most Flat files (text file) simply cannot offer this clear advantage. Common file exchange formats such as comma-delimited and tab-delimited text files merely contain data in predefined locations - Complex file format such as Microsoft Excel contain more structured information but are machine-readable, do not contain the level of structuring present in XML
- XML's structured document formats are text based	- Relational and object-oriented databases and formats can represent data as well as metadata, but their formats are not text based. - Most databases use a proprietary binary format to represent their information
- XML is universal standard for data exchange in the next generation's Internet applications	- Although text files can also be transmitted via e-mail and over the Web, structured formats such as relational and object-oriented databases are not easily accessible over the Internet due to their binary-based formats and proprietary connection mechanisms - gateway software and other mechanisms are needed to access these formats
- Processing tools for XML have become relatively widespread and inexpensive.	- Processing tools for flat file and database are custom, proprietary, or expensive - they are usually specific to the particular file format

1.9 DRAWBACKS TO XML

- XML takes up lots of space to represent data that could be similarly modeled using a binary format or a simpler text file format. The reason for this is simple: It's the price we pay for human-readable, platform-neutral, process-separated, metadata- enhanced, structured, validated code.
- XML documents can be 3 to 20 times as large as a comparable binary or alternate text file representation. It's possible that 1GB of database information can result in over 20GB of XML-encoded information
- Large XML documents may need to be loaded into memory before processing, and some XML documents can be gigabytes in size. This can result in sluggish processing, unnecessary reparsing of documents, and otherwise heavy system loads
- Much of the "stack" of protocols requires fairly heavy processing to make it work as intended

- In addition, a problem of many current XML parsers is that they read an entire XML document into memory before processing. This practice can be disastrous for XML documents of very large sizes
- Despite all the added value in representing data and metadata in a structured manner, some projects simply don't require the complexity that XML introduces. In these cases, simple text files do the job more efficiently
- Although XML does offer validation technology, it is not currently as sophisticated as many of the EDI syntax checkers. XML editors often lack the detail and helpfulness found in common EDI editors.

1.10 APPLICATION AREAS OF XML

XML is making it easier to conduct e-business and e-commerce, manage online content, work with distributed applications, communicate, and otherwise provide value.

- **E-Business and E-Commerce**
  - e-business practices include delivering information to customers via the Internet, implementing customer relationship management systems, and connecting branches together utilizing electronically distributed methods
  - E-commerce generally refers to the ability to perform a particular transaction with a customer in an electronic or online format. E-commerce is usually much smaller in scope and focused than overall e-business and usually implies a direct transaction between two parties. To make the distinction with e-business clear, buying a book online is considered an e-commerce transaction, whereas enabling the fulfillment and delivery of that book using electronic methods is considered e-business
  - One of the main uses of XML in e-business is the representation of the various business transactions that occur on a daily basis between partners in a trading process. This includes purchase orders, invoices, shipping, bills of lading, and warehousing information
  - One of the major steps in any e-business process is payment for services rendered or goods sold. Even in this area, XML is making a major impact. XML has been used to send payment information of all types, including credit cards, cash, vouchers, barter exchanges, and electronic funds transfers
  - XML is making waves in the area of security also. XML has been used for security specifications of all sorts, ranging from encryption and authorization to privacy

- **Content Management**
  - In Internet Era, any application or document can instantly be shared with others. This has led to the concept that all information or data can be

considered "content" that can be accessible and integrated with other systems. XML is being used to enable all forms of content management and application integration

- ❖ Content that formerly was locked into proprietary file formats. XML is now enabling this content to be searched, located, and integrated with applications. "Legacy" systems, such as Customer Relationship Management (CRM), Enterprise Resource Planning (ERP), accounting, finance, Human Resources (HR), and other systems, are now communicating with each other using XML

- ➢ **Web Services and Distributed Computing**

  - ❖ Distributed computing is the ability to distribute processing responsibilities and functions among machines on a local or wide area network
  - ❖ Programming functionality encapsulated within "objects" is exchanged via Remote Procedure Calls (RPCs)
  - ❖ Distributed computing, has been attempted through technologies such as the Component Object Model (COM) and CORBA
  - ❖ XML aims is to develop platform-neutral data-format through which applications running in the heterogeneous environment can communicate with each other
  - ❖ A Web service is not a Web site that a human reads, but for reading from other process running on different machine
  - ❖ A Web service is an interface that describes a collection of operations that are network accessible through standardized XML messaging
  - ❖ Web service is built on top of XML, SOAP, WSDL, UDDI In that XML is base for program to program communication

- ➢ **Peer-to-Peer Networking and Instant Messaging**

  - ❖ Individuals can quickly exchange messages, files, and other information with each other on an on-demand basis. Known as peer-to-peer networks (P2P), this "instant file sharing" technology
  - ❖ Instant messaging provides the ability to send messages to colleagues, friends, and business partners.
  - ❖ Instant messaging has spread to many different devices, ranging from desktop computers to cell phones, and has included such features as desktop application sharing, video conferencing, and voice communications
  - ❖ XML is quickly making its presence felt in both of these rapidly growing technology areas. Various XML specifications and protocols are being used to allow individuals and organizations to send instant messages, locate other users, and locate, exchange, and store files on peer-to-peer networks in an open and nonproprietary manner

- **Getting More Meaning out of the Web: The Semantic Web**
  - "The Semantic Web is an extension of the current Web in which information is given well-defined meaning, better enabling computers and people to work in cooperation.
  - The most practical of these implementations will help enable users to make better, more relevant searches.
  - The implications of the Semantic Web, made possible only through the use of XML

1.11 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.

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.

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

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.

XML simplifies Platform change

- Upgrading to new systems (hardware or software platforms), is always time consuming. Large amounts of data must be converted and incompatible data

is often lost.

- XML data is stored in text format. This makes it easier to expand or upgrade to new operating systems, new applications, or new browsers, without losing data.

XML increases data availability

- Different applications can access your data, not only in HTML pages, but also from XML data sources.
- With XML, your data can be available to all kinds of "reading machines" (Handheld computers, voice machines, news feeds, etc.), and make it more available for blind people, or people with other disabilities.

XML can be used to create new internet languages

- A lot of new Internet languages are created with XML. Here are some examples:
  - **XHTML**
  - **WSDL** for describing available web services
  - **WAP** and **WML** as markup languages for handheld devices
  - **RSS** languages for news feeds
  - **RDF** and **OWL** for describing resources and ontology
  - **SMIL** for describing multimedia for the web

1.12 XML RELATED TECHNOLOGIES

Here we have pointed out XML related technologies. There are following XML related technologies:

No.	Technology	Meaning	Description
1	**XHTML**	Extensible HTML	It is a clearer and stricter version of XML. It belongs to the family of XML markup languages. It was developed to make html more extensible and increase inter-operability with other data.
2	**XML DOM**	XML Document Object Model	It is a standard document model that is used to access and manipulate XML. It defines the XML file in tree structure.
3	**XSL it contain three parts:**	Extensible Style Sheet Language	i) It transforms XML into other formats, like HTML. ii) It is used for formatting XML to

No.	Technology	Meaning	Description
	i) XSLT (XSL Transform) **ii) XSL** **iii)XPath**		screen, paper etc. iii) It is a language to navigate
4	**XQuery**	XML Query Language	It is a XML based language which is used to query XML based data.
5	**DTD**	Document Type Definition	It is an standard which is used to define the legal elements in an XML document.
6	**XSD**	XML Schema Definition	It is an XML based alternative to DTD. It is used to describe the structure of an XML document.
7	**XLink**	XML Linking Language	xlink stands for XML linking language. This is a language for creating hyperlinks (external and internal links) in XML documents.
8	**XPointer**	XML Pointer Language	It is a system for addressing components of XML based internet media. It allows the xlink hyperlinks to point to more specific parts in the XML document.
9	**SOAP**	Simple Object Access Protocol	It is an acronym stands simple object access protocol. It is XML based protocol to let applications exchange information over http. in simple words you can say that it is protocol used for accessing web services.
10	**WSDL**	Web Services Description Languages	It is an XML based language to describe web services. It also describes the functionality offered by a web service.
11	**RDF**	Resource Description Framework	RDF is an XML based language to describe web resources. It is a standard model for data interchange on the web. It is used to describe the title, author, content and copyright information of a web page.

No.	Technology	Meaning	Description
12	**SVG**	Scalable Vector Graphics	It is an XML based vector image format for two-dimensional images. It defines graphics in XML format. It also supports animation.
13	**RSS**	Really Simple Syndication	RSS is a XML-based format to handle web content syndication. It is used for fast browsing for news and updates. It is generally used for news like sites.

CHAPTER 2

COMPONENTS OF XML

2.1 XML DOCUMENT STRUCTURE

- An XML document consists of a number of discrete components or sections. Although not all the sections of an XML document may be necessary, their use and inclusion helps to make for a well-structured XML document that can easily be transported between systems and devices.
- The major portions of an XML document include the following:
    - The XML declaration
    - The Document Type Declaration(DOCTYPE)
        - Used to define Schema / DTD definition / user-defined entity reference
    - Entity reference
    - The element data
    - The attribute data
    - The character data or XML content
        - PCDATA
        - CDATA
    - Comments

- The following diagram depicts the syntax rules to write different types of markup and text in an XML document.

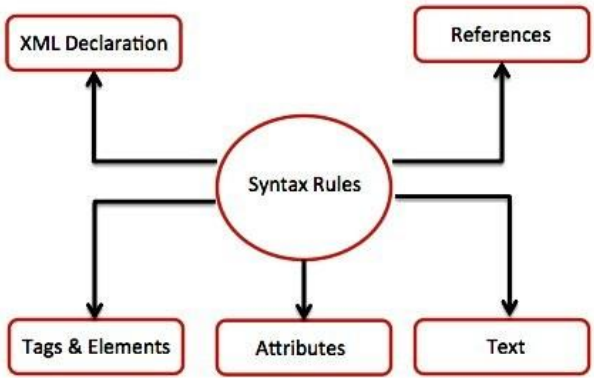
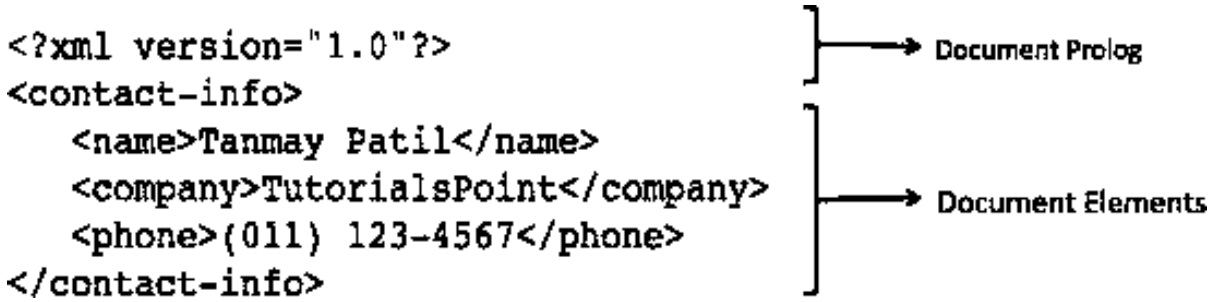


Fig. 2.1 XML Syntax Rules

- The following example describes the parts of XML document:



2.2 XML DECLARATION / PROLOG

- The first part of an XML document is the declaration. The XML declaration is a processing instruction that identifies the document as being XML
- Although it is not required, All XML documents should begin with an XML declaration.
- The syntax is given below:
  <?xml version="version number" encoding="encoding declaration" standalone="standalone status" ?>
- Example:
  <?xml version="1.0" encoding="UTF-8" standalone="no" ?>

- The following table shows a list of the possible attributes that may be used in the XML declaration.

Attribute Name	Possible Attribute Value	Attribute Description
version	1.0 , 1.1	• Specifies the version of the XML standard that the XML document conforms to. • The version attribute must be included if the XML declaration is declared. • In future it could be "2,0", etc
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	• Indicates the character encoding that the document uses. • The default is "US-ASCII". The most common alternate setting is "UTF-8"

Attribute Name	Possible Attribute Value	Attribute Description
standalone	yes, no	• Use 'yes' if the **XML document has** an **internal DTD**. • Use 'no' if the XML document is linked to an **external DTD**, or any external **entity** references

Rule:

- If the XML declaration is included, it must be situated at the first position of the first line in the XML document well-formedness constraint.
- If the XML declaration is included, it must contain the version number attribute

2.3 COMMENT

- XML comments are just like HTML comments. Comments are used to make codes more understandable other developers
- An XML comment should be written as:

  <!-- Write your comment -->

Rules for adding XML comments:

- Comments may be placed anywhere, but after the XML declaration
- You can use a comment anywhere in XML document except within attribute value.
- Nested comments are not allowed.

Example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!--Students marks are uploaded by months-->
<students>
  <student>
    <name>John</name>
    <marks>70</marks>
  </student>
```

<students>

2.4 DOCUMENT TYPE DECLARATION (DOCTYPE)

- This optional declaration may appear only once in an XML document
- A Document Type Declaration(DOCTYPE) gives a name to the XML content and identifies the internal content by specifying the root element
- A DOCTYPE can identify the constraints on the validity of the document by making a reference to an external DTD or include the DTD internally within the document (internal DTD)
- Although SGML requires a Document Type Declaration, XML has no restrictions of the sort
- The General Forms of the Document Type Declarations is listed below:

  <!DOCTYPE ROOT SYSTEM "file">
  <!DOCTYPE ROOT [ ]>

  <!DOCTYPE ROOT SYSTEM "file" [ ]>

  - The first form only allows use of an externally defined DTD subset
  - The second declaration only allows an internally defined DTD subset within the document
  - The third declaration provides a place for inclusion of an internally defined DTD subset between the square brackets while also making use of an external subset.

In the syntax:

- The keyword *ROOT* should be replaced with the actual root element contained in the document
- The keyword "file" should be replaced with a path to a valid DTD
- example:

  (i) <!DOCTYPE shirt SYSTEM "shirt.dtd">
  (ii) <!DOCTYPE book SYSTEM "DTDs/CWP.dtd">

- Here the DTD file is located in subdirectory "DTDs", under the directory where XML document is stored

The other forms of the Document Type Declarations is listed below:

<!DOCTYPE root PUBLIC FPI-identifier URL>
<!DOCTYPE root SYSTEM URL>

- **SYSTEM URL**
  - refers to a private DTD
  - Located on the local file system or HTTP server
- **PUBLIC URL**
  - refers to a DTD intended for public use
- example:

```
<!DOCTYPE book SYSTEM
"http://www.coreweb.com/DTDs/CWP.dtd">
```

- Here the DTD file is located at HTTP server: http://www.coreweb.com

2.5 XML ELEMENTS

- It is the Basic Unit of an XML. It is used to define content of the Xml document
- XML elements are either a **matched pair of XML tags**(**container Element**) or
- single XML tags(Empty Element) that are "self-closing"

(i) Container element

- Container elements are closed using following syntax:

```
<element-name [attribute1="value1" attribute2="value2" …] >
  ...

</element>
```

- A container Element can contain:
  - **text**
    - <custName>Dr. John</custName>
  - **attributes**
    - <book publisher="The Palm"></book>
  - **other elements**
    - <book>< publisher>The Palm</publisher></book>
  - **or a mix of the above**

```
<book publisher="The Palm">
        <title> Mobile Application Development</title>
</book>
```

(ii) Empty element

- Empty element must be closed as shown in the following syntax:
  <element [attribute1=”value1” attribute2=”value2” …] />
- it may contain attribute, but may not contain child element. an example is given below:
  <book publisher=" The Palm " />
- **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 correct nested tags:

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

Following example shows incorrect nested tags:

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

(iii) Root Element

- An XML document can have only one root element. The following example shows a correctly formed XML document:

```
<root>
  <x>...</x>
  <y>...</y>
 </root>
```

- 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>
```

Rules for defining XML Elements

- Element names must start with a letter or underscore
- The rest of the characters in the element-name can contain the following:

letters, digits, hyphens(-), underscores(_), and periods(.)

- Names of XML-elements are case sensitive. For example <contact-info> is different from <Contact-Info>.
- Start and end tags of an element must be identical
- A container element can contain text or other child-elements
- Element names cannot contain spaces
- Element names can be anything other than XML

2.6 ATTRIBUTES

- XML elements can have attributes. Attributes are used to add the information about the element. Attributes provide metadata for the element
- XML attributes enhance the properties of the elements.
- An attribute can be defined using name/value pair within an XML Element. An XML-element can have one or more attributes. For example:

```
<book category="Computer">
    <author> Dr. Shiny </author>
    <publication> The Palm </publication>
</book>
```

Note: In the above example, Metadata should be stored as attribute and data should be stored as element.

- **Rules for XML Attributes**
  - Attribute names in XML are case sensitive. That is, HREF and href are considered two different XML attributes
  - Same attribute cannot be declared more than once with in single element. The following example shows incorrect xml syntax because the attribute b is specified twice:

    .
  - An attribute value must always be quoted. We can use single or double quote. Following example demonstrates incorrect xml syntax, because attribute value is not enclosed in quotation:

    .

2.7 ENTITY REFERENCES

- An entity reference is a group of characters used in text as a substitute for a single specific character that is also a markup delimiter in XML. Using the entity reference prevents a literal character from being mistaken for a markup delimiter
- An Entity can be of two types: pre-defined, user-defined
- Entity references always begin with an ampersand (&) and end with a semicolon (;)

Pre-defined entity

- For example, if an attribute must contain a < symbol then you can substitute it with the entity reference "<" as shown in the following example:

```
<shirt price-range="&lt; Rs.500 and &gt;Rs.250 "
    color="Yellow" /> (or)
<shirt>
    <price-range>&lt; Rs.500 and &gt;Rs.250</price-range>
    < color>Yellow</color>
<shirt>
```

- It is typically used to represent special characters within the textual content of XML Element or Attributes, which may be:
    - markup delimiters such as <. >,",' (or)
    - Special symbols that is not present in the keyboard such as ©,®,¥,β,μ etc.
- The pre-defined entities in XML are listed in the following table.

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

- Alternatively, you can also substitute a predefined entity reference with either numeric reference(decimal character reference) or hexadecimal character reference as shown in the table, both together is simply referred to as character reference

Character References

- Another special form of entity reference is the character reference, which is

used to insert arbitrary Unicode characters into an XML document

- This allows international characters to be entered even if they can't be typed directly on a keyboard. Character entities use either decimal or hexadecimal references to describe their Unicode data values

(i) Decimal character reference (numeric reference)

- These contain references contains a hash mark ("#") followed by a number. The number always refers to the Unicode code of a character.
- for example Decimal character reference A refers to alphabet "A".

(ii) Hexadecimal character reference

- These contain references contains a hash mark ("#") followed character 'x' followed by a hexa-decimal number. The number always refers to the Unicode code of a character.
- For example Hexadecimal character reference A refers to alphabet "A".

User-defined entity

- The above table shows some predefined entities in XML. But you can also declare your own user-defined entities with in a DTD or XML Scheme. There are internal and external entities
- Each user-defined entity must have a unique name that is defined as part of an entity declaration in a DTD or XML Schema.
- User-defined entities are used to refer often repeated or varying text or to include the content of external files.
- For example, an entity &legal; can be replaced with an organization's legal disclaimer, consisting of any XML text that is included in the DTD or read from a file

Declaring entities

- You can declare entities in a DTD in the following format:

```
<!ENTITY name "text">
```

- Where name is the name of the entity and text is the referenced text that appears where the entity is used.
- This example declares an user-defined entity named 'COPYRIGHT' and uses it in the XML document:

```
<?xml version="1.0" standalone="yes" ?>
<!DOCTYPE book [<!ELEMENT book (title)>
                    <!ELEMENT title (#PCDATA)>
                    <!ENTITY COPYRIGHT "2001, Prentice Hall">]>
```

```
<book>
        <title>Core Web Programming, &COPYRIGHT;</title>
</book>
```

2.8 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:

Not allowed character	**Replacement - entity**	**Character description**
<	&lt;	less than
>	&gt;	greater than
&	&amp;	Ampersand
'	&apos;	Apostrophe
"	&quot;	quotation mark

- The Text value with in a container element, could be of two types:
  - **PCDATA –** data can contain any characters, except xml delimiters
  - **CDATA -** data can contain any characters, including xml delimiters
  -

PCDATA(Parsed Character Data) section

- PCDATA is text found between the start- tag and the end- tag of an XML element
- parsed character data should not contain any &, <, or > characters; these need to be represented by the & < and > entities, respectively.
- PCDATA is text that WILL be parsed by a parser. Tags inside the text will be treated as markup and entities will be expanded.

CDATA(Character Data) section or Marked CDATA section

- CDATA is text that will NOT be parsed by a parser. Tags inside the text will NOT be treated as markup and entities will not be expanded
- character data can contain any &, <, or > characters, or any special character
- CDATA sections are commonly used for storing scripting language content and sample XML and HTML content within the XML file
- The general syntax to include CDATA is:
  - <![CDATA[any characters (including markup) …]]>
- An example given below:

```
<?xml version="1.0"?>
<!DOCTYPE for-loop [ <!ELEMENT for-loop(syntax,example)>
                     <!ELEMENT syntax(#PCDATA)>
                      <!ELEMENT example(#PCDATA)>          ]>
<for-loop>
      <syntax>
            for(initialization;condition;increment){ Statement[s]; }
      </syntax>
      <example>
            <![CDATA[ while(i=0;i<10;i++) { printf("\n%d",i); }  ]]>
    </example>
</for-loop>
```

CDATA Rules:

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

2.9 PROCESSING INSTRUCTIONS

- Processing instructions (PIs) are used to embed application specific instructions into your xml documents
- It is commonly referred as PIs. PIs are not part of actual documents, which are passed up to the application
- Processing instructions have the following form:

  <? PI-Target processing-instructions ?>
- Where PI-Target – is the name of the application that is supposed to receive processing instructions
- A sample processing instruction is given below:

<?messageprocessor “process complete”?>

Example 2:

```
<Name nickname=”kaja”>
  <FirstName>kajendran</FirstName>
  <MiddleName /><!—kajendran missed his middle name in the fire -->
  <? Nameprocessor “select * from blob” ?>
  <LastName>krishnan</LastName>
</Name>
```

2.10 WELL FORMED XML DOCUMENTS

- A document is said to be well formed, only if it obeys the syntax of XML
- Any XML document said to be "Well Formed" XML document, if it satisfies the following syntax rules:

1. An XML document must contain only one root element and that must be the parent of all other elements

Correct	Incorrect
<?xml version="1.0"?> **<catalog>** <book> <title>C# complete reference</title> <price>$20</price> </book> <book> <title>XML step by step</title> <price>$30</price> </book> **</catalog>** Reason: contain only one root element	<?xml version="1.0"?> **<book>** <title>C# complete reference</title> <price>$20</price> **</book>** **<book>** <title>XML step by step</title> <price>$30</price> **</book>** Reason: contain more than one root element <book>

2. All XML Container Elements Must Have a Closing Tag

Correct	Incorrect
<?xml version="1.0"?> <name> <first-name>John</first-name> <last-name>Mesia Dhas</last-name> </name>	<?xml version="1.0"?> <name> <first-name>John <last-name>Mesia Dhas</last-name> </name> Reason: element <first-name> does not have closing tags

3. All XML Container Elements Must Have Proper Nesting

Correct	Incorrect
<?xml version="1.0"?> <address> <NAME> <first-name>John</first-name> <last-name>Mesia Dhas</last-name> </NAME> <steet>#1, MGR Street</street> <area>Charles Nagar</area> <city>Chennai-72</city> </address>	<?xml version="1.0"?> <address> <NAME> <first-name>John</NAME> <last-name>Mesia Dhas</last-name> </first-name> </address> **Reason:** Tags <name>& <first-name> are not properly nested

4. XML tags are case sensitive. In XML, the elements **<shirt>** and **<Shirt>** are different

Correct	Incorrect
<?xml version="1.0"?> <name> <first-name>John</first-name> <last-name>Mesia Dhas</last-name> </name>	<?xml version="1.0"?> <name> <First-Name>John</first-name> <last-name>Mesia Dhas</last-name> </name>

5. All XML Empty element must ends with /> (forward slash followed by greater than symbol)

Correct	Incorrect
<Name> <FirstName>John</FirstName> <MiddleName /> <LastName> Mesia Dhas</LastName> </Name>	(i) <MiddleName/ > (ii) <MiddleName / > **Reason:** An empty element must ends with />. But both (i) and (ii) contain space in between / and > symbol

6. XML attribute values must be quoted. Use either single quotes or double quotes

Correct	Incorrect
<book publisher="The Palm"> <title> Python 3.7.1</title> </book>	<book publisher= The Palm > <title> Python 3.7.1</title> </book> **Reason:** Attribute publisher's value is not quoted

7. No attribute may appear more than once on the same start-tag

Correct	Incorrect
<book category="computers”> <title> Python 3.7.1</title> </book>	<book category="computers” category="internet” > <title> Python 3.7.1</title> </book> **Reason:** Attribute category appears twice within same tag

8. Attribute Values Cannot Contain References to External Entities
9. All Entities Except amp, lt, gt, apos, and quot must be declared before they are used
10. A binary entity cannot be referenced in the flow of content, it can only be used in an attribute declared as ENTITY or ENTITIES
11. Neither text nor parameter entities are allowed to be recursive, directly or indirectly

CHAPTER 3

XML NAMESPACES

- XML Namespaces provide a mechanism to avoid element name conflicts and attribute name conflict
- The main purpose of namespace is to group elements and to differentiate an element from others with a similar name
- XML namespaces provide a simple method for qualifying element and attribute names used in XML documents by associating them with namespaces identified by URI references
- Namespaces are a simple and straightforward way to distinguish names used in XML documents, no matter where they come from
- Namespace is a mechanism by which element and attribute name can be assigned to group. The Namespace is identified by URI (Uniform Resource Identifiers).
- In an XML document, the URI is associated with a prefix, and this prefix is used with each element to indicate to which namespace the element belongs. For example: rdf:description xsl:template

In these examples,

- the part before the colon is the *prefix*
- the part after the colon is the *local part*
- any prefixed element is a *qualified name*
- any un-prefixed element is an *unqualified name*

3.1 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 in large XML- based distributed systems.
- For example, consider the following XML document that carries information about HTML table example:

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

```
</table>
```

- The following is an XML document that carries information about a table (a piece of furniture):

```
<table>
    <name>Indian Teak 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.

3.2 SOLVING THE NAME CONFLICT USING A PREFIX

- Name conflicts in XML can easily be avoided using a name prefix. The following 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>Indian Teak 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.

Drawback

- If prefix itself same for more than one xml document, then again name conflict occurs. To avoid this we can use xml namespace along with prefix

3.3 SOLVING NAME CONFLICT USING NAMESPACE DECLARATION

- The namespace can be defined by an **xmlns** attribute in the start tag of an element. The namespace declaration has the following syntax:

<element-name xmlns:*prefix*="*URI*">

Where

- The Namespace starts with the keyword **xmlns**.
- The word **prefix** is the Namespace prefix.
- The **URI** is the Namespace identifier.

➢ Consider the following example:

```
<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="http:/ www.pepperfry.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="http://
        www.pepperfry.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.
- Notice that the prefix f, h, has been added to both the start and the end tags and is followed by a colon and then the element's local name
- If you want the attributes in the document to be also in the namespace then you follow a similar procedure as shown in Listing:

```
<hr:ApplicationUsers
xlns:hr="http://wrox.com/namespaces/applications/hr/config">
        <user hr:firstName="Joe" hr:lastName="Fawcett" />
       <user hr:firstName="Danny" hr:lastName="Ayers" />
     <user hr:firstName="Catherine" hr:lastName="Middleton" />
</hr:applicationUsers>
```

- The namespace prefix is prepended to the attribute's name and followed by a colon

3.4 SCOPE OF NAMESPACE

- The namespace declaration must come either on the element that uses it or on one higher in the tree, an *ancestor* as it's often called
- The file content shown below is not well formed because the declaration is too far down the tree and therefore not in scope:

```
<hr:applicationUsers>
    <user
                xmlns:hr="http://wrox.com/namespaces/applications/hr/c
            onfig" firstName="Joe" lastName="Fawcett" />
     <user firstName="Danny" lastName="Ayers" />
    <user firstName="Catherine" lastName="Middleton" />
</hr:applicationUsers>
```

3.5 NAMING NAMESPACES

- XML namespace identifiers must conform to a specific syntax—the syntax for Uniform Resource Identifier (URI) references. This means that XML namespace identifiers must follow the generic syntax for URIs defined by RFC 2396.
- URI references are used to identify physical resources (Web pages, files to download, and so on), but in the case of XML namespaces, URI references

identify abstract resources, specifically, namespaces.

3.6 DEFAULT NAMESPACES

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

  <element-name xmlns="*URI*">
- This XML carries information about a piece of furniture:

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

3.7 DECLARING MORE THAN ONE NAMESPACE

Many XML documents use more than one namespace to group their elements. You have a number of choices when you need to design XML in this fashion

Option 1: Choose a default namespace for some elements and an explicit one for others.

- In the following example, we place the <*applicationUsers*> element in the hr namespace and the <*user*> elements themselves in a different one, which is used by
  <user> elements across all company documents
- *hr* namespace is created as the default and the entities namespace created as explicit. You need a prefix for the newer one so choose *ent*

```
<applicationUsers xmlns=”http://wrox.com/namespaces/hr”
                  xmlns:ent=”http://wrox.com/namespaces/entities”>
    <ent:user firstName=”John” lastName=” Mesia Dhas” />
    <ent:user firstName=”Sam” lastName=”Dhas” />
    <ent:user firstName=”Shiny” lastName=”Angel” />
    <user name=”Sheeba” />
</applicationUsers>
```

- Because both declarations are on the root element they are in scope for the whole of the document. Therefore any elements without a prefix fall into the *hr* namespace and any with an *ent* prefix fall into the entities namespace

Option 2: Make both namespace declarations explicit, to avoid a default namespace, because it makes which element is grouped under which namespace not clear. An example is given below:

```
<hr:applicationUsers       xmlns:hr=”http://wrox.com/namespaces/hr”
                           xmlns:ent=”http://wrox.com/namespaces/entities”>
        <ent:user firstName=”John” lastName=” Mesia Dhas” />
        <ent:user firstName=”Sam” lastName=”Dhas” />
        <ent:user firstName=”Shiny” lastName=”Angel” />
        <hr:user name=”Sheeba” />
</hr:applicationUsers>
```

Option 3: Declaring a namespace twice with different prefixes. An example is given below:

```
<hr1:applicationUsers xmlns:hr1=”http://wrox.com/namespaces/hr”
                xmlns:hr2=”http://wrox.com/namespaces/hr”>
                <hr2:user firstName=”John” lastName=” Mesia Dhas” />
                <hr2:user firstName=”Sam” lastName=”Dhas” />
                <hr2:user firstName=”Shiny” lastName=”Angel” />
</hr1:applicationUsers>
```

3.8 INVALID NAMESPACE

- Note: As shown in the above example, there are more than one prefix can point to the same namespace, it is well formed(or namespace-well-formed)
- But you cannot have the same prefix pointing to different namespace URIs, as shown below:

```
<hr:applicationUsers       xmlns:hr=”http://wrox.com/namespaces/hr”
                           xmlns:hr=”http://wrox.com/namespaces/entities”>
        <hr:user firstName=”John” lastName=” Mesia Dhas” />
        <hr:user firstName=”Sam” lastName=”Dhas” />
        <hr:user firstName=”Shiny” lastName=”Angel” />
</hr:applicationUsers>
```

- The above example is not *namespace–well-formed*.

3.9. UNDECLARING NAMESPACE MAPPING

- Sometime, you want to undeclare a namespace mapping completely.

- You can do this depends on if it's a default mapping and which version of the XML Namespaces recommendation you are using. Currently there are versions 1.0 and 1.1

Undeclaring default namespace

- The **default mapping can be undeclared** in all xml versions. To do this you just need to use an empty namespace URI in the child elements as shown in the following example:

```
<config xmlns="http://wrox.com/namespaces/hr/config">
  <applicationUsers xmlns="">
        <user firstName="John" lastName=" Mesia Dhas" />
        <user firstName="Sam" lastName="Dhas" />
        <user firstName="Shiny" lastName="Angel" />
  </applicationUsers>
</config>
```

- In this variation the<config> element is in the http://wrox.com/namespaces/hr/config namespace, but the other elements are not in any namespace (otherwise known as being in the empty or null namespace). This is because the xmlns="" on the <applicationUsers> element

Undeclaring explicit namespace or undeclaring namespace with prefix

- Undeclaring the namespace with prefix can be done only in version 1.1 xml documents
- The following example shows an XML document that declares the correct version and then maps and unmaps a namespace to a prefix:

```
<?xml version="1.1" ?>
<hr:config
xmlns:hr="http://wrox.com/namespaces/applications/hr/config">
<applicationUsers xmlns:hr=»»>
      <user firstName="John" lastName=" Mesia Dhas" />
      <user firstName="Sam" lastName="Dhas" />
      <user firstName="Shiny" lastName="Angel" />
</applicationUsers>
</hr:config>
```

- Here the ***hr*** prefix is mapped to a namespace URI on the <config> element and then unmapped on the <applicationUsers> element. This means that it would be illegal to try to use the prefix from this point.

CHAPTER 4

URL, URI, AND URN

4.1 UNIFORM RESOURCE LOCATOR (URL)

- URL specifies the location of a resource, for example a web page, and how it can be retrieved. It has the following format:
  - [Scheme]://[Domain]:[Port]/[Path]?[QueryString]#[FragmentId]
- The terms in square brackets are replaced by their actual values and the rest of the items other than Scheme and Domain are optional. So, a typical web URL would be
  - http://www.wrox.com/remtitle.cgi?isbn=0470114878
- The scheme is HTTP, the domain is www.wrox.com, followed by the path and a querystring. You can use many other schemes, such as FTP and HTTPS, but the main point about URLs is that they enable you to locate a resource, whether that is a web page, a file, or something else.

4.2 UNIFORM RESOURCE IDENTIFIER (URI)

- URI is just a unique string that identifies an Internet Resource or something else.
- According to the URI specification, there are two general forms of URI:
  - Uniform Resource Locators (URL) and
  - Uniform Resource Names (URN)
- Either type of URI may be used as a namespace identifier.

4.3 UNIVERSAL RESOURCE NAME (URN)

- Another, not so common type of URI is the Universal Resource Name (URN).
- A URN is a name that uniquely defines something. URNs take the following format:
  - urn:[namespace identifier]:[namespace specific string]

- The items in square brackets need to be replaced by actual values and the three- character prefix, urn, is not case-sensitive
- An example of a URN would be:

urn:isbn:9780470114872

- This URN uniquely identifi es the fourth edition of this book, but because it's a URN, not a URL, it doesn't tell you anything about how to retrieve either the book itself or any information about it
- Here are a few examples of URNs that could also be used as namespace identifiers:

  urn:www-develop-com:student
  urn:www.ed.gov:elementary.students
  urn:uuid:E7F73B13-05FE-44ec-81CE-F898C4A6CDB4

4.4 XML 1.1 NEW FEATURES

(i) Unicode character set support

- XML1.0 documents will be limited to the character set defined in Unicode 2.0, but XML1.1 document theoretically should handle any Unicode from 2.0 to the current 3.2 and beyond

(ii) Character Data formats

- In XML1.1, the Character Data should be resolved into one of five formats:
  - CData
  - CharData
  - Content
  - name
  - nmtoken

(iii) XML 1.1 Line-end characters

- XML 1.1 has the capability to handle line-end characters generated in IBM mainframe file format. This is very useful feature for sharing XML document across ASCII and EBCDIC –based platform
- XML 1.1 parsers are required to recognize and accept EBCDIC line-end characters(#x85) and Unicode line separator (#x2028). These values should be converted to one of XML 1.0 line-end characters:linefeed(declimal 10,#xA), or carriage return (declimal 13,#xD)

Example

- If you want to hard-code a carriage return in an XML1.0 documents, the following hex character reference can be used:

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

**<LineEndExample>first line
second line</LineEndExample>**

- When parsed, the result will look like the following:
  <?xml version=”1.0” encoding=”UTF-8” ?>
  <LineEndExample>first line

  second line</LineEndExample>

- In an XML1.1 documents, you could also hard-code an EBCDIC value to be used on IBM mainframe system as shown in the following example
  <?xml version=”1.0” encoding=”UTF-8” ?>
  <LineEndExample>first line &x85; IBM</LineEndExample>

- When parsed, the result will look like the following:
  <?xml version=”1.0” encoding=”UTF-8” ?>
  <LineEndExample>first line IBM </LineEndExample>
- When this code is parsed on non-IBM mainframe systemsm, the line-end should be replaced with an XML1.0 ASCII value

(iv) Undeclaring previously declared namespace with prefix

- Undeclaring the **namespace with prefix** can be done only in version 1.1 XML documents, but not in version 1.0 XML documents
- Default namespace can be undeclared in both version 1.0 and 1.1

(v) IRIs (Internationalized Resource Identifiers)

- Another change made in version 1.1 is that namespace URIs (Uniform Resource Identifiers) are now officially referred as IRIs (Internationalized Resource Identifiers)
- IRIs can contain characters from sets other than the basic ASCII ones. You may have noticed that web addresses sometimes have characters that need to be escaped using the %xx format, where xx represents the Unicode code point for the character
- With IRIs these characters can be used directly so, for example, a Russian website can use characters from the Cyrillic alphabet in its page names

CHAPTER 5

DOCUMENT TYPE DEFINITION

5.1 VALIDATING XML DOCUMENTS

- XML documents are mainly used to exchange information between program to program over the network. So that before transmitting XML documents from one program to another in the network, we want to make ensure it is suitable for processing in the remote machine, which reduce network traffic by avoiding repeated transmission of same xml documents due to invalid data.
- Validating xml documents means that the elements of the xml documents are in proper order, values of the elements and attributes are valid
- Validation of xml document can be done using:
    - DTD (Document Type Definition)
    - XML Schema
- DTD and schema are actually two different ways to specify the rules about the contents of an XML document.

5.2 WELL FORMED VS VALID XML DOCUMENTS

- By definition, if a xml document is not well-formed, it is not XML documents.

Well Formed XML document	Valid XML Document
An XML document is said to be well formed, only if it obeys the syntax of XML	A well-formed XML document is said to be valid, if it conforms to the constraints defined in a DTD or Schema
A "well-formed" XML document need not required to be "valid"	But a "valid" document is always "well- formed"

5.3 DTD (DOCUMENT TYPE DEFINITION)

- A DTD defines the structure and the legal elements and attributes of an XML document.
- The purpose of a DTD is to define the legal building blocks of an XML document. It defines the document structure with a list of legal elements
- DTD is used to validate the following in the XML document:
    - What are the elements allowed in the XML document?
    - In what order/sequence the elements should appear in the XML

document
- What are attributes of every element, and whether it is optional or required
- What is the relation between elements?

Why Use a DTD?

- With a DTD, independent groups of people can agree on a standard DTD for interchanging data.
- Your application can use a standard DTD to verify that data that you receive from the outside world is valid.
- You can also use a DTD to verify your own data.

5.4 DOCTYPE (DOCUMENT TYPE DECLARATION)

- The Document Type Declaration, or DOCTYPE, informs the parser that your document should conform to a DTD
- It also indicates where the parser can find the rest of the definition
- Internal DTD is embedded within the xml document through <!DOCTYPE>
- External DTD is linked with XML document through <!DOTTYPE>
- The generic structure of DOCTYPE declaration is shown below:

<!DOCTYPE ROOT-ELEMENT-NAME SYSTEM|PUBLIC "location1" "location2" […]>

5.5 TYPES OF DTD

- There are two ways of associating a DTD with an XML document. They are:
  - **Internal DTD -** A DTD can be declared inline in your XML document
  - **External DTD-** An XML document refers to external DTD

DOCTYPE with an Internal DTD Definition

- The simplest form of DOCTYPE declaration is given below, which is used to define internal DTD:

  <!DOCTYPE ROOT-ELEMENT-NAME [ ]>

An example of internal DTD is given below

```
<?xml version="1.0"?>
<!DOCTYPE name [ <!ELEMENT name (first, middle, last)>
                 <!ELEMENT first (#PCDATA)>
```

```
                    <!ELEMENT middle (#PCDATA)>
                    <!ELEMENT last (#PCDATA)>
                ]>
<name>
<first>John</first>
<middle>Fitzgerald Johansen</middle>
<last>Doe</last>
</name>
```

DOCTYPE with an External DTD Definition

- For linking external DTD with XML Document, you must use either SYSTEM or PUBLIC in the *DOCTYPE* declaration
- **SYSTEM identifier**
  - refers to a private DTD used by single organization
  - Located on the local file system or HTTP server

- **PUBLIC identifier**
  - refers to a DTD intended for public use, and shared by multiple organization

System Identifiers

- A system identifier allows you to specify the location of an external file containing DTD declarations. It is comprised of two parts:
  - the keyword SYSTEM and
  - a URI reference pointing to the document's location.
- A URI can be a file on your local hard drive, a file on your intranet or network, or even a file available on the Internet
- Syntax:

  <!DOCTYPE ROOT-ELEMENT-NAME "DTD-file-name">
- where DTD-*file-name* is the file with *.dtd* extension
- The following example uses system identifiers:

  <!DOCTYPE name SYSTEM "file:///c:/name.dtd" [ ]>

  <!DOCTYPE name SYSTEM "http://wiley.com/hr/name.dtd" [ ]>

  <!DOCTYPE name SYSTEM "name.dtd">

- Specifying an internal subset is optional.
- An XML document might conform to a DTD that uses only an internal subset, only an external subset, or both. If you do specify an internal subset, it appears between the [and], immediately following the system identifier

Public Identifiers

- *Public identifiers* provide a second mechanism to locate DTD resources. Its syntax:

  <!DOCTYPE root PUBLIC FPI-identifier URL>
- Using PUBLIC keyword allows non-specific reference to the DTD via a URL, even perhaps via secondary URL
- Example: BookCatalog DTD could become a well-known publishing industry standard, in which case we might refers to it using the following declaration:

  <!DOCTYPE BookCatalog "-//publishingconsortium//bookcatalog"
  " http://www.wrox.com/dtd/bookcatalog.dtd">

- In this example, the xml application would have more flexibility in locating the DTD.
- If the parser or application cannot locate the DTD using primary (PUBLIC) location, the second (SYSTEM) location is used. Note that SYSTEM keyword is implied
- According to the XML specification, public identifiers can follow any format; however, a commonly used format is called Formal Public Identifiers (FPIs).
- The syntax for FPIs matches the following basic structure:

  -//Owner//Class Description//Language//Version
- At the most basic level, public identifiers function similarly to namespace names, but public identifiers cannot be used to combine two different vocabularies in the same document. This makes namespaces much more powerful than public identifiers

An example of External DTD is given below

(i) The first step is to define external DTD file called "address.dtd". Its content is shown below:

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

(ii) Next, create an "address.xml" file and link the "address.dtd" as shown below:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE address SYSTEM "address.dtd">
<address>
```

```
  <name>Dr. John</name>
  <company>The Palm</company>
  <phone> (044) 123-4567</phone>
</address>
```

Combined DTD: Both Internal and External Together

- You can use both an internal DTD and an external one at the same time. This could be useful if you need to adhere to a common DTD, but also need to define your own definitions locally.
- This is an example of using both an external DTD and an internal one for the same XML document. The external DTD resides in "tutorials.dtd" and is called first in the DOCTYPE declaration. The internal DTD follows the external one but still resides within the DOCTYPE declaration:
- Here, I've added a new element called "summary". This element must be present under the "tutorial" element. Because this element hasn't been defined in the external DTD, I need to define it internally. Once again, we're setting the "standalone" attribute to "no" because we rely on an external resource.

tutorials.dtd

```
<!ELEMENT tutorials (tutorial)+>
<!ELEMENT tutorial (name, url)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT url (#PCDATA)>
<!ATTLIST tutorials type CDATA #REQUIRED>
```

tutorials.xml

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE tutorials SYSTEM "tutorials.dtd"
                                    [ <!ELEMENT tutorial (summary)>
                                      <!ELEMENT summary
                                      (#PCDATA)>
                                    ]>
<tutorials>
   <tutorial>
      <summary>Best XML tutorial on the web! </summary>
   </tutorial>
   <tutorial>
      <summary>Best HTML tutorial on the web! </summary>
   </tutorial>
</tutorials>
```

5.6 BASIC UNITS OF DTD

There are four basic keywords used in DTD declaration, which is listed below:

- Element declarations (ELEMENT)
- Attribute declarations (ATTLIST)
- Entity declarations (ENTITY)
- Notation declaration (NOTATION)

Keyword	Description
ELEMENT	Declare an XML element type name and its permissible sub elements(children)
ATTLIST	Declare an XML element's attribute names, plus permissible and/or default attribute values
ENTITY	Declare special character reference or text macro (similar to C/C++ #define statement) and other repetitive content from external source (similar to C/C++ #include statement)
NOTATION	Declares external non-XML content (example: binary image data) and external application that handle the content

Element Declarations

- When using a DTD to define the content of an XML document, you must declare each element that appears within the document.
- An element declaration can have one of two different forms:

  <!ELEMENT element-name category>
  - used for categories: EMPTY, ANY

  <!ELEMENT element-name (element-content-model)>
  - used for categories: #PCDATA, element, mixed
- An element's content model defines the allowable content within the element. An element may contain element children(sub-elements/child-elements), text, a combination of children and text, or the element may be empty.
- As far as the XML Recommendation is concerned, there are five categories of element content as listed below:

EMPTY	Element type may not contain any text or child elements, only attributes are permitted
ANY	Element type may contain any well-formed XML data. This could include character data
#PCDATA	Element type can contain text (character data) only
Element	Element type contain only child elements, but no additional text is permitted within this element type
Mixed	Element type may not contain text or child elements

- Let's look at each of these content models in more detail.

EMPTY

- Element declared as EMPTY cannot contain anything, except attributes. It means that the element neither contain textual data nor contain child element within it. Such as elements are known as empty elements

  Syntax: <!ELEMENT element-name EMPTY>
  Example: <!ELEMENT eof EMPTY>
  Valid xml: <eof />
    - It could be alternatively written as <eof></eof>
  Invalid xml: <eof>end of file</eof>

ANY

- Element declared as ANY can either contain textual data or contains child element within it or combination of both

  Syntax: <!ELEMENT element-name ANY>
  Example: <!ELEMENT PersonName ANY>

  Valid xml:

  (i) <PersonName> John T Mesia Dhas </PersonName>
  (ii)
  ```
  <PersonName>
      John T Mesia Dhas
     <father>Thanka Nadar</father>
  </PersonName>
  ```
  (iii)
  ```
  <PersonName> <Dr/>
        <FirstName> John T </FirstName>
        <LastName> Mesia Dhas </LastName>
  </PersonName>
  ```

 # PCDATA (Text only)

- Element declared as #PCDATA can only contain text data or character data. The text contents of an element should not contain any special characters such as <,>, &, ', " etc.

  Syntax: <!ELEMENT element-name (#PCDATA)> Example:
  <!ELEMENT name (#PCDATA)>

Valid xml: <name> John</name>

Invalid xml: <name> John < Mesia Dhas </name>

Element only

- Element declared as element content can only contain other(child) elements, but it cannot contain parsed character data
- When defining a content model with element content, you simply include the allowable child elements within parentheses separated by comma
- Here
  - The child elements are constrained to appear in specific sequence (sequence list)
  - Child elements are also constrained to choose only child elements among several choice (choice list)
  - The number of occurrences of child elements may be specified by cardinality operators

Syntax:

<!ELEMENT element-name (child-element1, child- element2,…)>

Example:

```
<!ELEMENT address (name, street, area, city, pincode)>
<!ELEMENT names (#PCDATA)>
<!ELEMENT street (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT pincode (#PCDATA)>
```

Valid xml:

```
<address>
        <name>Dr. John</name>
        <street>1, MGR Street </street>
        <area>Charles Nagar</area>
        <city>Chennai</city>
        <pincode>600072</pincode>
</address>
```

Invalid xml:

```
<address>
        <name>Dr. John</name>
        <street>1, MGR Street </street>
        <area>Charles Nagar</area>
        <pincode>600072</pincode>
        <city>Chennai</city>
</address>
```

Reason: element <pincode> must only appear after the element <city>

Mixed

- Element declared as mixed content can contain both text data as well as child elements, or any other combination
- In mixed content model, child elements are constrained to character data plus a simple list of valid child element types, and child element must be defined with our any sequence or choice specification further

Example:

```
<!ELEMENT PersonName (#PCDATA, FatherName)>
```

- Valid xml:

```
<PersonName>
   John
   <FatherName>Thanka Nadar</FatherName>
</PersonName>
```

Operators Used with Content-Models

- Operators are used impose more restriction on the content of an elements. DTD includes the following operators:
  - List operators(sequence-list(,), choice(|), ( ))
  - Cardinality operators (?, *, +)
- These operators are also called content model operators

List operators

There are two content model list operators. They are listed below:

Operator	**Syntax**	**Description**
, (comma)	**a,b**	- a followed by b Sequence: all the child elements must appear in the specified order
Syntax: *<!ELEMENT element-name (child-element1, child-element2,)>*		
| (vertical bar)	***a* | *b***	- either *a* or *b* - Choice: only one of the several child elements in the list is permitted
Syntax: <!ELEMENT element-name (child1 | child2 | …)>		
( )	**(expression)**	An expression surrounded by parentheses is treated as a unit and could have any one of the following suffixes?, *, or +.

Example: The following example demonstrate feature of all the three list operators

Let us look at an example of a simple five-element list for person's name could be declared as:

```
<!ELEMENT PersonName ((Mr | Ms | Dr), FirstName, MiddleName, LastName,
(Jr | Sr )) >
<!ELEMENT Mr       EMPTY   >
<!ELEMENT Ms       EMPTY   >
<!ELEMENT Dr       EMPTY   >
 <!ELEMENT FirstName       (#PCDATA)  >
 <!ELEMENT MiddleName      (#PCDATA)  >
 <!ELEMENT LastName        (#PCDATA)  >
<!ELEMENT Jr       EMPTY   >
<!ELEMENT Sr       EMPTY   >
```

- In the above example, PersonName consist of five child-elements in a specific sequence. But the two of its child elements are derived from a list of mutually exclusive choice
- But there are only limited number values are available for Title and Suffix. So that we could use choice list of empty elements to replace those two text-containing elements
- Conforming XML document instance would now become:

```
<PersonName>
      <Mr />
      <FirstName>John<FirstName>
      <MiddleName>T<MiddleName>
      <LastName>Mesia Dhas</LastName>
      <Sr />
</PersonName>
```

Cardinality operator

- Document can be controlled using the cardinality operators The number of times a child XML Element appears within the XML.

Operator	Syntax	Description
[none]	A	If no cardinality operator character is used then it indicates that the element must appear once and only once. This is the default behavior for elements used in content models(required)
?	a?	Indicates that the element may appear either zero or one time only – optionally singular element
+	a+	Indicates that the element may appear one or more times – optional elements
*	a*	Indicates that the element may appear zero or more times. –required elements

Example

- Let us look at s to use some cardinality operators.
  ELEMENT Address (name, street, area, city, mobile? email*) >
  <!ELEMENT name (#PCDATA) >
  <!ELEMENT street (#PCDATA)>
  <!ELEMENT city (#PCDATA)>
  <!ELEMENT mobile (#PCDATA) >
  <!ELEMENT email (#PCDATA)>

- In the above example, <***AddressList***> may contain one or more <***Address***> as its child element
- *Each* <**Address**> *element may contain* ***seven child elements*** *in the given sequence, in that* <**mobile**> *and* <***email***> *are optional*
- <***mobile***> can appear zero or one time only
- <**email**> can appear zero or more times
- *Conforming XML document instance would now become:*

```
<AddressList>
    <Address>
        <name>Dr. John <name>
        <street>1, MGR Street</street>
        <area>Charles Nagar</area>
        <city>chennai</city>
        <email>jtmdhasres@gmail.com</email>
        <email>jtmdhas@ymail.com</email>
    </Address>
    <Address>
        <name>Dr. Shiny<name>
        <street>10, Kannadasan Street </street>
        <area>Charles Nagar </area>
        <city>chennai</city>
        <mobile>9962143526</mobile>
    </Address>
</AddressList>
```

Attribute (ATTLIST) declaration

- In DTD, ***XML*** element attributes are declared with an ATTLIST declaration. Attribute declaration has the following syntax:

<!ATTLIST elementName attribName attribType attribDefault default-value>

- Were
  - ***elementName –*** is the name of the element to which the attribute belongs to

- ***attribName* –** is the name of the attribute
- ***attribType* –** possible values for ***attribType*** are listed in the following table

Type	Description
CDATA	- Indicates that the value of attribute is character data (text string)
ID	- Indicates that the value of attribute must uniquely identify the containing element. - This must be the text string that conforms to all XML name rules. It is similar to primary key in database
IDREF	- Indicates that the value of attribute is a reference to ID of another element
IDREFS	- Indicates that the value of attribute is a list of ID of another element
ENTITY	- Indicates that the value of attribute is the name of a predefined entity - The unparsed entity might be an image file or some other external resource such as an MP3 or some other binary file
ENTITIES	- Indicates that the attribute value is a whitespace-separated list of ENTITY values
NMTOKEN	- Indicates that the value of attribute is a valid XML name or name token A name token that conforms to the xml name rules
NMTOKENS	- Indicates that the value of attribute is list of **NMTOKENs** delimited by whitespace
(en1|en2|...) Enumerated List	- Attribute value must be one of a series that is explicitly defined in the DTD
NOTATION	An attribute value must be a **notation type** that is explicitly declared elsewhere in the DTD

- ***attribDefault(attribute default)* –** *ATTLIST* uses the attribute default parameter to indicate whether or not attribute's presence is required, and if it is not required , how parser should handle its absence. There are four different attribute defaults listed below:

Value	Explanation
default-value	- The default value of the attribute - The attribute may or may not appear in the instance of the element - If does not appear, the parser may supply default value specified in the ATTLIST declaration

Syntax

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

DTD example

```
<!ELEMENT employee (name, designation)>
<!ELEMENT name  (#PCDATA)>
<!ELEMENT designation  (#PCDATA)>
<!ATTLIST employee department CDATA
"MCA">
```

Valid XML conforms to DTD

(i) In the following instance, attribute department is not specified, so parser supply attribute ***department*** value as **"CSE"**

```
<employee>
<name>Dr. John</name>
<designation>Associate Professor</designation>
</employee>
```

(ii) In the following instance, attribute department is specified, so default-value **"CSE"** is ignored

```
<employee department="MBA">
<name>Vijay</name>
<designation>Professor</designation>
 </employee>
```

#REQUIRED	- The attribute is required - The attribute must appear in every instance of the element

Syntax

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

DTD example

```
<!ELEMENT person  EMPTY>
<!ATTLIST person number CDATA  #REQUIRED>
```

Valid XML conforms to DTD

```
<person number="5677" />
```

Invalid XML

```
<person />
```

#IMPLIED	- **The attribute is optional**. The attribute may or may not appear in the instance of the element

Syntax

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

DTD example

```
<!ELEMENT contact EMPTY>
<!ATTLIST contact f a x CDATA #IMPLIED>
```

Valid XML conforms to DTD

```
<contact fax="044-284259" />
<contact />
```

#FIXED *value*	- **The attribute value is fixed** - The attribute may or may not appear in the instance of the element. If the attribute does appear, its value must match with those in the ATTLIST declaration

Syntax

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

DTD example:

```
<!ELEMENT company EMPTY>
<!ATTLIST sender company CDATA #FIXED "Microsoft">
```

Valid XML

```
<sender company="The Palm" />
```

Invalid XML

```
<sender company="Sheeba Architects" />
```

(i) DTD Attribute Type : CDATA

- If attribute value is nothing more than plain text, then the attribute is declared using CDATA type

DTD Example

```
<!ELEMENT PersonName EMPTY>
<!ATTLIST PersonName firstname CDATA #REQUIRED
                     middlename CDATA #IMPLIED
                     lastname CDATA #REQUIRED>
```

Valid XML

```
<PersonName firstname="John" lastname="Mesia Dhas" />
```

(ii) DTD Attribute Type: Enumerate values

- If an attribute has only limited set of values, then the attribute is declared using enumerated list
- In the XML instance, the attribute value must exactly match any one of the value in the enumerated list declared in the corresponding DTD

Syntax

<!ATTLIST element-name attribute-name (value1 | value2 | …) >

Example

<!ELEMENT PersonName (FirstName,LastName) >
<!ELEMENT FirstName (#PCDATA) >
<!ELEMENT LastName (#PCDATA) >
<!ATTLIST PersonName honorific (Mr | Ms | Dr |Rev) #IMPLIED
suffix (Jr | Sr) #IMPLIED >

Valid XML

<PersonName honorific=”Dr” suffix=”Jr”>
<FirstName>John T</FirstName>
<LastName>Mesia Dhas</LastName>
</PersonName>

(iii) DTD Attribute Type : ID

- The attribute type of ID is used specifically to identify elements. Because of this, no two elements can contain the same value for attributes of type ID

Syntax

<!ATTLIST element_name attribute_name ID default_value>

Example

<!ELEMENT mountains (mountain+) >
<!ELEMENT mountain (name) >
<!ATTLIST mountain mountain_id ID #REQUIRED>

The following XML document would be valid, as it conforms to the above DTD:

```
<mountains>
    <mountain mountain_id="m10001">
    <name>Mount Cook</name>
</mountain>
    <mountain  mountain_id="m10002">
    <name>Cradle Mountain</name>
</mountain>
</mountains>
```

Invalid XML - The following XML document would be invalid because the value of the "mountain_id" attribute is the same for both elements:

```
<mountains>
  <mountain mountain_id="m10001">
    <name>Mount Cook</name>
  </mountain>
```

```
<mountain mountain_id="m10001">
  <name>Cradle Mountain</name>
</mountain>
</mountains>
```

(iv) DTD Attribute Type : IDREF

The attribute type of IDREF is used for referring to an ID value of another element in the document.

Syntax

```
<!ATTLIST element_name attribute_name IDREF default_value>
```

Example

```
<!ELEMENT employees (employee+) >
<!ELEMENT employee (name) >
<!ATTLIST employee employee_id ID #REQUIRED
                   manager_id IDREF #IMPLIED >
```

Valid XML - The following XML document would be valid, as it conforms to the above DTD:

```
<employees>
<employee employee_id="e10001">
<name>Adersh</name>
  </employee>
  <employee employee_id="e10002" manager_id="e10001">
    <name>Suresh</name>
  </employee>
</employees>
```

Invalid XML - The following XML document would be invalid. This is because the "manager_id" attribute of the second element contains a value that isn't the same as a value of another element that contains an attribute with a type of ID:

```
< employees>
<employee employee_id="e10001">
<name>K.Jeyachendran</name>
  </employee>
  <employee employee_id="e10002" manager_id="e10003">
    <name>N.Krishnan</name>
  </employee>
</employees>
```

(v) DTD Attribute Types : IDREFS

The attribute type of IDREFS is used for referring to the ID values of more than one other element in the document. Each value is separated by a space.

Syntax:

<!ATTLIST element_name attribute_name IDREFS default_value>

Example:

```
<!ELEMENT individuals (individual+) >
<!ELEMENT individual (name) >
<!ATTLIST individual individual_id  ID  #REQUIRED
                     parent_id     IDREFS #IMPLIED   >
```

Valid XML - The following XML document would be valid, as it conforms to the above DTD

```
<individuals>
    <individual individual_id="e10001">
        <name>Adesh </name>
    </individual>
    <individual individual_id="e10002">
        <name>Suresh</name>
    </individual>
    <individual individual_id="e10003" parent_id="e10001 e10002">
        <name>Ramesh</name>
    </individual>
</individuals>
```

(vi) DTD Attribute Types : NMTOKEN

- An **NMTOKEN (name token)** is any combination of Name characters. It cannot contain whitespace (although leading or trailing whitespace will be trimmed/ignored
- While Names have restrictions on the initial character (the first character of a Name cannot include digits, diacritics, the full stop and the hyphen), the NMTOKEN doesn't have these restrictions

Syntax

<!ATTLIST element_name attribute_name NMTOKEN default_value>

Example

```
<!ELEMENT mountains (mountain+) >
<!ELEMENT mountain (name) >
<!ATTLIST mountain country NMTOKEN #REQUIRED>
```

Valid XML - The following XML document would be valid, as it conforms to the above DTD:

```
<mountains>
    <mountain country="India">
        <name>Mount Everest</name>
```

```
    </mountain>
    <mountain country="AU">
      <name>Cradle Mountain</name>
    </mountain>
  </mountains>
```

Invalid XML - The following XML document would be invalid because the value of the first attribute contains internal whitespace:

```
<mountains>
    <mountain country="India">
      <name>Mount Everest</name>
    </mountain>
    <mountain country="Australia">
      <name>Cradle Mountain</name>
    </mountain>
</mountains>
```

(vii) DTD Attribute Types : NMTOKENS

- The attribute type of NMTOKENS allows the attribute value to be made up of multiple NMTOKENs, separated by a space.

Syntax:

```
<!ATTLIST element_name attribute_name  NMTOKENS  default_value>
```

Example

```
<!ELEMENT mountains (mountain+) >
<!ELEMENT mountain (name) >
<!ATTLIST mountains country NMTOKENS #REQUIRED>
```

Valid XML - The following XML document would be valid, as it conforms to the above DTD:

```
<mountains country="India">
    <mountain>
      <name>Mount Everest</name>
    </mountain>
    <mountain>
      <name>Thottabetta</name>
    </mountain>
</mountains>
```

(viii) DTD Attribute Types – ENTITY

The attribute type of ENTITY is used for referring to the name of an entity you've declared in your DTD

Syntax

<!ATTLIST element_name attribute_name ENTITY default_value>

Example

```
<!ELEMENT mountains (mountain+) >
<!ELEMENT mountain (name) >
<!ATTLIST mountain photo ENTITY #IMPLIED>
<!ENTITY mt_cook_1 SYSTEM "mt_cook1.jpg">
```

Valid XML - The following XML document would be valid, as it conforms to the above DTD

```
<mountains>
   <mountain photo="mt_cook_1">
      <name>Mount Cook</name>
   </mountain>
   <mountain>
      <name>Cradle Mountain</name>
   </mountain>
</mountains>
```

Invalid XML - The following XML document would be invalid. This is because the "photo" attribute of the second element contains a value that hasn't been declared as an entity:

```
<mountains>
   <mountain photo="mt_cook_1">
      <name>Mount Cook</name>
   </mountain>
   <mountain photo="None">
      <name>Cradle Mountain</name>
   </mountain>
</mountains>
```

(ix) DTD Attribute Types – ENTITIES

The attribute type of ENTITIES allows you to refer to multiple entity names, separated by a space

Syntax

<!ATTLIST element_name attribute_name ENTITIES default_value>

Example:

```
<!ELEMENT mountains (mountain+) >
<!ELEMENT mountain (name) >
<!ATTLIST mountain photo ENTITIES #IMPLIED>
<!ENTITY mt_cook_1 SYSTEM "mt_cook1.jpg">
```

```
<!ENTITY mt_cook_2 SYSTEM "mt_cook2.jpg">
```

Valid XML - The following XML document would be valid, as it conforms to the above DTD

```
<mountains>
   <mountain photo="mt_cook_1 mt_cook_2">
      <name>Mount Cook</name>
   </mountain>
   <mountain>
      <name>Cradle Mountain</name>
   </mountain>
</mountains>
```

Invalid XML - The following XML document would be invalid. This is because in the first element, a comma is being used to separate the two values of the "photo" attribute (a space should be separating the two values):

```
<mountains>
   <mountain photo="mt_cook_1, mt_cook_2">
      <name>Mount Cook</name>
   </mountain>
   <mountain>
      <name>Cradle Mountain</name>
   </mountain>
</mountains>
```

(x) DTD Attribute Types: NOTATION

- The attribute type of NOTATION allows you to use a value that has been declared as a notation in the DTD. A notation is used to specify the format of non-XML data. A common use of notations is to describe MIME types such as image/gif, image/jpeg etc

Syntax

- To declare a notation

```
<!NOTATION name SYSTEM "external_id">
```

- To declare the attribute

```
<!ATTLIST  element_name  attribute_name NOTATION
                         default_value>
```

Example

```
<!NOTATION GIF     SYSTEM      "image/gif">
<!NOTATION JPG     SYSTEM      "image/jpeg">
<!NOTATION  PNG    SYSTEM     "image/png">
<!ENTITY                      "mt_cook1.jpg">
```

```
mt_cook_1
<!ELEMENT mountains (mountain+) >
<!ELEMENT mountain (name) >
<!ATTLIST mountain photo ENTITY
                #IMPLIED photo_type NOTATION (GIF
                | JPG | PNG) #IMPLIED >
```

- In the DTD, we have specified that the value of the "photo_type" attribute can be one of the three values supplied. The following XML document would be valid, as it conforms to the above DTD:

```
<mountains>
   <mountain photo="mt_cook_1" photo_type="JPG">
      <name>Mount Cook</name>
   </mountain>
   <mountain>
      <name>Cradle Mountain</name>
   </mountain>
</mountains>
```

5.6 DTD ENTITY TYPES

- These are the key to replaceable content in both DTDs and XML documents. A general entity reference is used in the target XML document/s. The two main categroies of entities are:
  - **General entities** - They are usable within any XML document
  - **Parameter entities** – They may only be used in DTDs

General entities

- There are two kinds of general entities. They are:

- Parsed entities
  - **Unparsed entites**

Parsed entities

- It can be either internal or external.
- **Internal:** The actual replacement text is included with in the xml document
- **External:** the actual replacement text is located in external file or other resource
- The general syntax of parsed entity declaration is given below:

```
<!ENTITY entity-name "replacement-text">
```

- Once parsed entity is defined it can be referenced using the following syntax:

```
&entity-name;
```

- An example parsed entity declaration is given below:

<!ENTITY COPY "&#169">

- A reference of parsed entity from xml document is given below:

<copyright>©</copyright>

Unparsed entites

- To embed non-XML data (such as an image) into your XML document, you need to treat it as an external *unparsed entity*. To declare an external unparsed entity, you use the **<!ENTITY>**declaration along with the NDATA keyword.
- Unparsed entities are always external. All unparsed entities can be referenced only from attributes of types **ENTITY** or **ENTITIES**
- All unparsed entities must have an associated notation, which is also identified by name
- Syntax to declare unparsed entities:

<!ENTITY name SYSTEM value NDATA TYPE>

- You can also use public external unparsed entities by using the PUBLIC keyword along with a Formal Public Identifier (FPI):

<!ENTITY name PUBLIC FPI value NDATA TYPE>

Example

- Here's an example of a private external unparsed entity. Here, we declare a new notation named "JPG" for the "image/jpeg" MIME type. Then we declare an external unparsed entity called "mt_cook_1" that refers to an image file called "mt_cook1_jpg". We then create a new attribute of type ENTITY - this means that we can now use this attribute in our XML document to refer to the unparsed external entity.

```
//definition of notation
<!NOTATION JPG SYSTEM "image/jpeg">
//definition of unparsed entities
<!ENTITY mt_cook_1 SYSTEM "mt_cook1.jpg" NDATA JPG>
//declaration of attribute of ENTITY type
<!ATTLIST mountain photo ENTITY #IMPLIED>
```

- After declaring the external unparsed entity in our DTD and creating an attribute of type ENTITY, we can now embed it in our XML document:

```
<mountains>
        <!-- reference of unparsed entity -->
        <mountain photo="mt_cook_1">
            <name>Mount Cook</name>
        </mountain>
```

```
    <mountain>
        <name>Cradle Mountain</name>
    </mountain>
</mountains>
```

Parameter entities

- These types of entities are used extensively in DTDs and must always parsed entities. Parameter entities are always convenient shorthand for repetitive DTD declaration
- *Parameter Entiti*es used within the DTD itself. In other words, you can create an entity that can be used within your DTD declarations themselves.

Syntax

- When creating a parameter entity, you need to insert a percentage sign (%) between

ENTITY, and the name of the entity as shown below

```
<!ENTITY % entity-name "replacement-text">
```

- You can also declare external parameter entities. You do this using the following syntax
- Declaration of Private entity:

```
<!ENTITY % name SYSTEM uri>
```

- Declaration of Public entity:

```
<!ENTITY % name PUBLIC FPI uri>
```

- Once the parameter entity is defined, it can be referenced using the following syntax:

  %entity-name;

```
<!ENTITY % idfef_req “IDREF #REQUIRED”>
<!ATTLIST book bookid ID #REQUIRED
                publisher % idfef_req ;>
```

5.7 XML CONDITIONAL DTD SECTIONS

- Conditional sections of a DTD are defined using the **<![INCLUDE[ ]]>** tag to include the section or **<![IGNORE[ ]]>**to exclude the section.
- The format of conditional sections in a DTD is as follows:

```
<![ (INCLUDE|IGNORE) [DTD-Declarations] ]>
```

- Where **DTD-Declarations**—must be any valid DTD declarations or definitions. An example is given below:

```
<![ INCLUDE [
<!ELEMENT summary (comments*,title,shortdesc)> ]]>
<!%[IGNORE[<!ELEMENT summary (title,shortdesc)>]]>
```

- DTD declaration within the INCLUDE sections are used for validation by parser
- DTD declaration within the IGNORE sections are read, but not processed by parser
- Conditional sections are typically declared using parameter entities, and then referenced in other declarations for enhancing reusability.
- In the following example, an external DTD specifies different contents for draft and release (the commentselement is not present in release) .

5.8 LIMITATION OF DTDS

- The DTD has several shortcomings. First, a DTD document does not have to be coded in XML. That means that a DTD is itself not an XML document.
- Second, the data types available to define the contents of an attribute or element are very limited in DTD
- Some limitation of DTD include:
- DTDs are not extensible
    - DTD describes the rules of an XML vocabulary
    - All those rules must be present in single DTD
    - There is no mechanism for inclusion of mechanism from multiple sources
    - So not suitable for distributed environment
- Only one DTD may be associated with each XML document
    - large DTDs are hard to read and maintain
- DTD do not work well with XML namespace
    - All element names are global
      –Is <name> for people or companies?
      –can't declare both in the same DTD
- Very week data typing
    - There are no constraints imposed on the kind of data(int, float, string, date, etc) allowed within XML Element and Attribute, so data typing is not possible
- Limited content model description
  - **No OO type object inheritance**
    - Describing one element type in terms of another is not possible in DTD
- An XML document can override or ignore an external DTD using internal subset
    - Since the internal DTD has precedence over the external subset, there is no assurance that the rules of a DTD will

be followed by XML document with which it is associated

- Non-XML syntax
  - They are not written in XML syntax, which means you have to learn a new syntax in order to write them
  - DTD do not use well-formed XML Syntax
  - Most of the DTD represented in Extended Backus Naur Form (EBNF)
- No DOM supports
  - DOM is commonly used way to manipulate XML data, but it does not handle EBNF and provides no access to the rules of document model in the DTD
- Relatively few older and more expensive tools
  - DTD's design goal of interoperability with SGML and HTML, XML
  - However, most SGML tools are both expensive and complex

All xml document is comprised of unit of storage called entities. For example, document entity server as the entry point for an XML parser
The internal and external subset of the DTD are also entities, but unnamed one. Other types of entities are always identified and referred to by name. these are key replaceable content in both DTD

```
<!ENTITY % draft "IGNORE">
  <!ENTITY % release "INCLUDE">
  <![%draft;[<!ELEMENT summary (comments*,title,shortdesc)>]]>
  <!%[release;[<!ELEMENT summary (title,shortdesc)>]]>
  <!ELEMENT comments (#PCDATA)>
  <!ELEMENT title (#PCDATA)>
  <!ELEMENT shortdesc (#PCDATA)>
```

CHAPTER 6

SCHEMA

1. XML 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

2. WHY TO USE XML SCHEMA INSTEAD OF DTD?

1. XML Schemas Support almost all data types available in every programming language

- One of the greatest strengths of XML Schemas is the support for data types, which makes easier:
  - To describe allowable document content
  - To validate the correctness of data
  - To define data facets (restrictions on data)
  - To define data patterns

2. XML Schemas use XML Syntax:

- Another great strength about XML Schemas is that they are written in XML, so that:
  - You don't have to learn a new language
  - You can use your XML editor to edit your Schema files
  - You can use your XML parser to parse your Schema files
  - You can manipulate your Schema with the XML DOM
  - You can transform your Schema with XSLT

3. XML Schemas are extensible for future additions, because they are written in XML.

- With an extensible Schema definition, you can:
  - Reuse your Schema in other Schemas
  - Create your own data types derived from the standard types
  - Reference multiple schemas in the same document

4. XML Schemas supports Secure Data Communication

5. XSD is richer and more powerful than DTD.

6. XSD supports namespaces.

- Schema offers multiple **vocabulary support** based on namespace in xml

7. **XSD is W3C recommendation.**
8. **XSD offers inheritance supports**
   - New data type can be defined, which inherits feature from existing types, which may be pre-defined or user defined
9. **Dynamic schema support**
   - Schema can be selected and modified dynamically

3. CREATING XML SCHEMAS

The following Table shows a complete list of every element the XML Schema Definition Language supports. Table shows a complete list of every element the XML Schema

Element Name	***Description***
All	Indicates that the contained elements may appear in any order within a parent element.
Any	Indicates that any element within the specified namespace may appear within the parent element's definition. If a type is not specifically declared, this is the default
anyAttribute	Indicates that any attribute within the specified namespace may appear within the parent element's definition
annotation	Indicates an annotation to the schema
Appinfo	Indicates information that can be used by an application.
Attribute	Declares an occurrence of an attribute
attributeGroup	Defines a group of attributes that can be included within a parent element
Choice	Indicates that only one contained element or attribute may appear within a parent element
complexContent	Defines restrictions and/or extensions to a complexType
complexType	Defines a complex element's construction
documentation	Indicates information to be read by an individual.
Element	Declares an occurrence of an element
Extension	Extends the contents of an element
Field	dicates a constraint for an element using XPath
Group	Logically groups a set of elements to be included together within another element definition.
import	Identifies a namespace whose schema elements and attributes can be referenced within the current schema
include	Indicates that the specified schema should be included in the target namespace
Key	Indicates that an attribute or element value is a key within the specified scope

Keyref	Indicates that an attribute or element value should correspond with those of the specified key or unique element
List	Defines a simpleType element as a list of values of a specified data type
notation	Contains a notation definition
redefine	Indicates that simple and complex types, as well as groups and attribute groups from an external schema, can be redefined
restriction	Defines a constraint for the specified Element
schema	Contains the schema definition
selector	Specifies an XPath expression that selects a set of elements for an identity constraint
sequence	Indicates that the elements within the specified group must appear in the exact order they appear within the schema
simpleContent	Defines restrictions and/or extensions of a simpleType element
simpleType	Defines a simple element type
Union	Defines a simpleType element as a collection of values from specified simple data types
unique	Indicates that an attribute or element value must be unique within the specified scope

4. SCHEMA DATA TYPES

- XML Schcma support almost all data types present in every programming language.
- XML Schema provides two basic kinds of data-types:

(i) **primitive data types** – Those that are not defined in terms of other datatypes

(ii) **derived data types** – Those that are defined in terms existing data types

Primitive data types

- Primitive data type can be used for element or attribute values, but cannot contain child elements, primitive type is always built-in
- The following primitive data types that are built-in to xml schema

Data type	Explanation
Decimal	Decimal number (infinite precision)
Float	Single-precision floating-point number (32-bit)
Double	Double-precision floating-point number (64-bit)
Boolean	Boolean value
String	Arbitrary text string
URI reference	A standard Internet URI

recurringDuration	A recurring duration of Time
ID	XML 1.0 Specification ID type
IDREF	XML 1.0 Specification IDREF type
ENTITY	XML 1.0 Specification ENTITY type
NOTATION	Similar to DTD notation attribute type
QName	A legal QName string

Derived data types

- A derived data types in one that is defined in term of an existing types known as base type
- Derived types may have attributes and may have element or mixed content
- New types may be derived from either a primitive type or another derived type. For
- Example integer are subset of real number. The following definition in turn derive an even more restricted type of integers:

<simpleType name=”negativeInteger” base=”xsi:integer”>
<minInclusive value=”unbounded” />

<maxInclusive value=”-1” />

</simpleType>

The built-in derived types are listed below:

Data type	Explanation
Integer	Integers (infinite precision)
positiveInteger	Positive integers (infinite precision)
negativeInteger	Negative integers (infinite precision)
nonPositiveInteger	Negative integers including 0 (infinite precision)
nonNegativeInteger	Positive integers including 0 (infinite precision)
Byte	Integer represented by 8 bits
unsignedByte	Integer represented by 8 bits (no symbols)
Short	Integer represented by 16 bits
unsignedShort	Integer represented by 16 bits (no symbols)
Int	Integer represented by 32 bits
unsignedInt	Integer represented by 32 bits (no symbols)
Long	Integer represented by 64 bits
unsignedLong	Integer represented by 64 bits (no symbols)
language	A natural language identifier
Name	Name of the data type
NCName	Non-colonized name

Built-in derived types Representing Dates and Times

Name	Explanation
Time	Time of day
dateTime	Date and time of day
Date	Date
gYear	Year
gYearMonth	Year and month
gMonth	Month
gMonthDay	Month and day
gDay	Day

DTD-Compatible built-in derived types

Name	Explanation
IDREFS	XML 1.0 Specification IDREFS type
ENTITIES	XML 1.0 Specification ENTITIES type
NMTOKEN	XML 1.0 Specification NMTOKEN type
NMTOKENS	XML 1.0 Specification NMTOKENS type

From this, you can see that when declaring an attribute, you must specify a type. This type must be one of the simple types

5. ATOMIC AND LIST DATA TYPES

There is one last division of schema data types. They are
(i) Atomic datatype (ii) list datatype

Atomic type

- it is a type that have values that are defined to be indivisible. Atomic and primitive types are not the same.
- Numbers and strings are atomic types, since their values cannot be described using any smaller pieces. Xml schema has not concept of character datatype – thus string is atomic
- Atomic type may be either primitive or derived types. An example given below:

  String – atomic primitive type

  Date – atomic derived types

Integer – atomic derived types

List data type

- The list element defines a simple type element as a list of values of a specified data type
- It is a type that have defined in terms of existing types. List data type has a value that consist of length sequence of atomic value
- In Schema, one list cannot be made from other list. List types are always derived types, which must be delimited by white space characters, just like IDREF or NMTOKEN attributes types in the DTD
- List type must allows the presence of white space, but cannot use any white space within the individual values of list items
- The syntax to define list type is given below:

<list id=ID itemType=QName *any attributes*>(annotation?,(simpleType?))

</list>

- The ? sign declares that the element can occur zero or one time inside the list element

Attribute	Description
id	Optional. Specifies a unique ID for the element
itemType	Specifies the name of a built-in data type or simpleType element defined in this or another schema. This attribute is not allowed if the content contains a simpleType element, otherwise it is required
any attributes	Optional. Specifies any other attributes with non-schema namespace

- The following define simple type called **"valuelist"**. Element of **"valuelist" type can store** a list of integers:

```
<xs:simpleType name="valuelist">
        <xs:list itemType="xs:integer"/>

</xs:simpleType>
```

- The following define <**intValues**> element of **"valuelist"** type that stores list of integers:

<xs:element name="Marks" type="valuelist"/>

- Now, The <**Marks**> element in a document could look like this (notice that the list will have five list items):

<Marks>100 34 56 80 77</Marks>

- Note: White space is treated as the list item separator!

6. ASPECTS OF DATATYPES IN SCHEMA

All schema data-types are comprised of three parts:

1. **a value space**
2. **a lexical space**
3. **a set of facets**

value space

- Each data-type has a range of possible values. For example, float data type has value space that range from -∞ to +∞
- Derived types inherit their values from their base type, and may also constraint that the value space to an explicit subset of base type.
- For example, derived data type such as integer would allow any positive or negative whole number values, but without decimal fraction
- Value space always have certain facets (abstract properties) such as:

- **Equality**
- **Bounds**
- **Order**
- **CardinalityNumberic**
- **Non-numeric**

Lexical space

- It is a set of string literals that represent the value of a data type. In general string literal have only one lexical representation, whereas numeric value may have several equivalent and equally valid lexical representations
- Consider the following numeric literals:

  "100" "1.0E2" and "10^2"
- all have different lexical values, but have identical numerical values in floating point value space

Facets

- It is defining properties of data types which distinguish the data type from others
- It includes properties such as string length or bounds of numeric data type

Fundamental facets of data types

- There are five fundamental facets of data types
- **Equality** – similar to comparison operator in a programming language. With

these two values can be compared to determine whether equal or not. This is applicable to all data types

- **Order** – only for some data types, there is defined relationship exists between values. For examples, numbers may have ordered value. These properties can be applied to both numeric and non-numeric types. For example, number 10 followed by 11, proceeded by 9

- **Bounds**
  - Only ordered datatypes may be constrained to a range of values. Data type values may have either lower or upper bound or both
  - If value space has both upper and lower bounds, its data type is simply considered to be bounded
  - An example:

```
<simpleType name=”negativeInteger” >
<xs:restriction base="xs:integer">
        <minInclusive value=”unbounded” />
        <maxInclusive value=”-1” />
    </xs:restriction>
</simpleType>
```

- **Cardinality: minOccurs and maxOccurs**
  - Cardinality define the maximum number of allowed values within the value space. To define cardinality we use **minOccurs** and **maxOccurs** attributes
  - All value space may have an associated concept of cardinality, which derermine number of values within the value space. A value space may be
    - **finite** – list of enumerated values
    - **countably infinite**
    - **uncountably infinite**
  - the following table shows comparision of DTD cardinality operator with schema’s:

DTD Cardinality Operator	minOccurs value	minOccurs value	No. of child allowed
[none]	1	1	Only one
?	0	1	Zero or one
*	0	unbounded	Zero or more
+	1	Unbounded	one or more

- **Numberic type** – In includes floating pointer and integers, etc
- **Non-numeric type** – string, date, and other non-numeric types

7. USER DEFINED DATA TYPES / DATA TYPE DEFINITION IN SCHEMA

- Other than predefined data types, schema allows you to define your data types.

Schema provides two types of data definitions:

- **SimpleType**
- **complexType**

<simpleType> definition

- with simple type definition, we can create derived data types, including those were built in to the schema definition
- A simple type definition is a set of constraint on value space and lexical space of primitive data types

The <simpleType> element

- The general syntax of <simpleType> is given below:

<simpleType [id = ID] [name = NCName] [any attributes]> (annotation?, (restriction | list | union))
</simpleType>

- Definition of a simple type determines the constraints on and information about the values of attributes or elements with text-only content
- The ? sign declares that the element can occur zero or one time inside the simpleType element

Attribute	Description
Id	Optional. Specifies a unique ID for the element
Name	Specifies a name for the element. This attribute is required if the simpleType element is a child of the schema element, otherwise it is not allowed
any attributes	Optional. Specifies any other attributes with non-schema namespace

Scope of the data types:

- Each data type can be defined with two scope:

- **Data type with global scope**
- **Data type with local scope**

Data type with local scope (anonymous data-type)

- The data types is defined within the <element> or <attribute> will have local scope. They can be referenced only from containing <element> or <attribute>. These type are defined without name(anonymous data type)
- This example defines an element called "age" that is a simple type with a restriction.

The value of age can NOT be lower than 0 or greater than 100

```
<xs:element name="age">
        <xs:simpleType name="ageType">
          <xs:restriction base="xs:integer">
                <xs:minInclusive value="0"/>
                <xs:maxInclusive value="100"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
```

Data type with global scope (named data-type)

The data type defined within the <Schema> will have global scope. These data type can be referenced from <element> or <attribute> defined in the schema

```
<schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:simpleType name="ageType">
      <xs:restriction base="xs:integer">
          <xs:minInclusive value="0"/>
<xs:maxInclusive value="100"/>
</xs:restriction>
</xs:simpleType>
      <xs:element name="age" type="ageType" />
</xs:schema>
```

8. CONSTRAINING FACETS / RESTRICTIONS ON DATA TYPES / REFINING SIMPLE TYPES USING FACETS

- To give greater control over the definition of elements and attributes, the W3C added acets to the XML Schema Definition Language.
- A facet can only be specified for a <simpleType> element, and it helps determine the set of values for a <simpleType> element
- Constraining facets are those that limit the value space of a derived datatype, which in turn limit that data type's lexical space
- There are several constraining facets that may be applied to any appropriate derived data type
- The following table list the list of constraining facets or restriction:

Constraint	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

Enumeration: Restrictions on a Set of Values

- To limit the content of an XML element to a set of acceptable values, we would use the enumeration constraint.
- The example below defines an element called "car" with a restriction. The only acceptable values are: Audi, Golf, BMW:

```
<xs:element name="car">
  <xs:simpleType>
    <xs:restriction base="xs:string">
          <xs:enumeration value="Audi"/>
          <xs:enumeration value="Golf"/>
          <xs:enumeration value="BMW"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

length, minLength, maxLength: Restrictions on Length

- To limit the length of a value in an element, we would use the length, maxLength, and minLength constraints

length:

- The <length> facet determines the number of units of length for the specified data type. The nature of unit will vary, depending on base data type.

- For string data type length is number of Unicode point or characters. For binary type the length is number of octet(8 bit byte)
- for list type length is the number of items in the list
- This example defines an element called "password" with a restriction. The value must be exactly eight characters:

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

minLength – minimum number of unit permitted for the data types
maxLength – maximum number of unit permitted for the data types

- This example defines another element called "password" with a restriction. The value must be minimum five characters and maximum eight characters:

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

minInclusive, maxInclusive, minExclusive, maxExclusive: Restrictions on Values

- You can apply these facets to only data types that has an order relations
  - o“min” – define lower bound
  - o“max” – define upper bound

The following example defines an element called "age" with a restriction. The value of age cannot be lower than 0 or greater than 120:

```
<xs:element name="age">
 <xs:simpleType>
  <xs:restriction base="xs:integer">
   <xs:minInclusive value="0"/>
   <xs:maxInclusive value="120"/>
  </xs:restriction>
 </xs:simpleType>
</xs:element>
```

pattern - Restrictions on a Series of Values

- To limit the content of an XML element to define a series of numbers or letters that can be used, we would use the pattern constraint
- Pattern is a regular expression(refex) that data-type's lexical representation must match for.
- An example defines simple type that uses pattern:

```
<xs:simpleType name="orderidtype">
 <xs:restriction base="xs:string">
  <xs:pattern value="[0-9]{6}"/>
 </xs:restriction>
</xs:simpleType>
```

- The definition indicates that the value of the element or attribute must be a string, it must be exactly six characters long, and those characters must be a number from 0 to 9
- The example below defines an element called "letter" with a restriction. The acceptable value is zero or more occurrences of lowercase letters from a to z:

```
<xs:element name="letter">
 <xs:simpleType>
  <xs:restriction base="xs:string">
   <xs:pattern value="([a-z])*"/>
  </xs:restriction>
 </xs:simpleType>
</xs:element>
```

- relace the <pattern> definition in the above example with the following to accept one or more occurrences of lowercase letters from a to z:

<xs:pattern value="([a-z])+"/>

- The next example defines an element called "gender" with a restriction. The only acceptable value is male OR female:

```
<xs:element name="gender">
 <xs:simpleType>
  <xs:restriction base="xs:string">
   <xs:pattern value="male|female"/>
  </xs:restriction>
 </xs:simpleType>
</xs:element>
```

<fractionDigits> facet

- The <fractionDigits> facet specifies the maximum number of decimal digits in the fractional part. The value for this facet must be a nonNegativeInteger
- For example, look at the following attribute declaration:

```
<xsd:attribute name=”SubTotal”>
   <xsd:simpleType>
```

```
                    <xsd:restriction base=”xsd:decimal”>
                            <xsd:fractionDigits value=”2”/>
                    </xsd:restriction>
                </xsd:simpleType>
            </xsd:attribute>
```

<whitespace> facet

- The <whiteSpace> facet specifies how whitespace is treated for the type definition’s value. This particular facet can hold one of three values:
  - **collapse –** Specifying collapse indicates that all whitespace consisting of more than a single space will be converted to a single space and that all leading and trailing blanks will be removed
  - **preserve -** A value of preserve leaves the value as is
  - **replace -** Assigning a value of replace causes all tabs, line feeds, and carriage returns to be replaced with a single space.

<xsd:complexType> - COMPLEX TYPE DEFINITION

- A complex type definition specify element content model, which is much better than content model of DTD that describe attributes and children of particular element type
- A complex type definition can be extension or restriction of simple or complex base types
- A complex type can extends another type by appending additional content model declaration or additional attribute declaration

```
<complexType   id=ID                name=NCName
                                    abstract=true|false mixed=true|false
                    block=(#all|list of (extension|restriction))
                   final=(#all|list of (extension|restriction))        any
    attributes>          (annotation?,(simpleContent|complexContent|
    ((group|all|choice|sequence)?,((attribute|attributeGroup)*,
  anyAttribute?))))

</complexType>
```

- The ? sign declares that the element can occur zero or one time, and the * sign declares that the element can occur zero or more times inside the complexType element

Attribute	Description
Id	Optional. Specifies a unique ID for the element
name	Optional. Specifies a name for the element

abstract	Optional. Specifies whether the complex type can be used in an instance document. True indicates that an element cannot use this complex type directly but must use a complex type derived from this complex type. Default is false
mixed	Optional. Specifies whether character data is allowed to appear between the child elements of this complexType element. Default is false. If a simpleContent element is a child element, the mixed attribute is not allowed!
block	Optional. Prevents a complex type that has a specified type of derivation from being used in place of this complex type. This value can contain #all or a list that is a subset of extension or restriction: • extension - prevents complex types derived by extension • restriction - prevents complex types derived by restriction • #all - prevents all derived complex types
final	Optional. Prevents a specified type of derivation of this complex type element. Can contain #all or a list that is a subset of extension or restriction. • extension - prevents derivation by extension • restriction - prevents derivation by restriction • #all - prevents all derivation
any attributes	Optional. Specifies any other attributes with non-schema namespace

A complex type definition:

- Provides a mechanism to validate a document instance containing that type
- Describe the content of an element type, which may be element only, text only, mixed, or empty
- Describes element's attribute existence and content
- Derives it definition from another simpleType or complexType
- Control the ability to derive additional types.

- Similar to <simpleType>, <complexType> can be defined with global scope or local scope
- A <complexType> element in the XML Schema Definition Language may contain only

- one of the following elements:
  - **all**
  - **choice**

9. CREATING XML SCHEMA

- **Step 1:** The following declaration define **complex type with local scope**(**anonymous type declaration**)for validating structure of the **note.xml** file:

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

Step 2: A Referencing an XML Schema from XML document(or) linking Schema with XML Document

```
<?xml version="1.0"?>
<note xmlns="http://www.xyz.com"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.xyz.com note.xsd">

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

10.COMPONENTS OF XML SCHEMA

- **The <schema> Element**
  - **The <element> Element**
  - **The <attribute> Element**
  - **The <simpleType> Element**
  - **The <complexType> Element**
  - **The <annatotation>**
    - **The <documentation> Element**
    - **The <appInfo> Element**
  - The <schema> element is the root element of every XML Schema.
  - The <schema> element may contain some attributes.

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

- It indicates that the elements and data types used in the schema come from the "**http://www.w3.org/2001/XMLSchema"** namespace.
- It also specifies that the elements and data types that come from the **"http://www.w3.org/2001/XMLSchema"** namespace should be prefixed with **xs**

This fragment: targetNamespace="http://www.xyz.com"

- indicates that the default namespace is "http://www.xyz.com".

This fragment: elementFormDefault="qualified"

- indicates that any elements used by the XML instance document which were declared in this schema must be namespace qualified.

Referencing a Schema in an XML Document

This XML document has a reference to an XML Schema:

```
<?xml version="1.0"?>
<note xmlns="http://www.xyz.com"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.xyz.com  note.xsd">

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

The following fragment:
xmlns="http://www.xyz.com"

- Specifies the default namespace declaration. This declaration tells the schema- validator that all the elements used in this XML document are declared in the **"http://www.xyz.com"** namespace
- You can use the schemaLocation attribute. This attribute has two values, separated by a space. The first value is the namespace to use. The second value is the location of the XML schema to use for that namespace: **xsi:schemaLocation="http://www.xyz.com note.xsd"**

<xs:element> Element

The <xs:element> tag is used to describe every elements in your XML document. With this tag you can describe both **element of simple type**s or **element of complex types The general syntax of <xs:element> is given below:**

<element id=ID name=NCName ref=QName type=QName
substitutionGroup=QName default=string
fixed=string form=qualified|unqualified
maxOccurs=nonNegativeInteger|unbounded
minOccurs=nonNegativeInteger nillable=true|false
abstract=true|false
block=(#all|list of (extension|restriction)) final=(#all|list of (extension|restriction)) *any attributes*>
annotation?,(simpleType|complexType)?,(unique|key|keyref)*

</element>

- The ? sign declares that the element can occur zero or one time, and the * sign declares that the element can occur zero or more times inside the element.
-

Attribute	Description
Id	Optional. Specifies a unique ID for the element
Name	Optional. Specifies a name for the element. This attribute is required if the parent element is the schema element
Ref	Optional. Refers to the name of another element. The ref attribute can include a namespace prefix. This attribute cannot be used if the parent element is the schema element
Type	Optional. Specifies either the name of a built-in data type, or the name of a simpleType or complexType element

substitutionGroup	Optional. Specifies the name of an element that can be substituted with this element. This attribute cannot be used if the parent element is not the schema element
Default	Optional. Specifies a default value for the element (can only be used if the element's content is a simple type or text only)
Fixed	Optional. Specifies a fixed value for the element (can only be used if the element's content is a simple type or text only)
Form	Optional. Specifies the form for the element. "unqualified" indicates that this element is not required to be qualified with the namespace prefix. "qualified" indicates that this element must be qualified with the namespace prefix. The default value is the value of the elementFormDefault attribute of the schema element. This attribute cannot be used if the parent element is the schema element
maxOccurs	Optional. Specifies the maximum number of times this element can occur in the parent element. The value can be any number >= 0, or if you want to set no limit on the maximum number, use the value "unbounded". Default value is 1. This attribute cannot be used if the parent element is the schema element
minOccurs	Optional. Specifies the minimum number of times this element can occur in the parent element. The value can be any number >= 0. Default value is 1. This attribute cannot be used if the parent element is the schema element
Nillable	Optional. Specifies whether an explicit null value can be assigned to the element. True enables an instance of the element to have the null attribute set to true. The null attribute is defined as part of the XML Schema namespace for instances. Default is false
Abstract	Optional. Specifies whether the element can be used in an instance document. True indicates that the element cannot appear in the instance document. Instead, another element whose substitutionGroup attribute contains the qualified name (QName) of this element must appear in this element's place. Default is false

Block	Optional. Prevents an element with a specified type of derivation from being used in place of this element. This value can contain #all or a list that is a subset of extension, restriction, or equivClass: • extension - prevents elements derived by extension • restriction - prevents elements derived by restriction • substitution - prevents elements derived by substitution • #all - prevents all derived elements
Final	Optional. Sets the default value of the final attribute on the element element. This attribute cannot be used if the parent element is not the schema element. This value can contain #all or a list that is a subset of extension or restriction: • extension - prevents elements derived by extension • restriction - prevents elements derived by restriction • #all - prevents all derived elements
any attributes	Optional. Specifies any other attributes with non-schema namespace

XSD Simple Elements definition

- A simple element is an XML element that contains only text. It cannot contain any other elements or attributes.
- The text can be of many different types. It can be one of the types included in the XML Schema definition (boolean, string, date, etc.), or it can be a custom type that you can define yourself.
- You can also add restrictions (facets) to a data type in order to limit its content, or you can require the data to match a specific pattern.
- The syntax for defining a simple element is:

  <xs:element name="elementname" type="data-type-name"/>
- The following example is a schema with four simple elements named "fname", "lname", "age", and "dateborn", which are of type string, nonNegativeInteger, and date:

Schema Simple element definition	Valid xml example
<xs:element name="sname" type="xs:string"/>	**<name>ram</name>**
<xs:element name="age" type="xs:nonNegativeInteger"/>	**<age>22</age>**
<xs:element name="dateborn" type="xs:date"/>	**<dateborn>1970-03-7 </dateborn>**

<xs:element name="NativeIndia" type="xs:boolean"/>	**<NativeIndia> true </NativeIndia>**

Default and Fixed Values for Simple Elements

- Simple elements may have a default value OR a fixed value specified.

<element> with default value:

- A default value is automatically assigned to the element when no other value is specified.
- In the following example the default value is "red":
  <xs:element name="color" type="xs:string" default="red"/>

<element> with fixed value:

- A fixed value is also automatically assigned to the element, and you cannot specify another value.
- In the following example the fixed value for company name element is "panimalar":

<xs:element name="company Name" type="xs:string" fixed="panimalar"/>

XSD Complex Elements definition

- An element's type can be defined with either a **<complexType>** element, a **<simpleType>** element, a **<complexContent>** element, or a **<simpleContent>**

  element
- Simple elements are defined with **<simpleType>**, where as complex elements can be defined with **<complexType>** element
- Complex element definition describe structure of entire xml document or the structure of complex elements.complex elmenents will haves other elements as its children
- The following example defines complex element type called

```
<xsd:element name="PersonName">
        <xsd:complexType content="elementOnly">
                <xsd:sequence>
                        <xsd:element name="FirstName" type ="text" minOccurs="1"/>
                        <xsd:element name="MiddleName" type ="text" minOccurs="0"
                          maxOccurs="1" />
                        <xsd:element name="LastName" type ="text" minOccurs="1" />
                </xsd:sequence>
```

```
        </xsd:complexType>
</xsd:element>
```

Valid XML that conforms to Schema:

```
<PersonName>
        <FirstName>Rajarajan</FirstName>
        <LastName>Rajendran</LastName>
</PersonName>
```

Declaring Attributes

- Attributes in an XML document are contained by elements. To indicate that a complex element has an attribute, use the <attribute> element of the XML Schema Definition Language.
- For instance, if you look at the following section from the **PurchaseOrder** schema, you can see the basics for declaring an attribute:

```
<xsd:complexType name=”ProductType”>
    <xsd:attribute name=”Name” type=”xsd:string”/>
    <xsd:attribute name=”Id” type=”xsd:positiveInteger”/>
    <xsd:attribute name=”Price”>
        <xsd:simpleType>
            <xsd:restriction base=”xsd:decimal”>
                    <xsd:fractionDigits value=”2”/>
            </xsd:restriction>
        </xsd:simpleType>
    </xsd:attribute>
    <xsd:attribute name=”Quantity”
    type=”xsd:positiveInteger”/>
</xsd:complexType>
<xsd:element name=”Product” type=”ProductType”
        minOccurs=”1”
            maxOccurs=”unbounded”/>
```

Valid XML that conform to Schema:

```
<Product Name=”Soup” Id=”001254” Price=”1.33” Quantity=”1”/>
```

- The general syntax for defining attribute is given below:

<attribute id=ID
name=NCName ref=QName type=QName
default=string fixed=string
use=optional|prohibited|required
any attributes >
(annotation?,(simpleType?))

</attribute>

- When declaring an attribute, you must specify a type. This type must be one of the simple types defined in schema
- The ? sign declares that the element can occur zero or one time inside the attribute element)

(OR)

- Here's the basic syntax for the <attribute> element:

<attribute name="" type="" [use=""] [fixed=""] [default=""] [ref=""]/>

The use attribute can contain one of the following possible values:

- **optional**
- **prohibited**
- **required**

Attribute	Description
default	Optional. Specifies a default value for the attribute. Default and fixed attributes cannot both be present
Fixed	Optional. Specifies a fixed value for the attribute. Default and fixed attributes cannot both be present
Id	Optional. Specifies a unique ID for the element
Name	Optional. Specifies the name of the attribute. Name and ref attributes cannot both be present
Ref	Optional. Specifies a reference to a named attribute. Name and ref attributes cannot both be present. If ref is present, simpleType element, form, and type cannot be present
Type	Optional. Specifies a built-in data type or a simple type. The type attribute can only be present when the content does not contain a simpleType element
Use	Optional. Specifies how the attribute is used. Can be one of the following values: • **optional** - the attribute is optional (this is default) • **prohibited** - the attribute cannot be used • **required** - the attribute is required

any attributes	Optional. Specifies any other attributes with non-schema namespace

Attributes with Default and Fixed Values

- Attributes may have a default value OR a fixed value specified.
- A default value is automatically assigned to the attribute when no other value is specified.
- In the following example the default value is "EN":
  <xs:attribute name="lang" type="xs:string" default="EN"/>
- A fixed value is also automatically assigned to the attribute, and you cannot specify another value for the attribute
  <xs:attribute name="companyName" type="xs:string" fixed="panimalar"/>

Optional and Required Attributes

Attributes are optional by default. To specify that the attribute is required, use the "use" attribute:

<xs:attribute name="lang" type="xs:string" use="required"/>

Complete Schema example given below:

The following is an example of complete shcema definition, which also include definition complex type, definition of complex element, definition of attribute etc:

```
<?xml version="1.0" ?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:simpleType name="PersonTitle" base ="xsd:string">
        <xsd:enumeration value ="Mr" />
        <xsd:enumeration value ="Ms" />
        <xsd:enumeration value ="Dr" />
        <xsd:enumeration value ="Rev" />
 </xsd:simpleType>
  <xsd:complexType name="text" content ="textOnly"
                        base="xsd:string" derivedBy="restriction" />
 <xsd:element name="PersonName">
<xsd:complexType content="elementOnly">
       <xsd:choice>
              <xsd:element name="SingleName" type="Text"
              minOccurs="1"/>
             <xsd:sequence>
```

```
                    <xsd:element name="FirstName" type ="text"
                    minOccurs="1"/>
                    <xsd:element name="MiddleName" type
                                ="text" minOccurs="0"
                                maxOccurs="1" />
                    <xsd:element name="LastName" type ="text"
                    minOccurs="1" />
                </xsd:sequence>
            </xsd:choice>
            <xsd:attribute name="honorific" type="PersonTitle" />
            <xsd:attribute name="suffix">
                <xsd:simpleType base="xsd:string">
                    <xsd:enumeration value="Jr" />
                    <xsd:enumeration value="Sr" />
                </xsd:simpleType >
            </xsd:attribute>
</xsd:complexType>
</xsd:element>
</xsd:schema>
```

Valid XML that conforms to Schema:

(i)
```
<PersonName honorific="Mr"  suffix="Jr">
        <FirstName>Rajarajan</FirstName>
        <LastName>Rajendran</LastName>
    </PersonName>
```

(ii)
```
<PersonName honorific="Mr"  suffix="Jr">
        <SingleName>Rajarajan</SingleName>

    </PersonName>
```

Anonymous Type Declarations

Sometimes within an XML schema it may not be necessary to create a separate type definition for an element or attribute. In such cases, you may use "anonymous" type declarations. Consider the following example:

```
<xsd:element name="PersonName">
    <xsd:complexType content="elementOnly">
        <xsd:sequence>
            <xsd:element name="FirstName" type ="text"
            minOccurs="1"/>
          <xsd:element name="MiddleName" type ="text"
              minOccurs="0" maxOccurs="1" />
            <xsd:element name="LastName" type ="text"
            minOccurs="1" />
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
```

In the above example complex type defined within the element or attribute, without name. This is referred to as anonymous type declaration. This data type can be referemced only from containing element or attribute

Outside of containing element or attribute it cannot be accessible

Specifying Mixed Content for Elements

- Element may have **textOnly, elementOnly, empty,** or **mixed** and some contain elements and attributes
- You can also define element that contain both text as well as child elements. To do this, mixed attribute of complexType must be set to true. Elements based on this type definition can mix their contents with both text and child elements.
- For instance, let's examine the following sample XML document:

```
<Letter>
<Greeting>Dear Mr.<Name>John Smith</Name>.</Greeting>
Your order of <Quantity>1</Quantity>

 <Product>Big Screen TV</Product> has been shipped.

</Letter>
```

- Notice the appearance of text among the child elements of <Letter>. The schema for this XML document would appear as follows:

```
<xsd:element name="Letter">
    <xsd:complexType mixed="true">

        <xsd:element name="Greeting">
```

```
                    <xsd:complexType mixed="true">
                        <xsd:element name="Name"
                        type="xsd:string"/>
                    </xsd:complexType>
                </xsd:element>
                <xsd:element name="Quantity"
                type="xsd:postiveInteger"/>
                <xsd:element name="Product" type="xsd:string"/>
            </xsd:complexType>
        </xsd:element>
```

Annotating Schemas

The annotation element is a top-level element that specifies schema comments. The comments serve as inline documentation. You can add information(label) that could be for either humber readable or machine readable or both

The XML Schema Definition Language defines three new elements to add annotations to an XML schema:

- **<annotation>** **<appInfo>** **<documentation>**

<annotation>

- It is a container element for both <appInfo> and <documentation>
- It can appear any where within a schema, usually first child of some other element

<appInfo>

- this element includes schema description that read by another external program
- this define label for machine consumption

<documentation>

- this element includes schema description that read by human
- This define label for human consumption
- Note: both **<appInfo>** and **<documentation>** cannot be used alone, they must be used as children of the **<annotation>** element
- The syntax:

```
<annotation id=ID any attributes >(appinfo|documentation)*
</annotation>
```

The following example describe the feature of <annotation>,

<appInfo>,
<documentation>:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema ">

<annotation>
    <appInfo>Wrox Schema -Annotations example
    </appInfo>
    <documentation>Schema is Copyright 2000 by worx
    press Ltd.
</documentation>
</annotation>
<element name="PersonName">
<complexType content ="elementsOnly ">
<annotation>
 <documentation>
 The use of the &lt;SingleName &gt; element solves the 'Cher'
 problem.
</documentation>
</annotation>
<choice> ... </choice>
</complexType>
</element>
</schema>
```

Model Groups - <group>

- A *model group* is a logically grouped set of elements. It is used to define a group of elements to be used in complex type definitions.
- It is portion of complex type definition that describe element's content model. A model group con consists of element declaration wild cards, and other model groups:
- A model group can be constructed using one of the following XML Schema Definition elements:
    - **<all>**
        - **<choice>**
        - **<sequence>**
- Here's the basic syntax for the <group> element:

<group name="" [ID=""] [ref=""] [maxOccurs=""] [minOccurs=""] >
(annotation?,(all|choice|sequence)?)
</group>

- The ? sign declares that the element can occur zero or one time inside the group element.
- By default, the maxOccurs and minOccurs attributes are set to 1. The following table describe various attrinute of group

Attribute	Description
Id	Optional. Specifies a unique ID for the element
Name	Optional. Specifies a name for the group. This attribute is used only when the schema element is the parent of this group element. Name and ref attributes cannot both be present
Ref	Optional. Refers to the name of another group. Name and ref attributes cannot both be present
maxOccurs	Optional. Specifies the maximum number of times the group element can occur in the parent element. The value can be any number >= 0, or if you want to set no limit on the maximum number, use the value "unbounded". Default value is 1
minOccurs	Optional. Specifies the minimum number of times the group element can occur in the parent element. The value can be any number >= 0. Default value is 1
any attributes	Optional. Specifies any other attributes with non-schema namespace

The following example defines a group containing a sequence of four elements and uses the group element in a complex type definition:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:group name="custGroup">
  <xs:sequence>
    <xs:element name="customer" type="xs:string"/>
    <xs:element name="orderdetails" type="xs:string"/>
    <xs:element name="billto" type="xs:string"/>
    <xs:element name="shipto" type="xs:string"/>
  </xs:sequence>
</xs:group>

<xs:element name="order" type="ordertype"/>
<xs:complexType name="ordertype">
  <xs:group ref="custGroup"/>
  <xs:attribute name="status" type="xs:string"/>
</xs:complexType>
</xs:schema>
```

- Note:The <group> element can be used for both the definition of group and any reference to the named group

All Groups- <all>

- The <all> element specifies that the child elements can appear in any order within the parent element and that each child element can occur zero or one time.

Syntax:

<all [ID=””] [ref=””] [maxOccurs=””] [minOccurs=””] [*any attributes…* **]> (annotation?,element*)**

</all>

Attribute	**Description**
Id	Optional. Specifies a unique ID for the element
maxOccurs	Optional. Specifies the maximum number of times the element can occur. The value must be 1.
minOccurs	Optional. Specifies the minimum number of times the element can occur. The value can be 0 or 1. Default value is 1
any attributes	Optional. Specifies any other attributes with non-schema namespace

Example 1:

```
<xs:element name="person">
 <xs:complexType>
  <xs:all>
   <xs:element name="firstname" type="xs:string"/>
   <xs:element name="lastname" type="xs:string"/>
  </xs:all>
 </xs:complexType>
</xs:element>
```

The example above indicates that the "firstname" and the "lastname" elements can appear in any order but both elements MUST occur once and only once

Valid XML that conforms to Schema:

```
(i) <Person>
     <LastName>Rajendran</LastName>
    <FirstName>Rajarajan</FirstName>
    </Person>
(ii)<Person>
     <FirstName>Rajarajan</FirstName>
     <LastName>Rajendran</LastName>
```

```
</Person>
```

Example 2:

```
<xs:element name="person">
 <xs:complexType>
  <xs:all minOccurs="0">
   <xs:element name="firstname" type="xs:string"/>
   <xs:element name="lastname" type="xs:string"/>
  </xs:all>
 </xs:complexType>
</xs:element>
```

- The example above indicates that the "firstname" and the "lastname" elements can appear in any order and each element CAN appear zero or one time!

Choices

- Sometimes you might want to declare that any one of a possible group of elements may appear within an element, but not all of them. This is accomplished by using the <choice> element of the XML Schema Definition Language
- The choice element allows only one of the elements contained in the <choice> declaration to be present within the containing element

Syntax:

```
<choice [ID=""] [ref=""] [maxOccurs=""] [minOccurs=""]
    [any attributes... ] >
    (annotation?,(element|group|choice|sequence|any)*)
</choice> Example:
<xsd:element name="Person">
<xsd:complexType>
<xsd:choice>
     <xsd:element name="SingleName" type="Text" minOccurs="1"/>
    <xsd:sequence>
        <xsd:element name="FirstName" type ="text" minOccurs="1"/>
        <xsd:element name="MiddleName" type ="text" minOccurs="0"
            maxOccurs="1" />
        <xsd:element name="LastName" type ="text" minOccurs="1" />
```

```
        </xsd:sequence>
</xsd:choice>
</xsd:complexType>
</xsd:element>
```

Valid XML that conforms to Schema:

(i)
```
<Person>
    <FirstName>Rajarajan</FirstName>
     <LastName>Rajendran</LastName>
</Person>
```

(ii)
```
<Person>
    <FirstName>Ravi</FirstName>
    <MiddleName> Raja</MiddleName>
     <LastName>Rajendran</LastName>
</
Person>
<Person>
    <SingleName>Rajendran</SingleName>
</Person>
```

Sequences

- The <sequence> element specifies that the child elements must appear in the same order within the parent element, as specified in the sequence list. Each child element can occur from 0 to any number of times.

Syntax :

```
<sequence [ID=""] [ref=""] [maxOccurs=""] [minOccurs=""]
    [any attributes... ] >
    (annotation?,(element|group|choice|sequence|any)*)
</sequence>
```

An example:
This example shows a declaration for an element called "personinfo", which must contain the following five elements in order; "firstname", "lastname", "address", "city", and "country":

```
<xs:element name="personinfo">
  <xs:complexType>
    <xs:sequence>
     <xs:element name="firstname" type="xs:string"/>
```

```
      <xs:element name="lastname" type="xs:string"/>
      <xs:element name="address" type="xs:string"/>
      <xs:element name="city" type="xs:string"/>
      <xs:element name="country" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Attribute Groups

- The attributeGroup element is used to group a set of attribute declarations so that they can be incorporated as a group into complex type definitions.
- Here's the basic syntax for the<attributeGroup> element:

```
<attributeGroup [name=""] [ref=""]>
        <attribute …/>

        <attribute …/>

        ...

</attributeGroup> An example given below:

<attributeGroup name="PersonNameExtra">
      <attribute name="honorific">
            <simpleType base="string">
                  < enumeration value ="Mr." />
                  <enumeration value ="Ms." />
                  < enumeration value ="Dr." />
                  <enumeration value ="Rev." />
             </ simpleType>
      </attribute>
      <attribute name="suffix" >
            <simpleType base="string">
                  <enumeration value="Jr." />
                  <enumeration value="Sr." />
            </simpleType>
</attribute>
</attributeGroup>
  <element name="PersonName">
      < complexType>
            <xsd:element name="SingleName" type="Text"
            minOccurs="1"/>
            <attributeGroup ref="personNameExtra" />
```

```
        </complexType>
</element>
```

Valid XML that conforms to Schema:

```
(i) <PersonName honorific =”Mr.” suffix=”Jr.”>
        <SingleName>Rajarajan</SingleName>
    </Person>
```

Targeting Namespaces

You can view an XML schema as a collection of type definitions and element declarations targeted for a specific namespace. Namespaces allow us to distinguish element declarations and type definitions of one schema from another. We can assign an intended namespace for an XML schema by using the targetNamespace attribute on the <schema> element. By assigning a target namespace for the schema, we indicate that an XML document whose elements are declared as belonging to the schema’s namespace should be validated against the XML schema

Example:

```
 <xsd:schema targetNamespace=”http://www.eps-software.com/poschema”
                    xmlns:xsd=”http://www.w3.org/2001/XMLSchema”
                    xmlns=”http://www.eps-software.com/poschema”
                    elementFormDefault=”unqualified”
                    attributeFormDefault=”unqualified”>
```

elementFormDefault and attributeFormDefault. These attributes can possess one of two values:

- **qualified**
- **unqualified**

If a value of unqualified is specified or the elementFormDefault and attributeFormDefault attributes are omitted, the elements or attributes that are not globally declared within the schema (those that are not children of the <schema> element) do not require a prefix within the XML instance document. However, if a value of qualified is specified, all elements and attributes must have a prefix associated with them. For instance, we could make a change to our PurchaseOrder schema and specify that the elementFormDefault and attributeFormDefault attributes have a value of qualified Comparison between DTD and schema

XML Schema <restriction> Element

The restriction element defines restrictions o n a simpleType,

simpleContent, or complexContent definition.
Syntax:
< restriction id=ID base=QName *any attributes*>
…
</restriction>
Example:
This example defines an element called "age" with a restriction. The value of age can NOT be lower than 0 or greater than 100:

```
<xs:element name="age">
 <xs:simpleType>
  <xs:restriction base="xs:integer">
   <xs:minInclusive value="0"/>
   <xs:maxInclusive value="100"/>
  </xs:restriction>
 </xs:simpleType>
</xs:element>
```

XML Schema <extension> Element

- The extension element extends an existing simpleType or complexType element.**Parent elements of <extension>one of the following:** simpleContent, complexContent

Syntax

```
<extension id=ID base=QName any attributes>
 (annotation?,((group|all|choice|sequence)?,((attribute|attributeGroup)*,anyAt
 tribute?)))
</extension>
```

Example

- The following example extends an existing simpleType **"size"** to define new type **"jeans"** by adding an attribute "sex":

```
<xs:simpleType name="size">
 <xs:restriction base="xs:string">
  <xs:enumeration value="small" />
  <xs:enumeration value="medium" />
  <xs:enumeration value="large" />
 </xs:restriction>
</xs:simpleType>
```

```
<xs:complexType name="jeans">
 <xs:simpleContent>
  <xs:extension base="size">
   <xs:attribute name="sex">
    <xs:simpleType>
     <xs:restriction base="xs:string">
      <xs:enumeration value="male" />
      <xs:enumeration value="female" />
     </xs:restriction>
    </xs:simpleType>
   </xs:attribute>
  </xs:extension>
 </xs:simpleContent>
</xs:complexType>
</xs:schema>
```

Example 2: The following example, extends an existing complexType "personInfo" to define new complex type "fullnameinfo" by adding three elements:

```
<xs:complexType name="personinfo">
 <xs:sequence>
  <xs:element name="firstname" type="xs:string"/>
  <xs:element name="lastname" type="xs:string"/>
 </xs:sequence>
</xs:complexType>

<xs:complexType name="fullpersoninfo">
 <xs:complexContent>
  <xs:extension base="personinfo">
   <xs:sequence>
    <xs:element name="address" type="xs:string"/>
    <xs:element name="city" type="xs:string"/>
    <xs:element name="country" type="xs:string"/>
   </xs:sequence>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>
<xs:element name="employee" type="fullpersoninfo"/>
```

Inheriting from Other Schemas

You can define a common piece for multiple XML schemas and extend it from the individual schemas that need it. For this reason, the W3C included the **<include>** and
<import> elements in the XML Schema Definition Language. Through the use of these elements, you can effectively "inherit" elements and attributes from the referenced schema

XML Schema <import> Element

- The import element is used to add multiple schemas with different target namespace to a document. Parent elements for <import> is schema

Syntax:

<import id=ID namespace=anyURI
schemaLocation=anyURI …> (annotation?)
</import>

Attribute	Description
Id	Optional. Specifies a unique ID for the element
Namespace	Optional. Specifies the URI of the namespace to import
schemaLocation	Optional. Specifies the URI to the schema for the imported namespace
any attributes	Optional. Specifies any other attributes with non-schema namespace

- The <import> element doesn't care what the target namespace is in the referenced schema.

Example:

For example, we want the declaration of an <Address> element again and again in multiple schema. However, we wouldn't want to redefine this element in each schema. Instead, it would be nice to have that element declaration and type definition within a separate document as shown in the following example.

Address.xsd

```
<?xml version="1.0"?>
<xsd:schema
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.address.org"
    xmlns=" http://www.address.org "
elementFormDefault="unqualified">
<xsd:complexType name="AddressType">
```

```
<xsd:sequence>
<xsd:element name="street" type="xsd:string"/>
<xsd:element name="area" type="xsd:string"/>
<xsd:element name="city" type="xsd:string"/>
<xsd:element name="city" type="xsd:string"/>
<xsd:element name="pincode" type="xsd:string"/>
</xsd:sequence>
</xsd:complexType>
</xsd:schema>
```

Employee.xsd

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://www.person.org"
            xmlns="http://www.person.org"
            xmlns:adr="  http://www.address.org  "
            elementFormDefault="unqualified">
<xsd:import  namespace="  http://www.address.org  "
schemaLocation="Address.xsd"/>
  <xsd:complexType name="Employee">
    <xsd:sequence>
      <xsd:element name="ENumber" type="xsd:string"/>
      <xsd:element name="EName" type="xsd:string"/>
      <xsd:element name="Address" type="adr:AddressType" />
    </xsd:sequence>
  </xsd:complexType></xsd:schema>
```

Note: taget namespace in both the schema are different. They are:

targetNamespace="http://www.address.org"

targetNamespace="http://www.person.org"

XML Schema <include> Element

- The include element is used to add multiple schemas with the same target namespace to a document.
- **Syntax:**

```
<include                    id=ID
        schemaLocation=anyURI  any
        attributes > (annotation?)
</include>
```

- With included schemas, the included files must all reference the same target namespace. If the schema target namespace doesn't match, the include won't work (or) the <schema> element in the referenced XML schema is empty
- You can "inherit" any and all elements and attributes within the XML schema using the <include> element

Example:

Address.xsd

```
<?xml version="1.0"?>
<xsd:schema     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                        targetNamespace="http://www.person.org"
                        xmlns=" http://www.person.org "
                       elementFormDefault="unqualified">
   <xsd:complexType name="AddressType">
      <xsd:sequence>
        <xsd:element name="street" type="xsd:string"/>
        <xsd:element name="area" type="xsd:string"/>
        <xsd:element name="city" type="xsd:string"/>
        <xsd:element name="city" type="xsd:string"/>
       <xsd:element name="pincode" type="xsd:string"/>
     </xsd:sequence>
   </xsd:complexType>
</xsd:schema>
```

Employee.xsd

```
<?xml version="1.0"?>
<xsd:schema       xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                  targetNamespace="http://www.person.org"
                  xmlns="http://www.person.org"
                  elementFormDefault="unqualified">
                 <xsd:include schemaLocation="Person.xsd"/>
<xsd:complexType name="Employee">
      <xsd:sequence>
        <xsd:element name="ENumber" type="xsd:string"/>
        <xsd:element name="EName" type="xsd:string"/>
        <xsd:element name="Address" type="AddressType" />
      </xsd:sequence>
 </xsd:complexType></xsd:schema>
```

Note: tagetNamespace in both the schema are same, which is **targetNamespace="http://www.person.org"**

11. SCHEMA VS DTD

- There are many differences between DTD (Document Type Definition) and XSD (XML Schema Definition). In short, DTD provides less control on XML structure whereas XSD (XML schema) provides more control.
- The important differences are given below:

Feature	DTD	XSD
	DTD stands for **Document Type Definition**.	XSD stands for XML Schema Definition.
syntax	DTD uses EBNF (Extended Backus–Naur Form) syntax, which is derived from **SGML** syntax.	XSDs are written using XML syntax.
Data type support	Weak: Supports only few data types such as strings, name token, ID, etc - For element, it supports only two types: #PCDATA or CDATA - For attributes, it supports 10 data types	XSD **supports almost all datatypes** all data type available in modern programming language. Define almost 44 data type + your own derived data type'
Name space support	DTD **doesn't support namespace**.	XSD **supports namespace**.
Extensibility	DTD is **not extensible.**	XSD is **extensible**.
inheritance	No. New type cannot be derived from existing type	Yes. New type can be derived from existing type
DOM support	No DOM support	Extensive DOM supports
Content model	**Weak content model**: DTD provides **less control** on XML structure. You can define only simple sequence and choice list. You can only specify zero, one or many child elements	**Strong content model:** XSD provides **more control** on XML structure You can specify exact number of occurance of child elements with minOccurs and maxOccrus attributre
Multiple vocabulary support	No- one DTD per document	Yes – as many as needed, based upon name space in xml

associating with an XML document	using <!DOCTYPE> declaration	Using Namespace declaration with root elemtn of xml document
File suffix	*.dtd	*.xsd
Complexity of structure	Medium	Powerful (e.g. sets, element occurrence constraints)
Adoption	wide spread	Data-centric applications like web services
Dynamic schema	No dynamic support, since DTDs are read only	Schema can be dynamically selected and modified at runtime

- XML Schemas were created to define more precise grammars than with DTDs, in particular one can define Data Types and more sophisticated element structures
- DTD supports 10 datatypes, mostly for attributes. XML Schema supports 44 datatypes and in addition, you can define your own.

CHAPTER 7

THE X-FILES, X-PATH, X-POINTER, AND X-LINK

1. XPATH INTRODUCTION

- XPath is used to navigate through elements and attributes and find data within your XML documents.
- XPath is a major element in W3C's XSLT standard.
- Using XPath, you can select one or more nodes in order to retrieve the data they contain. XPath is used quite extensively with XSLT.

What is XPath?

- XPath is a syntax for defining parts of an XML document
- XPath uses path expressions to navigate in XML documents
- XPath contains a library of standard functions
- XPath is a major element in XSLT
- XPath is a W3C recommendation

XPath Path Expressions

XPath uses path expressions to select nodes or node-sets in an XML document. These path expressions look very much like the expressions you see when you work with a traditional computer file system.

XPath Standard Functions

XPath includes over 100 built-in functions. There are functions for string values, numeric values, date and time comparison, node and QName manipulation, sequence manipulation, Boolean values, and more.

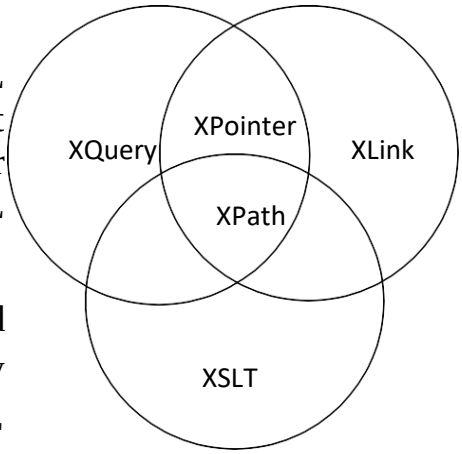
XPath is Used in XSLT

XPath is a major element in the XSLT standard. Without XPath knowledge you will not be able to create XSLT documents.

XPath is a W3C Recommendation

XPath became a W3C Recommendation 16. November 1999.
XPath was designed to be used by XSLT, XPointer and other XML parsing software.

XPath Purpose



- Since XPath is used for finding data within XML documents, this enables you to write applications that make use of the data within an XML document. In order
- to use XSLT (to transform the contents of your XML documents), you need to use XPath.

  Other XML based languages such as XQuery and XPointer also rely on XPath expressions, so XPath really does play an important role when writing XML applications.

2. XPATH TERMINOLOGY

XPath Nodes:

In XPath, there are seven **kinds of nodes**: **element, attribute, text, namespace, processing-instruction, comment**, and **document nodes**.

XML documents are treated as trees of nodes. The top most element of the tree is called the root element.
Look at the following XML document:

```
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>
  <book>
    <title lang="en">Harry Potter</title>
    <author>J K. Rowling</author>
    <year>2005</year>
    <price>29.99</price>
  </book>
</bookstore>
```

Example of nodes in the XML document above:

- <bookstore> (root element node)
- <author>J K. Rowling</author> (element node)
- lang="en" (attribute node)

Atomic values

- Atomic values are nodes with no children or parent.

Relationship of Nodes

- Consider the following example. We will describe relationship of the nodes with the following example:

```
<book>
  <title>Harry Potter</title>
  <author>J K. Rowling</author>
  <year>2005</year>
  <price>29.99</price>
</book>
```

Parent:

- Each element and attribute has one parent.
- In the above example; the **<book>** element is the parent of the **<title>, <author>, <year>, and <price>**

Children:

- Element nodes may have zero, one or more children.
- In the above example; the **<title>, <author>, <year>, and <price>** elements are all children of the **<book>** element

Siblings:

- Nodes that have the same parent.
- In the above example; the **<title>, <author>, <year>, and <price>** elements are all siblings

Ancestors:

- A node's parent, parent's parent, etc.
- In the above example; the **ancestors of** the **<title>** element are the **<book>** element and the **<bookstore>** element

Descendants

- A node's children, children's children, etc.
- In the above example; **descendants** of the **<bookstore>** element are the **<book>, <title>, <author>, <year>, and <price> elements**

XPath Syntax

XPath uses path expressions to select nodes or node-sets in an XML document.
The node is selected by following a path or steps
We will use the following XML document in the examples below.

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

```
<bookstore>
<book>
  <title lang="en">Harry Potter</title>
  <price>29.99</price>
</book>
<book>
  <title lang="en">Learning XML</title>
  <price>39.95</price>
</book>
</bookstore>
```

3. SELECTING NODES

XPath uses path expressions to select nodes in an XML document. The node is selected by following a path or steps. The most useful path expressions are listed below:

Expression	Description
Nodename	Selects all nodes with the name "*nodename*"
/	Selects from the root node
//	Selects nodes in the document from the current node that match the selection no matter where they are
.	Selects the current node
..	Selects the parent of the current node
@	Selects attributes

In the table below we have listed some path expressions and the result of the expressions:

Path Expression	Result
Bookstore	Selects all nodes with the name "bookstore"
/bookstore	Selects the root element bookstore **Note:** If the path starts with a slash (/) it always represents an absolute path to an element!
bookstore/book	Selects all book elements that are children of bookstore
//book	Selects all book elements no matter where they are in the document
bookstore//book	Selects all book elements that are descendant of the bookstore element, no matter where they are under the bookstore element
//@lang	Selects all attributes that are named lang

Predicates

- Predicates are used to find a specific node or a node that contains a specific value.
- Predicates are always embedded in square brackets.
- In the table below we have listed some path expressions with predicates and the

result of the expressions:

Path Expression	Result
/bookstore/book[1]	Selects the first book element that is the child of the bookstore element. **Note:** In IE 5,6,7,8,9 first node is[0], but according to W3C, it is [1]. To solve this problem in IE, set the SelectionLanguage to XPath: *In JavaScript:* *xml*.setProperty("SelectionLanguage","XPath");
/bookstore/book[last()]	Selects the last book element that is the child of the bookstore element
/bookstore/book[last()-1]	Selects the last but one book element that is the child of the bookstore element
/bookstore/book[position()<3]	Selects the first two book elements that are children of the bookstore element
//title[@lang]	Selects all the title elements that have an attribute named lang
//title[@lang='en']	Selects all the title elements that have a "lang" attribute with a value of "en"
/bookstore/book[price>35.00]	Selects all the book elements of the bookstore element that have a price element with a value greater than 35.00
/bookstore/book[price>35.00]/title	Selects all the title elements of the book elements of the bookstore element that have a price element with a value greater than 35.00

Selecting Unknown Nodes

XPath wildcards can be used to select unknown XML nodes.

Wildcard	Description
*	Matches any element node
@*	Matches any attribute node
node()	Matches any node of any kind

In the table below we have listed some path expressions and the result of the expressions:

Path Expression	Result
/bookstore/*	Selects all the child element nodes of the bookstore element
//*	Selects all elements in the document
//title[@*]	Selects all title elements which have at least one attribute of any kind

Selecting Several Paths

By using the | operator in an XPath expression you can select several paths.
In the table below we have listed some path expressions and the result of the expressions:

Path Expression	Result
//book/title | //book/price	Selects all the title AND price elements of all book elements
//title | //price	Selects all the title AND price elements in the document
/bookstore/book/title | //price	Selects all the title elements of the book element of the bookstore element AND all the price elements in the document

XPath Axes

An axis defines a node-set relative to the current node.

AxisName	Result
Ancestor	Selects all ancestors (parent, grandparent, etc.) of the current node
ancestor-or-self	Selects all ancestors (parent, grandparent, etc.) of the current node and the current node itself
Attribute	Selects all attributes of the current node
Child	Selects all children of the current node
Descendant	Selects all descendants (children, grandchildren, etc.) of the current node
descendant-or-self	Selects all descendants (children, grandchildren, etc.) of the current node and the current node itself
Following	Selects everything in the document after the closing tag of the current node
following-sibling	Selects all siblings after the current node
Namespace	Selects all namespace nodes of the current node
Parent	Selects the parent of the current node
Preceding	Selects all nodes that appear before the current node in the document, except ancestors, attribute nodes and namespace nodes
preceding-sibling	Selects all siblings before the current node
Self	Selects the current node

XPath Location Path

To select a node (or set of nodes) in XPath, you use a *location path*. A location path is used to specify the exact path to the node you need to select
Consider the following xml document:

```
<albums>
<rock>
        <title>Machine Head</title>
        <artist>Deep Purple</artist>
```

```
    </rock>
    <blues>
        <title>Greens From The Garden</title>
        <artist>Cory Harris</artist>
    </blues>
    <country>
        <title>The Ranch</title>
        <artist>The Ranch</artist>
    </country>
</albums>
```

- Now, here's a simple XPath expression to select the "title" node:
  albums/rock/title
- The above expression would result in the first "title" node being selected:
  <title>Machine Head</title>
- If we wanted to select the artist instead, we would use this location path:
  albums/rock/artist
- The above expression would select the first "artist" node instead:
  <artist>Deep Purple</artist>

Location Path Expression

- A location path can be **absolute** or **relative.**

Absolute path

- An absolute location path starts with a slash (/) and a relative location path does not. In both cases the location path consists of one or more steps, each separated by a slash:
- An absolute location path:
  /step/step/...
- Where each step is evaluated against the nodes in the current node-set. A step consists of:
  - **an axis** (defines the tree-relationship between the selected nodes and the current node)
  - **a node-test** (identifies a node within an axis)
  - **zero or more predicates** (to further refine the selected node-set)
- The syntax for a location step is:
  axisname::nodetest[predicate]

- Consider the XML document given above. In that, if you wanted to select the **"title"** node of all albums, we could use the following (absolute) location paths:
  albums/rock/title
  albums/blues/title
  albums/country/title
- Here are the nodes that are selected using the above location path.

<title>Machine Head</title>
<title>Greens From The Garden</title>

<title>The Ranch</title>

- Selecting root node:
  - If we wanted to select the root node, we could use either the node's name or a forward slash. Both of these options are absolute location paths and select the root node.

    Option 1 - use the root node's name:
    Albums
    Option 2 - use a forward slash:
    /

Relative path:

- If your location path begins with the name of a descendant, you're using a relative location path. This node is referred to as the *context node*.
- A relative location path is one where the path starts from the node of your choosing- it doesn't need to start from the root node.
- A relative location path:

step/step/...

- Consider the following XML document:
- If we wanted to select the "title" node of all albums, we could use the following (relative) location path:

title

Examples

Example	**Result**
child::book	Selects all book nodes that are children of the current node
attribute::lang	Selects the lang attribute of the current node
child::*	Selects all element children of the current node
attribute::*	Selects all attributes of the current node
child::text()	Selects all text node children of the current node
child::node()	Selects all children of the current node
descendant::book	Selects all book descendants of the current node
ancestor::book	Selects all book ancestors of the current node
ancestor-or-self::book	Selects all book ancestors of the current node - and the current as well if it is a book node
child::*/child::price	Selects all price grandchildren of the current node

XPath Operators

An XPath expression returns either a node-set, a string, a Boolean, or a number.

XPath Operators

Below is a list of the operators that can be used in XPath expressions:

Operator	Description	Example
|	Computes two node-sets	//book | //cd
+	Addition	6 + 4
-	Subtraction	6 – 4
*	Multiplication	6 * 4
Div	Division	8 div 4
=	Equal	price=9.80
!=	Not equal	price!=9.80
<	Less than	price<9.80
<=	Less than or equal to	price<=9.80
>	Greater than	price>9.80
>=	Greater than or equal to	price>=9.80
Or	Or	price=9.80 or price=9.70
And	and	price>9.00 and price<9.90
Mod	Modulus (division remainder)	5 mod 2

XPath functions

Functions on Boolean Values

Name	Description
boolean(*arg*)	Returns a boolean value for a number, string, or node-set Example: boolean('true') Example: boolean(0) Result: true Result: false
not(*arg*)	The argument is first reduced to a boolean value by applying the boolean() function. Returns true if the boolean value is false, and false if the boolean value is true Example: not(true()) Result: false
true()	Returns the boolean value true Example: true() Result: true
false()	Returns the boolean value false Example: false() Result: false

Functions on Nodes

Name	Description
name() name(*nodeset*)	Returns the name of the current node or the first node in the specified node set

local-name() local-name(*nodeset*)	Returns the name of the current node or the first node in the specified node set - without the namespace prefix
namespace-uri() namespace-uri(*nodeset*)	Returns the namespace URI of the current node or the first node in the specified node set
lang(*lang*)	Returns true if the language of the current node matches the language of the specified language Example: Lang("en") is true for ... Example: Lang("de") is false for ...
root() root(*node*)	Returns the root of the tree to which the current node or the specified belongs. This will usually be a document node

Functions on Numeric Values

Name	Description
number(*arg*)	Returns the numeric value of the argument. The argument could be a boolean, string, or node-set Example: number('100') Result: 100
abs(*num*)	Returns the absolute value of the argument Example: abs(3.14) Result: 3.14 Example: abs(-3.14) Result: 3.14
ceiling(*num*)	Returns the smallest integer that is greater than the number argument Example: ceiling(3.14) Result: 4
floor(*num*)	Returns the largest integer that is not greater than the number argument Example: floor(3.14) Result: 3
round(*num*)	Rounds the number argument to the nearest integer Example: round(3.14) Result: 3
round-half-to-even()	Example: round-half-to-even(0.5) Result: 0 Example: round-half-to-even(1.5) Result: 2 Example: round-half-to-even(2.5) Result: 2

Functions on Strings

Name	Description
string(*arg*)	Returns the string value of the argument. The argument could be a number, boolean, or node-set Example: string(314) Result: "314"
compare(*comp1*,*comp2*) compare(*comp1*,*comp2*,*collation*)	Returns -1 if comp1 is less than comp2, 0 if comp1 is equal to comp2, or 1 if comp1 is greater than comp2 (according to the rules of the collation that is used) Example: compare('ghi', 'ghi') Result: 0
concat(*string*,*string*,...)	Returns the concatenation of the strings Example: concat('XPath ','is ','FUN!') Result: 'XPath is FUN!'
string-join(*(string*,*string*,...*)*,*sep*)	Returns a string created by concatenating the string arguments and using the sep argument as the separator Example: string-join(('We', 'are', 'having', 'fun!'), ' ') Result: ' We are having fun! ' Example: string-join(('We', 'are', 'having', 'fun!')) Result: 'Wearehavingfun!' Example:string-join((), 'sep') Result: ''
substring(*string*,*start*,*len*) substring(*string*,*start*)	Returns the substring from the start position to the specified length. Index of the first character is 1. If length is omitted it returns the substring from the start position to the end Example: substring('Beatles',1,4) Result: 'Beat' Example: substring('Beatles',2) Result: 'eatles'
string-length(*string*) string-length()	Returns the length of the specified string. If there is no string argument it returns the length of the string value of the current node Example: string-length('Beatles') Result: 7
normalize-space(*string*) normalize-space()	Removes leading and trailing spaces from the specified string, and replaces all internal sequences of white space with one and returns the result. If there is no string argument it does the same on the current node Example: normalize-space(' The XML ') Result: 'The XML'

upper-case(*string*)	Converts the string argument to upper-case Example: upper-case('The XML') Result: 'THE XML'
lower-case(*string*)	Converts the string argument to lower-case Example: lower-case('The XML') Result: 'the xml'
translate(*string1,string2,string3*)	Converts string1 by replacing the characters in string2 with the characters in string3 Example: translate('12:30','30','45') Result: '12:45' Example: translate('12:30','03','54') Result: '12:45' Example: translate('12:30','0123','abcd') Result: 'bc:da'
escape-uri(*stringURI,esc-res*)	Example: escape-uri("http://example.com/test#car", true()) Result: "http%3A%2F%2Fexample.com%2Ftest#car" Example: escape-uri("http://example.com/test#car", false()) Result: "http://example.com/test#car"
contains(*string1,string2*)	Returns true if string1 contains string2, otherwise it returns false Example: contains('XML','XM') Result: true
starts-with(*string1,string2*)	Returns true if string1 starts with string2, otherwise it returns false Example: starts-with('XML','X') Result: true
ends-with(*string1,string2*)	Returns true if string1 ends with string2, otherwise it returns false Example: ends-with('XML','X') Result: false
substring-before(*string1,string2*)	Returns the start of string1 before string2 occurs in it Example: substring-before('12/10','/') Result: '12'
substring-after(*string1,string2*)	Returns the remainder of string1 after string2 occurs in it Example: substring-after('12/10','/') Result: '10'

matches(*string,pattern*)	Returns true if the string argument matches the pattern, otherwise, it returns false Example: matches("Merano", "ran") Result: true
replace(*string,pattern,replace*)	Returns a string that is created by replacing the given pattern with the replace argument Example: replace("Bella Italia", "l", "*") Result: 'Be**a Ita*ia' Example: replace("Bella Italia", "l", "") Result: 'Bea Itaia'
tokenize(*string,pattern*)	Example: tokenize("XPath is fun", "\s+") Result: ("XPath", "is", "fun")

2. XLINK

- XLink is used to create hyperlinks within XML documents
- Any element in an XML document can behave as a link
- With XLink, the links can be defined outside the linked files
- XLink is a W3C Recommendation

XLink Attribute Reference

Attribute	Value	Description
xlink:actuate	onLoad onRequest other none	Defines when the linked resource is read or retrieved and shown. The possible values are: • **onLoad** - the resource should be loaded and shown when the document loads • **onRequest** - the resource is not read or shown before the link is clicked
xlink:title	*Some text*	It is used in extended link. This attribute provides a human readable label
xlink:role	*qName*	It is used in extended link. This attribute provides a machine readable label
xlink:href	*URL*	Specifies the URL to retrieve external resource
xlink:show	embed new replace other none	Specifies where to open the link. Default is "replace" **replace** – replace the current content with target resource **new** – open in the new window **embed**- embed target resource along with current content. It is like <img> tag in html

xlink:type	simple extended locator arc resource title none	Specifies the type of link **Simple** - used to create external link similar to <a href> in Html **Extended** – used to create link with m multiple resource. All other types are as child-element of extended link
xlink:from & xlink:to	qName	The xlink:from and xlink:to attributes are used only with extended link. The value of xlink:from and xlink:to attribute are qName They are used to define direction of the link

- role and title attributes are called semantic attribute
- actuate and show attirbute are called behavior attributes

Types of links:

- xlink define two main types of link **(i)simple link (ii) extended link** **Simple link**
- It provides the same functionality as hyperlink(<a> - anchor element) provided to html
- They are the one way link, involving only two resources: the source and destination
- example of simple link given below:

```
<?xml version="1.0"?>
<person xmlns:xlink="http://www.w3.org/1999/xlink">

<name xlink:type="simple" xlink:href="name.xml" xlink:actuate="onLoad"
                          xlink:show="embed"/>

<picture xlink:type="simple"      xlink:href="picture.jpg"
         xlink:actuate="onLoad"   xlink:show="embed"

         xlink:title="Click to see picture!"> Click here to see a picture!  </picture>

 <homepage xlink:type="simple"

              xlink:href="http://www.xyz.com/homepages/persona1.htm"
             xlink:actuate="onRequest"

             xlink:show="replace">Click here for the homepage</homepage>

</person>
```

Extended link

- Extended link is the one that associate any number of resources.

- They allow you to specify complex traversal rule between various resources
- An external link can be either inline or out of line

(i) **inline-** if link's one of the resources is in the same xml document then it is referred to as inline

(ii) **out of line –** if all of the link recources are remote

- an extended resource con has one or more child elements, which define the local and remote resources participating in the extended link, traversal rules for those resources
- extended type element can have the semantic attributes: **role** and **title**, which define machine and human readable label for the link

locator type elements

- They are child element of extended type elements
- It is used to indicate remote resource taking part in an extended link
- locater type element must include ***href*** attributres. Optionally it can include title and role attribute alos

arc type elements

- arc type element define the direction in which the link must traverse using ***from*** and ***to*** attributes
- it also define the behavior that the link follow when it retrieve the resources using ***show*** and ***actuate*** attributes

- except href, all other xlink's attributes can be used in arc-type element

resource type elements

- They are used to create local resource
- An extended link can be inline, if atleast one of the resource specified is local
- These types of element can only have ***role*** and ***title*** attribute, both are optional

title type elements

- the benfit of using title type element, instead of title attribute is that the elements such as **<B>,<U>** can included within the text

example: The example, define extended link, with about three resource-type elements and three arc-type-elements and two locator-type elements

```
<PartNumber xmlns:xlink="http://www.w3.org/1999/xlink"
        xmlns:op="http://sernaferna.com/OrderProcessingSystem"
        xlink:type="extended">
  <item xlink:type="resource"  xlink:role="op:item"          xlink:title="Item">
    <part-number>E16-25A</part-number>
    <description>Production-Class Widget</description>
  </item>
```

```
 <salesperson xlink:type="locator"
 xlink:href="http://sernaferna.com/order256.xml#xpointer(/order/name)"
 xlink:role="op:salesperson"          xlink:title="Salesperson"/>
 <order xlink:type="locator" xlink:href="http://sernaferna.com/order256.xml"
          xlink:role="op:order" xlink:title="Order"/>
 <GetOrder xlink:type="arc"           xlink:from="op:salesperson" xlink:to="op:order"
              xlink:show="replace" xlink:actuate="onRequest"
              xlink:role="op:GetOrder" xlink:title="Last order processed."/>
 <GetSalesperson xlink:type="arc"           xlink:from="op:order"
                        xlink:to="op:salesperson" xlink:show="replace"
           xlink:actuate="onRequest" xlink:role="op:GetSalesperson"
           xlink:title="Salesperson's name"/>
 <GetItemOrder xlink:type="arc" xlink:from="op:item" xlink:to="op:order"
          xlink:show="new" xlink:actuate="onRequest" xlink:role="op:GetItemOrder"
          xlink:title="Last order placed for this item"/>
</PartNumber>
```

4. XPOINTER

- Pointer allows links to point to specific parts of an XML document
- XPointer uses XPath expressions to navigate in the XML document
- XPointer is a W3C Recommendation

XPointer Example

- In this example, we will use XPointer in conjunction with XLink to point to a specific part of another document.
- Instead of linking to the entire document (as with XLink), XPointer allows you to link to specific parts of the document. To link to a specific part of a page, add a number sign (#) and an XPointer expression after the URL in the xlink:href attribute, like this: **xlink:href="http://dog.com/dogbreeds.xml#xpointer(id('Rottweiler'))".** The expression refers to the element in the target document, with the id value of "Rottweiler"
- The following XML document contains links to more information of the dog breed for each of my dogs:

```
<?xml version="1.0" encoding="UTF-8"?>
<mydogs xmlns:xlink="http://www.w3.org/1999/xlink">
<mydog>
 <description>
 Anton is my favorite dog. He has won a lot of.....
 </description>
 <fact xlink:type="simple" xlink:href="http://dog.com/dogbreeds.xml#Rottweiler">
```

```
 Fact about Rottweiler
 </fact>
</mydog>
<mydog>
 <description>
 Pluto is the sweetest dog on earth.....
 </description>
 <fact                                                    xlink:type="simple"
 xlink:href="http://dog.com/dogbreeds.xml#FCRetriever">

 Fact about flat-coated Retriever
 </fact>
</mydog>
</mydogs>
```

XPointer short hand syntax:

In addition to the full Xpointer syntax, there are also a couple of shorthand syntaxes available. They are

- **The Bare name syntax**
- **The child sequence syntax**

Consider the following xml document, with an ID attribute(names.xml):

```
<?xml version= "1.0 " ?>
<!DOCTYPE order [       <!ELEMENT order (name, item, quantity,date)>
                         <!ELEMENT name (first, middle, last)>
                         <!ELEMENT first (#PCDATA)>
                         <!ELEMENT middle (#PCDATA)>
                         <!ELEMENT last (#PCDATA)>
                         <!ELEMENT item (#PCDATA)>
                         <!ELEMENT quantity (#PCDATA)>
                         <!ELEMENT date (#PCDATA)>
                         <!ATTLIST  first id ID #REQUIRED>        ]>
<order>
<name>
        <first id="section-1">John</first>
        <middle>Kennady</middle>
        <last>Doe</last>
</name>
        <item>production-class widget</item>
<quantity>16</quanity>
<date>12.12.15</date>
</order>
```

Assume this file is stored in URI: http://www.xyz.com

Full XPointer syntax:

- The common usage of XPointer will be retrieving an element from an XML document based on its **ID**. The full syntax for the href attribute would be like this:

  http://www.xyz.com/names.xml#xpointer(id("section-1"))

- This will retrieve an element which have an attribute of ID type with a value of attribute as **"section-1"**; in this case, **<first>** element will be retrieved

The Bare names syntax:

- The bare name syntax allows us to retrieve an element by its ID with value as "section=1" like this:

  http://www.xyz.com/names.xml#section-1

- This will produce the same result as full xpointer example. That is the **<first>** element will be retrieved
- XPointer provides this shorthand syntax is to provide mechanism which is similar to HTML's method of retrieving a document

The child sequence syntax:

- In the third way, we can specify XPointer expression at the end of the URI is to use a child sequence:

  http://www.xyz.com/names.xml#/1/1/2

- which says to select the second child of the first child of the document root. In this case, it is the **<middle>** element
- The number after **/** are child element numbers of previously selected elements
- This expression means that **"the second child element of first child element of first child element".** When used right at the beginning of the child sequence, /1 means the root element. This XPointer expression is equivalent to:

  http://www.xyz.com/names.xml#xpointer(/*[1]/*[1]/*[2])

- you can also append a child sequece to the Bare name expression as shown in the following:

  http://www.xyz.com/names.xml#section-1/3

this will take the third child element of the element with an ID of section-1

note: both child sequence and bare names syntaxes can be used to retrive element node types from xml document; for other node types, you need to use the full XPointer syntax

Using muliple xpointer expressions

- XPointer reads the expression from left to right. If one expression fails, the next

expression is evaluated

- The result of first successfully executed expression is taken and the rest is ignored
- Consider the following example:

XPointer(id("section-1"))xpointer(//*[@id="section-1"])

- If there is a DTD associated with this xml document, which define ID attribute then the first expression **XPointer(id("section-1")) is** evaluated and result will be returned
- Otherwise, evaluation of first expresion fails, so that second expression will be evaluated **xpointer(//*[@id="section-1"]),** its result will be returned

Location Point and range

- To address non-node locations, XPointer defines two new location types, **point** and **range**, that can appear in location-sets and can be operated on by XPath node tests and predicates.
- Points and ranges can be used as XPointer context locations. This allows the XPointer [] operator to be used to select from sets of ranges. Also, a point as a context location, when followed by a **range-to**function, selects a range

<u>Definition of Point Location</u>

- A point is simply a spot in the xml document. It is defined using the usual XPointer expression
- There are two pieces of information needed to define a point: a **container node** and an **index**
- Points are located between two elements or between two characters in a CDATA section
- Whether the point refers to characters or elements depends on the nature of the container node
- An index of zero indicates the point before any child nodes and a non-zero index *n* indicates the point immediately after the n^{th} child node.
- When container node is an element then the index becomes an index of the child elements and point is called *node point*
- **In the following example, the container node is <name> element:**

<name> ←——————————————— **Container node**

<first>John</first>

<middle>Kennady</middle> **node point**

<last>Doe</last>

</name>

- **The XPointer expression #xpointer(/name[2])** indicates the point right after second child element of **<name>,** which is **<middle>** element
- If the containder is any other node-type, the index refers to the characters of the string value of that node, and the point is called character-point. In the following example, the container node is PCDATA child of <middle>:

character-point

<name>

<first>John </first>

<middle>Kennady</middle>

<last>Doe</last> **Container node**

</name>

- **The XPointer expression #xpointer(/name/middle/text( )[2])** indicates the point right after the '***e***' nd right before '***n'*** of **Kennady**

Ranges:

- A range is defined by two points- a **start point** and an **end point** – and consist of all of the XML structure and content between those two points
- The **start point** and **end point** must both be in the same document. If the start and end point are equal then the range is called collapsed range
- If the container node of either point is other than an element node, text node, document, root node, then the container node of other point must be the same
- The following range is valid, because both start point and end point are in the same PI: **start point** **end point**

<root>

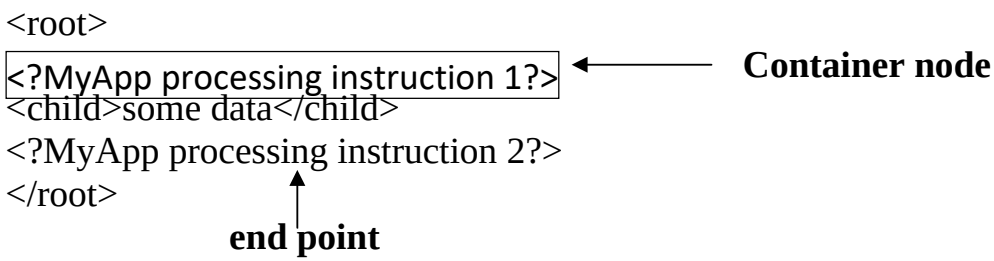
<?MyApp processing instruction 1?> **Container node**

<child>some data</child>

<?MyApp processing instruction 2?>

</root>

- The **following range is invalid**, because both start point and end point are in the same PI:



How do we select ranges using xpointer

- XPointer adds the keyword ***to*** which we can insert in our XPointer expression to specify a range. It is used as follows:

  http://www.xyz.com/order.xml**#xpointer(/order/name to /order/item)**

- This select a range where start point is before **<name>** element, and end point is just after the **<item>** element:

<?xml version="1.0" ?>
<order> **start point**

<name>

<first>John</first
>

<middle/>

<last>Doe</last>

</name>

<item>production-class **end point**
widget</item>

<quantity>16</quanity>

<date>12-1-2016</date>

<customer>ram kumar</customer>

</order>

Range with multiple location

- if the expression on eiterh side of the "***to***" keyword return multiple location in their location set the thing get complicated. Let's explain with an example using following XML document:

```
<people>
 <person name="John">
  <phone>(555)555-1212</phone>
  <phone>(555)555-1213</phone>
 </person>
 <person name="David">
  <phone>(555)555-1214</phone>
 </person>
 <person name="Andrea">
  <phone>(555)555-1215</phone>
  <phone>(555)555-1216</phone>
  <phone>(555)555-1217</phone>
 </person>
 <person name="Ify">
  <phone>(555)555-1218</phone>
  <phone>(555)555-1219</phone>
 </person>
 <!--more people could follow-->
</people>
```

In this we have list of people, and each person have one or more phone numbers. Consider the following XPointer:

#xpointer(//person to phone[1])

The first expression on the left side of to keyword will return number of <person> elements, the second expression will return the first <phone> element.

XPointer tackles this as follows:

(1) it evaluate the expression on the left side of ***to*** keyword, and saves the location set obtained from expression. In this case, the location set will have all of **<person>**elements :

```
<person name="John">
<person name="David">
<person name="Andrea">
```

Location set

(2) using the first location in that set as context location, XPath then evaluate the expression on the right side of ***to*** keyword. In this case it will select first <phone> child of **<person>**element in the location set on the left:

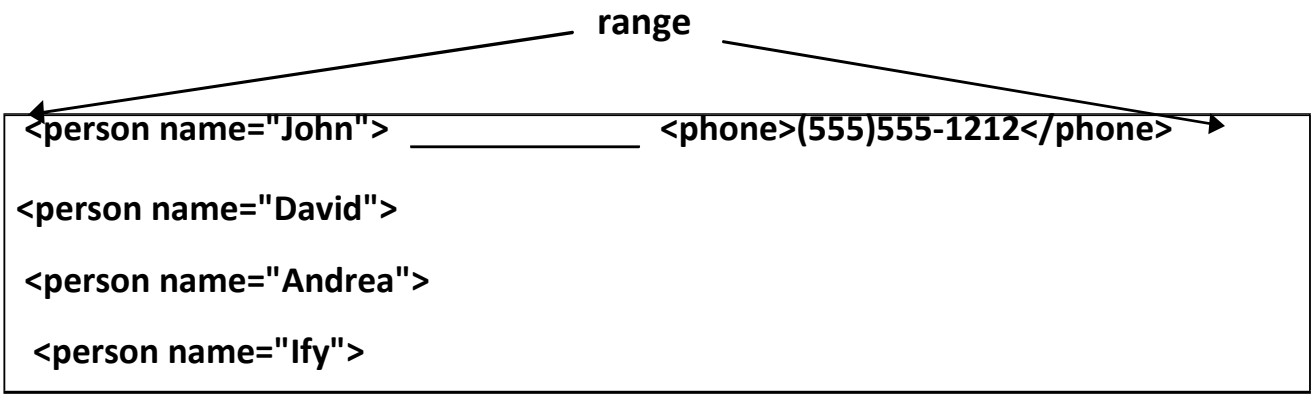
<person name="John"> _____ <phone>(555)555-1212</phone>

<person name="David">

<person name="Andrea">

<person name="Ify">

(3) for each location in this second location set , XPointer adds the range to the result, with start point beginning of location to the first location set , and end point at the end of the location in the second location set. In this case, only one range will be created, since the



second returned only one location

(4) step (2) and (3) repeated for each location in the first location set, with all additional range being added to the result. So that the result of Xpointer will return the following pieces of xml selected in our document

<person name="John">
<phone>(555)555-1212</phone>
<person name="David">
<phone>(555)555-1214</phone>
<person name="Andrea">
<phone>(555)555-1215</phone>
<person name="Ify">
<phone>(555)555-1218</phone>

CHAPTER 8

XSL

1. INTRODUCTION

XSL stands for EXtensible Stylesheet Language, and is a style sheet language for XML documents. XSLT stands for XSL Transformations. The World Wide Web Consortium (W3C) developed XSL.

XSL Technologies

XSL has two independent languages:

- XSLT - a language for transforming XML documents
- XSL-FO - a language for formatting XML documents (discontinued in 2013)

XLST extensively uses the following to retrieve data from XML documents:

- XPath - a language for navigating in XML documents
- XQuery - a language for querying XML documents

What is XSLT?

XSLT is a language for transforming XML documents into XHTML documents or to other XML documents.

- XSLT stands for XSL Transformations
- XSLT is the most important part of XSL
- XSLT uses XPath to navigate in XML documents
- XSLT is a W3C Recommendation

XSLT = XSL Transformations

- XSLT is used to transform an XML document into another XML document, or another type of document that is recognized by a browser, like HTML and XHTML. Normally XSLT does this by transforming each XML element into an (X)HTML element.
- With XSLT you can add/remove elements and attributes to or from the output file. You can also rearrange and sort elements, perform tests and make decisions about which elements to hide and display, and a lot more.

XSLT Uses XPath

- XSLT uses XPath to find information in an XML document. XPath is used to navigate through elements and attributes in XML documents.

How Does it Work?

- In the transformation process, XSLT uses XPath to define parts of the source document that should match one or more predefined templates. When a match is found, XSLT will transform the matching part of the source document into the result

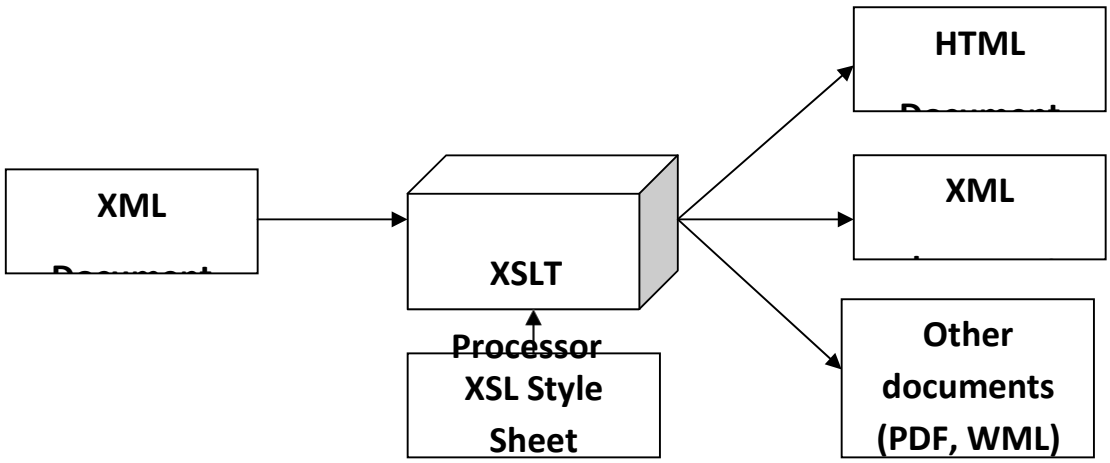
document.

XSLT is a W3C Recommendation

- XSLT became a W3C Recommendation 16. November 1999.

2. PROCESS OF XSLT

XSLT provides the mechanism for converting an XML document to another format. This is accomplished by applying an XSLT style sheet to the XML document. The style sheet contains conversion rules for accessing and transforming the input XML document to a different output format. An XSLT processor is responsible for applying the rules defined in the style sheet to the input XML document. The process is illustrated in the following Figure:



Transforming / Converting XML Document into HTML Document

This process involves four steps:

Step1: Creating the XML document that need to be transformed into HTML document
Step2: Creating the XSL Style Sheet that formatting rule about how to convert xml data into HTML document
Step3: link the XSL style sheet with XML document
Step4: viewing the transformed result XML document in browser

Step1: Creating the XML document – stock.xml

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="stock.xsl"?>
 <portfolio>
  <stock exchange="NSC">
        <name> HDFC bank </name>
        <symbol>HDFC</symbol>
        <price>28.875</price>
  </stock>
  <stock exchange=" NSC ">
```

```
        <name> ICICI BANK </name>
        <symbol> ICICI </symbol>
        <price>92.250</price>
  </stock>
  <stock exchange=" NSC ">
        <name> Coal India Ltd </name>
        <symbol>COALINDIA</symbol>
        <price>20.313</price>
  </stock>
</portfolio>
```

2. Creating the XSL Style Sheet – stock.xsl

- **In this section,** we'll convert an XML document to an HTML document. The XML document contains a list of companies in stock market:

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="stock">
   <DIV STYLE="font-weight:bold">
      Symbol: <xsl:value-of select="symbol" />,
      Price: <xsl:value-of select="price" />
   </DIV>
</xsl:template>
</xsl:stylesheet>
```

- The XML document, stock.xml contains element **<portfolio>,** which contain one or more child element as **<stock>. <stock>** contain the following child elements:**<name>**, **<symbol>**, **<price>.** Stock.xsl file contain transformation rule, that only retrieve values of **<symbol> and <price>** then display the result as HTML document
- In this example, we will apply the style sheet in a client-side Web browser. The XML document makes a reference to a style sheet using the following code:

Step3: link the XSL style sheet with XML document

- Linking of stock.xsl with stock.xml is done by inserting the following code as second line in xml documment

```
<?xml-stylesheet type="text/xsl" href="stock.xsl"?>
```

Step4: viewing the transformed result XML document in browser

- To see the transformed output, open the stock.xml file with XSLT complaint browser
- The output is shown in the following figure:

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

<?xml-stylesheet type="text/xsl"

                href="stock.xsl"?>

 <portfolio>

  <stock exchange="NSC">
```

3. THE XSLT PROCESSOR

XSLT processors are widely available. When you select an XSLT processor, you must ensure that it is fully compliant with the XSLT 1.0 specification. Table contains a list of the most popular XSLT 1.0–compliant processors

Company	Product	Web site address
Apache	**Xalan-J 1.2.2**	xml.apache.org
Microsoft	**MS XML Parser 3.0**	msdn.microsoft.com
Sun Microsystems	**JAXP 1.1**	java.sun.com/xml
James Clark	**XT**	www.jclark.com/xml/

Techniques available for performing the XSLT processing

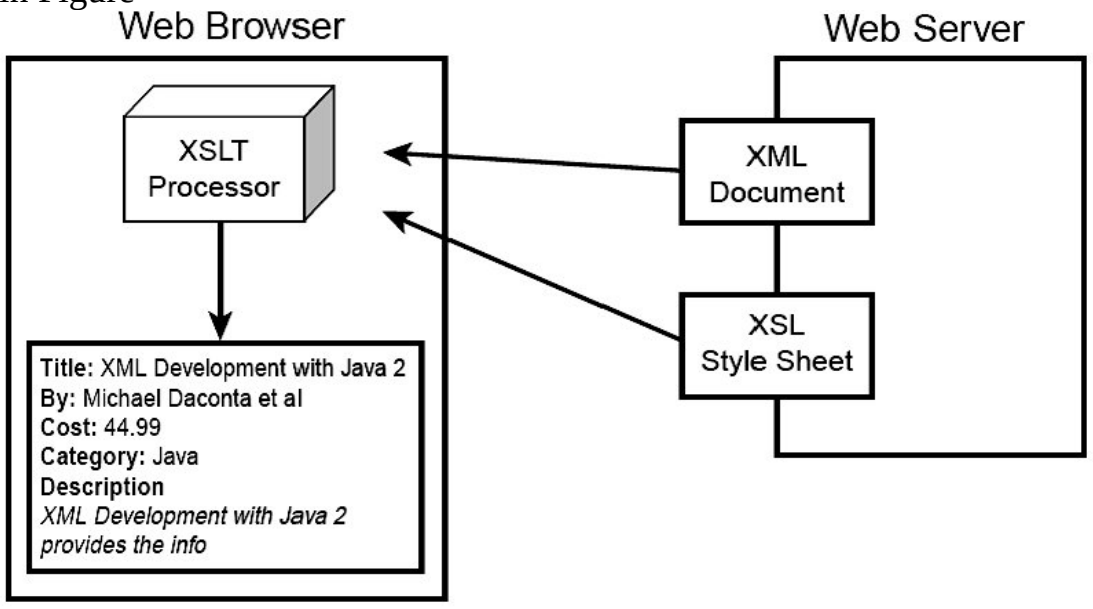
There are two techniques are available for performing the XSLT processing. They are:

(i) client-side XSLT processing

(ii) server-side XSLTprocessing

client-side XSLT processing

- Client-side XSLT processing commonly occurs in a Web browser. The Web browser includes an XSLT processor and retrieves the XML document and XSL style sheet, as shown in Figure



- The client-side technique offloads the XSLT processing to the client machine. This minimizes the workload on the Web server.

- **Netscape Communicator 1.0 specification** and **Microsoft Internet Explorer 6** support the XSLT
- The client-side technique-controlled environment is applicable when you're deploying an application in a For example, in a corporate environment, the system

  administrators can install the latest version of the Web browser that conforms to the XSLT 1.0 specification

Disadvantage:

- the disadvantage is that the Web browser must provide XSLT support

Implementing Client-Side XSLT Processing

- Implement client-side XSLT processing, you will need a browser that supports XSLT 1.0, such as Netscape Communicator 6 or Microsoft Internet Explorer 6
- For client-side p r o c e s s i n g , the XML document requires a special processing instruction to reference the XSL style sheet, *as in the following example*

  <?xml-stylesheet type="text/xsl" href="book_view.xsl"?>

- The processing instruction is <?xml-stylesheet>, and it has two attributes: ***type*** and ***href***

- Store the following two files **book.xml** and **book_view.xsl** in the virtual directory of Java Web Server (OR) IIS

book.xml

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="book_view.xsl"?>
<book>
        <author>Michael Daconta et al</author>
        <title>XML Development with Java 2</title>
        <category>Java</category>
        <price currency="USD”>44.99</price>
        <summary>
                XML Development with Java 2 provides the information
                and techniques a Java developer will need to integrate XML
                into Java-based applications.
        </summary>
</book>
```

book_view.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:template match="/book">
<html><body>
<b>Title: </b> <xsl:value-of select="title" /><br/>
<b>By: </b> <xsl:value-of select="author" /><br/>
<b>Cost: </b> <xsl:value-of select="price" /><br/>
<b>Category: </b> <xsl:value-of select="category" /><br/>
<b>Description</b><br/>
<i><xsl:value-of select="summary" /></i>
</body></html>
</xsl:template>
</xsl:stylesheet>
```

- Now open the browser, on the client machine and type the following address of web server:

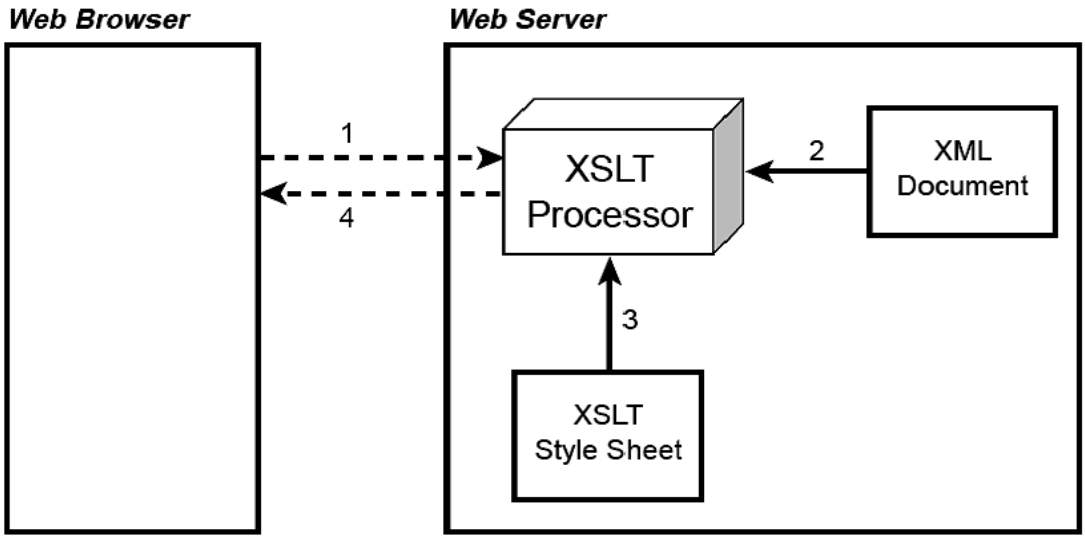
  http://localhost/xampp/book.xml

- The output is shown below:

SERVER-SIDE XSLT PROCESSING

Server-side XSLT processing occurs on the Web server or application server. A serverside process such as an Active Server Page (ASP), JavaServer Page (JSP), or Java servlet will retrieve the XML document and XSL style sheet and pass them to an XSLT processor. The output of the XSLT processor is sent to the client Web browser for

presentation. The output is generally a markup language, such as HTML, that is understood by the client browser. The application interaction is illustrated in Figure



When to go for server-side processing

- If you are deploying the application on an extranet or the Internet, you will probably have little control over the type/version of browser installed on the client machines. If this is the case, you should implement the server-side technique

Advantage:

- An advantage of the server-side technique is browser independence, since the output document to the browser is simply an HTML file
- This technique supports the older browser versions and makes the application more robust and versatile
-

IMPLEMENTING SERVER-SIDE XSLT PROCESSING

To implement the server-side processing technique, *many* number of server-side technologies are available, including the following:

- **Common Gateway Interface (CGI),**
- **ColdFusion**,
- **Hypertext Processor (PHP)**, and so on.

- We focuses on server-side processing with
  - **Microsoft's Active Server Pages (ASP)** and
  - **Sun Microsystem's JavaServer Pages (JSP)**

ASP: Server-Side XSLT Processing

- In order to develop using ASP, you will need the IIS Web server and the latest version of the Microsoft XML parser. The required components are listed below:

- ***Microsoft IIS Web Server 5.0***. This version of IIS is included with Microsoft Windows 2000 Professional. You can also use IIS 4.0 or **Personal Web Server (PWS)**; however, you will have to install the Windows NT Option Pack 4
- ***Microsoft XML Parser 3.0***. If you have IE 6 installed on your server machine, then MS XML Parser 3.0 is included

Steps:

Create a virtural directory called "**ServerSideXSLT"**. In that copy both "**book.xml"** and "**book_view.xsl"**. Then create the following "**book_test.asp**" file

book_test.asp

```
<%@ Language=VBScript %>
<%    set xml = Server.CreateObject("Microsoft.XMLDOM")
      xml.load(Server.MapPath("book.xml"))
      set xsl = Server.CreateObject("Microsoft.XMLDOM")
      xsl.load(Server.MapPath("book_view.xsl"))
      Response.Write(xml.transformNode(xsl))          %>
```

- Now open the browser and type the following in the address bar: **http://localhost/ServerSideXSLT/book.xml**
- The output of the **book_test.asp** is shown below

JSP: Server-Side XSLT Processing

- Sun Microsystems provides a server-side technology that is very similar to ASP. Of course, the server-side scripting is accomplished in Java.
- In order to perform the serverside processing with JSP, you will need to install the Java Software Development Kit (SDK) along with a compliant JSP server container
- Here's a list of required components:
- Sun Microsystems' Software Development Kit (SDK) 1.3 (or higher).
- *Apache Tomcat Server 4*. Apache Tomcat 4 is the official reference implementation for JSP 1.2 and Java Servlets 2.3.
- Once Tomcat 4 is installed, you need to add a new Web application that points to the

source code directory.

- This is accomplished by editing the file <tomcat_install_dir>\conf\server.xml. Move to the section where the <Context> elements are listed and then add the following entry:

<Context path="/bookch9" docBase="<install_dir>/public_html"
debug="0" reloadable="true" />

- Restart the Tomcat server to pick up the new configuration. By default, the Tomcat server is listening on port 8080. You can access files for the bookch9 Web application using the URL **http://localhost:8080/book_test.jsp**
- This example makes use of a JSP custom tag for the XSLT processing. A JSP custom tag is a special tag that is created by a developer. When the JSP server encounters the custom tag, it executes the handler code associated with the tag. JSP custom tags are conceptually similar to ASP server objects
- The Apache **<jakarta:apply>** tag provides the XSLT processing. The JSP code example "***book_test.jsp***" shown below utilizes the <jakarta:apply> tag

<%@ taglib uri="http://jakarta.apache.org/taglibs/xsl-1.0" prefix="jakarta" %>

<jakarta:apply xml="book.xml" xsl="book_view.xsl" />

4. COMPONENTS OF XSLT DOCUMENT

- An XSLT document is a valid XML document. An XSLT document consists of a number of elements/tags/attributes. These can be XSL elements or elements from another language (such as HTML).
- XLST Document can contain the following components:
    - **XML prolog / XML Declaration**
    - **< xsl:stylesheet> or < xsl:transform> element**
    - **<xsl:template> element**
    - **<value-of> Element**
    - **<xsl:for-each> element**
    - **<xsl:sort> element**
    - **<xsl:if> element**

XML prolog

- XSL documents are also XML documents and so we should include the XML version in the document's prolog. We should also set the standalone attribute to "no" as we now rely on an external resource (i.e. the external XSL file).

<?xml version="1.0" standalone="no"?>

< xsl:stylesheet> Element

- It is root element of every XSLT file. The root element needs to include the XSL version as well as the XSL namespace

- The root element that declares the document to be an XSL style sheet is **<xsl:stylesheet> or <xsl:transform>**
- An XSL style sheet has the following general structure:

  <?xml version="1.0"?>
  <xsl:stylesheet xmlns:xsl="*URI*" version="1.0">
  <!--XSLT CONVERSION RULES-->
  </xsl:stylesheet>
- The xmlns attribute is the namespace definition. The XSL Transformation engine reads the ***xmlns*** attribute and determines whether it supports the given namespace. The ***xmlns*** attribute specifies the XSL prefix. All XSL elements and types in the document use the prefix
- The XSLT 1.0 specification defines the following URI for the XSL namespace: **http://www.w3.org/1999/XSL/Transform**

  The correct way to declare an XSL style sheet according to the W3C XSLT Recommendation is:

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

  (OR)

  <xsl:transform version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
- **Note:** <xsl:stylesheet> and <xsl:transform> are completely synonymous and either can be used
- The XSL style sheet contains HTML text and XSL elements. The HTML text forms the basis of the desired output page. The XSL elements are template rules for the XSLT processor. A template is associated with a given element in the XML document

XSL Namespace Prefix

- All XSL elements in your XSLT document must include the *XSL* prefix.

Syntax: <xsl:element_name>

Example:

```
<xsl:template match="/">
....
</xsl:template>
```

XSLT <template> Element

- A template contains rules to apply when a specified node is matched. Following is the syntax declaration of **<xsl:template> element:**

<xsl:template name= Qname match = Pattern priority = number mode = QName>

…

</xsl:template>

Attributes

Name	Description
Name	Name of the element on which template is to be applied.
Match	Pattern which signifies the element(s) on which template is to be applied.
Priority	Priority number of a template. Matching template with low priority is not considered in front of high priority template.
Mode	Allows element to be processed multiple times to produce a different result each time

XSLT <value-of> Element

- The <xsl:value-of> element is used to extract the value of a selected node.
- The <xsl:value-of> element can be used to extract the value of an XML element and add it to the output stream of the transformation
- Following is the syntax declaration of **<xsl:value-of>** element:

  <xsl:value-of select = Expression disable-output-escaping = "yes" | "no" />

Attributes

Name	Description
Select	It is required attributes. The Expressions (XML) to be evaluated against the current context. The results are converted to a string, as by a call to the string() function. A node-set is converted to a string by inserting the string value of the first node in the set
disable-output-escaping	Default is "no". If the value is "yes", a text node generated by instantiating the <xsl:value-of> element will be output without any escaping. For example, the following generates the single character "<". **<xsl:value-of disable-output-escaping="yes" select="string('<')"/>**

<xsl:for-each> element – for looping

- It is used for looping through a list of elements. This is very useful when you have a collection of related items and you'd like to process them in a sequential fashion. The <xsl:for-each> element is commonly used in the Web development world to convert an XML document to an HTML table
- Here's the syntax for <xsl:for-each>:

  <xsl:for-each select=node-set-expression>
  <!-- content -->
  </xsl:for-each>
- The <xsl:for-each> element has a required attribute: select. The value of the select attribute is an expression. The expression contains an XPath expression for selecting the appropriate elements from the list
- See the following code snippet:

  <xsl:for-each select="booklist/book" >
  <tr>

<td> <xsl:value-of select=”author” /> </td>
<td> <xsl:value-of select=”title” /> </td>
<td> <xsl:value-of select=”category” /> </td>
<td> <xsl:value-of select=”price” /> </td>
</tr>
</xsl:for-each>

Sorting

- In XSLT, the <xsl:sort> element is used for sorting the XML data. It is possible to sort based on a single key or multiple keys.
- The syntax for the <xsl:sort> element is shown here:

<xsl:sort select = *string-expression*
order = { “ascending” | “descending” } data-
type = { “text” | “number” }
case-order = {“upper-first” | “lower-first” } lang
= { nmtoken } />

Sorting with Multiple Keys

- In certain situations, you might want to sort using multiple keys. For example, you could sort the books by category and then by price. This is accomplished by inserting multiple **<xsl:sort>** elements within an **<xsl:for-each>** element
- The **<xsl:sort>** element is used in conjunction with the **<xsl:for-each>** element. For example, the following code snippet sorts the book based on category. Then records of each category sorted based on price:

<xsl:for-each select=”booklist/book”>
<xsl:sort select=”category” order=”descending”/>
<xsl:sort select=”price” order=”ascending” data-type=”number” />

<tr>
<td> <xsl:value-of select=”author” /> </td>
<td> <xsl:value-of select=”title” /> </td>
<td> <xsl:value-of select=”category” /> </td>
<td> <xsl:value-of select=”price” /> </td>
</tr>
</xsl:for-each>

Conditionals(formatting)

- During an XSLT transformation, the style sheet can perform conditional tests on the data. XSLT contains a very simple if-then conditional. The syntax for the <xsl:if> element is shown here:

<xsl:if test=Boolean-expression>
<!-- content -->
</xsl:if>

The test attribute refers to a Boolean expression. If the Boolean expression evaluates to true, the content within the <xsl:if> element is included in the output
The following code snippet performs a test for Fiction-Thriller books:

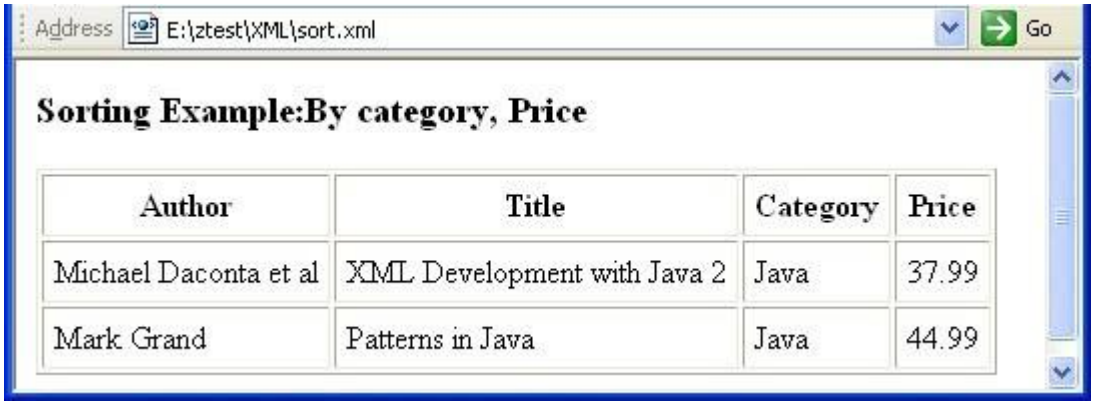
```
<xsl:for-each select="booklist/book" >
<tr>
<xsl:if test="category='Fiction-Thriller'">
<xsl:attribute name="bgcolor">pink</xsl:attribute>
</xsl:if>
<td> <xsl:value-of select="author" /> </td>
<td> <xsl:value-of select="title" /> </td>
<td> <xsl:value-of select="category" /> </td>
<td> <xsl:value-of select="price" /> </td>
</tr>
</xsl:for-each>
```

If a Fiction-Thriller book is found, the background color of the row is set to ***pink***. In this example, we've introduced a new XSLT element, <xsl:attribute>. The <xsl:attribute> element creates a new attribute for the parent element. In this example, the parent is the <tr> element
If the conditional is true, the <tr> element will have the attribute bgcolor and its value set to red. The end result is <tr bgcolor="pink">.

Filters

Using XSLT, you can also filter the data based on a given expression. When data is selected using the <xsl:for-each> element, the expression can contain a filter. For example, you can filter the data to contain only Java books. The following code snippet performs the desired operation:

```
<xsl:for-each select="booklist/book[category='Java']" >
<tr>
<td> <xsl:value-of select="author" /> </td>
<td> <xsl:value-of select="title" /> </td>
<td> <xsl:value-of select="category" /> </td>
<td> <xsl:value-of select="price" /> </td>
</tr>
</xsl:for-each>
```



A complete example given below: Create the following files sort.xml and sort.xsl: **sort.xml**

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="sort.xsl"?>
<booklist>
	<book>
		<author>Mark Grand</author>
		<title>Patterns in Java</title>
		<category>Java</category>
		<price currency="USD">44.99</price>
	</book>
	<book>
		<author>Michael Daconta et al</author>
		<title>XML Development with Java 2</title>
		<category>Java</category>
		<price currency="USD">37.99</price>
	</book>
	<book>
		<author>E. Lynn Harris</author>
		<title>Any Way The Wind Blows</title>
		<category>Fiction-Romance</category>
		<price currency="USD">19.95</price>
	</book>
	<book>
		<author>E. Lynn Harris</author>
		<title>Invisible Life</title>
		<category>Fiction-Romance</category>
		<price currency="USD">16.95</price>
	</book>
	<book>
		<author>Tom Clancy</author>
		<title>Executive Orders</title>
		<category>Fiction-Thriller</category>
		<price currency="USD">7.99</price>
	</book>
	<book>
		<author>Tom Clancy</author>
		<title>The Sum of All Fears</title>
		<category>Fiction-Thriller</category>
```

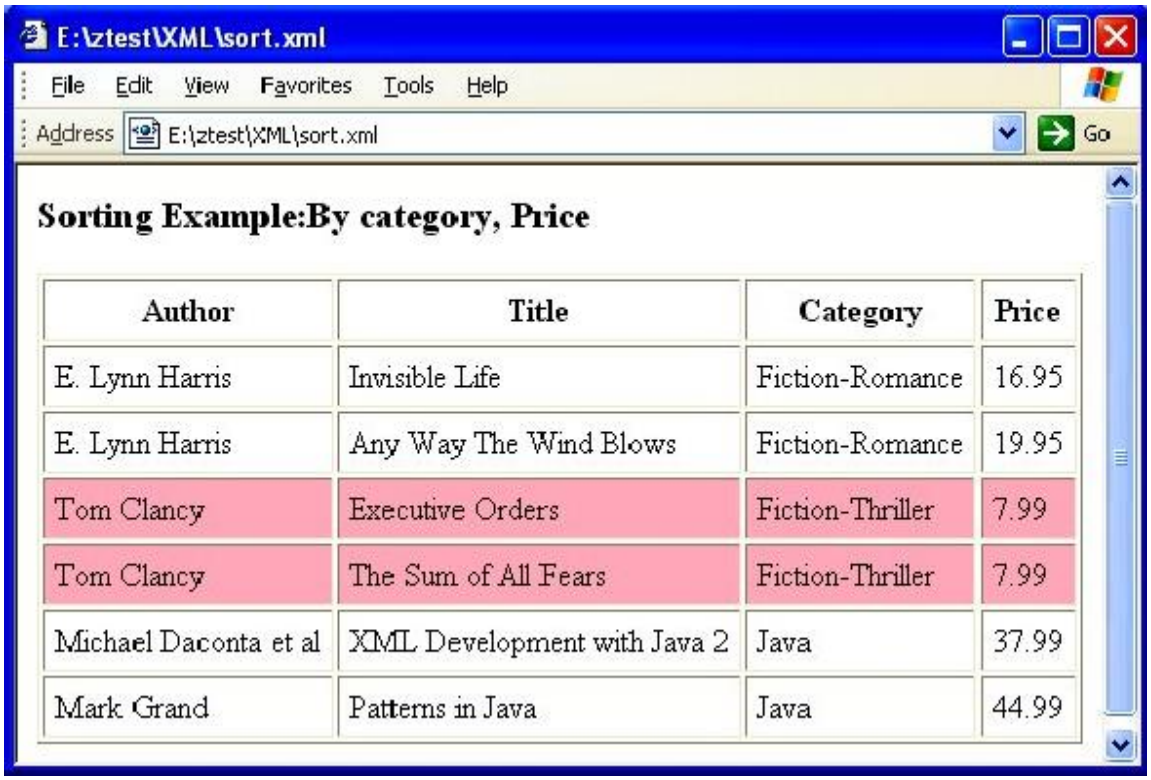
```
            <price currency="USD">7.99</price>
        </book>
</booklist>
```

sort.xsl

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

<xsl:template match="/">

<html>
<body>
<h3>Sorting Example:By category, Price</h3>

<table border="1" cellpadding="5">
<tr>
<th>Author</th> <th>Title</th> <th>Category</th> <th>Price</th>
</tr>
<xsl:for-each select="booklist/book" >
        <xsl:sort select="category" />
        <xsl:sort select="price" order="ascending" data-type="number" />
        <tr>
        </tr><xsl:if test="category='Fiction-Thriller'">
    <xsl:attribute name="bgcolor">pink</xsl:attribute>
</xsl:if>
<td> <xsl:value-of select="author" /> </td>
<td> <xsl:value-of select="title" /> </td>
<td> <xsl:value-of select="category" /> </td>
<td> <xsl:value-of select="price" /> </td>
</xsl:for-each>

</table> </body> </html> </xsl:template> </xsl:stylesheet>
```

Next Open sort.xml in the Internet

Explorer, you will get following output:



You can write the same program for sorting, sorting with multiple column, and conditional formatting

5. XSL FOR BUSINESS-TO-BUSINESS (B2B) COMMUNICATION

XSLT is not only for document publishing, but it can also be used in for B2B communication—the process of exchanging data between two different companies. Developers can leverage XML to describe the data in a vendor-independent fashion. In the ideal case, both companies will agree upon a standard vocabulary for describing the data using a DTD or schema. The vocabulary is composed of the XML element names used in the XML document. However, in certain cases one of the companies might like to use a different vocabulary. This is where XSL enters the picture

The example in this section describes a B2B scenario between a training company, Hot Shot Training, and a software development company, AcmeSoft. The computer training company maintains a database for the students that have attended its courses. The training company has developed an XML application that produces the list of students for a given class. The management team at AcmeSoft would like to retrieve this list from the training company's XML application. However, once the data is retrieved, AcmeSoft would like to store the data in a different XML format using its own XML element names. The application interaction is illustrated in Figure

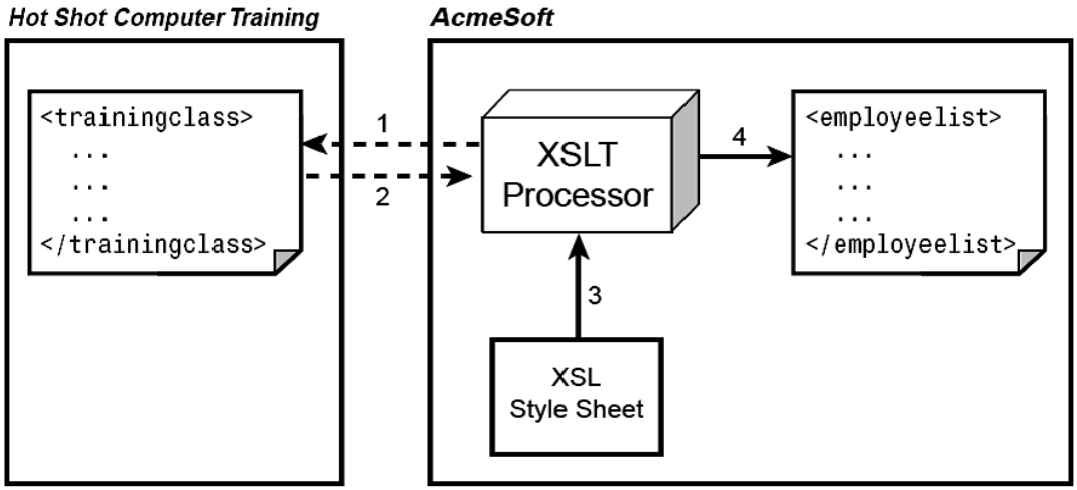


Figure:Converting XML data in B2B communication

Step involved in conversion of xml data in **B2B communication given below:**
Step1:The first step is to request the XML document from the training company
step 2: the XML document is retrieved.
step 3: the document is transformed using the supplied XSLT style sheet
step 4:Finally, the desired output document is produced

A sample output of the XML document used in training company is shown here:
trainingclass.xml

```
<?xml version=”1.0”?>
<?xml-stylesheet type="text/xsl"  href="train2employee.xsl"?>
<trainingclass>
        <title>J2EE Essentials</title>
        <start_date>24 Sep 2001</start_date>
        <end_date>28 Sep 2001</end_date>
        <location>Philadelphia, PA</location>
        <student>
                <first_name>Riley</first_name>
                <last_name>Scott</last_name>
                <email>riley@acmesoft.web</email>
        </student>
        <student>
                <first_name>Torrance</first_name>
                <last_name>Lee</last_name>
                <email>torrance.lee@acmesoft.web</email>
        </student>
</trainingclass>
```

The AcmeSoft wants the above information in the XML file, but with different XML structure. The structure of XML document required by AcmeSoft is shown here:

testoutput.xml

```
<?xml version=”1.0”?>
<employeelist>
        <course_title>J2EE Essentials</course_title>
        <course_date start=”24 Sep 2001” end=”28 Sep 2001” />
        <location>Philadelphia, PA</location>
        <employee>
                <name>
                        <first>Riley</first>
                        <last>Scott</last>
                </name>
                <email>riley.scott@acmesoft.web</email>
        </employee>
        <employee>
                <name>
                        <first>Torrance</first>
                        <last>Lee</last>
                </name>
                <email>torrance.lee@acmesoft.web</email>
                </employee>
</employeelist>
```

Notice in both instances that the data is the same; it’s simply in a different format. The format is different because of the element names used by AcmeSoft. The translation of source XML document into target XML document can be done using the following XSLT file:

train2employee.xsl

```
<?xml version=”1.0”?>
<xsl:stylesheet xmlns:xsl=”http://www.w3.org/1999/XSL/Transform” version=”1.0”>
<xsl:template match=”/trainingclass”>
<employeelist>
<course_title><xsl:value-of select=”title” /></course_title>
<!-- create attributes for the start and end course dates -->
<course_date>
<xsl:attribute name=”start”><xsl:value-of select=”start_date”/></xsl:attribute>
<xsl:attribute name=”end”><xsl:value-of select=”end_date”/></xsl:attribute>
</course_date>
<location><xsl:value-of select=”location” /></location>
<!-- Perform a loop for each student in the training class -->
<xsl:for-each select=”student”>
<employee>
```

```
<name>
        <first><xsl:value-of select="first_name"/></first>
        <last><xsl:value-of select="last_name"/></last>
</name>
<email><xsl:value-of select="email"/></email>
</employee>
</xsl:for-each></employeelist></xsl:template></xsl:stylesheet>
```

Using the XSLT Processor

- For most B2B applications, the source XML document is retrieved by another applications rather than browser
- The another application could be standalone application or a component of a larger B2B application
- In the case of a standalone application, the necessary code to perform the XSLT processing needs to be developed.
- For example, a Visual Basic or Visual C++ application can use the XSLT processor available with the Microsoft XML API
- Java application can use the XSLT processor available with the Apache Xalan API
- Here's the code for a standalone Java application that uses the Apache Xalan API

```
import org.apache.xalan.xslt.*; public class XslTester{
public static void main(String[] args) {
try {
if (args.length != 3) {
System.out.println("Usage: java XslTester <input XML> <input XSL> <output file>");
return;
}
System.out.println("Processing: " + args[0] + " and " + args[1]);
// Step 1: Get a reference to the XSLT Processor
XSLTProcessor myEngine = XSLTProcessorFactory.getProcessor();
// Step 2: Get the XML input document
XSLTInputSource xmlSource = new XSLTInputSource(args[0]);
// Step 3: Get the XSL style sheet
XSLTInputSource xslStylesheet = new XSLTInputSource(args[1]);
// Step 4: Setup the output target
XSLTResultTarget xmlOutput = new XSLTResultTarget(args[2]);
// Step 5: Now process it!
myEngine.process(xmlSource, xslStylesheet, xmlOutput); System.out.println("Created
=> " + args[2]);
System.out.println("Done!");
}
catch (Exception exc) { exc.printStackTrace();
}
}
```

}

Compile the program by typing the following:

>javac XslTester.java

Execute the application by typing the following:

>java XslTester trainingclass.xml train2employee.xsl testoutput.xml

The content of testoutput.xml is given in the previous page.

6. XSL FORMATTING OBJECTS

The XSL technology is also composed of XSL Formatting Objects (XSL-FO). XSL-FO was designed to assist with the printing and displaying of XML data. The main emphasis is on the document layout and structure. This includes the dimensions of the output document, including page headers, footers, and margins. XSL-FO also allows the developer to define the formatting rules for the content, such as font, style, color, and positioning. XSL-FO is a sophisticated version of Cascading Style Sheets (CSS). In fact, XSL-FO borrows a lot of the terminology and elements from CSS

You can use two techniques for creating XSL-FO documents. They are:

(i) develop the XSL-FO file with the manually included data

(ii) dynamically create the XSL-FO file using an XSLT translation

XSL-FO documents are well-formed XML documents. An XSL-FO formatting engine processes XSL-FO documents

XSL-FO Formatting Engines

Many of the XSL-FO formatting engines implement a subset of the XSL-FO specification. The browser support for XSL-FO is nonexistent.

Table contains a list of XSL-FO formatting engines in that we'll use the Apache XSL-FOP to generate PDF documents from XML.

XSL-FO Engine	Web Site
Apache XSL-FOP	xml.apache.org
XEP	www.renderx.com
IText	www.lowagie.com/iText/
Unicorn	www.unicorn-enterprises.com

Basic Document Structure

- An XML-FO document follows the syntax rules of XML; as a result, it is well formed. The following code snippet shows the basic document setup for XSL-FO:

```
<?xml version="1.0" encoding="utf-8"?>
<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
<!-- layout master set -->
<!-- page masters: size and layout -->
<!-- page sequences and content -->
</fo:root>
```

- XSL-FO elements use the following namespace:

http://www.w3.org/1999/XSL/Format

- The element <fo:root> is the root element for the XSL-FO document. An XSL-FO documentcan contain the following components:

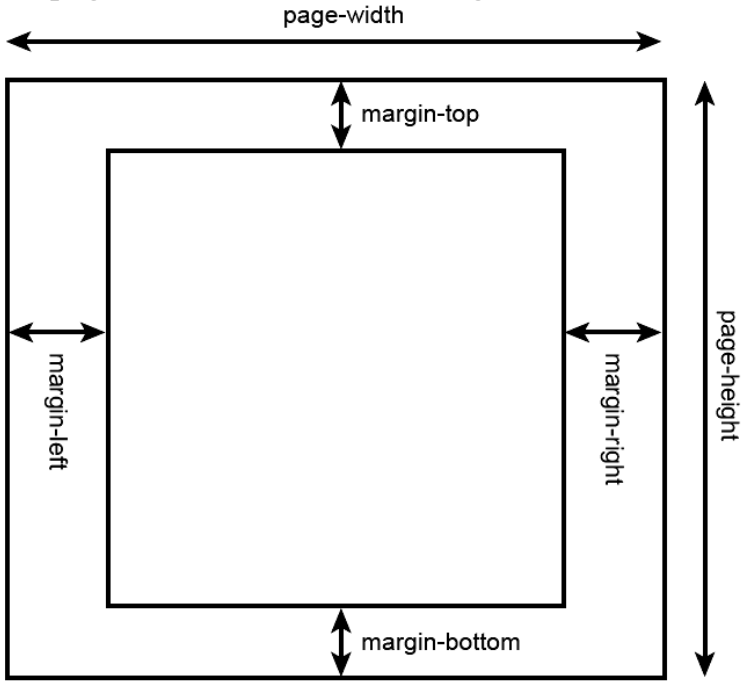
• Page master **• Page master set** **• Page sequences**

Page Master: <fo:page-master>

- The page master describes the page size and layout. For example, we could use an 8.5□11-inch page or an A4 letter. The page master contains the dimensions for a page, including width, height, and margins
- The <fo:simple-page-master> element defines the layout of a page. The following code snippet describes a U.S. letter:

```
<fo:simple-page-master  master-name=”simple”
                        page-height=”11in”   page-width=”8.5in”
                        margin-top=”1in”     margin-bottom=”1in”
                        margin-left=”1.25in”  margin-right=”1.25in”>

</fo:simple-page-master>
```

The components of the page master are shown in Figure:

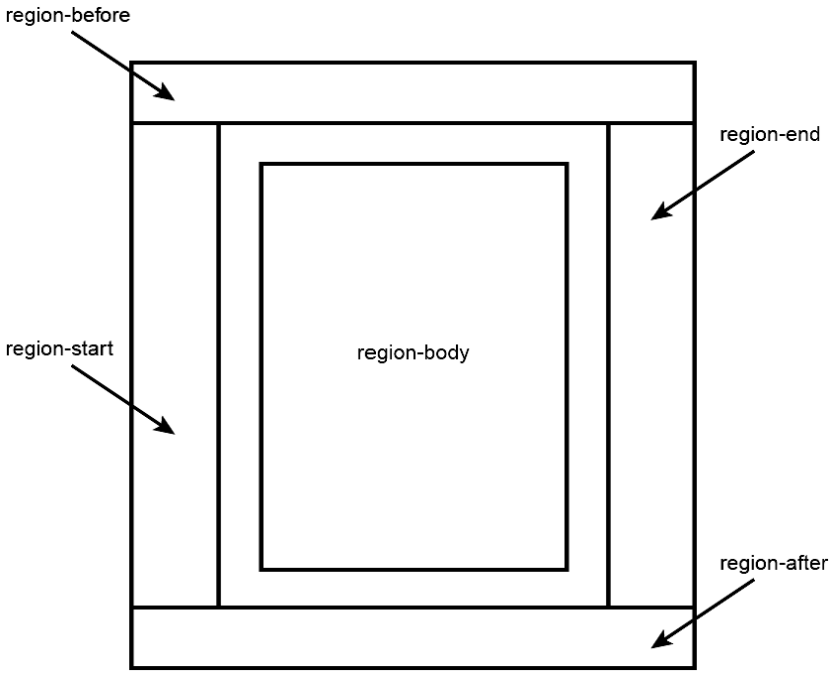


Notice the attributes for <fo:simple-page-master>. The attributes define the height and width of the page, along with the size of the margins. The dimensions in this example are listed in inches (in). the following Table lists the dimensions supported in XSL-FO :

Table : XSL-FO Dimensions

Unit Suffix	Description
in	Inches
mm	Millimeters
cm	Centimeters
pt	Points

pc	Picas
em	Font size of the relevant font
ex	X-height of the relevant font
px	Pixels



- Each **page is divided into five *regions***. Regions serve as containers for the document content. The regions are depicted the in Figure:
- The **region-before** and **region-after** areas are commonly **used for page headers** and **footers**
- The **region-body** area is the center of the page and **contains the main content**.
- The **region-start** and **region-end** sections are commonly used for left and right sidebars, respectively.

- During the definition of a page master, you specify the size of these regions using the following elements:
    - **<fo:region-before>**
    - **<fo:region-after>**
    - **<fo:region-body>**
    - **<fo:region-start>**
    - **<fo:region-end>**

Page Master Set: <fo:page-master-set>

A document can be composed of multiple pages, each with its own dimensions. The page master set refers to the collection of page masters.

Page Sequences: <fo:page-sequence>

A page sequence defines a series of printed pages. Each page sequence refers to a page master for its dimensions. The page sequence contains the actual content for the document. The <fo:page-sequence> element contains:

(i) <fo:static-content> element

(ii) <fo:flow> element

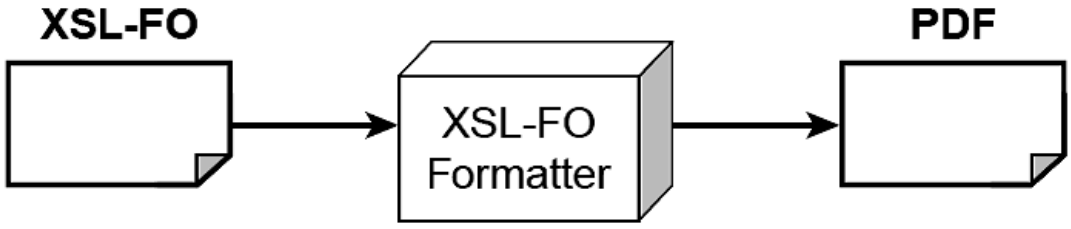
<fo:static-content> element

The <fo:static-content> element is used for page headers and footers. For example, we can define a header for the **company name and page number**, and this information will appear on every page.

<fo:flow> element
The <fo:flow> element contains a collection of text blocks. The <fo:flow> element is similar to a collection of paragraphs. A body of text is defined using the <fo:block> element.

<fo:block> element
The <fo:block> element is a child element of <fo:flow>. The <fo:block> element contains free-flowing text that will wrap to the next line in a document if it overflows

EXAMPLE:
- In the following example, we develop the Xsl-Fo Document with the manually included data. The header information includes book catalog title and page number, footer information includes web site address of the company. Body includes set of paragraphs
- The XSL-FO document will hava ".fo" as extension. Once the XSL-FO document we can generate PDF document by using Apache-FOP compiler.
- The process of conversion is shown in the following figure:



Example: Developing the XSL-FO file with the manually included data

Header_footer.fo

<?xml version="1.0" encoding="utf-8"?>
<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">

<!-- layout master set -->
<fo:layout-master-set>
<!-- page masters: size and layout -->
<fo:simple-page-master master-name="simple" page-height="11in" page-width="8.5in" margin-top="1in" margin-bottom="1in" margin-left="1.25in" margin-right="1.25in">

```
            <fo:region-body margin-top="0.5in"/>
            <fo:region-before extent="3cm"/>
            <fo:region-after extent="1.5cm"/>
        </fo:simple-page-master>
</fo:layout-master-set>

<fo:page-sequence master-name="simple">
<!-- header -->
<fo:static-content flow-name="xsl-region-before">

<fo:block    text-align="end"        font-size="10pt"
             font-family="serif"     line-height="14pt"
                  > Ez Books Catalog - page
<fo:page-number/>
</fo:block>
</fo:static-content>

<!-- footer -->

<fo:static-content flow-name="xsl-region-after">

<fo:block       text-align="center"     font-size="10pt" font-family="serif"
                                        line-height="14pt" >

        Visit our website http://www.ezbooks.web
</fo:block>
</fo:static-content>

<!-- body -->
<fo:flow flow-name="xsl-region-body">
<!-- this defines a level 1 heading with orange background -->
<fo:block    font-size="18pt" font-family="sans-serif" line-height="24pt"
             space-after.optimum="15pt"
             background-color="orange" color="white"
             text-align="center"
             padding-top="3pt">
        Ez Books Online
</fo:block>
<!-- Paragraph that contains info about the company -->
<fo:block    font-size="12pt"  font-family="sans-serif" line-height="15pt" space-
             after.optimum="14pt"          text-align="justify">
Welcome to Ez Books Online, the world's smallest online book store. Our company's
mission is to sell books on Java, Thrillers and Romance. We have something for
everyone...so we think. Feel free to browse our catalog and if you find a book of interest
```

then send us an e-mail.
Thanks for visiting!
</fo:block>

<!-- insert page break for second page -->

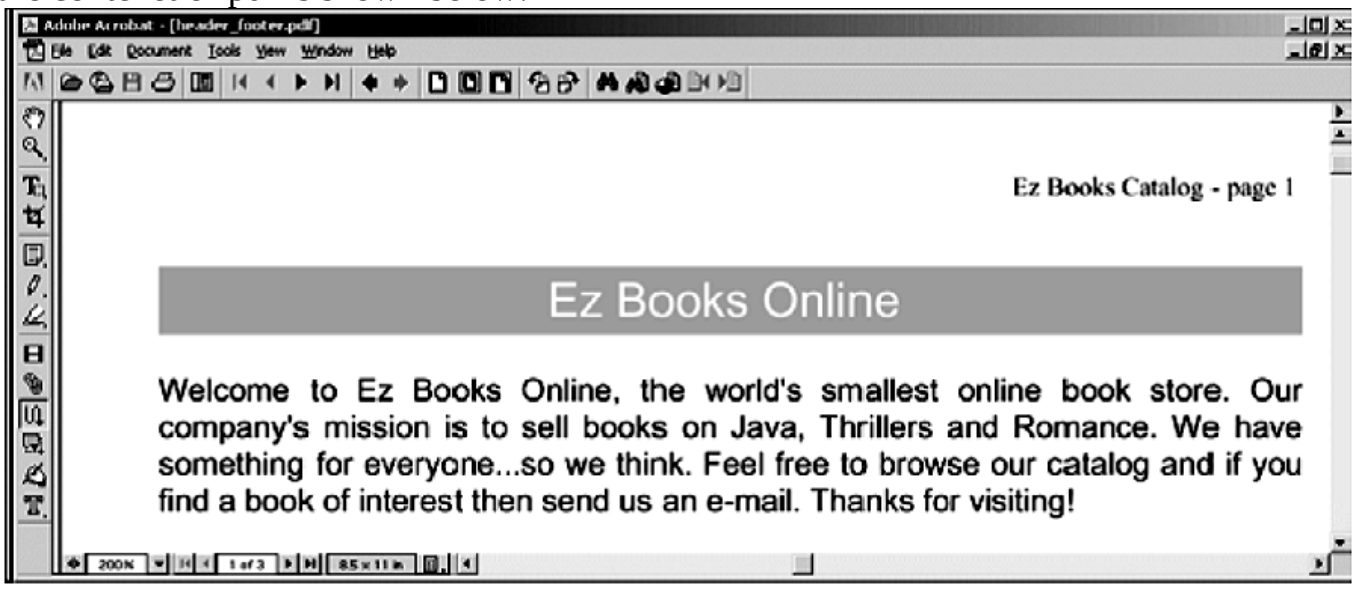
<fo:block break-before=”page”>

A page break is inserted before this block. Notice we have the headers and footers in place. This was accomplished with the fo-static-content elements. We can continue on...business as usual.
</fo:block>
</fo:flow>
</fo:page-sequence>
</fo:root>

Follow these steps to generate a PDF document from header_footer.fo:

1. Open an MS-DOS window.
2. Move to the directory c:\xsl_fo, in which header_footer.fo is stored
3. Set up the Java classpath by typing **setpaths**.
4. Execute Apache-FOP by typing the following command:
   c:\xsl_fo >fop header_footer.fo header_footer.pdf.

the contenet of pdf is shown below:



Graphics

- XSL-FO also allows for the insertion of external graphic images. The graphic formats supported are dependent on the XSL-FO formatting engine. The Apache-FOP formatting engine supports the popular graphics formats: GIF, JPEG, and BMP.
- The following code fragment inserts the image smiley.jpg:
  <fo:block text-align=”center”>

<fo:external-graphic src=”smiley.jpg” width=”200px” height=”200px”/>
</fo:block>

Tables

- XSL-FO has rich support for structuring tabular data. There are many similarities
  - between HTML tables and XSL-FO tables. *The following* Table lists the HTML table elements with their corresponding XSL-FO table elements:

HTML Element	XSL-FO Element
TABLE	fo:table-and-caption (or) fo:table
CAPTION	fo:table-caption
COL	fo:table-column
COLGROUP	Not applicable
TBODY	fo:table-body
TFOOT	fo:table-footer
TR	fo:table-row
TH	fo:table-header
TD	fo:table-cell

  - The following code fragment defines the basic structure of the table:
  - <fo:table>
  - <!-- define column widths -->
  - <fo:table-column column-width=”120pt”/>
  - <fo:table-column column-width=”200pt”/>
  - <fo:table-column column-width=”80pt”/>
  - <fo:table-header>
  - <fo:table-row>
  - <fo:table-cell><fo:block font-weight=”bold”>Author</fo:block></fo:table-cell>
  - <fo:table-cell><fo:block font-weight=”bold”>Title</fo:block> </fo:table-cell>
  - <fo:table-cell><fo:block font-weight=”bold”>Price (USD)</fo:block></fo:table-cell>
  - </fo:table-row>
  - </fo:table-header>
  - <!-- insert table body and rows here -->
  - **<fo:table-body>**
  - <fo:table-row>
  - <fo:table-cell><fo:block>Michael Daconta</fo:block></fo:table-cell>
  - <fo:table-cell><fo:block>XML Development with Java 2</fo:block></fo:table-cell>
  - <fo:table-cell><fo:block>37.99</fo:block></fo:table-cell>
  - </fo:table-row>
  - <fo:table-row>
  - <fo:table-cell><fo:block>E. Lynn Harris</fo:block></fo:table-cell>
  - <fo:table-cell><fo:block>Any Way The Wind Blows</fo:block></fo:table-cell>

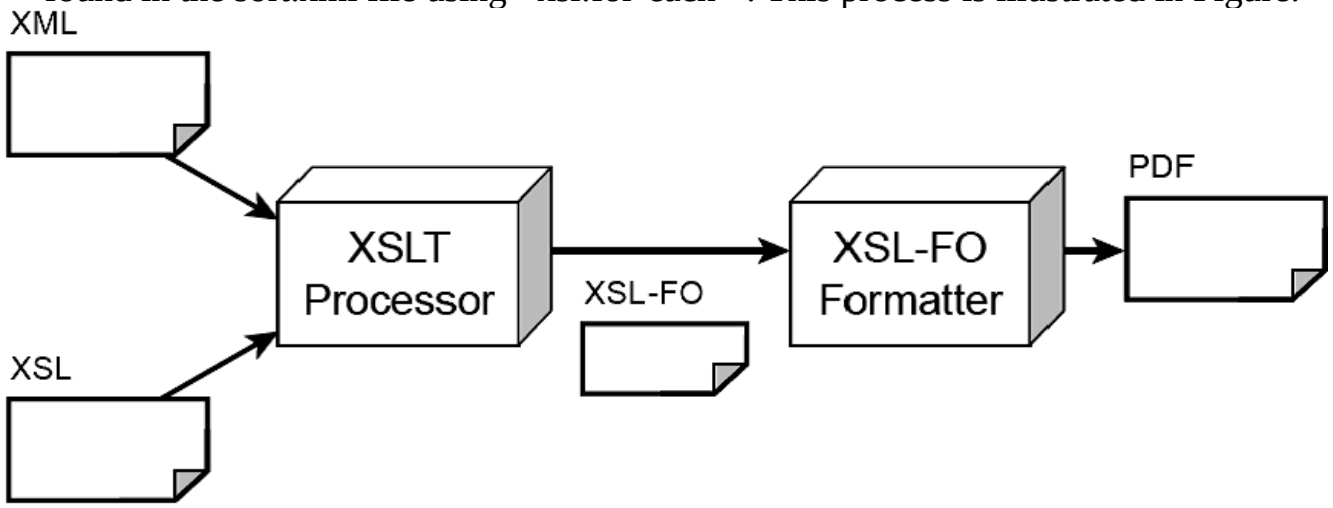
- <fo:table-cell><fo:block>19.95</fo:block></fo:table-cell>
- </fo:table-row>
- </fo:table-body>
- </fo:table>
- <fo:table-column> - element - it is used to specify column width

The output for the above code in the PDF document is shown below:

Author	Title	Price (USD)
Michael Daconta	XML Development with Java 2	37.99
E. Lynn Harris	Any Way The Wind Blows	19.95

7. GENERATING XSL-FO TABLES USING XSLT (OR) DYNAMICALLY CREATE THE XSL-FO FILE USING AN XSLT TRANSLATION

- In this section, we'll use XSLT to automatically generate the XSL-FO document. The file sort.xml contains a list of the books.
- We can develop an XSL style sheet(**booklist_table.xsl**) that will automatically construct the XSL-FO document by dynamically contructing tables row for each book found in the sort.xml file using <xsl:for-each> . This process is illustrated in Figure:



The content of booklist_table.xsl shown below:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:fo="http://www.w3.org/1999/XSL/Format"  version="1.0">
<xsl:template match="/">
<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
<fo:layout-master-set>
<!-- layout information -->
<fo:simple-page-master master-name="simple" page-height="11in"
                       page-width="8.5in"     margin-top="1in"
                       margin-bottom="2in"    margin-
```

```
                    left=”1.25in” margin-right=”1.25in”>
        <fo:region-body margin-top=”0.5in”/>
        <fo:region-before extent=”3cm”/>
        <fo:region-after extent=”1.5cm”/>
</fo:simple-page-master>
</fo:layout-master-set>
<!-- end: defines page layout -->
<fo:page-sequence master-name=”simple”>
<fo:flow flow-name=”xsl-region-body”>
<!-- table start -->
<fo:table border-style=”solid” border-width=”.1mm” >
<!-- define column widths -->
<fo:table-column column-width=”120pt”/>
<fo:table-column column-width=”200pt”/>

<fo:table-column column-width=”80pt”/>

<fo:table-header>

        <fo:table-row >
        <fo:table-cell border-style=”solid”  border-width=”.1mm”>
                <fo:block>Author</fo:block>
        </fo:table-cell>
        <fo:table-cell border-style=”solid” border-width=”.1mm”>
                <fo:block >Title</fo:block>
        </fo:table-cell>
        <fo:table-cell border-style=”solid” border-width=”.1mm”>
                <fo:block>Price (USD)</fo:block>
        </fo:table-cell>
</fo:table-row>
</fo:table-header>
<fo:table-body>
<xsl:for-each select=”booklist/book” >
<fo:table-row>
        <fo:table-cell border-style=”solid” border-width=”.1mm”>
                <fo:block><xsl:value-of select=”author” /></fo:block>
        </fo:table-cell>
        <fo:table-cell border-style=”solid” border-width=”.1mm”>
                <fo:block><xsl:value-of select=”title” /></fo:block>
        </fo:table-cell>
        <fo:table-cell border-style=”solid” border-width=”.1mm”>
                <fo:block><xsl:value-of select=”price” /></fo:block>
        </fo:table-cell>
```

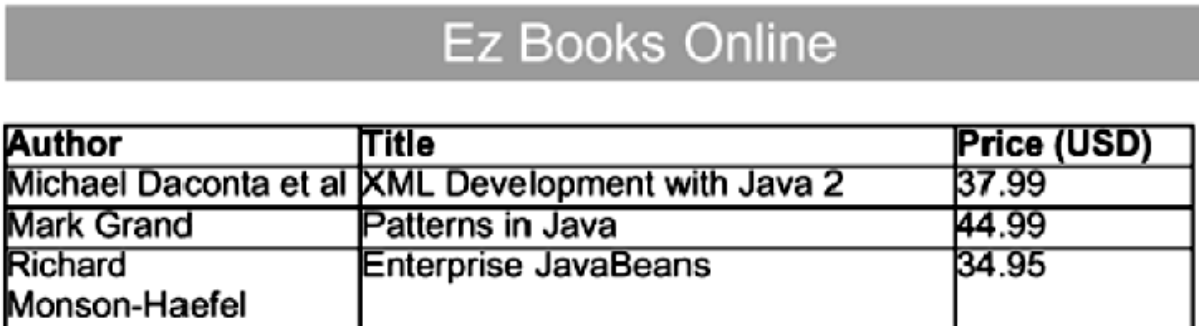
```
</fo:table-row>
</xsl:for-each>
</fo:table-body>
</fo:table>
<!-- table end -->
</fo:flow>
</fo:page-sequence>
</fo:root>
</xsl:template>
</xsl:stylesheet>
```

Generating a PDF Document

1. Open an MS-DOS window.
2. Move to the directory c:\xsl_fo, in which sort.xml and booklist_table.xsl are stored
3. Set up the Java classpath by typing **setpaths**.
4. Execute Apache-FOP by typing the following command:

c:\xsl_fo >fop -xml booklist.xml -xsl booklist_table.xsl dyntable.pdf

The output of dyntable.pdf is shown below:



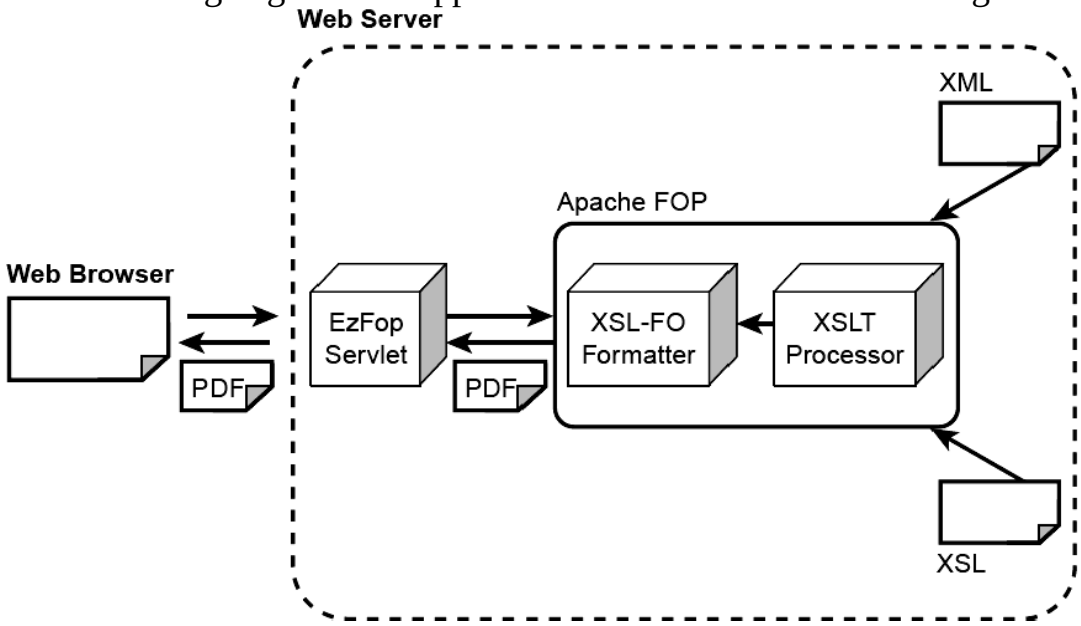
Ez Books Online

Author	Title	Price (USD)
Michael Daconta et al	XML Development with Java 2	37.99
Mark Grand	Patterns in Java	44.99
Richard Monson-Haefel	Enterprise JavaBeans	34.95

8. WEB APPLICATION INTEGRATION: JAVA SERVLETS, XSLT, AND XSL-FO

In this section, develop a Web application that integrates Java servlets, XSLT, and XSL-FO. We will develop a Java servlet to pass an XML document and XSL style sheet to the Apache-FOP formatting engine. The XML document is sort.xml. The XSL style sheet, booklist_table.xsl, contains the XSL-FO, contains the XSL-FO template code to generate a table. The servlet will respond with the PDF document generated by the

Apache-FOP formatting engine. The application interaction is shown in Figure



DEVELOPING THE JAVA SERVLET

The Java servlet handles an HTTP GET request. The servlet sets up a reference to the files booklist.xml and booklist_table.xsl. The Apache-FOP API provides access to the Apache-FOP formatting engine via the class org.apache.fop.apps.Driver. The following code creates an instance of the driver and sets the renderer to PDF:

```
// setup the driver for PDF
Driver driver = new Driver()
driver.setRenderer(Driver.RENDER_PDF);
```

Next, the servlet creates a file reference for the XML document and XSL style sheet. Because the servlet is running in the context of a servlet engine, we need to retrieve the real path to the Web application's root. The XSLTInputHandler class transforms the XML document using the XSL style sheet, and the resulting document is input for the Apache-FOP processing engine

The complete program is given below:

```
public class EzFopServlet extends HttpServlet {
public void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
try {
        // get the application root for this web app
        String xmlFileName = getServletContext().getRealPath("/") +"booklist.xml";
        String xslFileName = getServletContext().getRealPath("/")
        +"booklist_table.xsl"; File xmlFile = new File(xmlFileName);
        File xslFile = new File(xslFileName);


        // setup the driver for PDF
```

```
        Driver driver = new Driver();
        driver.setRenderer(Driver.RENDER_P
        DF);
        // create an input handler for the XSLT transformation
        XSLTInputHandler inputHandler =new XSLTInputHandler(xmlFile,
        xslFile); XMLReader parser = inputHandler.getParser();
        // setup the output for XSL-FO formatter process
        // temporarily place in a ByteArrayOutputStream
        ByteArrayOutputStream out = new
        ByteArrayOutputStream(); driver.setOutputStream(out);

        // Run the formatter based on the XSL-FO document
        driver.render(parser, inputHandler.getInputSource());
        // The out object has the result of the XSL-FO formatter process
        // Retrieve the content from ByteArray
        byte[] content = out.toByteArray();
        // Setup the response for the web browser
        response.setContentType("application/pdf")
        ;
        response.setContentLength(content.length);

        // Finally, send the result to the browser!
        OutputStream outputToBrowser =
        response.getOutputStream();
        outputToBrowser.write(content);
        outputToBrowser.flush();
    }catch (Exception exc){ }
    }
}
```

After deploying the servlet program in the Tomcat server, access the EzFopServlet Web application using http://localhost:8080/ EzFopServlet

CHAPTER 9

MODELING DATABASES IN XML

1. INTEGRATING XML WITH DATABASES

XML and database integration is important because XML provides a standard technique to describe data. By leveraging XML, a company can convert its existing corporate data into a format that is consumable by its trading partners. XML allows the development team to define a set of custom tags specific to its industry. A trading partner can import the XML data into its system using the given format. The trading partner also has the option of converting the data to a different XML format using XSLT

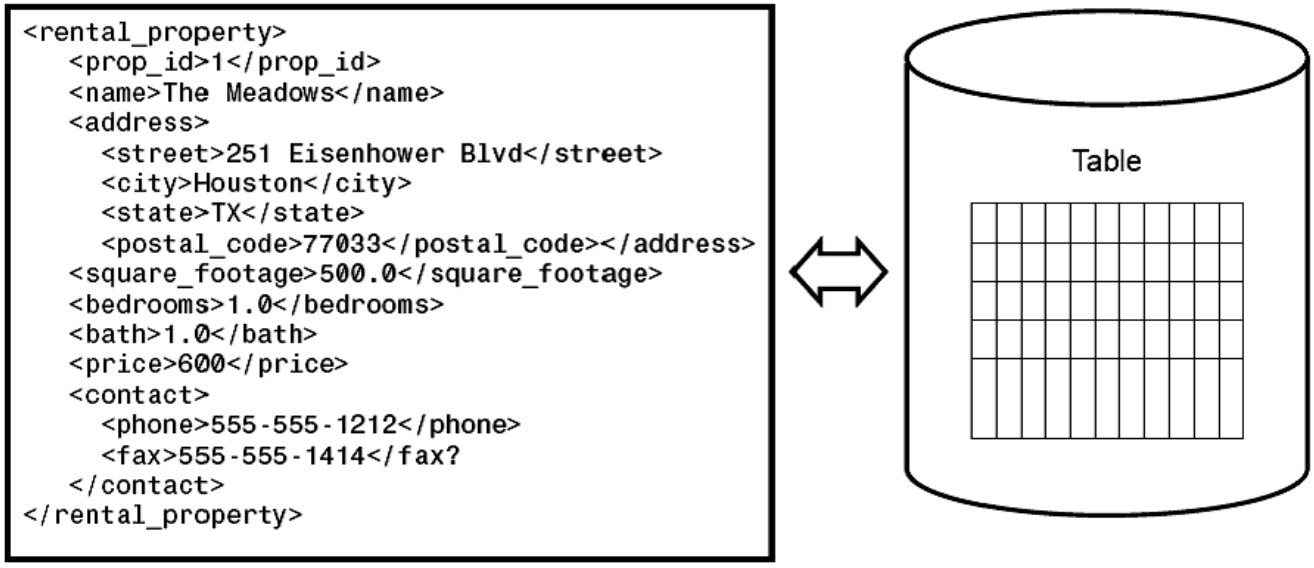
XML Database Solutions

XML database solutions generally come in two flavors:

- **XML database mapping**
- **Native XML support**

2. XML DATABASE MAPPING

It provides a mapping between the XML document and the database fields. The system dynamically converts SQL result sets to XML documents. Depending on the sophistication of the product, it may provide a graphical tool to map the database fields to the desired XML elements. Other tools support a configuration file that defines the mapping. These tools continue to store the information in relational database management system (RDBMS) format. They simply provide an XML conversion process that is normally implemented as a server-side Web application. This solution is depicted in the following **Figure**: **Mapping XML documents to database fields**



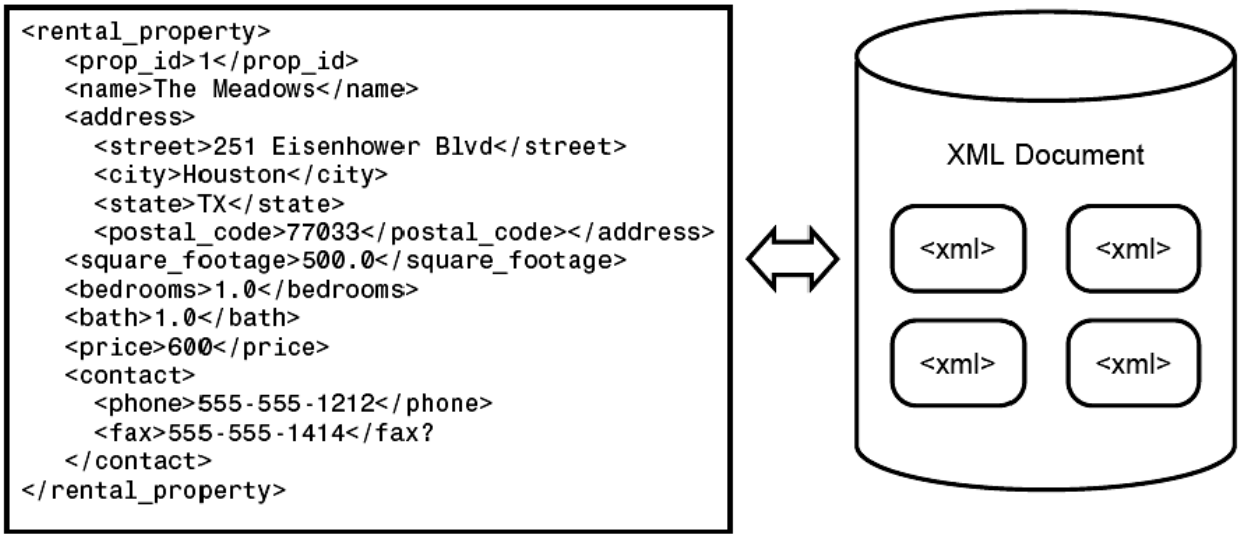
XML document **Relational Database**

The products that provide XML database mappings is listed below:

Product	*Company*
DB2 Extender	IBM
SQL Server 2000	Microsoft
Oracle 8i & 9i	Oracle
DataMirror DB/XML	DataMirror
webMethods	webMethods
Excelon	Excelon

3. NATIVE XML SUPPORT

The second type of XML database solution actually stores the XML data in the document in its native format. Each product uses its own proprietary serialization technique to store the data. However, when the data is retrieved, it represents an XML document. This solution is depicted in the following Figure: Native XML databases



XML Document **Native XML Database**

Products that provide native XML document support **are listed below:**

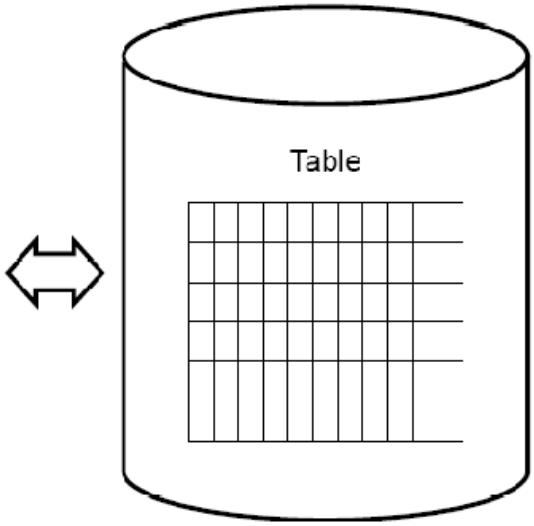
Product	*Company*
TEXTML	IXIA Soft
Oracle 8i and 9i	Oracle
Excelon	Excelon
dbXML	dbXML Group
Tamino	Software AG

4. MODELING DATABASES IN XML

In this section, you'll learn how to model a database in XML using Java. When we model a database, we provide an external representation of the database contents For our sample

program, we'll utilize a database that contains information on properties. We'll model the rental property database as an XML document.

```
<rental_property>
   <prop_id>1</prop_id>
   <name>The Meadows</name>
   <address>
     <street>251 Eisenhower Blvd</street>
     <city>Houston</city>
     <state>TX</state>
     <postal_code>77033</postal_code></address>
   <square_footage>500.0</square_footage>
   <bedrooms>1.0</bedrooms>
   <bath>1.0</bath>
   <price>600</price>
   <contact>
     <phone>555-555-1212</phone>
     <fax>555-555-1414</fax?
   </contact>
</rental_property>
```



XML document **Relational Database**

- One possible solution is to use Java servlets and JDBC. Java servlets are server-side components that reside in a Web server or application server. Java servlets are commonly used to handle requests from Web browsers using the HTTP protocol.
- A key advantage to using servlets is the thin-client interface. The servlets handle the request on the server side and respond by generating an HTML page dynamically The browser only has to provide support of HTML. As a result, there is zero client-side administration
- We can develop a servlet that uses JDBC. The servlet will make the appropriate query to the database and use Java Database Connectivity (JDBC) API result set metadata to create the elements
- In this section we will learn XML data binding features of Java Architecture for XML Binding (JAXB). JAXB provides a framework for representing XML documents as Java objects. Using the JAXB framework, we can guarantee that the documents processed by our system are well formed. Also, we have the option of validating the XML data against a schema.
- In the JAXB framework, we can parse XML documents into a suitable Java object. This technique is referred to as *unmarshaling*. The JAXB framework also provides the capability to generate XML documents from Java objects, which is referred to as *marshaling*. The process is illustrated in the Figure: *JAXB marshaling and unmarshaling*

JAXB is easier to use and a more efficient technique for processing XML documents than the SAX or DOM API. Using the SAX API, you have to create a custom content handler for each XML document structure.

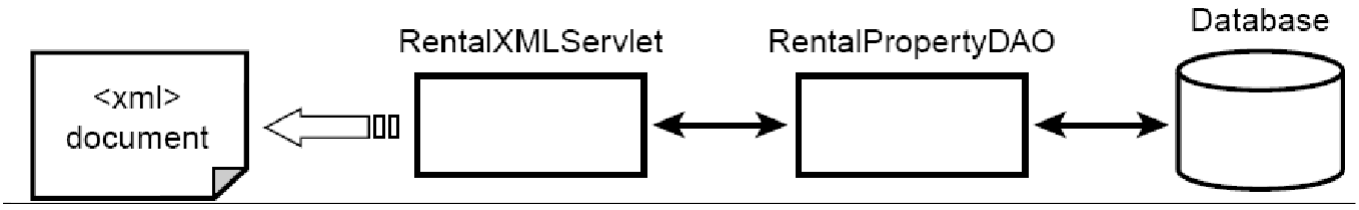
Using JAXB, an application can parse an XML document by simply unmarshaling the data from an input stream. JAXB is similar to DOM in that we can create XML documents programmatically and perform validation. However, the hindrance with DOM is the complex API. If we have an XML tree, using the DOM API, we have to traverse through the tree to retrieve elements. However, with JAXB, we retrieve the data from the XML document by simply calling a method on an object. JAXB allows us to define Java objects that map to XML documents, so we can easily retrieve data. The JAXB framework also ensures the type safety of the data

5. JAXB SOLUTION

In the JAXB solution, we will model the rental property database as an XML document. This process involves the following steps:

1. Review the database schema.
2. Construct the desired XML document.
3. Define a schema for the XML document.
4. Create the JAXB binding schema.
5. Generate the JAXB classes based on the schema.
6. Develop a Data Access Object (DAO).
7. Develop a servlet for HTTP access.

Figure: *"The rental property application architecture"* illustrates the application architecture. RentalXMLServlet communicates with RentalDAO to retrieve information from the database. Once the information is retrieved by RentalDAO, RentalXMLServlet generates an XML document



REVIEWING THE DATABASE SCHEMA

We have an existing database for the rental properties Table 10.3 contains the database schema.

TABLE 10.3 Rental Property Database Schema

FIELD	TYPE
prop_num	NUMBER
Name	VARCHAR2
street_address	VARCHAR2
City	VARCHAR2
State	VARCHAR2
zip_code	VARCHAR2
size_sq	NUMBER
bed_count	NUMBER
bath_count	NUMBER
monthly_rent	NUMBER
voice_phone	VARCHAR2
fax_phone	VARCHAR2

CONSTRUCTING THE DESIRED XML DOCUMENT

The desired output XML document describes the rental property. However, the XML document does not use the exact field names listed in the database schema. Instead, the XML document provides a custom mapping of the database fields to XML element names. Table 10.4 contains the mapping.

XML Database Mapping

DatabaseField	*XMLElementName*
prop_num	<prop_id>
Name	<name>
street_address	<street>
City	<city>
State	<state>
zip_code	<postal_code>
size_sq	<square_footage>
bed_count	<bedrooms>
bath_count	<bath>
monthly_rent	<price>
voice_phone	<phone>
fax_phone	<fax>

A rental property is described with a root element of <rental_property>, as shown in the following code:

```
<rental_property>
        <prop_id>1</prop_id>
        <name>The Meadows</name>
        <address>
                <street>251 Eisenhower Blvd</street>
                <city>Houston</city>
```

```
            <state>TX</state>
            <postal_code>77033</postal_code>
        </address>
        <square_footage>500.0</square_footage>
        <bedrooms>1.0</bedrooms>
        <bath>1.0</bath>
        <price>600</price>
        <contact>
            <phone>555-555-1212</phone>
            <fax>555-555-1414</fax>
        </contact>
</rental_property>
```

Notice how the <address> element contains the subelements <street>, <city>, <state>, and <postal_code>. A similar approach is taken for the contact information. The <contact> element contains the <phone> and <fax> elements for the voice number and fax number, respectively

In our system, we'll normally work with a collection of rental properties. This collection is modeled using a <rental_property_list> element, as shown here:

```
<rental_property_list>
        <rental_property> … </rental_property>
        <rental_property> … </rental_property>
        … …
</rental_property_list>
```

DEFINING A SCHEMA FOR THE XML DOCUMENT

We will define the Document Type Definition (DTD). The DTD schema format was chosen because JAXB 1.0 (early access) only supports DTDs. The following Listing contains the DTD for our rental property list.

rental_property.dtd

```
<!ELEMENT rental_property_list (rental_property)*>

<!ELEMENT rental_property (prop_id, name, address, square_footage,
                                                    bedrooms, bath, price, contact)>

<!ELEMENT prop_id (#PCDATA)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT address (street, city, state, postal_code)>
<!ELEMENT street (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT state (#PCDATA)>
```

<!ELEMENT postal_code (#PCDATA)>
<!ELEMENT square_footage (#PCDATA)>
<!ELEMENT bedrooms (#PCDATA)>
<!ELEMENT bath (#PCDATA)>
<!ELEMENT price (#PCDATA)>
<!ELEMENT contact (phone, fax)>

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

CREATING THE JAXB BINDING SCHEMA

The JAXB binding schema is an XML document that contains instructions on how to bind a DTD to a Java class.
Using the JAXB binding schema, we can define the names of the generated Java classes, map element names to specific properties in the Java class, and provide the mapping rules for attributes
The following code example informs the JAXB system that the element <rental_property_list> should be mapped to a Java class and that it is the root element for the XML document:

```
<element name=”rental_property_list” type=”class” root=”true”/>
```

The binding schema also allows us to define a conversion rule for elements. For example, the numerical data for the rental property, such as price, square footage, and number of rooms, is always represented in the DTD as text data (#PCDATA). This is one of the limitations of the DTD format. However, by using JAXB, we can specify that a given element.
The following code defines the package name as xml.jaxb:

```
<options package=”xml.jaxb”/>
```

See the JAXB specification for details on the binding schema file format. Now, let’s look at the JAXB binding schema file for our rental property example. The schema files normally use the filename extension .xjs (for XML Java schema). The following Listing contains the complete code for our JAXB binding schema, rental_property.xjs.
rental_property.xjs

```
<?xml version=”1.0” encoding=”ISO-8859-1” ?>
<!DOCTYPE xml-java-binding-schema
                SYSTEM  ”http://java.sun.com/dtd/jaxb/1.0-ea/xjs.dtd”>
<xml-java-binding-schema version=”1.0-ea”>
<options package=”xmlunleashed.ch10.jaxb”/>
<element name=”rental_property_list” type=”class” root=”true”>
<content property=”list”/>
</element>
```

<element name=”square_footage” type=”value” convert=”double”/>
<element name=”bedrooms” type=”value” convert=”double”/>
<element name=”bath” type=”value” convert=”double”/>
<element name=”price” type=”value” convert=”BigDecimal”/>
<conversion name=”BigDecimal” type=”java.math.BigDecimal”/>
</xml-java-binding-schema>

6. GENERATING THE JAXB CLASSES BASED ON SCHEMAS

Now we are ready to generate the Java source files based on our schemas. JAXB provides a schema compiler for generating the Java source files. The schema compiler takes as input the DTD and the JAXB binding schema. The following Figure illustrates the process.

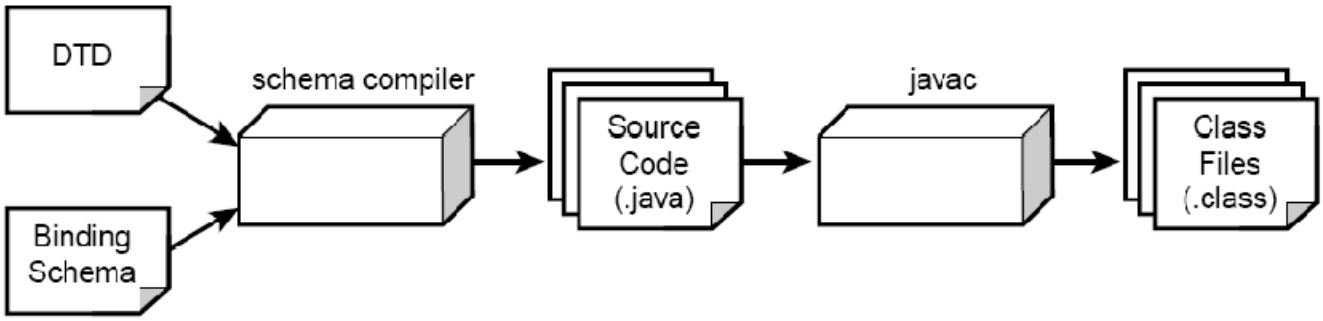


Figure:Generating Java classes with the JAXB compiler

Type everything on one line:
>java com.sun.tools.xjc.Main rental_property.dtd rental_property.xjs -d source_code

This command generates source code in the source_code directory. The following files are generated:
• RentalPropertyList.java. This file models the <rental_property_list> element.
• RentalProperty.java. This file models the <rental_property> element.
• Address.java. This file models the <address> subelement.
• Contact.java. This file models the <contact> subelement

The partial source code for **“RentalProperty.java”** is given below:

```
public class RentalProperty extends MarshallableObject implements Element
{
        private String _PropId;
        private String _Name;
        private Address _Address;
        private double _SquareFootage;
        private boolean has_SquareFootage = false;
        private double _Bedrooms;
        private boolean has_Bedrooms = false;
```

```
        private double _Bath;
        private boolean has_Bath = false;
        private BigDecimal _Price;
        private Contact _Contact;
        public String getPropId() { return _PropId; }
        public void setPropId(String _PropId) { this._PropId = _PropId; }
        }

        public void marshal(Marshaller m) throws IOException
        {
                // code to output the XML document
        }
        public void unmarshal(Unmarshaller u) throws UnmarshalException
        {
        // code to read in the XML document
        }
        … …
}
```

7. DEVELOPING A DATA ACCESS OBJECT (DAO)

A Data Access Object (DAO) provides access to the backend database. The goal of the DAO design pattern is to provide a higher level of abstraction for database access. The DAO encapsulates the complex JDBC and SQL calls. The DAO provides access to the backend database via public methods. The DAO converts a result set to a collection of objects. The objects model the data stored in the database. The application interaction with a DAO is shown in the following Figure:

Figure:Data Access Object design pattern

Let's examine the components of the RentalPropertyDAO source code

```
public class RentalPropertyDAO
{
protected Connection myConn;
public RentalPropertyDAO()
{
```

```
    try {
Class.forName(“sun.jdbc.odbc.JdbOdbcDriver”); String
conStr=”jdbc:odbc:RentalPropertyDSN”;
myConn = DriverManager.getConnection (conStr, ”test”, ”test”);
        }catch (Exception exc) {  }
  }
  public RentalPropertyList getRentalProperties()
  {
     RentalPropertyList theRentalPropertyList = new
     RentalPropertyList(); java.util.List theList =
     theRentalPropertyList.getList();
    try {
         Statement myStmt = myConn.createStatement();
         ResultSet myRs = myStmt.executeQuery(“SELECT *FROM
         rental_properties”); RentalProperty tempProperty = null;
         // build a collection of JAXB RentalProperty
         objects while (myRs.next()) {
               tempProperty = createRentalProperty(myRs);
               theList.add(tempProperty);
         }
        // be sure to validate the new list
        theRentalPropertyList.validate();
        myRs.close();
       myStmt.close();
    }catch (Exception exc) { }
   return theRentalPropertyList;
  }
  protected RentalProperty createRentalProperty(ResultSet rs)
  {
     RentalProperty theProperty = new RentalProperty();
     Address theAddress = new Address();
     Contact theContact = new Contact();
     try {
          // set the rental property number and name
         theProperty.setPropId(rs.getString(“prop_num”));
         theProperty.setName(rs.getString(“name”));

         // set the address
         theAddress.setStreet(rs.getString(“street_address”));
         theAddress.setCity(rs.getString(“city”));
         theAddress.setState(rs.getString(“state”));
         theAddress.setPostalCode(rs.getString(“zip_code”));
         theProperty.setAddress(theAddress);
```

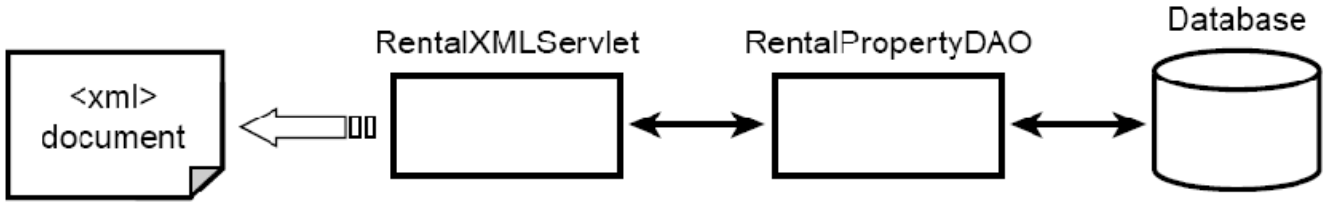
```
        // set the square footage, bedrooms, bath count and rent
        theProperty.setSquareFootage(rs.getDouble("size_sq"));
        theProperty.setBedrooms(rs.getDouble("bed_count"));
        theProperty.setBath(rs.getDouble("bath_count"));
        theProperty.setPrice(new BigDecimal(rs.getDouble("monthly_rent")));
        // set the contact information
        theContact.setPhone(rs.getString("voice_phone"));
        theContact.setFax(rs.getString("fax_phone"));
        theProperty.setContact(theContact);
    }catch (SQLException exc) {

    } return theProperty;
  }
}
```

Now that we have the DAO in place, a client program can easily retrieve information from the database. The RentalPropertyList collection contains JAXB RentalProperty objects. These objects are capable of producing an XML representation of their data The XML data is available by calling the marshal() method.

8. DEVELOPING A SERVLET FOR HTTP ACCESS

At this point, we have constructed the RentalPropertyDAO Data Access Object. This DAO is capable of retrieving information from a database and providing a collection of objects. Thanks to the JAXB framework, these objects can be marshaled into XML. Now we need to provide an HTTP interface for RentalPropertyDAO so that a Web browser can interact with our system. Java servlets provides support for the HTTP protocol.

we'll use a servlet to handle the requests to the DAO. In the servlet, we'll call the appropriate method and return the result as an XML document. The following Figure depicts the application interaction



RentalXMLServlet.java

public class RentalXMLServlet extends HttpServlet

{

public void doGet(HttpServletRequest request,HttpServletResponse response) IOException

```
{
  ServletOutputStream out = null;
  RentalPropertyList  theList  =
  null; try {
        response.setContentType("text/xml");
        // Retrieve the servlet output stream
        out = response.getOutputStream();
        // Retrieve a list of rental properties
        theList = myRentalDAO.getRentalProperties();
        // Marshal the list as an XML document

        theList.marshal(out);
    }catch (Exception e ) {

    } finally {
                out.close();
    }
  }
}
```

Testing the Application

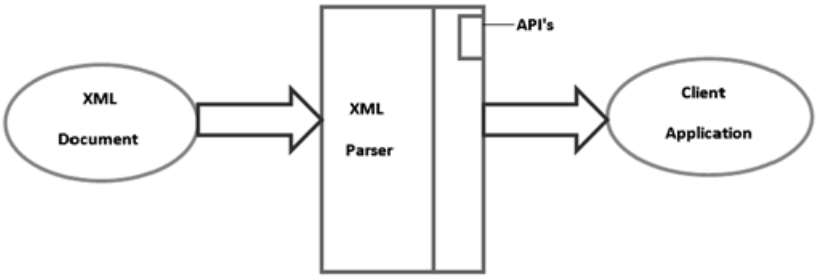
- Once Tomcat 4 is installed, we need to add a new Web application that points to the source code directory
- Now we need to test RentalXMLServlet. In a Web browser, open
  http://localhost:8080/servlet/RentalXMLServlet

CHAPTER 10

XML PARSER

1. WHAT ARE XML PARSERS?

- An XML parser is a software library or package that provides interfaces for client applications to work with an XML document
- XML Parser provides way how to access or modify data present in an XML document from programming languages
- 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:



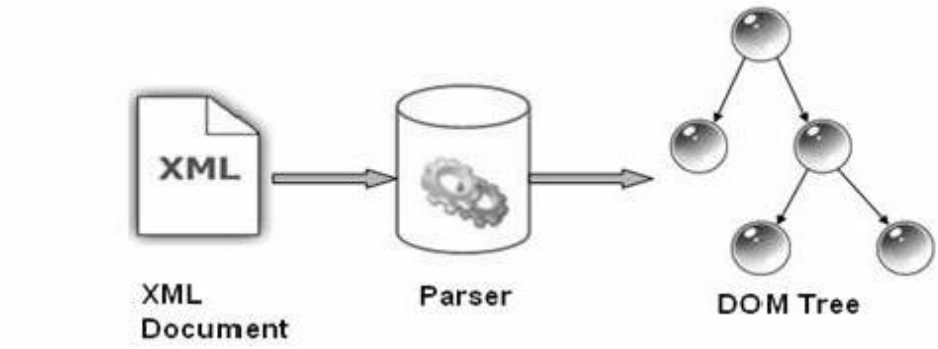
Types of XML Parsers

These are the two main types of XML Parsers:

1. DOM(Document Object Model)
2. SAX (Simple API for XML)

2. PARSING XML USING DOCUMENT OBJECT MODEL

- The Document Object Model (DOM) provides a way of representing an XML document in main memory as tree structure, so that it can be manipulated by your software.
- DOM is a standard application programming interface (API) that makes it easy for programmers to access elements and delete, add, or edit content and attributes.



- DOM was proposed by the World Wide Web Consortium (W3C) in August of 1997

- DOM by itself is just a specification for a set of interfaces defined by W3C
- The DOM interfaces are defined independent of any particular programming language. You can write DOM code in just about any programming language, such as Java, ECMAScript (a standardized version of JavaScript/JScript), or C++
- There are DOM APIs for each of these languages. W3C uses the Object Management Group's (OMG) Interface Definition Language (IDL) to define DOM in a language-neutral way. Language-specific *bindings*, or DOM interfaces, exist for these languages. The DOM specification itself includes bindings for Java and ECMAScript

What DOM Is Not?

- DOM is not a mechanism for *storing an* object as XML documents. DOM is an object model for representing XML documents in your code.
- DOM is not a set of data documents. structures; rather it is an object model describing XML
- DOM does not specify what information in a document is relevant or how information should be structured.
- DOM has nothing to do with COM, CORBA, or other technologies that include the words *object model*.

Why Do I Need DOM?

- DOM is to create or modify an XML document programmatically.
- You can use DOM just to read an XML document, but, SAX is often a better

is often a better

candidate for the read-only case

- If you want to create a document, you start by creating a root element and then add attributes, content, sub-elements, and so on. Once you are finished, you can write the document out to disk or send it over a network
- If you want to modify an existing XML document, you can read it in from a file or other I/O source. The entire document is read into memory all at once, so you can change any part of it at any time. The representation in memory is a tree structure that starts with a root element that contains attributes, content, and sub-elements. You can traverse this tree, search for a specific node, and change its attributes or data. You can also add attributes or elements anywhere in the tree, as long as you don't violate the

rules of a wellformed document. Again, you can write the modified document back out to disk or to the network

Disadvantages of Using DOM

- One of the big issues is that DOM can be memory intensive. When an XML document is loaded, the entire document is read in at once. A large document will require a large amount of memory to represent it
- SAX, don't read in the entire document, so they are better in terms of memory efficiency for some applications
- DOM is not practical for small devices such as PDAs and cellular phones

DOM Levels

- The DOM working group works on phases (or *levels*) of the specification. There are three levels are in the works.
- The DOM Level 1 and Level 2 specifications are W3C recommendations. The specification of level 1and 2 are final

Level 1:

- Level 1 allows traversal of an XML document as well as the manipulation of the content in that document

Level2:

- Level 2 extends Level 1 with additional features such as namespace support, events, ranges, and so on

Level 3:

- It is currently a working draft. This means that it is under active development and subject to change.

DOM Core

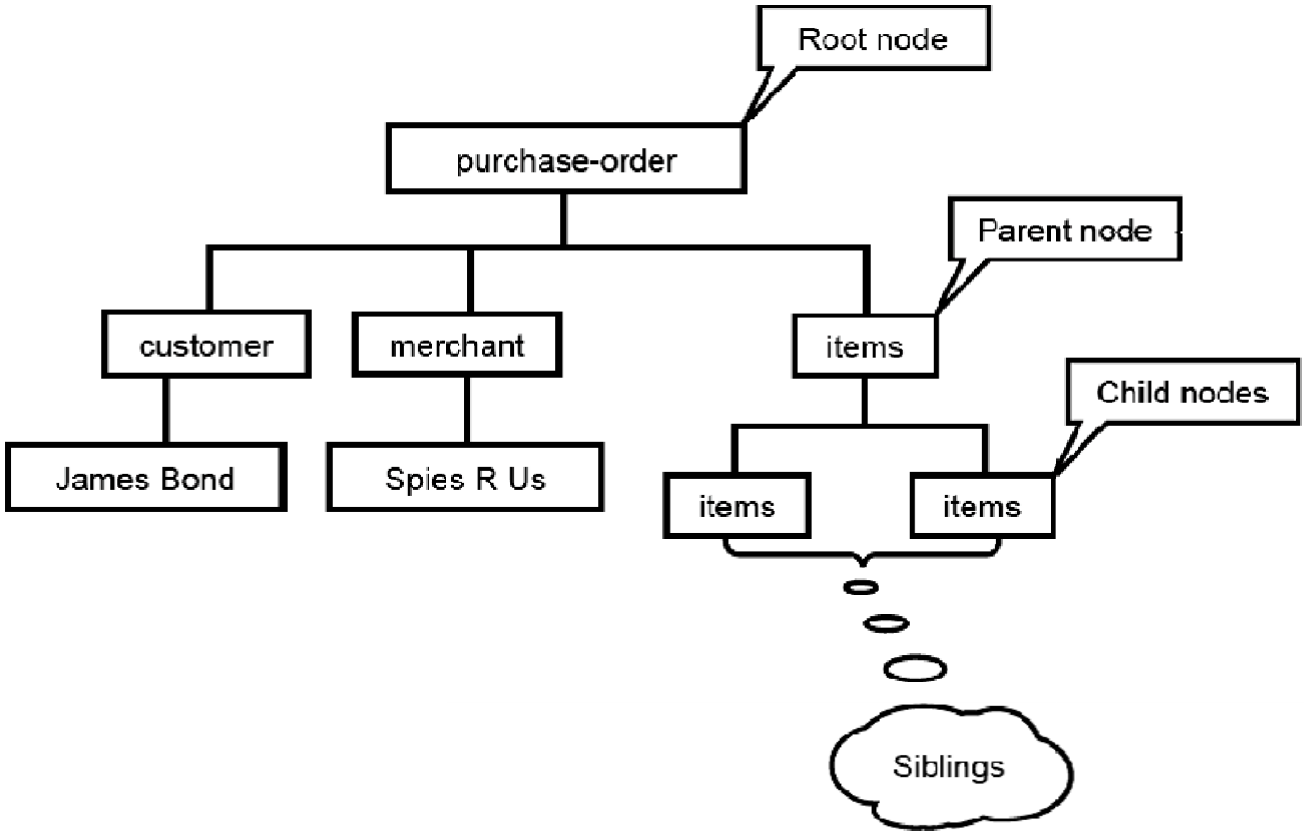
The DOM core is available in DOM Level 1 and beyond. It permits you to create and manipulate XML documents in memory.
DOM is a tree structure that represents elements, attributes, and content. As an example, let's consider a simple XML document, as shown below:

```
<purchase-order>
        <customer>James Bond</customer>
        <merchant>Spies R Us</merchant>
        <items>
                <item>Night vision camera</item>
                <item>Vibrating massager</item>
        </items>
</purchase-order>
```

Parents, Children, and Siblings

DOM specification uses the words *parents*, *children*, and *siblings* to represent nodes and their relationships to one another Parent nodes may have zero or more child nodes. Parent nodes themselves may be the child nodes of another parent node. The ultimate parent of all nodes is, of course, the *root* node. Siblings represent the child nodes of the same parent. These abstract descriptions of nodes are mapped to elements, attributes, text, and other information in an XML document. DOM interfaces contain methods for obtaining the parent, children, and siblings of any node. The root node has no parent, and there will be nodes that have no children or siblings.

The following Figureshows a diagram of the tree structure representing the XML document <purchase-order> listed above:

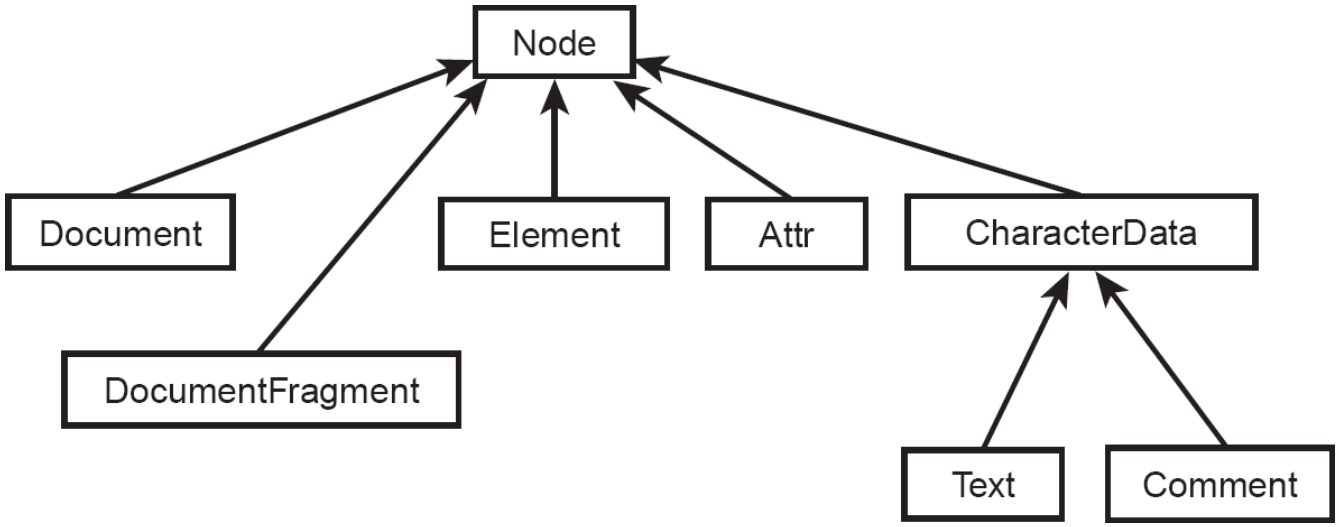


DOM Interfaces

The DOM interfaces are defined in IDL so that they are language neutral The fundamental interfaces are listed in the following table, along with a brief description of each:

INTERFACE	DESCRIPTION
Node	The primary interface for the DOM. It can be an element, attribute, text, and so on, and contains methods for traversing a DOM tree
NodeList	An ordered collection of Nodes
NamedNodeMap	An unordered collection of Nodes that can be accessed by name and used with attributes.
Document	A Node representing an entire document. It contains the root Node
DocumentFragment	A Node representing a piece of a document. It's useful for extracting or inserting a fragment into a document
Element	A Node representing an XML element.
Attr	A Node representing an XML attribute
CharacterData	A Node representing character data.
Comment	A CharacterData node representing a comment
DOMException	An exception raised upon failure of an operation.
DOMImplementation	Methods for creating documents and determining whether an implementation has certain features

The following figure shows shows the relationships among the interfaces described in The previous Table:



Common DOM methods

When you are working with the DOM, there are several methods you'll use often:

- **Document.getDocumentElement()** - Returns the root element of the document.
- **Node.getFirstChild()** - Returns the first child of a given Node.
- **Node.getLastChild()** - Returns the last child of a given Node.
- **Node.getNextSibling()** - These methods return the next sibling of a given Node.
- **Node.getPreviousSibling()** - These methods return the previous sibling of a given Node.
- **Node.getAttribute(attrName)** - For a given Node, returns the attribute with the requested name

The extended interfaces are listed in Table in the following table along with a brief description of each.

Interface	***Description***
CDATASection	Text representing CDATA
DocumentType	A node representing document type
Notation	A node with public and system IDs of a notation
Entity	A node representing an entity that's either parsed or unparsed
EntityReference	A node representing an entity reference
ProcessingInstruction	A node representing an XML processing instruction

Java Binding

- The DOM working group supplies Java language bindings as part of the DOM specification.
- These bindings are sets of Java source files containing Java interfaces, and they map exactly to the DOM interfaces described earlier

Popular DOM parser

- The package org.w3c.dom contains the Java interfaces but does not include a usable

implementation. In order to make the interfaces do something useful, you will need an implementation, or a *parser*

- A number of DOM implementations are available for Java. Two of the most popular are
    - **Java APIs for XML Processing (JAXP)**, developed by Sun Microsystems
    - **Xerces** developed as part of the Apache XML project

3. STEPS TO BE FOLLOWED WHEN USING DOM

Following are the steps used while parsing a document using DOM Parser.

- Import XML-related packages.
- Create a DocumentBuilder
- Create a Document from a file or stream
- Extract the root element
- Examine attributes
- Examine sub-elements

➢ **Import XML-related packages**

```
import org.w3c.dom.*;
import javax.xml.parsers.*;
```

➢ **Create a DocumentBuilder**

```
DocumentBuilderFactory factory =DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
```

➢ **Create a Document from a file or stream**

```
StringBuilder xmlStringBuilder = new StringBuilder();
xmlStringBuilder.append("<?xml version="1.0"?> <class>
</class>");

ByteArrayInputStream input = new ByteArrayInputStream(
  xmlStringBuilder.toString().getBytes("UTF-8"));
Document doc = builder.parse(input);
```

➢ **Extract the root element**

```
Element root = document.getDocumentElement();
```

➢ **Examine attributes**

```
//returns specific attribute
element.getAttribute("attributeName");
//returns a Map (table) of names/values
element.getAttributes();
```

➢ **Examine sub-elements**

```
//returns a list of subelements of specified name
element.getElementsByTagName("subelementName");
```

```
//returns a list of all child nodes
element.getChildNodes();
```

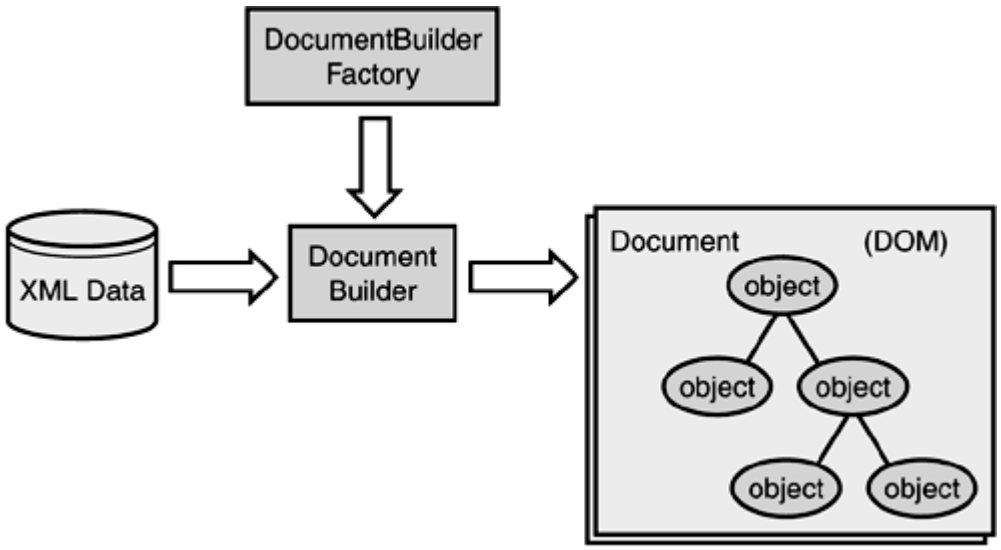
4. WALKING THROUGH AN XML DOCUMENT

- Let's look at an example in which we load an XML document from disk and print out some of its contents. In the first example, we will print out just the element names using getNodeName() from the Node interface.
- We will start from the root and recursively print all child node names, then indenting each level for clarity
- The source code for **SimpleWalker.java** is shown below:

```
import java.io.*; import org.w3c.dom.*;
import javax.xml.parsers.*;
public class SimpleWalker
{
        protected DocumentBuilder docBuilder;
        protected Element root;
public SimpleWalker() throws Exception
{
        DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
        docBuilder = dbf.newDocumentBuilder();
        DOMImplementation domImp = docBuilder.getDOMImplementation();
        if (domImp.hasFeature("XML", "2.0"))
                System.out.println("Parser supports extended interfaces");
}
public void parse(String fileName) throws Exception
{
        Document doc = docBuilder.parse(new FileInputStream(fileName));
        root = doc.getDocumentElement();
        System.out.println("Root element is " + root.getNodeName());
        printElement("", root);

}
public void printElement(String indent, Node aNode)
{
        System.out.println(indent + "<" + aNode.getNodeName() + ">");
        Node child = aNode.getFirstChild();
        while (child != null) {
                printElement(indent + "\t", child);
                child = child.getNextSibling();
        }
        System.out.println(indent + "</" + aNode.getNodeName() + ">");
}
public static void main(String args[]) throws Exception
```

```
{
        SimpleWalker sw = new SimpleWalker();
        sw.parse(args[0]);
}
}
```

- Looking at the code, javax.xml.parsers package contains two critical classes for DOM: DocumentBuilder and DocumentBuilderFactory. These classes are needed because the DOM interfaces do not provide a way to load or create documents
- There are several methods in the DocumentBuilder class for loading and parsing an XML file. You can supply a java.io.File, an InputStream, or other sourceWe will use FileInputStream to load our file, but first we need to get an instance of DocumentBuilder which is an abstract class, so we can't create an instance directly.
- That's the job of DocumentBuilderFactory, which is also abstract, but it has a static factory method, newInstance(), that we can use to create a DocumentBuilder.
- From there Once we have a Document object, we can get the root element by calling the getDocumentElement() method. It turns out that the Document object itself is a node, but it's not the root node. We must call getDocumentElement() to get the root we can use one of the parse() methods to give us a Document object.
- Now we are totally in the DOM world. We can also obtain a DOMImplementation to find out what features our parser has. In this case, we are trying to find out whether extended interfaces are supported



Sample input file:

A sample XML input file, **library.xml is shown below:**

```
<?xml version="1.0" encoding="UTF-8"?>
<library>
<fiction>
<book>Moby Dick</book>
</fiction>
<biography>
```

<book>The Last Lion, Winston Spencer Churchill</book>
</biography>
</library>
To run the program:
The example can be executed using the following command:
E:\xmldom>java SimpleWalker library.xml

Sample output:
Parser supports extended interfaces
Root element is library
<#text>
</#text>
<fiction>
<#text>
</#text>
<book>
<#text>
</#text>
</book>
</fiction>
<#text>
</#text>
<biography>
<#text>
</#text>
<book>
<#text>
</#text>
</book>
<#text>
</#text>
</biography>
<#text>
</#text>
</library>

Look at the output, what are all those <#text> elements. any text in an XML document becomes a child node in DOM. If we call
getNodeName() on a text node, we get #text, not the text itself. If we want to get the text, we must determine whether we have a text node and then call getNodeValue(). We need only make a minor modification to the printElement() method of the above program. The modified version is shown below:
public void printElement(String indent, Node aNode)

```
{
        if (aNode.getNodeType() == Node.TEXT_NODE)
                System.out.println(indent + aNode.getNodeValue());
        else
        {
                System.out.println(indent + "<" + aNode.getNodeName() + ">");
                Node child = aNode.getFirstChild();
                while (child != null) {
                        printElement(indent + "\t", child);
                        child = child.getNextSibling();
                }
                System.out.println(indent + "</" + aNode.getNodeName() + ">");
        }
}
```

Once again compile and run the above program:

The example can be executed using the following command:

E:\xmldom>java SimpleWalker library.xml

Parser supports extended interfaces

Root element is library

```
<library>
<fiction>
        <book>
                Moby Dick
        </book>
</fiction>
<biography>
        <book>
                The Last Lion, Winston Spencer Churchill
        </book>
</biography>
</library>
```

- **Notice that** DOM parser treats any whitespace between elements as text.
- Depending on the type of node, we might need to use getNodeName(), getNodeValue(), or maybe getAttributes().the following Table summarizes what each of the methods gives you, depending on the interface type:

Interface	**getNodeName()**	**getNodeValue()**	**getAttributes()**
Attr	Name of the attribute	Value of the attribute	Null
CDATASection	#cdata-section	Content of the CDATA section	Null
Comment	#comment	Content of the comment	Null

Document	#document	Null	Null
DocumentFragment	#document-fragment	null	Null
DocumentType	Document type name	Null	Null
Element	Tag name	Null	NamedNodeMap
Entity	Entity name	Null	-
EntityReference	Name of the entity referenced	Null	Null
Notation	Notation name	Null	Null
ProcessingInstruction	Target	Entire content excluding the target	Null
Text	#text	Content of the text node	Null

It's important to note that attributes are not child nodes of elements. You must explicitly call getAttributes() to obtain a NamedNodeMap containing the attributes. NamedNodeMap is convenient for attributes because you can easily get a specific attribute by name or by index (starting from 0)

5. CREATING AN XML DOCUMENT

- We will create an XML document in memory, from scratch, and then write it out to disk. This is very useful, if you have data from a non-XML source, such as a database, and you want to create an XML document based on the data

Here is the XML we need to create:

```
<?xml version="1.0" encoding="UTF-8"?>
<cars>
   <carname type="formula one">Ferrari 101</carname>
   <carname type="sports">Ferrari 202</carname>
</cars>
```

Here is the java source code that create the xml file shown above:

CreateXmlFileDemo.java

```
Import java.io.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;
import javax.xml.transform.dom.*;
public class CreateXmlFileDemo {
  public static void main(String argv[])
    { try {
      DocumentBuilderFactory dbFactory =
      DocumentBuilderFactory.newInstance(); DocumentBuilder dBuilder =
      dbFactory.newDocumentBuilder();
      Document doc = dBuilder.newDocument();
```

```
        // root element
        Element cars = doc.createElement("cars");
        doc.appendChild(rootElement);

        // carname  element
        Element car1 = doc.createElement("carname");
        Attr attrType = doc.createAttribute("type");
        attrType.setValue("formula one");
        car1.setAttributeNode(attrType);
        car1.appendChild(doc.createTextNode("Ferrari 101"));
        cars.appendChild(car1);

        // carname  element
        Element car2 = doc.createElement("carname");
        Attr attrType1 = doc.createAttribute("type");
        attrType1.setValue("sports");
        car2.setAttributeNode(attrType1);

        car2.appendChild(doc.createTextNode("Ferrari 202"));
        car2.appendChild(car2);

        // write the content into xml file
        TransformerFactory transformerFactory = TransformerFactory.newInstance();
        Transformer transformer = transformerFactory.newTransformer();
        DOMSource source = new DOMSource(doc);

        //output to c:\cars.xml file
        StreamResult result = new StreamResult(new
        File("C:\\cars.xml")); transformer.transform(source, result);

        // Output to console for testing
        StreamResult consoleResult = new StreamResult(System.out);
        transformer.transform(source, consoleResult);
      } catch (Exception e)
        {
        e.printStackTrace();
      }
    }
}
```

Run the program:

e:\dom> java CreateXmlFileDemo

now you will see the XML output, just shown above in the console. You can also open cars.xml file from c:\

6. DOM TRAVERSAL AND RANGE

- Traversal and range are features added in DOM Level 2. They are supported by Apache Xerces. You can determine whether traversal is supported by calling the ***hasFeature("Traversal", "2.0")*** method of the DOMImplementation interface.

Traversal

- Traversal is a convenient way to walk through a DOM tree and select specific nodes. This is useful when you want to find certain elements and perform operations on them.

Traversal Interfaces

- The traversal interfaces are listed in the following Table, along with a brief description of each

Interface	*Description*
NodeIterator	Used to walk through nodes linearly. Represents a subtree as a linear list
TreeWalker	Represents a subtree as a tree view
NodeFilter	Can be used in conjunction with NodeIterator and TreeWalker to select specific nodes
DocumentTraversal	Contains methods to create NodeIterator and TreeWalker instances

Traversal Example

Let's look at an example in which traversal is used. Let's say we want to print out just the names of books in our library. We can use NodeIterator to iterate through all the nodes and define a NodeFilter to select only the nodes with the name "book." When we find a book node, we can get the value of the text content and print it out

There are two classes we need to define. The first one, IteratorApp.java, contains the application code. The second one, NameNodeFilter.java, selects nodes with a given name

The source code for **IteratorApp.java** is shown below:

```
public class IteratorApp
{
        protected DocumentBuilder docBuilder;
        protected Document document;
        protected Element root;
public IteratorApp() throws Exception {
        DocumentBuilderFactory dbf =
        DocumentBuilderFactory.newInstance(); docBuilder =
        dbf.newDocumentBuilder();
        DOMImplementation domImp =
        docBuilder.getDOMImplementation(); if
        (domImp.hasFeature("Traversal", "2.0"))
                        System.out.println("Parser supports Traversal");
}
```

```
public void parse(String fileName) throws Exception {
        document = docBuilder.parse(new FileInputStream(fileName));
        root = document.getDocumentElement();
        System.out.println("Root element is " + root.getNodeName());
}
public void iterate() {
        NodeIterator iter =((DocumentTraversal)document).createNodeIterator(
                                root,
                                NodeFilter.SHOW_ELEMENT, new
                                NameNodeFilter("book"), true);
        Node n = iter.nextNode();
        while (n != null) {
                System.out.println(n.getFirstChild().getNodeValue())
                ; n = iter.nextNode();

        }
}
public static void main(String args[]) throws Exception
        { IteratorApp ia = new IteratorApp();
        ia.parse(args[0]);
        ia.iterate();
}
}
```

The source code for **IteratorApp.java** is shown below:

```
public class NameNodeFilter implements NodeFilter
        { protected String name;
public NameNodeFilter(String inName)
                { name = inName;
}
public short acceptNode(Node n) {
        if (n.getNodeName().equals(name))
                        return FILTER_ACCEPT;
        else
                return FILTER_REJECT;
}
}
```

- We can create an instance of NodeIterator from the DocumentTraversal interface. But how do we get an instance of a DocumentTraversal interface? It turns out that if traversal is supported, the Document instance will also implement DocumentTraversal. If you look carefully at the iterate() method, you will see that the document is downcast into DocumentTraversal. The cast succeeds because traversal is supported by our

implementation (Xerces). If it wasn't supported, a ClassCastException would be raised at runtime

- The method for creating a NodeIterator is createNodeIterator(...), which accepts four parameters:
    - the **root node**
    - **a flag** determining which nodes to show
    - a possible NodeFilter
    - **a flag** determining whether entity references are to be expanded.
- In our example, we start at the document root, because we want to search the entire document. Constants in the NodeFilter interface define which nodes will be visible. You can choose options such as elements, attributes, text, and so on**.** The NodeFilter is optional. If you don't want to use a NodeFilter, just supply **"null"** and no filter will be applied.
- Here's the output from IteratorApp:

Parser supports
Traversal Root element
is library Moby Dick
The Last Lion, Winston Spencer Churchill

Range

Range interfaces provide a convenient way to select, delete, extract, and insert content.
A range consists of two boundary points corresponding to the start and the end of the range. A boundary point's position in a Document or DocumentFragment tree can be characterized by a node and an offset. The node is the container of the boundary point and its position. The container and its ancestors are the ancestor containers of the boundary point and its position. The offset within the node is the offset of the boundary point and its position. If the container is an Attr, Document, DocumentFragment, Element, or EntityReference node, the offset is between its child nodes. If the container is a CharacterData, Comment, or ProcessingInstruction node, the offset is between the 16- bit units of the UTF-16 encoded string contained by it.

The content of a range must be entirely within the subtree rooted by a single Document, DocumentFragment, or Attr node. This common ancestor container is known as the *root container* of the range. The tree rooted by the root container is known as the range's *context tree*
The boundary points of a range must have a common ancestor container that is either a **Document, DocumentFragment**, or **Attr** node. **DocumentType, Entity**, or **Notation** nodes cannot be ancestor of boundary point

Range Interfaces

The range interfaces are listed in the following table, along with a brief description of each:

Interfac	*Description*
Range	This interface describes a range and contains methods to define, delete, insert content.
DocumentRange	This interface creates a range

Range Example

Let's look at an example in which range is used. Let's say we want to delete the first child node under the root. The source code for RangeApp.java is shown in the following Listing. We must import **org.w3c.dom.range** in order to refer to the range interfaces.

```
import org.w3c.dom.*;
import org.w3c.dom.ranges.*;
import javax.xml.parsers.*;
public class RangeApp {
        protected DocumentBuilder docBuilder;
        protected Document document;
        protected Element root;
public RangeApp() throws Exception {
        DocumentBuilderFactory dbf =
        DocumentBuilderFactory.newInstance(); docBuilder =
        dbf.newDocumentBuilder();
        DOMImplementation domImp =
        docBuilder.getDOMImplementation(); if
        (domImp.hasFeature("Range", "2.0"))
                System.out.println("Parser supports Range");
}
public void parse(String fileName) throws Exception {
        document = docBuilder.parse(new FileInputStream(fileName));
        root = document.getDocumentElement();
        System.out.println("Root element is " + root.getNodeName());
}
public void deleteRange() {
        Range r = ((DocumentRange)document).createRange();
        r.selectNodeContents(root.getFirstChild()); //select the range of first child node of root
        r.deleteContents();//remove the range of first child node of root

}
public static void main(String args[]) throws Exception
        { RangeApp ra = new RangeApp();
        ra.parse(args[0]);
        ra.deleteRange();
}
}
```

Consider the following xml file:address.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<address>
   <fname> kajendran </fname>
   <street>11/20, valluvar street</street>
   <city>Chennai</city>
</address>
```

If you run the above program using the following command:

E:\dom>java RangeApp address.xml

After successful execution of this command, the content of address.xml file would be:

```
<?xml version="1.0" encoding="UTF-8"?>
<address>
   <street>11/20, valluvar street</street>
   <city>Chennai</city>
</address>
```

7. OTHER DOM IMPLEMENTATIONS

For resource-constrained devices such as PDAs and cellular phones, DOM is not suitable, because it will takeup lots of memory. For these applications, a number of DOM-like APIs have appeared.

1) JDOM

- It was originally developed as an open-source API for XML but has been accepted by the Java Community Process (JCP JSR-102)
- JDOM was designed specifically for Java. In contrast, DOM is purely an interface specification independent of any language
- The goal of W3C DOM is to be language independent, which works but can add a lot of unnecessary complications. Here are some of the guiding principles of JDOM:
  - ✓ JDOM should be straightforward for Java programmers.
  - ✓ JDOM should support easy and efficient document modification.
  - ✓ JDOM should hide the complexities of XML wherever possible
  - ✓ JDOM should integrate with DOM and SAX.
  - ✓ JDOM should be lightweight and fast.
  - ✓ JDOM should solve 80 percent (or more) of Java/XML problems with 20 percent(or less) of the effort when compare with DOM

JDOM Example

Let’s create an XML document using JDOM. The source code for JDOMCreate.java appears in the following Listing

```
import org.jdom.*;
```

```
import org.jdom.output.*;
public class JDOMCreate
{
    public static void main(String args[]) throws Exception
        { Element root = new Element("library");
        Document doc = new Document(root);
        Element fiction = new Element("fiction");
        Element book = new Element("book");
        book.setAttribute("author", "Herman Melville");
        book.addContent("Moby Dick");
        fiction.addContent(book);
        root.addContent(fiction);
        XMLOutputter outputter = new XMLOutputter("\t", true);

        outputter.output(doc, System.out);
    }
}
```

If you run the above program using:

```
e:\dom> java JDOMCreate
```

The output is shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<library>
    <fiction>
        <book author="Herman Melville">Moby Dick</book>
    </fiction>
</library>
```

Reading XML document

- Reading and parsing an XML document is even easier. As mentioned earlier, JDOM is not meant to be a parser replacement. JDOM uses existing parsers to avoid reinventing the wheel.
- If you have an existing DOM or SAX parser, you can use it with JDOM
- The following example parses an XML document and then prints it out using XMLOutputter. The source code for JDOMParse.java appears in the following Listing:

```
public class JDOMParse  {
    public static void main(String args[]) throws Exception
        { SAXBuilder builder = new SAXBuilder();
        Document doc = builder.build(new
        File(args[0]));
        XMLOutputter outputter = new XMLOutputter("\t", true);
        outputter.output(doc, System.out);
    }
}
```

Other popular parser includes:

- **StAX Parser** - Parses the document in similar fashion to SAX parser but in more efficient way.
- **XPath Parser** - Parses the XML based on expression and is used extensively in conjuction with XSLT.
- **DOM4J Parser** - A java library to parse XML, XPath and XSLT using Java Collections Framework , provides support for DOM, SAX and JAXP.

Small DOM-like Implementations for hand-held devices

- **PDAs** and cellular phones are rapidly becoming the terminals of choice for people on the run. They are a lot easier to carry compared to a laptop. If you're going to work with XML on a PDA, something like DOM is a great help.
- There are smaller, simpler alternatives API available for PDAs, and you have several solutions from which to choose.

NanoXML

- It looks a lot like DOM, but it's much smaller Version 2.0 is about 33KB, but a light version is available that's less than 6KB
- The API contains a class called XMLElement, which is very similar to the Node interface found in DOM

TinyXML

- It's primarily for reading in an XML document, because it does not provide facilities to create a document. It's extremely simple, based primarily on one class, TinyParser, and one interface, ParsedXML. All you need to do is call a static method in TinyParser to parse a stream, file, or URL. This gives you an instance of a ParsedXML interface that has only seven methods

kXML

- kXML is a DOM-like parser in the spiritof JDOM. designed specifically for J2ME resource-constrained devices.

8. JAVA ARCHITECTURE FOR XML BINDING (JAXB)

- JAXB provides a means of automatically binding XML with Java objects. JAXB is being developed through the Java Community Process (JCP) under JSR-31.
- JAXB can be considered a *serialization* mechanism from Java objects to XML.
- Serialization is the process of converting an object in memory into a stream of data, and vice versa. Serialization is a convenient way of storing objects on disk or sending them over a network. Object serialization based on serializable and externalizable interfaces
- In the case of JAXB, a set of binding classes is generated using a schema compiler. The classes manage *marshalling*, meaning translating Java objects to XML and back again. Here is a brief summary of some of benefits of JAXB:
    - Valid data is guaranteed. Marshalling is based on a schema, which constrains

the structure of the XML.

- JAXB is faster and requires less memory when compared with DOM. DOM includes a lot of functionality for manipulating arbitrary documents.
- JAXB is relatively easy to use. All you need to do is supply a schema and generate binding classes using a schema compiler. From there, reading, writing, and modifying XML is simply a matter of a few method calls
- JAXB applications are extensible. The generated classes can be used as is, or they can be subclassed for reusability and added functionality

Data Binding

- **A** class and a schema perform similar functions. Classes describe Java objects, whereas schemas describe XML documents. An object is an instance of a class, and a document follows a schema. The diagram in the figure shown in next page, illustrates the relationships between schemas, classes, documents, and objects.
- If we have a schema, perhaps in the form of a DTD, we can automatically generate classes that translate between objects and documents

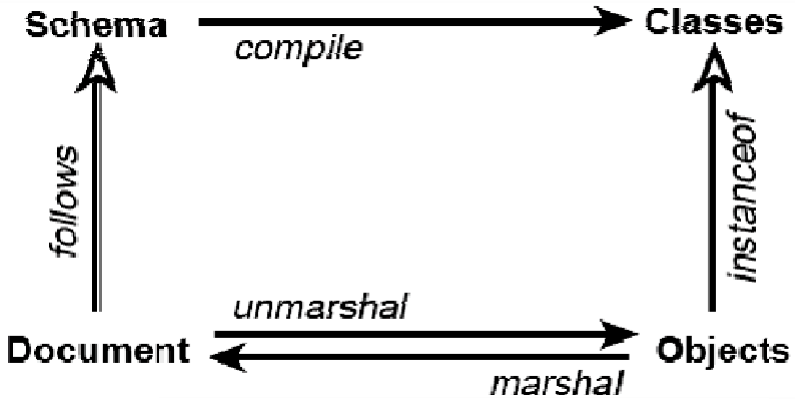


Figure : binding relationship

- One way to define a binding is to generate one Java class for every element in a schema.
- Attributes within an element are mapped to String fields. Content within an element is a little more complicated. The following Table summarizes how the content is mapped within a Java class.
- Default Content Binding

Content Type	***Field Type***
PCDATA	String
Fixed number of elements	References to sub-element types
Varying number of elements	java.util.List
Any	Can be defined using additional information described in a binding schema

9. PARSING XMLUSING SAX

- Simple API for XML (SAX) can only be used for parsing existing documents, but cannot be used for creating xml documents

- SAX is an event-based API. It provides a framework for defining event listener or handler. Unlike a DOM parser, a SAX parser creates no parse tree
- Instead of loading an entire document into memory all at once, SAX parsers read documents and notify a client program when **elements, text, comments**, and **other data of interest are found**. SAX parsers send you events continuously, telling you what was found next

- An applications using SAX receive event notifications about the XML document being processed an element, and attribute, at a time in sequential order starting at the top of the document, and ending with the closing of the ROOT element
- SAX Parser does the following for the client application:
    - Reads an XML document from top to bottom, recognizing the tokens that make up a well-formed XML document
    - Tokens are processed in the same order that they appear in the document
    - Reports the application program the nature of tokens that the parser has encountered as they occur
    - The application program provides an "event" handler that must be registered with the parser
    - As the tokens are identified, callback methods in the handler are invoked with the relevant information

10. SAX VERSIONS

SAX 1.0

- The first version, SAX 1.0, was released in May 1998, It provided the basic functionality needed to read elements, attributes, text, and to manage errors. There was also some DTD support.

SAX 2.0

- The current version, SAX 2.0, was released two years later in May 2000. Many of the SAX 2.0 interfaces are departures from SAX 1.0. Older interfaces are included, but deprecated, for backward compatibility
- SAX 2.0 also includes support for namespaces and extensibility through features and properties.

11. SAX Vs DOM

- SAX is, in many ways, much simpler than DOM. DOM is an in-memory tree structure of an XML document or document fragment.
- DOM is a natural object model of an XML document, but it's not always practical. Large documents can take up a lot of memory

DOM	***SAX***
The DOM parses XML *in space*	SAX parses XML *in time*
Tree based API	Event based API

DOM parser loads whole XML document in memory	SAX only loads a small part of the XML file in memory, based on event notification
It is not suitable for large volume of document, since it loads the entire xml document	Good for very large documents, since it loads only portion of xml document
DOM can be used for creating xml document	SAX cannot be used for creating xml document
DOM contain many interfaces, and each interfaces containing many methods	SAX parser tends to smaller than DOM implementation
There is formal specification for DOM	There is no formal specification for SAX
You cannot process documents larger than available system memory	You can process documents larger than available system memory
DOM stands for Document object model	SAX stands for Simple API for XML
DOM is read and write parser	SAX is a read-only parser
DOM is slower than SAX, because you **have to wait for the entire document to be loaded**	**SAX can be faster**, because you don't have to wait for the entire document to be loaded
DOM support random access	**No support for random access**

12. SAX PACKAGES

- The SAX 2.0 API is comprised of two standard packages and one extension package. The standard packages are **org.xml.sax** and **org.xml.helpers**.

org.xml.sax package

- The **org.xml.sax** package contains the **basic classes, interfaces,** and **exceptions** needed for parsing documents
- A summary of the org.xml.sax package is shown in the following Table:

Name	***Description***
Interfaces	
Attributes	Interface for a list of XML attributes.
AttributeList	Deprecated. This interface has been replaced by the SAX2 Attributes interface, which includes namespace support
ContentHandler	Receives notification of the logical content of a document.
DocumentHandler	Deprecated. This interface has been replaced by the SAX2 ContentHandler interface, which includes namespace support.
ContentHandler	interface, which includes namespace support.
DTDHandler	Receives notification of basic DTD-related events.

EntityResolver	Basic interface for resolving entities.
ErrorHandler	Basic interface for SAX error handlers.
Locator	Interface for associating a SAX event with a document location.
Parser	Deprecated. This interface has been replaced by the SAX2 XMLReader interface, which includes namespace support.
XMLFilter	Interface for an XML filter.
XMLReader	Interface for reading an XML document using callbacks
Classes	
HandlerBase	Deprecated. This class works with the deprecated DocumentHandler interface.
InputSource	A single input source for an XML entity
Exceptions	
SAXException	Encapsulates a general SAX error or warning.
SAXNotRecognizedException	Exception class for an unrecognized identifier.
SAXNotSupportedException	Exception class for an unsupported operation.
SAXParseException	Encapsulates an XML parse error or warning

ContentHandler Methods

Descriptions of all the methods defined in ContentHandler are provided in the following Table:

Method	***Description***
characters()	Receives notification of character data
endDocument()	Receives notification of the end of a document
endElement()	Receives notification of the end of an element
endPrefixMapping()	Ends the scope of a prefix-URI mapping
ignorableWhitespace()	Receives notification of ignorable whitespace in element content
processingInstruction()	Receives notification of a processing instruction
setDocumentLocator()	Receives an object for locating the origin of SAX document events
skippedEntity()	Receives notification of a skipped entity
startDocument()	Receives notification of the beginning of a document
startElement()	Receives notification of the beginning of an element
startPrefixMapping()	Begins the scope of a prefix-URI namespace mapping

Attributes Interface methods

- This interface specifies methods for processing the attributes connected to an element.
  - **int getLength()** - Returns number of attributes.
  - **String getQName(int index)**

- **String getValue(int index)**
- **String getValue(String qname)**

The org.xml.sax.helpers package

- This package contains additional classes that can simplify some of your coding and make it more portable. You will find a number of adapters that implement many of the handler interfaces, so you don't need to fill in all the methods defined in the interfaces. Factory classes provide a mechanism for obtaining a parser independent of the implementation
- A summary of the org.xml.sax.helpers package is shown in the following Table:The **org.xml.sax.helpers** Package

Class	*Description*
AttributeListImpl	Deprecated. This class implements a deprecated interface, AttributeList that has been replaced by Attributes, which is implemented in the AttributesImpl helper class.
AttributesImpl	Default implementation of the Attributes interface.
DefaultHandler	Default base class for SAX2 event handlers.
LocatorImpl	Provides an optional convenience implementation of Locator.
NamespaceSupport	Encapsulate namespace logic for use by SAX drivers.
ParserAdapter	Adapts a SAX1 Parser as a SAX2 XMLReader.
ParserFactory	Deprecated. This class works with the deprecated Parser interface.
XMLFilterImpl	Base class for deriving an XML filter.
XMLReaderAdapter	Adapts a SAX2 XMLReader as a SAX1 Parser.
XMLReaderFactory	Factory for creating an XML reader

The org.xml.sax.ext package

- It is an extension package that is not shipped with all implementations. It contains two handler interfaces for capturing declaration and lexical events
- A summary of the org.xml.sax.ext package is shown in the following Table:

The org.xml.sax.ext Package

Interface	*Description*
DeclHandler	SAX2 extension handler for DTD declaration events
LexicalHandler	SAX2 extension handler for lexical events

13. WORKING WITH SAX

Let's look at a simple example in which we read an XML document from disk and print out some of the contents

In this example, we will print out just the element names and the text between the elements. The source code for **SAXDemo.java** is shown in the following List:

```
import java.io.*;
import org.xml.sax.*;
```

```
import org.xml.sax.helpers.*;
import javax.xml.parsers.*;
public class SAXDemo extends DefaultHandler
{
	public void startDocument()
	{
		System.out.println("***Start of Document***");
	}
	public void endDocument()
	{
		System.out.println("***End of Document***");
	}
	public void startElement(String uri, String localName,String qName, Attributes
	                                                                    attributes)
	 {
		System.out.print("<" + qName);
		int n = attributes.getLength();
		for (int i=0; i<n; i+=1)
			System.out.print(" " + attributes.getQName(i) +
					"="" + attributes.getValue(i) + """);


		System.out.println(">");
	}
	public void characters(char[] ch, int start, int length)
	{
		System.out.println(new String(ch, start, length).trim());
	}
	public void endElement(String namespaceURI, String localName,String qName)
	                                        throws SAXException
	{
		System.out.println("</" + qName + ">");
	}
	public static void main(String args[]) throws Exception
	{
		if (args.length != 1)
		 {
			System.err.println("Usage: java SAXDemo <xml-
			file>"); System.exit(1);
		}
		SAXDemo handler = new SAXDemo();
		SAXParserFactory factory =
```

```
            SAXParserFactory.newInstance(); SAXParser parser =
            factory.newSAXParser(); parser.parse(new File(args[0]),
            handler);
        }
}
```

- Our class extends ***DefaultHandler*** in order to capture events. DefaultHandler is a convenience adapter class defined in org.xml.sax.helpers. **It implements four interfaces: EntityResolver, DTDHandler, ContentHandler,** and **ErrorHandler**. DefaultHandler defines empty stub methods for all the events defined in all four interfaces
- In order to register our handler, we can create a SAXParser instance and call its **parse( )** method with a file and handler instance. The code to do this is located in the main() method of the example
- In the example, **we have defined five methods: startDocument(), endDocument(), startElement(), characters(), and endElement()**. These methods will be called in response to related events, and they are defined in the ContentHandler interface
- Once the **parse( )** method is called, our methods will be called in response to events until the end of input is reached or an error occurs.
- A sample XML document, **library.xml**, is used for testing the above program:

```
<?xml version=”1.0” encoding=”UTF-8”?>
<!-- A short list of books in a library -->
<!DOCTYPE library SYSTEM “library.dtd”>
<library>
        <fiction>
                <book author=”Herman Melville”>Moby Dick</book>
        </fiction>
        <biography>
                <book author=”William Manchester”>
                The Last Lion, Winston Spencer
                Churchill
                </book>
        </biography>
</library>
```

To execute SAXDemo, you can enter the following command:

D:\SAX>java SAXDemo library.xml

```
***Start of Document***
<library>
<fiction>
<book author=’Herman
Melville’> Moby Dick
</book>

</fiction>
```

<biography>
<book author=’William Manchester’>
The Last Lion, Winston Spencer
Churchill
</book>
</biography>
</library>
End of Document

Validation

- SAX parsers come in two varieties: **validating parsers** and **nonvalidating parsers**.
- Validating parsers can determine whether an XML document is valid based on a Document Type Definition (DTD) or Schema
- The SAX parser shipped with **Apache Xerces** is a validating parser**.**
- In order to use validation, you must turn it on by setting the validation feature to true. If you attempt to turn on validation with a nonvalidating parser, a ***SAXNotSupportedException*** will be thrown. If the parser does not recognize the feature, a ***SAXNotRecognizedException*** will be thrown. This helps in determining whether you mistyped the feature name
- Consider the following library.dtd file for validating library.xml file:

  <?xml version=”1.0” encoding=”US-ASCII”?>
  <!ELEMENT library (fiction|biography)*>

  <!ELEMENT fiction (book)+>

  <!ELEMENT biography (book)+>

  <!ELEMENT book (#PCDATA)>

  <!ATTLIST book author CDATA #REQUIRED>

- In the following example, we will write a simple program to validate an XML document. In this example, the DTD will be located on the local hard drive in the same directory as the document itself.
- The source code for SAXValidator.java is shown in the following Listing:

```
import java.io.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;
public class SAXValidator extends DefaultHandler
{
        private boolean valid; private
        boolean wellFormed; public
        SAXValidator() {
                valid = true;
                wellFormed = true;
```

```
    }
    public void error(SAXParseException e)
        { valid = false;
    }
    public void fatalError(SAXParseException e)
        { wellFormed = false;
    }
    public void warning(SAXParseException
        e) { valid = false;
    }
    public boolean isValid()
        { return valid;
    }
    public boolean isWellFormed()
        { return wellFormed;
    }
    public static void main(String args[]) throws Exception
        { if (args.length != 1) {
            System.out.println("Usage: java SAXValidate <xml-
            file>"); System.exit(1);
        }
        XMLReader parser = XMLReaderFactory.createXMLReader(
                                "org.apache.xerces.parsers.SAXParser");
        parser.setFeature("http://xml.org/sax/features/validation", true);
        SAXValidator handler = new SAXValidator();
        parser.setContentHandler(handler);
        parser.setErrorHandler(handler);
        parser.parse(new InputSource(new FileReader(args[0])));
        if (!handler.isWellFormed())
            System.out.println("Document is NOT well
         formed."); if (!handler.isValid())
            System.out.println("Document is NOT
        valid."); if (handler.isWellFormed() &&
        handler.isValid()) {
            System.out.println("Document is well formed and valid.");
      }
    }
}
```

To execute **SAXValidate**, you can enter the following command:

D:\SAX>java SAXValidate library.xml

Document is NOT valid.

14. HANDLING ERRORS

- The **Locator** interface can give us the parse position where error occurred within a **ContentHandler** method.

- The position information includes **line number** and **column number**. It is important to note that the **Locator object should not be used** in any other methods, including **ErrorHandler** methods.
- Fortunately, **ErrorHandler** methods supply a **SAXParseException** object that can also give us position information
- The source code for **SAXErrors.java** is shown in the following Listing:

```
import java.io.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;
public class SAXErrors extends DefaultHandler
        { private Locator locator;
        public void startDocument() {
                System.out.println("***Start of Document***");
        }
        public void endDocument() {
                System.out.println("***End of Document***");
        }
        public void setDocumentLocator(Locator inLocator)
                { System.out.println("***Got Locator***");
                locator = inLocator;
                int line = locator.getLineNumber();
                int column = locator.getColumnNumber();
                System.out.println("Line " + line + ", column " +
                column);
         }
        public void printLocation(SAXParseException
                e) { int line = e.getLineNumber();
                int column = e.getColumnNumber();
                System.out.println("Line " + line + ", column " +
                column);
          }
        public void error(SAXParseException e)
                { printLocation(e);
                System.out.println("Recoverable error: " + e.getMessage());
                Exception ex = e.getException();
        }
        public void fatalError(SAXParseException e)
                { printLocation(e);
                System.out.println("Non-recoverable error: " + e.getMessage());
```

```
    }
    public void warning(SAXParseException e)
        { printLocation(e);
        System.out.println("Warning: " + e.getMessage());
    }
    public static void main(String args[]) throws Exception
        { if (args.length != 1) {
            System.err.println("Usage: java SAXErrors <xml-
            file>"); System.exit(1);
        }
        XMLRe0ader parser = XMLReaderFactory.createXMLReader(
                                "org.apache.xerces.parsers.SAXParser");
        parser.setFeature("http://xml.org/sax/features/validation", true);
        SAXErrors handler = new SAXErrors();
        parser.setContentHandler(handler);
        parser.setErrorHandler(handler);
        parser.parse(new InputSource(new FileReader(args[0])));
    }
}
```

- The ContentHandler method **setDocumentLocator( )** is added to obtain a Locator instance. Detailed information is printed in the error methods
- In library.xml file change **<fiction>** into **<fictions>, then save the xml file**
- Now, To execute **SAXValidate**, you can enter the following command:

D:\SAX>java SAXErrors library.xml
*****Got**
Locator* Line**
1, column 1

*****Start of**
Document* Line 4,**
column 12

Recoverable error: Element type "fictions" must be declared.

*****End of Document*****

- As expected, a validation error occurs at line 4. The fictions tag should be fiction\

LEXICAL EVENTS

- ***DefaultHandler does not handle*** CDATA, DTD references, comment, entity,etc.
- We can receive these events as well using an extension interface called **LexicalHandler**. LexicalHandler is part of the **org.xml.sax.ext** package

- The methods in the LexicalHandler interface are listed below:

-

Method	*Description*
comment()	Receives notification when comment found while processing xml document
endCDATA()	Receives notification when the end of a CDATA section encountered
endDTD()	Receives notification when the end of DTD declarations encountered
endEntity()	Receives notification when the end of an entity encountered
startCDATA()	Receives notification when the start of a CDATA section encountered
startDTD()	Receives notification when the start of DTD declarations encountered
startEntity()	Receives notification when the beginning of some internal and external XML entities encountered

- The source code for **SAXLexical.java** is shown below:

```
public class SAXLexical extends DefaultHandler implements
        LexicalHandler { public SAXLexical() {}
        public void startDTD(String name, String publicId,String systemId) throws
                                                SAXException {
                System.out.print("*** Start DTD, name " +
                name);
          }
        public void endDTD() throws SAXException
                { System.out.println("*** End DTD
                ***");
        }
        public void startEntity(String name) throws SAXException {
                System.out.println("*** Start Entity " + name + " ***");
        }
        public void endEntity(String name) throws SAXException {
                System.out.println("*** End Entity " + name + " ***");
        }
        public void startCDATA() throws SAXException {
                System.out.println("*** Start CDATA ***");
        }
        public void endCDATA() throws SAXException {
                System.out.println("*** End CDATA ***");
        }
        public void comment(char[] ch, int start, int length) throws SAXException
                { System.out.println("*** Comment Encountered***");
        }

        parser.setFeature("http://xml.org/sax/features/validation", true);

         SAXLexical handler = new SAXLexical(); parser.setContentHandler(handler);
        parser.setProperty("http://xml.org/sax/properties/lexical-handler",handler);
```

```
        parser.parse(new InputSource(new FileReader(args[0])));
    }
}
```

- Now, To execute **SAXValidate**, you can enter the following command:

```
    D:\SAX>java SAXLexical library.xml
    *** Comment Encountered***
    *** Start DTD, name library SYSTEM library.dtd ***
    *** Start Entity [dtd] ***
    *** End Entity [dtd] ***
    *** End DTD ***

  public static void main(String args[])
    throws Exception { if (args.length
    != 1) {
        System.err.println("Usage: java
        SAXLexical <xml-file>");
        System.exit(1);
    }
    XMLReader parser = XMLReaderFactory.createXMLReader(
                                "org.apache.xerces.parsers.SAXParser");
```

IMPORTANT QUESTIONS

1. **What is the difference between Document Type Declaration (DOCTYPE) and Document Type Definition (DTD)?**

- A Document Type Declaration (DOCTYPE) and a DTD serve very different, although related purposes. The **DOCTYPE** is used to **identify and name the XML content**, whereas the **DTD** is **used to validate the metadata** contained within.

2. **Why should we avoid XML-attributes and use sub-element in xml definition?(or) What is the different between attribute and sub-element?**

- Generally, attributes are used to store Metadata, where as sub-elements are suer to store actual data
- Attributes cannot contain multiple values, whereas child elements can have multiple values.
- Attributes are not easily expandable. If you want to change in attribute's vales in future, it may be complicated.
- Attributes cannot describe structure, but child elements can.
- Attributes are more difficult to be manipulated by program code.
- Attributes values are not easy to test against a DTD

3. **Is XML declaration is processing instruction?**

- It seems like yes, but Xml declaration is not processing instruction
- Because, it provides instruction to the parser, but not to the application

4. **Difference between XML declaration and Processing Instruction?**

- The difference between PIs and XML declaration are listed below

XML declaration	Processing Instruction(PIs)
XML declaration provides instruction to the parser, but not the to application	PIs are used to embed application specific instructions into your xml documents
Xml declaration must always be the first line of your xml file	But, you can put PIs any where in the Xml document
Example:<?xml version="1.0"?>	**<?messageprocessor "process complete" ?>**

5. **What is XML processor (or) Parser?**

- It is a software module used to read xml documents and provides access to their content and structure

- It is doing its work onbehalf of another module called the application
- Xml processors is more commonly called Parsers

6. Why do we need XML namespaces?

There are really two fundamental needs for namespaces:

- To disambiguate between two elements that happen to share the same name
- To group elements relating to a common idea together

7. What is the difference between URN and URL?

- URLs and URNs are both used to create are used to create XML Namespace URIs

URL	URL
URL specifies the location of a resource, and how it can be retrieved	**the URN is simply a unique name,** which doesn't tell you anything about how to retrieve either the book itself or information about book

8. What is Qualified Name (or) QName?

An element or attribute name with a prefi x is known as a **Qualified Name**, often abbreviated to **QName**. The part after the prefi x is technically known as a **Local Name**

9. What is the drawback attribute? (OR) compare attribute with child element?

- attributes cannot contain multiple values (child elements can)
- attributes are not easily expandable (for future changes)
- attributes cannot describe structures (child elements can)
- attributes are more difficult to manipulate by program code

10.In XML, the element can have what types of content?

In, XML the elements can have 4 content types:

- **Text only** – elements consists its content as text

o **Example**: **<firstname>Kajendran</firstname>**

- **Element-only** – elements consists entirely of nested elements

o **Example**

```
<name>
    <firstname>Kajendran</firstname>
    <middlename/>
    <lastname>Krishnan</lastname>
</name>
```

- **Mixed content** – elements consist of combination of nested elements and text

```
<address>
    <name>
        <firstname>Kajendran</firstname>
        <lastname>Krishnan</lastname>
```

</name>
11/20, valluvar street Saidapet west
</address>

- **Empty content**
  - **Example:<name firstname=”ram” lastname=”kumar” />**

11. What is the benefits of an XML Editor than text editor?

- An XML editor will help prevent you from making errors, while typing xml file
- XML editors are similar to HTML editors (or any other programming editor) in that they provide syntax highlighting which helps with readability when you're coding.
- They'll automatically insert a closing tag once you're added an opening tag
- A good XML editor should also provide validation - a way for you to validate that the documents you create are well formed.

12. List some examples of XML Editors?

- **XML Notepad**
- **XML Cooktop**
- **XML Pro**
- **XML Spy**
- **Liquid XML Studio**

13. What is the difference between TreeWalker and NodeIterator interfaces?

The TreeWalker interface provides many of the same benefits as NodeIterator. The main difference is that

NodeIterator	TreeWalker
NodeIterator presents a list-oriented view	TreeWalker presents a tree-oriented view
It allows you to move forward and backward	In addition to forward and backward movement, TreeWalker also allows you to - move to the **parent of a node** - **move** to one of its children - **move** to a sibling

14. What is the difference between JDOM and DOM interfaces?

DOM	JDOM
DOM is an interface-based API	JDOM is a class-based API
No classes available in DOM specification, only interfaces are given	There are classes that encapsulate documents, elements, attributes, text, and so on. This simplifies usage b minimizing downcasts
DOM is a strict hierarchy based on a node, which leads to lots of downcasts. Downcasts add complexity to source code and also reduce performance	No downcast involved, the leads to improved performance

DOM parse XML by itself	JDOM does not parse XML by itself; rather, it can build JDOM objects from a DOM tree or a SAX parser

15. What is the drawback of SAX parser?

Drawback of SAX:

- SAX parsing is "single pass," so you can't back up to an earlier part of the document any more than you can back up from a serial data stream.
- You have no random access at all. Handling parent/child relationships can be more challenging
- SAX is read only parser, they do not provide ability to manipulate a document or its structure
- There is no formal specification for SAX

16. What is the advantage SAX parser?

- SAX is an event-based API. It provides a framework for defining event listener or handler
- SAX is best suitable for very large xml document, because instead of loading an entire document into memory all at once, SAX parsers read documents and notify a client program when **elements, text, comments**, and **other data of interest are found**

17. Why do I need SAX?

- SAX is completely free, so it can be embedded in a larger application without royalty fees or even copyright notices SAX is an event-based API.
- Some SAX parsers can validate a document against a Document Type Definition (DTD). Validating parsers can also tell you specifically where validation has failed

18. List out some SAX parsers?

- The most commonly used parsers are
    - **Xerces** from the Apache XML project
    - **Java API for XML Processing (JAXP)** from Sun Microsystems

19. List out some DOM parsers?

A number of DOM implementations are available for Java. Two of the most popular are

- **Java APIs for XML Processing (JAXP)**, developed by Sun Microsystems
- **Xerces** developed as part of the Apache XML project
    - *JDOM*
- **List out some DOM parsers?**

20. List out small DOM-like parsers for hand-held devices or PDAs?

- NanoXML - TinyXML - kXML

ABOUT AUTHORS

John T Mesia Dhas received his Ph.D. in Computer Science and Engineering from Vel Tech University, Chennai, India. He has 16 years of Experience in the field of Education and Industry, currently he is working as Associate Professor with Computer Science and Engineering Department of T John Institute of Technology, Bangalore under VTU, India.

He is also doing researches in Software Engineering and Data Science fields. He has published more than 25 research articles in conferences and Journals.

T. S. Shiny Angel received her Ph.D. in Computer Science and Engineering from SRM University, Chennai, India. She has 19 years of Experience in the field of Education and Industry, currently she is working as Assistant Professor Sr. Grade with Software Engineering Department of SRM Institute of Science and Technology (formerly known as SRM University), Chennai, Tamil Nadu, India.

She is also doing researches in Software Engineering, Machine Learning and Data Analytics fields. She has published more than 45 research papers in conferences and Journals.

ISBN: 978-93-5493-336-3

Price: Rs. 450/-

OTHER BOOKS

S. No	Title	ISBN
1	C LOGIC PROGRAMMING	978-93-5416-366-1
2	MODERN METRICS (MM): THE FUNCTIONAL SIZE ESTIMATOR FOR MODERN SOFTWARE	978-93-5408-510-9
3	PYTHON 3.7.1 Vol - I	978-93-5416-045-5
4	SOFTWARE SIZING APPROACHES	978-93-5437-820-1
5	DBMS PRACTICAL PROGRAMS	978-93-5437-572-9
6	SERVICE ORIENTED ARCHITECTURE	978-93-5416-496-5
7	ANDROID APPLICATIONS DEVELOPMENT PRACTICAL APPROACH	978-93-5445-403-5
8	MOBILE APPLICATIONS DEVELOPMENT	978-93-5445-406-6
9	XML HAND BOOK	978-93-5493-336-3
10	PARALLEL COMPUTING IN ENGINEERING APPLICATIONS	978-93-5578-655-5

For free E-Books: jtmdhasres@gmail.com

ISBN: 978-93-5493-336-3

Price: Rs. 450/-