XML Simplified Learner's Guide

Onlinevarsity
your e-way to learning

XML Simplified
Learner's Guide

APTECH LIMITED
Contact E-mail: ov-support@onlinevarsity.com

First Edition - 2013

Dear Learner,

We congratulate you on your decision to pursue an Aptech course.

Aptech Ltd. designs its courses using a sound instructional design model – from conceptualization to execution, incorporating the following key aspects:

- Scanning the user system and needs assessment

Needs assessment is carried out to find the educational and training needs of the learner

Technology trends are regularly scanned and tracked by core teams at Aptech Ltd. TAG* analyzes these on a monthly basis to understand the emerging technology training needs for the Industry.

An annual Industry Recruitment Profile Survey# is conducted during August - October to understand the technologies that Industries would be adapting in the next 2 to 3 years. An analysis of these trends & recruitment needs is then carried out to understand the skill requirements for different roles & career opportunities.

The skill requirements are then mapped with the learner profile (user system) to derive the Learning objectives for the different roles.

- Needs analysis and design of curriculum

The Learning objectives are then analyzed and translated into learning tasks. Each learning task or activity is analyzed in terms of knowledge, skills and attitudes that are required to perform that task. Teachers and domain experts do this jointly. These are then grouped in clusters to form the subjects to be covered by the curriculum.

In addition, the society, the teachers, and the industry expect certain knowledge and skills that are related to abilities such as *learning-to-learn, thinking, adaptability, problem solving, positive attitude etc*. These competencies would cover both cognitive and affective domains.

A precedence diagram for the subjects is drawn where the prerequisites for each subject are graphically illustrated. The number of levels in this diagram is determined by the duration of the course in terms of number of semesters etc. Using the precedence diagram and the time duration for each subject, the curriculum is organized.

- Design & development of instructional materials

The content outlines are developed by including additional topics that are required for the completion of the domain and for the logical development of the competencies identified. Evaluation strategy and scheme is developed for the subject. The topics are arranged/organized in a meaningful sequence.

The detailed instructional material – Training aids, Learner material, reference material, project guidelines, etc.- are then developed. Rigorous quality checks are conducted at every stage.

- Strategies for delivery of instruction

  Careful consideration is given for the integral development of abilities like thinking, problem solving, learning-to-learn etc. by selecting appropriate instructional strategies (training methodology), instructional activities and instructional materials.

  The area of IT is fast changing and nebulous. Hence considerable flexibility is provided in the instructional process by specially including creative activities with group interaction between the students and the trainer. The positive aspects of web based learning –acquiring information, organizing information and acting on the basis of insufficient information are some of the aspects, which are incorporated, in the instructional process.

- Assessment of learning

  The learning is assessed through different modes – tests, assignments & projects. The assessment system is designed to evaluate the level of knowledge & skills as defined by the learning objectives.

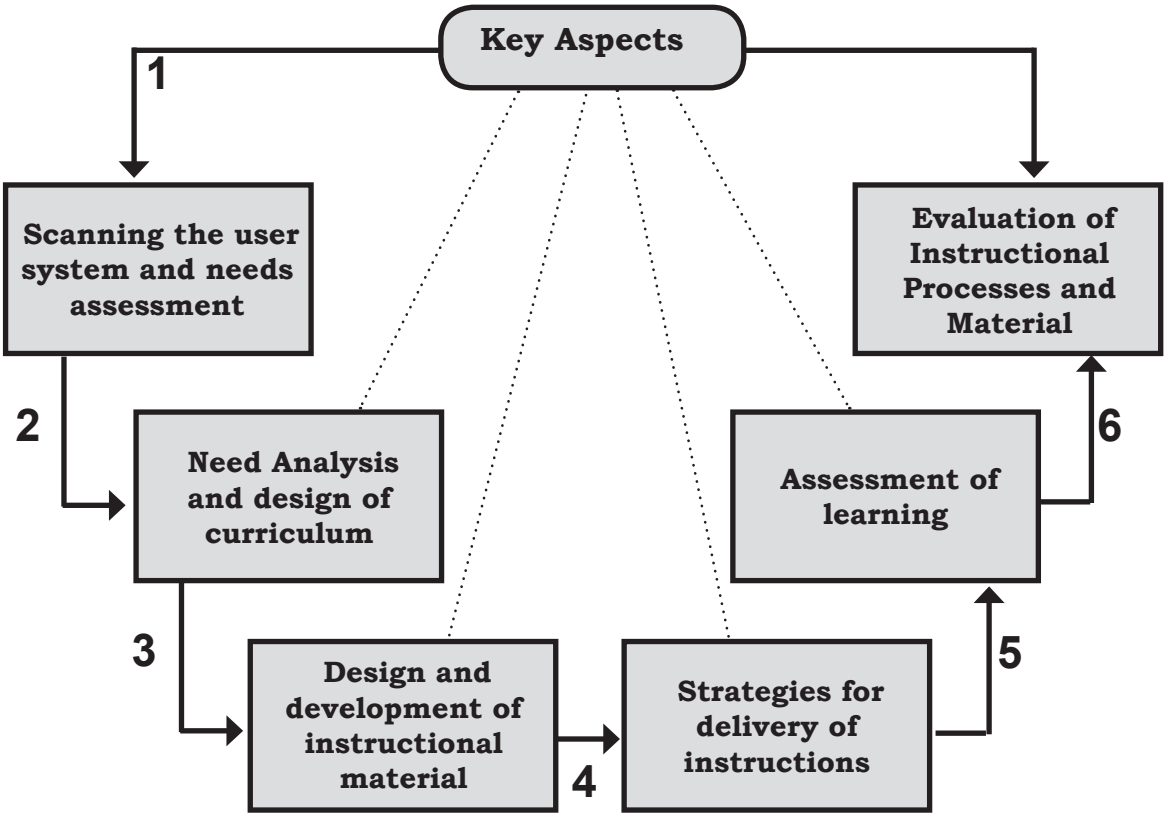
- Evaluation of instructional process and instructional materials

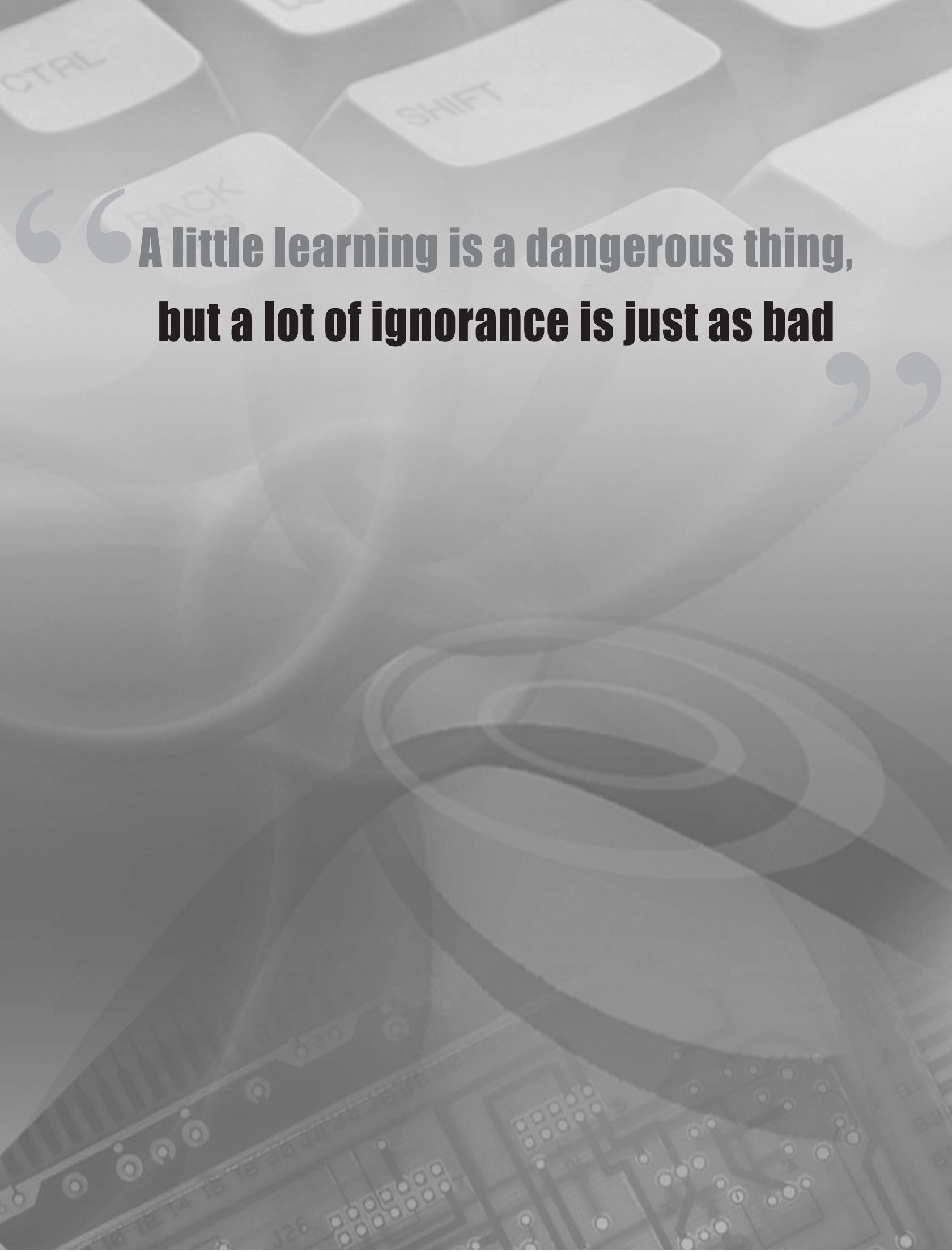
  The instructional process is backed by an elaborate monitoring system to evaluate - on-time delivery, understanding of a subject module, ability of the instructor to impart learning. As an integral part of this process, we request you to kindly send us your feedback in the reply pre-paid form appended at the end of each module.

*TAG – Technology & Academics Group comprises members from Aptech Ltd., professors from reputed Academic Institutions, Senior Managers from Industry, Technical gurus from Software Majors & representatives from regulatory organizations/forums.

Technology heads of Aptech Ltd. meet on a monthly basis to share and evaluate the technology trends. The group interfaces with the representatives of the TAG thrice a year to review and validate the technology and academic directions and endeavors of Aptech Ltd.

Aptech New Products Design Model




“A little learning is a dangerous thing,
but a lot of ignorance is just as bad”

Preface

In this book, **XML Simplified**, students will learn the fundamentals of XML. This book is an introduction to XML that prepares students with a strong foundation in one of the key elements of Web programming, that is, XML. The book describes and explains various features and concepts of XML.

This book is the result of a concentrated effort of the Design Team, which is continuously striving to bring you the best and the latest in Information Technology. The process of design has been a part of the ISO 9001 certification for Aptech-IT Division, Education Support Services. As part of Aptech's quality drive, this team does intensive research and curriculum enrichment to keep it in line with industry trends.

We will be glad to receive your suggestions.

Design Team


“Knowing is not enough
we must apply;
Willing is not enough,
we must do”

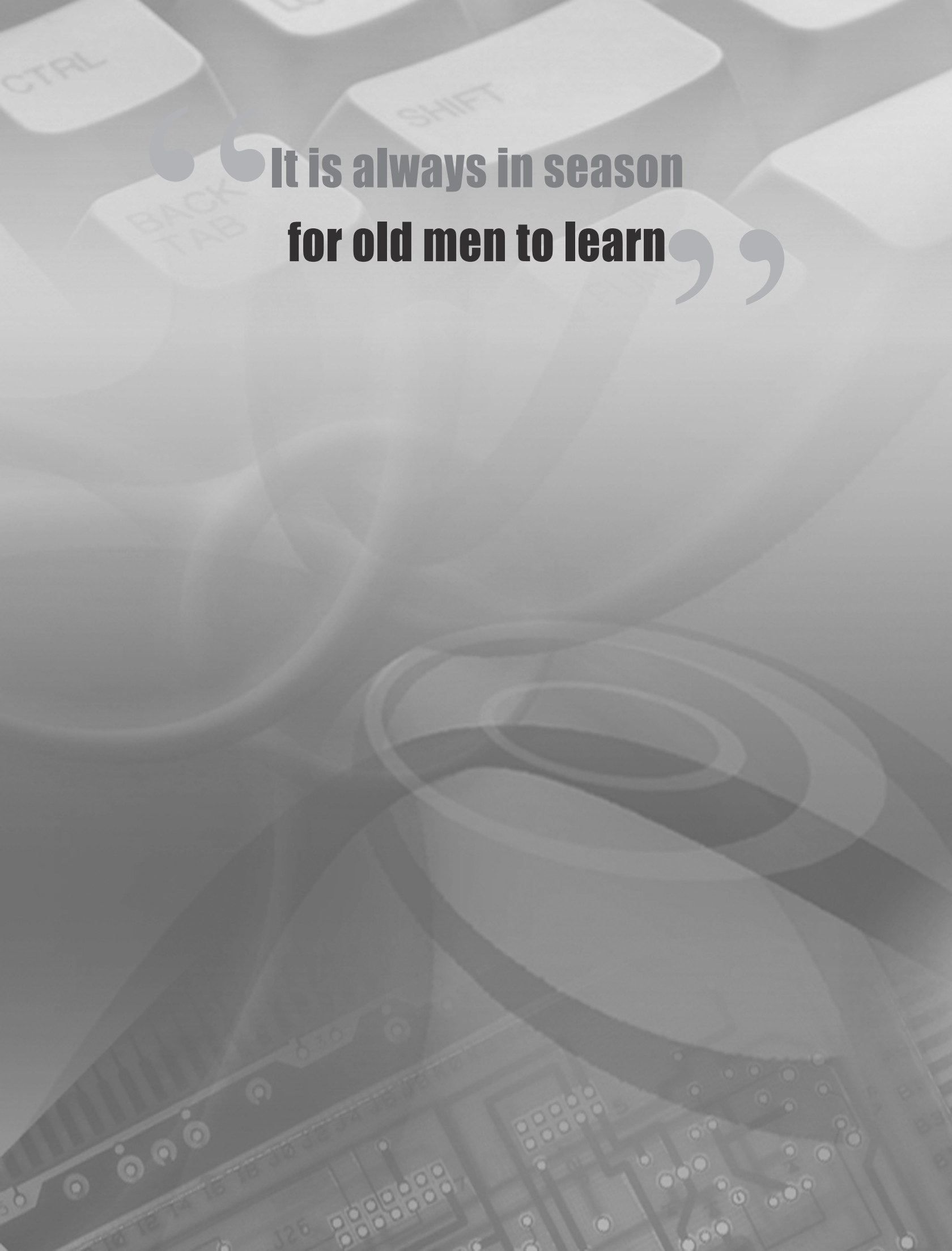
Aptech™
COMPUTER EDUCATION
Unleash your potential

Table of Contents

Sessions

1. Introduction to XML 1
2. Namespaces 39
3. DTDs 51
4. XML Schema 81
5. Style Sheets 125
6. XSL and XSLT 159
7. More on XSLT 213

Answers to Knowledge Checks i



“It is always in season
for old men to learn”

Module

1 Introduction to XML

Concepts

Module Overview

Welcome to the module, **Introduction to XML**. The module describes drawbacks of earlier markup languages that led to the development of XML. The module also explains the structure and lifecycle of the XML document. This module covers more on the XML syntax and the various parts of the XML document.

In this module, you will learn about:

- Introduction to XML
- Exploring XML
- Working with XML
- XML Syntax

1.1 Introduction to XML

In this first lesson, **Introduction to XML**, you will learn to:

- Outline the features of markup languages and list their drawbacks.
- Define and describe XML.
- State the benefits and scope of XML.

1.1.1 Features of Markup Languages

Early electronic formats like troff and TEX, were more concerned about the presentation of the content rather than with document structure.

Markup languages are used to define the meaning and organize the structure of a text document. Markup languages help the documents by giving them a new look and formatting the text.

Markup languages are categorized into following categories:

- **Presentational Markup**: Presentational markup language focuses on the structure of the document.
- **Procedural Markup**: Procedural markup is similar to presentation markup but in the former, the user will be able to edit the text file. Here, the user is helped by the software to arrange the text. These markup languages are used in professional publishing organizations.
- **Descriptive Markup**: Descriptive markup is also known as semantic markup. This kind of markup determines the content of the document.

Two types of markup languages are popular in recent times. They are as follows:

- **Generalized Markup Languages**: These type of languages describe the structure and meaning of the text in a document.
- **Specific Markup Languages**: These type of markup languages are used to generate application specific code.

Generalized Markup Language (GML), a project by IBM, helped the documents to be edited, formatted, and searched by different programs using its content-based tags. Generalized Markup Language (GML) was developed to accomplish the following:

- The markup should describe only the structure of the document but not its flair.
- The syntax of the markup language should be strictly followed so that the code can clearly be read by a software program or by a human being.

In 1980, ANSI Committee created Standard Generalized Markup Language (SGML), an all-encompassing coding scheme and flexible toolkit for developing specialized markup languages. Standard Generalized Markup language (SGML) is the successor to GML. In 1986, International Organization for Standardization (ISO) acquired it as a standard. SGML is a meta language as other languages are created from it. SGML has a syntax to include markup in documents. SGML also has a syntax to describe what tags are allowed in different locations. SGML application consists of SGML declaration and SGML Document Type Definition (DTD).

In 1989, Hyper Text Markup Language (HTML), a technology for sharing information by using hyperlinked text documents was developed. HTML (Hyper Text Markup Language) was created from SGML. In the early years, it was extensively accessed by scientists and technicians. HTML was originally created to mark up technical papers, so that they could be transferred across different platforms. However, with time, it began to be used for marking up non-technical documents too. As the use of the Internet became popular, browser manufacturers started developing different tags to display documents with more creativity. But, this created problems for implementation in different browsers with the increase in the number of tags used.

Figure 1.1 depicts the evolution of markup languages.

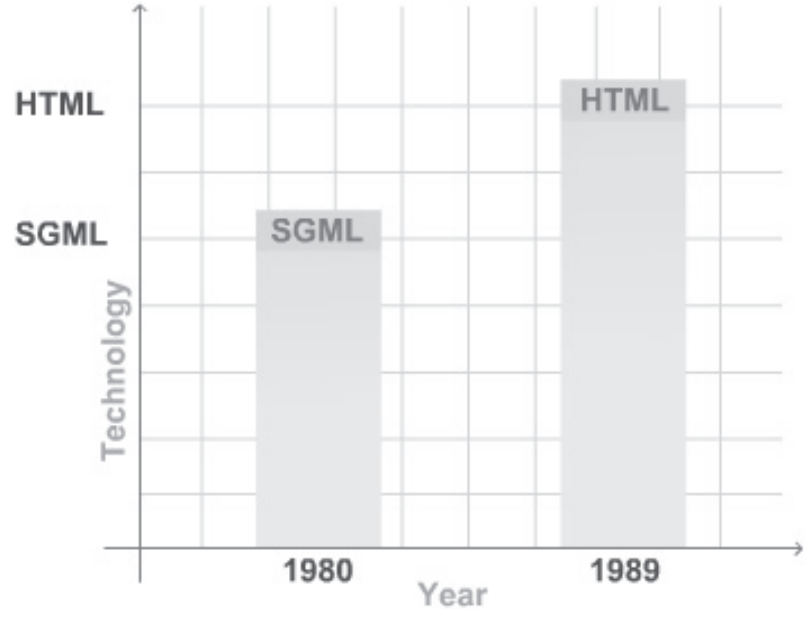


Figure 1.1: Evolution of Markup Languages

The features and drawbacks of earlier markup languages are:

- **Features**

  1. GML describes the document in terms of its format, structure, and other properties.
  2. SGML ensures that the system can represent the data in its own way.
  3. HTML used ASCII text, which allows the user to use any text editor.

- **Drawbacks**

  1. GML and SGML were not suited for data interchange over the Web.
  2. HTML possesses instructions on how to display the content rather than the content they encompass.

Concepts

1.1.2 Evolution of XML

In order to address the issues raised by earlier markup languages, the Extensible Markup Language (XML) was created. XML is a W3C recommendation.

XML is a set of rules for defining semantic tags that break a document into parts and identify the different parts of the document. XML was developed over HTML because of the basic differences between them given in table 1.1.

HTML	XML
HTML was designed to display data.	XML was designed to carry data.
HTML displays data and focuses on how data looks.	XML describes data and focuses on what data is.
HTML displays information.	XML describes information.

Table 1.1: Differences between HTML and XML

Figure 1.2 shows an XML code.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
- <Watch>
    <name>Titan</name>
    <price>$50</price>
    <description>A brown diamond studded watch</description>
  </Watch>
```

Figure 1.2: An XML Code

1.1.3 Features of XML

Features of XML are as follows:

- XML stands for Extensible Markup Language
- XML is a markup language much like HTML
- XML was designed to describe data
- XML tags are not predefined. You must define your own tags
- XML uses a DTD or an XML Schema to describe the data
- XML with a DTD or XML Schema is designed to be self-descriptive

1.1.4 XML Markup

XML markup defines the physical and logical layout of the document. XML can be considered as an information container. It contains shapes, labels, structures and also protects information. XML employs a tree-based structure to represent a document. The basic foundation of XML is laid down by symbols embedded in the text known as markup. The markup combines the text and extra information about the text like its structure and presentation. The markup divides the information into a hierarchy of character data and container-like elements and its attributes. A number of software programs process electronic documents use a markup.

The underlying unit of XML is a character. The combination of characters is known as an entity. These entities are either present in the entity declaration or in a text file stored externally. All the characters are grouped together to form an XML document.

XML's markup divides a document into separate information containers called elements. A document consists of one outermost element called root element that contains all the other elements, plus some optional administrative information at the top, known as XML declaration. The following code demonstrates the elements.

Code Snippet:

```
<?xml version="1.0" encoding="iso-8859-1" ?>
-  <FlowerPlanet>
  <Name>Rose</Name>
  <Price>$1</Price>
  <Description>Red in color</Description>
  <Number>700</Number>
</FlowerPlanet>
```

where,

`<Name>`, `<Price>`, `<Description>` and `<Number>` inside the tags are elements.

`<FlowerPlanet>` and `</FlowerPlanet>` are the root elements.

The usage of XML can be observed in many real-life scenarios. It can be used in the fields of information sharing, single application usage, content delivery, re-use of data, separation of data and presentation, semantics, and so forth. News agencies are a common place where XML is used. News producers and news consumers often use a standard specification named XMLNews to produce, retrieve, and relay information across different systems in the world.

Note: XML is a subset of SGML, with the same goals, but with as much of the complexity eliminated as possible. This means that any document which follows XML's syntax rules will also follow SGML's syntax rules, and can therefore be read by existing SGML tools.

Concepts

1.1.5 Advantages and Disadvantages of XML

The benefits of XML are as follows:

- **Data independence**

  Data independence is the essential characteristic of XML. It separates the content from its presentation. Since an XML document describes data, it can be processed by any application.

- **Easier to parse**

  The absence of formatting instructions makes it easy to parse. This makes XML an ideal framework for data exchange.

- **Reducing Server Load**

  Because XML's semantic and structural information enables it to be manipulated by any application, much of the processing that was once earlier limited to servers can now be performed by clients. This reduces server load and network traffic, resulting in a faster, more efficient Web.

- **Easier to create**

  It is text-based, so it is easy to create an XML document with even the most primitive text processing tools. However, XML also can describe images, vector graphics, animation or any other data type to which it is extended.

- **Web Site Content**

  The W3C uses XML to write its specifications and transforms it to a number of other presentation formats. Some Web sites also use XML for their content and get it transformed to HTML using XSLT and CSS and display the content directly in browsers.

- **Remote Procedure Calls**

  XML is also used as a protocol for Remote Procedure Calls (RPC). RPC is a protocol that allows objects on one computer to call objects on another computer to do work, allowing distributed computing.

- **e-Commerce**

  XML can be used as an exchange format in order to send data from one company to another.

Note: The process of manipulating an XML document is called as XML Parsing. The parser loads the document into the memory. After the document is loaded into the memory, Document Object Model (DOM) manipulates the data.

Even though XML has many advantages, it has a few disadvantages too. Some of the disadvantages are as follows:

- Usage of XML leads to increase in data size and processing time. Since XML uses Unicode encoding set for characters, it also consumes more memory.
- XML lacks adequate amount of processing instructions. If the process of translation is not used, then the developers globally are forced to prepare their own processing instructions to display XML in the required form.
- XML is more demanding and difficult as compared to HTML.
- XML is verbose and wordy as the tags used in it are predefined.
- Versions of Internet Explorer (IE) earlier than 5.0 do not support XML.

Knowledge Check 1

1. Which of the following statements are true and which are false in the case of XML?

(A)	XML was designed to describe data.
(B)	XML tags are predefined.
(C)	XML consists of rules to identify and define different parts of the document.
(D)	XML offers a standard way to add markup to documents.
(E)	XML forms the basis to create languages like WAP and WML.

2. Which of the statements about XML are true and which of the statements are false?

(A)	XML describes its data along with its presentation.
(B)	Client reduces the server load by sending large amount of information in one XML document to the server.
(C)	XML uses only XSLT to be transformed to HTML.
(D)	XML can be implemented as middle-tier for client server architectures.
(E)	XML allows data exchange as it has no formatting instructions.

Concepts

1.2 Exploring XML

In this second lesson, **Exploring XML**, you will learn to:

- Describe the structure of an XML document.
- Explain the lifecycle of an XML document.
- State the functions of editors for XML and list the popularly used editors.
- State the functions of parsers for XML and list names of commonly used parsers.
- State the functions of browsers for XML and list the commonly used browsers.

1.2.1 XML Document Structure

XML documents are commonly stored in text files with extension `.xml`. The two sections of an XML document are:

- **Document Prolog**: XML parser gets information about the content in the document with the help of document prolog. Document prolog contains metadata and consists of two parts - XML Declaration and Document Type Declaration. XML Declaration specifies the version of XML being used. Document Type Declaration defines entities' or attributes' values and checks grammar and vocabulary of markup.
- **Root Element**: The second is an element called the root element. The root element is also called a document element. It must contain all the other elements and content in the document. An XML element has a start tag and end tag.

Following are some of the relationships:

- **Parent**: It is an element that contains other elements.
- **Child**: It is an element that is present within another element.
- **Sibling**: They are elements which have the same parent element.
- **Nesting**: It is a process where an element contains other elements.

An XML document consists of a set of unambiguously named "entities". Every XML document starts with a "root" or document entity. All other entities are optional. Entities are aliases for more complex functions. A single entity name can represent a large amount of text. The alias name is used each time some text is referenced and the processor expands the contents of the alias.

Figure 1.3 shows the XML document structure.

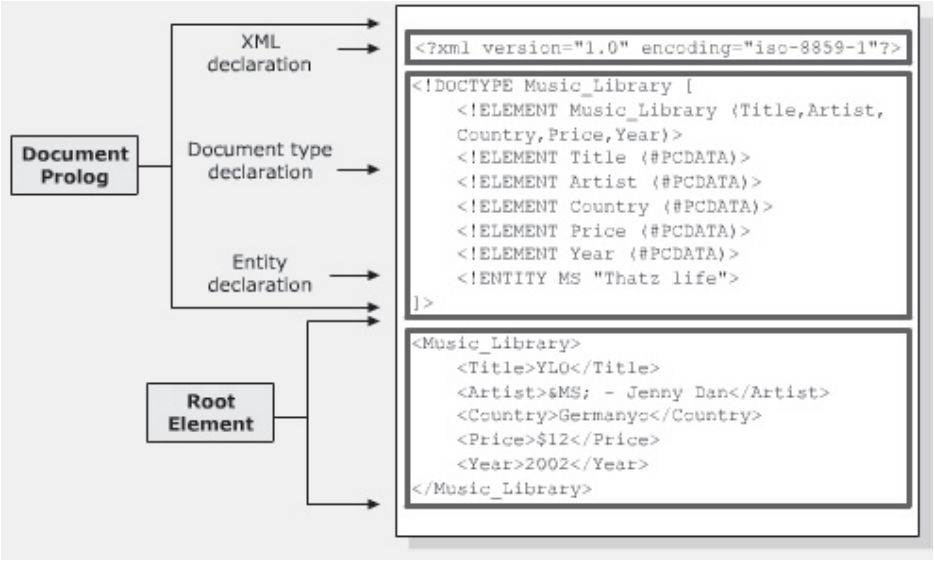


Figure 1.3: XML Document Structure

The following code demonstrates the XML document structure.

Code Snippet:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE Music_Library [
     <!ELEMENT Music_Library (Title,Artist,Country,Price,Year)>
     <!ELEMENT Title (#PCDATA)>
     <!ELEMENT Artist (#PCDATA)>
     <!ELEMENT Country (#PCDATA)>
     <!ELEMENT Price (#PCDATA)>
     <!ELEMENT Year (#PCDATA)>
     <!ENTITY MS "Thatz life">
]>
<Music_Library>
    <Title>YLO</Title>
    <Artist>&MS; - Jenny Dan</Artist>
    <Country>Germanyo</Country>
    <Price>$12</Price>
    <Year>2002</Year>
</Music_Library>
```

where,

The first block indicates xml declaration and document type declaration. `Music_Library` is the root element.

1.2.2 Logical Structure

The logical structure of an XML document gives information about the elements and the order in which they are to be included in the document. It shows how a document is constructed rather than what it contains.

Document Prolog forms the basis of the logical structure of the XML document. XML Declaration and Document Type Definition are its two basic and optional components as shown in figure 1.4.

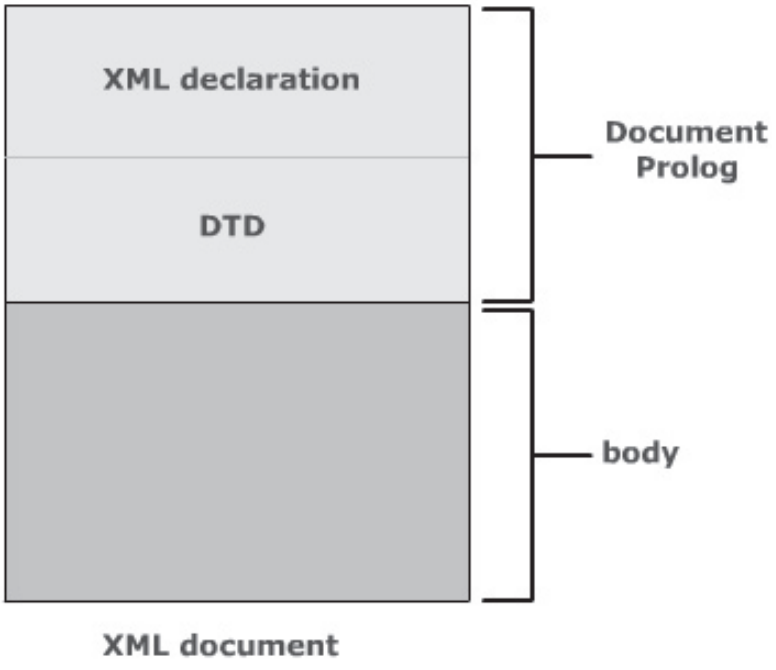


Figure 1.4: Logical Structure of XML Document

The following code demonstrates the XML declaration.

Code Snippet:

```
<?xml version="1.0" encoding="iso-8859-1" ?>
```

XML Declaration gives the version of the XML specification and also identifies the character encoding scheme.

The following code demonstrates the use of a DTD.

Code Snippet:

```
<!DOCTYPE Music_Library SYSTEM "Mlibrary.dtd">
```

A document type definition defines a set of rules to which the XML document conforms. A DTD can be either external or internal. In this case, the DTD is an external file.

1.2.3 XML Document Life Cycle

An XML editor refers to the DTD and creates an XML document. After the document is created, the document is scanned for elements and attributes in it. This stage is known as scanning. The XML parser builds a complete data structure after parsing. The data is then extracted from elements and attributes of the document. This stage is known as access. It is then converted into the application program. The document structure can also be modified during the process by inserting or deleting elements or by changing textual content of element or attribute. This stage is known as modification. The data is then serialized to a textual form and is passed to a browser or any other application that can display it. This stage is known as serialization.

Figure 1.5 displays the XML document life cycle.

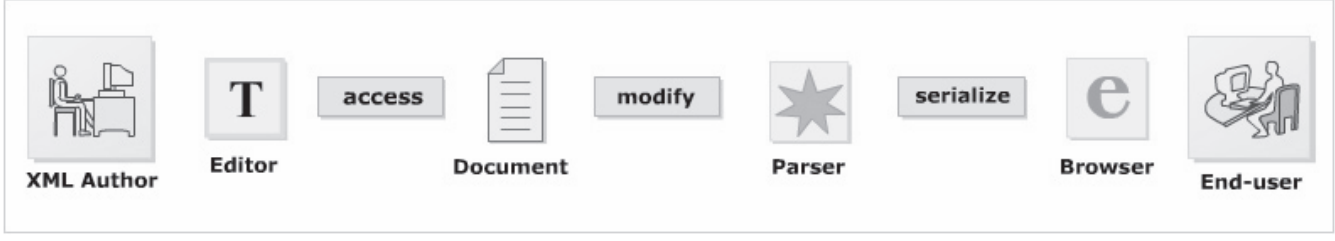


Figure 1.5: XML Document Life Cycle

Note: XML parser creates, manipulates and updates the XML document. It will read the list of records in the XML document and the handler stores it in the form of a data structure. The handler then processes it and displays it in HTML.

1.2.4 Editors

An XML Editor is used to create and edit XML documents. Any application can be used as an editor in XML. Since all XML documents are text-based markup languages, a standard Windows Notepad or Wordpad can also be used. However, for various reasons, Notepad should not be used for writing professional XML. Notepad does not know that the text written in it is XML code, and thus, can create problem. For an XML document to be error-free and possess XML-specific features, like the ability to edit elements and attributes, a professional XML editor should be used.

The main functions that editors provide are as follows:

- Add opening and closing tags to the code
- Check for validity of XML
- Verify XML against a DTD/Schema
- Perform series of transforms over a document
- Color the XML syntax
- Display the line numbers
- Present the content and hide the code
- Complete the word

The popularly used editors are:

- XMLwriter
- XML Spy
- XML Pro
- XMLmind
- XMetal

1.2.5 Parsers

An XML parser/XML processor reads the document and verifies it for its well-formedness.

An XML parser keeps the entire representation of XML data in memory with the help of Document Object Model (DOM). The in-built parser used in IE 5.0 is also known as Microsoft XML Parser (MSXML). It is a component, which is available once IE 5.0 is installed.

Microsoft's XML parser goes through the entire data structure and accesses the values of the attributes in the elements. The parser also creates or deletes the elements and converts the tree structure into XML.

MSXML supports some of the COM objects for backward interoperability.

Some of them are XSLProcessor objects, XSLTemplate objects, XMLDOMSelection objects, XMLSchemaCache objects, and XMLDOMDocument2 objects.

MSXML 3.0 supports XSL transformations, which contain data manipulation operations. The XML parsers ignore the white space by default, but if the default value of Boolean preserveWhiteSpace property of DOMDocument object is true, the XML parsers preserve the white space. Using Microsoft Data type Schema, MSXML 3.0 specifies a data type for the element or the attribute. MSXML also improves the performance of the applications with the help of different caching features.

XML parsing in Mozilla performs functions like going through the elements, accessing their values, and so on. Using JavaScript, an instance of XML parser can be created in Mozilla browsers. The XML parsing in Firefox automatically parses the data into Document Object Model (DOM). Opera uses only those XML parsers that do not validate DTDs by default.

Speed and performance are the criteria against which XML parsers are selected.

Commonly used parsers are:

- Crimson
- Xerces
- Oracle XML Parser
- JAXP(Java API for XML)
- MSXML

The two types of parsers are:

- **Non Validating parser**
  - It checks the well-formedness of the document.
  - Reads the document and checks for its conformity with XML standards.
- **Validating parser**
  - It checks the validity of the document using DTD.

Concepts

1.2.6 Browsers

After the XML document is read, the parser passes the data structure to the client application. The application can be a Web browser. The browser then formats the data and displays it to the user. Other programs like database, MIDI program or a spreadsheet program may also receive the data and present it accordingly.

The final output of XML data is viewed in a browser. XML is not supported by all browsers, for example, Netscape Navigator 4.0 does not support XML but later versions of the browser like Netscape 6.0 do support it. Only browsers like IE 5.0 or greater give full support for XML specifications. In IE, XML can be directly viewed using style sheets. It gives support to namespaces and handles a mechanism known as data islands where XML is embedded into HTML.

Mozilla 5.0 uses an interface to the XML DOM (Document Object Model) via JavaScript and plug-ins. It also supports elements from the HTML namespace.

Commonly used Web browsers are as follows:

- Netscape
- Mozilla
- Internet Explorer
- Firefox
- Opera

Knowledge Check 2

1. Which of the statements about the structure of XML documents are true and which statements are false?

(A)	XML documents are stored with .xml extension.
(B)	Document prolog can consist of version declaration, DTD comments and processing instructions.
(C)	XML declaration informs the processing agent about the version of XML being used.
(D)	Root element must not be a nonempty tag.
(E)	The logical structure gives information about the elements and the order in which they are to be included in the document.

2. Which of the characteristics are true or false accordingly when an XML document is created by an XML editor?

(A)	XML syntax is colored.
(B)	XML is not validated.
(C)	XML parser reads the document after it is created.
(D)	XML is transformed only via XSLT.
(E)	XML document is edited by keeping DTD in mind.

3. Which of the statements about XML browsers and parsers are true and which statements are false?

(A)	XML parser is calculated against its speed and performance.
(B)	XML parser checks for validity and well-formedness.
(C)	XML parser does its work after the processor converts the document into a data structure.
(D)	Browser displays the content directly after the parser passes the data.
(E)	Crimson and Xerces are some of the browsers.

1.3 Working with XML

In this third lesson, **Working with XML**, you will learn to:

- Explain the steps towards building an XML document.
- Define what is meant by well-formed XML.

1.3.1 Creating an XML Document

An XML document has three main components:

1. Tags (markup) and text (content)
2. DTD or Schema
3. Formatting or display specifications

The steps to build an XML document are as follows:

- **Create an XML document in an editor**

  To create a simple XML document, you type XML code in any text or XML editor as demonstrated in the following code.

  Code Snippet:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<Soft_drink>
  <Name>Refresh</Name>
  <Price>$1</Price>
</Soft_drink>
```

- **Saving the XML file**

  After the typing is done, you save the contents of the file. To save the file, you click the File menu, click the "Save" option and provide the file name such as, for example, `SoftDrink.xml`. The extension `.xml` is compulsory if you are typing the code in Notepad or any kind of word processor such as Microsoft Word.

- **Load XML document in a browser**

  After the document is saved, you can open the file directly in a browser that supports XML. A common browser used is Internet Explorer 5.0 or later.

1.3.2 Exploring the XML Document

The various building blocks of an XML document are as follows:

- XML Version Declaration
- Document Type Definition
- Document instance in which the content is defined by the markup

Figure 1.6 depicts a XML document.

```
1  <?xml version="1.0" encoding="iso-8859-1" standalone="yes"?>
2
3  <!DOCTYPE student [
4   <!ELEMENT student (name,dob,bloodgroup,rollnumber)>
5   <!ELEMENT name (#PCDATA)>
6   <!ELEMENT dob (#PCDATA)>
7   <!ELEMENT bloodgroup (#PCDATA)>
8   <!ELEMENT rollnumber (#PCDATA)>
9  ]>
10
11 <student>
12      <name>James</name>
13      <dob>26th September, 1983</dob>
14      <bloodgroup>A positive</bloodgroup>
15      <rollnumber>17</rollnumber>
16 </student>
```

Figure 1.6: XML Document

Explanation for highlighted areas in the code:

`<?xml`

The XML declaration should start with the five character string, `<?xml`. It indicates that the document is an XML document.

`version="1.0"`

The XML declaration tells the version of XML, the type of character encoding being used and the markup declaration used in the XML document.

`encoding="iso-8859-1"`

Characters are encoded using various encoding formats. The character encoding is declared in encoding declaration.

`standalone="yes"`

The standalone declaration indicates the presence of external markup declarations. "Yes" indicates that there are no external markup declarations and "no" indicates that external markup declarations might exist.

The Document Type Declaration declares and defines the elements used in the document class. This is a DTD used internally as demonstrated in the following code:

```
<!DOCTYPE Student [
<!ELEMENT Student (Name,Dob,BloodGroup,RollNumber)>
<!ELEMENT Name (#PCDATA)>
<!ELEMENT Dob (#PCDATA)>
<!ELEMENT BloodGroup (#PCDATA)>
<!ELEMENT RollNumber (#PCDATA)>
]>
```

If the same DTD is used externally, then the code is:

```
<!DOCTYPE student SYSTEM "studatabase.dtd">
<Student>
    <Name>James</Name>
    <Dob>26th September, 1983</Dob>
    <BloodGroup>A positive</BloodGroup>
    <RollNumber>17</RollNumber>
</Student>
```

This part defines the content of the XML document called as markup. It describes the purpose and function of each element.

Note: The tags in XML are not predefined. XML allows the user to create the tags when needed.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<Message>
  Please Call
</Message>

<?xml version="1.0" encoding="iso-8859-1"?>
<P>
  Please Call
</P>
```

The two XML code snippets listed show the same output with different names. They are equal as they have the same structure and content.

1.3.3 Meaning in Markup

Markup can be divided into following three parts:

- **Structure**

  It describes the form of the document by specifying the relationship between different elements in the document. It emphasizes to specify a single non-empty, root element that contains other elements and the content.

- **Semantic**

  Semantics describes how each element is specified to the outside world of the document. For example, an HTML enabled Web browser assigns "paragraph" to the tags `<P>` and `</P>` but not to the tags `<Message>` and `</Message>`.

- **Style**

  It specifies how the content of the tag or element is displayed. It indicates whether the tag is bold, normal, and pink in color or with the font size 10.

1.3.4 Well-formed XML Document

Well-formedness refers to the standards that are to be followed by the XML documents. Well-formedness makes XML processors and browsers read XML documents. A document is well formed, if it fulfills the following rules:

- **Minimum of one element is required**

  Every well formed XML document should consist of a minimum of one element.

- **XML tags are case sensitive**

  All the tags used are case sensitive, that is `<URGENT>` is different from `<urgent>`.

- **Every start tag should end with end tag**

  All the tags in XML should begin with a start tag and a matching end tag. The end tags only have an additional forward slash as compared to their start tag.

- **XML tags should be nested properly**

  All the XML tags used should be nested properly. The following code demonstrates this property.

  Code Snippet:

```
<Book>
<Name>Good XML</Name>
<Cost>$20</Cost>
</Book>
```

- **XML tags should be valid**

  - Tags should begin with letter, an underscore (_), or a colon (:).
  - Tags should contain combination of letters, numbers, periods (.), colons, underscores, or hyphens (-).
  - Tags should not contain white space.
  - Tags should not start with reserved words like "xml".

- **Length of markup names**

  The length of the tags depends on the XML processors.

- **XML attributes should be valid**

  - Attributes should not be duplicated
  - Attributes are specified by a name and value pair which is delimited by equal (=) sign. The values are delimited by quotation marks. For example,

```
<DOLL NAME = "BARBIE" COLOR = "PINK">
```

  - Attributes should follow the same rules as followed by tags.

- **XML documents should be verified**

  To be read by the XML browsers, the document should be checked against the XML rules.

Figure 1.7 shows a well-formed XML document.

Figure 1.7: Well-formed XML Document

Concepts

Knowledge Check 3

1. Which one of the following code snippets will give an output as

```
"Hi,
The test is on Wednesday.
Regards
Sam.
12/2/2007."
```

(A)

```
<?xml version="1.0" encoding="iso-8859-1"?>
<Matter>
  <Wish> Hi, </Wish>
  <Body> The test is on Wednesday. </Body>
  <Close>
   Regards
   Sam.
  </Close>
  <Date>
  <Day>12</Day>
  <Month>2</Month>
  <Year>2007</Year>
  </Date>
</Matter>
```

(B)

```
<?xml version="1.0" encoding="iso-8859-1"?>
<Matter>
  <Wish> Hi, </Wish></Matter>
  <Body> The test is on Wednesday. </Body>
  <Close>
   Regards
   Sam.
  </Close>
<Date>
  <Day>12</Day>
  <Month>2</Month>
  <Year>2007</Year>
</Date>
```

(C)	<?xml version="1.0" encoding="iso-8859-1"?> <Matter> <Wish> Hi, </Wish> <Body> The test is on Wednesday. </Body> <Close> Regards Sam. </Close> <Date> <Day>12</Day> <Month>2</Month> <Year>2007</Year> <Date></Matter>
(D)	<?xml version="1.0" encoding="iso-8859-1"?> <Matter> Hi, <Body> The test is on Wednesday. </Body> <Close> Regards Sam. </Close> <Date> <Day>12</Day> <Month>2</Month> <Year>2007</Year> <Date></Matter>

2. Which of the following code snippets produce an output?

(A)	<?xml version="1.0" encoding="iso-8859-1"?> <xml> 9100203845902 Did you understand the concept? 2345823-23934 </xml>
(B)	<?xml version="1.0" encoding="iso-8859-1"?> <confirmation><question> Did you understand the concept?</question></confirmation> <sender> Sender : 9100203845902 <recipient> Recipient: 2345823-23934 <recipient>
(C)	<?xml version="1.0" encoding="iso-8859-1"?> <confirmation> <sender> Sender : 9100203845902 </sender> <question> Did you understand the concept?</question> <recipient> Recipient: 2345823239 </recipient> </confirmation>
(D)	<?xml version="1.0" encoding="iso-8859-1"?> Sender : 9100203845902 Recipient: 2345823239 <confirmation><question> Did you understand the concept?</ question></ confirmation>

1.4 XML Syntax

In this last lesson, **XML Syntax**, you will learn to:

- State and describe the use of comments and processing instructions in XML.
- Classify character data that is written between tags.
- Describe entities, `DOCTYPE` declarations and attributes.

1.4.1 Comments

XML comments are used for the people to give information about the code in the absence of the developer. It makes a document more readable. Comments are not restricted to document type definitions but may be placed anywhere in the document. Comments in XML are similar to those in HTML. Comments should be used only when needed, as they are not processed. Comments are used only for human consumption rather than machine consumption. Since the comments are not parsed, their presence or absence does not make any difference to the processors.

They are inserted into the XML document and are not part of the XML code. They can appear in the document prolog, DTD or in the textual content. These comments will not appear inside the tags or attribute values.

Comments start with the string `<!--` and end with the string `-->`. The parser believes that the comment has come to an end when it finds a `>` as shown in figure 1.8.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
- <Name NickName="John">
    <First>John</First>
    <!-- John is yet to pay the term fees  -->
    <Last>Brown</Last>
    <Semester>Final</Semester>
  </Name>
```

Figure 1.8: Comments in XML Document

Comments are written by following some rules.

➢ **Rules**

The rules that are to be followed while writing the comments are as follows:

- Comments should not include "-" or "—" as it might lead to confuse the XML parser.
- Comments should not be placed within a tag or entity declaration.
- Comments should not be placed before the XML declaration.
- Comments can be used to comment the tag sets.

  The following code demonstrates an example,

```
<!-- Highlight these
   <ChiefGuest1> King </ChiefGuest1>
   <Judge>Queen</Judge>
-->
```

- Comment should not be nested.

The following code demonstrates an example of comment,

Code Snippet:

```
<Name NickName='John'>
  <First>John</First>
  <!--John is yet to pay the term fees-->
  <Last>Brown</Last>
  <Semester>Final</Semester>
</Name>
```

1.4.2 Processing Instructions

Processing instructions are information which is application specific. These instructions do not follow XML rules or internal syntax. With the help of a parser these instructions are passed to the application. The application can either use the processing instructions or pass them on to another application.

The main objective of a processing instruction is to present some special instructions to the application.

All processing instructions must begin with `<?` and end with `?>`.

Though an XML declaration also begins with `<?` and end with `?>`, it is not considered as a processing instruction. It is because an XML declaration provides information only for the parsers and not for the application. In some cases, the application might need the information in the processing instruction only if it displays the output to the user.

Syntax:

```
<?PITarget <instruction>?>
```

where,

`PITarget` is the name of the application that should receive the processing instructions.

`<instruction>` is the instruction for the application.

The following code demonstrates an example of processing instruction.

Code Snippet:

```
<Name NickName='John'>
    <First>John</First>
    <!--John is yet to pay the term fees-->
    <Last>Brown</Last>
    <?feesprocessor SELECT fees FROM STUDENTFEES?>
    <Semester>Final</Semester>
</Name>
```

where,

`feesprocessor` is the name of the application that receives the processing instruction.

`SELECT fees FROM STUDENTFEES` is the instruction.

Concepts

1.4.3 Classification of Character Data

An XML document is divided into markup and character data.

Character data describes the document's actual content with the white space. The text in character data is not processed by the parser and thus, not treated as a regular text. The character data can be classified into:

- CDATA
- PCDATA

1.4.4 PCDATA

The data that is parsed by the parser is called as parsed character data (`PCDATA`). The `PCDATA` specifies that the element has parsed character data. It is used in the element declaration.

Escape character like "<" when used in the XML document will make the parser interpret it as a new element. As a result it will generate an error as shown in figure 1.9.

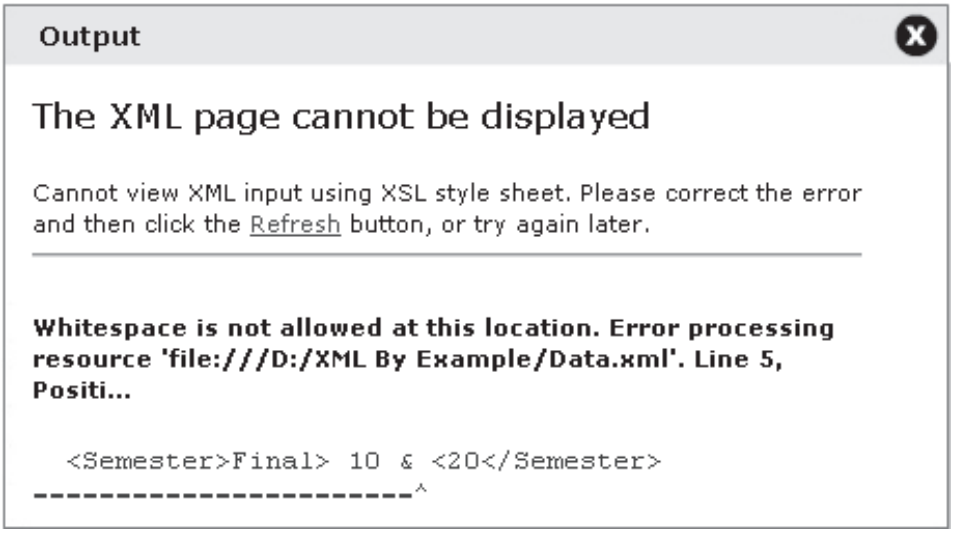


Figure 1.9: Error in Output

The following code demonstrates an example of PCDATA.

Code Snippet:

```
<Name nickname='John'>
  <First>John</First>
  <!--John is yet to pay the term fees-->
  <Last>Brown</Last>
  <Semester>Final> 10 & <20</Semester>
</Name>
```

1.4.5 CDATA

CDATA stands for character data that has reserved and white space characters in it. Though the text inside the CDATA is not parsed by the parser, it is commonly used for scripting code. XML parser ignores all tags and entity references inside the CDATA blocks. The CDATA block indicates to the parser that it is just a text but not a markup. A CDATA block always begins with the delimiter `<![CDATA[` and ends with the delimiter `]]>`. Since the ending delimiter marks the end of the CDATA block, the character string `]]/` is not allowed in the middle of the CDATA block. This will signal the end of the CDATA section.

The syntax of CDATA is as follows:

```
<![CDATA[ data ]]>
```

The following code demonstrates an example of CDATA.

Code Snippet:

```
<Sample>
<![CDATA[
    <Document>
    <Name>Core XML</Name>
    <Company>Aptech</Company>
    </Document>]]>
</Sample>
```

Concepts

1.4.6 Entities

The XML document is made up of large amount of information called as entities. Entities are used to avoid typing long pieces of text repeatedly, within the document. They can be categorized into the following:

- **Character entities**: They form the mechanism, which is used in place of a character's literal form. They provide the meaning of '>' when the symbol '>' is typed. Character entities can also be used with decimal or hexadecimal values with a condition that the numbers support Unicode coding. An XML processing program replaces character entities with their equivalent characters.

- **Content entities**: These entities are used to replace certain values. They are similar to text substitution macros in programming languages such as C. Content entity has the following syntax:

```
<!Entity name value>
```

- **Unparsed entities**: These entities when used, turn off the parsing process. They can be used to include multimedia content in the XML document.

Every entity consists a name and a value. The value ranges from a single character to a XML markup file. As the XML document is parsed, it checks for entity references. For every entity reference, the parser checks the memory to replace the entity reference with a text or markup.

An entity reference consists of an ampersand (&), the entity name, and a semicolon (;).

All the entities must be declared before they are used in the document. An entity can be declared either in a document prolog or in a DTD.

Some of the entities are defined in the system and are known as pre-defined entities. These entities are described in table 1.2.

Predefined Entity	Description	Output
&lt;	Produces the left angle bracket	<
&gt;	Produces the right angle bracket	>
&amp;	Produces the ampersand	&
&apos;	Produces a single quote character	‘
&quot;	Produces a double quote character	"

Table 1.2: Entities

The following code demonstrates an example of entities.

Code Snippet:

```
<?xml version="1.0"?>
<!DOCTYPE Letter [
<!ENTITY address "15 Downing St Floor 1">
<!ENTITY city "New York">
]>
<Letter>
<To>&quot;Tom Smith&quot;</To>
     <Address>&address;</Address>
     <City>&city;</City>
   <Body>
    Hi! How are you?
    The sum is &gt; $1000
   </Body>
   <From>ARNOLD</From>
</Letter>
```

1.4.7 Entity Categories

Entities are used as shortcuts to refer to the data pages. Figure 1.10 shows the entity categories.

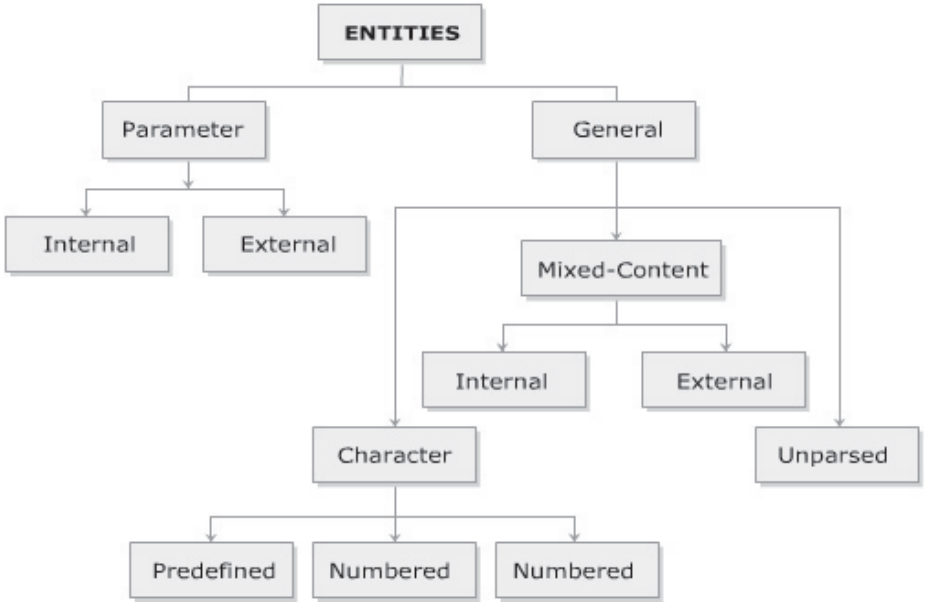


Figure 1.10: Entity Categories

The two types of entities are as follows:

- **General Entity**

  These are the entities used within the document content. General entities can either be declared internally or externally. The references for general entities start out with an ampersand (&) and end with a semicolon (;). The entity's name is present within these two characters.

  Every internal general entity is defined in the DTD. It is declared with the keyword <!ENTITY>. The syntax is as follows:

```
<!ENTITY Name "text that is to replaced">
```

  where,

  Name: value for the text that is to be replaced.

  External entities refer to the storage units outside the document containing the root element. Using external entity references, external entities can be embedded inside the document. An external entity reference indicates the location where the parser should insert the external entity in the document. The external entity has Uniform Resource Locator (URL) in its declaration; the URL specified indicates the document where the text of the entity is present. The syntax is as follows:

```
<!ENTITY Name SYSTEM "URL">
```

  The following code demonstrates an example of general entity.

  Code Snippet:

```
<!DOCTYPE MusicCollection [
    <!ENTITY R "Rock">
    <!ENTITY S "Soft">
    <!ENTITY RA "Rap">
    <!ENTITY HH "Hiphop">
    <!ENTITY F "Folk">
    ]>
```

- **Parameter Entity**

  These types of entities are used only in the DTD. These type of entities are declared in DTD. Both internal and external parameter entities should not be used in the content of the XML document as the processor does not recognise them. A well-formed parameter entity is similar to general entity, except that it will include the % specifier. The reference is also similar to the general entity reference.

References to these entities are made by using percent-sign (%) and semicolon (;) as delimiters.

The following code demonstrates an example of parameter entity.

Code Snippet:

```
<!ENTITY % ADDRESS "text that is to be represented by an entity">
```

A well-formed parameter entity will look like a general entity, except that it will include the "%" specifier.

1.4.8 Doctype Declarations

The doctype declaration defines the elements to be used in the document.

A doctype declaration is declared to indicate what DTD the document adheres to. It can be declared either in the XML document or can reference an the external document.

Syntax:

The syntax of document type declaration is as follows:

```
<! DOCTYPE name_of_root_element
         SYSTEM "URL of the external DTD subset" [
          Internal DTD subset
]>
```

where,

`name_of_root_element` is the name of the root element.

`SYSTEM` is the URL where the DTD is located.

`[Internal DTD subset]` are those declarations that are in the document.

The following code demonstrates an example of document type declaration.

Code Snippet:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE program SYSTEM "HelloJimmy.dtd">
<Program>
<Comments>
  This is a simple Java Program. It will display the message "Hello Jimmy,How
are you?" on execution.
</Comments>
<Code> public static void main(String[] args) {
           System.out.println("Hello Jimmy,How are you?"); // Display the
string.
    }
}
</Code>
</Program>
```

DTD File:

```
<!ELEMENT Program (comments, code)>
<!ELEMENT Comments (#PCDATA)>
<!ELEMENT Code (#PCDATA)>
```

The output is shown in figure 1.11.

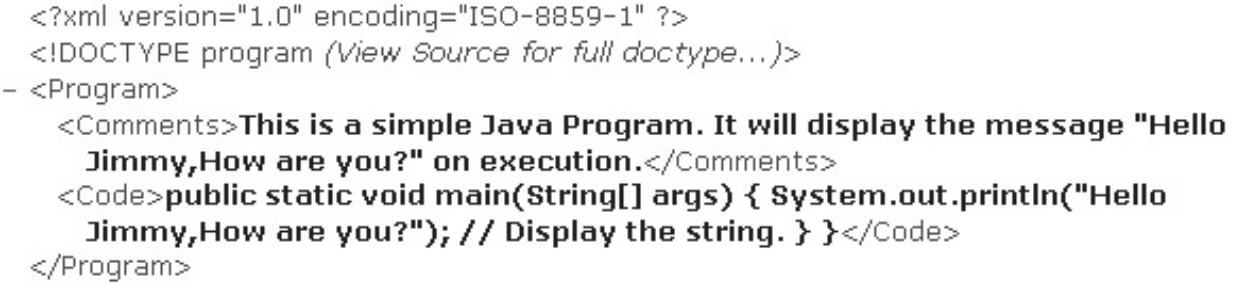


Figure 1.11: Output After Applying DTD

1.4.9 Attributes

Attributes are part of the elements. They provide information about the element and are embedded in the element start-tag. An attribute consists of an attribute name and an attribute value. The name always precedes its value, which are separated by an equal sign. The attribute value is enclosed in the quotes to delimit multiple attributes in the same element. An attribute can be of `CDATA`, `ENTITY`, `ENUMERATION`, `ID`, `IDREF`, `NMTOKEN` or `NOTATION`.

An enumerated attribute value is used when the attribute values are to be one of a fixed set of legal values. The attribute of identifier type (`ID`) should be unique. It is used to search a particular instance of an element. Each element can only have one attribute of type `ID`. `IDREF` is also an identifier type and it should only point to one element. `IDREF` attributes can be used to refer to an element from other elements. An attribute of type `NMTOKEN` allows any combination of name token characters. The characters can be letters, numbers, periods, dashes, colons or underscores. An `NMTOKENS` type attribute allows multiple values but separated by white space. A `NOTATION` type attribute must refer to a notation declared elsewhere in the DTD. A declaration can also be an example for a list of notations.

Syntax:

```
<elementName attName1="attValue2" attName2="attValue2"...>
```

The following code demonstrates an example of attributes.

Code Snippet:

```
<?xml version="1.0" ?>
  <Player Sex="male">
     <FirstName>Tom</FirstName>
     <LastName>Federer</LastName>
  </Player>
```

Attributes have some limitations, which are as follows:

- Unlike child elements, they do not contain multiple values and do not describe structures.
- They are not expandable for future changes.
- They are not easily manipulated by the code.
- They are not easy to test against a DTD.

Knowledge Check 4

1. Which of the following statements are true in the case of comments and processing instructions in XML?

(A)	Comments are processed by the processor.
(B)	Comments appear only in the document prolog.
(C)	Processing instructions are application specific.
(D)	Processing instructions are passed to the target.
(E)	`</PITarget <instruction>/>` is a processing instruction.

2. Which of the following statements are valid for character data in XML?

(A)	Character data is treated as regular text.
(B)	Characters like ">" and "&" can be used in `PCDATA` sections.
(C)	Characters like ">" and "&" can be used in CDATA sections.
(D)	CDATA starts with "`<![CDATA[`" and ends with "`]]>`".
(E)	Parameter entities use ampersand (&) and semicolon (;) as delimiters.

Module Summary

In this module, **Introduction to XML,** you learnt about:

- **Introduction to XML**

  XML was developed to overcome the drawbacks of earlier markup languages. XML consists of set of rules that describe the content to be displayed in the document. XML markup contains the content in the information containers called as elements.

- **Exploring XML**

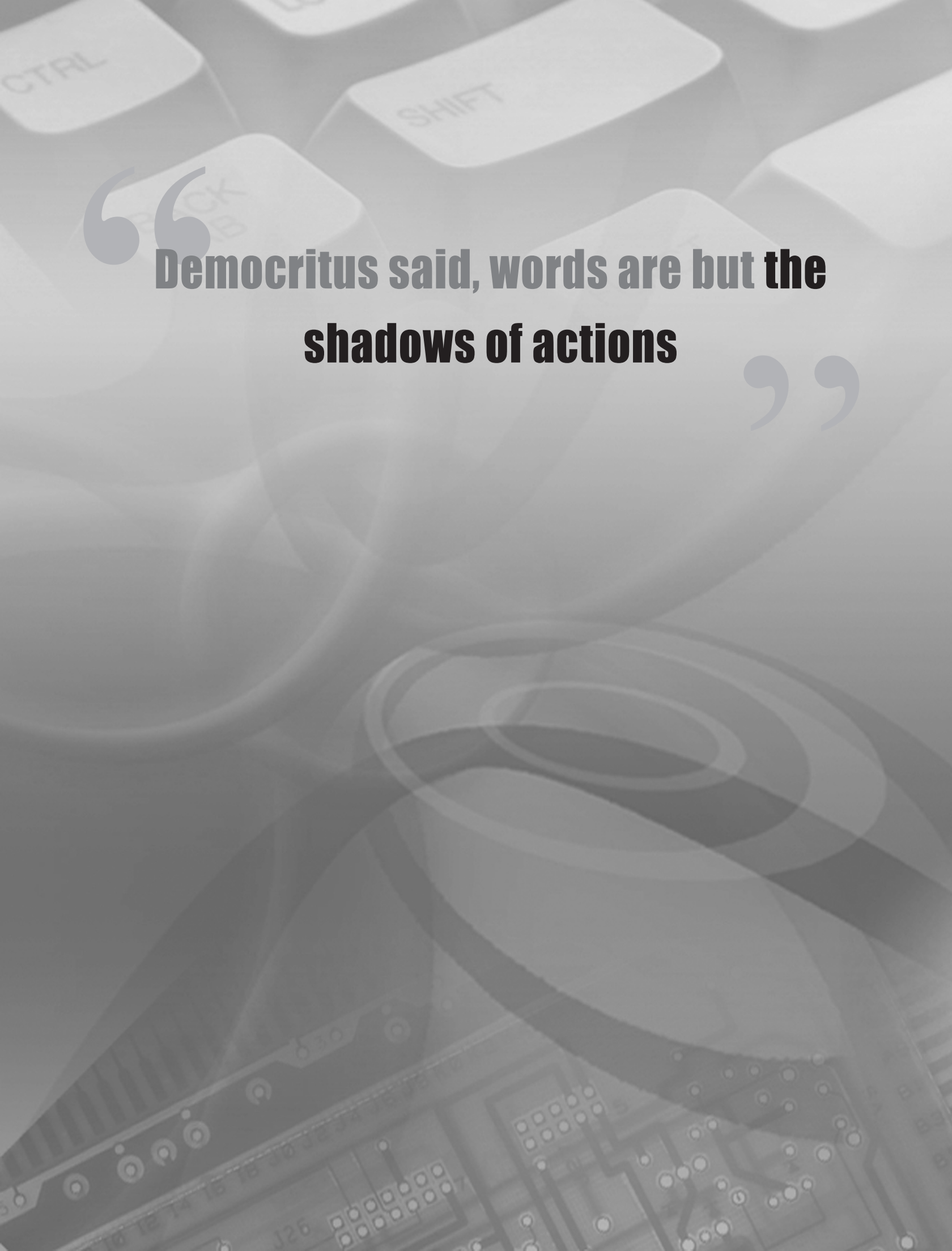
  XML document is divided into two parts namely document prolog and root element. An XML editor creates the XML document and a parser validates the document.

- **Working with XML**

  XML document is divided into XML Version Declaration, DTD and the document instance in which the markup defines the content. The XML markup is again categorized into structural, semantic and stylistic. The output of the XML document is displayed in the browser if it is well formed.

- **XML Syntax**

  Comments are used in the document to give information about the line or block of code. The content in XML document is divided into mark up and character data. The entities in XML are divided into general entities and parameter entities. A DTD can be declared either internally or externally.



“Democritus said, words are but **the shadows of actions**”

Module

2 Namespaces

Concepts

Module Overview

Welcome to the module, **Namespaces**. This module introduces XML Namespaces and the reasons for using Namespaces in XML documents. This module aims at giving a clear understanding of Namespaces syntax.

In this module, you will learn about:

- XML Namespaces
- Working with Namespaces Syntax

2.1 XML Namespaces

In this first lesson, **XML Namespaces**, you will learn to:

- Identify the need for a namespace.
- Define and describe namespaces in XML.

2.1.1 Duplicate Element Names

XML allows developers to create their own elements and attributes for their own projects. These elements or attributes can be shared by the developers working on similar projects all over the world.

For example, an author of an XML document includes element called `<title>` in a `<CD>` element and another author creates an element `<title>` in a `<Book>` element. A problem will arise if the `<title>` element created by two different authors, is merged into a single XML document. XML developer has to ensure the uniqueness of the element names and attributes in a document.

2.1.2 Consequences of Duplicate Element Names

When authors begin integrating the XML documents from different developers, name conflicts are inevitable. In such a scenario, it becomes difficult for the browser to distinguish a conflicting element in a XML document.

Consider an example where there are two occurrences of the `<title>` element in an XML document, one suggesting the title of a CD whereas another suggesting the title of a book. Imagine that an application is supposed to access this document and search for the `<title>` element. An XML parser will be lost if it does not get an additional information to search for a specific title.

Figure 2.1 depicts duplicate elements.

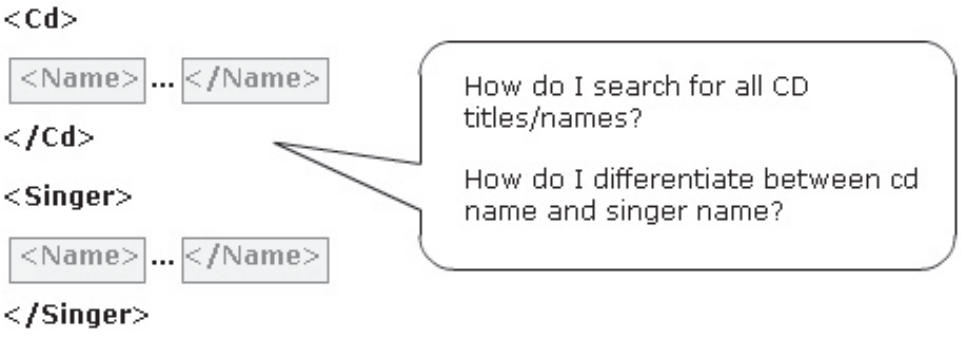


Figure 2.1: Duplicate Elements

2.1.3 Namespaces

In XML, elements are distinguished by using namespaces. XML Namespaces provide a globally unique name for an element or attribute so that they do not conflict one another.

A namespace is a collection of names that can be used as element names or attribute names in XML document.

XML namespaces provide the following advantages:

- **Reusability**

  XML Namespaces enable reuse of markup by making use of the elements and attributes that were defined earlier.

- **Modularity**

  Reusable code modules can be written, and these can be invoked for specific elements or attributes. Elements and attributes from different modules can be integrated into a single XML document. Universally unique element names and attributes guarantee that such modules can be invoked for certain elements and attributes.

- **Extensibility**

  XML Namespaces provide the XML documents with the ability to embed elements and attributes from other vocabularies like MathML, XHTML (Extensible HyperText MarkUp Language), and so forth.

Note: A Namespace is identified by the Uniform Resource Identifier (URI). For example, we can have all three elements named as 'batch'. The first referring the batch of students in Aptech Education Center, the second a batch of products, and the third, a batch of tourists. The element batch can be identified with a unique URI as given.

```
http://www.Aptech_edu.ac.batch
http://www.tea.org.batch
http://www.tourism.org.batch
```

Knowledge Check 1

1. Which of these statements about XML elements and XML Namespaces are true and which statements are false?

(A)	Browser has the ability distinguish duplicate element names in an XML document.
(B)	XML developer has to ensure the uniqueness of the element names and attributes in a document.
(C)	A namespace is a collection of names that can be used as element names or attribute names in XML document.
(D)	In XML, elements are distinguished by using DTD.

2.2 Working with Namespaces Syntax

In this last lesson, **Working with Namespaces syntax**, you will learn to:

- Explain the syntax for XML namespaces.
- Discuss attributes and namespaces.
- Discuss how to use default namespaces.

2.2.1 Prefixing Element Names

Namespaces are a mechanism by which element and attribute names can be assigned to groups. They also ensure that there is no conflict within element names. The best way to solve this problem is for every element in a document to have a completely distinct name. Therefore, the XML Namespace prefixes are directly attached to names as well as names of its attributes and descendants.

Using prefixes in the element names provide a means for the document authors to prevent name collisions, as demonstrated in the following code.

Code Snippet:

```
<CD:Title> Feel </CD:Title>
```

and

```
<Book:Title> Returning to Earth </Book:Title>.
```

In the example both `CD` and `Book` are namespace prefixes.

2.2.2 Problems Posed By Prefixes

There is a drawback to the prefix approach of the namespaces in an XML document. The reason for prefixing an element in a XML document is to prevent it from being duplicated. However, if the prefixes are not unique, the original problem of duplication would still exist.

To solve this problem, each namespace prefix is added to a Uniform Resource Identifier or URI that uniquely identifies the namespace. URI is a series of characters used to differentiate names.

To guarantee name uniqueness in any XML documents that the company creates, the documents are associated with their namespace. The following code demonstrate a company's URI which is located at `http://www.spectrafocus.com`.

Code Snippet:

```
<S:Student xmlns:S= "http://www.spectrafocus.com /student/">
<S:First>John</S:First>
<S:Last>Dewey</S:Last>
<S:Title>Student</S:Title>
</S:Student>
```

Note: The namespace prefix is an abbreviation for the namespace identifier (URI). For the student application, the prefix S or s can be chosen. The elements with the S prefix are said to have qualified names. The part of the name after the colon is called the local name.

2.2.3 Namespace Syntax

The XML namespace attribute is placed in the start tag of an element and the syntax for the namespace is given in figure 2.2.

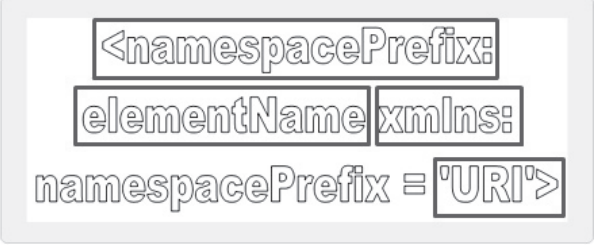


Figure 2.2: Namespace Syntax

- **namespacePrefix**

  Each namespace has a prefix that is used as a reference to the namespace. Prefixes must not begin with `xmlns` or `xml`. A namespace prefix can be any legal XML name that does not contain a colon. A legal XML name must begin with a letter or an underscore.

- **elementName**

  It specifies the name of the element.

- **xmlns**

  The xmlns attribute is what notifies an XML processor that a namespace is being declared. xmlns stands for XML Namespace.

- **URI**

  A Uniform Resource Identifier (URI) is a string of characters which identifies an Internet Resource. The URI includes Uniform Resource Name (URN) and a Uniform Resource Locator (URL). URLs contain the reference for a document or an HTML page on the Web. The URN is a universally unique number that identifies Internet resources. Additionally, URIs are also case-sensitive, which means that the following two namespaces are different: `http://www.XMLNAMESPACES.com`, `http://www.xmlnamespaces.com`.

  The purpose of namespaces, when used in an XML document, is to prevent the collision of similar elements and attribute names. A namespace in XML is simply a collection of element and attribute names identified by a URI reference. A URI reference is nothing but a string identifier. A URI need not be valid, for example, there may not be an actual Web site with the name `http://www.xmlnamespaces.com`. In particular, the URIs do not need to be valid or point to an actual resource.

URIs need not be valid, therefore, XML namespaces treat them like strings. In particular, comparisons are done character-by-character. According to this definition, the following URIs are not identical even though they point to the same document: `http://www.xmlnamespaces.com`, `http://xmlnamespaces.com`.

The following code demonstrates the properties of namespace.

Code Snippet:

```
<Auc:Books xmlns:Auc="http://www.auction.com/books"
            xmlns:B="http://www.books.com/HTML/1998/xml1">
...
<Auc:BookReview>
<B:Table>
...
```

The elements that are prefixed with `Auc` are associated with a namespace with name `http://www.auction.com/books`, whereas those prefixed with `B` are associated with a namespace whose name is `http://www.books.com/HTML/1998/xml1`.

Note: URIs encompasses both URLs and URNs. URNs differ from URLs in that URLs describe the physical location of the particular resource, whereas URNs define a unique location-independent name for a resource that maps to one or more URLs. URLs all begin with Internet service prefix such as `ftp:`, `http:`, and so on, whereas URNs begin with urn: prefix.

2.2.4 Placing Attributes in a Namespace

Attributes belong to particular elements and they are not a part of namespace, even if the element is within some namespace. However, if an attribute name has no prefix, it has no namespace. An attribute without a prefix is in default namespace. If an attribute name has a prefix, its name is in the namespace indicated by the prefix.

Syntax:

The syntax for including an attribute in a namespace is,

```
prefix:localname='value'
```

or

```
prefix:localname="value"
```

where,

`prefix` is used as a reference to the namespace. Prefixes must not begin with `xmlns` or `xml`.

`localname` is the name of an attribute.

`value` mentions a user defined value for an attribute.

In the following code, the `type` attribute is associated with the `book` namespace since it is preceded by the prefix `Book`.

Code Snippet:

```
<Catalog xmlns:Book = "http://www.aptechworldwide.com">
<Book:Booklist>
<Book:Title Book:Type = "Fiction">Evening in Paris</Book:Title>
<Book:Price>$123</Book:Price>
</Book:Booklist>
</Catalog>
```

Note: Attributes from a particular namespace can also be added to elements from a different namespace. The following example demonstrates this concept.

```
<Catalog xmlns= "http://www.Aptech_edu.ac" xmlns:Author= "http://www.
aptechworldwide.com">
<Book Type= "Adventure">The Last Samurai</Book>
<Book Author:Type= "Fiction">Hannibal</Book>
<Book>American Dream</Book>
</Catalog>
```

2.2.5 Default Namespaces

Consider an XML document which contains the details of all books in a library. Since this document includes a lot of markup all in the same namespace it may be inconvenient to add a prefix to each element name. Consider another scenario where an XML document is to merge with a `MathML` document.

This problem can be solved by adding a default namespace to an element and to its child elements using an xmlns attribute with no prefix.

A default namespace is used by an element and its child elements if the element does not have a namespace prefix.

Concepts

- **MathML Document**

  Mathematical Markup Language (MathML) is an XML-based markup language to represent complex mathematical expressions. It comes in two types, as markup language for presenting the layout of mathematical expressions and as a markup language for presenting the mathematical content of the formula. For example, expression `x + 1` can be written in `MathML` as demonstrated in the following code.

  Code Snippet:

```
<MRow>
  <Mi>x</Mi>
  <Mo>+</Mo>
  <Mn>1</Mn>
</MRow>
```

Syntax:

The syntax of a default namespace is given as,

```
<elementName xmlns='URL'>
```

where,

`elementName` specifies the name of the element belonging to the same namespace.

`URL` specifies the namespace which is reference for a document or an HTML page on the Web.

The following code demonstrates the default namespace.

Code Snippet:

```
<Catalog xmlns="http://www.aptechworldwide.com">
<BookList>
  <Title type = "Thriller">African Safari</Title>
  <Price>$12</Price>
  <ISBN>23345</ISBN>
</Booklist>
</Catalog>
```

A default namespace using the `xmlns` attribute with a URI as its value. Once this default namespace is declared, child elements that are part of this namespace do not need a namespace prefix.

2.2.6 Override Default Namespaces

The default namespace applies to the element on which it was defined and all descendants of that element. If one of the descendants has another default namespace defined on it, this new namespace definition overrides the previous one and becomes the default for that element and all its descendants as demonstrated in the following code.

Code Snippet:

```
<Catalog xmlns = "http://www.aptechworldwide.com">
<Book>
<Title type = "Fiction">Evening in Paris</Title>
<Price>$123</Price>
</Book>
<Book>
<Title type = "Non-Fiction">Return to Earth</Title>
<Price xmlns = "http://www.aptech.ac.in">$23</Price>
<Title type = "Non-Fiction">Journey to the center of the Moon</Title>
<Price>$123</Price>
</Book>
</Catalog>
```

Notice that in the `price` element in the second book element a different namespace is provided. This namespace applies only to the `price` element and overrides the namespace in the `catalog` element.

2.2.7 Best Practices for Using XML Namespaces

Some of the best practices that are used for declaring XML Namespaces are as follows:

- Namespaces should be properly planned.
- Logical and consistent prefix names should be used for convenience while creating an XML document.
- Namespaces should be applied to a personal vocabulary, even when there is just one.
- Namespaces should be used to separate or isolate elements that may otherwise seem similar.
- When designing two vocabularies that have some elements in common, one namespace should be used to hold the common items.
- HTTP URLs should be used for the URIs.

Concepts

- URIs should be coordinated by naming under a particular domain name.
- Namespace URI should change for every substantive change to the vocabulary, including the addition and deletion of new elements.
- If possible, one prefix should be used for one namespace throughout all XML documents in a system.
- All namespace declarations should be made in the start tag of the document element.
- Domain names like namespaces.com, namespaces.net, or namespaces.org should be used in the XML document.

Knowledge Check 2

1. Which of the following line of code are correct namespace declarations?

(A)	`<Title:Catalog xmlns:Book ="http://www.aptechworldwide.com">`
(B)	`<CD:Catalog CD:xmlns = "http://www.aptechworldwide.com">`
(C)	`<Car xmlns:Vehicle xmlns = http://www.aptechworldwide.com>`
(D)	`<CD:xmlns CD:Catalog = "http://www.aptechworldwide.com">`

2. Which of these statements about attributes and namespaces are true and which statements are false?

(A)	Attributes belonging to a particular elements within some namespace is also a part of the same namespace.
(B)	An attribute without a prefix is in default namespace.
(C)	`xmlns:localname="value"` is the correct syntax for including a attribute in a namespace.
(D)	`<Student:Name age = "12">Kevin</Student:Name>` is the correct for associating age with the student namespace.
(E)	The prefix used in an attribute is used as a reference to the namespace.

3. Which of these statements about default namespaces are true and which statements are false?

(A)	`<elementName xmlns='URL'>` is the correct syntax for declaring a default namespace.
(B)	The descendant has the same namespace as the parent element even if it has a new namespace definition.
(C)	A default namespace is used by an element and its child elements if the element has a namespace prefix.
(D)	A default namespace applies to the element on which it was defined and all descendants of that element.
(E)	A descendant having a new namespace cannot override the namespace defined by the parent element.

Module Summary

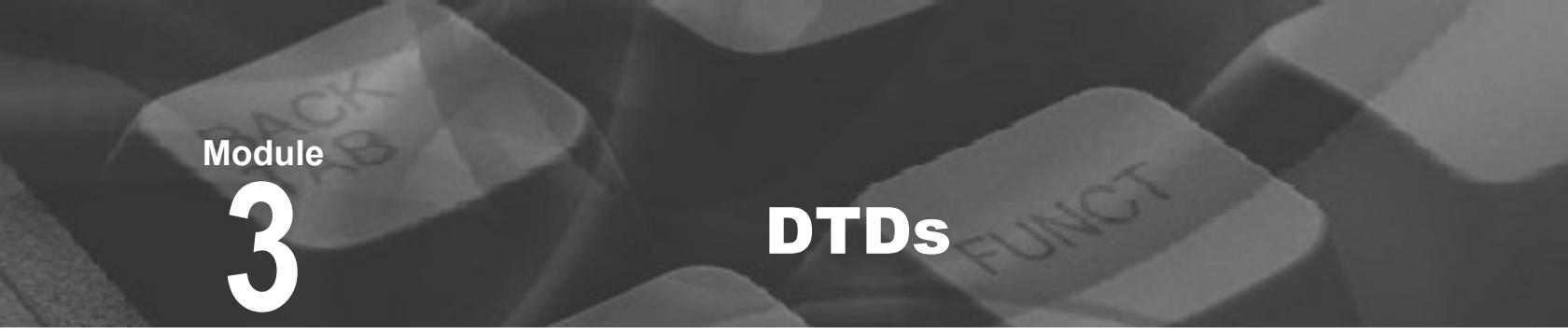
In this module, **Namespaces**, you learnt about:

- **XML Namespaces**

  Namespaces distinguish between elements and attributes with the same name from different XML applications. It is a collection of names that can be used as element names or attribute names in XML document. XML Namespaces provide a globally unique name for an element or attribute to avoid name collisions.

- **Working with Namespaces syntax**

  Namespaces are declared by an `xmlns` attribute whose value is the URI of the namespace. If an attribute name has no prefix, it has no namespace. A default namespace is used by an element and its child elements if the element does not have a namespace prefix.



Module 3

DTDs

Concepts

Module Overview

Welcome to the module, **Document Type Definitions (DTDs)**. This module focuses on how to create DTDs for XML files. In this module, topics such as DOCTYPE declarations, types of DTDs, well-formedness, and validity of XML files, declaration of elements, attributes, and entities in a DTD are also covered.

In this module, you will learn about:

- Document Type Definition
- Working with DTDs
- Valid XML Documents
- Declarations

3.1 Document Type Definition

In this first lesson, **Document Type Definition**, you will learn to:

- Define what is meant by a DTD.
- Identify the need for a DTD.

3.1.1 Definition of a DTD

A Document Type Definition (DTD) is a non XML document made up of element, attribute and entity declarations. The elements, attributes and entities defined belong to the XML document(s) the DTD is defined for.

The DTD helps XML parsers to validate the XML document. It also helps authors to set default values for element attributes. The document's tree structure can be determined based on the DTD.

Note:

Tree Structure

Hierarchical representation of an XML document, its root element, other elements, entities, and so on. This structure is conceptually interpreted as the document's tree structure.

Concepts

XML Parsers

An XML parser is a software program or set of programs that parses XML documents for its structure. XML parsers are also capable of validating XML files depending on DTDs.

3.1.2 Need for a DTD

The fact that XML allowed its user to define his/her own tag was one of its biggest advantages. However, it also led to a genuine problem. People working on parts of the same XML document sometimes used similar tags for different purposes and at other times, used different tags for the same purpose. At times, same elements possessed or exhibited different attributes. Some sort of standardization of elements and attributes was needed.

That is how DTDs came into picture. A DTD can define all the possible combinations and sequences for elements to be used in an XML document along with their attributes and their acceptable values.

3.1.3 DTD versus DOCTYPE

The difference between a Document Type Definition (DTD) and a Document Type Declaration (DOCTYPE) can be derived from table 3.1:

Properties	Document Type Definition	Document Type Declaration
Definition	A list of legal element, attribute and entity declarations for an XML document.	The declaration of the DTD to which the XML document adheres to.
Abbreviation	DTD	DOCTYPE
Location	Within the DOCTYPE declaration or an external file	Within the XML document's prolog
Syntax	<!ELEMENT element-name (element-content)> ... <!ENTITY entity-name "entity-value"> ... <!DOCTYPE name_of_root_element SYSTEM "URL of the external DTD subset" >	<!ATTLIST element-name attribute-name attribute-type default-value> Or <!DOCTYPE name_of_root_element [internal DTD subset]>
Language Type	Non XML	XML

Properties	Document Type Definition	Document Type Declaration
Variations	Present within a DOCTYPE (Internal DTD) or in an external document (External DTD)	Declaring either an internal or external DTD
Dependency	Is useful only if declared using DOCTYPE declaration	Is useful only if it has a DTD to define

Table 3.1: Difference between DTD and DOCTYPE Declarations

The difference between the two is clear from Table 3.1. The DOCTYPE declaration is very important for the DTD to be functional and in the absence of a DTD, a DOCTYPE declaration becomes ineffective.

DOCTYPE declarations can declare DTDs within themselves or refer to DTDs present in external documents. Depending on either of the two, DTDs can be termed as internal or external respectively. The following example shows how to declare a DTD within a DOCTYPE declaration.

Example:

```
File: Mobile.xml
...
<!DOCTYPE Mobile [
     <!ELEMENT Mobile (Company, Model, Price, Accessories)>
         <!ELEMENT Company (#PCDATA)>
         <!ELEMENT Model (#PCDATA)>
         <!ELEMENT Price (#PCDATA)>
         <!ELEMENT Accessories (#PCDATA)>
     <!ATTLIST Model Type CDATA "Camera">
     <!ENTITY HP "Head Phones">
     <!ENTITY CH "Charger">
     <!ENTITY SK "Starter's Kit">
]>
...
```

Additionally, the following example displays the same DTD shown in the previous example but in the form of an external DTD.

Example:

```
File: Mobile.dtd
     <!ELEMENT Mobile (Company, Model, Price, Accessories)>
     <!ELEMENT Company (#PCDATA)>
     <!ELEMENT Model (#PCDATA)>
     <!ELEMENT Price (#PCDATA)>
     <!ELEMENT Accessories (#PCDATA)>
<!ATTLIST Model Type CDATA "Camera">
<!ENTITY HP "Head Phones">
<!ENTITY CH "Charger">
<!ENTITY SK "Starter's Kit">

File: Mobile.xml
...
<!DOCTYPE Mobile SYSTEM "Mobile.dtd">
...
```

The DTD is declared in a separate file "Mobile.dtd". The DOCTYPE declaration in the XML file refers to it using the `SYSTEM` keyword.

3.1.4 Limitations of DTD

The purpose of DTDs is to facilitate validation. This fact makes DTDs both popular and essential. However, DTDs have limitations of their own. They are:

- DTDs validate XML documents but themselves are non-XML documents. One of the biggest advantages of XML is its extensibility. However, DTDs cannot boast of this extensibility.
- DTDs can be either external or internal depending on their DOCTYPE declaration. In spite of such provisions, every XML document can have only one DTD to adhere to. This limits the advantage of having both internal and external DTDs.
- DTDs do not follow the latest trends of being object oriented, inheritance, and so on. This puts them behind widely used programming languages and programmer preferences.
- XML documents can use multiple namespaces. To use a namespace, XML documents require the namespace to be declared within its DTD. However, DTDs do not support multiple namespaces. This defeats the very advantage of being able to declare multiple namespaces.

- DTDs support only one datatype, the text string. This is one of its biggest disadvantages depending on the number of applications XML is being used for today.
- Being able to define DTDs for standardizing XML documents is surely one of DTDs biggest tenets.
- However, this is proving to be helpless with internal DTDs being able to override external DTDs. Hours of efforts in creating effective DTDs could be ruined by malicious internal DTDs. Thus, the security aspect of DTDs is very doubtful.

Knowledge Check 1

1. Which of the statements about DTDs and their purpose are true and which statements are false?

(A)	A DTD is an XML document.
(B)	DTDs contain declarations for elements and entities.
(C)	DTDs are used to validate XML documents.
(D)	Each XML document can be represented as a tree structure.
(E)	Ability to create one's own tags is XML's greatest disadvantage.

3.2 Working with DTDs

In this second lesson, **Working with DTDs**, you will learn to:

- Describe the structure of a DTD.
- Explain how to create a simple DTD.
- Describe what is meant by document type declarations.

3.2.1 Structure of DTD

A typical DTD structure is composed of the following three blocks:

- Element Declarations
- Attribute Declarations
- Entity Declarations

Each element declaration specifies the name of the element, and the content which that element can contain. Each attribute declaration specifies the element that owns the attribute, the attribute name, its type and its default value (if any). Each entity declaration specifies the name of the entity and either its value or location of its value.

These groups of element, attribute and entity declarations that form the three blocks of a DTD respectively can be declared in any sequence.

3.2.2 Creating Internal DTDs

Creating DTDs is a simple six step process. These steps can be listed as follows:

1. Declare all the possible elements
2. Specify the permissible element children, if any
3. Set the order in which elements must appear
4. Declare all the possible element attributes
5. Set the attribute data types and values
6. Declare all the possible entities

Steps 1 to 3 deal with element declaration, steps 4 and 5 with attribute declarations and step 6 with entity declarations. This is in clear accordance with the structure of a DTD.

Syntax:

```
<!ELEMENT element-name (element-content)>
…
<!ATTLIST element-name attribute-name attribute-type default-value>
…
<!ENTITY entity-name "entity-value">
…
```

The following code demonstrates the properties of internal DTDs.

Code Snippet:

```
<!ELEMENT Mobile (Company, Model, Price, Accessories)>
<!ELEMENT Company (#PCDATA)>
<!ELEMENT Model (#PCDATA)>
<!ELEMENT Price (#PCDATA)>
<!ELEMENT Accessories (#PCDATA)>
<!ATTLIST Model Type CDATA "Camera">
<!ENTITY HP "Head Phones">
<!ENTITY CH "Charger">
<!ENTITY SK "Starters Kit">
```

3.2.3 DOCTYPE Declarations

A Document Type Declaration declares that the XML file in which it is present adheres to a certain DTD. The declaration also specifies the name of that DTD and either its content or location.

Placed in the XML document's prolog, a DOCTYPE declaration begins with <!DOCTYPE and ends with a >. In between is the name of the root element, followed either by a pair of square brackets containing the DTD itself or by the SYSTEM keyword and a URL specifying where the DTD can be found.

Without the document type declaration, a DTD is nothing but plain text.

Figure 3.1 shows the DOCTYPE declaration.

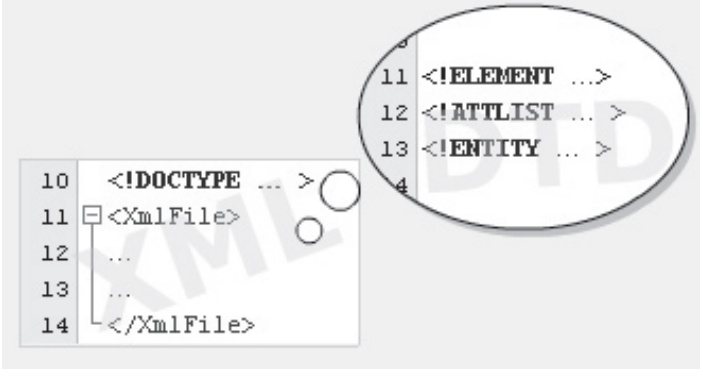


Figure 3.1: DOCTYPE Declarations

Syntax:

```
<!DOCTYPE name_of_root_element [ internal DTD subset ]>
```

or

```
<!DOCTYPE name_of_root_element SYSTEM "URL of the external DTD subset" >
```

3.2.4 Types of DTDs

DOCTYPE declarations have two basic forms of syntax. Depending on the document type declaration syntax used, DTDs can be classified as Internal or External DTDs.

- **Internal DTDs**

  Internal DTDs are characterized by the presence of the DTD in the document type declaration itself. The document type declaration consists of the DTD name followed by the DTD enclosed in square brackets.

  Figure 3.2 depicts the internal DTD.

Internal DTD

```
14  ...
15  <!DOCTYPE Mobile SYSTEM "mobile.dtd">
16  ...
```

Figure 3.2: Internal DTD

- **External DTDs**

  External DTDs are characterized by the presence of the DTDs address path in the document type declaration. The document type declaration consists of the DTDs name followed by the `SYSTEM` keyword followed by the address of the DTD document.

Figure 3.3 depicts the external DTD.

External DTD Reference

```
<!DOCTYPE Mobile [
  <!ELEMENT Mobile (Company, Model, Price, Accessories)>
  <!ELEMENT Company (#PCDATA)>
  <!ELEMENT Model (#PCDATA)>
  <!ELEMENT Price (#PCDATA)>
  <!ELEMENT Accessories (#PCDATA)>
  <!ATTLIST Model Type CDATA "Camera">
  <!ENTITY HP "Head Phones">
  <!ENTITY CH "Charger">
  <!ENTITY SK "Starters Kit">
]>
...
```

```
11 <!ELEMENT Mobile (Company, Model, Price, Accessories)>
12 <!ELEMENT Company (#PCDATA)>
13 <!ELEMENT Model (#PCDATA)>
14 <!ELEMENT Price (#PCDATA)>
15 <!ELEMENT Accessories (#PCDATA)>
16 <!ATTLIST Model Type CDATA "Camera">
17 <!ENTITY HP "Head Phones">
18 <!ENTITY CH "Charger">
19 <!ENTITY SK "Starters Kit">
```

Figure 3.3: External DTD

Knowledge Check 2

1. Which of the statements about DTD structure and DOCTYPE declarations are true and which statements are false?

(A)	DTDs are made up of three blocks of declarations and the DOCTYPE declaration.
(B)	Elements, attributes and entities can be declared in any order.
(C)	DOCTYPE declarations are specified in the prolog of the XML document.
(D)	Internal DTDs specify the DTD within square brackets in the declaration itself.
(E)	External DTDs use the keyword URL to specify the location of the DTD.

2. Can you arrange the steps of creating DTDs in order?

(1)	Declare all the possible element attributes.
(2)	Specify the permissible element children, if any.
(3)	Set the attribute data types and values.
(4)	Set the order in which elements must appear.
(5)	Declare all the possible entities.
(6)	Declare all the possible elements.

Concepts

3.3 Valid XML Documents

In this third lesson, **Valid XML Documents**, you will learn to:

- Define document validity.
- Describe in brief how to test for document validity.

3.3.1 Well-Formed XML Documents

For an XML document to execute properly, it should be well-formed. A well-formed XML document adheres to the basic XML syntax rules. The World Wide Web Consortium in its specifications, states that XML documents with errors should not be processed by any program. The basic XML syntax rules are as follows:

- All elements must be enclosed by the root element
- All elements must have closing tags
- All tags should be case sensitive
- All elements must be properly nested
- All attribute values must always be quoted

Figure 3.4 shows an example of well-formed XML document.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
- <Mail>
    <To>anne@xyz.com</To>
    <From>bob@xyz.com</From>
    <Date>27th February 2007</Date>
    <Time>11:30 am</Time>
    <Cc />
    <Bcc />
    <Subject>Meeting at Main Conference Room at 4:30pm</Subject>
    <Message>Hi, Kindly request you to attend the cultural body
      general meeting in the main conference room at 4:30 pm.
      Please be present to learn about the new activities being
      planned for the employees for this year. Yours sincerely,
      Bob</Message>
    <Signature />
  </Mail>
```

Figure 3.4: Well Formed XML Document

3.3.2 Valid XML documents

A Valid XML document is a well-formed XML document that adheres to its DTD. The validity of an XML document is determined by checking it against its DTD. Once it has been confirmed that the components used in the XML document adhere to the declarations in the DTD, a well-formed XML document can be termed as a valid XML document.

Validity of XML documents plays an important role in all XML applications. Be it creating modular parts of the code, or data interchange, or processing of transferred data, or database access, and so forth, validity of the XML document is a recommended feature. The following code demonstrates a valid XML document.

Code Snippet:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE Mail [
<!ELEMENT Mail (To, From, Date, Time, Cc, Bcc, Subject, Message,
Signature)>
<!ELEMENT To (#PCDATA)>
<!ELEMENT From (#PCDATA)>
<!ELEMENT Date (#PCDATA)>
<!ELEMENT Time (#PCDATA)>
<!ELEMENT Cc (#PCDATA)>
<!ELEMENT Bcc (#PCDATA)>
<!ELEMENT Subject (#PCDATA)>
<!ELEMENT Message (#PCDATA)>
<!ELEMENT Signature (#PCDATA)>
]>
<Mail>
<To> anne@xyz.com </To>
<From> bob@xyz.com </From>
<Date> 27th February 2007 </Date>
<Time> 11:30 am </Time>
<Cc> </Cc>
<Bcc> </Bcc>
<Subject> Meeting at Main Conference Room at 4:30pm </Subject>
<Message> Hi, Kindly request you to attend the cultural body general
meeting in the main conference room at 4:30 pm. Please be present to learn
about the new activities being planned for the employees for this year.
Yours sincerely, Bob </Message>
<Signature> </Signature>
</Mail>
```

Concepts

3.3.3 Testing XML for Validity

Validity being a desirable trait, it is necessary to be able to validate XML documents after their creation. Validity of XML documents can be determined by using a validating parser such as MSXML 6.0 Parser.

MSXML enables the Internet Explorer (IE) browser to validate the code. Once the code is displayed in IE, right-click the code to display the context menu. The menu provides the option of validating the code. On selection, IE internally validates the code against the code's DTD.

The first image of figure 3.5 displays the validation result of a valid `mobile.xml` file.

The second image of figure 3.5 displays the validation result after the removal of the signature element from the document's DTD.

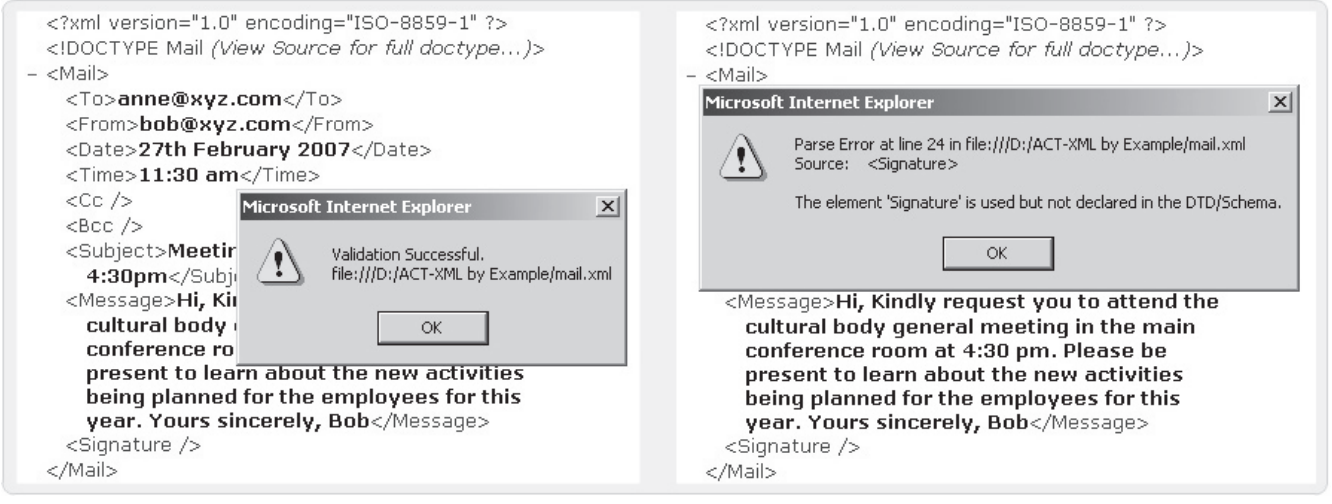


Figure 3.5: Testing the validity of an XML file

Knowledge Check 3

1. Which of the following XML code snippets is valid?

(A)

```
<!DOCTYPE mobile [
     <!ELEMENT Mobile (Company, Model, Price, Accessories)>
     <!ELEMENT Company (#PCDATA)>
     <!ELEMENT Model (#PCDATA)>
     <!ELEMENT Accessories (#PCDATA)>
     <!ATTLIST Model Type CDATA "Camera">
     <!ENTITY HP SYSTEM "hp.txt">
     <!ENTITY CH SYSTEM "ch.txt">
     <!ENTITY SK SYSTEM "sk.txt">
]>
<Mobile>
<Company> Nokia </Company>
<Model Type="Camera"> 6600 </Model>
<Price> 9999 </Price>
<Accessories> &HP;, &CH; and a &SK; </Accessories>
</Mobile>

hp.txt
Head Phones

ch.txt
Charger

sk.txt
Starter's Kit
```

(B)

```
<!DOCTYPE Mobile [
     <!ELEMENT Mobile (Company, Model, Price, Accessories)>
     <!ELEMENT Company (#PCDATA)>
     <!ELEMENT Model (#PCDATA)>
     <!ELEMENT Price (#PCDATA)>
     <!ELEMENT Accessories (#PCDATA)>
     <!ATTLIST Model Type CDATA "Camera">
     <!ENTITY HP SYSTEM "hp.txt">
     <!ENTITY SK SYSTEM "sk.txt">
]>
<Mobile>
<Company> Nokia </Company>
<Model Type="Camera"> 6600 </Model>
<Price> 9999 </Price>
<Accessories> &HP;, &CH; and a &SK; </Accessories>
</Mobile>

hp.txt
Head Phones

ch.txt
Charger

sk.txt
Starter's Kit
```

(C)

```
<!DOCTYPE Mobile [
     <!ELEMENT Mobile (Company, Model, Price, Accessories)>
     <!ELEMENT Company (#PCDATA)>
     <!ELEMENT Model (#PCDATA)>
     <!ELEMENT Price (#PCDATA)>
     <!ELEMENT Accessories (#PCDATA)>
     <!ATTLIST Model Type CDATA "Camera">
     <!ENTITY HP SYSTEM "hp.txt">
     <!ENTITY CH SYSTEM "ch.txt">
     <!ENTITY SK SYSTEM "sk.txt">
]>
<Mobile>
<Company> Nokia </Company>
<Model Type="Camera"> 6600 </Model>
<Price> 9999 </Price>
<Accessories> &HP;, &CH; and a &SK; </Accessories>
</Mobile>

hp.txt
Head Phones

ch.txt
Charger

sk.txt
Starter's Kit
```

(D)

```
<!DOCTYPE Mobile [
     <!ELEMENT Mobile (Company, Model, Price, Accessories)>
     <!ELEMENT Company (#PCDATA)>
     <!ELEMENT Model (#PCDATA)>
     <!ELEMENT Price (#PCDATA)>
     <!ELEMENT Accessories (#PCDATA)>
     <!ATTLIST Model Type CDATA "Camera">
     <!ENTITY HP SYSTEM "hp.txt">
     <!ENTITY CH SYSTEM "ch.txt">
     <!ENTITY SK SYSTEM "sk.txt">
]>
<Mobile>
<Company> Nokia </Company>
<Model Type="Camera"> 6600 </Model>
<Price> 9999
<Accessories> &HP;, &CH; and a &SK; </Accessories>
</Mobile>

hp.txt
Head Phones

ch.txt
Charger

sk.txt
Starter's Kit
```

3.4 Declarations

In this last lesson, **Declarations**, you will learn to:

- Explain how to declare elements.
- Explain how to declare attributes.
- Describe entity declaration in a DTD.

3.4.1 Declaring Elements

In the DTD, XML elements are declared with an element declaration. An element declaration has the following syntax:

Syntax:

```
<!ELEMENT element-name element-rule>
```

where,

`ELEMENT` is the keyword,

`element-name` is the name of the element,

`element-rule` can be one of the following: No Content, Only Parsed Character Data. Any Contents, Children, Only One Occurrence, Minimum One Occurrence, Zero or More Occurrences, Zero or One Occurrence, Either/Or Content or Mixed Content.

Figure 3.6 depicts an example of element declaration.

```
15 <!ELEMENT Mobile ( Company, Model, Price, Accessories)>
16   <!ELEMENT Company (#PCDATA)>
17   <!ELEMENT Model (#PCDATA)>
18   <!ELEMENT Price (#PCDATA)>
19   <!ELEMENT Accessories (#PCDATA)>
20 <!ATTLIST Model Type CDATA "Camera">
21 <!ENTITY HP "Head Phones">
22 <!ENTITY CH "Charger">
23 <!ENTITY SK "Starters Kit">
```

Figure 3.6: Element Declaration

Some of the syntax and declarations for the element-rule options are shown in table 3.2.

Value	Description	Syntax	Example
No Content	These elements are empty and accept no data.	`<!ELEMENT element-name EMPTY>`	`<!ELEMENT Signature EMPTY>`
Only Parsed Character Data	Elements contain character data that needs to be parsed.	`<!ELEMENT element-name (#PCDATA)>`	`<!ELEMENT Signature (#PCDATA)>`

Value	Description	Syntax	Example
Any Contents	Elements can contain any combination of data that can be parsed.	<!ELEMENT element-name ANY>	<!ELEMENT Signature ANY>
Children	Elements with one or more children are defined with the name of the children elements inside parentheses followed by their declarations in the same order.	<!ELEMENT element-name (child-element-name)> or <!ELEMENT element-name (child-element-name, child-element-name,.....)>	<!ELEMENT Mail (To, From, Date, Time, Cc, Bcc, Subject, Message, Signature)> <!ELEMENT To (#PCDATA)> … …

Table 3.2: Element-Rule Options

Other declarations and the syntax that can be given for the element-rule options are shown in table 3.3.

Value	Description	Syntax	Example
Only One Occurrence	Elements with children declared only once within the parentheses implicitly appear only once in the XML document.	<!ELEMENT element-name (child-name)>	<!ELEMENT Mail (To, From, Date, Time, Cc, Bcc, Subject, Message, Signature)>
Minimum One Occurrence	Element children names accompanied by a '+' sign inside the parentheses should appear at least once in the XML document.	<!ELEMENT element-name (child-name+)>	<!ELEMENT Mail (To+, From+, Date+, Time+, Cc, Bcc, Subject+, Message+, Signature)>
Zero or More Occurrences	Elements with children names accompanied by a '*' sign inside the parentheses can or cannot appear in the XML document.	<!ELEMENT element-name (child-name*)>	<!ELEMENT Mail (To*, From+, Date+, Time+, Cc, Bcc, Subject+, Message+, Signature)>

Value	Description	Syntax	Example
Zero or One Occurrences	Elements with children names accompanied by a '?' sign inside the parentheses can be skipped or only appear once in the XML document.	`<!ELEMENT element-name (child-name?)>`	`<!ELEMENT Mail (To*, From+, Date+, Time+, Cc, Bcc, Subject+, Message+, Signature?)>`
Either/Or Content	Elements can have either one of two or more children by specifying them in parentheses separated by '|'.	`<!ELEMENT element-name (child-name, (child-name|child-name)…)>`	`<!ELEMENT Mail (To, From, Date, Time, (Cc|Bcc), Subject, Message, Signature)>`
Mixed Content	Elements can be declared to accept mixed content, either data type or children, etc.	`<!ELEMENT element-name (type|child-name|…)>`	`<!ELEMENT Mail (To|From|Date|Time|Cc|Bcc|Subject|Message|Signature)>`

Table 3.3: Other Element-Rule Options

3.4.2 Declaring Attributes

An attribute declaration has the following syntax:

Syntax:

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

where,

the `element-name` is the element the attribute belongs to,

the `attribute-name` is the name of the attribute,

the `attribute-type` is type of data the attribute can accept,

and `default-value` id the default value for the attribute.

The attribute-type can have values as shown in table 3.4.

Value	Description
PCDATA	Parsed character data
CDATA	Character data
(en1|en2|..)	Enumerated list
ID	A unique id
IDREF	Id of another element
IDREFS	List of other ids
NMTOKEN	Valid XML name
NMTOKENS	List of valid XML names
ENTITY	An entity
ENTITIES	List of entities
NOTATION	Name of a notation
xml:	Predefined xml value

Table 3.4: Attribute-Type

3.4.3 Specifying Attribute Values

The attribute–value in a DTD declaration can have values as shown in table 3.5.

Value	Description
value	Default value
#REQUIRED	Value must be included
#IMPLIED	Value does not have to be included
#FIXED	Value is fixed
en1|en2|...	Listed enumerated values

Table 3.5: Attribute-Value

Syntax:

- **#IMPLIED**

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

- **#REQUIRED**

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

- **#FIXED**

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

- **Enumerated Attribute Values**

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

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

Each of the following codes demonstrates the attribute values.

Code Snippet:

- **Default Value**

```
<!ATTLIST Model Type CDATA "Camera">
```

- **#IMPLIED**

```
<!ATTLIST Model Type CDATA "Camera" #IMPLIED>
```

- **#REQUIRED**

```
<!ATTLIST Model Type CDATA #REQUIRED>
```

- **#FIXED**

```
<!ATTLIST Model Type CDATA #FIXED "Camera">
```

- **Enumerated Attribute Values**

```
<!ATTLIST Model Type (Camera|Bluetooth) "Camera">
```

3.4.4 Entities in DTD

An entity, in XML, is a placeholder that consists of a name and a value. It is declared once and then repeatedly used through out the document in place of its value.

Entities can be categorized as parameter entities and general entities. Parameter entities have been discussed earlier.

General entities are further classified as:

- Character
- Mixed Content
- Unparsed

Character entities are entities that have a single character as their replacement value. Character entities are further classified as:

- Pre-defined
- Numbered
- Name

XML has a set of pre-defined entities to facilitate the use of characters like <, >, &, ", ', since these are used by XML in its tags.

The **Pre-defined** entities and their references are shown in table 3.6.

Entity Name	Value
Entity Name	Value
amp	&
apos	'
gt	>
lt	<
quot	"

Table 3.6: Pre-defined Entities

Numbered entities are entities which use a character reference number, for a character from the Unicode character set, in place of the character itself. These entity references have a special syntax. Instead of the "&", "&#" is used. The reference number is used as the name, followed by a semicolon. Thus, a numbered entity reference would look like this: `&#245;`. The numbers can also be specified in hexadecimal, if required, using the following syntax:

```
&#x(reference number in hexadecimal);
```

Name entities are the same as Numbered entities, except that the numbers are replaced by a descriptive name for the character.

Mixed content entities are entities that contain either text or markup language text as their replacement value. For example, the entity test could also have "trial" or "<trial>Hi!</trial>" as its replacement value. The text "<trial>Hi!</trial>" will be replaced with the entity reference and will act as an element trial.

Unparsed entities are entities that have data that is not to be parsed. The replacement value might be an image or a file or something else. The syntax for an unparsed entity is as follows:

```
<!DOCTYPE name [
    <!ENTITY entity-name SYSTEM "entity-location" NDATA Notation Identifier>
]>
```

The keyword `NDATA` stands for notation data and notation identifier specifies the format of the file or object.

At the time of processing, the XML parser goes through the document declarations. The parser then scans the document for all entity references. When the parser encounters an entity reference, it replaces the entity reference with the entity value associated with that entity. The parser then resumes parsing from the replaced text. Entity references within the replaced text are also replaced by doing so.

Syntax:

- **Entity declaration**:

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

- **Entity Reference**:

```
&entity-name;
```

Concepts

Figure 3.7 depicts the entities in DTD.

```
<!DOCTYPE Mobile [
  <!ELEMENT Mobile (Company, Model, Price, Accessories)>
    <!ELEMENT Company (#PCDATA)>
    <!ELEMENT Model (#PCDATA)>
    <!ELEMENT Price (#PCDATA)>
    <!ELEMENT Accessories (#PCDATA)>
  <!ATTLIST Model Type CDATA "Camera">
  <!ENTITY HP "Head Phones">
  <!ENTITY CH "Charger">
  <!ENTITY SK "Starters Kit">
]>
<Mobile>
  <Company> Nokia </Company>
  <Model Type="Camera"> 6600 </Model>
  <Price> 9999 </Price>
  <Accessories> &HP;, &CH; and a &SK; </Accessories>
</Mobile>
```

Figure 3.7: Entities in DTD

3.4.5 Kinds of Entity Declarations

Entity references are used extensively in XML. That leads to Internal and External entity declarations, the two basic kinds of entity declarations used in XML.

- **Internal Entity Declaration**

  In internal entity declarations, the entity value is explicitly mentioned in the entity declaration.

  Syntax:

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

The following code demonstrates the properties of internal entity declaration.

Code Snippet:

```
<!DOCTYPE Mobile [
     <!ELEMENT Mobile (Company, Model, Price, Accessories)>
     <!ELEMENT Company (#PCDATA)>
     <!ELEMENT Model (#PCDATA)>
     <!ELEMENT Price (#PCDATA)>
     <!ELEMENT Accessories (#PCDATA)>
     <!ATTLIST Model Type CDATA "Camera">
     <!ENTITY HP "Head Phones">
     <!ENTITY CH "Charger">
     <!ENTITY SK "Starters Kit">
]>
<Mobile>
<Company> Nokia </Company>
<Model Type="Camera"> 6600 </Model>
<Price> 9999 </Price>
<Accessories> &HP;, &CH; and a &SK; </Accessories>
</Mobile>
```

- **External Entity Declaration**

In External entity declaration, a link or path to the entity value is mentioned in place of the entity value with the help of the SYSTEM keyword. External entity declarations are in a way similar to declaring External DTDs.

Syntax:

```
<!ENTITY entity-name SYSTEM "URI/URL">
```

The following code demonstrates the properties of external entity declaration.

Code Snippet:

```
<!DOCTYPE Mobile [
     <!ELEMENT Mobile (Company, Model, Price, Accessories)>
     <!ELEMENT Company (#PCDATA)>
     <!ELEMENT Model (#PCDATA)>
     <!ELEMENT Price (#PCDATA)>
     <!ELEMENT Accessories (#PCDATA)>
     <!ATTLIST Model Type CDATA "Camera">
     <!ENTITY HP SYSTEM "hp.txt">
     <!ENTITY CH SYSTEM "ch.txt">
     <!ENTITY SK SYSTEM "sk.txt">
]>
<Mobile>
<Company> Nokia </Company>
<Model Type="Camera"> 6600 </Model>
<Price> 9999 </Price>
<Accessories> &HP;, &CH; and a &SK; </Accessories>
</Mobile>

hp.txt
Head Phones

ch.txt
Charger

sk.txt
Starters Kit
```

Knowledge Check 4

1. Can you match the element declaration descriptions with their correct syntaxes?

Description		Syntax	
(A)	Element can contain character data but to be parsed, incase it contains entity references.	**(1)**	`<!ELEMENT element-name (type|child-name|…)>`
(B)	Element with a number of children appearing only once in the XML document.	**(2)**	`<!ELEMENT element-name (child-name+)>`
(C)	Element can accept either data type or children, and so forth.	**(3)**	`<!ELEMENT element-name (child-name, (child-name|child-name)…)>`
(D)	Element's children appear at least once in the XML document.	**(4)**	`<!ELEMENT element-name (#PCDATA)>`
(E)	Element can have either one of two or more children.	**(5)**	`<!ELEMENT element-name (child-element-name)>`

2. Can you match the attribute descriptions with their correct values?

Description		Value	
(A)	Name of a notation	**(1)**	`NMTOKEN`
(B)	Predefined xml value	**(2)**	`NOTATION`
(C)	id of another element	**(3)**	`(en1|en2|..)`
(D)	valid XML name	**(4)**	`xml:`
(E)	enumerated list	**(5)**	`IDREF`

Concepts

3. Which of the following XML code is correct?

(A)

```
<!DOCTYPE Mobile [
     <!ELEMENT Mobile (Company, Model, Price, Accessories)>
     <!ELEMENT Company (#PCDATA)>
     <!ELEMENT Model (#PCDATA)>
     <!ELEMENT Price (#PCDATA)>
     <!ELEMENT Accessories (#PCDATA)>
     <!ATTLIST Model Type CDATA "Camera">
     <!ENTITY HP "Head Phones">
     <!ENTITY SK "Starters Kit">
]>
<Mobile>
<Company> Nokia </Company>
<Model Type="Camera"> 6600 </Model>
<Price> 9999 </Price>
<Accessories> &HP;, &CH; and a &SK; </Accessories>
</Mobile>
```

(B)

```
<!DOCTYPE Mobile [
     <!ELEMENT Mobile (Company, Model, Price, Accessories)>
     <!ELEMENT Company (#PCDATA)>
     <!ELEMENT Model (#PCDATA)>
     <!ELEMENT Price (#PCDATA)>
     <!ELEMENT Accessories (#PCDATA)>
     <!ATTLIST Model Type CDATA "Camera">
     <!ENTITY HP "Head Phones">
     <!ENTITY CH "Charger">
     <!ENTITY SK "Starters Kit">
]>
<Mobile>
<Company> Nokia </Company>
<Model Type="Camera"> 6600 </Model>
<Price> 9999 </Price>
<Accessories> HP, CH and a SK </Accessories>
</Mobile>
```

(C)

```
<!DOCTYPE Mobile [
     <!ELEMENT Mobile (Company, Model, Price, Accessories)>
     <!ELEMENT Company (#PCDATA)>
     <!ELEMENT Model (#PCDATA)>
     <!ELEMENT Price (#PCDATA)>
     <!ELEMENT Accessories (#PCDATA)>
     <!ATTLIST Model Type CDATA "Camera">
     <!ENTITY HP "hp.txt">
     <!ENTITY CH "ch.txt">
     <!ENTITY SK "sk.txt">
]>
<Mobile>
<Company> Nokia </Company>
<Model Type="Camera"> 6600 </Model>
<Price> 9999 </Price>
<Accessories> &HP;, &CH; and a &SK; </accessories>
</Mobile>
```

(D)

```
<!DOCTYPE Mobile [
     <!ELEMENT Mobile (Company, Model, Price, Accessories)>
     <!ELEMENT Company (#PCDATA)>
     <!ELEMENT Model (#PCDATA)>
     <!ELEMENT Price (#PCDATA)>
     <!ELEMENT Accessories (#PCDATA)>
     <!ATTLIST Model Type CDATA "Camera">
     <!ENTITY HP "Head Phones">
     <!ENTITY CH "Charger">
     <!ENTITY SK "Starters Kit">
]>
<Mobile>
<Company> Nokia </Company>
<Model Type="Camera"> 6600 </Model>
<Price> 9999 </Price>
<Accessories> &HP;, &CH; and a &SK; </Accessories>
</Mobile>
```

Concepts

Module Summary

In this module, **Document Type Definitions**, you learnt about:

- **Document Type Definition**

  This lesson defines DTDs and identifies the need for DTDs.

- **Working with DTDs**

  This lesson talks about the structure of DTDs, steps to create DTDs, DOCTYPE declarations and the type of DTDs.

- **Valid XML Documents**

  This lesson talks about the well-formed valid documents and the steps to determine the validity of XML documents.

- **Declarations**

  This lesson deals with declarations of elements, attributes, and entities in DTDs.

Module

4 XML Schema

Concepts

Module Overview

Welcome to the module, **XML Schema**. This module focuses on exploring XML schema and its features. The structure of an XML document with the help of schema is also discussed. This module aims at providing a clear understanding of the different elements and data types supported by an XML schema.

In this module, you will learn about:

- XML Schema
- Exploring XML Schemas
- Working with complex types
- Working with simple types

4.1 XML Schema

In this first lesson, **XML Schema**, you will learn to:

- Define and describe what is meant by schema.
- Identify the need for schema.
- Compare and differentiate the features of DTDs and schemas.

4.1.1 Schema

DTDs define document structure and validate XML documents, but have some limitations. Hence, an XML-based alternative to DTDs, known as XML schema has been introduced, with an objective to overcome the drawbacks of DTDs.

The word schema originated from a Greek word symbolizing form or shape. The dictionary meaning of schema is: "A diagrammatic representation; an outline or a model". Initially, the word schema was only in the reach of the philosophers till it entered the zone of computer science. In the context of software, a schema is generally understood to be a model used to describe the structure of a database. It defines internal structures such as tables, fields, and the relationship between them.

Concepts

However, in the context of XML, as defined by the W3C, a schema is "a set of rules to constrain the structure and articulate the information set of XML documents". A schema describes a model for a whole class of documents. The model describes the way in which the data is marked up, and also specifies the possible arrangement of tags and text in a valid document. A schema might be considered as a common vocabulary that is needed to exchange documents between different organizations.

An XML Schema defines the valid building blocks of an XML document. It can be considered as a common vocabulary that different organizations can share to exchange documents. The XML Schema language is referred as XML Schema Definition (XSD).

Figure 4.1 depicts the XML data validation.

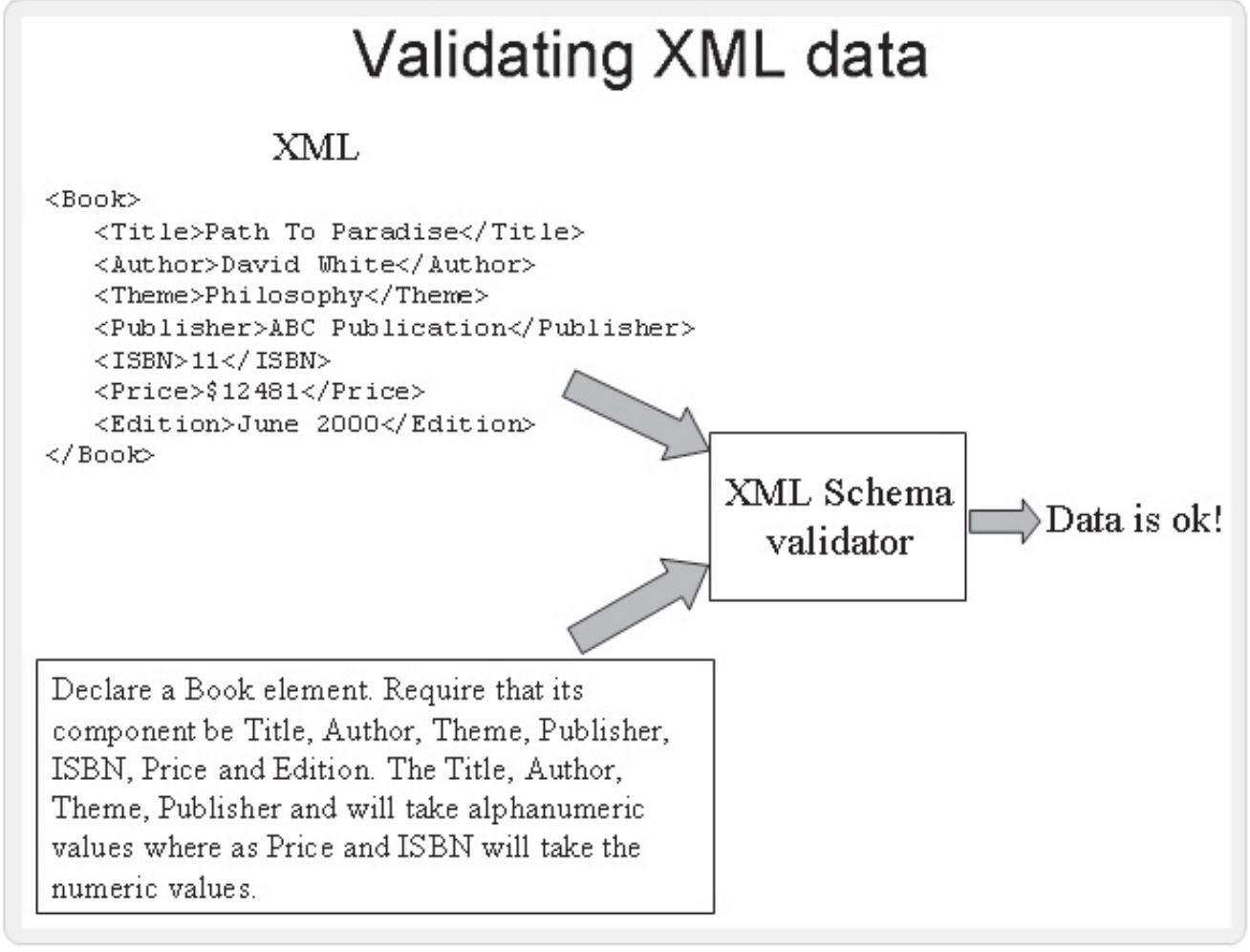


Figure 4.1: XML Data Validation

The following XML code demonstrates an entity 'BOOK'. When this document is accessed through a browser, it will represent the details of a book.

Code Snippet:

```
<Book>
      <Title>Path To Paradise</Title>
      <Author>David White</Author>
      <Theme>Philosophy</Theme>
      <Publisher>ABC Publication</Publisher>
      <ISBN>11</ISBN>
      <Price>$12481</Price>
      <Edition>June 2000</Edition>
</Book>
```

The logical reasoning would compare the attributes of this 'BOOK', with the attributes of a book in general. In other words, one's previous knowledge of what a book is, and what its attributes are, could be a kind of schema, against which this 'BOOK' is compared. This would help to validate the attributes of the book.

4.1.2 Definition of XML Schema

The objective of an XML Schema is to define the valid structure of an XML document, similar to a DTD.

An XML Schema defines:

- elements and attributes that can appear in a document
- which elements are child elements
- the order and number of child elements
- whether an element is empty or can include text
- data types for elements and attributes
- default and fixed values for elements and attributes

Schemas overcome the limitations of DTDs and allow Web applications to exchange XML data more robustly, without relying on ad hoc validation tools.

Concepts

Some of the basic features that an XML schema offers are:

- XML syntax is used as the basis for creating XML schema documents. It does not require learning a new cryptic language as is the case with DTDs.
- XML schemas can be manipulated just like any other XML document.

Figure 4.2 depicts XML schema.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <!--
4      Document   : books.xml
5      Author     : vincent
6      Description:
7          This document defines a schema for a library of books.
8  -->
9
10 <library>
11 <fiction>
12 <book>The Firm, John Grisham</book>
13 <book>Coma, Robin Cook</book>
14 </fiction>
15 <nonfiction>
16 <book>Freakonomics , Malcolm Gladwell</book>
17 </nonfiction>
18 <?latest editions only?>
19 </library>
```

Figure 4.2: XML Schema

4.1.3 Writing an XML Schema

To begin the exploration of schemas with the Hello World example, you need two files, they are:

- **XML File**

  Given XML document contains a single element, `<Message>`. A schema for this document has to declare the `<Message>` element.

- **XSD File**

  For storing schema documents, the file is saved with "`.xsd`" as the extension. Schema documents are XML documents and can have DTDs, DOCTYPE declarations.

Figure 4.3 depicts XSD file.

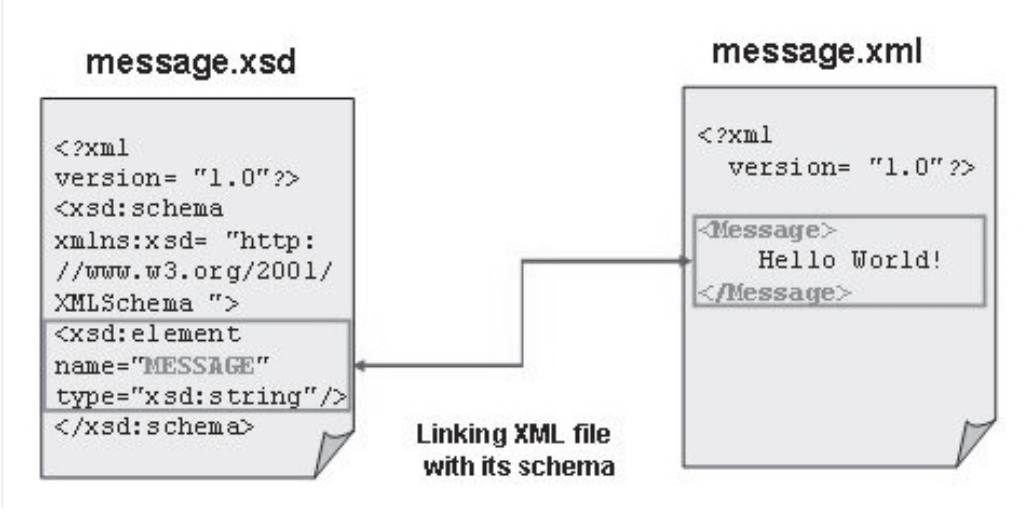


Figure 4.3: XSD File

The following code demonstrates the properties of XML schema.

Code Snippet:

XML File: `message.xml`

```
<?xml version= "1.0"?>
<Message>
    Hello World!
</Message>
```

XSD File: `message.xsd`

```
<?xml version= "1.0"?>
<xsd:schema xmlns:xsd= "http://www.w3.org/2001/XMLSchema ">
<xsd:element name= "MESSAGE " type= "xsd:string "/>
</xsd:schema>
```

Given XSD file is a schema for `<Message>` elements. The file is stored as `message.xsd`. It is an XML document; therefore it has an XML declaration. This schema file may declare other elements too, including ones that are not present in this XML file, but it must at least declare the `<Message>` element.

Concepts

4.1.4 Features of Schema

XML schemas allow Web applications to exchange XML data more robustly using a range of new features. They are:

- **Schemas Support Data Types**

  The support of XML schema for the data types and the ability to create the required datatypes has overcome the drawbacks of DTDs. It means it is easy to define and validate valid document content and data formats. A developer can find it simple to work with data from a database. It can also be easy to implement the restrictions on data.

- **Schemas are portable and efficient**

  Schemas are portable and efficient as they use the same XML syntax. Hence, there is no need to learn any new language or any editor to edit the schema files. A similar XML parser can be used to parse the Schema files.

- **Schemas secure data communication**

  During data transfer, it is essential that both sender and receiver should have the same "vocabulary". With XML Schemas, the sender can specify the data in a way that the receiver will understand.

- **Schemas are extensible**

  A schema definition can be extended; hence, it is possible to reuse an existing schema to create another schema. A developer can create his own data types derived from the standard types. Schemas also support reference of multiple schemas in the same document.

- **Schemas catch higher-level mistakes**

  XML checks for well-formedness of an XML document to validate the basic syntax. A document is said to be well-formed if it has an XML declaration, unique root element, matching start and end tag, properly nested elements, and so on. A well-formed document can still have errors. XML Schemas can catch higher-level mistakes that arise, such as a required field of information is missing or in a wrong format, or an element name is mis-spelled.

- **Schemas support Namespace**

  The support for XML Namespaces allows the programmer to validate documents that use markup from multiple namespaces. It means that constructs can be re-used from schemas already defined in a different namespace.

The additional features of schema are as follows:

- **Richer Datatypes**

  The Schema draft defines booleans, dates and times, URIs, time intervals, and also numeric types like decimals, integers, bytes, longs, and many more.

- **Archetypes**

  An archetype is used to define the custom-named datatype from pre-existing data types. For example, a 'ContactList' datatype is defined, and then two elements, 'FriendsList' and 'OfficialList' are defined under that type.

- **Attribute Grouping**

  It is possible to have common attributes that apply to all elements, or several attributes that include graphic or table elements. Attribute grouping allows the schema author to make this relationship between elements explicit. Parameter entity supports grouping in DTDs, which simplifies the process of authoring a DTD, but the information is not passed on to the processor.

- **Refinable Archetypes**

  A DTD follows a 'closed' type of content model. Content model is defined as the constraint on the content of elements in an instance XML document. The 'closed' content model describes all, and only those elements and attributes that may appear in the content of the element. XML Schema allows two more possibilities:

  - Open type content model
  - Refinable content model

  In an 'open' content model, elements other than the required elements can also be present. The open content model allows the inclusion of child elements and attributes within an element that are not declared in the document's schema. DTDs only support closed content models, which require declaring all elements and attributes in order to use them in a document. Additional elements may be present in a 'refinable' content model, but the schema should define those additional elements.

Concepts

4.1.5 Comparing DTDs with Schemas

XML inherited the concept of DTDs from Standard Generalized Markup Language (SGML), which is an international standard for markup languages. DTDs are used to define content models, nesting of elements in a valid order, and provide limited support to data types and attributes. The drawbacks of using DTDs are:

- **DTDs are written in a non-XML syntax**

  DTDs do not use XML notation and therefore they are difficult to write and use.

- **DTDs are not extensible**

  DTDs are not extensible. For example, if there is an Address DTD to catalog friends, and one wants to add a new section to the code for official contacts, then the entire DTD has to be rewritten.

- **DTDs do not support namespaces**

  Namespaces can be used to introduce an element type into an XML document. However, a Namespace cannot be used to refer to an element or an entity declaration in the DTD. If a Namespace is used, then the DTD has to be modified to include any elements taken from the Namespace.

- **DTDS offer limited data typing**

  DTDs can only express the data type of attributes in terms of explicit enumerations and a few coarse string formats. DTDs do not have a facility to describe numbers, dates, currency values, and so forth. Furthermore, DTDs do not have the ability to express the data type of character data in elements. For example, a `<Zip>` element can be defined to contain `CDATA`. However, the element cannot be constrained to just numerals when it uses a DTD.

  Figure 4.4 shows a sample DTD.

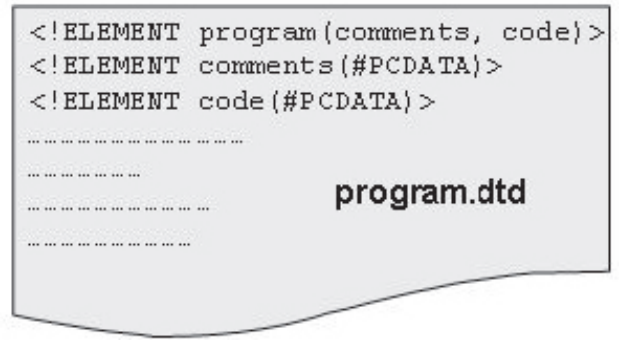


Figure 4.4: A Sample DTD

The following code demonstrates a sample external DTD file: `program.dtd`

Code Snippet:

```
<!ELEMENT program (comments, code)>
<!ELEMENT comments (#PCDATA)>
<!ELEMENT code (#PCDATA)>
```

The following code demonstrates a sample XML File with a reference to dtd: `program.xml`

Code Snippet:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE program SYSTEM "program.dtd">
<program>
    <comments>
        This is a simple Java Program. It will display the message
        "Hello world!" on execution.
    </comments>
    <code>
        public static void main(String[] args)
        System.out.println("Hello World!"); // Display the string.
    </code>
</program>
```

Additionally, examples are shown here to demonstrate how DTDs and schemas are referenced.

Simple XML Document: **Book.xml**

The following code demonstrates a simple XML document called "**`Book.xml`**":

Code Snippet:

```
<?xml version="1.0"?>
<Book>
    <Title> Million Seconds </Title>
    <Author> Kelvin Brown </Author>
<Chapter> The plot of the story starts from here. </Chapter>
</Book>
```

DTD File for Book.xml

The following code demonstrates a DTD file called "Book.dtd" that defines the elements of the **Book.xml** document.

Code Snippet:

```
<!ELEMENT Book (Title, Author, Chapter)>
    <!ELEMENT Title (#PCDATA)>
    <!ELEMENT Author (#PCDATA)>
    <!ELEMENT Chapter (#PCDATA)>
```

The first line of code defines the **Book** element to be the root element which in turn encloses three child elements: 'Title', 'Author', and 'Chapter'. The rest of the block defines the 'Title', 'Author', and 'Chapter' elements to be of type "#PCDATA".

XML Schema for Book.xml

The following code demonstrates that the corresponding XML Schema file called "Book.xsd" defines the elements of the XML document **Book.xml**.

Code Snippet:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.booksworld.com" xmlns="http://www.booksworld.
com" elementFormDefault="qualified">
    <xs:element name="Book">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="Title" type="xs:string"/>
                    <xs:element name="Author" type="xs:string"/>
                    <xs:element name="Chapter" type="xs:string"/>
                </xs:sequence>
            </xs:complexType>
        </xs:element>
</xs:schema>
```

The **Book** element is a complex type because it encloses other elements. The child elements **Title**, **Author**, **Chapter** are simple types because they do not contain other elements.

Reference to Book.DTD in the instance document named Book.xml

The following code demonstrates that a reference to "Book.dtd" document is placed inside in an XML document named `Book.xml`.

Code Snippet:

```
<?xml version="1.0"?>
<!DOCTYPE note SYSTEM "http://www.booksworld.com/dtd/Book.dtd">
<Book>
    <Title>Million Seconds</Title>
    <Author>Kelvin Brown</Author>
    <Chapter>The plot of the story starts from here.</Chapter>
</Book>
```

Reference to XML Schema Book.xsd in the instance document named Book.xml

This XML document has a reference to an XML Schema.

The following code along with the previous code demonstrate how schema and DTD are declared and referred in instance XML documents.

Code Snippet:

```
<?xml version="1.0"?>
<Book xmlns="http://www.booksworld.com" xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance"   xsi:schemaLocation="http://www.booksworld.com   Book.
xsd">
    <Title>Million Seconds</Title>
    <Author>Kelvin Brown</Author>
    <Chapter>The plot of the story starts from here.</Chapter>
</Book>
```

Concepts

4.1.6 Advantages of XML Schemas over DTD

Schemas overcome the limitations of DTDs and allow Web applications to exchange XML data more robustly, without relying on ad hoc validation tools.

The XML schema offers a range of new features:

- **Richer data types**

  The Schema draft defines booleans, numbers, dates and times, URIs, integers, decimal numbers, real numbers, and also time intervals.

- **Archtypes**

  An archetype allows to define own named datatype from pre-existing data types. For example, one can define a '`ContactList`' datatype, and then define two elements, "`FriendsList`" and '`OfficialList`' under that type.

- **Attribute grouping**

  There can be common attributes that apply to all elements, or several attributes that include graphic or table elements. Attribute grouping allows the schema author to make this relationship explicit.

- **Refinable archetypes**

  A DTD follows a 'closed' type of model. It describes all, and only those elements and attributes that may appear in the content of the element. XML Schema allows two more possibilities: 'open' and 'refinable'. In an 'open' content model, elements other than the required elements can also be present. Additional elements may be present in a refinable content model, but the schema should define those additional elements.

Code Snippet:

The following code demonstrates a sample schema File: `mail.xsd`

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

```
      <xs:sequence>
     <xs:element name="to" type="xs:string"/>
     <xs:element name="from" type="xs:string"/>
     <xs:element name="header" type="xs:string"/>
     <xs:element name="body" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
</xs:element>
</xs:schema>
```

The following code demonstrates a sample XML File with a reference to schema: `mail.xml`

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

<to>John</to>
<from>Jordan</from>
<header>Scheduler</header>
<body>3rd March Monday, 7:30 PM: board meeting!</body>
</mail>
```

Knowledge Check 1

1. Which of these statements about schemas are true and which statements are false?

(A)	An XML Schema defines the structure of an XML document.
(B)	An XML Schema is an XML-based add-on to DTDs.
(C)	XML syntax is used as the basis to create a schema, so it can be stored with the same extension `.xml` (dot XML).
(D)	An XML Schema defines how many child elements can appear in an XML document.
(E)	An XML Schema defines whether an element is empty or can include text.

Concepts

2. Can you match the different features for DTD and schemas against their corresponding description?

Description		Feature	
(A)	Allows to define own named data type from pre-existing data types.	**(1)**	Attribute grouping
(B)	Allows the schema author to make the attributes common that apply to all elements, or several attributes that include graphic or table elements.	**(2)**	Namespace support
(C)	Describes elements which are not required to be present in the XML document.	**(3)**	Closed model
(D)	Describes only those elements and attributes that may appear in the content of the element.	**(4)**	Archetypes
(E)	Allows to validate documents that use markup from multiple namespaces.	**(5)**	Open model

4.2 Exploring XML Schemas

In this second lesson, **Exploring XML Schemas**, you will learn to:

- List the data types supported by schemas.
- Explain the XML Schema vocabulary.

4.2.1 Data Types Supported by Schema

XML Schema describes a number of built-in data types, which can be used to specify and validate the intended data type of the content. It also allows a user to create a user-defined data type by extending the built-in data types using facets. Hence, XML Schema recommendation defines two sorts of data types, these are:

- Built-in data types, are available to all XML Schema authors, and should be implemented by a conforming processor.
- User-derived data types, are defined in individual schema instances, and are particular to that schema (although it is possible to import these definitions into other definitions).

In XML schema, the data types can be broadly classified into built-in and derived types.

The built-in data types specified by XML working group are:

- **string**

  A group of characters is called a string. The `string` data type can contain characters, line feeds, carriage returns, and tab characters. The string may also consist of a combination of Unicode characters. Unicode is a universal standard to describe all possible characters of all languages with a library of symbols with one unique number for each symbol.

  Syntax:

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

  Code Snippet:

  The `string` declaration in the schema is:

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

  An element in the xml document will be:

  ```
  <Customer>John Smith</Customer>
  ```

- **boolean**

  The `boolean` data type is used to specify a mathematical representation. Legal values for boolean data type are true and false. True can be replaced by the numeric value 1 and false can be replaced by the value 0.

  Syntax:

  ```
  <xs:attribute name="attribute_name" type="xs:boolean"/>
  ```

  Code Snippet:

  The boolean declaration in the schema is:

  ```
  <xs:attribute name="Disabled" type="xs:boolean"/>
  ```

  An element in the xml document will be:

  ```
  <Status Disabled="true">OFF</Status>
  ```

- **numeric**

The data type `numeric` represents a numerical value. It includes numbers such as whole numbers, and real numbers.

Syntax:

```
<xs:element name="element_name" type="xs:numeric"/>
```

Code Snippet:

The numeric declaration in the schema is:

```
<xs:element name="Price" type="xs:numeric"/>
```

An element in the xml document will be:

```
<Price>500</Price>
```

- **dateTime**

It represents a particular time on a given date, written as a string. For example, "`2001-05-10T12:35:40`" can be considered as a dateTime string. The date is specified in the following form "`YYYY-MM-DDThh:mm:ss`":

where,

`YYYY` indicates the year,

`MM` indicates the month,

`DD` indicates the day,

`T` indicates the start of the required time,

`hh` indicates the hours,

`mm` indicates the minutes,

`ss` indicates the seconds.

Syntax:

```
<xs:element name="element_name" type="xs:dateTime"/>
```

Code Snippet:

The datetime declaration in the schema is:

```
<xs:element name="BeginAt" type="xs:dateTime"/>
```

An element in the xml document will be:

```
<start>2001-05-10T12:35:40</start>
```

- **binary**

The `binary` type can include graphic files, executable programs, or any other string of binary data. Binary data types are used to express binary-formatted data of two types such as base64Binary (Base64-encoded binary data) and hexBinary (hexadecimal-encoded binary data).

Syntax:

```
<xs:element name="image_name" type="xs:hexBinary"/>
```

Code Snippet:

```
<xs:element name="Logo" type="xs:hexBinary"/>
```

- **anyURI**

A universal resource identifier (URI) represents a file name or location of the file.

Syntax:

```
<xs:attribute name="image_name" type="xs:anyURI"/>
```

Code Snippet:

The `anyURI` declaration in the schema is:

```
<xs:attribute name="flower" type="xs:anyURI"/>
```

An element in the xml document will be:

```
<image flower="http://www.creativepictures.com/gallery/flower.gif" />
```

Note: Unicode character is a character that has two bytes as its size.

4.2.2 Additional Data Types

Additional data types are derived from the basic built-in data types, which are called base type data types.

The generated or derived data types, supported by the XML schema include:

- **integer**

  The base type for integer is the numeric data type. It includes both positive and negative numbers. For example, the numbers -2, -1, 0, 1, 2 are integers. The `integer` data type is used to specify a numeric value without a fractional component.

  Syntax:

  ```
  <xs:element name="element_name" type="xs:integer"/>
  ```

  Code Snippet:

  The integer declaration in the schema is:

  ```
  <xs:element name="Age" type="xs:integer"/>
  ```

  An element in the xml document will be:

  ```
  <Age>999</Age>
  ```

- **decimal**

  It can represent exact fractional parts such as 3.26. The basetype for decimal is the number data type. The decimal data type is used to specify a numeric value.

  Syntax:

  ```
  <xs:element name="element_name" type="xs:decimal"/>
  ```

Code Snippet:

The decimal declaration in the schema is:

```
<xs:element name="Weight" type="xs:decimal"/>
```

An element in the xml document will be:

```
<prize>+70.7860</prize>
```

- **time**

The base type is the `dateTime` data type. The default representation is 16:35:26. The time data type is used to specify a time. The time is specified in the following form "`hh:mm:ss`" where, `hh` stands for hour, `mm` indicates the minute and `ss` indicates the second.

Syntax:

```
<xs:element name="element_name" type="xs:time"/>
```

Code Snippet:

The decimal declaration in the schema is:

```
<xs:element name="BeginAt" type="xs:time"/>
```

An element in the xml document will be:

```
<BeginAt>09:30:10.5</BeginAt>
```

Concepts

4.2.3 Schema Vocabulary

Creating a schema using XML schema vocabulary is like creating any other XML document using a specialized vocabulary. To understand the XML Schema vocabulary and the elements, the example discusses an XML document that will be validated against a schema.

Figure 4.5 depicts schema declaration.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
4     targetNamespace="http://www.abc.com"
5     xmlns="http://www.abc.com"
6     elementFormDefault="qualified">
7     <xs:element name="Mail">
8        <xs:complexType>
9           <xs:sequence>
10             <xs:element name="To" type="xs:string"/>
11             <xs:element name="From" type="xs:string"/>
12             <xs:element name="Header" type="xs:string"/>
13             <xs:element name="Body" type="xs:string"/>
14          </xs:sequence>
15       </xs:complexType>
16    </xs:element>
17 </xs:schema>
```

Figure 4.5: Schema Declaration

Explanation for highlighted areas in the image:

`xs:schema xmlns:xs=http://www.w3.org/2001/XMLSchema`

Every XML Schema starts with the root element `<schema>`. The code indicates that that the elements and data types used in the schema are derived from the "`http://www.w3.org/2001/XMLSchema`" namespace and prefixed with `xs`.

`targetNamespace=http://www.abc.com`

It specifies that the elements defined in an XML document that refers this schema, come from the "`http://www.abc.com`" namespace.

`xmlns=http://www.abc.com`

This line of code points to "`http://www.abc.com`" as the default namespace.

`elementFormDefault="qualified"`

It indicates that elements used by the XML instance document which were declared in this schema needs to be qualified by the namespace.

Figure 4.6 illustrates using of schema declaration.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <Mail xmlns="http://www.abc.com"
4     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5     xsi:schemaLocation="http://www.abc.com mail_schema.xsd">
6     <To>John</To>
7     <From>Jordan</From>
8     <Header>Scheduler</Header>
9     <Body>3rd March Monday, 7:30 PM: board meeting!</Body>
10 </Mail>
```

Figure 4.6: Using Schema Declaration

Explanation for highlighted areas in the image:

`xmlns="http://www.abc.com"`

It indicates the default namespace declaration. This declaration informs the schema-validator that all the elements used in this XML document are declared in the default ("`http://www.abc.com`") namespace.

`xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"`

This is the instance namespace available for the XML schema.

`xsi:schemaLocation="http://www.abc.com mail.xsd">`

This `schemaLocation` attribute has two values. The first value identifies the namespace. The second value is the location of the XML schema where it is stored.

Concepts

Knowledge Check 2

1. Can you match the xml data against their corresponding data type?

Description		Features	
(A)	`<prize disabled="true">999</prize>`	**(1)**	`dateTime`
(B)	`<img src="http://www.abc.com/images/flowers.gif"/>`	**(2)**	`boolean`
(C)	`<start>09:30:10.5</start>`	**(3)**	`decimal`
(D)	`<start>2002-09-24</start>`	**(4)**	`time`
(E)	`<prize>+999.5450</prize>`	**(5)**	`anyURI`

2. The given is an instance of XML document referencing an XML schema. Can you arrange the XML file in its proper order using the schema vocabulary?

(1)	`<heading>Scheduler</heading>` `<body>3rd March Monday, 7:30 PM: board meeting!</body>`
(2)	`<to>John</to>` `<from>Jordan</from>`
(3)	`</mail>`
(4)	`<?xml version="1.0"?>` `<mail xmlns="http://www.abc.com"`
(5)	`xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"` `xsi:schemaLocation="http://www.abc.com mail.xsd">`

4.3 Working with Complex Types

In this third lesson, **Working with Complex types**, you will learn to:

- Describe complex type elements.
- Describe `minOccurs` and `maxOccurs`.
- Explain element content and mixed content.
- Describe how grouping can be done.

4.3.1 Complex Types

A schema assigns a type to each element and attribute that it declares. Elements with complex type may contain nested elements and have attributes. Only elements can contain complex types. Complex type elements have four variations.

- **Empty elements**

  Empty elements optionally specify attributes types, but do not permit content as shown in the following example.

  Code Snippet:

```
<xs:element name="Books">
  <xs:complexType>
    <xs:attributename="BookCode"
     type="xs:positiveInteger"/>
  </xs:complexType>
</xs:element>
```

- **Only Elements**

  These elements can only contain elements and do not contain attributes as shown in the following example.

  Code Snippet:

```
<xs:element name="Books">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="ISBN" type="xs:string"/>
      <xs:element name="Price" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

- **Only Text**

These elements can only contain text and optionally may or may not have attributes as shown in the following example.

Code Snippet:

```
<xs:complexType name="Books">
    <xs:simpleContent>
        <xs:extension base="xs:string">
            <xs:attribute name="BookCode" type="xs: positiveInteger"/>
        </xs:extension>
    </xs:simpleContent>
</xs:complexType>
```

- **Mixed**

These are elements that can contain text content as well as sub-elements within the element as shown in the following example. They may or may not have attributes.

Code Snippet:

```
<xs:element name="Books">
  <xs:complexType mixed="true">
    <xs:sequence>
      <xs:element name="BookName" type="xs:string"/>
      <xs:element name="ISBN" type="xs:positiveInteger"/>
      <xs:element name="Price" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

4.3.2 Defining Complex Types

A complex element can be defined in two different ways. The following codes define the properties of complex element.

- **By directly naming the element**

  Code Snippet:

```
<xs:element name="Student">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="FirstName" type="xs:string"/>
      <xs:element name="MiddleName" type="xs:string"/>
      <xs:element name="LastName" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

  The element `Student` can be declared by directly mentioning it in the schema. The complex type mentions that the XML document contains nested XML elements. The sequence element indicates that the child elements `FirstName`, `MiddleName` and `LastName` appear in the same order as they are declared.

- **By using the name and type attribute of the complex type**

  Code Snippet:

```
<xs:element name="Student" type="PersonInfo"/>
<xs:complexType name="personinfo">
  <xs:sequence>
    <xs:element name="FirstName" type="xs:string"/>
    <xs:element name="LastName" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
```

  The `Student` element can have a type such as `PersonInfo` that refers to the name of the complex type. Several elements can reuse this complex type by referring to the name `PersonInfo` in their complex type declarations.

Concepts

4.3.3 minOccurs and maxOccurs

When working with DTDs, we used the markers *, ?, and + to indicate the number of times a particular child element could be used as content for an element. Similarly, XML schema allows specifying a minimum and maximum number of times an element can occur. In schemas, both elements and attributes use the following attributes:

- **minOccurs**

  `minOccurs` specify the minimum number of occurrences of the element in an XML document. The default value for the `minOccurs` attribute is 1. If an element has a `minOccurs` value of 0, it is optional. An attribute is optional, if the `minOccurs` is 0. If minOccurs is set to 1, the attribute is required.

- **maxOccurs**

  `maxOccurs` specify the maximum number of occurrences of the element in an XML document. The default value for the `maxOccurs` attribute is 1. If its value is kept unbounded, it means that the element can appear unlimited number of times. The `maxOccurs` attribute defaults to 1 unless it is specified.

The following code demonstrates the use of `minOccurs` and `maxOccurs` attributes.

Code Snippet:

```
…
<xs:element name= "Books">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="ISBN" type="xs:string"/>
      <xs:element name="Quantity" type="xs:string"
      maxOccurs="100" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
…
```

The example demonstrates that the `Quantity` element can occur a minimum of zero times and a maximum of hundred times in the `Books` element.

The relationship between the `minOccurs` and `maxOccurs` attributes is displayed in table 4.1.

minOccur	MaxOccur	Number of times an element can occur
0	1	0 or 1
1	1	1
0	*	Infinite
1	*	At least once
>0	*	At least minOccurs times
>maxOccurs	>0	0
Any value	<minOccurs	0

Table 4.1: minOccurs and maxOccurs

4.3.4 Element Content

A complex type can mention element content. The `Element` content can contain only elements. There can be instances wherein the contained elements can also have child elements.

The following code demonstrates the properties of element content.

Code Snippet:

Books.xml

```
<?xml version="1.0"?>
<Books xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation= "Books.xsd">
<Title>A cup of tea</Title>
<Author>
<Name>Dennis Compton</Name>
</Author>
<Author>
<Name>George Ford</Name>
</Author>
<Publisher>
<Name>Orange</Name>
</Publisher>
</Books>
```

This is an XML schema constrained document depicting book details.

Books.xsd

```
<?xml version="1.0"?>
 <xs:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xs:element name= "Books" type= "BookType">
     <xs:complexType name= "AuthorType">
       <xs:sequence>
        <xs:element name= "Name" type="xs:string"/>
       </xs:sequence>
     </xs:complexType>
       <xs:complexType name= "PublisherType">
        <xs:sequence>
         <xs:element name= "Name" type= "xs:string"/>
          </xs:sequence>
        </xs:complexType>
     <xs:complexType name= "BookType">
       <xs:sequence>
        <xs:element name= "Title" type= "xs:string"/>
        <xs:element name= "Author" type="ComposerType"
maxOccurs= "unbounded"/>
        <xs:element name= "Publisher" type="PublisherType"
minOccurs="0" maxOccurs= "unbounded"/>
```

In the XML document the `Author` and the `Publisher` elements each contain `Name` elements. So built-in data types like `xs:string` cannot be used, instead `AuthorName` and `PublisherName` can be defined using top level `xs:complexType` elements.

4.3.5 Mixed Content

A complex type can specify its content as a mixed content. Mixed content can contain text mixed with elements. The order and the number of elements that appears in the mixed content can also be specified in the schema. This type of content may or may not have attributes.

The mixed content is declared in exactly the same way, the declaration of element content is done. The only add-on to the declaration is, it takes a mixed attribute set to the true value.

The following code demonstrates the properties of mixed content.

Code Snippet:

Book.xml

```
<Books>
Apocalypse written by<Author>Mary Jane</Author>
is of Genre<Category>Fiction</Type>.
</Books>
```

Book.xsd

```
<xs:element name="Books">
  <xs:complexType mixed="true">
    <xs:sequence>
      <xs:element name="Author" type="xs:string"/>
      <xs:element name="Category" type="xs:string"/>
      </xs:sequence>
  </xs:complexType>
</xs:element>
```

4.3.6 Grouping Constructs

The XML Schema Language provides three grouping constructs that specify whether and how ordering of individual elements is important:

- **`xs:all`**

  This grouping construct requires that each element in the group must occur at most once, but that order is not important. The only caution is that in this type of grouping the `minOccurs` attribute can be 0 or 1 and the `maxOccurs` attribute has to be 1. The following code demonstrates this concept.

  Code Snippet:

```
<xs:element name= "Books">
  <xs:complexType>
    <xs:all>
<xs:element name="Name" type="xs:string" minOccurs= "1" maxOccurs= "1"/>
<xs:element name="ISBN" type="xs:string" minOccurs= "1" maxOccurs= "1"/>
     <xs:element name="items"  type="Items" minOccurs= "1" />
     </xs:all>
  </xs:complexType>
</xs:element>
```

- **xs:choice**

The `choice` element is the opposite of all elements. Instead of requiring all the elements to be present, it will allow only one of the choices to appear. The `choice` element provides an XML representation for describing a selection from a set of element types. The `choice` element itself can have `minOccurs` and `maxOccurs` attributes that establish exactly how many selections may be made from the choice. The following code demonstrates this concept.

Code Snippet:

```
<xs:complexType name="AdressInfo">
<xs:group>
     <xs:choice>
    <xs:element name="Address" type="USAddress" />
    <xs:element name="Address" type="UKAddress" />
    <xs:element name="Address" type="FranceAddress" />
  </xs:choice>
</xs:group>
```

In the code snippet, there are three child elements that are mutually exclusive. With the `xs:choice` element declared, only one element among the choices can be a child element of the parent element `AddressInfo`.

- **xs:sequence**

An `xs:sequence` element specifies each member of the sequence to appear in the same order in the instance document as mentioned in the `xs:sequence` element. The number of times each element is allowed to appear can be controlled by the element's `minOccurs` and `maxOccurs` attributes. The following code demonstrates this concept.

Code Snippet:

```
<xs:element name="Books">
<xs:complexType>
<xs:sequence>
<xs:element name="Name" type="xs:string" />
<xs:element name="ISBN" type=" xs:string " />
<xs:element name="Price" type=" xs:string " />
</xs:sequence>
</xs:complexType>
</xs:element>
```

Knowledge Check 3

1. Which of these statements about complex types are true and which statements are false?

(A)	The order and the number of elements that appears in the mixed content cannot be specified in the schema.
(B)	If the value of `maxOccurs` attribute is kept unbounded, it means that the element can appear unlimited number of times.
(C)	Elements with complex type may contain nested elements and have attributes.
(D)	The default value for the `minOccurs` attribute is 0.
(E)	When a `minOccurs` attribute is used, there cannot be a `maxOccurs` attribute in the same line.

2. Which of these statements about element and mixed content are true and which statements are false?

(A)	Mixed content means that an element whose structure is the complex type can contains elements with attributes.
(B)	Element content means a complex type element that contains only elements.
(C)	The order and the number of elements appearing in the mixed content cannot be specified in schemas.
(D)	Element content cannot have attributes.

3. Which of these statements about grouping are true and which statements are false?

(A)	The `sequence` element provides an XML representation for describing a selection from a set of element types.
(B)	The `all` element requires that each element in the group must occur at most once.
(C)	For each element type associated with a `sequence` element, there must be an element in the XML instance in the same order.
(D)	The `choice` element cannot mention the `minOccurs` and `maxOccurs` attribute.

Concepts

4.4 Working with Simple Types

In this last lesson, **Working with Simple Types**, you will learn to:

- Describe simple types.
- List and describe the data types used with simple types.
- Explain restrictions and facets.
- Identify the usage of attributes.

4.4.1 Defining a Simple Type Element

A simple type is an XML element or attribute, which contains only text and no other elements or attributes.

The simple type declarations are used to form the textual data and specify the type of data allowed within attributes and elements.

Syntax:

```
<xs:element name="XXXX" type="YYYY"/>
```

In this syntax, '`XXXX`' is the name of the element, '`YYYY`' is the data type of the element.

The following code demonstrates that the element `TotalNoOfPages` can be specified as an integer type, which is again a positive integer with three digits.

Code Snippet:

`Book.xml`: **XML Elements**

```
<Book_name>The Da vinci code</Book_name>
<TotalNoOfPages>360</TotalNoOfPages>
<Author_name>Dan Brown</Author_name>
```

`Book.xsd`: **Corresponding Simple Element Definitions**

```
<xs:element name="Book_name" type="xs:string" />
<xs:element name="TotalNoOfPages" type="xs:integer"/>
<xs:element name="Author_name"  type="xs:string"/>
```

4.4.2 Datatypes Used with Simple Types

Elements of simple type tend to describe the content and the datatype of a document, rather than its structure.

In XML Schema, one can mention the type of data an element can contain by assigning it a particular simple type definition. So, based upon the type of data it supports, XML schema divides the elements of simple types into two broad categories:

- built-in simple type
- user-defined simple type

4.4.3 Built-in Simple Types

There are several built-in simple types, such as integer, date, float and string that one can use without further modifications.

A built-in simple element can contain a default value or a fixed value. A 'default' value is the value that is assigned automatically to the element when no other value has been specified. A 'fixed' value is assigned to an element, when there is no need to change the value for that element. Figure 4.7 depicts built-in simple types.

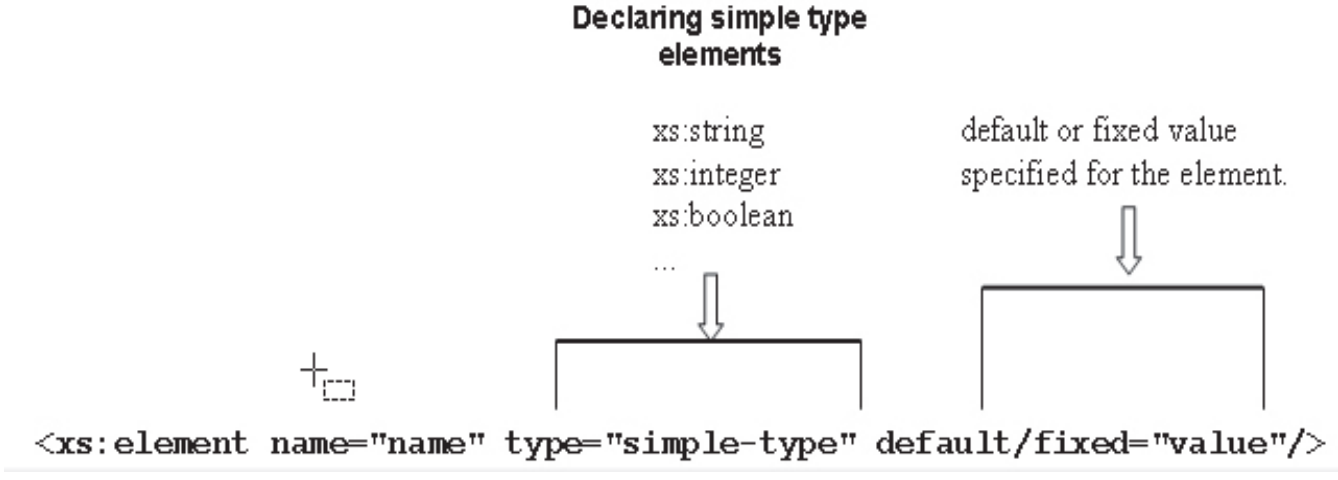


Figure 4.7: Built-in Simple Types

Syntax:

```
<xs:element name="XXXX" type="YYYY" default="ZZZZ"/>
```

In this syntax, '`XXXX`' is the name of the element, '`YYYY`' is the data type of the element and `ZZZZ` is the default value specified for the element.

The following code demonstrates the declaration of built-in simple type.

Code Snippet:

built-in simple type: **element declaration**

```
<xs:element name="AccountType" type="xs:string" fixed="Savings"/>
<xs:element name="BalanceAmount" type="xs:integer" default="5000"/>
```

The code shows the declaration of built-in simple types "`AccountType`" and "`BalanceAmount`". The fixed value for the "`AccountType`" is "`Savings`" and the default value for "`BalanceAmount`" is "`5000`".

4.4.4 User-defined Simple Types

XML-schema supports a library of built-in datatypes. However, during the course of writing complex schema documents, a developer may need a type of data that is not defined in the schema recommendation. For instance, consider the declaration of an element `<AngleMeasure>`. It implements the available restrictions to its content to accept non-negative numeric values, but still it accepts unwanted values like:

```
<AngleMeasure>490</AngleMeasure>
```

As per schema declaration, "490" is a valid non-negative numeric value, but still it needs numeric type that allows the values with a range from 0 to 360. Hence, a custom user defined datatype can be created using the `<simpleType>` definition.

Syntax:

```
<xs:simpleType name="name of the simpleType">
<xs:restriction base="built-in data type">
  <xs:constraint="set constraint to limit the content"/>
</xsd:restriction>
</xsd:simpleType>
```

The following codes demonstrate the properties of user-defined simple types.

Code Snippet:

```
<xs:simpleType name="AngleMeasure">
  <xs:restriction base="xs:integer">
    <xs:minInclusive value="0"/>
    <xs:maxInclusive value="360"/>
  </xs:restriction>
</xs:simpleType>
```

- This creates a new datatype called "`AngleMeasure`".
- Elements of this type can hold integer values.
- The value of "`AngleMeasure`" must be within the range of "0" to "360".

Code Snippet:

```
<xs:simpleType name="triangle">
    <xs:restriction base="xsd:string">
        <xs:enumeration value="isosceles"/>
        <xs:enumeration value="right-angled"/>
        <xs:enumeration value="equilateral"/>
    </xs:restriction>
</xs:simpleType>
```

- This creates a new datatype called "`triangle`".
- Elements of this type can hold elements of triangle type.
- Elements of this type can have either the value "`isosceles`", or "`right-angled`", or "`equilateral`".

Concepts

4.4.5 Restrictions

Declaration of a data type puts certain limitations on the content of an XML element or attribute. If an XML element is of type "`xs:integer`" and contains a string like "Welcome", the element will not be validated. These limitations are called restrictions, which defines allowable values for XML elements and attributes.

Restrictions can be specified for the simpleType elements and restriction types are declared using the `<restriction>` declaration. Basically, the `<restriction>` declaration is used to declare a derived simpleType, which is a subset of its base simpleType. The value of the base attribute can be any existing simpleType, or built-in XML Schema datatype.

Syntax:

```
<restriction base="name of the simpleType you are deriving from">
```

In this `<restriction>` declaration, the `base` data type can be specified using the `base` attribute.

The following code demonstrates a simpleType "`Age`" is derived by specifying the base as integer type.

Code Snippet:

```
<xs:simpleType name="Age">
    <xs:restriction base="xs:integer">
     ...
     ...
  </xs:restriction>
</xs:simpleType>
```

4.4.6 Facets

With XML Schemas, custom restrictions can be specified on XML elements and attributes. These restrictions are called facets.

Facets are used to restrict the set or range of values a datatype can contain. The value range defined by the facet must be equal to or narrower than the value range of the base type.

There are 12 facet elements, declared using a common syntax. They each have a compulsory value attribute that indicates the value for the facet. One restriction can contain more than one facet. Any values appearing in the instance and value spaces must conform to all the listed facets.

Table 4.2 depicts the constraining facets.

Facet	Description
minExclusive	Specifies the minimum value for the type that excludes the value provided.
minInclusive	Specifies the minimum value for the type that includes the value provided.
maxExclusive	Specifies the maximum value for the type that excludes the value provided.
maxInclusive	Specifies the maximum value for the type that includes the value provided.
totalDigits	Specifies the total number of digits in a numeric type.
fractionDigits	Specifies the number of fractional digits in a numeric type.
length	Specifies the number of items in a list type or the number of characters in a string type.
minLength	Specifies the minimum number of items in a list type or the minimum number of characters in a string type.
maxLength	Specifies the maximum number of items in a list type or the maximum number of characters in a string type.
enumeration	Specifies an allowable value in an enumerated list.
whiteSpace	Specifies how whitespace should be treated within the type.
pattern	Restricts string types.

Table 4.2: Constraining Facets

Syntax:

```
<xs:simpleType name= "name">
    <xs:restriction base= "xs:source">
        <xs:facet value= "value"/>
        <xs:facet value= "value"/>
        …
    </xs:restriction>
</xs:simpleType>
```

The following code demonstrates that the `value` attribute gives the value of that facet.

Code Snippet:

```
<xs:simpleType name="triangle">
    <xs:restriction base="xs:string">
        <xs:enumeration value="isosceles"/>
        <xs:enumeration value="right-angled"/>
        <xs:enumeration value="equilateral"/>
    </xs:restriction>
</xs:simpleType>
```

Here, the facet enumeration is added to the restriction with the value attribute as either `isosceles`, or `right-angled`, or `equilateral`. So, an element declared to be of type triangle must be a string with a value of either `isosceles`, or `right-angled`, or `equilateral`.

4.4.7 Attributes

XML elements can contain attributes that describe elements. Within XML Schemas, attribute declarations are similar to element declarations. To declare an attribute, `<xs:attribute>` element is used.

An attribute can be indicated whether it is required or optional or whether it has a default value. A default value is automatically assigned to the attribute when no other value is specified.

The use attribute specifies whether the attribute is required or optional. Here are the different values that can be assigned to the attributes:

- **`Default`**

  A `default` value is automatically assigned to the attribute when no other value is specified. For example,

  ```
  <xs:attribute name="genre" type="xs:string" default="fiction"/>
  ```

  In this code, the `default` value of the attribute genre is fiction.

- **Fixed**

  This value makes the attribute fixed. A `fixed` value is automatically assigned to the attribute, and another value cannot be specified. For example,

  ```
  <xs:attribute name="genre" type="xs:string" fixed="fiction"/>
  ```

  In this code, `fixed` value of **fiction** is assigned to the attribute **genre**, so another value for the attribute cannot be specified.

- **Optional**

  This value makes the attribute optional, which means that the attribute may have any value. The default value for any an attribute is optional. For example,

  ```
  xs:attribute name="genre" type="xs:string" use="optional"/>
  ```

  In this code, the attribute **genre** can take any string value.

- **Prohibited**

  This value means that the attribute cannot be used. For example,

  ```
  xs:attribute name="genre" type="xs:string" use="prohibited"/>
  ```

  In this line of code, the element instance will not have the attribute **genre**.

- **Required**

  This value makes the attribute required. The attribute can have any value. For example,

  ```
  xs:attribute name="genre" type="xs:string" use="required"/>
  ```

In this line of code, the attribute **genre** has to be used in the XML element declaration.

Figure 4.8 depicts the attribute declaration.

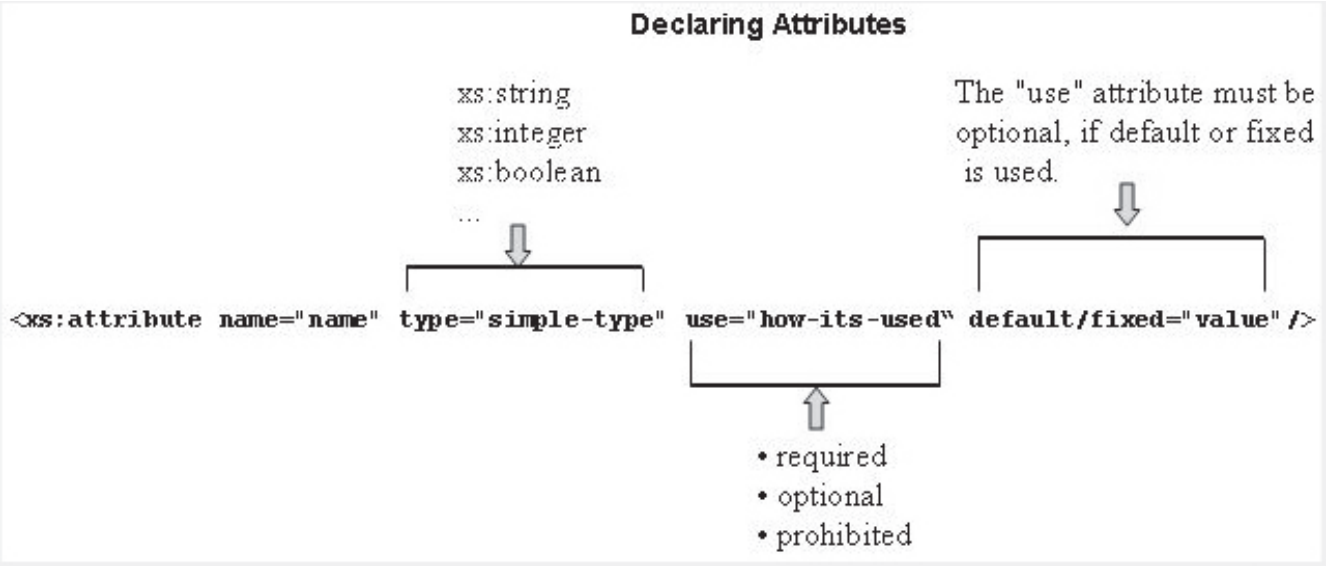


Figure 4.8: Attributes

Syntax:

The syntax for defining an attribute is:

```
<xs:attribute name="Attribute_name" type="Attribute_datatype"/>
```

where,

`Attribute_name` is the name of the attribute.

`Attribute_datatype` specifies the data type of the attribute. There are lot of built in data types in XML schema such as string, decimal, integer, boolean, date, and time.

The following code demonstrates the properties of attribute.

Code Snippet:

In the following example, `xs:attribute` elements comes after `xs:sequence` and `xs:group` that forms the body of the element. The element `Name` can have an optional attribute named `age` with type positiveInteger.

```
....
<xs:complexType name= "SingerType">
 <xs:sequence>
  <xs:element name= "Name">
   <xs:complexType>
    <xs:all>
     <xs:element name= "FirstName" type="xs:string"/>
     <xs:element name= "LastName" type="xs:string"/>
    </xs:all>
   </xs:complexType>
  </xs:element>
</xs:sequence>
<xs:attribute name= "age" type="xs:positiveInteger" use= "optional/>
</xs:complexType>
....
```

Knowledge Check 4

1. Which of the following statements about simple type elements are true and which statements are false?

(A)	A custom user defined datatype can be created using the `<simpleType>` definition.
(B)	Elements of simple type describe the content and data type of an element.
(C)	Elements of simple type constitute the structure of an XML document.
(D)	A built-in simple element can contain a default value or a facet value.
(E)	A `default` value is the value that is assigned automatically to the element when there is no other value specified.

2. Can you match the different keywords against their corresponding description?

Description		Term	
(A)	Specifies the number of digits after decimal point	**(1)**	`pattern`
(B)	Restricts string types using regular expressions	**(2)**	`use`
(C)	Specifies an allowable value in an enumerated list	**(3)**	`prohibited`
(D)	Specifies whether the attribute is `required` or `optional`	**(4)**	`fractionDigits`
(E)	Specifies that the attribute cannot be used	**(5)**	`enumeration`

Module Summary

In this module, **XML Schema**, you learnt about:

- **XML Schema**

  An XML Schema is an XML-based alternative to DTDs, which describes the structure of an XML document. The XML Schema language is also referred to as XML Schema Definition (XSD). An XML Schema can define elements, attributes, child elements and the possible values that can appear in a document. Schemas overcome the limitations of DTDs and allow Web applications to exchange XML data robustly, without relying on ad hoc validation tools.

- **Exploring XML Schemas**

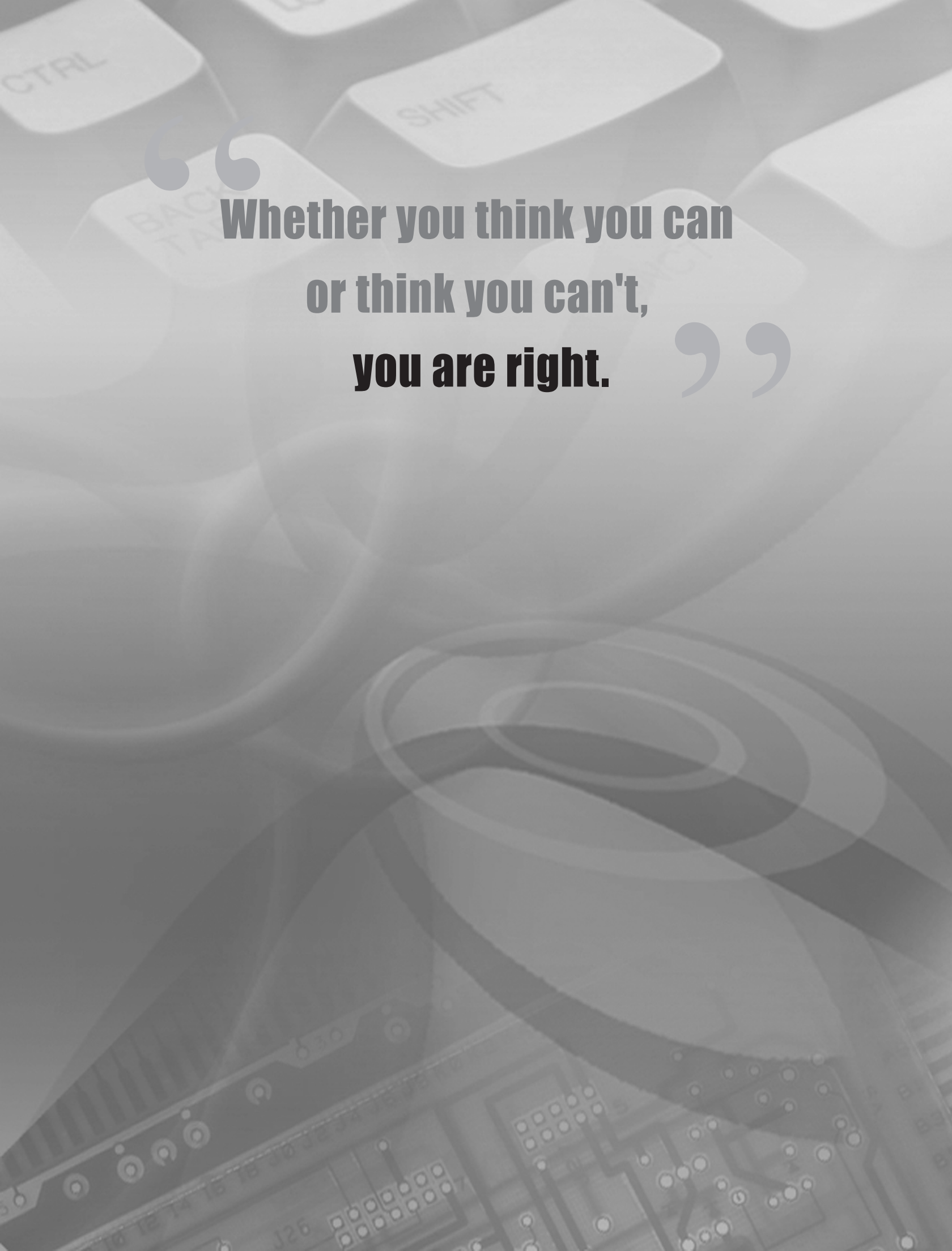
  XML schema offers a range of data types. It supports built-in data types like string, boolean, number, dateTime, binary and uri. Additional data types like integer, decimal, real, time, timeperiod and user defined data types.

- **Working with Complex Type**

  Elements with complex type may contain nested elements and have attributes. A complex element can be defined by directly naming the element and by using the name and the type attribute of the complex type. `minOccurs` specifies the minimum number of occurrences of the element in an XML document. `maxOccurs` specifies the maximum number of occurrences of the element in an XML document. Element content in an XML document contains only XML elements and mixed content contains text mixed with elements. The grouping constructs in XML schema specify the order of XML elements.

- **Working with simple types**

  The elements of simple type describe the content and data type of the element rather than its structure. The simple type can have built-in and user defined data types. Simple type definition takes one of the two values, default or fixed as per the requirement. The user-defined data type can be derived from a built in type or an existing simple type. Use of restrictions and facets restricts its content to user prescribed values. Schema supports usage of attributes in elements, which is declared with a simple type definition.


Whether you think you can
or think you can't,
you are right.

Module

5 Style Sheets

Concepts

Module Overview

Welcome to the module, **Style Sheets**. This module introduces you to style sheets. It also discusses how to use Cascading Style Sheets to format XML document. Finally, the module explains the cascading order of style rules.

In this module, you will learn about:

- Style Sheets
- Selectors in CSS
- Properties and Values
- Inheritance and Cascades in CSS

5.1 Style Sheets

In this first lesson, **Style Sheets**, you will learn to:

- Define and describe style sheets.
- Define and describe cascading style sheets (CSS).
- Explain how to implement styles with CSS.

5.1.1 Need of Style Sheets

Style sheets are a set of rules that describe the appearance of data in an XML document.

XML was inspired by the problems posed by presentational markup. Presentational markup does not describe data; it defines the appearance of data. In a document, if you had to change all proper nouns from bold to italics, you would have to do it manually for each proper noun. Presentation markup failed to provide same look and feel across multiple devices such as computers, Personal Digital Assistant (PDA) devices, and cell phones.

Style sheets and XML solve these problems. XML describes data. Style sheets define the appearance of data. However, both XML and style sheets are defined in separate files.

Figure 5.1 depicts formatted document with style sheet.

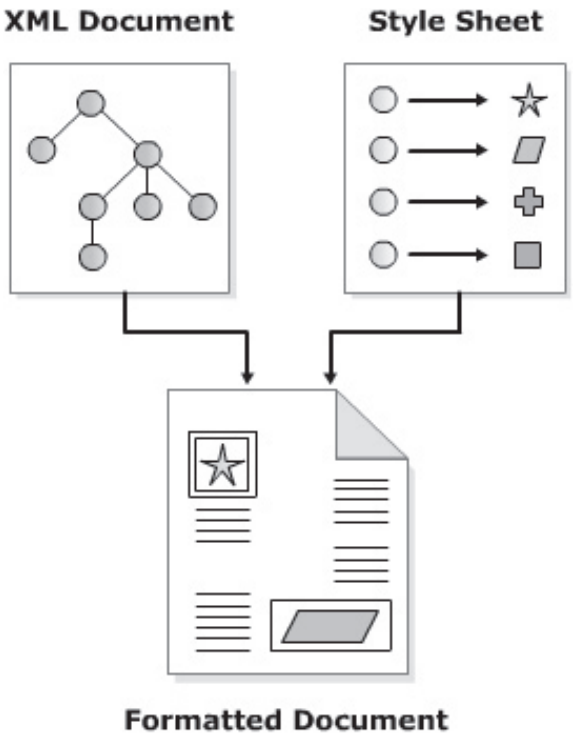


Figure 5.1: Formatting Document with Style Sheet

5.1.2 Various Style Sheets

There are several style sheets available. However, following two are the most popular style sheets:

- **Cascading Style Sheet (CSS)**

  CSS allows you to control the appearance of data in HTML and XML documents by providing various properties to define:

  - Position and size of data to be displayed
  - Foreground and background color of data
  - Font to be used to display data
  - Spacing between data

- **Extensible Style Sheet (XSL)**

  XSL is a style sheet language used to define the appearance of data contained only in XML documents. However, it also allows you to transform XML documents.

5.1.3 Cascading Style Sheets

Cascading style sheets is a rule-based language invented in 1996 to write formatting instructions for data contained in HTML documents.

A CSS style sheet comprises a set of rules for various elements in a document. Each rule defines how the data enclosed in the element should be displayed.

Style sheet rules can be combined from different sources, or subsets can be created or the rules can be overridden. The term cascading is derived from this ability to mix and match rules.

Figure 5.2 shows an example of cascading style sheet.

XML Document

```
<Library>
  <Films>
    <Film>
     <Name>Ghost Rider</Name>
     <Sound>Yes</Sound>
        <Year>2007</Year>
     <Cast>Nicolas Cage and others</Cast>
    <Film>
  </Films>
</Library>
```

Style Sheet

```
Name { font-style: bold }
Cast { color: aqua }
```

Figure 5.2: Cascading Style Sheet

Concepts

5.1.4 Benefits of CSS

Some of the benefits of cascading style sheets are:

- Any style or presentation changes to data can be achieved faster as changes are to be implemented in one place.
- Device independence can be achieved by defining different style sheets for different device. For example, you can have different style sheet for desktop computers, PDAs, and cell phones.
- Reduction in document code as the presentation information is stored in a different file and can be reused.

Note: CSS is supported by most browsers available today. Some of these browsers are Netscape (6.0 or higher), Mozilla, Opera (4.0 or higher), and Internet Explorer (5.0 or higher). However, these browsers support only parts of CSS specification. The current CSS specification is CSS 2. CSS 3 specification is under development.

5.1.5 Style Rules

A style sheet in CSS comprises a set of style rules. These rules define how the content enclosed in an element will be displayed. These rules are applicable to all child elements within an element. A style rule comprises a selector, a property, and a value. A property and a value separated with a colon are referred as property declaration. Figure 5.3 depicts the style rules.

Figure 5.3: Style Rules

- **selector**

  Selector is an element name of an XML document. A typical element name could be CD, Name, or Title.

- **property**

  Property is a CSS style property that defines how the element will be rendered. Some of the CSS properties are border, font, and color.

- **value**

  Value is the value associated with a CSS property. One CSS property can have several values. The various values for property font-family are the various font family names such as "times", "arial", and "courier" to name a few.

In Cascading Style Sheets (CSS), style rules can comprise more than one selector. To include multiple selectors or group multiple selectors, one needs to provide a comma-separated list of element names. The following code depicts a sample XML document containing information about endangered species.

Code Snippet:

```
<?xml version="1.0" ?>
<Endangered_Species>
  <Animal>
    <Name language="English">Tiger</Name>
    <Threat>poachers</Threat>
    <Weight>500 pounds</Weight>
  </Animal>
</Endangered_Species>
```

The following code demonstrates an ideal way to define style rules by displaying each element on a separate row.

Code Snippet:

```
Name { display: block }
Threat { display: block }
Weight { display: block }
```

However, these three style rules can be converted to a single style rule by grouping the selectors as demonstrated in the following code.

Code Snippet:

```
Name, Threat, Weight { display: block }
```

Concepts

5.1.6 Ways of Writing Style Rules

A single selector can have more than one property-value pairs associated with it. For example, figure 5.4 shows a CD element having two property declarations – one to set the font family to sans-serif, and other to set the color of text to black. Notice the property-value pairs are separated by a semi-colon.

Similarly, a collection of one or more property-value pairs can be associated with more than one selector. For example, figure 5.4 shows two property declarations assigned to three elements namely CD, Title, and Name.

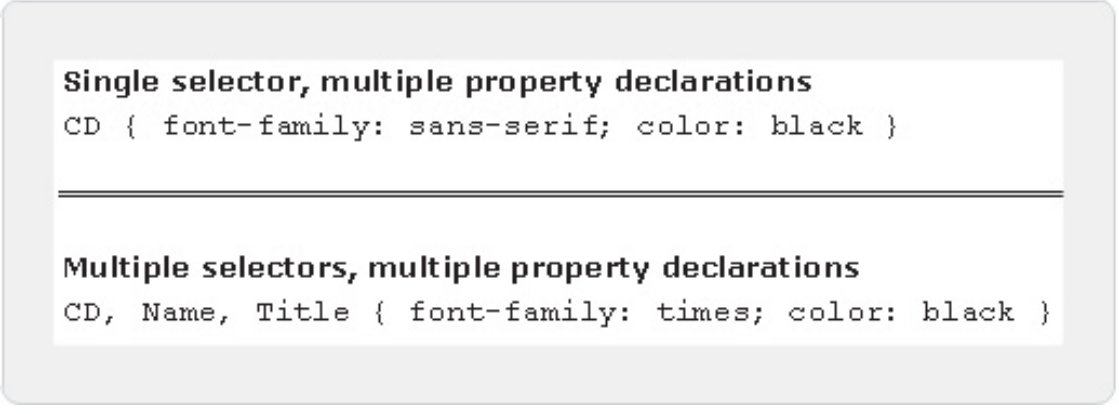


Figure 5.4: Writing Styles

5.1.7 External Style Sheets

In CSS, the style rules are written in a file with the extension `.css`. This file is associated with an XML document using a style sheet processing instruction. A few points to note about style sheet processing instruction are:

- It should appear in the prolog section of an XML document, that is, it should appear before any tag of an element.
- One XML document can have more than one style sheet processing instructions, each linked to one `.css` file.

Syntax:

```
<?xml-stylesheet href="url" [type="text/css"]?>
```

where,

`xml-stylesheet` is the processing instruction.

`url` is the URL of a `.css` file; the `.css` file can on a local system or anywhere on the Internet.

`type="text/css"` is optional; however if a browser does not support CSS, it informs the browser that it does not have to download a `.css` file.

The following code demonstrates an example of external style sheet.

Code Snippet:

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

Here the style rules are defined in a file named `headers.css`.

Knowledge Check 1

1. Which of the following statements about style sheets are true and which are false?

(A)	Cascading Style Sheets derived the term cascade from the ability to mix and match rules from different sources.
(B)	Cascading Style Sheets lack support to define spacing between data.
(C)	A CSS style sheet is associated with an XML document using the processing instruction `xml-stylesheet`.
(D)	Style sheets allow you to mix presentation markup with data.
(E)	Style sheets contain one or more rules about the appearance of data.

5.2 Selectors in CSS

In this second lesson, **Selectors in CSS**, you will learn to:

- Identify simple selectors in CSS.
- State the use of universal selector in CSS.
- Describe ID selectors.

5.2.1 Simple Selectors

Simple selector comprises an element name followed by one or more property declarations. Same property declarations can be assigned to multiple elements by separating element names with a comma. Simple selectors match every occurrence of the element in a document.

The following code demonstrates an example of simple selector.

Code Snippet:

```
/* Simple selector */
CD { color: black }
/* Single element, multiple property declarations */
CD { color: white; background-color: blue }
/* Multiple elements, multiple property declarations */
CD, Name, Title { color: white; background-color: blue }
```

5.2.2 Universal Selector

A universal selector comprises an asterisk followed by property declarations. It is used when you want to assign the same style rules to all the elements in a document. A universal selector matches all the elements in a document. The following code displays the content of all elements in a document in blue.

Code Snippet:

```
* { color : blue }
```

5.2.3 ID Selector

An ID selector comprises a hash (#) symbol, immediately followed by an attribute's value followed by property declarations. It is used to define styles for unique elements in a document. For example, if you want the data of a unique element to be in a different style, you would define an ID selector for it. Unique element is one which has one of its attributes named as id as shown in figure 5.5.

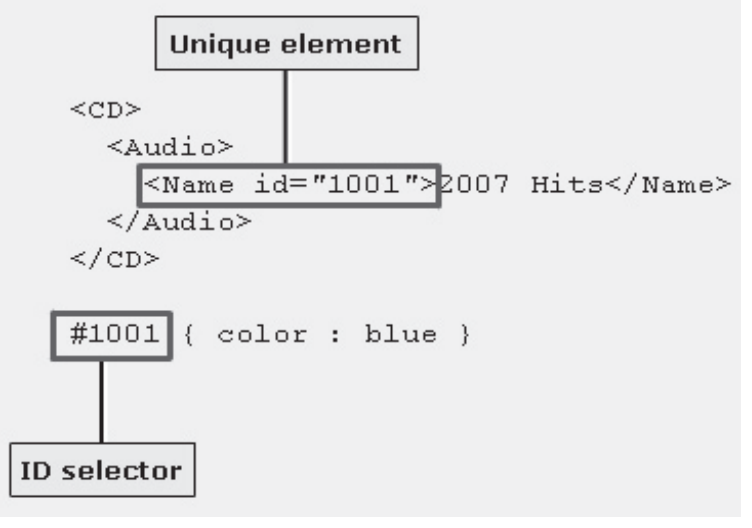


Figure 5.5: ID Selector

Syntax:

```
#attribute_value { property_declarations }
```

The following code demonstrates id attribute's value.

Code Snippet:

```
#1001 { color : blue }
```

Displays the content of an element in blue if its id attribute's value equals 1001.

ID selectors are used to emphasize the data contained in unique elements in an XML document. However, not all browsers support ID selectors. A browser has to read the document type definition (DTD) to identify which attributes have an ID type. Browsers such as Safari, Mozilla, and Netscape do not read external DTD subsets. Opera lacks the ability to read internal DTDs as well. Hence, these browsers may not apply the style rules involving ID selectors. Internet Explorer does support ID selectors as it reads external DTDs.

Knowledge Check 2

1. Which of the following statements about selectors are true and which are false?

(A)	Simple selectors match occurrence of an element based on a condition.
(B)	ID selectors are used to define style rules for unique elements.
(C)	In simple selectors you define same property declarations for multiple elements by specifying a comma separated list of element names.
(D)	ID selector match elements which have an id attribute.
(E)	Universal selectors match every occurrence of parent and child elements.

5.3 Properties and Values

In this third lesson, **Properties and Values**, you will learn to:

- State and describe how to use color properties.
- Describe the font property.
- Describe the other properties such as margins, borders, padding.
- Explain briefly about positioning and alignment.

Concepts

5.3.1 Color Properties

The CSS provides properties to set the foreground and background color of text. CSS uses color name, red-green-blue (RGB) values, RGB percentages and hexadecimal values to specify color values. The various ways in which color values are specified is shown in table 5.1.

Color Names	RGB Percentages	RGB Values	Hexadecimal Values
aqua	rgb(0%,65%,65)	rgb(0,160,160)	#00a0a0
black	rgb(0%,0%,0%)	rgb(0,0,0)	#000000
blue	rgb(0%,32%,100)	rgb(0,80,255)	#0050ff
gray	rgb(65%,65%,65%)	rgb(160,160,160)	#a0a0a0
green	rgb(0%,100%,0%)	rgb(0,255,0)	#00ff00
lime	rgb(0%,65%,0%)	rgb(0,160,0)	#00a000
maroon	rgb(70%,0%,32%)	rgb(176,0,80)	#b00050
navy	rgb(0%,0%,65%)	rgb(0,0,160)	#0000a0
olive	rgb(65%,65%,0%)	rgb(160,160,0)	#a0a000
purple	rgb(65%,0%,65%)	rgb(160,0,160)	#a000a0
red	rgb(100%,0%,32%)	rgb(255,0,80)	#ff0050
silver	rgb(90%,90%,90%)	rgb(225,225,255)	#d0d0d0
teal	rgb(0%,65%,100%)	rgb(0,160,255)	#00a0ff
white	rgb(100%,100%,100%)	rgb(255,255,255)	#ffffff
yellow	rgb(100%,100%,0%)	rgb(255,255,0)	#ffff00

Table 5.1: Color Values

5.3.2 Setting Color Properties

The CSS specification provides the properties `color` and `background-color` to set the foreground and background color of text enclosed in elements.

Figure 5.6 depicts the syntax for setting properties.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <?xml-stylesheet href="Colors.css" type="text/css"?>
3  <Cars>
4    <Vehicle>
```

Figure 5.6: Syntax for Setting Properties

where,

`color`

Property to set the foreground color of text in an element.

`colorValue`

colorValue can take up any value from the CSS color table.

`background-color`

Property to set the background color of text in an element.

Figure 5.7 shows the code for style rules defined in `Colors.css` file.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <?xml-stylesheet href="Colors.css" type="text/css"?>
3  <Cars>
4    <Vehicle>
5      <Manufacturer>BMW</Manufacturer>
6      <Model>M3</Model>
7      <Color>Metallic Silver</Color>
8      <Price>40,000</Price>
9      <Location>Liverpool</Location>
10   </Vehicle>
11 </Cars>
```

Figure 5.7: XML File

Figure 5.8 depicts the style sheet for **`Colors.css`** file.

```
1 Cars { background-color: rgb(0%,32%,100%); color: #ffffff }
2 Price { color: yellow; }
```

Figure 5.8: Colors.css File

where,

`Cars { background-color: rgb(0%,32%,100%); color: #ffffff }`

Causes the text enclosed in `Cars` element to be displayed in white color with a background color of blue.

`Price { color: yellow; }`

Causes the text enclosed in `Price` element to be displayed in yellow color.

The output is shown in figure 5.9.

Figure 5.9: Output of Using Color Properties

5.3.3 Font Properties

The CSS specification defines several font properties to set the font family, font size, font style such as bold, italics, and so forth. Some of these properties are listed in table 5.2.

Property Name	Description
`font-family`	To specify the font family
`font-size`	To specify the size of font
`font-style`	To specify the style of font
`font-weight`	To specify the weight of font

Table 5.2: Font Properties

5.3.4 font-family Property

The `font-family` property is used to specify the name of the font family to be applied to an element.

Figure 5.10 shows the syntax for font-family property.

```
font-family : "font-family name(s)"
```

Figure 5.10: Font-family Property

where,

`font-family`

Property to specify the font-family to be used.

`font-family name(s)`

Comma separated list of font-family names such as serif, san-serif, monospace, cursive, and fantasy. The list should start with the most specific font in which you want to display the data and end with the most generic font.

Figure 5.11 shows the code for style rules stored in `FontFamily.css` file.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <?xml-stylesheet href="FontFamily.css" type="text/css"?>
3  <Cars>
4    <Vehicle>
5      <Manufacturer>BMW</Manufacturer>
6      <Model>M3</Model>
7      <Color>Metallic Silver</Color>
8      <Price>40,000</Price>
9      <Location>Liverpool</Location>
10   </Vehicle>
11 </Cars>
```

Figure 5.11: XML File

Figure 5.12 depicts the style sheet for **`fontfamily.css`** file.

```
1 Cars { font-family: "arial, times, serif" }
```

Figure 5.12: FontFamily.css File

The output is shown in figure 5.13.

BMW M3 Metallic Silver 40,000 Liverpool

Figure 5.13: Output of Using font-family Property

5.3.5 font-size Property

The `font-size` property is used to specify the size of font used to display element data.

Figure 5.14 shows the syntax for `font-size` property.

```
font-size : "xx-small | x-small | small | medium | large | x-large | xx-large"
```

Figure 5.14: Syntax for font-size Property

where,

`font-size`

Property to specify the size of font.

`xx-small | x-small | small | medium | large | x-large | xx-large`

One of various values that can be assigned to the property font-size.

Figure 5.15 shows the code for style rules defined in `FontSize.css` file.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <?xml-stylesheet href="FontSize.css" type="text/css"?>
3  <Cars>
4    <Vehicle>
5      <Manufacturer>BMW</Manufacturer>
6      <Model>M3</Model>
7      <Color>Metallic Silver</Color>
8      <Price>40,000</Price>
9      <Location>Liverpool</Location>
10   </Vehicle>
11 </Cars>
```

Figure 5.15: XML File

Figure 5.16 depicts the style sheet for **`fontsize.css`** file.

```
1 Cars { font-size: medium }
```

Figure 5.16: FontSize.css File

where,

`Cars { font-size: medium }`

All the text enclosed in Cars element and its child elements will be displayed with medium font size.

The text enclosed in the element Cars will be displayed in either arial, times or serif font. The rule becomes applicable to all the elements enclosed in Cars element. If the system does not support arial font, then times will be selected; if times is not supported then serif will be selected.

The output is shown in figure 5.17.

Figure 5.17: Output of Using font-size Property

5.3.6 Font Style and Weight Properties

CSS provides two properties namely `font-style` and `font-weight` to add emphasis or meaning to parts of data.

Figure 5.18 shows the syntax for font style and weight properties.

```
font-style: normal | oblique | italic
font-weight: light | normal | bold
```

Figure 5.18: Font Style and Weight Properties

where,

`font-style`

Property to specify the style of text in an element.

`normal | oblique | italic`

One of the values that can be assigned to font-style property.

`font-weight`

Property to specify the weight style of the text in an element.

`light | normal | bold`

One of the values that can be assigned to font-weight property.

Figure 5.19 shows the code for style rules defined in `FontStyle.css` file.

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet href="FontStyle.css" type="text/css"?>
<Cars>
  <Vehicle>
    <Manufacturer>BMW</Manufacturer>
    <Model>M3</Model>
    <Color>Metallic Silver</Color>
    <Price>40,000</Price>
    <Location>Liverpool</Location>
  </Vehicle>
</Cars>
```

Figure 5.19: XML File

Figure 5.20 depicts the style sheet for **`fontstyle.css`** file.

```
Manufacturer { font-weight: bold }
Location { font-style: italic }
```

Figure 5.20: FontStyle.css File

where,

`Manufacturer { font-weight: bold }`

The text enclosed in Manufacturer element will be displayed in bold.

`Location { font-style: italic }`

The text enclosed in Location element will be displayed in italics.

The output is shown in figure 5.21.

Figure 5.21: Output of Using Font Styles

where,

`BMW`

Text is displayed in bold

`Liverpool`

Text is displayed in italics

5.3.7 Margins in CSS

Every element in an XML document is displayed in its own box. CSS provides four properties namely `margin-left`, `margin-right`, `margin-top` and `margin-bottom` to insert space around the element's box.

Figure 5.22 shows the syntax for margins in css.

```
margin-left | margin-right | margin-top | margin-bottom =
marginValue
```

Figure 5.22: Syntax for Margins in CSS

where,

`margin-left | margin-right | margin-top | margin-bottom`

The various margin properties to set left, right, top and bottom margins.

`marginValue`

The value to be assigned to one of the margin properties. This value can be a fixed value or a percentage.

Figure 5.23 shows the code for style rules defined in `Margin.css` file.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <?xml-stylesheet href="Margin.css" type="text/css"?>
3  <Cars>
4    <Vehicle>
5      <Manufacturer>BMW</Manufacturer>
6      <Model>M3</Model>
7      <Color>Metallic Silver</Color>
8      <Price>40,000</Price>
9      <Location>Liverpool</Location>
10   </Vehicle>
11 </Cars>
```

Figure 5.23: XML File

Figure 5.24 depicts the style sheet for `margin.css` file.

```
1 Manufacturer { margin-left: 20; margin-right: 50; }
2 Manufacturer { background-color: aqua }
3 Vehicle { background-color: orange }
```

Figure 5.24: Margin.css File

where,

```
Manufacturer { margin-left: 20; margin-right: 50; }
```

Inserts a space of 20 pixels to the left and a space of 50 pixels to the right of text enclosed in element Manufacturer.

```
Manufacturer { background-color: aqua }
```

Sets the background of text enclosed in element Manufacturer to aqua.

```
Vehicle { background-color: orange }
```

Sets the background of text enclosed in element Vehicle to orange.

The output is shown in figure 5.25.

Figure 5.25: Output of Using Margins

5.3.8 Border Properties in CSS

Borders are rectangular outlines surrounding an element's box. CSS provides properties to create dotted, solid and groove borders to name a few.

Figure 5.26 shows the syntax for border properties.

```
border : border_width border_style border_color
```

Figure 5.26: Syntax for Border Properties

where,

`border`

Property to set the border of the box surrounding an element's data.

`border_width`

Specifies the thickness of the border. Possible values are thin, medium and thick.

`border_style`

Specifies the style of the border. Possible values are solid, dashed, dotted, groove, ridge, double, inset, outset.

`border_color`

Specifies the color of the border. All values that are applicable to CSS color property are allowed.

Concepts

Figure 5.27 shows the code for style rules defined in `Border.css` file.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <?xml-stylesheet href="Border.css" type="text/css"?>
3  <Cars>
4    <Vehicle>
5      <Manufacturer>BMW</Manufacturer>
6      <Model>M3</Model>
7      <Color>Metallic Silver</Color>
8      <Price>40,000</Price>
9      <Location>Liverpool</Location>
10   </Vehicle>
11 </Cars>
```

Figure 5.27: XML File

Figure 5.28 depicts the style sheet for **`border.css`** file.

```
1 Manufacturer {border: thick dashed magenta }
2 Model { border: thick solid olive }
3 Color { border: thick groove aqua }
4 Price { border: thick inset gray }
```

Figure 5.28: Border.css File

where,

`Manufacturer {border: thick dashed magenta }`

Displays a thick and dashed magenta border around the content of Manufacturer element.

`Model { border: thick solid olive }`

Displays a thick and solid olive border around the content of Model element.

`Color { border: thick groove aqua }`

Displays a thick and groove aqua border around the content of Color element.

`Price { border: thick inset gray }`

Displays a thick and inset gray border around the content of Price element.

The output is shown in figure 5.29.

Figure 5.29: Output of Using Borders

where,

`BMW`

Thick dashed magenta border

`M3`

Thick solid olive border

`Metallic Silver`

Thick groove aqua border

`40,000`

Thick inset gray border

5.3.9 Padding Properties in CSS

The border property surrounds the text in an element with an outline. To insert space between the border and the text of an element, CSS provides the padding property.

Figure 5.30 shows the syntax for padding property.

```
padding : padding_width
```

Figure 5.30: Syntax for Padding Property

where,

`padding`

Property specifying padding between the text and border of an element.

`padding_width`

Composite value that can have maximum four values in the following order: top, right, bottom, and left. These values are followed by length unit designators. However, default is pixel unit.

Figure 5.31 shows the code for style rules defined in `Padding.css` file.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <?xml-stylesheet href="Padding.css" type="text/css"?>
3  <Cars>
4    <Vehicle>
5      <Manufacturer>BMW</Manufacturer>
6      <Model>M3</Model>
7      <Color>Metallic Silver</Color>
8      <Price>40,000</Price>
9      <Location>Liverpool</Location>
10   </Vehicle>
11 </Cars>
```

Figure 5.31: XML File

Figure 5.32 depicts the style sheet for **`padding.css`** file.

```
1 Manufacturer {border: thick dashed magenta }
2 Model { border: thick solid olive }
3 Color { border: thick groove aqua }
4 Price { border: thick inset gray }
5 Manufacturer { padding: 2 }
6 Model { padding: 2 5 }
7 Color { padding: 2 5 8 }
8 Price { padding: 2 5 8 10 }
```

Figure 5.32: Padding.css File

where,

`Manufacturer { padding: 2 }`

Inserts padding of 2 pixels between the four borders and text of Manufacturer element. The four borders applicable are top, right, bottom, and left.

`Model { padding: 2 5 }`

Inserts padding between the borders and text of Model element. The value 2 is applied to top and bottom borders and 5 is applied to left and right borders.

```
Color { padding: 2 5 8 }
```

Inserts padding between the borders and text of Color element. The value 2 is applied to top border, 5 to left and right borders, and value 8 to bottom border.

```
Price { padding: 2 5 8 10 }
```

Inserts padding between the borders and text of Price element. The value 2 is applied to top border, 5 to right border, 8 to bottom border, and 10 to left border.

The output is shown in figure 5.33.

Figure 5.33: Output of Using Padding

5.3.10 CSS Units

The values assigned to CSS properties are expressed in length units. Table 5.3 describes these units.

Unit Type	Unit Designator
Relative	em – defines the height of element's font. ex – defines the x-height of element's font. px – defines the pixel relative to display device. % - percentage.
Absolute	in – inches cm – centimeters mm – millimeters pt – 1/72 inch pc – 12 pt

Table 5.3: CSS Units

Concepts

5.3.11 Position Properties

Every element's text is placed in a box of its own. Table 5.4 lists the CSS positioning properties to position the text inside the box. Note that the properties top, left, bottom, and right are used only if value of position property is not static.

Property	Description	Possible Values
`position`	Property to place an element in a static, relative, absolute or fixed position	`static`, `fixed`, `relative` or `absolute`
`top`	Property specifying how far the top edge of an element is above/below the top edge of the parent element	`auto`, `integer` or `floating point` values adhering to CSS length units
`left`	Property specifying how far the left edge of an element is to the right/left of the left edge of the parent element	`auto`, `integer` or `floating point` values adhering to CSS length units
`bottom`	Property specifying how far the bottom edge of an element is above/below the bottom edge of the parent element	`auto`, `integer` or `floating point` values adhering to CSS length units
`right`	Property specifying how far the right edge of an element is to the left/right of the right edge of the parent element	`auto`, `integer` or `floating point` values adhering to CSS length units

Table 5.4: Position Properties

5.3.12 Position Properties Example

Let us study an example based on positioning properties of CSS.

Figure 5.34 shows the code for style rules defined in `Position.css` file.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <?xml-stylesheet href="Position.css" type="text/css"?>
3  <Cars>
4    <Vehicle>
5      <Manufacturer>BMW</Manufacturer>
6      <Model>M3</Model>
7      <Color>Metallic Silver</Color>
8      <Price>40,000</Price>
9      <Location>Liverpool</Location>
10   </Vehicle>
11 </Cars>
```

Figure 5.34: XML File

Figure 5.35 depicts the style sheet for **position.css** file.

```
1 Location { position: relative; top: 20; left: 20 }
2 Price { position: absolute; top: 40; left: 20 }
```

Figure 5.35: Position.css File

where,

`Location { position: relative; top:20; left :20 }`

Positions the text of Location element relative to the previous element. However, inside the box, the content is placed at 20 pixels from top and 20 pixels from left.

`Price { position: absolute; top:40; left :20 }`

Positions the text of Price element at an absolute location of 40 pixels from the top and 20 pixels from the left.

The output is shown in figure 5.36.

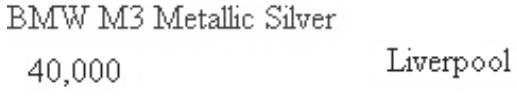


Figure 5.36: Output of Using Location

where,

`40,000`

Text displayed using absolute positioning

`Liverpool`

Text displayed using relative positioning

Concepts

5.3.13 Display Property

In HTML, if you wanted the text to appear as new paragraph, you would use the <P> tag. The same can be achieved in XML by using CSS display property.

Figure 5.37 shows the syntax for display property.

```
display : value

where value = none | inline | block
```

Figure 5.37: Syntax for Display Property

where,

`display`

Property to specify how the element is to be rendered.

`none`

No rendering is applied to the element.

`inline`

Displays the element text as inline. This is the default value if no value is specified.

`block`

Displays the element text on a new line in a block of its own.

Figure 5.38 shows the code for style rules defined in `Display.css`.

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet href="Display.css" type="text/css"?>
<Cars>
  <Vehicle>
    <Manufacturer>BMW</Manufacturer>
    <Model>M3</Model>
    <Color>Metallic Silver</Color>
    <Price>40,000</Price>
    <Location>Liverpool</Location>
  </Vehicle>
  <Vehicle>
    <Manufacturer>TOYOTA</Manufacturer>
    <Model>INNOVA</Model>
    <Color>Gray</Color>
    <Price>38,000</Price>
    <Location>California</Location>
  </Vehicle>
</Cars>
```

Figure 5.38: XML File

Figure 5.39 depicts the style sheet for **`display.css`** file.

```
Price { display: block }
```

Figure 5.39: Display.css File

where,

`Price { display: block }`

Inserts a new line before and after the text of element `Price`.

The output is shown in figure 5.40.

Figure 5.40: Output of Using Block

Data in each element is displayed on a separate row in block of its own.

5.3.14 Text Alignment and Indentation

The `display:block` property renders the text of an element in a separate block. Inside this block, you can align and indent the text using the CSS properties `text-align` and `text-indent` properties respectively.

Figure 5.41 shows the syntax for text alignment and indentation.

```
text-align : alignment_value

text-indent : value
```

Figure 5.41: Syntax

where,

`text-align`

Property to align the text in a block.

`alignment_value`

Can be one of the following: left (default), right, center and justify.

`text-indent`

Property to indent the text in a block.

`value`

Floating point value followed by absolute units designators or relative units designators; or an integer value followed by percentage (%) symbol.

Figure 5.42 shows the code for style rules defined in `Align.css`.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <?xml-stylesheet href="Align.css" type="text/css"?>
3  <Cars>
4    <Vehicle>
5      <Manufacturer>BMW</Manufacturer>
6      <Model>M3</Model>
7      <Color>Metallic Silver</Color>
8      <Price>40,000</Price>
9      <Location>Liverpool</Location>
10   </Vehicle>
11   <Vehicle>
12     <Manufacturer>TOYOTA</Manufacturer>
13     <Model>INNOVA</Model>
14     <Color>Gray</Color>
15     <Price>38,000</Price>
16     <Location>California</Location>
17   </Vehicle>
18 </Cars>
```

Figure 5.42: XML File

Figure 5.43 depicts the style sheet for **`align.css`** file.

```
1 Price { display: block }
2 Price { text-align: left }
3 Price { text-indent: 20 }
```

Figure 5.43: Align.css File

where,

`Price { display: block }`

Inserts a new line before and after the text of element Price and displays the text in a separate block.

```
Price { text-align: left }
```

Left aligns the text of Price element.

```
Price { text-indent: 20 }
```

Indents the text of Price element at 20 pixels.

The output is shown in figure 5.44.

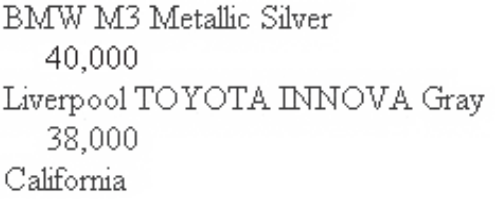


Figure 5.44: Output of Using Indents

where,

```
40,000
```

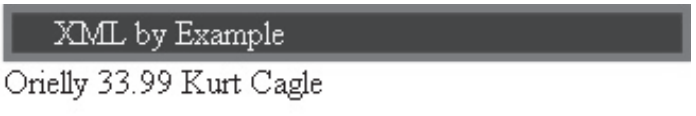
The text in element Price is display on a separate row and left indented at 20 pixels.

Knowledge Check 3

1. Can you match the properties with the corresponding descriptions?

	Description		Property
(A)	To display an element's data in italics	**(1)**	`font-family`
(B)	To display an element's data in bold	**(2)**	`font-size`
(C)	To display an element's data in a small font	**(3)**	`font-style`
(D)	To display an element's data in Times New Roman	**(4)**	`font-weight`
(E)	To display an element's data a big font	**(5)**	`font-size: small`

2. Can you write the style rules to display the data "XML by Example" in the format shown in the following figure?



Note: In the image, display type is `block`, background color is `blue`, color is `white`, border is `medium solid magenta`, text indent is `20`.

3. Can you match the properties with the corresponding descriptions?

	Description		Property
(A)	To insert space around an element	**(1)**	`text-align`
(B)	To insert space between the text and border of element's box	**(2)**	`border`
(C)	To display an outline around the element's data	**(3)**	`padding`
(D)	To place element's data at the specified location	**(4)**	`position`
(E)	To place element's data in the center	**(5)**	`margin`

5.4 Inheritance and Cascades in CSS

In this last lesson, **Inheritance and Cascades in CSS**, you will learn to:

- Define the process of cascading.
- Explain inheritance.

5.4.1 Cascading in CSS

There are several ways in which style sheets can be defined. For example, one document can have more than one style sheet defined for it. A browser has its own style sheet. In such cases, it is possible that more than one style rule may exist for an element. Hence, the World Wide Web Consortium (W3C) defines the following rules to determine what style to apply to an element. The sorting of style rules begins with rule number 1. If the style rule matches rule number 1, then the search is over; otherwise the search continues with rule number 2 and so on.

1. Find all property declarations for an element in question. Apply the style rules if the element name matches the element in the selector. If there is no style rule defined for an element, the element inherits the style rules defined for the parent element. However, if the parent element does not have any style rules defined, then the element is rendered with default values.

2. Style rules declared as important are considered next. A style rule can be declared important in the following manner:

```
Threat { display: block ! important }
```

Important property declarations have a higher precedence over normal property declarations.

Note: Style rules defined as important are said to have higher weight.

3. Next, the origin of style sheet is determined. A style sheet can have following sources:

   - **Author**: The author of the XML document can define a style sheet in an external document to format the XML document.
   - **User/Reader**: The end-user viewing the XML document can specify his/her personal style sheet to be used to format the XML document.
   - **User-agent**: The browser is referred to as user-agent. A browser has its own default style sheet.

   The style rules defined in the author's style sheet override the style rules defined in the user's style sheet. The author's and the user's style sheet both override the user-agent's style sheet.

4. Next, the specificity of a selector is determined in that a more specific selector will override the general selector. The specificity is determined by carrying out the following three activities:

   - Count the number of ID attributes in the selector.
   - Count the number of class attributes in a selector.
   - Count the number of element names in the selector.

5. Next, write the three numbers with no commas or spaces and in the same sequence as shown earlier. Higher the number, higher is the specificity. Rules with higher specificity override the ones with lower specificity. For example, following is a list of selectors sorted by specificity:

```
#favorite  {...}            /* a=1;b=0;c=0; specificity = 100 */
Name, CD, Artist.caps {...} /* a=0;b=1;c=3; specificity = 013 */
Artist.caps  {...}          /* a=0;b=0;c=1; specificity = 001 */
```

6. If two rules have the same weight, then the one specified last wins.

5.4.2 Inheritance in CSS

Inheritance is the ability of one entity to acquire the characteristics of another entity. In CSS, the child elements inherit the styles rules defined for parent element. However, this is applicable only if the child has no explicit style rule defined for it. Otherwise, the style rule defined for child element overrides the style rule defined for parent element.

Figure 5.45 shows that the text in element Name is displayed in italics. This is because there are no style rules defined for elements Audio and Name. Hence, the style rule defined for CD is inherited by child element Name.

XML Document

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet href="CD.css" type="text/css"?>
<CD>
  <Audio>
   <Name>2007 Hits</Name>
  </Audio>
</CD>
```

Style Sheet

```
CD { font-style: italic }
```

Output

2007 Hits

Figure 5.45: Inheritance in CSS

Knowledge Check 4

1. Which of the following statements about cascading and inheritance are true and which are false?

(A)	Declarations are sorted by weight and origin.
(B)	Inheritance enables one entity to acquire the properties of another entity.
(C)	A child element inherits the properties of an ancestor element only if it is an immediate child.
(D)	Declarations are sorted by the order specified.
(E)	There is always only one style rule defined for every element.

Concepts

Module Summary

In this module, **Style Sheets**, you learnt about:

- **Style Sheets**

  Style Sheets are a set of rules that define the appearance of data. These rules are written in a file with the extension `.css`. The `.css` file is associated with an XML document using the xml-processing instruction.

- **Selectors in CSS**

  Selectors define the elements to which the styles will be applied. The various types of selectors are simple, universal and ID selectors.

- **Properties and Values**

  CSS provides several properties to define the appearance of data. Some of these properties are color, background-color, position, padding, font, text- align to name a few.

- **Inheritance and Cascades in CSS**

  In CSS, a child element inherits the styles rules applied to its ancestor element. Also, there are several sources of style sheets, hence CSS follows W3C defined cascading order when applying style rules to elements.

Module

6 XSL and XSLT

Concepts

Module Overview

Welcome to the module, **XSL and XSLT**. This module deals with the techniques of transforming XML documents into XML, HTML, and Text documents using XSL style sheets. This module also aims at providing a clear understanding of creating an XSL style sheet using the various XSL elements.

In this module, you will learn about:

- Introduction to XSL
- Working with XSL

6.1 Introduction to XSL

In this first lesson, **Introduction to XSL**, you will learn to:

- Define XSL, XSLT, and their purpose.
- Explain the structure and syntax of XSL.
- Distinguish between CSS and XSL.

6.1.1 Style Sheets

A style sheet is a collection of commands that tells a processor (such as a Web browser, or a document reader) how to render the visual appearance of content in a Web page.

Style sheets typically contain instructions like:

- Number figures sequentially throughout the document.
- Display hypertext links in blue.
- Position text and images on a Web page.
- Create a layout optimized for small displays for mobile phones.

Cascading Style Sheet (CSS) is a style sheet technology that is used for HTML content formatting.

Concepts

Extensible Style sheet Language (XSL) is developed by World Wide Web Consortium (W3C) to describe how the XML document should be displayed. Figure 6.1 shows an example of style sheet.

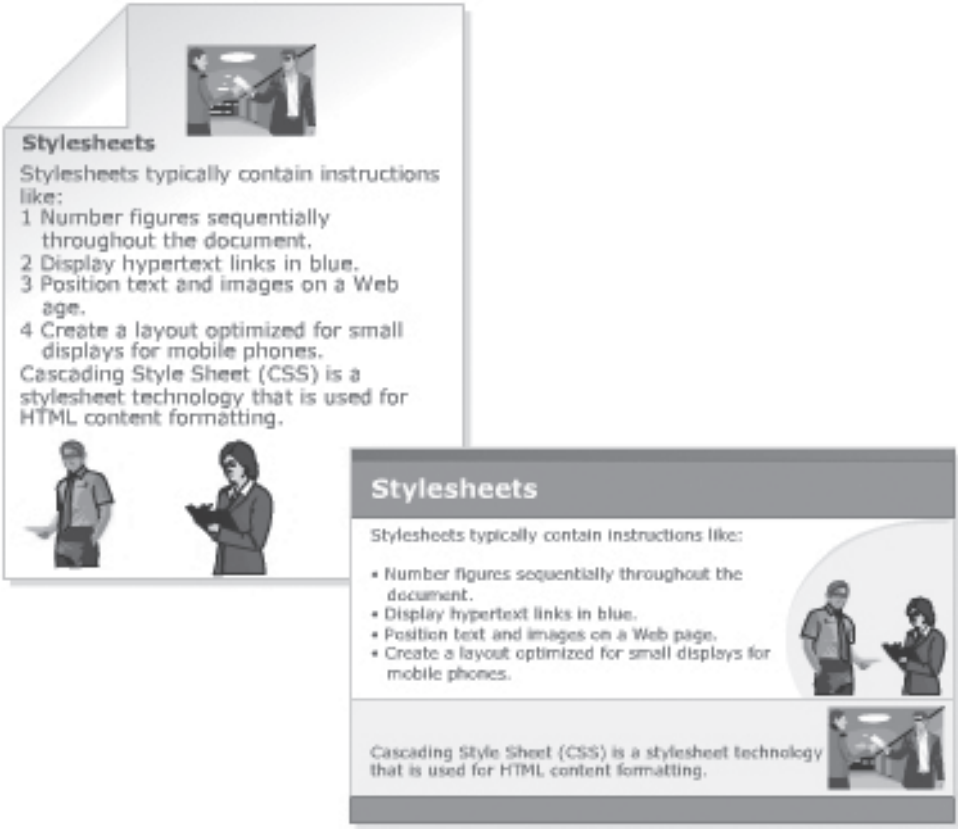


Figure 6.1: Style Sheet

6.1.2 Extensible Style sheet Language (XSL)

XSL is an XML-based language used to create style sheets. XSL is designed to format XML content for display purposes and also has the ability to completely transform XML documents. XSL is a specification from the World Wide Web Consortium (W3C). XSL Transformation Recommendation describes the process of transforming an XML document, using a transformation engine and XSL. The XML document and XSL style sheet are provided as input to the XML transformation engine, also known as the XSLT processor. The output of the processor is in the form of a result tree in which the elements of the XML document are converted into nodes having specific text values and attributes. XSLT processors use XSL style sheets to gather instructions for transforming source XML documents to output XML documents. Style sheets are used to change the structure of the XML document and modify the output.

XSL consists of three languages:

- **XSL Transformations (XSLT)**

  XSLT transforms an XML document into another XML document that can actually be understood and displayed by a computer.

- **XML Path Language (XPath)**

  XPath is used as the navigator for XSL. XSL uses XPath to find parts of the source document that should match a certain predefined template. When XPath finds the node, XSLT takes over and performs a transformation, turning the source document into a result document.

- **XSL Formatting Objects (XSL-FO)**

  An XML language for formatting XML documents. Once XPath has searched through the source document and used XSLT to transform the source document into the result document, the document then needs to be formatted so that the Web browser will be able to present the document with the proper layout and structure. Simply put, XSL-FO is the part of XSL that produces the final output.

Figure 6.2 depicts the structure of XSL.

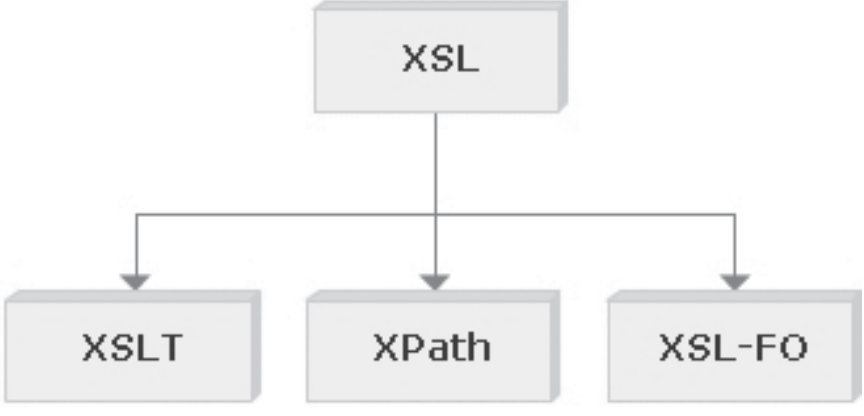


Figure 6.2: Structure of XSL

6.1.3 XSL Transformations

The transformation component of the XSL style sheet technology is XSLT. Its purpose is to transform XML documents. It describes the process of transforming an XML document, using a transformation engine and XSL. The XML document and XSL style sheet are provided as input to the XML transformation engine, also known as the XSLT processor. The output of the processor is in the form of a result tree in which the elements of the XML document are converted into nodes having specific text values and attributes.

Concepts

Note: You can transform an XML document into any text-based format, including HTML, plain text, or another schema of XML using XSLT.

An XSLT processor is an application that connects an XML document with an XSLT style sheet. The transformation of the XML document is in the form of a node tree, which can be displayed as output or sent for further transformations.

6.1.4 XSL Processing Model

The XML processor reads an XML document and processes it into a hierarchical tree containing nodes for each piece of information in a document. After a document has been processed into a tree, the XSL processor begins applying the rules of an XSL style sheet to the document tree.

The XSL processor starts with the root node in the tree and performs pattern matching in the style sheet. These patterns are stored within constructs known as templates in XSL style sheets. The XSL processor analyzes templates and the patterns associated with them to process different parts of the document tree. When a match is made, the portion of the tree matching the given pattern is processed by the appropriate style sheet template. At this point, the rules of the template are applied to the content to generate a result tree. The result tree is itself a tree of data and the data has been transformed by the style sheet.

Figure 6.3 displays the XSL processing model.

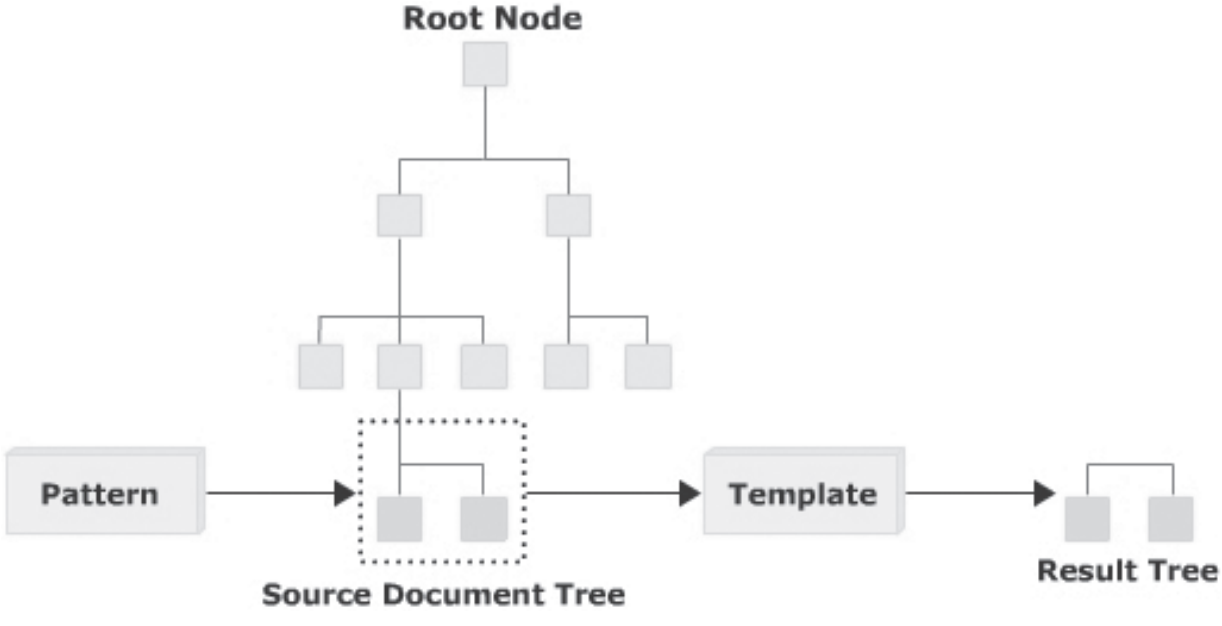


Figure 6.3: XSL Processing Model

Note: Pattern matching is the process of using patterns to identify nodes in the tree that are to be processed according to XSL styles.

6.1.5 XSLT Structure and Syntax

XSLT follows normal rules of XML syntax. It uses a standard document introduction, matching closing tags for any opening tags that contain content, and a proper syntax for empty elements.

In XSL, the style rules are written in a file with the extension `.xsl`. This file is associated with an XML document by using the statement `<?xml-stylesheet href="xsl file" type="text/xsl"?>`. Here, the `xml-stylesheet` is the processing instruction.

Syntax:

All XSLT style sheets use the `xsl:stylesheet` element as the document element.

An XSLT document takes the following basic form:

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

where,

`<xsl:stylesheet>`: Root element of the style sheet.

`xmlns:xsl="http://www.w3.org/1999/XSL/Transform"`: refers to the official W3C XSLT namespace. You must include the attribute `version="1.0"` if you use this namespace.

The XSLT namespace must be declared at the top of the document in order to get access to the XSLT elements, attributes and features.

Note: The `<xsl:transform>` element holds the same syntactic value as `<xsl:stylesheet>`. The `<xsl:transform>` and `<xsl:stylesheet>` are completely synonymous and either can be used. The `xsl:stylesheet` element is used more commonly than `xsl:transform` element.

6.1.6 Top Level XSLT Elements

The top level XSLT elements can occur directly inside the `xsl:stylesheet` element. An element occurring as a child of an `xsl:stylesheet` element is called a top-level element. These elements provide the building blocks for creating XSLT documents.

The top level elements in the XSLT syntax are listed in table 6.1. These elements can occur directly inside the `xsl:stylesheet` or `xsl:transform` elements.

Element	Description
xsl:decimal-format	Defines the default decimal format for number to text conversion.
xsl:include	Used to include one style sheet into another.
xsl:key	Defines a named key to be used with the key() function for operating on patterns and expressions.
xsl:output	Defines the format of the output created by the style sheet.
xsl:preserve-space	Defines the format of the output created by the style sheet.

Table 6.1: Top level XSLT Elements

- **xsl:decimal-format XSLT element**

Syntax:

```
<xsl:decimal-format decimal-separator="character" digit="character"
grouping-separator="character" infinity="string" minus-sign="character"
name="name" NaN="string" pattern-separator="character"
percent="character" per-mille="character" zero-digit="character"/>
```

where,

`decimal-separator="character"`: Defines the character that is used to separate the integer and fraction part of a number. The default is a dot (.).

`digit="character"`: Defines the character that is used to signify that a digit is needed in a format pattern. The hash (#) is the default.

`grouping-separator="character"`: Defines the character that is used to separate groups of digits. The default is a comma (,).

`infinity="string"`: Defines the string that is used to represent the infinite value. The default is the string, "infinity".

`minus-sign="character"`: Defines the character that is used to represent a minus sign. The default is the hyphen (-).

`name="name"`: Assigns a name to the element.

`NaN="string"`: Defines the string that is used to indicate that the value is not a number. The default is string, "NaN".

`pattern-separator="character"` **:** Define the character that is used to separate positive and negative sub patterns in a format pattern. The default is semicolon (;).

`percent="character"`: Defines the character that is used to represent a percent sign. The default is the percent sign (%).

`per-mille="character"`: Defines the character that is used to represent the per thousand sign. The default is the Unicode per mille character (%).

`zero-digit="character"`: Defines the character that is used to represent the digit zero. The default is (0).

The following code demonstrates the several numeric format variations.

Code Snippet:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
<xsl:decimal-format name="name" digit="D" />
<xsl:output method="html"/>
    <xsl:template match="/">
        <xsl:value-of select='format-number(45665789, "#.000")' />
        <xsl:value-of select='format-number(0.3456789, "###%")' />
        <xsl:value-of select='format-number(789, "D.0", "name")' />
        <xsl:value-of select='format-number(123456789, "$DDD,DDD,DDD.
DD","name")' />
        <xsl:value-of select="format-number(193 div 200, '###.#%')"/>
        <xsl:value-of select="format-number(a div 0, '###,###.00')"/>
        <xsl:value-of select="format-number(1 div 0, '###,###.00')"/>
    </xsl:template>
</xsl:stylesheet>
```

The `select` statement returns the values 45665789.000, 35%, 789.0, $123,456,789, 96.5%, NaN and Infinity respectively.

- **xsl:include XSLT element**

Syntax:

```
<xsl:include href = "uri"/>
```

where,

`href = "uri"`: A Uniform Resource Identifier address identifying the XSLT file to be included.

The following code demonstrates how to include another style sheet file within a style sheet file.

Code Snippet:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
<xsl:include href="ExampleTemplate.xsl"/>
<xsl:template match="/">
    ..........
    ..........
    </xsl:template>
</xsl:stylesheet>
```

In this code, the **`ExampleTemplate.xsl`** style sheet file is included within the current style sheet file.

- **xsl:key XSLT element**

Syntax:

```
<xsl:key match="pattern" name="qname" use="expression">
</xsl:key>
```

where,

`match="pattern"`: Defines the nodes to which the key will be applied.

`name="qname"`: Specifies the qualified names of the key.

`use="expression"`: Used to determine the value of the key for each node.

The following code contains the XML file and demonstrates how to use the `xsl:key` element.

Code Snippet:

```
<?xml version="1.0" encoding = "UTF-8"?>
<?xml-stylesheet type="text/xsl" href="key.xsl"?>
<APTstaff>
<Developer name="David Blake" address="B-602, East West Coast" phone="567-877-9766"/>
<Developer name="Roger Blake" address="B-345, East West Coast" phone="345-865-9777"/>
</APTstaff>
```

The following code contains the style sheet file and demonstrates how to use the `xsl:key` element.

Code Snippet:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
<xsl:key name="stafflist" match="Developer" use="@name"/>
<xsl:template match="/">
<xsl:for-each select="key('stafflist', 'David Blake')">
NAME: <xsl:value-of select="@name"/>
ADDRESS: <xsl:value-of select="@address"/>
PHONE: <xsl:value-of select="@phone"/>
</xsl:for-each>
</xsl:template>
</xsl:stylesheet>
```

The result upon execution would be:

```
NAME: David Blake ADDRESS: B-602, East West Coast PHONE: 567-877-9766
```

- **xsl:output XSLT element**

Syntax:

```
<xsl:output cdata-section-elements="namelist" doctype-public="string"
doctype-system="string" encoding="string" indent="yes" | "no" media-
type="mimetype" method="html" | "name" | "text" | "xml" omit-xml-
declaration="yes" | "no" standalone="yes" | "no" version="version_number"
/>
```

The details of the attributes are given in table 6.2.

Attribute	Description
cdata-section-elements	Optional. A white space delimited list of elements whose text contents should be written as CDATA sections.
doctype-public	Optional. Specifies the public identifiers that go in the document type declaration.
doctype-system	Optional. Specifies the system identifiers that go in the document type declaration.
encoding	Optional. Sets the value of the encoding attribute in the output.
indent	Optional. Specifies whether or not to indent the output. If set to "yes", the XML and HTML outputs are step-indented to make them more readable.
media-type	Optional. Defines the MIME type of the output. The default is "text/xml".
method	Optional. Defines the type of output. The three permitted values are HTML, text and XML. The default is XML. However, if the first child element of the root node is the HTML <html> tag and there are no preceding text nodes, then the default output type is set to HTML.
omit-xml-declaration	Optional. A "yes" specifies that the XML declaration (<?xml...?>) should be ignored in the output. The "no" specifies that the XML declaration should be included in the output. The default is "no".
standalone	Optional. Specifies whether the XSLT processor should output a standalone declaration. Yes specifies that a standalone declaration should occur in the output. No, the default, signifies that a standalone declaration should occur in the output.
version	Optional. Provides the W3C version number for the output format. If the output is XML, the default version is 1.0. Or if the output type is HTML, the default version is 4.0.

Table 6.2: xsl:output element Attributes

The following code demonstrates the use of `xsl:output` element. It shows that the output will be an HTML document, version 4.0 and indented for readability.

Code Snippet:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html" version="4.0" indent="yes"/>
...
...
</xsl:stylesheet>
```

- **xsl:preserve-space XSLT element**

Syntax:

```
<xsl:preserve-space elements="names"/>
```

where,

`elements`: Specifies a white space separated list of element names for which white space should be preserved or removed.

The following code demonstrates the use of `xsl:preserve-space` element and shows that the white space is preserved for **`name`** and **`address`** elements.

Code Snippet:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet  version="1.0"  xmlns:xsl="http://www.w3.org/1999/XSL/
Transform">
    <xsl:preserve-space elements="name address"/>
    <xsl:template match="/">
        <xsl:for-each select="Company/Programmer">
        <xsl:value-of select="name"/>
        <br/>
        <xsl:value-of select="age"/>
        <br/>
        <xsl:value-of select="address"/>
        <br/>
        <xsl:value-of select="company"/>
```

```
        <br/>
        </xsl:for-each>
    </xsl:template>
</xsl:stylesheet>
```

6.1.7 CSS and XSL

XSL and CSS are two different style languages recommended by the World Wide Web Consortium (W3C). XSL is more powerful and complex than CSS. Some of the differences between CSS and XSL are listed in table 6.3.

CSS	XSL
Style sheet language to create a style for HTML and XML documents	Style sheet language to create a style for XML documents
Determines the visual appearance of a page, but does not alter the structure of the source document	Provides a means of transforming XML documents
Does not support decision structures and it cannot calculate quantities or store values in variables	Supports decision structures and can calculate quantities or store values in variables
Uses its own notation	Uses an XML notation
Highly effective and easy to learn for simple applications	Designed to meet the needs of more complex applications for richer style sheets

Table 6.3: Difference between CSS and XSL

Knowledge Check 1

1. The steps to process an XSL style sheet and apply it to an XML document are given here. Can you arrange the steps in sequence to achieve the processing?

(1)	Apply the rules of an XSL style sheet to document tree.
(2)	Portion of a tree matching the given pattern is processed by appropriate style sheet template.
(3)	XSL processor starts with root node in tree and performs pattern matching.
(4)	XML processor reads an XML document.
(5)	XML processor creates a hierarchical tree containing nodes for each piece of information.

2. Can you match the XSL elements against their corresponding description?

	Description		XPath Node
(A)	Used to add one style sheet to another	**(1)**	`xsl:template`
(B)	Allow to set a variable in the `xsl` file	**(2)**	`xsl:output`
(C)	Allow the style sheet to alias one namespace prefix for another in the result tree	**(3)**	`xsl:import`
(D)	Used to build templates	**(4)**	`xsl:namespace-alias`
(E)	Allow style sheet authors to specify how they wish the result tree to be output	**(5)**	`xsl:variable`

3. Which of the statements about CSS and XSL style sheet languages are true and which statements are false?

(A)	CSS uses complex elements to format the documents.
(B)	XSL describes how the XML document should be displayed.
(C)	XSL elements can be used to perform complex calculations.
(D)	XSL is a style sheet application specifically for HTML.
(E)	CSS is a simple, styling-based approach that does not require advanced programming skills.

6.2 Working with XSL

In this last lesson, **Working with XSL**, you will learn to:

- Explain XSL templates.
- Describe the use of `select` attribute.
- State how to use `xsl:value-of` element.
- Describe how to use `xsl:for-each` element.
- Explain briefly how to use `xsl:text` element.
- Describe how to use `xsl:number` element.
- Describe how to use `xsl:if` element.
- Describe how to use `xsl:choose` element.
- Explain how to perform sorting using XSL.

Concepts

6.2.1 XSL Templates

A template is the main component of a style sheet. Templates are defined with the help of rules. A template rule is used to control the output of the XSLT processor. It defines the method by which an XML element node is transformed into an XSL element node. A template rule consists of a pattern that identifies the XML node and an action that selects and transforms the node.

Each template rule is represented by the `xsl:template` element. The `xsl:template` is an element that defines an action for producing output from a source document.

Figure 6.4 shows an example of XSL template.

Figure 6.4: XSL Template

The built-in template rules are as follows:

- **Built-in template rule for modes**: Ensures that elements and roots in any mode are processed.

  The built-in template rule for modes is as follows:

```
<xsl:template match="*|/" mode="x">
<xsl:apply-templates mode="x"/>
</xsl:template>
```

- **Built-in template rule for element and root nodes**: Processes the root node and its children. The built-in template rules for element and root nodes are as follows:

```
<xsl:template match="*|/">
<xsl:apply-templates/>
</xsl:template>
```

- **Built-in template rule for text and attribute nodes**: Copies text and attribute nodes to the result tree. The built-in template rules for text and attribute nodes are as follows:

```
<xsl:template match="text()|@*">
<xsl:value-of select="."/>
</xsl:template>
```

- **Built-in template rule for comment and processing instruction nodes**: Does not perform any function. It is applied to the comments and processing instructions in the XML document. The built-in template rules for comment and processing instruction nodes are as follows:

```
<xsl:template match="comment()|processing-instruction()"/>
```

- **Built-in template rule for namespaces node**: Does not perform any function. It is applied to the namespaces node in the XML document. The built-in template rule for namespaces node is as follows:

```
<xsl:template match="namespace()"/>
```

Note: The Built-in rules have a lower priority than the other template rules.

While traversing an XML document, template rules are activated in the order in which they match the elements. A template can change the order of traversal, and it can also prevent particular elements from being processed. The `xsl:apply-templates` element allows the user to choose the order in which the document is to be traversed.

Concepts

For example, consider an element **Order** having child elements as **OrderNumber**, **ItemInfo**, and **Price**. The **ItemInfo** element has further child elements as **ItemName** and **Quantity**. The **ItemName** element comes before the **Quantity** element in the tree structure.

To get an output where **Quantity** is displayed before ItemName, the following template can be used:

```
<xsl: template match= "ItemInfo">
<xsl: value-of select= "Quantity"/>
<xsl: value-of select= "ItemName"/>
</xsl: template>
```

6.2.2 The xsl:template Element

The `xsl:template` element is used to define a template that can be applied to a node to produce desired output.

The match attribute in `xsl:template` is used to associate the template with an XML element. You can also define a template for a whole branch of the XML document by using the `match` attribute (for example, `match="/"` defines the whole XML document).

Syntax:

```
<xsl:template
   match="pattern"
   mode="mode"

   name="name"
   priority="number"
>
</xsl:template>
```

where,

`match`: Is a pattern that is used to define which nodes will have which template rules applied to them. If this attribute is omitted there must be a `name` attribute.

`mode`: Allows the same nodes to be processed more than once.

`name`: Specifies a name for the template. If this attribute is omitted there must be a `match` attribute.

`priority`: Is a real number that sets the priority of importance for a template. The higher the number, the higher the priority.

The syntax for the some of the match patterns are represented as follows:

The `<xsl:template match ="/">` represents the root element in the XML document, `<xsl:template match = "name of the element">` represents all the elements in the XML document that have the specified name, `<xsl:template match = parent1/child>` represents the child element by specifying the exact parent element and `<xsl:template match = "/ | *">` matches the entire document.

The code in figure 6.5 demonstrates the usage of `xsl:template` element in style sheets.

```
1  <?xml version="1.0"?>
2  <xsl:stylesheet version="1.0"
3    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4    <xsl:template match="/">
5      <html>
6        <body>
7          <h1> XSL TEMPLATE SAMPLE</h1>
8        </body>
9      </html>
10   </xsl:template>
11 </xsl:stylesheet>
```

Figure 6.5: XSL Template Element

An XSL style sheet is an XML document itself, it always begins with the XML declaration: `<?xml version="1.0" ?>`.

The next element, `<xsl:stylesheet>`, defines that this document is an XSLT style sheet document. It also contains the version number and XSLT namespace attributes.

The `<xsl:template>` element defines a template. The `match="/"` attribute associates the template with the root of the XML source document.

The content inside the `<xsl:template>` element defines some HTML to write to the output.

The last two lines define the end of the template and the end of the style sheet.

Concepts

6.2.3 The xsl:apply-templates Element

The `xsl:apply-templates` element defines a set of nodes to be processed. This element, by default, selects all child nodes of the current node being processed, and finds a matching template rule to apply to each node in the set.

Figure 6.6 shows the syntax for `xsl:apply-templates`.

```
<xsl:apply-templates
  select="expression"
  mode="name"
>
</xsl:apply-templates>
```

Figure 6.6: xsl:apply-templates Syntax

where,

`select`

Used to process nodes selected by an expression.

`mode`

Allows the same nodes to be processed more than once. Each time the nodes are processed, they can be displayed in a different manner.

Figure 6.7 shows the code for style rules that are defined in `GEM_Stylesheet.xsl` file.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <?xml-stylesheet type="text/xsl" href="GEM_Stylesheet.xsl"?>
3  <GEMEmployees>
4          <Designer>
5                  <Name>David Blake</Name>
6                  <DOJ>18/11/1973</DOJ>
7                  <Address>512-B Lamington Road</Address>
8                  <Phone>1564-754-111</Phone>
9          </Designer>
10         <Designer>
11                 <Name>Susan Jones</Name>
12                 <DOJ>03/05/1953</DOJ>
13                 <Address>Palm Beach Road</Address>
14                 <Phone>8755-211-111</Phone>
15         </Designer>
16 </GEMEmployees>
```

Figure 6.7: XML File

Concepts

Figure 6.8 depicts the GEM_Stylesheet.xsl file.

```
1  <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
2          <xsl:template match="GEMEmployees/Designer">
3                  <html>
4                          <body>
5                                  <xsl:apply-templates select="Name"/>
6                                  <xsl:apply-templates select="DOJ"/>
7                                  <br/>
8                          </body>
9                  </html>
10         </xsl:template>
11         <xsl:template match="Name">
12                 Name:
13                 <span style="font-size=22px;  color:green">
14                         <xsl:value-of select="."/>
15                 </span>
16                 <br/>
17         </xsl:template>
18         <xsl:template match="DOJ">
19                 Date of Join:
20                 <span style="color:blue;">
21                         <xsl:value-of select="."/>
22                 </span>
23                 <br/>
24         </xsl:template>
25 </xsl:stylesheet>
```

Figure 6.8: GEM_Stylesheet.xsl File

where,

`match="GEMEmployees/Designer"`

Represents the `Designer` child element by specifying the `GEMEmployees` parent element.

`xsl:apply-templates select="Name"`

Applies the template on `Name` element.

`xsl:apply-templates select="DOJ"`

Applies the template on `DOJ` element.

`xsl:value-of`

Used to extract the value of a selected node.

`xsl:template match="Name"`

If the template is matched with Name element, the value of the `Name` element is displayed in green color with font size 22 pixels.

`xsl:template match="DOJ"`

If the template is matched with `DOJ` element, the value of the `DOJ` element is displayed in blue color.

The output is shown in figure 6.9.

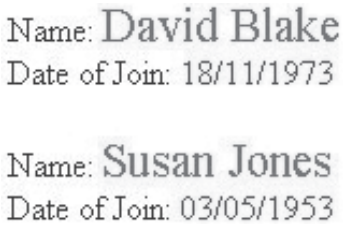


Figure 6.9: Output of Using Templates

6.2.4 The select Attribute

The `select` attribute can be used to process nodes selected by an expression instead of processing all children. The `xsl:apply-templates` with a `select` attribute can be used to choose a particular set of children instead of all children.

Figure 6.10 shows the syntax for select Attribute.

```
<xsl:template match = "element">
   <xsl:apply-templates select ="name of the element"/>
</xsl:template>
```

Figure 6.10: select Attribute Syntax

Explanation for highlighted areas in the image:

`select`

Uses the same kind of patterns as the `match` attribute of the `xsl:template` element. If `select` attribute is not present, all child element, comment, text, and processing instruction nodes are selected.

Figure 6.11 shows the code for style rules that are defined in `book_stylesheet.xsl` file.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <?xml-stylesheet type="text/xsl" href="book_stylesheet.xsl"?>
3  <Catalog>
4          <Book>
5                  <Title>XML By Example</Title>
6                  <Author>David Blake</Author>
7                  <Price>$20.90</Price>
8                  <Year>1990</Year>
9          </Book>
10
11         <Book>
12                 <Title>XML Bible 1.1</Title>
13                 <Author>David Troff</Author>
14                 <Price>$53</Price>
15                 <Year>2004</Year>
16         </Book>
17
18         <Book>
19                 <Title>XML Cookbook</Title>
20                 <Author>Susan Jones</Author>
21                 <Price>$11.10</Price>
22                 <Year>1995</Year>
23         </Book>
24 </Catalog>
```

Figure 6.11: XML File

Figure 6.12 depicts the style sheet for **book_style sheet.xsl** file.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <html>
      <body>
        <h1>Popular XML Books</h1>
        <xsl:apply-templates/>
      </body>
    </html>
  </xsl:template>
  <xsl:template match="Book">
    <p>
      <xsl:apply-templates select="Title"/>
      <xsl:apply-templates select="Author"/>
    </p>
  </xsl:template>
  <xsl:template match="Title">
    Title:
    <span style="color:green;font-size:20pt">
      <xsl:value-of select="."/>
    </span>
    <BR/>
  </xsl:template>
  <xsl:template match="Author">
    Author:
    <span style="color:red;font-size:15pt">
      <xsl:value-of select="."/>
    </span>
    <br/>
  </xsl:template>
</xsl:stylesheet>
```

Figure 6.12: book_stylesheet.xsl File

where,

`select="Title"`

Applies the template on `Title` element.

`select="Author"`

Applies the template on `Author` element.

The output is shown in figure 6.13.

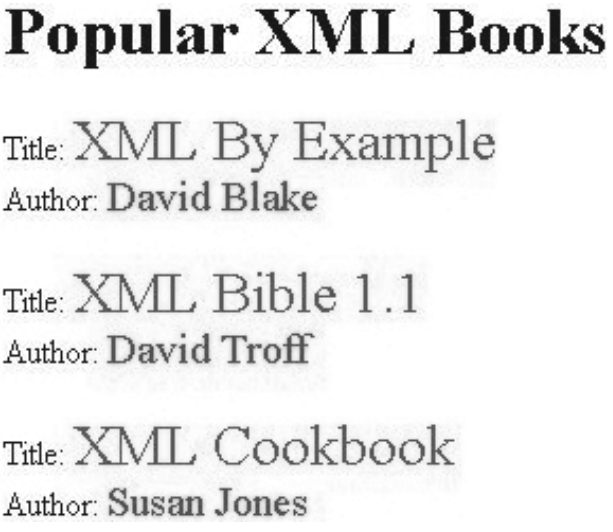


Figure 6.13: Output of Using Select

6.2.5 The xsl:value-of element

The `xsl:value-of` element is used to write or display in the result tree a text string representing the value of the element specified by the `select` attribute. A `xsl:value-of` element can only have one node assigned to it.

Figure 6.14 shows the syntax for `xsl:value-of` element.

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

Figure 6.14: xsl:value-of Syntax

where,

`select`

A mandatory attribute that assigns the node (by name) to the element.

`disable-output-escaping`

Specifies how special characters should appear in the output string.

`yes`

Indicates that special characters should be displayed as is (for example, a < or >).

`no`

Indicates that special characters should not be displayed as is (for example, a > is displayed as `>`).

Figure 6.15 shows the code for style rules that are defined in `person_stylesheet.xsl` file.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <?xml-stylesheet type="text/xsl" href="person_stylesheet.xsl"?>
3  <House>
4    <Person>
5      <FirstName age="20">David</FirstName>
6      <LastName>Blake</LastName>
7    </Person>
8    <Person>
9      <FirstName age="34">Susan</FirstName>
10     <LastName>Jones</LastName>
11   </Person>
12   <Person>
13     <FirstName>Martin</FirstName>
14     <LastName>King</LastName>
15   </Person>
16   <Person>
17     <FirstName>Justin</FirstName>
18     <LastName>Nora</LastName>
19   </Person>
20 </House>
```

Figure 6.15: XML File

Figure 6.16 depicts the style sheet for **person_stylesheet.xsl** file.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
3    <xsl:template match="Person">
4      <p>
5        <xsl:value-of select="FirstName"/>
6        ,
7        <xsl:value-of select="LastName"/>
8      </p>
9    </xsl:template>
10 </xsl:stylesheet>
```

Figure 6.16: person_stylesheet.xsl File

where,

`xsl:value-of select="FirstName"`

Display the value of the element `FirstName`.

`xsl:value-of select="LastName"`

Display the value of the element `LastName`.

The output is shown in figure 6.17.

David , Blake

Susan , Jones

Martin , King

Justin , Nora

Figure 6.17: Output of Using xsl:value-of

6.2.6 The xsl:for-each element

The `xsl:for-each` element can be used to iterate through the XML elements of a specified node set. It applies a template repeatedly to each node in a set.

Figure 6.18 shows the syntax for `xsl:for-each` element.

```
<xsl:for-each select="expression">
</xsl:for-each>
```

Figure 6.18: xsl:for-each Syntax

Explanation for highlighted areas in the image:

`select="expresion"`

The expression is evaluated on the current context to determine the set of nodes to iterate over.

Figure 6.19 shows the code for style rules that are defined in `APTEmployees_stylesheet.xsl` file.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <?xml-stylesheet type="text/xsl" href="APTEmployees_stylesheet.xsl"?>
3  <Employees>
4    <Employee>
5      <Name>John Thorp</Name>
6      <Department>Marketing</Department>
7      <Language>EN</Language>
8      <Salary>$1000</Salary>
9    </Employee>
10   <Employee>
11     <Name>David Blake</Name>
12     <Department>Admin</Department>
13     <Language>GR</Language>
14     <Salary>$1200</Salary>
15   </Employee>
16   <Employee>
17     <Name>Ben Johns</Name>
18     <Department>Physical Education</Department>
19     <Language>TR</Language>
20     <Salary>$800</Salary>
21   </Employee>
22   <Employee>
23     <Name>Susan Lopez</Name>
24     <Department>External Affairs</Department>
25     <Language>EN</Language>
26     <Salary>$2800</Salary>
27   </Employee>
28 </Employees>
```

Figure 6.19: XML File

Concepts

Figure 6.20 depicts the style sheet for **APTEmployees_stylesheet.xsl** file.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:output method="html"/>
  <!-- Template for "employees" elements -->
  <xsl:template match="Employees">
    <hl>Employee Information</hl>
    <!-- Table Header Creation -->
    <table border="1">
      <tr>
        <th>Department</th>
        <th>Name</th>
        <th>Salary</th>
        <th>Language</th>
      </tr>
      <xsl:for-each select="Employee">
        <tr>
          <td>
            <xsl:value-of select="Department"/>
          </td>
          <td>
            <xsl:value-of select="Name"/>
          </td>
          <td>
            <xsl:value-of select="Salary"/>
          </td>
          <td>
            <xsl:value-of select="Language"/>
          </td>
        </tr>
      </xsl:for-each>
      <!-- End of Table -->
    </table>
  </xsl:template>
</xsl:stylesheet>
```

Figure 6.20: APTEmployees_stylesheet.xsl File

where,

`xsl:for-each select="Employee"`

Iterates through the `Employee` node and applies a template.

```
xsl:value-of select="Department"
```

Displays the value of `Department` element.

```
xsl:value-of select="Name"
```

Displays the value of `Name` element.

```
xsl:value-of select="Salary"
```

Displays the value of `Salary` element.

```
xsl:value-of select="Language"
```

Displays the value of `Language` element.

```
</xsl:for-each>
```

End of for-each loop.

The output is shown in figure 6.21.

Employee Information

Department	Name	Salary	Language
Marketing	John Thorp	$1000	EN
Admin	David Blake	$1200	GR
Physical Education	Ben Johns	$800	TR
External Affairs	Susan Lopez	$2800	EN

Figure 6.21: Output of Using xsl:for-each

Concepts

6.2.7 The xsl:text element

The `xsl:text` element is used to add literal text to the output. This element cannot contain any other XSL elements. It can contain only text.

Figure 6.22 shows the syntax for `xsl:text` element.

```
<xsl:text
   disable-output-escaping="yes"|"no">
</xsl:text>
```

Figure 6.22: xsl:text Syntax

Explanation for highlighted areas in the image:

`disable-output-escaping`

Turns on or off the ability to escape special characters.

`yes`

If the value is yes, a `>` will appear as a >.

`no`

If the value is no, a `>` will appear as a `>` in the text.

Figure 6.23 shows the code for style rules that are defined in `Orders_stylesheet.xsl` file.

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="Orders_stylesheet.xsl"?>
<Orders>
  <Item>
    <Name>Dell Laptop</Name>
    <Price>$45000</Price>
    <ShippingDate>10-Mar-07</ShippingDate>
  </Item>
  <Item>
    <Name>Mouse</Name>
    <Price>$450</Price>
    <ShippingDate>10-Mar-07</ShippingDate>
  </Item>
  <Item>
    <Name>Dell Keyboard</Name>
    <Price>$150</Price>
    <ShippingDate>10-Mar-07</ShippingDate>
  </Item>
</Orders>
```

Figure 6.23: XML File

Figure 6.24 depicts the style sheet for **`orders_stylesheet.xsl`** file.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:output method="html"/>
  <xsl:template match="/">
    <html>
      <body>
        <xsl:text>
          Following are the items which are shipped on 10th March
        </xsl:text>
        <br/>
        <xsl:for-each select="Orders/Item">
          <xsl:value-of select="Name"/>
          <xsl:text>,</xsl:text>
        </xsl:for-each>
        <xsl:text>!!!!</xsl:text>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

Figure 6.24: Orders_stylesheet.xsl File

where,

`xsl:for-each select="Orders/Item"`

Iterates through the Item element.

`xsl:value-of select="Name"`

Displays the value of Name element.

`<xsl:text>,</xsl:text>`

Inserts a comma (,) after each name value.

`<xsl:text>!!!!</xsl:text>`

Inserts four exclamation marks (!) at the end of the output.

The output is shown in figure 6.25.

Following are the items which are shipped on 10th March
Dell Laptop,Mouse,Dell Keyboard,!!!!

Figure 6.25: Output of Using xsl:text

6.2.8 The xsl:number element

The `xsl:number` element can be used to determine the sequence number for the current node. It can also be used to format a number for display in the output.

Figure 6.26 shows the syntax for `xsl:number` element.

```
<xsl:number
  count="pattern"
  format="{ string }"
  value="expression"
>
</xsl:number>
```

Figure 6.26: xsl:number Syntax

where,

`count="pattern"`

Indicates what nodes are to be counted. Only nodes that match the pattern are counted.

`format="{ string }"`

Sequence of tokens that specifies the format to be used for each number in the list.

`value="expression"`

Specifies the expression to be converted to a number and output to the result tree.

Figure 6.27 shows the code for style rules that are defined in `item_stylesheet.xsl` file.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <?xml-stylesheet type="text/xsl" href="item_stylesheet.xsl" ?>
3  <Items>
4    <Item>Water Bottle</Item>
5    <Item>Chocolates</Item>
6    <Item>Computer Book</Item>
7    <Item>Mobile Phone</Item>
8    <Item>Personal Computer</Item>
9  </Items>
```

Figure 6.27: XML File

Concepts

Figure 6.28 depicts the style sheet for **item_stylesheet.xsl** file.

```
1  <?xml version="1.0"?>
2  <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
3
4    <xsl:template match="Items">
5      <h3> Items numbered in Western and then upper-case Roman numbering</h3>
6      <xsl:for-each select="Item">
7        <xsl:number value="position()" format="1."/>
8        <xsl:value-of select="."/>
9        ,
10       <xsl:number value="position()" format="I."/>
11       <xsl:value-of select="."/>
12       <br/>
13     </xsl:for-each>
14   </xsl:template>
15 </xsl:stylesheet>
```

Figure 6.28: item_stylesheet.xsl File

where,

`position()`

The current node's position in the source document.

`format="1."`

User-provided number starts with 1.

`format="I."`

User-provided roman number starts with I.

The output is shown in figure 6.29.

Items numbered in Western and then upper-case Roman numbering

1.Water Bottle , I.Water Bottle
2.Chocolates , II.Chocolates
3.Computer Book , III.Computer Book
4.Mobile Phone , IV.Mobile Phone
5.Personal Computer , V.Personal Computer

Figure 6.29: Output of Using xsl:number

where,

`1.Water Bottle, I.Water Bottle`

The first item is numbered with number 1 and roman number I.

`4.Mobile Phone, IV. Mobile Phone`

The fourth item is numbered with number 4 and roman number IV.

6.2.9 The xsl:if element

The `xsl:if` element evaluates a conditional expression against the content of the XML file. The `test` attribute of `xsl:if` element contains a conditional expression that evaluates to a boolean value of `true` or `false`.

Figure 6.30 shows the syntax for `xsl:if` element.

```
<xsl:if
  test="expression"
>
</xsl:if>
```

Figure 6.30: xsl:if Syntax

where,

`test=expression`

The condition in the source data to test with either a true or false answer.

Figure 6.31 shows the code for style rules are that defined in `Librarybooks_stylesheet.xsl` file.

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="Librarybooks_stylesheet.xsl"?>
<Catalog>
  <Book>
    <Title>XML By Example</Title>
    <Author>David Blake</Author>
    <Price>20.90</Price>
    <Year>1990</Year>
  </Book>
  <Book>
    <Title>XML Bible 1.1</Title>
    <Author>David Troff</Author>
    <Price>53</Price>
    <Year>2004</Year>
  </Book>
  <Book>
    <Title>XML Cookbook</Title>
    <Author>Susan Jones</Author>
    <Price>11.10</Price>
    <Year>1995</Year>
  </Book>
  <Book>
    <Title>XML Complete Reference </Title>
    <Author>Andrew Nel</Author>
    <Price>193</Price>
    <Year>2001</Year>
  </Book>
</Catalog>
```

Figure 6.31: XML File

Figure 6.32 depicts the style sheet for **librarybooks_stylesheet.xsl** file.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <html>
      <body>
        <h2>Library Books</h2>
        <table border="1">
          <tr bgcolor = "lime">
            <th>Book Title</th>
            <th>Author</th>
            <th>Year</th>
          </tr>
          <xsl:for-each select="Catalog/Book">
            <xsl:if test="Price &gt; 50">
              <tr>
                <td><xsl:value-of select="Title"/></td>
                <td><xsl:value-of select="Author"/></td>
                <td><xsl:value-of select="Year"/></td>
              </tr>
            </xsl:if>
          </xsl:for-each>
        </table>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

Figure 6.32: Librarybooks_stylesheet.xsl File

where,

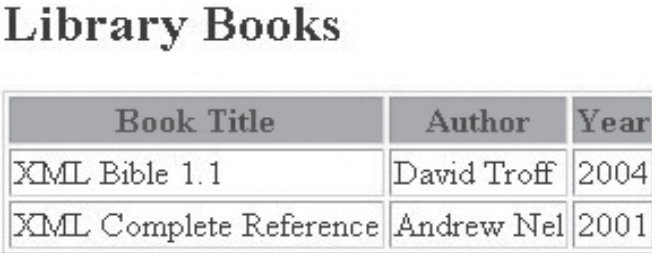
`<xsl:for-each select="Catalog/Book">`

Iterates through the `Book` node.

`<xsl:if test="Price > 50">`

Checks if the price of the book is greater than 50. If true, `Author`, `Title` and `Year` are displayed.

The output is shown in figure 6.33.



Library Books

Book Title	Author	Year
XML Bible 1.1	David Troff	2004
XML Complete Reference	Andrew Nel	2001

Figure 6.33: Output of Using xsl:if

6.2.10 The xsl:choose element

The `xsl:choose` element is used to make a decision when there are two or more possible courses of action. The `xsl:choose` element is used in conjunction with `xsl:when` and `xsl:otherwise` to express multiple conditional tests.

Figure 6.34 shows the syntax for `xsl:choose` element.

```
<xsl:choose>
  <xsl:when test="expression">
   template body
   </xsl:when>
   ...
  <xsl:otherwise>
    template body
  </xsl:otherwise>
</xsl:choose>
```

Figure 6.34: xsl:choose Syntax

where,

`xsl:when test="expression"`

The `xsl:when` element is examined in the order of occurrence. If the test expression is true, the code contained in that element is executed.

`xsl:otherwise`

If all the test conditions in any `xsl:when` element are false, then the `xsl:otherwise` element is automatically selected and the code associated with that element is executed.

Figure 6.35 shows the code for style rules that are defined in `Publisherbooks_stylesheet.xsl` file.

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="Publisherbooks_stylesheet.xsl"?>
<Publisher>
  <Book>
    <Title>XML By Example</Title>
    <Author>David Blake</Author>
    <Price>20.90</Price>
    <Year>1990</Year>
  </Book>
  <Book>
    <Title>XML Cookbook</Title>
    <Author>Susan Jones</Author>
    <Price>11.10</Price>
    <Year>1995</Year>
  </Book>
  <Book>
    <Title>XML Complete Reference </Title>
    <Author>Andrew Nel</Author>
    <Price>193</Price>
    <Year>2001</Year>
  </Book>
<Publisher>
```

Figure 6.35: XML File

Figure 6.36 depicts the style sheet for **publisherbooks_stylesheet.xsl** file.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <html>
      <body>
        <h2>Publisher Books</h2>
        <table border="1">
          <tr bgcolor="pink">
            <th>Title</th>
            <th>Author</th>
            <th>Price</th>
            <th>Year</th>
          </tr>
          <xsl:for-each select="Publisher/Book">
            <tr>
              <td>
                <xsl:value-of select="Title"/>
              </td>
              <xsl:choose>
                <xsl:when test="Price > 100">
                  <td bgcolor="magenta">
                    <xsl:value-of select="Author"/>
                  </td>
                  <td bgcolor="magenta">
                    <xsl:value-of select="Price"/>
                  </td>
                  <td bgcolor="magenta">
                    <xsl:value-of select="Year"/>
                  </td>
                </xsl:when>
                <xsl:otherwise>
                  <td>
                    <xsl:value-of select="Author"/>
                  </td>
                  <td>
                    <xsl:value-of select="Price"/>
                  </td>
                    <xsl:value-of select="Price"/>
                  </td>
                  <td>
                    <xsl:value-of select="Year"/>
                  </td>
                </xsl:otherwise>
              </xsl:choose>
            </tr>
          </xsl:for-each>
        </table>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

Figure 6.36: Publisherbooks_stylesheet.xsl File

where,

`xsl:when test="Price > 100"`

Checks whether the price of the book is greater than 100. If true, the details like `Author`, `Price` and `Year` are displayed with magenta as the background color.

`xsl:otherwise`

If all the conditions are false, this block is executed. Here, all other book details are printed in normal background color.

The output is shown in figure 6.37.

Publisher Books

Title	Author	Price	Year
XML By Example	David Blake	20.90	1990
XML Cookbook	Susan Jones	11.10	1995
XML Complete Reference	Andrew Nel	193	2001

Figure 6.37: Output of Using xsl:choose

6.2.11 Sorting in XSLT

The `xsl:sort` element in XSLT can be used to sort a group of similar elements. The sorting can be done in various ways by using the attributes of this element.

Figure 6.38 shows the syntax for `xsl:sort` element.

```
<xsl:sort
  case-order="upper-first" | "lower-first"
  data-type="number" "qname" | "text"
  order="ascending" | " descending"
  select="expression"
>
</xsl:sort>
```

Figure 6.38: xsl:sort Syntax

Explanation for highlighted areas in the image:

`case-order`

Indicates whether the sort will have upper or lowercase letters listed first in the sort output. The default option is to list uppercase first.

`data-type`

Specifies the data type of the strings.

`number`

Sort key is converted to a number.

`qname`

Sort is based upon a user-defined data type.

`text`

Specifies that the sort keys should be sorted alphabetically.

`order`

The sort order for the strings. The default value is "ascending".

`select`

Expression that defines the key upon which the sort will be based. The expression is evaluated and converted to a string that is used as the sort key.

Figure 3.69 shows the code for style rules that are defined in **products_stylesheet.xsl** file.

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="Products_stylesheet.xsl"?>
<Products>
  <Item>
    <ItemCode>CD01</ItemCode>
    <ItemName>Music CD</ItemName>
    <UnitPrice>9.90</UnitPrice>
  </Item>
  <Item>
    <ItemCode>PN01</ItemCode>
    <ItemName>Parker Pens</ItemName>
    <UnitPrice>12.50</UnitPrice>
  </Item>
  <Item>
    <ItemCode>CK01</ItemCode>
    <ItemName>Coca Cola</ItemName>
    <UnitPrice>2.20</UnitPrice>
  </Item>
  <Item>
    <ItemCode>BK01</ItemCode>
    <ItemName>Computer Books</ItemName>
    <UnitPrice>8.76</UnitPrice>
  </Item>
</Products>
```

Figure 6.39: XML File

Concepts

Figure 6.40 depicts the style sheet for **products_stylesheet.xsl** file.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:template match="/">
    <html>
      <body>
        <xsl:for-each select="Products/Item">
          <xsl:sort data-type="number" select="UnitPrice" order="descending"/>
          <xsl:value-of select="ItemName"/>
          <xsl:text>-</xsl:text>
          <xsl:value-of select="UnitPrice"/>
          <br/>
        </xsl:for-each>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

Figure 6.40: Products_stylesheet.xsl File

Explanation for highlighted areas in the style sheet:

`select="UnitPrice"`

Sort is based on `UnitPrice` element.

`order="descending"`

The sorting order for `UnitPrice` is descending in that the higher value will be displayed first.

The output is shown in figure 6.41.

Parker Pens-12.50
Music CD-9.90
Computer Books-8.76
Coca Cola-2.20

Figure 6.41: Output

Knowledge Check 2

1. Can you identify the correct code to display the following output "`March 20, 2001 The west coast of Atlanta some 150 dolphins`"?

(A)

XML File

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="stylesheet.xsl" ?>
<Article>
     <Date>March 20, 2001</Date>
     <Para>
        The west coast of
         <Place>Atlanta</Place>
         some 150 dolphins
     </Para>
</Article>
```

Style sheet File

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="Article">
     <xsl:apply-templates />
</xsl:template>
<xsl:template match="Date">
   <xsl:apply-templates/>
<xsl:template match="Para">
   <xsl:apply-templates/>
</xsl:stylesheet>
```

XML File

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="stylesheet.xsl" ?>
<Article>
     <Para>
        The west coast of
         <Place>Atlanta</Place>
           <Date>March 20, 2001</Date>
         some 150 dolphins
     </Para>
</Article>
```

(B)

Stylesheet File

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="Article">
     <xsl:apply-templates />
</xsl:template>
<xsl:template match="Date">
   <xsl:apply-templates/>
</xsl:template>
<xsl:template match="Para">
   <xsl:apply-templates/>
</xsl:template>
</xsl:stylesheet>
```

(C)

XML File

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="stylesheet.xsl" ?>
<Article>
     <Date>March 20, 2001</Date>
     <Para>
        The west coast of
         <Place>Atlanta</Place>
         some 150 dolphins
     </Para>
</Article>
```

Style sheet File

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="Article">
     <xsl:apply-templates />
</xsl:template>
<xsl:template match="Date">
   <xsl:apply-templates/>
</xsl:template>
<xsl:template match="Para">
   <xsl:apply-templates/>
</xsl:template>
</xsl:stylesheet>
```

(D)	**XML File**
	
	`<?xml version="1.0" encoding="UTF-8"?>`
	`<?xml-stylesheet type="text/xsl" href="stylesheet.xsl" ?>`
	`<Article>`
	`<Date>March 20, 2001</Date>`
	`<Para>`
	`<Place>The west coast of</Place>`
	`<Place>Atlanta</Place>`
	`<Place>some 150 dolphins</Place>`
	`</Para>`
	`</Article>`
	
	Stylesheet File
	
	`<xsl:stylesheet version="1.0"`
	`xmlns:xsl="http://www.w3.org/1999/XSL/Transform">`
	`<xsl:template match="Article">`
	`<xsl:apply-templates />`
	`</xsl:template>`
	`<xsl:template match="Date">`
	`<xsl:template match="Para">`
	`<xsl:apply-templates/>`
	`<xsl:apply-templates/>`
	`</xsl:template>`
	`</xsl:template>`
	`</xsl:stylesheet>`

2. Can you match the XSL elements against their corresponding description?

	Description		XPath Node
(A)	Puts a conditional test against the content of the XML file	(1)	`xsl:value-of`
(B)	Adds literal text to the output	(2)	`xsl:for-each`
(C)	Extracts the value of a selected node	(3)	`xsl:text`
(D)	Applies a template repeatedly	(4)	`xsl:choose`
(E)	Inserts a multiple conditional test against the XML file	(5)	`xsl:if`

3. Can you identify the correct code to display the following output "A. David Blake18/11/1973"?

	XML File
	`<?xml version="1.0" encoding="UTF-8"?>`
	`<?xml-stylesheet type="text/xsl" href="stylesheet.xsl"?>`
	`<MichiganStaff>`
	`<Faculty>`
	`<Name>David Blake</Name>`
	`<DOB>18/11/1973</DOB>`
	`</Faculty>`
	`</MichiganStaff>`

(A)	**Style sheet File**
	`<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">`
	`<xsl:template match="/">`
	`<xsl:for-each select="MichiganStaff/Faculty/Name">`
	`<xsl:for-each select="MichiganStaff/Faculty">`
	`<xsl:number value="position()" sequence="A."/>`
	`<xsl:value-of select="Name"/>`
	`<xsl:value-of select="DOB"/>`
	`</xsl:for-each>`
	`</xsl:template>`
	`</xsl:stylesheet>`

XML File

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="stylesheet.xsl"?>
<MichiganStaff>
    <Faculty>
        <Name>David Blake</Name>
        <DOB>18/11/1973</DOB>
    </Faculty>
</MichiganStaff>
```

(B) **Stylesheet File**

```
<xsl:stylesheet    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
     <xsl:template match="/">
<xsl:for-each select="MichiganStaff/Faculty/Name">
<xsl:for-each select="MichiganStaff/Faculty/DOB">
             <xsl:number value="position()" format="A."/>
             <xsl:value-of select="Name"/>
             <xsl:value-of select="DOB"/>
         </xsl:for-each>
     </xsl:template>
</xsl:stylesheet>
```

XML File

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="stylesheet.xsl"?>
<MichiganStaff>
    <Faculty>
        <Name>David Blake</Name>
        <DOB>18/11/1973</DOB>
    </Faculty>
</MichiganStaff>
```

(C) **Style sheet File**

```
<xsl:stylesheet    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
     <xsl:template match="/">
         <xsl:for-each select="MichiganStaff">
             <xsl:number value="position()" format="A."/>
             <xsl:value-of select="Name"/>
             <xsl:value-of select="DOB"/>
         </xsl:for-each>
     </xsl:template>
</xsl:stylesheet>
```

XML File

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="stylesheet.xsl"?>
<MichiganStaff>
    <Faculty>
        <Name>David Blake</Name>
        <DOB>18/11/1973</DOB>
    </Faculty>
</MichiganStaff>
```

(D) **Style sheet File**

```
<xsl:stylesheet    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
     <xsl:template match="/">
         <xsl:for-each select="MichiganStaff/Faculty">
             <xsl:number value="position()" format="A."/>
             <xsl:value-of select="Name"/>
             <xsl:value-of select="DOB"/>
         </xsl:for-each>
     </xsl:template>
</xsl:stylesheet>
```

Module Summary

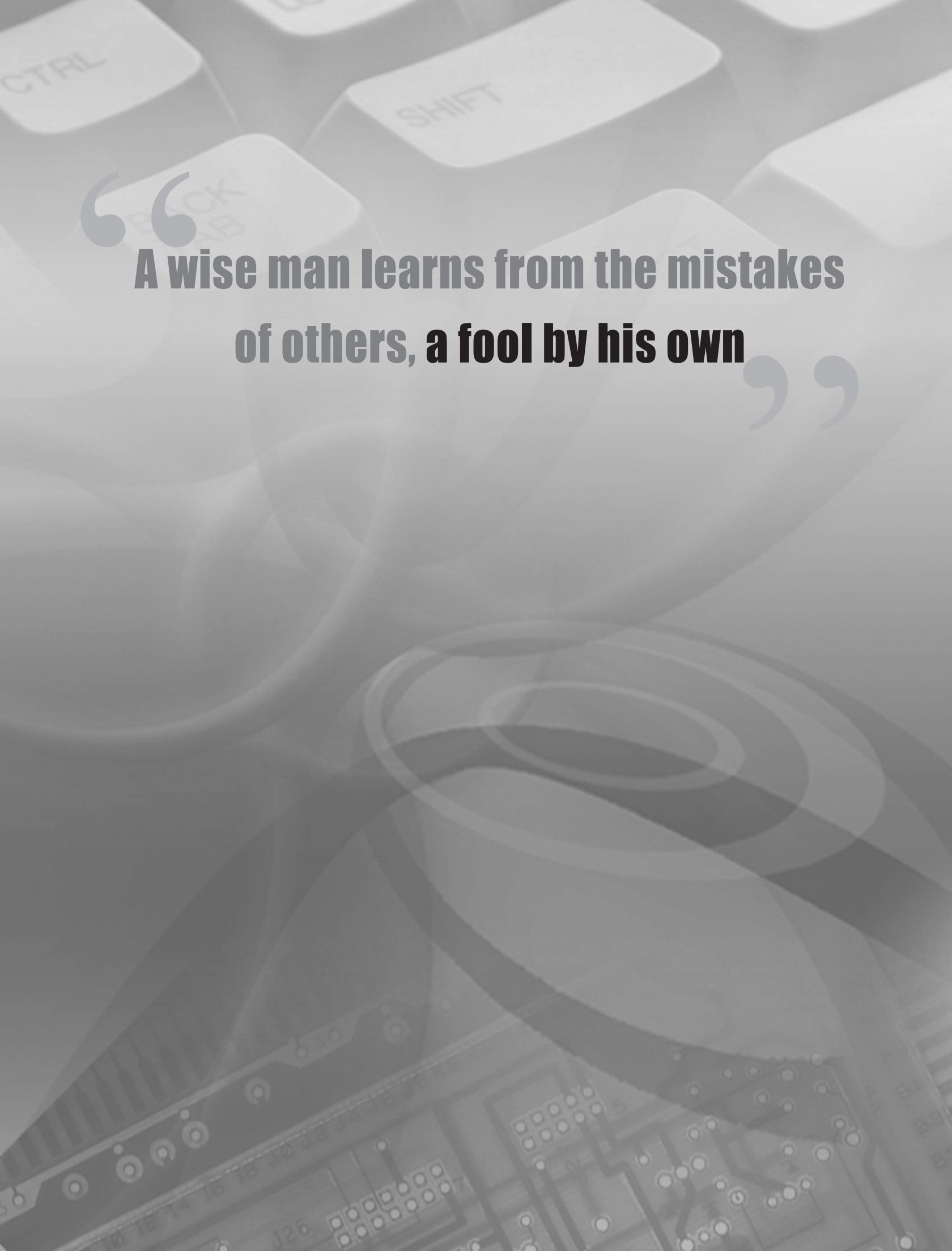
In this module, **XSL and XSLT**, you learnt about:

- **Introduction to XSL**

  XML provides the ability to format document content. XSL provides the ability to define how the formatted XML content is presented. An XSL Transformation applies rules to a source tree read from an XML document to transform it into an output tree written out as an XML document.

- **Working with XSL**

  An XSL template rule is represented as an `xsl:template` element. You can process multiple elements in two ways: using the `xsl:apply-templates` element and the `xsl:for-each` element. The `xsl:stylesheet` element allows you to include a style sheet directly in the document it applies to. The `xsl:if` element produces output if, and only if, its `test` attribute is `true`.


A wise man learns from the mistakes
of others, a fool by his own

Module

7

More on XSLT

Concepts

Module Overview

Welcome to the module, **More on XSLT**. This module aims at giving a clear understanding on XPath, and identifying the various nodes of XPath. This module lists the different operators used with XPath and describes the various XPath expressions and functions. Finally, this module explains how to switch between styles and how to transform XML documents into HTML using XSLT.

In this module, you will learn about:

- XPath
- XPath Expressions and Functions
- Working with different styles

7.1 XPath

In this first lesson, **XPath**, you will learn to:

- Define and describe XPath.
- Identify nodes according to XPath.
- List operators used with XPath.
- Describe the types of matching.

7.1.1 XPath

XPath can be thought of as a query language like SQL. However, rather than extracting information from a database, it extracts information from an XML document. XPath is a language for retrieving information from a XML document. XPath is used to navigate through elements and attributes in an XML document. Thus, XPath allows identifying parts of an XML document.

XPath provides a common syntax as shown in figure 7.1 for features shared by XSLT and XQuery.

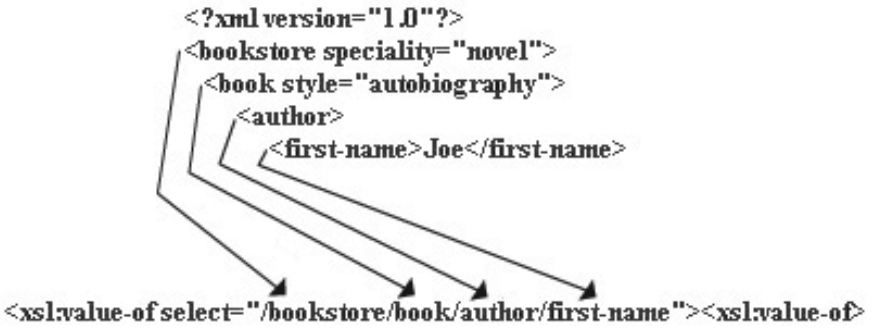


Figure 7.1: XPath

- **XSLT**

  XSLT is a language for transforming XML documents into XML, HTML, or text.

- **XQuery**

  XQuery builds on XPath and is a language for extracting information from XML documents.

7.1.2 Benefits of XPath

XPath is designed for XML documents. It provides a single syntax that you can use for queries, addressing, and patterns. XPath is concise, simple, and powerful.

XPath has many benefits:

- Syntax is simple for the simple and common cases.
- Any path that can occur in an XML document and any set of conditions for the nodes in the path can be specified.
- Any node in an XML document can be uniquely identified.

XPath is designed to be used in many contexts. It is applicable to providing links to nodes, for searching repositories, and for many other applications.

7.1.3 XML Document in XPath

In XPath, an XML document is viewed conceptually as a tree in which each part of the document is represented as a node as shown in figure 7.2.

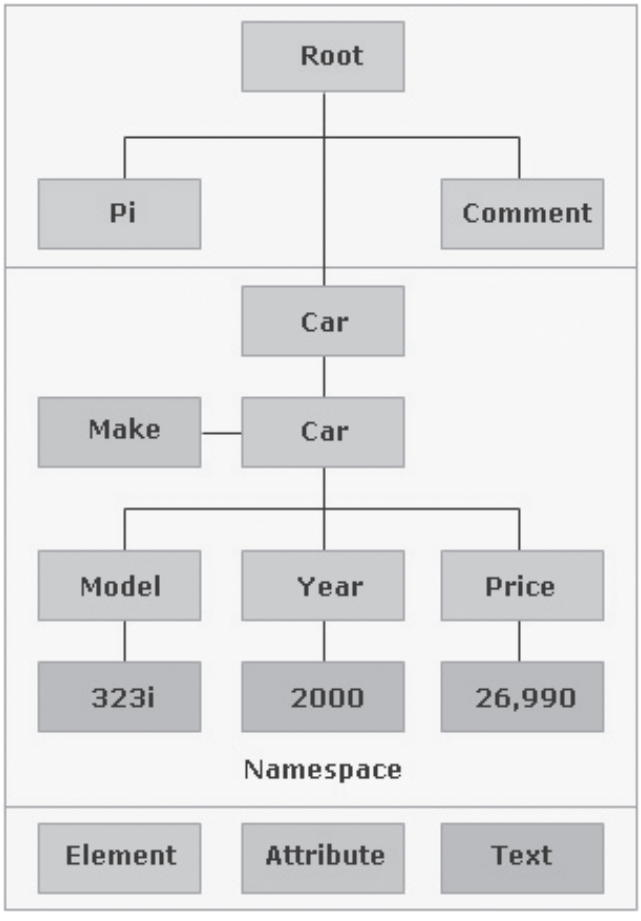


Figure 7.2: XML Document in XPath

XPath have seven types of nodes. They are:

- **Root**

  The XPath tree has a single `root` node, which contains all other nodes in the tree.

- **Element**

  Every element in a document has a corresponding `element` node that appears in the tree under the `root` node. Within an `element` node appear all of the other types of nodes that correspond to the element's content. Element nodes may have a unique identifier associated with them that is used to reference the node with XPath.

- **Attribute**

  Each `element` node has an associated set of attribute nodes; the element is the parent of each of these attribute nodes; however, an attribute node is not a child of its parent element.

- **Text**

  Character data is grouped into text nodes. Characters inside comments, processing instructions and attribute values do not produce text nodes. The `text` node has a parent node and it may be the `child` node too.

- **Comment**

  There is a `comment` node for every comment, except for any comment that occurs within the document type declaration. The `comment` node has a parent node and it may be the child node too.

- **Processing instruction**

  There is a `processing instruction` node for every processing instruction, except for any processing instruction that occurs within the document type declaration. The `processing instruction` node has a `parent` node and it may be the `child` node too.

- **Namespace**

  Each element has an associated set of namespace nodes. Although the `namespace` node has a `parent` node, the `namespace` node is not considered a child of its parent node because they are not contained in a `parent` node, but are used to provide descriptive information about their `parent` node.

A summary of different types of nodes in XPath is given in table 7.1.

Node Type	String Value	Expanded Name
root	Determined by concatenating the string-values of all text-node descendents in document order.	None.
element	Determined by concatenating the string-values of all text-node descendents in document order.	The element tag, including the namespace prefix (if applicable).
attribute	The normalized value of the attribute.	The name of the attribute, including the namespace prefix (if applicable).
text	The character data contained in the text node.	None.
comment	The content of the comment (not including <!-- and -->).	None.
processing instruction	The part of the processing instruction that follows the target and any whitespace.	The target of the processing instruction.
namespace	The URI of the namespace.	The namespace prefix.

Table 7.1: Different Types of Nodes

7.1.4 XPath Representation

An XPath query operates on a well-formed XML document after it has been parsed into a tree structure. Figure 7.3 depicts a simple XML document along with its generated XPath tree.

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

<!-- Fig. 11.1 : simple.xml -->
<!-- Simple XML document -->

<book title = "C++ How to Program" edition = "3">
<sample>
<![CDATA[
// C++ comment
 if ( this->getX() < 5 && value[ 0 ] != 3 )
cerr << this->displayError();
 ]]>
 </sample>

C++ How to Program by Deitel &amp;amp; Deitel
</book>
```

Figure 7.3: XML Document

The corresponding XPath tree that is generated is shown in figure 7.4.

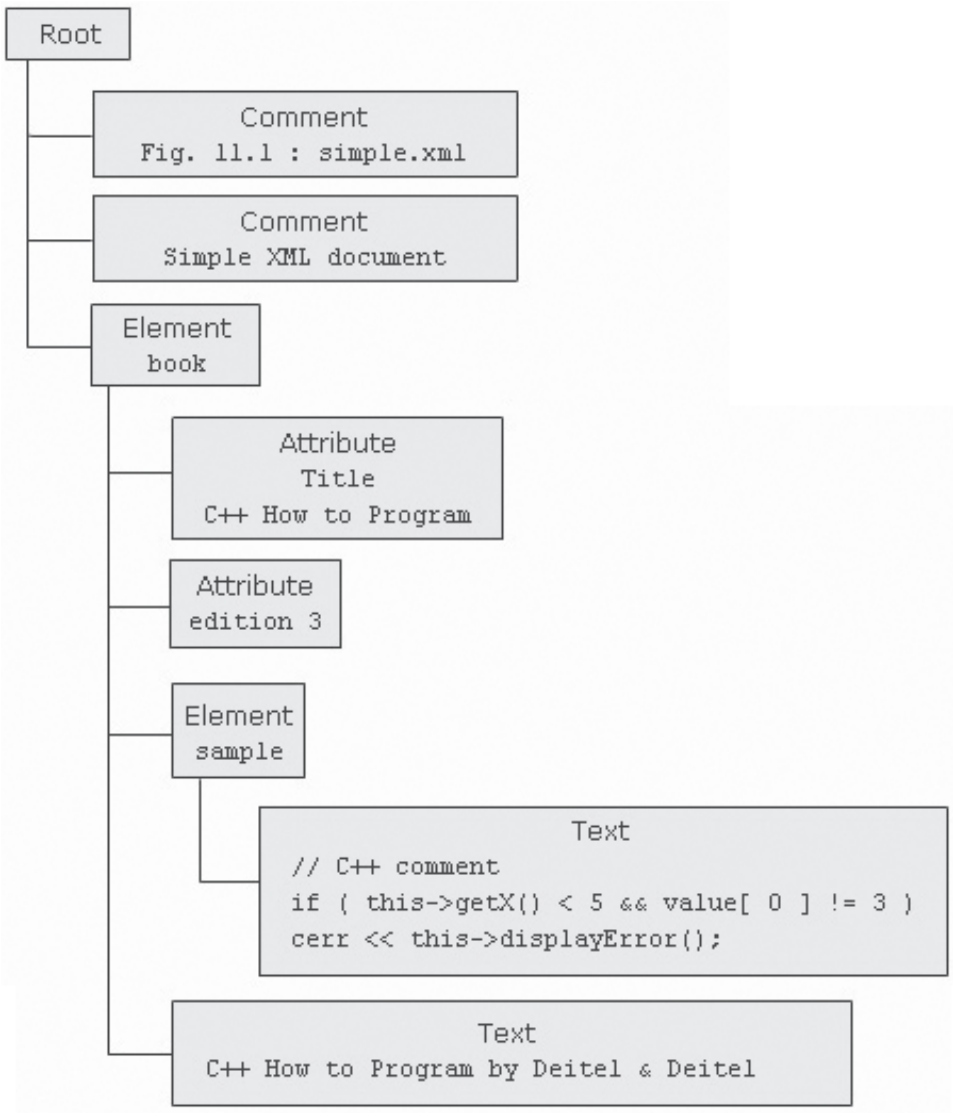


Figure 7.4: XPath Tree of XML Document

7.1.5 Operators in XPath

An XPath expression returns a node set, a boolean, a string, or a number. XPath provides basic floating point arithmetic operators and some comparison and boolean operators.

The XPath expressions are constructed using the operators and special characters as shown in table 7.2.

Operator	Description
/	Child operator; selects immediate children of the left-side collection
//	Recursive descent; searches for the specified element at any depth
.	Indicates the current context
..	The parent of the current context node
*	Wildcard; selects all elements regardless of the element name
@	Attribute; prefix for an attribute name
:	Namespace separator; separates the namespace prefix from the element or attribute name

Table 7.2: XPath Operators

Note: In floating point operators, instead of / and % the keywords div and mod are used respectively.

7.1.6 Examples of XPath Operators

Table 7.3 shows few examples along with their description that use XPath operators in an expression.

Expression	Refers to
Author/FirstName	All <FirstName> elements within an <Author> element of the current context node
BookStore//Title	All <Title> elements one or more levels deep in the <BookStore> element
BookStore/*/Title	All <Title> elements that are grandchildren of <BookStore> elements
BookStore//Book/Excerpt//Work	All <Work> elements anywhere inside <Excerpt> children of <Book> elements, anywhere inside the <BookStore> element
.//Title	All <Title> elements one or more levels deep in the current context

Table 7.3: Examples of XPath Operators

Additionally, a list of the operators that can be used in XPath expressions is given in table 7.4.

Operator	Description	Example	Return value
|	Computes two node-sets	//book | //cd	Returns a node-set with all book and cd elements.
+	Addition	36+14	50
-	Subtraction	36-14	22
*	Multiplication	6*4	24
div	Division	28 div 14	2
=	Equal	price=9.50	true, if price is 9.50 false, if price is 9.60
!=	Not Equal	price!=9.50	true, if price is 9.90 false, if price is 9.50
<	less than	price<9.50	true, if price is 9.00 false, if price is 9.50
<=	less than or equal to	price<=9.50	true, if price is 9.00 false, if price is 9.60
>	greater than	price>9.50	true, if price is 9.60 false, if price is 9.50
>=	greater than or equal to	price>=9.50	true, if price is 9.60 false, if price is 9.40
or	or	price=9.50 or price=9.90	true, if price is 9.90 false, if price is 9.40
and	and	price=9.50 and price=9.90	true, if price is 9.70 false, if price is 8.70
mod	Modulus (division remainder)	15 mod 4	3

Table 7.4: List of Operators

7.1.7 Types of Matching

XPath is used in the creation of the patterns. The `match` attribute of the `xsl:template` element supports a complex syntax that allows to express exactly which nodes to be matched. The `select` attribute of `xsl:apply-templates`, `xsl:value-of`, `xsl:for-each`, `xsl:copy-of`, and `xsl:sort` supports an even more powerful superset of this syntax that allows to express exactly which nodes to selected and which nodes not to be selected.

Some important types of matching are:

- **Matching by name**

  The source element is simply identified by its name, using the `match` attribute. The value given to the `match` attribute is called the pattern. The following code demonstrate the example.

  Code Snippet:

  ```
  <xsl:template match = "Greeting">
  ```

  matches all greeting elements in the source document.

- **Matching by ancestry**

  As in CSS, a match can be made using the element's ancestry. The following tag will match any '`EM`' element that has '`P`' as an ancestor.

  Code Snippet:

  ```
  <xsl:template match = "P//EM">
  ```

- **Matching the element names**

  The most basic pattern contains a single element name which matches all elements with that name. The following code demonstrate the example. This template matches `Product` elements and marks their `Product_ID` children bold.

  Code Snippet:

  ```
  <xsl:template match="Product">
  <xsl:value-of select="Product_ID"/>
  </xsl:template>
  ```

- **Matching the root**

To enable all the descendant nodes to inherit the properties on the root of document, use a single forward slash to represent the root. The following code demonstrate the example.

Code Snippet:

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

will select the root pattern.

- **Matching by attribute**

The syntax used to match the attribute is:

Syntax:

```
<xsl:template match = " element name['attribute' (attribute-name)=attribute-value]">
```

This syntax uses square brackets to hold the attribute name and value.

The following code demonstrate the example.

Code Snippet:

```
<xsl:template match="Product">
  <xsl:apply-templates select="@Units"/>
</xsl:template>
```

The given example applies the templates to the non-existent Units attributes of Product elements.

Knowledge Check 1

1. Which of these statements about XPath are true and which of these are false?

(A)	XPath provides multiple syntax that can be used for queries, addressing and patterns.
(B)	XPath can be thought of as a query language like SQL.
(C)	In XPath, the structure of an XML document is viewed conceptually as a pyramid.
(D)	XPath provides a common syntax for features shared by XSLT and XQuery.
(E)	XPath is used to navigate through elements and attributes in an XML document.

2. Can you match the XPath nodes against their corresponding description?

	Description		XPath Node
(A)	Has a parent node and it may be the `child` node too	**(1)**	`Element`
(B)	Contains all other nodes in the tree	**(2)**	`Attribute`
(C)	Is not considered a child of its `parent` node because they are not contained in a `parent` node	**(3)**	`Text`
(D)	May have a unique identifier associated with them, which is useful when referencing the node with XPath	**(4)**	`Namespace`
(E)	Has a `parent` node that is either an element or root node	**(5)**	`Root`

3. Can you match the different types of matching against their corresponding description?

	Description		Types of Matching

(A)	`<xsl:template match="/">`
	`...`
	`</xsl:template>`
		(1)	`Matching by name`

(B)	`<xsl:template match = "Greeting">`	**(2)**	`Matching by ancestry`
(C)	`<xsl:template match = "P//EM">`	**(3)**	`Matching the element names`

(D)	`<xsl:template match="Product">`
	`<xsl:apply-templates select="@Unit"/>`
	`</xsl:template>`
		(4)	`Matching the root`

(E)	`<xsl:template match="Product">`
	`<xsl:value-of select="Product_ID"/>`
	`</xsl:template>`
		(5)	`Matching by attribute`

Concepts

7.2 XPath Expressions and Functions

In this second lesson, **XPath Expressions and Functions**, you will learn to:

- State and explain the various XPath expressions and functions.
- List the node set functions.
- List the boolean functions.
- State the numeric functions.
- Describe the string functions.
- Explain what result tree fragments are.

7.2.1 XPath Expressions

XPath Expressions are statements that can extract useful information from the XPath tree. Instead of just finding nodes, one can count them, add up numeric values, compare strings, and more. They are much like statements in a functional programming language. Every XPath expression evaluates to a single value.

There are four types of expressions in XPath. They are:

- **Node-set**

  A `node-set` is an unordered group of nodes from the input document that match an expression's criteria.

- **Boolean**

  A `boolean` has one of two values: `true` or `false`. XSLT allows any kind of data to be transformed into a `boolean`. This is often done implicitly when a `string` or a `number` or a `node-set` is used where a `boolean` is expected.

- **Number**

  XPath numbers are numeric values useful for counting nodes and performing simple arithmetic. The numbers like 43 or –7000 that look like integers are stored as doubles. Non-number values, such as strings and booleans, are converted to numbers automatically as necessary.

- **String**

  A `String` is a sequence of zero or more Unicode characters. Other data types can be converted to strings using the `string()` function.

Note: The XPath expression syntax includes literal forms for strings and numbers as well as operators and functions for manipulating all four XPath data types.

7.2.2 XPath Functions

XPath defines various functions required for XPath 2.0, XQuery 1.0 and XSLT 2.0. The different functions are `Accessor`, `AnyURI`, `Node`, `Error` and `Trace`, `Sequence`, `Context`, `Boolean`, `Duration/Date/Time`, `String`, `QName` and `Numeric`.

XML Path Language (XPath) functions can be used to refine XPath queries and enhance the programming power and flexibility of XPath. Each function in the function library is specified using a function prototype that provides the return type, function name, and argument type. If an argument type is followed by a question mark, the argument is optional; otherwise, the argument is required. Function names are case-sensitive.

The default prefix for the function namespace is `fn`.

7.2.3 Node-set Functions

`Node-set` functions take a `node-set` argument. These return a `node-set`, or information about a particular node within a `node-set`. The different `node-set` functions are `name()`, `local-name()`, `namespace-uri()` and `root()`.

Figure 7.5 shows the syntax for `node-set` functions.

```
name()
fn:name()
fn:name(nodeset)

local-name()
fn:local-name()
fn:local-name(nodeset)

namespace-uri()
fn:namespace-uri()
fn:namespace-uri(nodeset)

root()
fn:root()
fn:root(node)
```

Figure 7.5: Node-set Functions

where,

`name()`

The function returns the name of the current node or the first node in the specified node-set.

`local-name()`

The function returns the name of the current node or the first node in the specified node-set without the namespace prefix.

`namespace-uri()`

The function returns the namespace URI of the current node or the first node in the specified node-set.

`root()`

The function returns the root of the tree to which the current node or the specified node belongs. This will usually be a document node.

Figure 7.6 shows the code and schema.

```
<!--Book.xml -->
<?xml-stylesheet type="text/xsl" href="Sample.xsl"?>
<Catalog xmlns="http://www.BookCatalog.com"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.BookCatalog.com BookSchema.xsd">
        <Book>
                <Author>Gambardella, Matthew</Author>
                <Title>XML Developer's Guide</Title>
                <Genre>Computer</Genre>
                <Price>44.95</Price>
                <Publish>2000-10-01</Publish>
                <Description>An in-depth look at creating applications
           with XML.</Description>
        </Book>
</Catalog>
```

```
<!—BookSchema.xsd -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
   targetNamespace="http://www.BookCatalog.com"
   xmlns="http://www.BookCatalog.com"
   elementFormDefault="qualified">

   <!--definition for simple elements-->

   <xs:element name="Author" type="xs:string"/>
   <xs:element name="Title" type="xs:string"/>
   <xs:element name="Genre" type="xs:string"/>
   <xs:element name="Price" type="xs:float"/>
   <xs:element name="Publish" type="xs:string"/>
   <xs:element name="Description" type="xs:string"/>

   <!--definition for complex elements-->
   <xs:element name="Book">
     <xs:complexType>
       <xs:sequence>
         <xs:element ref="Author"/>
         <xs:element ref="Title"/>
         <xs:element ref="Genre"/>
         <xs:element ref="Price"/>
         <xs:element ref="Publish"/>
         <xs:element ref="Description"/>
       </xs:sequence>
     </xs:complexType>
   </xs:element>

   <xs:element name="Catalog">
     <xs:complexType>
       <xs:sequence>
         <xs:element ref="Book" minOccurs="1" maxOccurs="unbounded"/>
       </xs:sequence>
     </xs:complexType>
   </xs:element>
</xs:schema>
```

Figure 7.6: Code and Schema

Figure 7.7 depicts the style sheet.

```
<!—Sample.xsl -->
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
   <xsl:output method="html"/>
  <xsl:template match="/">
      <html>
         <body>
            <h3>Node-set Function</h3>

            <table width="100%" border="1">
               <tr>
                  <td width="25%"><b>namespace-uri()</b></td>
                  <td width="25%"><b>name()</b></td>
                  <td width="25%"><b>local-name</b></td>
                  <td width="25%"><b>text()</b></td>
               </tr>
               <xsl:apply-templates />
            </table>
         </body>
      </html>
  </xsl:template>

  <xsl:template match="*">
      <tr>
         <td>
            <xsl:value-of select="namespace-uri()"/>
         </td>
         <td>
            <xsl:value-of select="name()"/>
         </td>
         <td>
            <xsl:value-of select="local-name()"/>
         </td>
         <td>
            <xsl:value-of select="text()"/>
         </td>
      </tr>
      <xsl:apply-templates select="*"/>
</xsl:template>
</xsl:stylesheet>
```

Figure 7.7: Style sheet

The output is shown in figure 7.8. The formatted output gets displayed as shown here.

Node-set Function

namespace-uri()	name()	local-name	text()
http://www.BookCatalog.com	Catalog	Catalog	
http://www.BookCatalog.com	Book	Book	
http://www.BookCatalog.com	Author	Author	Gambardella, Matthew
http://www.BookCatalog.com	Title	Title	XML Developer's Guide
http://www.BookCatalog.com	Genre	Genre	Computer
http://www.BookCatalog.com	Price	Price	44.95
http://www.BookCatalog.com	Publish	Publish	2000-10-01
http://www.BookCatalog.com	Description	Description	An in-depth look at creating applications with XML.

Figure 7.8: Output of Using Node-set Functions

7.2.4 Boolean Functions

The XML Path Language (XPath) syntax supports boolean functions that return true or false, and can be used with comparison operators in filter patterns. Some of these functions are `boolean`, `not`, `true` and `false`.

- **boolean(arg)**

  The function returns a boolean value for a number, string, or node-set. The syntax, code, and output are shown.

  Syntax:

  ```
  fn:boolean(arg)
  ```

  Code Snippet:

```
<ul>
    <li><b>boolean(0)</b> = <xsl:value-of select="boolean(0)"/>
    </li>
    <li><b>boolean(1)</b> = <xsl:value-of select="boolean(1)"/>
    </li>
    <li><b>boolean(-100)</b> = <xsl:value-of select="boolean(-100)"/>
    </li>
```

```
    <li><b>boolean('hello')</b> = <xsl:value-of select="boolean('hello
')"/>
    </li>
    <li><b>boolean('')</b> = <xsl:value-of select="boolean('')"/>
    </li>
<li><b>boolean(//book)</b> = <xsl:value-of select="boolean(//book)"/>
    </li>
</ul>
```

Output:

```
boolean(0) = false
boolean(1) = true
boolean(-100) = true
boolean('hello') = true
boolean('') = false
boolean(//book) = false
```

- **not(arg)**

The sense of an operation can be reversed by using the `not()` function. The syntax and code is shown.

Syntax:

```
fn:not(arg)
```

Code Snippet:

This template rule selects all Product elements that are not the first child of their parents:

```
<xsl:template match="PRODUCT[not(position()=1)]">
    <xsl:value-of select="."/>
</xsl:template>
```

The same template rule could be written using the `not equal` operator `!=` instead:

Code Snippet:

```
<xsl:template match="PRODUCT[position()!=1]">
    <xsl:value-of select="."/>
</xsl:template>
```

- **true()**

  The `true()` function returns the boolean value true. The syntax, code, and output are shown.

  Syntax:

  ```
  fn:true()
  ```

  Code Snippet:

  ```
  <xsl:value-of select="true()"/>
  ```

  Output:

  ```
  true
  ```

- **false()**

  The `false()` function returns the boolean value true. The syntax, code, and output are shown.

  Syntax:

  ```
  fn:false()
  ```

  Code Snippet:

  The code snippet shows how to use `true()` and `false()` in XSL.

  ```
  <xsl:value-of select="false() or false()"/>
  <xsl:value-of select="true() and false()"/>
  <xsl:value-of select="false() and false()"/>
  ```

  Output:

  ```
  true
  false
  false
  ```

The value derived from an expression depends on some rules as shown in table 7.5.

Expression Type	Rule
Node-set	True if the set contains at least one node, false if it is empty.
String	True unless the string is zero-length.

Number	True unless the value is zero or NaN (not a number).
Result tree fragment	Always true, because every fragment contains at least one node, its root node.

Table 7.5: Boolean Conversion Rules

Certain operators compare numerical values to arrive at a Boolean value. All the nodes in a node-set are tested to determine whether any of them satisfies the comparison or not. The Comparison operators are shown in table 7.6.

Operator	Returns
expr = expr	True if both expressions (string or numeric) have the same value, otherwise false.
expr != expr	True if the expressions do not have the same value (string or numeric), otherwise false.
expr < expr	True if the value of the first numeric expression is less than the value of the second, otherwise false.
expr > expr	True if the value of the first numeric expression is greater than the value of the second, otherwise false.
expr <= expr	True if the value of the first numeric expression is less than or equal to the value of the second, otherwise false.
expr >= expr	True if the value of the first numeric expression is greater than or equal to the value of the second, otherwise false.

Table 7.6: Comparison Operators

Note: To use some of the comparison operators inside an XML document such as an XSLT style sheet or a schema, one must use character references < and > instead of < and >.

The different functions that return Boolean functions are shown in table 7.7.

Function	Returns
expr and expr	True if both Boolean expressions are true, otherwise false.
expr or expr	True if at least one Boolean expression is true, otherwise false.
true()	True.
false()	False.
not(expr)	Negates the value of the Boolean expression: true if the expression is false, otherwise false.

Table 7.7: Boolean Functions

7.2.5 Numeric Functions

XPath syntax supports number functions that return strings or numbers and can be used with comparison operators in filter patterns. The different numeric functions are `number(arg)`, `ceiling(num)`, `floor(num)` and `round(num)`.

The rules for converting any expression into a numeric value are listed in table 7.8.

Expression Type	Rule
Node-set	The first node is converted into a string, then the string conversion rule is used.
Boolean	The value true is converted to the number 1, and false to the number 0.
String	If the string is the literal serialization of a number (i.e., -123.5), it is converted into that number. Otherwise, the value NaN is used.
Result-tree fragment	Like node-sets, a result-tree fragment is converted into a string, which is then converted with the string rule.

Table 7.8: Expressions into Numbers Conversion Rules

To manipulate numeric values, there are a variety of operators and functions as shown in table 7.9.

Function	Returns
expr + expr	The sum of two numeric expressions.
expr - expr	The difference of the first numeric expression minus the second.
expr * expr	The product of two numeric expressions.
expr div expr	The first numeric expression divided by the second expression.
expr mod expr	The first numeric expression modulo the second expression.
round(expr)	The value of the expression rounded to the nearest integer.
sum(node-set)	The sum of the values of the nodes in node-set. Unlike the other functions in this table, this function operates over a node-set instead of expressions.

Table 7.9: Numeric Operators and Functions

Figure 7.9 shows the syntax for numeric functions.

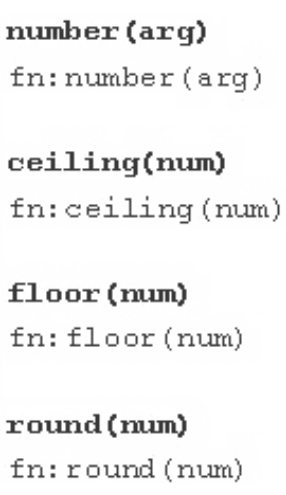


Figure 7.9: Numeric Functions

where,

`number(arg)`

The function returns the numeric value of the argument. The argument could be a boolean, a string, or a node-set.

`ceiling(num)`

The function returns the smallest integer that is greater than the number argument.

`floor(num)`

The function returns the largest integer that is not greater than the number argument.

`round(num)`

The function rounds the number argument to the nearest integer.

The following code depicts the style sheet for numeric functions.

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="Number.xsl"?>
<xsl:stylesheet    version="1.0"    xmlns:xsl="http://www.w3.org/1999/XSL/
Transform">
    <xsl:output method="html"/>
    <xsl:template match="/">
    <html>
        <body>
            <h3>Numeric Functions</h3>
            <ul>
                <li>
                    <b>number('1548')</b>
                    =
                    <xsl:value-of select="number('1548')"/>
                </li>
                <li>
                    <b>number('-1548')</b>
                    =
                    <xsl:value-of select="number('-1548')"/>
                </li>
                <li>
                    <b>number('text')</b>
                    =
                    <xsl:value-of select="number('text')"/>
                </li>
                 <li>
                    <b>number('226.38' div '1')</b>
                    =
                    <xsl:value-of select="number('226.38' div '1')"/>
                </li>
            </ul>
            <ul>
                 <li>
                    <b>ceiling(2.5)</b>
                    =
                    <xsl:value-of select="ceiling(2.5)"/>
```

```
            </li>
                <b>ceiling(-2.3)</b>
                =
                <xsl:value-of select="ceiling(-2.3)"/>
            </li>
            <li>
                <b>ceiling(4)</b>
                =
                <xsl:value-of select="ceiling(4)"/>
            </li>
        </ul>
        <ul>
            <li>
                <b>floor(2.5)</b>
                =
                <xsl:value-of select="floor(2.5)"/>
            </li>
            <li>
                <b>floor(-2.3)</b>
                =
                <xsl:value-of select="floor(-2.3)"/>
            </li>
            <li>
                <b>floor(4)</b>
                =
                <xsl:value-of select="floor(4)"/>
            </li>
        </ul>
        <ul>
            <li>
                <b>round(3.6)</b>
                =
                <xsl:value-of select="round(3.6)"/>
            </li>
            <li>
                <b>round(3.4)</b>
                =
                <xsl:value-of select="round(3.4)"/>
```

```
                </li>
                <li>
                    <b>round(3.5)</b>
                    =
                    <xsl:value-of select="round(3.5)"/>
                </li>
                <li>
                    <b>round(-0.6)</b>
                    =
                    <xsl:value-of select="round(-0.6)"/>
                </li>
                <li>
                    <b>round(-2.5)</b>
                    =
                    <xsl:value-of select="round(-2.5)"/>
                </li>
            </ul>
        </body>
    </html>
     </xsl:template>
</xsl:stylesheet>
```

The output is shown in figure 7.10.

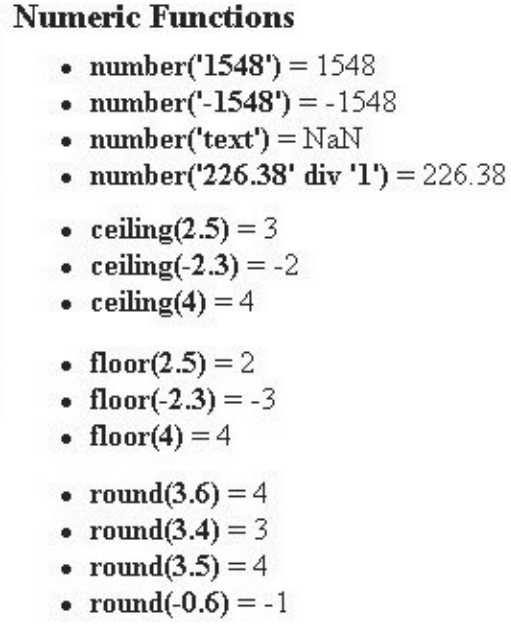


Figure 7.10: Output of Using Numeric Functions

Concepts

7.2.6 String Functions

String functions are used to evaluate, format, and manipulate string arguments, or to convert an object to a string. The different String functions are shown in table 7.10.

Function	Returns
concat(string, string, . . .)	A string that is the concatenation of the `string` arguments.
format-number(number, pattern, decimal-format)	A `string` containing the number, formatted according to `pattern`. The optional `decimal-format` argument points to a format declaration which assigns special characters like the grouping character, which separates groups of digits in large numbers for readability.
normalize-space(string)	The string with leading and trailing whitespace removed, and all other strings of whitespace characters replaced with single spaces.
substring(string, offset, range)	A substring of the string argument, starting `offset` characters from the beginning and ending `range` characters from the offset.
substring-after(string, to-match)	A substring of the string argument, starting at the end of the first occurrence of the string to-match and ending at the end of string.
substring-before(string, to-match)	A substring of the string argument, starting at the beginning of string and ending at the beginning of the first occurrence of the string `to-match`.
translate(string, characters-to-match, characters-replace-with)	The string with all characters in the string `characters-to-match` replaced with their counterpart characters in the string `characters-replace-with`.

Table 7.10: String Functions

Some functions operate on strings and return numeric or Boolean values, are listed in table 7.11.

Function	Returns
contains(string, sub)	True if the given substring sub occurs within the string, otherwise false.
starts-with(string, sub)	True if the string begins with the substring sub, otherwise false.
string-length(string)	The number of characters inside the string.

Table 7.11: Other String Functions

Figure 7.11 shows the syntax for string functions.

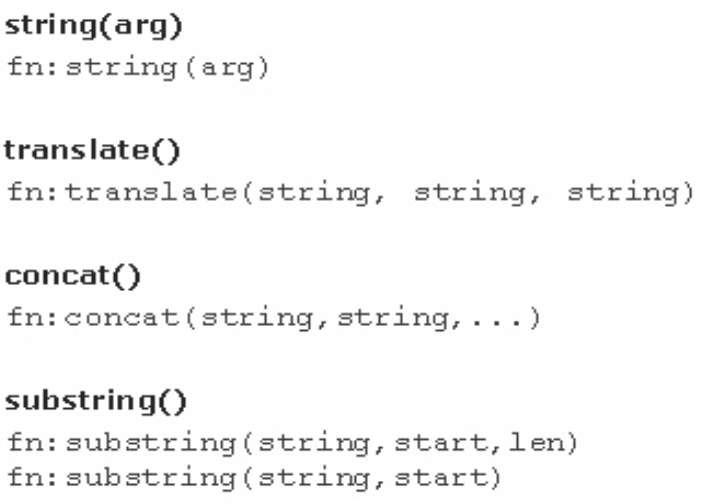


Figure 7.11: String Functions

where,

`string(arg)`

The function returns the string value of the argument. The argument could be a number, boolean, or node-set. For example: `string(314)` returns "314".

`translate()`

The function returns the first argument string with occurrences of characters in the second argument string replaced by the character at the corresponding position in the third argument string. For example: `translate("bar","abc","ABC")` returns `Bar`.

`concat()`

The function returns the concatenation of the strings. For example: `concat('XPath ','is ','FUN!')` returns 'XPath is FUN!'.

`substring()`

The function returns the substring from the start position to the specified length. Index of the first character is 1. For example: `substring('Beatles',1,4)` returns 'Beat'.

Figure 7.12 shows the code in XML file.

```
1  <?xml version="1.0"?>
2  <?xml-stylesheet type="text/xsl" href="BookDetail.xsl"?>
3  <BookStore>
4    <Book>
5      <Title>The Weather Pattern</Title>
6      <Author>Weather Man</Author>
7      <Price>100.00</Price>
8    </Book>
9    <Book>
10     <Title>Weaving Patterns</Title>
11     <Author>Weaver</Author>
12     <Price>150.00</Price>
13   </Book>
14   <Book>
15     <Title>Speech Pattern</Title>
16     <Author>Speaker</Author>
17     <Price>15.00</Price>
18   </Book>
19   <Book>
20     <Title>Writing Style</Title>
21     <Author>Writer</Author>
22     <Price>1500.00</Price>
23   </Book>
24 </BookStore>
```

Figure 7.12: XML File

Figure 7.13 depicts the style sheet for **bookdetail.xsl** file.

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="BookDetail.xsl"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <html>
      <head>
        <title>example</title>
      </head>
      <body>
        <xsl:value-of select='translate("-----------------Bboks-------
        ---------","bok","ook")'/>
        <br/>
        <xsl:value-of select="concat('AWARD WINNING', 'BOOK DETAILS')"/>
        <xsl:apply-templates select="//Book"/>
      </body>
    </html>
  </xsl:template>
  <xsl:template match="Book">
    <xsl:if test="contains(Title, 'Pattern')">
      <DIV>
        <B>
          <xsl:value-of select="Title"/>
        </B>
        by
        <I>
          <xsl:value-of select="Author"/>
        </I>
        costs
        <xsl:value-of select="Price"/>
        .
      </DIV>
    </xsl:if>
  </xsl:template>
</xsl:stylesheet>
```

Figure 7.13: BookDetail.xsl File

The output is shown in figure 7.14.

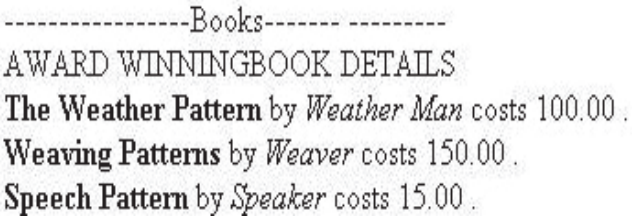


Figure 7.14: Output of Using String Functions

7.2.7 Result Tree Fragments

A Result tree fragment is a portion of an XML document that is not a complete node or set of nodes.

The only allowed operation in a result tree fragment is on a string. The operation on the string may involve first converting the string to a number or a boolean. Result tree fragment is an additional data type other than four basic XPath data types (such as, string, number, boolean, node-set).

A result tree fragment represents a fragment of the result tree. In particular, it is not permitted to use the /, //, and [] XPath operators on Result tree fragments.

Knowledge Check 2

1. Which of these statements about XPath expressions are true and which of these are false?

(A)	XSLT allows any kind of data to be transformed into a boolean value.
(B)	A node-set is an unordered group of nodes from the input document.
(C)	A string is a sequence of zero or more Unicode characters.
(D)	The numbers like 43 or –7000 that look like integers are stored as float.
(E)	A string is a sequence of one or more Unicode characters.

2. Which of these statements about XPath functions are true and which of these are false?

(A)	The `local-name()` function returns the name of the current node or the first node in the specified node set – without the namespace prefix.
(B)	The `floor(num)` function returns the largest integer that is not greater than the number argument.
(C)	The only allowed operation in a result tree fragment is on a number.
(D)	In a `substring()` function, the index of the first character is 0.
(E)	The `translate()` function returns the first argument string with occurrences of characters in the second argument string replaced by the character at the corresponding position in the third argument string.

Concepts

7.3 Working with Different Styles

In this last lesson, **Working with different styles**, you will learn to:

- Explain how to switch between styles.
- Describe how to transform XML documents into HTML using XSLT.

7.3.1 Transformation of XML Documents

Transformation is one of the most important and useful techniques for working with XML. XML can be transformed by changing its structure, its markup, and perhaps its content into another form. The most common reason to transform XML is to extend the reach of a document into new areas by converting it into a presentational format.

Some uses of transformation are:

- Formatting a document to create a high-quality presentational format.
- Changing one XML vocabulary to another.
- Extracting specific pieces of information and formatting them in another way.
- Changing an instance of XML into text.
- Reformatting or generating content.

Figure 7.15 shows the trasformation of XML document.

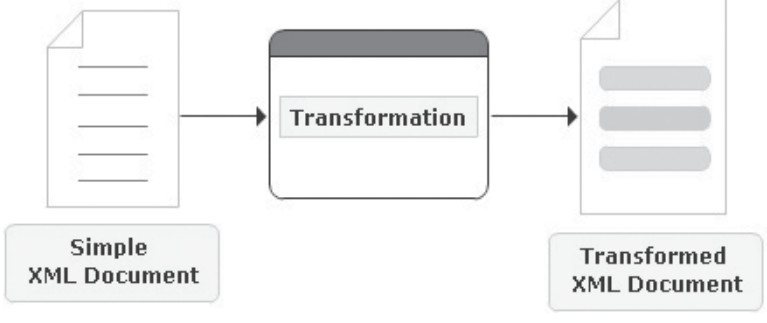


Figure 7.15: Transformation of XML Document

Note: Transformation can be used to alter the content, such as extracting a section, or adding a table of numbers together. It can even be used to filter an XML document to change it in very small ways, such as inserting an attribute into a particular kind of element.

Concepts

7.3.2 Transformation using XSLT Processor

An XSLT processor takes two things as input: an XSLT style sheet to govern the transformation process and an input document called the source tree. The output is called the result tree.

The XSLT engine begins by reading in the XSLT style sheet and caching it as a look-up table. XPath locates the parts of XML document such as Element nodes, Attribute nodes, and Text nodes. Thus, for each node the XSLT processes, it will look in the table for the best matching rule to apply. Starting from the root node, the XSLT engine finds rules, executes them, and continues until there are no more nodes in its context node set to work with. At that point, processing is complete and the XSLT engine outputs the result document.

Figure 7.16 depicts XSLT transformation.

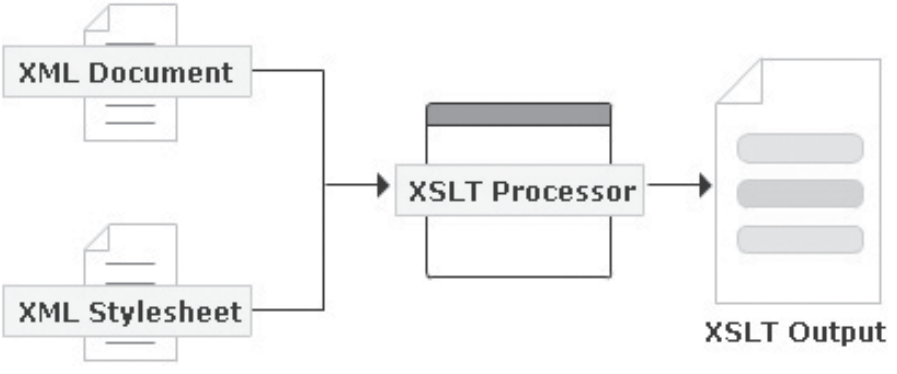


Figure 7.16: XSLT Transformation

Note: The XSLT style sheet controls the transformation process. While it is usually called a style sheet, it is not necessarily used to apply style. Since, XSLT is used for many other purposes, it may be better to call it an XSLT script or transformation document.

7.3.3 Transforming XML using XSLT

The transformation of XML document can be done in various steps:

- **Step 1**

  Start by creating a normal XML document:

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

➢ **Step 2**

Then add the lines shown to create an XSL style sheet:

Code Snippet:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0" >
   ...
</xsl:stylesheet>
```

➢ **Step 3**

Now, set it up to produce HTML-compatible output:

Code Snippet:

```
<xsl:stylesheet>
    <xsl:output method="html"/>
    ...
</xsl:stylesheet>
```

To output anything besides well-formed XML, an `<xsl:output>` tag should be used like the one shown, specifying either "`text`" or "`html`". (The default value is "`xml`".). Figure 7.17 depicts XML transformation.

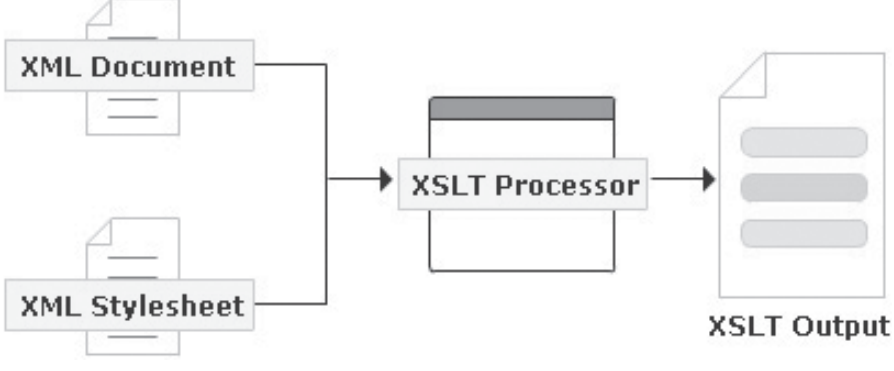


Figure 7.17: XML Transformation

Note: When an XML output is specified, the indent attribute can be added to produce nicely indented XML output. The specification looks like this: `<xsl:output method="xml" indent="yes"/>`.

Concepts

7.3.4 Transforming XML using XSLT Example

An example code of transforming XML documents into HTML using XSLT processor has been provided to explain the process.

Figure 7.18 shows the code in XML file.

```
1  <?xml version="1.0"?>
2  <?xml-stylesheet type="text/xsl" href="ProductInfo.xsl"?>
3
4  <Company>
5
6    <Product>
7      <Product_ID>S002</Product_ID>
8      <Name>Steel</Name>
9      <Price>1000/tonne</Price>
10     <Boiling_Point Units="Kelvin">20.28</Boiling_Point>
11     <Melting_Point Units="Kelvin">13.81</Melting_Point>
12   </Product>
13
14   <Product>
15     <Product_ID>S004</Product_ID>
16     <Name>Iron</Name>
17     <Price>5000/tonne</Price>
18     <Boiling_Point Units="Kelvin">34.216</Boiling_Point>
19     <Melting_Point Units="Kelvin">23.81</Melting_Point>
20   </Product>
21
22 </Company>
```

Figure 7.18: XML File

where,

`Company`

The root `Company` element contains `Product` child elements.

`Units`

A `Units` attribute specifies the units for the respective products melting point and boiling point.

Figure 7.19 depicts the style sheet for **productioninfo.xsl** file.

```
1  <?xml version="1.0"?>
2
3  <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
4
5    <xsl:template match="Company">
6      <html>
7        <xsl:apply-templates/>
8      </html>
9    </xsl:template>
10
11   <xsl:template match="Product">
12     <P>
13       <xsl:apply-templates/>
14     </P>
15   </xsl:template>
16
17 </xsl:stylesheet>
```

Figure 7.19: ProductInfo.xsl File

The output is shown in figure 7.20.

The output displays the transformed HTML from the XML document in the browser.

S002Steel1000/tonne20.2813.81

S004Iron5000/tonne34.21623.81

Figure 7.20: Output Showing Transformed HTML

Knowledge Check 3

1. Which of these statements about switching between styles are true and which of these are false?

(A)	An XSLT processor takes three things as input such as XSLT style sheet, XML document and Document Type Declaration.
(B)	The XSLT engine begins by reading in the XSLT style sheet and caching it as a look-up table.
(C)	For each node it processes, it will look in the table for the best matching rule to apply.
(D)	Starting from the root node, the XSLT engine finds rules, executes them, and continues until there are no more nodes in its context node set to work with.
(E)	XSLT can also be called as XSLT document or transformation script.

2. Can you specify the correct code snippet for transforming the XML document into HTML using XSLT?

(A)

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0"
>
......
<xsl:output method="html"/>
......
</xsl:stylesheet>
```

(B)

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:output method="html"/>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0"
>
......
</xsl:stylesheet>
```

(C)	<?xml version="1.0" encoding="UTF-8"?> <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0" > <xsl:template match="/"> ...... </xsl:template> </xsl:stylesheet>
(D)	<?xml version="1.0" encoding="UTF-8"?> <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0" > <xsl:template match="/"> <xsl:output method="html"/> ...... <xsl:template match="/"> </xsl:stylesheet>

Concepts

Module Summary

In this module, **More on XSLT**, you learnt about:

- **XPath**

  XPath is a notation for retrieving information from a document. XPath provides a common syntax for features shared by Extensible Style sheet Language Transformations (XSLT) and XQuery. XPath have seven types of node as Root, Element, Attribute, Text, Comment, Processing instruction and Namespace. XPath is used in the creation of the patterns.

- **XPath Expressions and Functions**

  The four types of expressions in XPath are Node-sets, Booleans, Numbers and Strings. The different functions defined for XPath are Accessor, AnyURI, Node, Error and Trace, Sequence, Context, Boolean, Duration/Date/Time, String, QName and Numeric. A Result tree fragment is a portion of an XML document that is not a complete node or set of nodes.

- **Working with different styles**

  Transformation is one of the most important and useful techniques for working with XML. To transform XML is to change its structure, its markup, and perhaps its content into another form. Transformation can be carried out using an XSLT processor also.

Answers

A

Answers to Knowledge Checks

Module 1

Knowledge Check 1

1. (A) - True, (B) – False, (C) – True, (D) – True, (E) - True

2. (A) - False, (B) - True, (C) - False, (D) - True, (E) – True

Knowledge Check 2

1. (A) - True, (B) – True, (C) – True, (D) – False, (E) - True

2. (A) – True, (B) – False, (C) – True, (D) – False, (E) - True

3. (A) – True, (B) – False, (C) – True, (D) – False, (E) – False

Knowledge Check 3

1. (A)

2. (C)

Knowledge Check 4

1. (A) – False, (B) – False, (C) – True, (D) – True, (E) - False

2. (A) – False, (B) – False, (C) – True, (D) – True, (E) - False

Module 2

Knowledge Check 1

1. (A) - False, (B) - True, (C) - True, (D) - False

Knowledge Check 2

1. (A)

2. (A) - False, (B) - True, (C) - False, (D) - False, (E) - True

3. (A) - True, (B) - False, (C) - False, (D) - True, (E) - False

Answers to Knowledge Checks

Module 3

Knowledge Check 1

1. (A) - (False), (B) - (True), (C) - (True), (D) - (True), (E) - (False)

Knowledge Check 2

1. (A) - (False), (B) - (True), (C) - (True), (D) - (True), (E) - (False)

2. (A) - Declare all the possible elements, (B) - Specify the permissible element children, if any, (C) - Set the order in which elements must appear, (D) - Declare all the possible element attributes, (E) - Set the attribute data types and values, (F) - Declare all the possible entities.

Knowledge Check 3

1. (C)

Knowledge Check 4

1. (A) - (3), (B) - (4), (C) - (5), (D) - (1), (E) - (2)

2. (A) - (2), (B) - (4), (C) - (5), (D) - (1), (E) - (3)

3. (D)

Module 4

Knowledge Check 1

1. (A) - True, (B) - False, (C) - False, (D) - True, (E) - True

2. (A) – (4), (B) - (1), (C) - (5), (D) - (3), (E) - (2)

Knowledge Check 2

1. (A) – (2), (B) - (5), (C) - (4), (D) - (1), (E) - (3)

2. (A) - `<?xml version="1.0"?> <note xmlns="http://www.abc.com"`, (B) - `xmlns: xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.abc.com mail.xsd">`, (C) - `<to>John</to> <from>Jordan</from>`, (D) - `<heading>Scheduler</heading> <body>3rd March Monday, 7:30 PM: board meeting!</body>`, (E) - `</mail>`

Knowledge Check 3

1. (A) - False, (B) - True, (C) - True, (D) - False, (E) - False

2. (A) - False, (B) - True, (C) - False, (D) - True

Answers to Knowledge Checks

3. (A) - False, (B) - True, (C) - True, (D) - False

Knowledge Check 4

1. (A) - False, (B) - True, (C) - True, (D) - False, (E) - True

2. (A) – (4), (B) - (1), (C) - (5), (D) - (2), (E) - (3)

Module 5

Knowledge Check 1

1. (A) - True, (B) - False, (C) - True, (D) - False, (E) - True

Knowledge Check 2

1. (A) - False, (B) - True, (C) - True, (D) - True, (E) - False

Knowledge Check 3

1. (A) - (3), (B) - (4) , (C) - (5) , (D) - (1) , (E) - (2)

2.
```
display: block;

background-color: blue;

color: white;

border: medium solid magenta; text-indent: 20
```

3. (A) - (5), (B) - (3), (C) - (2), (D) - (4) , (E) - (1)

Knowledge Check 4

1. (A) - True, (B) - True, (C) - False, (D) - True, (E) - False

Module 6

Knowledge Check 1

1. (A) - XML processor reads an XML document, (B) - XML processor creates a hierarchical tree containing nodes for each piece of information, (C) - Apply the rules of an XSL style sheet to document tree., (D) - XSL processor starts with root node in tree and performs pattern matching, (E) - Portion of a tree matching the given pattern is processed by appropriate style sheet template.

2. (A) - (3), (B) - (5), (C) - (4), (D) - (1), (E) - (2)

3. (A) - (False), (B) - True, (C) - (True), (D) - (False), (E) - (True)

Answers to Knowledge Checks

Knowledge Check 2

1. (C)
2. (A) - (5), (B) - (3), (C) - (1), (D) - (2), (E) - (4)
3. (D)

Module 7

Knowledge Check 1

1. (A) - False, (B) – True, (C) – False, (D) – True, (E) - True
2. (A) – (3), (B) - (5), (C) - (4), (D) - (1), and (E) - (2)
3. (A) – (4), (B) – (1), (C) – (2), (D) – (5), and (E) – (3)

Knowledge Check 2

1. (A) - True, (B) – True, (C) – True, (D) – False, (E) - False
2. (A) - True, (B) – True, (C) – False, (D) – False, (E) - True

Knowledge Check 3

1. (A) - False, (B) – True, (C) – True, (D) – True, (E) - False
2. (A)