BCA-17

Vardhaman Mahaveer Open University, Kota

Web Technology

Course Development Committee

Chairman
Prof. (Dr.) Naresh Dadhich
Former Vice-Chancellor
Vardhaman Mahaveer Open University, Kota

Co-ordinator/Convener and Members

Convener

Dr. Anuradha Sharma
Assistant Professor,
Department of Botany, Vardhaman Mahaveer Open University, Kota

Members :

1. **Dr. Neeraj Bhargava**
   Department of Computer Science
   Maharshi Dyanand Saraswati University, Ajmer
2. **Prof. Reena Dadhich**
   Department of Computer Science
   University of Kota, Kota
5. **Dr. Nishtha Keswani**
   Department of Computer Science
   Central University of Rajasthan, Ajmer
3. **Dr. Madhavi Sinha**
   Department of Computer Science
   Birla Institute of Technology & Science, Jaipur
4. **Dr. Rajeev Srivastava**
   Department of Computer Science
   LBS College, Jaipur

Editing and Course Writing

Editor

Dr. Neeraj Bhargava
Department of Computer Science
Maharshi Dyanand Saraswati University, Ajmer

Unit Writers	Unit No.
1. **Dr. Ritu Bhargava** Department of Computer Application Govt. Women Engineering Collge, Ajmer	(1,2,3)
2. **Dr. Neeraj Bhargava** Department of Computer Science Maharshi Dyanand Saraswati University, Ajmer	(4,5,6)
3. **Dr. Ajay Singh Gaur** Department of Computer Science Thakur Jai Singh College, Kota	(7,8,9)
4. **Sh. Prafull Chandra Narooka** Department of Computer Science Agarwal College, Merta City	(10,11)
5. **Sh. Pankaj Sharma** Department of Computer Engineering Govt. Women Engineering College, Ajmer	(12,13)
6. **Sh. Prakash Singh Tanwar** Department of Computer Science Aryabahatt College, Ajmer	(14,15)

Academic and Administrative Management

Prof. (Dr.) Vinay Kumar Pathak
Vice-Chancellor
Vardhaman Mahaveer Open University, Kota

Prof. (Dr.) B.K. Sharma
Director (Academic)
Vardhaman Mahaveer Open University, Kota

Prof. (Dr.) P.K. Sharma
Director (Regional Services)
Vardhaman Mahaveer Open University, Kota

Course Material Production

Mr. Yogendra Goyal
Assistant Production Officer
Vardhaman Mahaveer Open University, Kota

Vardhaman Mahaveer Open University, Kota

Web Technology

Unit No.	Units	Page No.
1.	Introduction and Overview of XML	1-8
2.	XML Fundamentals	9-15
3.	XML Syntax	16-21
4.	XML Namespaces	22-25
5.	XML Document Type Definition (DTD)	26-34
6.	XML Schema Definition (XSD)	35-45
7.	XQuery and XPath	46-65
8.	Publishing XML	66-79
9.	XML Formatting with XSL-FO	80-91
10.	XML Transformation with XSLT	92-102
11.	XLink and XPointer	103-105
12.	XForms	106-112
13.	Applying XML	113-121
14.	Parsing XML in Java	122-132
15.	Security in XML	133-140

Preface

The course XML has been designed for students who want to learn XML. The text has been designed for versatile and complete insight into XML. This book provides some advance topics.

The text contains 15 Chapters intended primarily for graduate courses. The wide range of topics covered in the book makes it an excellent handbook on XML.

It covered topics such as introduction of XML, evolution, creation XML pages, schema and data types, XML query, transformation of XML in various fields.

It also explains the XLinking and atlast XML security is explained.

The text has been supplemented with appropriate figures for better understanding. Extensive references and pointers to the current literature have also been provided.

Each Chapter ends with self-assessment exercises that can be used for practice.

Every chapter is ended with summary and end questions to facilitate the learner for self study and some challenging tasks.

Unit - 1 : Introduction and Overview of XML

Structure of the Unit:

1.0 Objective
1.1 Introduction
1.2 Why XML
1.3 Adapting Markup Languages from SGML, XML and XHTML
1.4 Advantages and Disadvantages
1.5 Introduction of eXtensible Markup Language
1.6 Representing Mixed Data
1.7 Summary
1.8 Self Assessment Question

1.0 Objective

In this module we have fundamental introduction about the XML and their merits and demerits. The module also define the information about the evolution of XML.

1.1 Introduction

The computing press has found a new savior for the ills that afflict computing and the web: XML. XML is new, it's exciting, and it's got to be good, because the specification for it looks indecipherable. XML's hype level has already drawn fire from some quarters, from those accusing it of 'balkanizing the web' or of increasing the load on an already strained Internet. Most important, many developers are wondering why exactly they need to learn yet another language.

1.2 Why XML

XML's set of tools allows developers to create web pages - and much more. XML allows developers to set standards defining the information that should appear in a document, and in what sequence. XML, in combination with other standards, makes it possible to define the content of a document separately from its formatting, making it easy to reuse that content in other applications or for other presentation environments. Most important, XML provides a basic syntax that can be used to share information between different kinds of computers, different applications, and different organizations without needing to pass through many layers of conversion.

Web developers are the initial target audience, but database developers, document managers, desktop publishers, programmers, scientists, and other academics are all getting involved. XML provides a simple format that is flexible enough to accommodate wildly diverse needs. Even developers performing tasks on different types of applications with different interfaces and different data structures can share XML formats and tools for parsing those formats into data structures that applications can use. XML offers its users many advantages, including:

- Simplicity
- Extensibility
- Interoperability
- Openness
- A core of experienced professionals

What is XML?

Extensible Markup Language (XML) provides a foundation for creating documents and document systems. XML operates on two main levels: first, it provides syntax for document markup; and second, it provides syntax for declaring the structures of documents. XML is clearly targeted at the Web, though it certainly has applications beyond it. Users who have worked with HTML before should be able to learn the basics of XML without too much difficulty. XML's simplicity is its key selling point, perhaps even its strongest feature.

XML is derived from (and is technically a subset of) the Standard Generalized Markup Language (SGML). SGML has found its main customer base in organizations handling enormous quantities of documents - the U.S. Government Printing Office, IBM, the U.S. Department of Defense and Internal Revenue Service, and many publishers. SGML's development provides the foundations for XML, but XML has a smaller and simpler syntax, targeted at web developers and others who need a simple solution to document creation, management, and display.

1.3 Adapting Markup Languages from SGML, XML and XHTML

A **markup language** is a modern system for annotating a document in a way that is syntactically distinguishable from the text. The idea and terminology evolved from the "marking up" of manuscripts, i.e., the revision instructions by editors, traditionally written with a blue pencil on authors' manuscripts. Examples are typesetting instructions such as those found in troff and LaTeX, or structural markers such as XML tags. Markup is typically omitted from the version of the text that is displayed for end-user consumption. Some markup languages, such as HTML, have presentation semantics, meaning that their specification prescribes how the structured data are to be presented, but other markup languages, like XML, have no predefined semantics.

A well-known example of a markup language in widespread use today is HyperText Markup Language (HTML), one of the document formats of the World Wide Web. HTML, which is an instance of SGML (though, strictly, it does not comply with all the rules of SGML), follows many of the markup conventions used in the publishing industry in the communication of printed work between authors, editors, and printers

Scribe, GML and SGML

The first language to make a clean distinction between structure and presentation was Scribe, developed by Brian Reid and described in his doctoral thesis in 1980.[7] Scribe was revolutionary in a number of ways, not least that it introduced the idea of styles separated from the marked up document, and of a grammar controlling the usage of descriptive elements. Scribe influenced the development of Generalized Markup Language (later SGML) and is a direct ancestor to HTML and LaTeX.

In the early 1980s, the idea that markup should be focused on the structural aspects of a document and leave the visual presentation of that structure to the interpreter led to the creation of SGML. The language was developed by a committee chaired by Goldfarb. It incorporated ideas from many different sources, including Tunnicliffe's project, GenCode. Sharon Adler, Anders Berglund, and James A. Marke were also key members of the SGML committee.

SGML specified a syntax for including the markup in documents, as well as one for separately describing what tags were allowed, and where (the Document Type Definition (DTD) or schema). This allowed

authors to create and use any markup they wished, selecting tags that made the most sense to them and were named in their own natural languages. Thus, SGML is properly ameta-language, and many particular markup languages are derived from it. From the late '80s on, most substantial new markup languages have been based on SGML system, including for example TEI and DocBook. SGML was promulgated as an International Standard by International Organization for Standardization, ISO 8879, in 1986.

SGML found wide acceptance and use in fields with very large-scale documentation requirements. However, it was generally found to be cumbersome and difficult to learn, a side effect of attempting to do too much and be too flexible. For example, SGML made end tags (or start-tags, or even both) optional in certain contexts, because it was thought that markup would be done manually by overworked support staff who would appreciate saving keystrokes.

HTML

By 1991, it appeared to many that SGML would be limited to commercial and data-based applications while WYSIWYG tools (which stored documents in proprietary binary formats) would suffice for other document processing applications. The situation changed when Sir Tim Berners-Lee, learning of SGML from co-worker Anders Berglund and others at CERN, used SGML syntax to create HTML. HTML resembles other SGML-based tag languages, although it began as simpler than most and a formal DTD was not developed until later. Steven DeRose argues that HTML's use of descriptive markup (and SGML in particular) was a major factor in the success of the Web, because of the flexibility and extensibility that it enabled (other factors include the notion of URLs and the free distribution of browsers). HTML is quite likely the most used markup language in the world today.

Some would restrict the term "markup language" to systems that directly support non-hierarchical structures (see Hierarchical model). By this definition HTML, XML, and even SGML (apart from its rarely used CONCUR option) would be disqualified and called "container languages" instead. However, the term "container language" is not in widespread use, and such hierarchical languages are almost universally considered markup languages. There is active research on non-hierarchical markup models, some expressed within XML and related languages (for example, using the Text Encoding Initiative Guidelines and derivatives such as the Open Scripture Information Standard and CLIX), and some not (for example, MECS and the Layed Markup and Annotation Language or LMNL). Much of this research is published in the proceedings of the Extreme Markup and Balisage conferences, generally held in Montreal.

XML

XML (Extensible Markup Language) is a meta markup language that is now widely used. XML was developed by the World Wide Web Consortium, in a committee created and chaired by Jon Bosak. The main purpose of XML was to simplify SGML by focusing on a particular problem — documents on the Internet. XML remains a meta-language like SGML, allowing users to create any tags needed (hence "extensible") and then describing those tags and their permitted uses.

XML adoption was helped because every XML document can be written in such a way that it is also an SGML document, and existing SGML users and software could switch to XML fairly easily. However, XML eliminated many of the more complex and human-oriented features of SGML to simplify implementation environments such as documents and publications. However, it appeared to strike a happy medium between simplicity and flexibility, and was rapidly adopted for many other uses. XML is now widely used for communicating data between applications. Like HTML, it can be described as a 'container' language.

XHTML

Since January 2000 all W3C Recommendations for HTML have been based on XML rather than SGML, using the abbreviation XHTML (Extensible HyperText Markup Language). The language specification requires that XHTML Web documents must be well-formed XML documents – this allows for more rigorous and robust documents while using tags familiar from HTML.

One of the most noticeable differences between HTML and XHTML is the rule that all tags must be closed: empty HTML tags such as
 must either be closed with a regular end-tag, or replaced by a special form:
 (the space before the '/' on the end tag is optional, but frequently used because it enables some pre-XML Web browsers, and SGML parsers, to accept the tag). Another is that all attribute values in tags must be quoted. Finally, all tag and attribute names must be lowercase in order to be valid; HTML, on the other hand, was case-insensitive.

1.4 Advantages and Disadvantages

Advantages of XML

- It is text-based.
- It supports Unicode, allowing almost any information in any written human language to be com municated.
- It can represent the most general computer science data structures: records, lists and trees.
- Its self-documenting format describes structure and field names as well as specific values.
- The strict syntax and parsing requirements make the necessary parsing algorithms extremely simple, efficient, and consistent.

XML is heavily used as a format for document storage and processing, both online and offline.

- It is based on international standards.
- It can be updated incrementally.
- It allows validation using schema languages such as XSD and Schematron, which makes effective unit-testing, firewalls, acceptance testing, contractual specification and software construction easier.
- The hierarchical structure is suitable for most (but not all) types of documents.
- It manifests as plain text files, which are less restrictive than other proprietary document formats.
- It is platform-independent, thus relatively immune to changes in technology.
- Forward and backward compatibility are relatively easy to maintain despite changes in DTD or Schema.
- Its predecessor, SGML, has been in use since 1986, so there is extensive experience and software available.
- An element fragment of a well-formed XML document is also a well-formed XML document.

Disadvantages of XML

- XML syntax is redundant or large relative to binary representations of similar data.
- The redundancy may affect application efficiency through higher storage, transmission and processing costs.
- XML syntax is verbose relative to other alternative 'text-based' data transmission formats.
- No intrinsic data type support: XML provides no specific notion of "integer", "string", "boolean", "date", and so on.
- The hierarchical model for representation is limited in comparison to the relational model or an object oriented graph.
- Expressing overlapping (non-hierarchical) node relationships requires extra effort.
- XML namespaces are problematic to use and namespace support can be difficult to correctly implement in an XML parser.
- XML is commonly depicted as "self-documenting" but this depiction ignores critical ambiguities.

1.5 Introduction of eXtensible Markup Language

The Extensible Markup Language (XML), as specified in the World Wide Web Consortium's (W3C) Recommendation approved on February 10, 1998, is a subset of the Standard Generalized Markup Language (SGML) defined in ISO standard 8879:1986 that is designed to make it easy to interchange structured documents over the Internet. XML files always clearly mark where the start and end of each of the logical parts (called elements) of an interchanged document occurs. XML restricts the use of SGML constructs to ensure that fallback options are available when access to certain components of the document is not currently possible over the Internet. It also defines how Internet Uniform Resource Locators can be used to identify component parts of XML data streams.

By defining the role of each element of text in a formal model, known as a Document Type Definition (DTD), users of XML can check that each component of a document occurs in a valid place within the interchanged data stream. An XML DTD allows computers to check, for example, that users do not accidentally enter a third-level heading without first having entered a second-level heading, something that cannot be checked using the Hypertext Markup Language (HTML) previously used to code documents that form part of the World Wide Web (WWW) of documents accessible through the Internet.

However, unlike SGML, XML does not require the presence of a DTD. If no DTD is available, either because all or part of it is not accessible over the Internet or because the user failed to create it, an XML system can assign a default definition for undeclared components of the markup. XML allows users to

- bring multiple files together to form compound documents
- identify where illustrations are to be incorporated into text files, and the format used to encode each illustration
- provide processing control information to supporting programs, such as document validators and browsers
- add editorial comments to a file.

It is important to note, however, that XML is not

- a predefined set of tags, of the type defined for HTML, that can be used to markup documents
- a standardized template for producing particular types of documents.

1.6 Representing Mixed Data

Mixed Data Representation

If you've been working with XML as a format for representing data, its usefulness for that purpose is probably obvious to you. Likewise, if you've been working with it as a format for representing narrative text (XHTML is an example), the structure XML provides has probably been handy. But if you've ever taken a programming model that's designed for one purpose and tried to bend it to the other, there's a good chance you've cursed a bit over it. That line between data and text can feel like a wall. In this article, I'm going to introduce a way to keep the strongly-typed access that's best suited for data while also handling the mixed content that comes with narrative text.

An API that uses strong types to represent the XML is great for getting tidy pieces of data out and putting them in. If you're using XMLBeans, a technology currently incubating at Apache, you've probably been getting at your XML using the types you can generate from schema. Imagine having the following XML and a schema that describes it.

```
<item id="123456">
  <name>
    flangie
  </name>
  <description>
    Brand new technology to accompany our callioscim for those really tough jobs.
  </description>
</item>
```

Compiling the schema would generate types that provide methods such as getItem(), setName(String), and so on. In your code, you'd ease into a getting and setting rhythm that would make handling the XML pretty straightforward – until you needed to handle something like this:

```
<item id="123456">
  <name>
    flangie
  </name>
  <description>
    Brand new technology to accompany our
    <item-link id="654321">callioscim</item-link> for those really tough jobs.
  </description>
```

</item>

In this case, the <description> element contains what's known as "mixed content". Mixed content is element content that contains a mixture of child elements and character data. Sometimes mixed content takes the form of frustratingly intermingled text and embedded elements, as in the preceding example. With XMLBeans' generated types, a getItemLinkArray()method would give you all of the embedded <item-link> elements, but you'd get them as, well, an array. The array wouldn't be able to tell you what text came before and after each<item-link> – in other words, it would lack the context that gives the each <item-link> element's value part of its meaning. Setting the <description> element's mixed content, needing text-element-text, would be as frustrating.

Minding your place with a cursor

As it turns out, if you're willing to venture over that data/text line, XMLBeans provides a way to get at mixed content. With its XmlCursor interface you can step outside of the getting-, setting-, data-oriented world and move into text (and "tokens", as we'll see). Let's start with this very simple XML snippet:

<item id='123456'/>

The XmlCursor interface provides a way for you to "walk" the XML, moving past elements, attributes, and text, and even moving character by character through text. An XmlCursor instance sees your XML as a series of tokens, each of which belongs to a category, called a "token type". The following example shows a cursor traversing the XML above, printing the token type and corresponding XML along the way.

```
// Put a snippet of XML into an XMLBeans type.
XmlObject itemXml = XmlObject.Factory.parse("<item id='123456'/>");
// Insert a new cursor at the very top of the document containing the XML.
XmlCursor cursor = itemXml.newCursor();
/*
 * Before moving the cursor anywhere, print out the XML and token type
 * where it is now, at the outset.
 */
System.out.println(cursor.currentTokenType().toString());
System.out.println(cursor.toString());
/*
 * Start moving the cursor forward. The toNextToken().isNone() method
 * returns true if there aren't any more tokens after the cursor to move to. At
 * each token, print out the token type and corresponding XML.
 */
while (!cursor.toNextToken().isNone())
{
```

```
    System.out.println(cursor.currentTokenType().toString());
    System.out.println(cursor.toString());
}
// Signal that the cursor may be garbage collected.
cursor.dispose();
```

The code prints this:

```
    STARTDOC
    <item id="123456"/>
    START
    <item id="123456"/>
    ATTR
    <xml-fragment id="123456"/>
    END
    <xml-fragment/>
    ENDDOC
    <xml-fragment/>
```

The STARTDOC and ENDDOC tokens are bookends for the XML document as a whole, but they don't represent any piece of the XML's content. The START and END tokens, on the other hand, represent the start and end of the <item> element. The ATTR token represents, of course, the id attribute. As you can see, XMLBeans attempts to print valid XML for each token along the way, even when the token corresponds to something that's not valid on its own – such as the id attribute, the end of the <item> element, or the end of the document.

1.7 Summary

The unit provide information about the XML and their advantage, disadvantage in the current technology for web designing.

The unit provide how to use mix data with the xml.

1.8 Self Assessment Question

1. How can we use mixed data with our xml page?
2. Define the evolution of XML?
3. Differentiate HTML, XML and XHTML?
4. Give Advantages and disadvantages of XML?
5. Explain mixed data representation with example?

Unit - 2 : XML Fundamental

Structure of the Unit:

2.0 Objective
2.1 Introduction
2.2 Creating XML Document & Define XML Structure
2.3 Rules for Well Formed XML
2.4 Summary
2.5 Self Assessment Question

2.0 Objective

If we want to create a XML page and doesn't have information about their rules and structure then this unit have sufficient topic to learn them.

2.1 Introduction

We've discussed some of the reasons why XML makes sense for communicating data, so now let's get our hands dirty and learn how to create our own XML documents. Well-formed XML is XML that meets certain grammatical rules outlined in the XML 1.0 specification.

You will learn:

- How to create XML elements using start- and end-tags
- How to further describe elements with attributes
- How to declare your document as being XML
- How to send instructions to applications that are processing the XML document
- Which characters aren't allowed in XML, and how to put them in anyway

2.2 Creating XML Document & Define XML Structure

XML and HTML appear so similar, and because you're probably already familiar with HTML, we'll be making comparisons between the two languages in this chapter. However, if you don't have any knowledge of HTML, you shouldn't find it too hard to follow along.

If you have Internet Explorer 5, you may find it useful to save some of the examples in this chapter on your hard drive, and view the results in the browser. If you don't have IE5, some of the examples will have screenshots to show what the end results look like.

It's time to stop calling things just "items" and "text"; we need some names for the pieces that make up an XMLdocument. To get cracking, let's break down the simple <name> document we created in Chapter 1:

```
<name>
  <first>John</first>
  <middle>Fitzgerald Johansen</middle>
```

<last>Doe</last>

</name>

The words between the < and > characters are XML tags. The information in our document (our data) is contained within the various tags that constitute the markup of the document. This makes it easy to distinguish the information in the document from the markup.

As you can see, the tags are paired together, so that any opening tag also has a closing tag. In XML parlance, these are called start-tags and end-tags. The end-tags are the same as the start-tags, except that they have a "/" right after the opening < character.

In this regard, XML tags work the same as start-tags and end-tags do in HTML. For example, you would create an HTML paragraph like this:

<P>This is a paragraph.</P>

As you can see, there is a <P> start-tag, and a </P> end-tag, just like we use for XML.

All of the information from the start of a start-tag to the end of an end-tag, and including everything in between, is called an element. So:

- <first> is a start-tag
- </first> is an end-tag
- <first>John</first> is an element

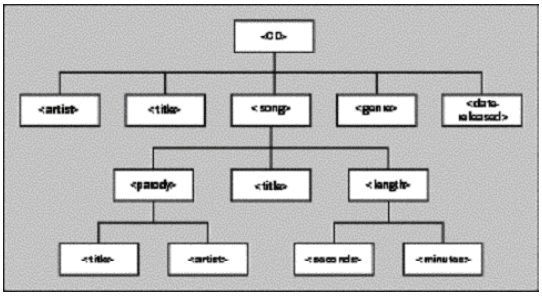
The text between the start-tag and end-tag of an element is called the element content. The content between our tags will often just be data (as opposed to other elements). In this case, the element content is referred to as Parsed Character DATA, which is almost always referred to using its acronym, PCDATA.

Whenever you come across a strange-looking term like PCDATA, it's usually a good bet the term is inherited from SGML. Because XML is a subset of SGML, there are a lot of these inherited terms.

The whole document, starting at <name> and ending at </name>, is also an element, which happens to include other elements. (And, in this case, the element is called the root element, which we'll be talking about later.)

To put this new-found knowledge into action, let's create an example that contains more information than just a name.We're going to build an XML document to describe one of the greatest CDs ever produced, Dare to be Stupid, by Weird Al Yankovic. But before we break out Notepad and start typing, we need to know what information we're capturing.

1. Since this is a CD, we'll need to capture information like the artist, title, and date released, as well as the genre of music. We'll also need information about each song on the CD, such as the title and length. And, since Weird Al is famous for his parodies, we'll include information about what song (if any) this one is a parody of. Here's the hierarchy we'll be creating:



Some of these elements, like <artist>, will appear only once; others, like <song>, will appear multiple times in the document. Also, some will have PCDATA only, while some will include their information as child elements instead. For example, the <artist> element will contain PCDATA for the title, whereas the <song> element won't contain any PCDATA of its own, but will contain child elements that further break down the information.

2 this in mind, we're now ready to start entering XML. If you have Internet Explorer 5 installed on your machine, type the following into Notepad, and save it to your hard drive as cd.xml:

```
<CD>
  <artist>"Weird Al" Yankovic</artist>
  <title>Dare to be Stupid</title>
  <genre>parody</genre>
  <date-released>1990</date-released>
  <song>
    <title>Like A Surgeon</title>
    <length>
      <minutes>3</minutes>
      <seconds>33</seconds>
    </length>
    <parody>
      <title>Like A Virgin</title>
      <artist>Madonna</artist>
    </parody>
  </song>
  <song>
    <title>Dare to be Stupid</title>
    <length>
```

```
    <minutes>3</minutes>
    <seconds>25</seconds>
  </length>
  <parody></parody>
 </song>
</CD>
```

For the sake of brevity, we'll only enter two of the songs on the CD, but the idea is there nonethel ess.

3. Now, open the file in IE5. (Navigate to the file in Explorer and double click on it, or open up the browser and type the path in the URL bar.) If you have typed in the tags exactly as shown, the cd.xmlfile will look something like this:

Here we've created a hierarchy of information about a CD, so we've named the root element accordingly.

The <CD> element has children for the artist, title, genre, and date, as well as one child for each song on the disc. The <song> element has children for the title, length, and, since this is Weird Al we're talking about, what song (if any) this is a parody of. Again, for the sake of this example, the <length> element was broken down still further, to have children for minutes and seconds, and the <parody> element broken down to have the title and artist of the parodied song.

You may have noticed that the IE5 browser changed <parody></parody> into <parody/>. We'll talk about this shorthand syntax a little bit later, but don't worry: it's perfectly legal.

If we were to write a CD Player application, we could make use of this information to create a play-list for our CD. It could read the information under our <song> element to get the name and length of each song to display to the user, display the genre of the CD in the title bar, etc. Basically, it could make use of any information contained in our XMLdocument.

2.3 Rules for Well Formed XML

further along than our text file examples from the previous chapter. There must be some rules for elements, which are fundamental to the understanding of XML.

XML documents must adhere to these rules to be well-formed.

We'll list them, briefly, before getting down to details:

- Every start-tag must have a matching end-tag
- Tags can't overlap
- XML documents can have only one root element
- Element names must obey XML naming conventions
- XML is case-sensitive
- XML will keep white space in your text

Every Start-tag Must Have an End-tag

One of the problems with parsing SGML documents is that not every element requires a start-tag and an end-tag. Take the following HTML for example:

<HTML>

<BODY>

<P>Here is some text in an HTML paragraph.

Here is some more text in the same paragraph.

<P>And here is some text in another HTML paragraph.</p>

</BODY>

</HTML>

Notice that the first <P> tag has no closing </P> tag. This is allowed – and sometimes even encouraged – in HTML, because most web browsers can detect automatically where the end of the paragraph should be. In this case, when the browser comes across the second <P> tag, it knows to end the first paragraph. Then there's the
 tag (line break), which by definition has no closing tag.

Also, notice that the second <P> start-tag is matched by a </p> end-tag, in lower case. HTML browsers have to be smart enough to realize that both of these tags delimit the same element, but as we'll see soon, this would cause a problem for an XML parser.

The problem is that this makes HTML parsers much harder to write. Code has to be included to take into account all of these factors, which often makes the parsers much larger, and much harder to debug. What's more, the way that files are parsed is not standardized – different browsers do it differently, leading to incompatibilities.

For now, just remember that in XML the end-tag is required, and has to exactly match the start-tag.

Tags Can Not Overlap

Because XML is strictly hierarchical, you have to be careful to close your child elements before you close your parents. (This is called properly nesting your tags.) Let's look at another HTML example to demonstrate this:

<P>Some <STRONG>formatted <EM>text</STRONG>,

but</EM> no grammar no good!</P>

This would produce the following output on a web browser:

Some formatted text, but no grammar no good!

As you can see, the <STRONG> tags cover the text formatted text, while the <EM> tags cover the text text,but.

But is a child of , or is a child of ? Or are they both siblings, and children of <p>? According to our stricter XML rules, the answer is none of the above. The HTML code, as written, can't be arranged as a proper hierarchy, and could therefore not be well-formed XML.

If ever you're in doubt as to whether your XML tags are overlapping, try to rearrange them visually to be hierarchical. If the tree makes sense, then you're okay. Otherwise, you'll have to rework your markup.

For example, we could get the same effect as above by doing the following:

<P>Some <STRONG>formatted

<EM>text</EM></STRONG><EM>, but</EM> no grammar

no good!</P>

Which can be properly formatted in a tree, like this:

```
<P>
  Some
  <STRONG>
    formatted
    <EM>
      text
    </EM>
  </STRONG>
```

<EM>

, but

</EM>

no grammar no good!

</P>

An XML Document Can Have Only One Root Element

In our <name> document, the <name> element is called the root element. This is the top-level element in the document, and all the other elements are its children or descendents. An XML document must have one and only one root element: in fact, it must have a root element even if it has no content.

For example, the following XML is not well-formed, because it has a number of root elements:

<name>John</name>

<name>Jane</name>

To make this well-formed, we'd need to add a top-level element, like this:

<names>

<name>John</name>

<name>Jane</name>

</names>

So while it may seem a bit of an inconvenience, it turns out that it's incredibly easy to follow this rule. If you have a document structure with multiple root-like elements, simply create a higher-level element to contain them.

2.4 Summary

The unit define how to use XML and also provide valid rules to create formatted pages. In this unit we have the structure to define how many elements in the XML.

2.5 Self Assessment Question

1. Define the document structure of XML?
2. Define the rules to create a well formed XML?
3. How many elements are use in a XML documents?
4. What is a wellformed XML document?
5. Explain steps of creating an XML document.

Unit - 3 : XML SYNTAX

Structure of the Unit:

3.0 Objective
3.1 Introduction
3.2 Tag, attribute and Name Rules
3.3 Empty and Non Empty Elements
3.4 Processing Instruction for XML
3.5 Summary
3.6 Self Assessment Question

3.0 Objective

The objective of this unit is to provide necessary syntaxes to create a page. The instruction which we use known as <tag>.

3.1 Introduction

When we create a XML page we have various tag like HTML, DHTML. Although XML 1.0 is not a complicated format, there are many more details (and much terminology) that this tutorial does not cover. If you are planning to implement software that reads or writes XML directly (rather than through a specialized library), then you will need to refer to the XML 1.0 Recommendation, which is available online and free of charge from the World Wide Web Consortium: the Recommendation is the single authoritative source for all XML work.

3.2 Tag, attribute and Name Rules

XML tags begin with the less-than character ("<") and end with the greater-than character (">"). You use tags to mark the start and end of elements, which are the logical units of information in an XML document.

An element consists of a start tag, possibly followed by text and other complete elements, followed by an end tag. The following example highlights the tags to distinguish them from the text:

<person>Tony Blair</person> is

<function>Prime Minister </function> of

<location><country>Great Britain</country></location>

Note that the end tags include a solidus ("/") before the element's name. There are five elements in this example:

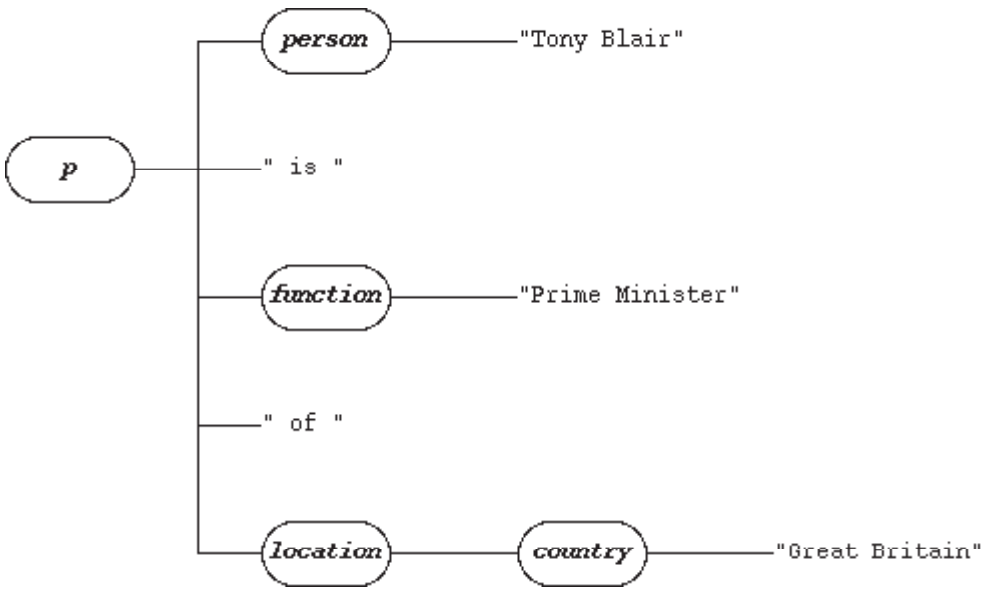
1. the p element, that contains the entire example (the person element, the text " is ", the function element, the text " of ", and the location element);
2. the person element, that contains the text "Tony Blair";

3. the function element, that contains the text “Prime Minister”;

4. the location element, that contains the country element; and

5. the country element, that contains the text “Great Britain”.

The following illustration shows this structure as a tree, with p (the outermost element) at the root:



There are few rules to keep in mind about XML elements:

1. Elements may not overlap: an end tag must always have the same name as the most recent unmatched start tag. The following example is not well-formed XML, because “</person>” appears when the most recent unmatched start tag was “<function>”:

2. <!-- WRONG! -->

3. <function><person>President</function> Habibe</person>

The following example shows the tags properly nested:

<person><function>President</function> Habibe</person>

4. An XML document has exactly one root element. As a result, the following example is not a well- formed XML document, because both the a and b elements occur at the top level:

5. <!-- WRONG! -->

6. <a>...

7. ...

The following example fixes the problem by including both the a and b elements within a new x root element:

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

8. XML element (and attribute) names are case-sensitive, so “location” and “Location” refer to different elements. This is a very nasty trap for people used to working with HTML or other SGML document types, because it can cause surprising bugs in processing software, or can even lead to malformed XML documents, as in the following example:

```
9.  <!-- WRONG! -->
10. <a href="pbear.html">polar bear</A>
```

This example will cause a parser error because an XML processor considers a and A to be separate elements, so the start and end tags do not match.

In some cases, an element may exist that has no content (for example, the HTML hr element), but the tag is still read by processors. Rather than type a start and end tag with nothing between them (for example, “<hr></hr>”), XML has a special empty-element tag that represents both the start tag and the end tag:

```
<p>Stuff<hr/>
More stuff.</p>
```

Attribute

In addition to marking the beginning of an element, XML start tags also provide a place to specify attributes. An attribute specifies a single property for an element, using a name/value pair. One very well known example of an attribute is href in HTML:

```
<a href="http://www.yahoo.com/">Yahoo!</a>
```

In this example, the content of the a element is the text “Yahoo!”; the attribute href provides extra information about the element (in this case, the Web page to load when a user selects the link).

Every attribute assignment consists of two parts: the attribute name (for example, href), and the attribute value (for example, http://www.yahoo.com/). There are a few rules to remember about XML attributes:

1. Attribute names in XML (unlike HTML) are case sensitive: HREF and href refer to two different XML attributes.
2. You may not provide two values for the same attribute in the same start tag. The following example is not well-formed because the b attribute is specified twice:

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

4. Attribute names should never appear in quotation marks, but attribute values must always appear in quotation marks in XML (unlike HTML) using the " or ' characters. The following example is not well-formed because there are no delimiters around the value of the b attribute:

```
5. <!-- WRONG! -->
```

6. ...

3.3 Empty and Non Empty Elements

In a more detailed hierarchy, the Complex Type is one of the following:

1. **An empty element, or**
2. **A non-empty element**
   - **a. With text only, or**
   - **b. With elements only, or**
   - **c. With text and other elements (mixed content)**

And each of the above categories can in turn be:

Empty Elements

An empty element has no content and may or may not have attributes. Consider the following example:

```
<xs:complexType name="emptyType">
  <xs:complexContent>
    <xs:extension base="xs:anyType">
      <xs:attribute name="anAttributeToTheEmptiness" type="xs:string"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

The key points to note are the xs:complexType tag which is standard for declaring complex types and the xs:complexContent tag which is used to create elements with no sub-elements. Also note that an attribute has been specified (called anAttributeToTheEmptiness) and the type extends “xs:anyType” since the tag of an empty element is inconsequential anyway.

Non Empty Elements

Non Empty elements as we have seen before, can be of three types – text only, elements only or mixed. Let us consider each now.

Text Only elements

These elements can only contain text and optionally may or may not have attributes. For instance, the type “bioData” below defines a Complex Type with text-only content:

```
<xs:complexType name="bioData">
  <xs:simpleContent>
    <xs:extension base="xs:string">
      <xs:attribute name="personName" type="xs:string"/>
```

```
        </xs:extension>
    </xs:simpleContent>
</xs:complexType>
```

And an example of an element based on the above complex type is:

<bioData personName=”Jaidev”>This is a short biodata of the person named Jaidev. He is the author of this XML Tutorial</bioData>

3.4 Processing Instruction for XML

A Processing Instruction (PI) is an SGML and XML node type, which may occur anywhere in the document, intended to carry instructions to the application.

Processing instructions are exposed in the Document Object Model as Node.PROCESSING_INSTRUCTION_NODE, and they can be used in XPath and XQuery with the 'processing-instruction()' command.

Syntax

An SGML processing instruction is enclosed within '<? and '>'.

An XML processing instruction is enclosed within '<?' and '?>', and contains a target and optionally some content, which is the node value, that cannot contain the sequence '?>'.

<?PITarget PIContent?>

The XML Declaration at the beginning of an XML document (shown below) is not a processing instruction, however its similar syntax has often resulted in it being referred to as a processing instruction.

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

Examples

The most common use of a processing instruction is to request the XML document be rendered using a stylesheet using the 'xml-stylesheet' target, which was standardized in 1999.[6] It can be used for both XSLT and CSS stylesheets.

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

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

Another use is the programming language PHP, which can be embedded within an (X)HTML document as shown in the following example.

<?php echo $a; ?>

The DocBook XSLT stylesheets understand a number of processing instructions to override the default behavior

3.5 Summary

The unit has tag and their attributes to know how to use them in XML page. To create a XML page have instruction set to use them. The unit also define the tags to access the data from the XML elements.

3.6 Self Assessment Question

1. Define the various tags and attributes of XML?
2. How can we access the data from XML elements?
3. Define the processing instruction for XML?
4. What are empty and non- empty elements?
5. Explain tag, attribute and namerules?

Unit - 4 : XML Namespace

Structure of the Unit:

4.0 Objective
4.1 Introduction
4.2 What is the use of Namespace
4.3 Prefix and Declaration
4.4 Default and Multiple Mamespace
4.5 Summary
4.6 Self Assessment Question

4.0 Objective

The objective of this unit is to create a centralized environment for our application, which is helpful to access data from one to another module in our application.

4.1 Introduction

XML namespaces are used for providing uniquely named elements and attributes in an XML document. They are defined in a W3C recommendation. An XML instance may contain element or attribute names from more than one XML vocabulary. If each vocabulary is given a namespace, the ambiguity between identically named elements or attributes can be resolved.

A simple example would be to consider an XML instance that contained references to a customer and an ordered product. Both the customer element and the product element could have a child element named id. References to the id element would therefore be ambiguous; placing them in different namespaces would remove the ambiguity.

4.2 What is the use of Namespace

A namespace name is a uniform resource identifier (URI). Typically, the URI chosen for the namespace of a given XML vocabulary describes a resource under the control of the author or organization defining the vocabulary, such as a URL for the author's Web server. However, the namespace specification does not require nor suggest that the namespace URI be used to retrieve information; it is simply treated by an XML parser as a string. For example, the document at http://www.w3.org/1999/xhtml itself does not contain any code. It simply describes the XHTMLnamespace to human readers. Using a URI (such as "http://www.w3.org/1999/xhtml") to identify a namespace, rather than a simple string (such as "xhtml"), reduces the probability of different namespaces using duplicate identifiers.

Although the term namespace URI is widespread, the W3C Recommendation refers to it as the namespace name. The specification is not entirely prescriptive about the precise rules for namespace names (it does not explicitly say that parsers must reject documents where the namespace name is not a valid Uniform Resource Identifier), and many XML parsers allow any character string to be used. In version 1.1 of the recommendation, the namespace name becomes an Internationalized Resource Identifier, which licenses the use of non-ASCII characters that in practice were already accepted by nearly all XML software. The term namespace URI persists, however, not only in popular usage, but also in many other specifications from W3C and elsewhere.

4.3 Prefix and Declaration

An XML namespace is declared using the reserved XML pseudo-attribute xmlns or xmlns:prefix, the value of which must be a valid namespace name.

For example, the following declaration maps the "xhtml:" prefix to the XHTML namespace:

xmlns:xhtml="http://www.w3.org/1999/xhtml"

Any element or attribute whose name starts with the prefix "xhtml:" is considered to be in the XHTML namespace, if it or an ancestor has the above namespace declaration.

It is also possible to declare a default namespace. For example:

xmlns="http://www.w3.org/1999/xhtml"

In this case, any element without a namespace prefix is considered to be in the XHTML namespace, if it or an ancestor has the above default namespace declaration.

Attributes are never subject to the default namespace. An attribute without an explicit namespace prefix is considered not to be in any namespace.

Namespace undeclaration

When an element carries the attribute xmlns="", the default namespace for that element and its descendants reverts to "no namespace": that is, unprefixed names are considered not to be in any namespace.

XML Namespaces 1.1 also introduces the option to undeclare other namespace prefixes. For example, if the attribute xmlns:p="" appears on an element, the namespace prefix p is no longer in scope (and therefore cannot be used) on that element or on its descendants, unless reintroduced by another namespace declaration.

Namespaces in APIs and XML object models

Different specifications have taken different approaches on how namespace information is presented to applications.

Nearly all programming models allow the name of an element or attribute node to be retrieved as a three-part name: the local name, the namespace prefix, and the namespace URI. Applications should avoid attaching any significance to the choice of prefix, but the information is provided because it can be helpful to human readers. Names are considered equal, if the namespace URI and local name match.

In addition, most models provide some way of determining which namespaces have been declared for a given element. This information is needed because some XML vocabularies allow qualified names (containing namespace prefixes) to appear in the content of elements or attributes, as well as in their names. There are three main ways this information can be provided:

- As attribute nodes named "xmlns" or "xmlns:xxx", exactly as the namespaces are written in the source XML document. This is the model presented by DOM.
- As namespace declarations: distinguished from attributes, but corresponding one-to-one with the relevant attributes in the source XML document. This is the model presented by JDOM.

4.4 Default and Multiple Namespace

The use of default and namespace is performed when we works on single namespace but if we have multiple name space then we have to use prefix with them to all them at same place.

Default Namespaces

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

xmlns="namespaceURI"

This XML carries HTML table information:

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

This XML carries information about a piece of furniture:

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

Multiple Namespaces

XSLT is an XML language that can be used to transform XML documents into other formats, like HTML.

In the XSLT document below, you can see that most of the tags are HTML tags.

The tags that are not HTML tags have the prefix xsl, identified by the namespace xmlns:xsl="http://www.w3.org/1999/XSL/Transform":

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
<html>
<body>
```

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

4.5 Summary

The unit define purpose of namespace. The namespace also categorized according to their use like it is available as default and multiple. To create a namespace there are various syntaxes.

4.6 Self Assessment Question

1. Define the syntaxes to create the namespace?
2. Define the differences and similarities between default and multiple namespace?
3. What is use of prefix in declaration of namespace define with the help of an example?
4. Explain the use of namespace ?

Unit - 5 : XML Document Type Definition (DTD)

Structure of the Unit:

5.0 Objective
5.1 Introduction
5.2 XML DTD and XML Schema
5.3 Create DTD
5.4 Element Condition and Quantifier
5.5 Referencing DTD Declaration
5.6 Validating DTD Compliance
5.7 Summary
5.8 Self Assessment Question

5.0 Objective

The objective of this unit is to define how to create document definition and schema to create a XML page with different data of user's problem.

5.1 Introduction

If we have created your own XML elements, attributes, and/or entities, then you should create a DTD.

If we are creating an XML document using pre-defined elements/attributes/entities (i.e. ones that have been created by someone else), then a DTD should already exist. All you need to do is link to that DTD using the DOCTYPE declaration.

5.2 XML DTD and XML Schema

Document Type Definitions (DTD's) and XML Schemas (XSD's, also known as WXS) are industry-standard ways to define XML-based data models, and you'll find many tools and utilities for working with both DTD and XML Schema in Stylus Studio. There are many technical benefits of migrating older DTDs to XML Schema, including:

- Support for primitive (built-in) data types (eg: xsd:integer, xsd:string, xsd:date, and so on), which facilitates using XML in conjunction with other typed-data, including relational data.
- The ability to define custom data types, using object-oriented data modeling principles: encapsulation, inheritance, and substitution.
- Compatibility other XML technologies, for example, Web services, XQuery, XSLT and other technologies can optionally be schema-aware.

5.3 Create DTD

A document type definition contains a set of rules that can be used to validate an XML file. After you have created a DTD, you can edit it, adding declarations that define elements, attributes, entities and notations for any XML files that reference the DTD file. You can also establish constraints for how each element, attribute, entity and notation may be used within any XML files that reference the DTD file.

The following instructions were written for the XML perspective, but they will also work in many other perspectives.

Follow these steps to create a new DTD:

1. If necessary, create a project to contain the DTD. If you intend to generate Java™ beans from your DTD, we recommend you create a project that is configured to work with Java source to contain both the DTD and the beans (the beans must be contained in a project configured to work with Java source, however, the DTD does not have to be). Otherwise, it does not matter what kind of project you create.
2. In the workbench, select File > New > Other > XML > DTD and click Next.
3. Click the Create DTD file from scratch radio button.
4. Click Next.
5. Select the project or folder that will contain the DTD.
6. In the File name field, type the name of the DTD, for example MyDTD.dtd. The name of your DTD file must end with the extension .dtd
7. Click Finish.

The DTD appears in the Navigator view and automatically, by default, opens in the DTD editor. In the DTD editor, you can add elements, attributes, notations, entities, and comments to the DTD. If you close the DTD editor, and want to later re-open the file in it, double-click the file in the Navigator view.

5.4 Element Condition and Quantifier

General syntax elements

<comment> </comment>

Inserts a comment. Comment elements can be inserted anywhere inside any SCL element and are considered to be 'attached' to the immediate parent element. Comment elements can comprise any text, can be mixed content, and can have any user-defined attributes; they are ignored by logical processors, but conforming SCL applications are required to preserve them and their position relative to other elements. Comments inside other comments are considered to be comments on the comment; to add comments, place new comment elements adjacent to the comment. Note that SCL text inside a comment is not considered to be part of the SCL containing element.

Several elements also allow comments as attribute values; this allows comments to be attached to in-line SCL text without altering the syntactic form of the text, where this is required.

Top-level syntax elements

<scl> </scl>

Used to surround any piece of SCL content, to mark it off from other content in a document. Text inside this element should be legal SCL. It need not be a module (ontology).

attributes:

dialect (optional) Indicates that the PCDATA in the element is SCL written in a particular dialect. This is

used to include SCL text written in a non-XML dialect within legal XCL. For example, this is a legal SCL element:

<scl dialect="scl:core">(forall ex:romanceNovel ((x man)) (exists ((y woman)) (and (loves x y) (not (loves y x)))))</scl>

If this attribute is present then the body of the element should not be treated as mixed content.

The element may contain other user-defined attributes.

children:

<module> </module> (optional)

<phrase> </phrase> (optional) Used to indicate a top-level sentence in SCL text. The child element should be a single sentence element. This element may be omitted, and the sentence element incorporated directly in the text.

<comment> </comment> (optional) inserts a comment.

<module> </module>

Indicates that the enclosed text is considered to be a module (aka ontology). An SCL module is a piece of SCL text which has special attributes and satisfies certain extra constraints. Normally a piece of SCL content will comprise a single module, so this element, if present, will be the single child element of the <scl> element; but this is not required.

attributes:

moduleName (required) Used to assign an 'importing name' to a module. Normally this will be a URI reference, and often it will be the same as the xmlns default namespace and/or the URL of the containing document. However, this coincidence of naming is not required. No logical relationship is assumed between names based on their URI or XML namespace structure, so it is acceptable to use a URI reference containing a fragID to name an ontology. One document may contain several modules named with URI references formed from the URI of the document itself. (Intuitively, rather than being required to be the absolute URI of a namespace, the ontology importing name may be treated like other names in a namespace. This allows one ontology to treat another ontology as an entity in its namespace.)

dialect (optional) Indicates that the PCDATA in the element is SCL written in a particular dialect.

children:

<import /> (optional)

<nothings> </nothings> (optional)

<phrase> </phrase> (optional)

<comment> </comment> (optional)

<phrase> </phrase>

Used to indicate a top-level sentence in SCL text. The single child element should be a sentence element. This element may be omitted, and the sentence element incorporated directly in the text.

attributes:

dialect (optional) Indicates that the PCDATA in the element is SCL written in a particular dialect. This overrides any enclosing dialect attributes on parent elements; however, such clashes between parent and child dialect values are deprecated.

<import source="..." />

Empty element used to import one module into another. The value of the attribute should be the module name of an SCL module.

attributes:

The dialect attribute may optionally be included in an import element to indicate that the source should be encoded in that dialect. Conforming applications which find the SCL source to be encoded in a different dialect than the one indicated may either treat this as an error condition, or translate the source into the indicated dialect.

This element has no child elements.

<nothings> </nothings>

The PCDATA in this element is a sequence of SCL names. This element is used to indicate that these names do not denote 'things' in the current ontology, i.e. that they are nondenotingoperators not included in the local domain of quantification. This allows an ontology to have strict control over the size of its intended domain of discourse. Names not listed in such a header are treated as denoting names in the ontology.

Nondenoting names can be used only in the relation or function position of atoms and terms in this module. They cannot be used as arguments or be bound by quantifiers in this module. Note however that a name may be nondenoting in one ontology but denoting in another ontology: the assignment of nondenoting status is local to this module.

This element has no attributes and no child elements.

Sentence elements

General attributes applying to all sentence elements:

The dialect attribute should not be used inside a sentence element.

Any sentence element may have the attribute lfo (logical form of) whose value is some content (usually a URIreference) which indicates a syntactic category in some external language specification. This is intended to indicate that the sentence is the result of a content-preserving translation into SCL of the syntactic form indicated. Any lfo attributes of subelements of this element should indicate the appropriate categories of subexpressions of the external expression. For example, a Horn rule might appear in SCL as a univerally quantified implication between a conjunction and an atom; the lfo value of the

<forall> might be "rulespec:hornrule", and that of the internal conjunction and atomic conclusion might then be "rulespec:body" and "rulespec:head". Only one 'external' translation can be specified for a given piece of SCL. To indicate that a single SCL sentence is the logical form of two distinct external syntaxes, it is necessary to write it twice, with suitable lfo values for each external specification. (The repetition may be indicated by the use of XML standards external to SCL.) Currently there are no such external mappings defined.

Syntax type

Quantified and boolean sentences can be stated 'generically' with the attribute syntaxtype whose value is some content (usually a URIreference) which specifies or indicates the appropriate logical form. This attribute is provided for extensibility, to allow the inclusion of alternative logical syntax forms or patterns, eg numerical quantifiers or alternative boolean operators such as a binary conjunction or a Scheffer stroke. Currently there are no such extensions defined.

The attribute syntaxtype cannot be used with the normal SCL quantifier and boolean sentence elements, which can be viewed as abbreviations of generic elements with scl:-prefixed properties, eg

<forall>...</forall> abbreviates

<quantifiedSentence syntaxtype="scl:forall">...</quantifiedSentence>

Quantified sentence elements

<forall> </forall>

<exists> </exists>

These both enclose a quantified sentence and indicate the quantifier. The alternative generic form is

<quantifiedSentence syntaxtype="...."></quantifiedSentence>

attributes:

lfo (optional)

children:

<guard></guard> (optional; unique)

<bvar></bvar> (required)

[any sentence element] (required; unique)

<comment> </comment> (optional)

<guard> </guard>

This optional element contains a name which is used as the guard of the quantifier, if present. This has no attributes and no children. It is used only inside a quantifier element, and must be the first child element if present.

<bvar> </bvar>

Encloses a name bound by a quantifier. This must be a denoting name.

attributes:

sort (optional). Value must be a name indicating the sort of the variable. This can be a nondenoting name.

No children.

Boolean sentence elements

<and> </and>

<or> </or>

<implies> </implies>

<iff> </iff>

<not> </not>

These all enclose boolean sentences and indicate the truth-function. The alternative generic form is

<booleanSentence syntaxtype="...."> </booleanSentence>

The only one of these which is order-sensitive is implies. By convention, the first child element is the antecendent, the second is the conclusion. (SCL does not require 'role' elements for boolean sentences, eg. antecedent/conclusion.)

children:

Any boolean sentence element can have children of any sentence type. And and or take any number of children (including zero); implies and if take two, and not takes one.

<comment></comment> (optional)

attributes:

lfo (optional)

Atomic sentence elements

<holds> </holds>

Atomic sentence indicating that a relation holds true of some arguments. The first child element always indicates the relation; this may be a nondenoting name. SCL allows two distinct forms for specifying arguments of a relation, as an ordered list or as a collection of role-value pairs. The latter form is indicated by the attribute syntaxtype on the holds element with the value roleset .

attributes:

lfo (optional)

syntaxtype (optional)

children:

One <rel> element (required)

either:

some number (maybe zero) of <name> or <app> elements;

or:

some number (maybe zero) of <role> elements.

<comment> </comment> (optional)

<rel> </rel>

Encloses a name or a <app>. The name may be nondenoting.

No attributes.

children:

<comment> </comment> (optional)

<app> </app> (optional: unique)

<equal> </equal>

No attributes.

Children:

exactly two <name> or <app> elements.

<role> </role>

attributes:

name. (Required) The value is a name which indicates the role of the content in the atom. Role names denote binary relations in SCL, and may be nondenoting.

The content of this element is a name or <app> element indicating the argument 'filler' of the slot.

Example of an atom using the role syntax:

<holds syntaxtype="roleset"> <rel>married</rel>

<role name="wife">Jill</role>

<role name="husband">Jack</role>

</holds>

5.5 Referencing DTD Declaration

The Document Type Declaration is a file that contains the necessary rules that the XML code in this file must follow. You may think of the DTD as the grammar that the XML document must abide by to be a valid XML file.

In later lessons we will discuss how to create your own DTD, which will allow you to make your own XML rules. For now we will simply show you how to reference an existing DTD file.

referencing an external dtd

There are two type declarations that may be used to reference an external DTD: PUBLIC and SYSTEM. When creating an XML document under the rules of a publicly distributed DTD, use PUBLIC. Otherwise, use the SYSTEM type declaration.

The example below shows a prolog that would be used for an HTML document that is using an XML prolog. The DTD is publicly available thanks to the W3C.

XML Code:

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"

"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

Let's take a look at each piece of this external DTD reference.

- !DOCTYPE - Tell the XML processor that this piece of code defines the Document Type Definition
- html - Specifies the root element of the XML document. Here our example is an HTML file, which has <html> as the root element.
- PUBLIC - Specifies that this is a publicly available DTD.

5.6 Validating DTD Compliance

Validating With the XML Parser

If you try to open an XML document, the XML Parser might generate an error. By accessing the parseError object, you can retrieve the error code, the error text, or even the line that caused the error.

Note: The load( ) method is used for files, while the loadXML( ) method is used for strings.

Example

```
var xmlDoc = new ActiveXObject("Microsoft.XMLDOM");
xmlDoc.async="false";
xmlDoc.validateOnParse="true";
xmlDoc.load("note_dtd_error.xml");
document.write("<br />Error Code: ");
document.write(xmlDoc.parseError.errorCode);
document.write("<br />Error Reason: ");
```

```
document.write(xmlDoc.parseError.reason);
document.write("<br />Error Line: ");
document.write(xmlDoc.parseError.line);
```

Turn Validation Off

Validation can be turned off by setting the XML parser's validateOnParse="false"

.**Example**

```
var xmlDoc = new ActiveXObject("Microsoft.XMLDOM");
xmlDoc.async="false";
xmlDoc.validateOnParse="false";
xmlDoc.load("note_dtd_error.xml");
document.write("<br />Error Code: ");
document.write(xmlDoc.parseError.errorCode);
document.write("<br />Error Reason: ");
document.write(xmlDoc.parseError.reason);
document.write("<br />Error Line: ");
document.write(xmlDoc.parseError.line);
```

Errors in XML documents will stop your XML program.

The W3C XML specification states that a program should not continue to process an XML document if it finds an error. The reason is that XML software should be easy to write, and that all XML documents should be compatible.

With HTML it was possible to create documents with "errors" (like dropping end tags). One of the main reasons that HTML browsers are so big and incompatible, is that they have their own ways to figure out what a document should look like when they encounter an HTML error.

With XML this should not be possible.

5.7 Summary

To create a XML page with well formed logic we also have use the validater to confirm the validate of data. The unit have use of validater and also provide use of DTD to create the page.

5.8 Self Assessment Question

1. What is DTD? Define steps to create it?
2. How can we validate the data in XML?
3. Define the element conditions for DTD?
4. Explain automatic sentence element?
5. Explain quantified sentence element?
6. Explain quantifiers with suitable example?

Unit - 6 : XML Schema Definition

Structure of the Unit:

6.0 Objective
6.1 Introduction
6.2 Element and Attribute Declaration
6.3 Simple, Complex and Built in Type
6.4 Associating XML with a Schema
6.5 Validate XSD
6.6 Summary
6.7 Self Assessment Question

6.0 Objective

This unit is design to provide introduction about the schema and their data types. Unit also provide concepts to use the validator with schema.

6.1 Introduction

XML Schema Definition (XSD) language is the current standard schema language for all XML documents and data. On May 2, 2001, the World Wide Web Consortium (W3C) published XSD in its version 1.0 format.

The XML Schema definition language (XSD) enables you to define the structure and data types for XML documents. An XML Schema defines the elements, attributes, and data types that conform to the World Wide Web Consortium (W3C) XML Schema Part 1: Structures Recommendation for the XML Schema Definition Language. The W3C XML Schema Part 2: Datatypes Recommendation is the recommendation for defining data types used in XML schemas. The XML Schema Reference (XSD) is based on the W3C 2001 Recommendation specifications for Datatypes and for Structures

6.2 Element and Attribute Declaration

The attribute element defines an attribute.

Element Information

- Parent elements: attributeGroup, schema, complexType, restriction (both simpleContent and complexContent), extension (both simpleContent and complexContent)

Syntax

<attribute

default=string

fixed=string

form=qualified|unqualified

id=ID

name=NCName

ref=QName

type=QName

use=optional|prohibited|required

any attributes

>

(annotation?,(simpleType?))

</attribute>

(The ? sign declares that the element can occur zero or one time inside the attribute element)

Attribute	**Description**
Default	Optional. Specifies a default value for the attribute. Default and fixed attributes cannot both bepresent
Fixed both	Optional. Specifies a fixed value for the attribute. Default and fixed attributes cannot be present
Form	Optional. Specifies the form for the attribute. The default value is the value of the attributeFormDefault attribute of the element containing the attribute. Can be set to one of the following:
prefix	• "qualified" - indicates that this attribute must be qualified with the namespace and the no-colon-name (NCName) of the attribute • unqualified - indicates that this attribute is not required to be qualified with the namespace prefix and is matched against the (NCName) of the attribute
Id	Optional. Specifies a unique ID for the element
Name	Optional. Specifies the name of the attribute. Name and ref attributes cannot both be present
Ref	Optional. Specifies a reference to a named attribute. Name and ref attributes cannot both be present. If ref is present, simpleType element, form, and type cannot be present
type	Optional. Specifies a built-in data type or a simple type. The type attribute can only be present when the content does not contain a simpleType element
Use	Optional. Specifies how the attribute is used. Can be one of the following values: • optional - the attribute is optional (this is default) • prohibited - the attribute cannot be used • required - the attribute is required
any attributes	Optional. Specifies any other attributes with non-schema namespace

6.3 Simple, Complex and Built in Type

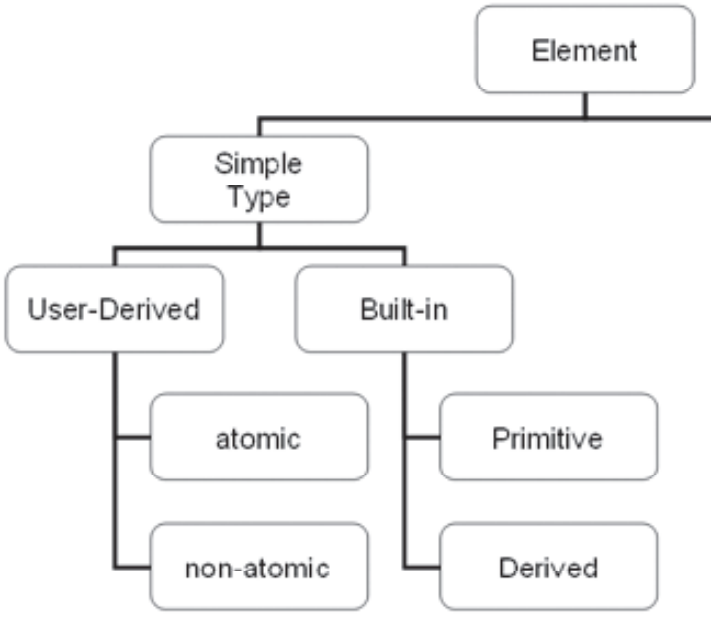
Simple-type elements have no children or attributes. For example, the Name element below is a simple-type element; whereas the Person and HomePage elements are not.

Code Sample: SimpleTypes/Demos/SimpleType.xml

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

<Person>

 <Name>Mark Twain</Name>

 <HomePage URL="http://www.marktwain.com"/>

</Person>
```

Code Explanation

As the diagram below shows, a simple type can either be built-in or user-derived. In this lesson, we will examine both.



Built-in Simple Types

XML Schema specifies 44 built-in types, 19 of which are primitive.

19 Primitive Data Types

The 19 built-in primitive types are listed below.

1. string
2. boolean

3. decimal
4. float
5. double
6. duration
7. dateTime
8. time
9. date
10. gYearMonth
11. gYear
12. gMonthDay
13. gDay
14. gMonth
15. hexBinary
16. base64Binary
17. anyURI
18. QName
19. NOTATION

Built-in Derived Data Types

The other 25 built-in data types are derived from one of the primitive types listed above.

1. normalizedString
2. token
3. language
4. NMTOKEN
5. NMTOKENS
6. Name
7. NCName
8. ID
9. IDREF
10. IDREFS
11. ENTITY
12. ENTITIES

13. integer
14. nonPositiveInteger
15. negativeInteger
16. long
17. int
18. short
19. byte
20. nonNegativeInteger
21. unsignedLong
22. unsignedInt
23. unsignedShort
24. unsignedByte
25. positiveInteger

Defining a Simple-type Element

A simple-type element is defined using the type attribute.

Code Sample: SimpleTypes/Demos/Author.xsd

```
<?xml version="1.0" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="Author">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="FirstName" type="xs:string"/>
    <xs:element name="LastName" type="xs:string"/>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

Code Explanation

Notice the FirstName and LastName elements in the code sample below. They are not explicitly defined as simple type elements. Instead, the type is defined with the type attribute. Because the value (string in both cases) is a simple type, the elements themselves are simple-type elements.

6.4 Associating XML with a Schema

Creating a New Windows Application Project

To begin this walkthrough you will first need to create a new Windows Application in Visual Basic or Visual C#.

To create a new project and name it "XMLCustomerList"

1. From the File menu, point to New, and then click Project to display the New Project dialog box.
2. Depending on what language you want to use, select Visual Basic Projects or Visual C# Projects in the Project Types pane, and then select Windows Application.
3. Name the project XMLCustomerList, then click OK to create the project.

Visual Studio will add the XMLCustomerList project to Solution Explorer.

Add a New XML File item to your Project

Because this walkthrough requires an XML file, you need to add one to your project.

To add a new XML file item to the project

1. From the Project menu, select Add New Item.

   The Add New Item dialog box appears.
2. In the Templates area of the Add New Item dialog box, select XML File.
3. Name the XML file CustomerList, and then click Open.

Visual Studio will add a file called CustomerList.xml to the project and automatically load it into the designer.

Add a New XML Schema item to your Project

Because this walkthrough requires an XML Schema, you need to add one to your project.

To add a new XML Schema item to the project

1. From the Project menu, select Add New Item.

   The Add New Item dialog appears.
2. In the Templates area of the Add New Item dialog box select XML Schema.
3. Name the schema CustomerListSchema and then click Open.

Visual Studio will add a file called CustomerListSchema.xsd to the project and automatically load it into the designer.

Add a Simple Type Definition to your Schema

Now you need to define the elements that will contain the data in your XML file. Because an XML Schema defines the data in an associated XML file, you create the element definitions in the schema.

The first definition will be a simple type element that will be used to define a standard US postal code (ZIP code). For this walkthrough we will use 5-digit codes only.

To create a simple type element that represents a 5 digit postal code

1. From the XML Schema tab of the Toolbox, drag a simpleType onto the design surface.
2. Select the default name simpleType1 and rename this type to postalCode.
3. Use the TAB key to navigate one cell to the right and select positiveInteger from the drop-down list.
4. Use the TAB key to navigate to the next row.
5. Click the drop-down box.

The only choice is facet. This is because simple types cannot include elements or attributes as part of their content models. Only facets can be used to build simple types.

6. Use the TAB key to navigate one cell to the right and select pattern from the drop-down list.
7. TAB over one cell to the right again and type \d{5}.

```
<xs:simpleType name="postalCode">
        <xs:restriction base="xs:positiveInteger">
          <xs:pattern value="\d{5}" />
         </xs:restriction>
</xs:simpleType>
```

8. From the File menu, select Save All.

Add a Complex Type Definition to your Schema

The next definition will be a complex type element that will be used to define an address. As part of this complex type definition, we will use the simple type created in the previous steps.

To create a complex type element that represents a standard US address

1. Switch back to Schema view.
2. From the XML Schema tab of the Toolbox, drag a complexType onto the design surface.
3. Select the default name complexType1 and rename this type to usAddress. Do not select a data type for this element.
4. Using the TAB key, navigate to the next row.
5. Click the drop-down list box to see the many choices of elements you can add to a complex type. You can select element, but for the rest of the walkthrough you will just TAB over this cell because element is the default.
6. Using the TAB key, navigate one cell to the right and type Name.
7. TAB one cell to the right and set the data type to string.
8. Repeat Steps 4 – 7 and create new rows in the usAddress element for the following:

Element name	Data type
Street	string
City	string
State	String
Zip	postalCode

9. Notice the data type that is assigned to the Zip element. It is the postalCode simple type you created previously.

10. If you switch to XML view, you should see the following code within the root level schema tags (that means the code sample does not include the actual declaration part of the schema, nor does it include the actual schema tags which are called the root or document level tags):

```
<xs:simpleType name="postalCode">
  <xs:restriction base="xs:positiveInteger">
    <xs:pattern value="\d{5}" />
  </xs:restriction>
</xs:simpleType>
<xs:complexType name="usAddress">
  <xs:sequence>
    <xs:element name="Name" type="xs:string" />
    <xs:element name="Street" type="xs:string" />
    <xs:element name="City" type="xs:string" />
    <xs:element name="State" type="xs:string" />
    <xs:element name="Zip" type="postalCode" />
  </xs:sequence>
</xs:complexType>
```

You have now defined two separate types that can be used in element definitions as well as types.

11. From the File menu, select Save All.

Add the Main Elements to your Schema

Now that you have defined some data types lets construct the actual data definition for the XML file you will be creating. The XML file will contain the data for the customer list, so lets create the actual element that will define the data that will be valid in the XML file.

To create the Customer element

1. Switch to Schema view.

2. From the XML Schema tab of the Toolbox, drag an element onto the design surface.

3. Select the default name element1 and rename this to customer. Do not select a data type for this element.

4. Using the TAB key navigate to the center cell of the next row, and type CompanyName.

5. TAB one cell to the right and set the data type to string.

6. Repeat Steps 4 – 5 and create new rows in the Customer element for the following:

Element name	**Data type**
ContactName	string
Email	string
Phone	string
BillToAddress	USAddress
ShipToAddress	USAddress

7. Notice the data type that is assigned to the BillToAddress as well as the ShipToAddress ele ments. It is the USAddress complex type created previously. We could have defined simple types for the Email, Phone elements and so on.

8. If you switch your schema to XML view, you should see the following code within the root-level schema tags (that means the code sample does not include the actual declaration part of the schema, nor does it include the actual schema tags which are called the root or document level tags):

```
<xs:simpleType name="postalCode">
  <xs:restriction base="xs:positiveInteger">
    <xs:pattern value="\d{5}" />
  </xs:restriction>
</xs:simpleType>
<xs:complexType name="usAddress">
  <xs:sequence>
    <xs:element name="Name" type="xs:string" />
    <xs:element name="Street" type="xs:string" />
    <xs:element name="City" type="xs:string" />
    <xs:element name="State" type="xs:string" />
    <xs:element name="Zip" type="postalCode" />
  </xs:sequence>
</xs:complexType>
<xs:element name="Customer">
```

```
    <xs:complexType>
      <xs:sequence>
        <xs:element name="CompanyName" type="xs:string" />
        <xs:element name="ContactName" type="xs:string" />
        <xs:element name="Email" type="xs:string" />
        <xs:element name="Phone" type="xs:string" />
        <xs:element name="ShipToAddress" type="usAddress" />
        <xs:element name="BillToAddress" type="usAddress" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
```

9. From the File menu, select Save All.

In order to allow multiple instances of customer data within the XML document, we will create an element named customerList that will contain all the individual customerelements.

To create the customerList element

1. From the XML Schema tab of the Toolbox, drag an element onto the design surface.
2. Select the default name element1 and rename this to customerList. Do not select a data type for this element.
3. Select the customer element (created previously) and drag it onto the customerList element.

   The individual design grids bind to represent the hierarchical structure of the data.
4. From the File menu, select Save All.

Associating the Schema and the XML file

In this part of the walkthrough, you will start working with the XML file. Within the XML file you will add a reference to the schema you just created.

To create an association between the XML file and the XML Schema

1. In Solution Explorer double-click the CustomerList.xml file.
2. The XML file opens in the designer in XML view.

   In the Properties window, click the cell to the right of the target Schema property and select http://tempuri.org/CustomerListSchema.xsd.

6.5 Validate XSD

Defining the XHTML Validation Schemas

To validate content against XHTML standards, follow these steps.

1. Using Visual Studio, open site root/Workarea/edit.aspx.vb.
2. Find the section indicated below.
3. Using Intellisense, enter SchemaFiles and SchemaNamespaces properties to define the validating Web sites. They accept an array of strings and go in pairs.

6.6 Summary

To perform on schema we have to understand the datatypes of it with different variations according to requirements. The schema also have various validation techniques to accept or process selected data by user.

6.7 Self Assessment Question

1. What do you mean by data types of XSD? Define various types with syntax?
2. How can we associate XML with schema?
3. Define an example to validate the data?

Unit - 7 : XQuery and XPath

Structure of the Unit:

7.0 Objective
7.1 Introduction
7.2 Importance of XQuery
7.3 What is XQuery?
7.4 What is XPath?
7.5 XPath Nodes and Types
7.6 Node Axes and Function
7.7 Structure of XQuery
7.8 Usages of XQuery
7.9 XPath and XSD
7.10 Selection and Filtration Elements
7.11 Summary
7.12 Self Assessment Question

7.0 Objective

The unit provide knowledge about the XML Query and its working in our application. It provide the details about.

* XQuery and It's structure.
* XPath and it's function to be implement.
* Nodes and their syntax for their implementation,
* Use of Xquery during Filter the data.

7.1 Introduction

XQuery was devised primarily as a query language for data stored in XML form. So its main role is to get information out of XML databases — this includes relational databases that store XML data, or that present an XML view of the data they hold.

Some people are also using XQuery for manipulating free-standing XML documents, for example, for transforming messages passing between applications. In that role XQuery competes directly with XSLT, and which language you choose is largely a matter of personal preference.

In fact, some people like XQuery so much that they are even using it for rendering XML into HTML for presentation. That's not really the job XQuery was designed for, and I wouldn't recommend people to do that, but once you get to know a tool, you tend to find new ways of using it.

7.2 Importance of XQuery

Since XML is primarily a syntax for messages, rather than a database model, one might ask why it needs a query language. It's also worth asking why existing query languages (such as SQL) don't meet the requirement.

There are two ways XQuery can be used with XML messages. It can be used to extract information from a single message, or it can be used to search a collection of messages.

Either way, there are a number of tasks you can perform with XQuery. These include:

* Finding the data you need, and perhaps extracting this data selectively to another tool
* Analyzing and aggregating the data: finding totals, averages, trends
* Transforming the data into another format. In this last role, XQuery overaps with another XML processing language, XSLT, which was designed more specifically for this purpose.

One difference between SQL and XQuery (in version 1.0, at any rate) is that XQuery is a read-only query language. This again reflects the fact that XML focuses on messages rather than long-term persistent data. Modifications to messages and documents are generally made by transforming an input document to an output document, and this is reflected in the design of XQuery, which allows creation of a new document, but does not allow the original document to be changed.

7.3 What is XQuery?

XQuery is a specification for a query language that allows a user or programmer to extract information from an Extensible Markup Language (XML) file or any collection of data that can be XML-like. The syntax is intended to be easy to understand and use. Using XQuery, it is possible to view a relational database table as an XML document. XQuery is an evolving specification under development by the World Wide Web Consortium (W3C) and has broad support from several major vendors including IBM, Microsoft, and Oracle.

In a relational database, data is stored in normalized tables. That means that whenever something can occur more than once, you need to create another table, and link it to the main table with some kind of identifier. For example, if a person can have more than one phone number, then the phone number can't be a column of the person table; instead, you have to create a new table containing just the person identifiers and phone numbers, with one entry for each phone number; and when someone does a query, they need to join data from the two tables using a SQL query like the one at the top of this article. The more flexible your data model becomes, the more you tend to find that almost any data can be optional or repeated: for example, a customer can have more than one account, an account can have more than one contact person, a contact person can have more than one address, an address can even, in the case of a large site, have more than one postal code. With SQL, every one-to-many relationship means another table, and every extra table adds complexity to the queries, even when you're interested in data that doesn't actually take advantage of any of the flexibility.

The XML data model is fundamentally different. Repeated data is the norm; data that can't repeat is treated as a special case. The operators in XQuery are designed to make it easy to work with repeated data.

7.4 What is XPath?

XPath is the result of an effort to provide a common syntax and semantics for functionality shared between XSL Transformations [XSLT] and XPointer [XPointer]. The primary purpose of XPath is to address parts of an XML [XML] document. In support of this primary purpose, it also provides basic facilities for manipulation of strings, numbers and booleans. XPath uses a compact, non-XML syntax to

facilitate use of XPath within URIs and XML attribute values. XPath operates on the abstract, logical structure of an XML document, rather than its surface syntax. XPath gets its name from its use of a path notation as in URLs for navigating through the hierarchical structure of an XML document.

This document is an informal guide to the syntax of XPath expressions, which are used in SAXON both within XSLT stylesheets, and in the Java API. For formal specifications, see the XSLT and XPath standards, except where differences are noted here.

We can classify expressions according to the data type of their result: string, number, boolean, node-set, and document-fragment. These categories are examined in the following sections.

SAXON expressions may be used either in an XSL stylesheet, or as a parameter to various Java interfaces. The syntax is the same in both cases. In the Java interface, expressions are encapsulated by the com.icl.saxon.Expression class, and are parsed using a call such as Expression.make("$a + $b"). To exploit the full power of XPath expressions in the Java API, you will need to supply some support classes to perform functions such as resolving namespace references: this cannot be done automatically because there is no stylesheet to use as a refernce point.

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

7.5 XPath Nodes and Types

In XPath, there are seven kinds of nodes: element, attribute, text, namespace, processing-instruction, comment, and document nodes. XML documents are treated as trees of nodes. The topmost element of the tree is called the root element. This set of nodes can contain zero or more nodes.

Look at the following XML document:

```
<?xml version="1.0" encoding="UTF-8"?>
 <list>
   <item new='true'>
     <title land="en">DVD Music</title>
     <author>K. A. Bred</author>
     <year>2012</year>
     <price currency="USD" >29.99</price>
   </item>
 </list>
```

Example of nodes in the XML document above:

<list> (root element node)

<author>K. Bred</author> (element node)

lang="en", new='true' (attribute nodes)

Atomic values are nodes with no children or parent. Items are atomic values or nodes.

1. Parent -

Each element and attribute has one parent. In the example above the itemelement is the parent of the title, author, year, and price.

2. Children -

Element nodes may have zero, one or more children. In the example above, the title, author, year and price elements are all children of the item element.

3. Siblings -

These are nodes that have the same parent. The the title, author, year, and price nodes are all siblings.

4. Ancestors -

These are node's parent, parent's parent, etc. In the example above, the ancestors of the title element are the item element and the list element.

5. Descendants -

These are node's children, children's children, etc. In the example, descendants of the list element are the item, title, author, year, and price nodes.

NodeSet Expressions

NodeSet expressions can be written

Construct	**Meaning**
nodeset-expression1 | nodeset-expression2	This forms the union of the two nodesets
nodeset-expression1 [predicate]	This returns the set of all nodes in nodeset-expression1 that satisfy the predicate. The predicate may be a boolean expression (which is evaluated with the particular node as current node, and the full node set as the current node set); or it may be a numeric expression, which is a short hand for the boolean expression position ()=predi cate. The nodeset-expression may of course itself have one or more predicates, so a chain of filters can be set up.
nodeset-expression1 / relative-path nodeset-	This follows the given path for each node in expression1 (the "original nodes"), and returns all the nodes reached (the "target nodes"). The

relative-path may be one of the following:

- name - Select all the element children of the original nodes with the given element name
- prefix:* - Select all the element children of the original nodes with the given namespace prefix
- * - Select all the element children of the original nodes regardless of element name
- @name - Select all the attributes of the original nodes with the given attribute
- name ancestor Selects ancestor nodes starting with the current node and ending with the document node ancestor-or Se lects the current node plus-self all ancestor nodes attribute Selects all attributes of the current node (if it is an element)child Se lects the children of the current node, in documetn order descendant Selects the children of the current node and their chil dren, recursively (in document order) de scendant - Selects the current node plus all de or-self scendant nodes following Selects the nodes that follow the current node in document order, other than its descendants following Selects all subsequent child nodes of ibling the same parent node Parent Se lects the parent of the current node pre ceding Selects the nodes that precede the current node in document order, other than its ancestors preceding -Selects all preced ing child nodes of sibling the same parent node Self Selects the current node
- name
- @prefix:* - Select all the attributes of the original nodes with the given namespace prefix
- @* - Select all the attributes of the origi nal nodes regardless of attribute name
- text() - Select all the text node children of the original nodes
- .. - Select the parents of the original nodes
- node() - Select all the children of the origi nal nodes
- axis-name :: node-test optional-predicates () - a generalised construct for navigating in any direction. The axis-name may be any of the following:
- The node-test may be:

	1. a node name 2. "prefix:*" to select nodes with a given namespace prefix 3. "text()" (to select text nodes) 4. "node()" (to select any node) 5. "processing-instruction()" (to select any processing instruction) 6. "processing-instruction('literal')" to select processing instructions with the given name (target) 7. comment() to select comment nodes The optional-predicates is a sequence of zero or more predicates, each enclosed in square brackets, each being either a boolean expression or a numeric expression (as a shorthand for testing position()).
nodeset-expression1 // relative-path	This is a shorthand for nodeset-expression1/ descen dant-or-self::node()/relative-path In effect "//" selects descendants, where "/" selects immediate children: but where predicates are used, the expansion above defines the precise meaning.
	This selects the current node
	This selects the document root node. Note that this nodeset-expression cannot be followed by the "/" or "//" operator or by a predicate.
/ relative-path	This is a shorthand for "root()/relative-path" where root() is an imaginary designation of the docu ment root node.
// relative-path	This is a shorthand for "root()//relative-path" where root() is an imaginary designation of the document root node.
document(expression1, expression2?)	The first string expression is a URL, or a nodeset containing a set of URLs; the function returns the nodeset consisting of the root nodes of the documents referenced (which must be XMLdocuments). The optional second argument is node-set used to provide a base URL for resolv ing relative URLs: the de fault is the URL of the document containing the relative URL, which may be either a source document or a stylesheet docu ment. Saxon allows the first argument to contain a fragment identifier, e.g. "my.xml#xyz", or simply

	"#xyz", in which case "xyz" must be the value of an ID attribute of an element within the referenced document. The effect is to retrieve a tree rooted at this element.
id(expression)	This returns the node, if any, that has an ID attribute equal to the given value,a nd which is in the same document as the current node. To use ID attributes, there must be a DTD that defines the attribute as being of type ID, and you must use a SAX parser that notifies ID attributes to the appli cation. If the argument is a nodeset, the function returns the set of nodes that have an id attribute equal to a value held in any of the nodes in the nodeset-expression: each node in the nodeset ex pression is converted to a string and treated as a white-space-separated list of id values. If the ar gument is of any other type, its value is converted to a string and treated as a white-space-separated list of id values.
key(string-expression1, expression2)	The first string expression is a key name; the function returns the set of nodes in the current document that have a key with this name, with the key value given by the second expression. If this is a nodeset, the key values are the values of the nodes in the nodeset; othewise, the key value is the string value of the argument. Note that keys must be registered using the xsl:key element.

Elements of XPath -

1. Constants

String literals are written as "London" or 'Paris'. In each case you can use the opposite kind of quotation mark within the string: 'He said "Boo"', or "That's rubbish". In a stylesheet XSL expressions always appear within XML attributes, so it is usual to use one kind of delimiter for the attribute and the other kind for the literal. Anything else can be written using XML character entities.

Numeric constants follow the Java rules for decimal literals: for example, 12 or 3.05; a negative number can be written as (say) -93.7, though technically the minus sign is not part of the literal. (Also, note that you may need a space before the minus sign to avoid it being treated as a hyphen within a preceding name).

There are no boolean constants as such: instead use the function calls true() and false().

2. Variable References

The value of a variable (local or global variable, local or global parameter) may be referred to using the

construct $name, where name is the variable name.

The variable is always evaluated at the textual place where the expression containing it appears; for example a variable used within an xsl:attribute-set must be in scope at the point where the attribute-set is defined, not the point where it is used.

A variable may take a value of any data type (string, number, boolean, node-set, or result-tree-fragment), and in general it is not possible to determine its data type statically.

It is an error to refer to a variable that has not been declared.

3. Parentheses and operator precedence

In general an expression may be enclosed in parentheses without changing its meaning. (There are places where parentheses cannot be used within a path-expression, however.)

If parentheses are not used, operator precedence follows the sequence below, starting with the operators that bind most tightly. Within each group the operators are evaluated left-to-right

Operator	Meaning
[]	predicate
/, //	child nodes, descendant nodes
|	Union
*, div, mod	multiply, divide, modulo
+, -	plus, minus
<, <=;, >, >=;	Less-than, less-or-equal, greater-than, greater-or-equal
=	equals
And	Boolean and
Or	Boolean or

4. String Expressions

There are some constructs that are specifically string expressions, but in addition any other kind of expression can be used in a context where a string expression is required:

- A numeric expression is converted to a string by giving its conventional decimal representation, for example the value -3.5 is displayed as "-3.5", and 2.0 is displayed as "2".
- A boolean expression is displayed as one of the strings "true" or "false".
- When a node-set expression is used in a string context, only the first node of the node-set (in document order) is used: the value of this node is converted to a string. The value of a text node

is the character content of the node; the value of any other node is the concatenation of all its descendant text nodes.

- A result tree fragment is technically converted to a string in the same way as a node-set; but since the corresponding node-set will always contain a single node, the effect is to generate all the descendant text nodes ignoring all element tags.

The specific string expressions are as follows:

Construct	Meaning
string(expression)	This performs an explicit type conversion to a string, which willalways give the same result as the implicit conversion described above. The main case where explicit conversion is useful is when assigning a value to a variable.
concat(expression1, expression2 may {,expression3}*)	This concatenates the string values of the arguments. There be any number of arguments (two or more).
substring(expression1, expression2 [,expression3]) is	This extracts a substring of the string value of expression1. Expression2 gives the start position (starting at 1), expression3 gives the length: if omitted, the rest of the string used. For example, substring("Michael", 2, 4) is "icha".
substring-before(expression1, first expression2)	This returns the substring of expression1 that precedes the occurrence of expression2. If expression1 does not contain expression2, it returns the empty string. For example, substring before("c:\dir", ":\") returns "c".
substring-after(expression1 , expression2)	This returns the substring of expression1 that follows the first occurrence of expression2. If expression1 does not contain expression2, it returns the empty string. For example, substring before("c:\dir", ":\") returns "dir".
normalize-space(expression1)	This removes leading and trailing white space, and converts all other sequences of white space to a single space character. For example, 'normalize(" Mike Kay ")' returns "Mike Kay"
translate(expression1, expression2, expression3)	This replaces any character in expression1 that also occurs in expression2 with the corresponding character from expression3. For example, translate ("ABBA", "ABC", "123") returns "1221". If there is no corresponding character in expression3 (because it is shorter than expression2), the character is removed from the string.
name(nodeset-expression)	Returns the name of the first node in the nodeset-expression, or the current node if the argument is omitted. The name here is the "display name"; it will use the same namespace prefix as

	in the original source document.
localpart(nodeset-expression)	Returns the local part (after the colon) of the name of the first node in the nodeset-expression, or the current node if the argument is omitted
Namespace-uri(nodeset-expression)	Returns the URI of the namespace of the name of the first node in the nodeset-expression, or the current node if the argument is omitted
unparsed-entity-uri(string-expression)	Returns the URI of the unparsed entity with the given name in the current document, if there is one; otherwise the empty string
generate-id(nodeset-expression)	Returns a system-generated identifier for the first node in the nodeset-expression, or the current node if the argument is omitted. The generated identifiers are always alphanumeric (except for the document node, where the identifier is the empty string), and have three useful properties beyond those required by the XSLT specification: • The alphabetic order of identifiers is the same as the document order of nodes • If generate-id(A) is a leading substring of generate-id(B), then A is an ancestor node of B • The identifier is unique not only within the document, but within all documents opened during the run.

5. Numeric Expressions

There are some constructs that are specifically numeric expressions, but in addition any string whose value is convertible to a number can be used as a number. (A string that does not represent any number is treated as zero).

A boolean is converted to a number by treating false as 0 and true as 1.

The specific numeric expressions are as follows:

Construct	**Meaning**
number(expression)	This performs an explicit type conversion to a number, which will always give the same result as the implicit conversion described above. Explicit conversion can be useful when assigning a value to a variable. It is also useful when creating an qualifier in a nodeset expression, since the meaning of a numeric qualifier is different from a boolean one.
count(node-set-expression)	This returns the number of nodes in the node-set.
sum(node-set-expression)	This converts the value of each node in the node-set to a number, and totals the result.

string-length(expression)	This returns the number of characters in the string value of expression. Characters are counted using the Java length() function, which does not necessarily give the same answer as the XPath rules, particularly when combining characters are used.
numeric-expression1 op numeric-expression2	This performs an arithmetic operation on the two values. The operators are + (plus), - (minus), * (multiply), div (divide), mod (modulo), and quo (quotient). Note that div does a floating point division; quo returns the result of div truncated to an integer; and n mod m returns n - ((n quo m) * m).
- numeric-expression2	Unary minus: this subtracts the value from zero. floor(numeric-expression1)This returns the largest inte ger that is <= the argument
ceiling(numeric-expression1)	This returns the smallest integer that is >= the argument
round(numeric-expression1)	This returns the closest integer to the argument. The round ing rules follow Java conventions which are not quite the same as the XSL rules.
position()	This returns the position of the current node in the current node list. Positions are numbered from one.
last()	This returns the number of nodes in the current node list

6. Boolean Expressions

Expressions of other types are converted to booleans as follows:

- Numeric values: 0 is treated as false, everything else as true.
- String values: the zero-length string is treated as false, everything else as true.
- Node-sets: the empty node set is treated as false, everything else as true.

The specific boolean expressions are as follows:

Construct	Meaning
boolean(expression)	This performs an explicit type conversion to a boolean, which will always give the same result as the implicit con version described above. The main case where explicit conversion is useful is when assigning a value to a variable.
false(), true()	These function calls return false and true respectively.
not(boolean-expression1)	This returns the logical negation of the argument.

expression1 ("=" | "!=") expression2	This tests whether the two values are equal (or not-equal). • An operand that is a result tree fragment is treated as if it were a node set containing a single node that acts as the root of the result tree fragment. • If both operands are node sets, it tests whether there is a value in the first node set that is equal (or not equal) to some value in the second node-set, treating the values as strings. Note that if either or both node sets is empty, the result will be false (re gardless of whether the operator is "=" or "!="). • If one operand is a node set and the other is a string or number, it tests whether there is a value in the node set that is equal (or not equal) to the other operand. If the node set is empty, the result will be false. • If one operand is a node set and the other is a boolean, it converts the nodeset to a boolean and compares the result. A nodeset that is empty is thus equal to false, while one that is non-empty is equal to true. • Otherwise if one operand is a boolean, both oper ands are converted to boolean and compared. • Otherwise if one operand is a number, both are converted to numbers and compared. • Otherwise, they are both converted to strings and compared; two strings are equal if they contain exactly the same characters.
numeric-expression1 op numeric-expression2	This performs a numeric comparison of the two values. If both expressions are node sets, the result is true if there is a pair of values from the two node sets that satisfies the comparison. If one expression is a nodeset, the result is true if there is a value in that nodeset that satisfies the com parison with the other operand. The operators are < (less-than), <= (less-or-equal), > (greater-than), >= (greater-or-equal). The operators, when used in an XSL stylesheet, will need to be written using XML entities such as "<".
lang(string-expression)	This returns true if the xml:lang attribute on (or inherited by) the current node is equal to the argument, or if it contains a suffix starting with "-" and ending with the ar gument, ignoring case.

7.6 Node Axes and Function

An axis stores certain information about the context node or other nodes within the document. The information it stores depends on the axis being used. For example, an axis called "child" stores information about the children of the context node. Therefore, we could use this axis to select a child from the context node.

Syntax

You use an axis by using it's name, along with a node test, and optionally, one or more predicates. The axis and node test are separated by ::, and the predicate/s are enclosed in [].

List of Axes

There are many other axes you can use within your XPath expressions. Here's a list of the axes you can use
with XPath:

Axis	Description
Ancestor	Contains the ancestors of the context node. Ancestors include the parent, and it's parent, and it's parent etc all the way back up to the root node.
ancestor-or-self	Contains the context node and it's ancestors.
Attribute	Contains the attributes of the context node.
Child	Contains the children of the context node.
descendant	Contains the descendants of the context node. Descendants include the node's children, and that child's children, and it's children etc (until there are no more children)
descendant-or-self	Contains the context node and it's descendants.
Following	Contains all nodes that come after the context node (i.e. after it's closing tag).
Following-sibling	Contains the following siblings of the context node. Siblings are at the same level as the context node and share it's parent.
namespace	Contains the namespace of the context node.
Parent	Contains the parent of the context node.
Preceding	Contains all nodes that come before the context node (i.e. before it's opening tag).
Self	Contains the context node.

Axis Function –

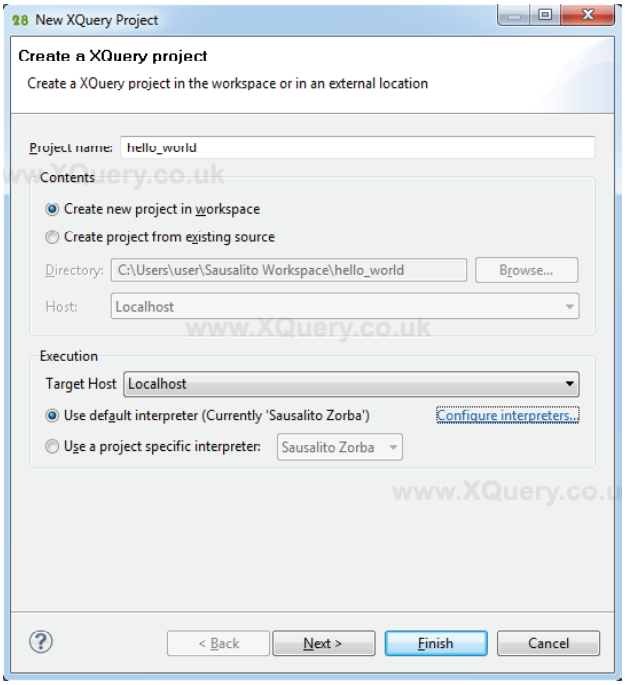
1. last()	-	Returns a number equal to the context size from the expression evaluation context.
2. position()	-	Function returns a number equal to the context positionfrom the expression evaluation context.
3. count(node-set)	-	Returns the number of nodes in the argument node-set.
4. node-set id(object)	-	selects elements by their unique ID.

7.7 Structure of XQuery

The structure of XQuery have set of components which can be define with help of steps using an example for each.

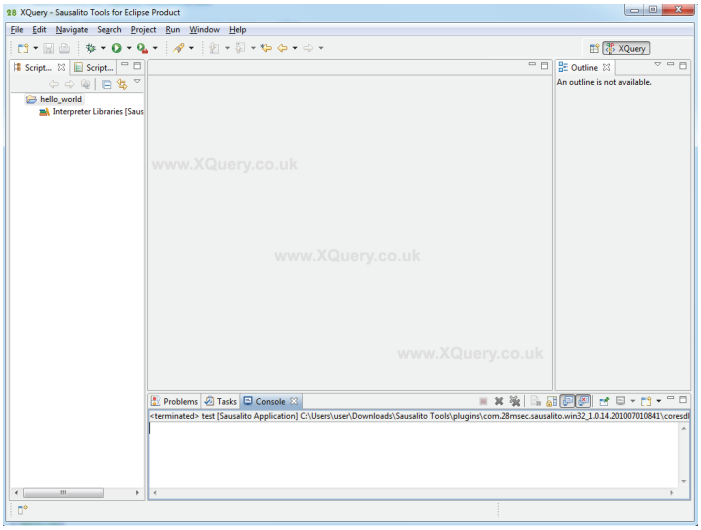
Create a new XQuery project

In the top main menu of the Sausalito Tools click File > New > XQuery Project. When the "New XQuery Project" wizzard popped up, enter a name for the new XQuery project. Click Finish.



Create a new XQuery project using the XQuery Project wizard.

Your first empty XQuery project should appear in the left explorer panel. Of course, it doesn't contain any XQuery code, yet.



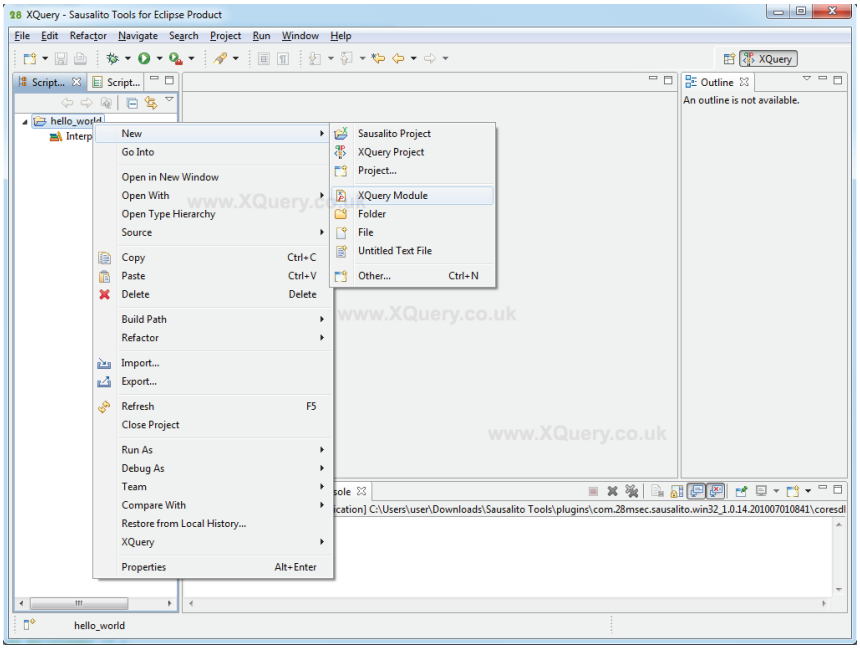
A plain new XQuery project without any XQuery code yet..

Create a New XQuery Module

Right click on the XQuery project name and select New > XQuery Module.

Add a new XQuery module to your XQuery project.

Enter a name for the new XQuery module and select "Main Module". A main XQuery module can directly be executed by an XQuery processor whereas a library module can only be imported by another module.



Enter a name for the new XQuery module file. The file ending '.xq' is common for XQuery modules.

The newly generated module will show up with some dummy content. In XQuery the '(:' is used to open a comment and the ':)' to close a comment. Comments are text parts ignored by the processor.

A new XQuery module containing some dummy content.

Write Some XQuery Code

Insert some XQuery code into the newly created XQuery module. For example insert:

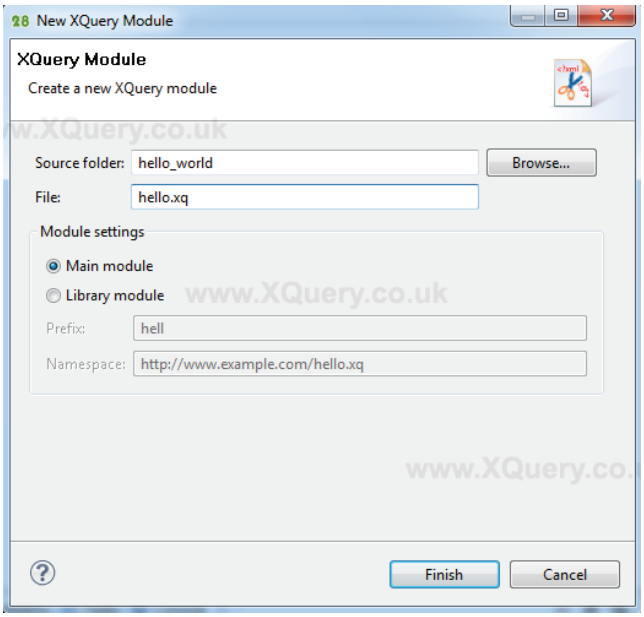
XQuery

```
<test>
   <message1>Hello World1</message1>
   <message2>Hello World2</message2>
</test>
```

XML is parsed as valid XQuery code.

In XQuery you can write XML data inline with no escaping needed. Therefore, plain XML is also parsed as valid XQuery code. In the given example, the output result is the same as the input:

```
<?xml version="1.0" encoding="UTF-8"?>
<test>
   <message1>Hello World1</message1>
   <message2>Hello World2</message2>
</test>
```



Execute Your XQuery Module

In order to run your first XQuery module simply:

1. make sure to save the latest version of your module file (shortcut to save the current file: 'Ctrl' + 's')
2. select the module file that you want to execute
3. press the green Run button
4. see the result in the bottom console window.

7.8 Usages of XQuery

XML is used in many aspects of web development, often to simplify data storage and sharing.

XML Separates Data from HTML

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

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

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

XML Simplifies Data Sharing

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

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

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

XML Simplifies Data Transport

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

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

XML Simplifies Platform Changes

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

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

XML Makes Your Data More Available

Different applications can access your data, not only in HTML pages, but also from XML data sources.

With XML, your data can be available to all kinds of "reading machines" (Handheld computers, voice machines, news feeds, etc), and make it more available for blind people, or people with other disabilities.

XML is Used to Create New Internet Languages

A lot of new Internet languages are created with XML.

Here are some examples:

* XHTML
* WSDL for describing available web services
* WAP and WML as markup languages for handheld devices
* RSS languages for news feeds
* RDF and OWL for describing resources and ontology
* SMIL for describing multimedia for the web

7.9 XPath and XSD

XML Schema Definition (XSD) language is the current standard schema language for all XML documents and data. On May 2, 2001, the World Wide Web Consortium (W3C) published XSD in its version 1.0 format.

The XML Schema definition language (XSD) enables you to define the structure and data types for

XML documents. An XML Schema defines the elements, attributes, and data types that conform to the World Wide Web Consortium (W3C) XML Schema Part 1: Structures Recommendation for the XML Schema Definition Language. The W3C XML Schema Part 2: Datatypes Recommendation is the recommendation for defining data types used in XML schemas. The XML Schema Reference (XSD) is based on the W3C 2001 Recommendation specifications for Datatypes and for Structures.

An XML Schema is composed of the top-level schema element. The schema element definition must include the following namespace:

http://www.w3.org/2001/XMLSchema

The schema element contains type definitions (simpleType and complexType elements) and attribute and element declarations.

In addition to its built-in data types (such as integer, string, and so on), XML Schema also allows for the definition of new data types using thesimpleType and complexType elements.

simpleType

A type definition for a value that can be used as the content (textOnly) of an element or attribute. This data type cannot contain elements or have attributes.

complexType

A type definition for elements that can contain attributes and elements. This data type can contain elements and have attributes.

In the remainder of this XSD Starter Kit, we will look at how to get started with XSD. For in-depth learning or reading on XSD, refer to the following draft recommendations that describe it:

* XML Schema Part 0: Primer
* XML Schema Part 1: Structures
* XML Schema Part 2: Datatypes.

7.10 Selection and Filtration Elements

Selecting and Filtering Elements

We are selecting and filtering elements with either a Path expression or with a FLWOR expression.

Look at the following FLWOR expression:

```
for $x in doc("books.xml")/bookstore/book
where $x/price>30
order by $x/title
return $x/title
```

* for - (optional) binds a variable to each item returned by the in expression
* let - (optional)
* where - (optional) specifies a criteria

* order by - (optional) specifies the sort-order of the result
* return - specifies what to return in the result

The for Clause

The for clause binds a variable to each item returned by the in expression. The for clause results in iteration. There can be multiple for clauses in the same FLWOR expression.

To loop a specific number of times in a for clause, you may use the to keyword:

```
for $x in (1 to 5)
return <test>{$x}</test>
```

The at keyword can be used to count the iteration:

```
for $x at $i in doc("books.xml")/bookstore/book/title
return <book>{$i}. {data($x)}</book>
```

It is also allowed with more than one in expression in the for clause. Use comma to separate each in expression:

```
for $x in (10,20), $y in (100,200)
return <test>x={$x} and y={$y}</test>
```

The let Clause

The let clause allows variable assignments and it avoids repeating the same expression many times. The let clause does not result in iteration.

```
let $x := (1 to 5)
return <test>{$x}</test>
```

The where Clause

The where clause is used to specify one or more criteria for the result:

```
where $x/price>30 and $x/price<100
```

The order by Clause

The order by clause is used to specify the sort order of the result. Here we want to order the result by category and title:

```
for $x in doc("books.xml")/bookstore/book
order by $x/@category, $x/title
return $x/title
```

The return Clause

The return clause specifies what is to be returned.

```
for $x in doc("books.xml")/bookstore/book
return $x/title
```

7.11 Summary

The unit provide introduction about the XML with their working with database and perform various task to manipulate user's data during working on web application or web services. In this unit we have the concept which define the method of XML to integrate it with Query and also enable the user to use different types of clause and function to process user's data.

7.12 Self Assessment Question

1. What is the difference between Xpath and Xquery? Define functions of Xpath?
2. How can we implement the Xquery in our application define with help of Structure?
3. What do you mean by Nodes? Define the various node set functions?
4. Explain the axis in XML? Define the various axes with their syntax?

Unit - 8 : Publishing XML

Structure of the Unit:

8.0 Objective
8.1 Introduction
8.2 What is Style Sheet & Language
8.3 Using Style Sheet with XML
8.4 Page Layout with Cascading Style Sheet
8.5 CSS Syntax and Classes
8.6 Introduction to XSL
8.7 Summary
8.8 Self Assessment Question

8.0 Objective

This unit provide the knowledge about how create a platform for web page and websites. In the unit we have the description about the style sheet and their use in our application. Unit include the topic -

* Style Sheet with XML
* Various Syntax of CSS
* Use of XSL

8.1 Introduction

To understand the significance of CSS, let's consider the analogy of a book. A book has both structure and appearance. The structure of a book involves levels of information, such as chapters, pages, paragraphs, and the textual content of the book. The appearance of a book involves page size, margin, indentations, font size, color, and so forth. When an author writes a book, the author focuses on the structure and organizes the textual information into chapters, pages, and paragraphs. After the manuscript is accepted by a publisher, the publisher focuses on the appearance of the book. Notice that because the structure and appearance are separated, the author and the publisher focus on the aspect of the book that is of most concern to them. So it is with web pages.

Web pages have structure and they have appearance. The XHTML markup language provides the structure, and CSS handles the appearance. The use of CSS can save the designer hours of time. The designer can change a rule in CSS to use a different font, for example, and every HTML page that uses that rule will automatically be updated to use the new font. In addition, CSS increases the accessibility of web sites by allowing visitors to control the appearance of the web pages (a boon to persons using PDAs to access web pages and to persons who might need special font sizes, colors, etc).

8.2 What is Style Sheet & Language

Style sheet language

A style sheet language, or style language, is a computer language that expresses the presentation of structured documents. One attractive feature of structured documents is that the content can be reused in many contexts and presented in various ways. Different style sheets can be attached to the

logical structure to produce different presentations.

One modern style sheet language with widespread use is Cascading Style Sheets (CSS), which is used to style documents written in HTML, XHTML, SVG, XUL, and other markup languages.

For content in structured documents to be presented, a set of stylistic rules – describing, for example, colors, fonts and layout – must be applied. A collection of stylistic rules is called a style sheet. Style sheets in the form of written documents have a long history of use by editors and typographers to ensure consistency of presentation, spelling and punctuation. In electronic publishing, style sheet languages are mostly used in the context of visual presentation rather than spelling and punctuation.

Components

All style sheet languages offer functionality in these areas:

Syntax

A style sheet language needs a syntax in order to be expressed in a machine-readable manner. For example, here is a simple style sheet written in the CSS syntax:

```
h1 { font-size: 1.5em }
```

Selectors

Selectors specify which elements are to be influenced by the style rule. As such, selectors are the glue between the structure of the document and the stylistic rules in the style sheets. In the example above, the "h1" selector selects all h1 elements. More complex selectors can select elements based on, e.g., their context, attributes and content.

Properties

All style sheet languages have some concept of properties that can be given values to change one aspect of rendering an element. The "font-size" property of CSS is used in the above example. Common style sheet languages typically have around 50 properties to describe the presentation of documents.

Values and units

Properties change the rendering of an element by being assigned a certain value. The value can be a string, a keyword, a number, or a number with a unit identifier. Also, values can be lists or expressions involving several of the aforementioned values. A typical value in a visual style sheet is a length; for example, "1.5em" which consists of a number (1.5) and a unit (em). The "em" value in CSS refers to the font size of the surrounding text. Common style sheet languages have around ten different units.

Value propagation mechanism

To avoid having to specify explicitly all values for all properties on all elements, style sheet languages have mechanisms to propagate values automatically. The main benefit of value propagation is less-verbose style sheets. In the example above, only the font size is specified; other values will be found through value propagation mechanisms. Inheritance, initial values and cascading are examples of value propagation mechanisms.

Formatting model

All style sheet languages support some kind of formatting model. Most style sheet languages have a visual formatting model that describes, in some detail, how text and other content is laid out in the final presentation. For example, the CSS formatting model specifies that block-level elements (of which "h1" is an example) extends to fill the width of the parent element. Some style sheet languages also have an aural formatting model.

8.3 Using Style Sheet with XML

To use the style sheets with XML we have some set of steps and applications. These are as follow.

Adding a style sheet to an XML document

To add a style sheet to an XML document, you need only to insert the following declaration at the beginning of your file.

Adding a style sheet to an XML document

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

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

Keep in mind that the href attribute of the style sheet declaration works exactly as its (X)HTML counter-part, the link element. In this case we're using a relativeURL because the style sheet shares the same directory with the XML document.

Choosing the appropriate markup

When choosing the markup for our document, we should only keep in mind that the names of the elements should be meaningful and semantic. For example, the markup for a blog could be the following.

Listing. The markup for a blog

```
<blog>

<header>

<title>My Blog</title>

</header>

<navigation>

<current>Home</current>

<nlink xmlns:xlink="http://www.w3.org/1999/xlink" xlink:type="simple"

xlink:href="#">Articles</nlink>

...omission...

</navigation>

<content>

<post>
```

```
<ptitle>Post title</ptitle>
<pdate>7/19/2008</ptitle>
<para>Lorem ipsum dolor...</para>
...omission...
<plink xmlns:xlink="http://www.w3.org/1999/xlink" xlink:type="simple"
xlink:href="#">Permanent link to this article</plink>
</post>
...omission...
</content>
<extra>
<etitle>Blogroll</etitle>
<elist>
<elink xmlns:xlink="http://www.w3.org/1999/xlink" xlink:type="simple"
xlink:href="#">Link 1</elink>
...omission...
</elist>
</extra>
<footer>
<signature>Gabriele Romanato</signature>
</footer>
</blog>
```

As you can see, the document is rendered as a whole set of anonymous text, since browsers can't actually use their default style sheet nor their predefined algorithms to give a basic formatting to our elements which are, in fact, completely unknown to them. However, the resulting DOM tree is perfectly consistent and we can summarize it as follows.

* blog

  It's the actual root element of our document.

  - header

    The header section of our blog.

    - title

      The document's title.

- navigation

  The navigation menu of our blog.

  - current

    The current section of our blog.

  - nlink

    A navigation link. To insert hyperlinks in a XML document, we can use XLink. For the time being, this feature is supported only by Gecko-based browsers.

- content

  The main section of our blog.

  - post

    A post of our blog.

  - ptitle

    The title of a post.

  - pdate

    The date of a post.

  - para

    A simple paragraph.

  - plink

    The permanent link to a post.

- extra

  A section with miscellaneous things.

  - etitle

  The title for this section.

  - elist

    A list of links.

  - elink

    A link to an external site, for example a friend's site.

- footer

  The footer section of our blog.

  - signature

    This element can contain the author's name or a copyright notice.

8.4 Page Layout with Cascading Style Sheet

A CSS page layout uses the cascading style sheets format, rather than traditional HTML tables or frames, to organize the content on a web page. The basic building block of the CSS layout is the div tag—an HTML tag that in most cases acts as a container for text, images, and other page elements. When you create a CSS layout, you place div tags on the page, add content to them, and position them in various places. Unlike table cells, which are restricted to existing somewhere within the rows and columns of a table, div tags can appear anywhere on a web page. You can position div tags absolutely (by specifying x and y coordinates), or relatively (by specifying their distance from other page elements).

Creating CSS layouts from scratch can be difficult because there are so many ways to do it. You can create a simple two-column CSS layout by setting floats, margins, padding, and other CSS properties in a nearly infinite number of combinations. Additionally, the problem of cross-browser rendering causes certain CSS layouts to display properly in some browsers, and display improperly in others. Dreamweaver makes it easy for you to build pages with CSS layouts by providing over 30 pre-designed layouts that work across different browsers.

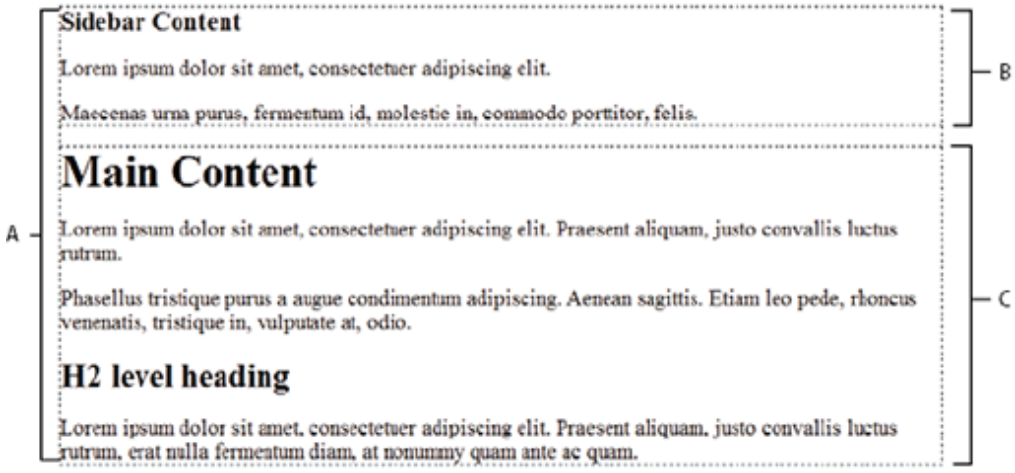
Using the pre-designed CSS layouts that come with Dreamweaver is the easiest way to create a page with a CSS layout, but you can also create CSS layouts using Dreamweaver absolutely-positioned elements (AP elements). An AP element in Dreamweaver is an HTML page element—specifically, a div tag, or any other tag—that has an absolute position assigned to it. The limitation of Dreamweaver AP elements, however, is that since they are absolutely positioned, their positions never adjust on the page according to the size of the browser window.

If you are an advanced user, you can also insert div tags manually and apply CSS positioning styles to them to create page layouts.

About CSS page layout structure

Before proceeding with this section, you should be familiar with basic CSS concepts.

The basic building block of the CSS layout is the div tag—an HTML tag that in most cases acts as a container for text, images, and other page elements. Figure 1 shows an HTML page that contains three separate div tags: one large “container” tag, and two other tags—a sidebar tag, and a main content tag—within the container tag.



Here is the code for all three div tags in the HTML:

```
<!--container div tag-->
<div id="container">
<!--sidebar div tag-->
        <div id="sidebar">
   <h3>Sidebar Content</h3>
   <p>Lorem ipsum dolor sit amet, consectetuer adipiscing elit.</p>
   <p>Maecenas urna purus, fermentum id, molestie in, commodo  porttitor, felis.</p>
        </div>
<!--mainContent div tag-->
        <div id="mainContent">
   <h1> Main Content </h1>
  <p>Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Praesent aliquam,  justo convallis luctus rutrum.</p>
    <p>Phasellus tristique purus a augue condimentum adipiscing. Aenean  sagittis. Etiam leo pede, rhoncus venenatis, tristique in, vulputate at, odio.</p>
   <h2>H2 level heading </h2>
  <p>Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Praesent aliquam,  justo convallis luctus rutrum, erat nulla fermentum diam, at nonummy quam  ante ac quam.</p>
        </div>
</div>
```

In the above example, there is no "styling" attached to any of the div tags. Without CSS rules defined, each div tag and its contents fall into a default location on the page. However, if each div tag has a unique id (as in the above example), you can use the ids to create CSS rules that, when applied, change the style and positioning of the div tags.

The following CSS rule, which can reside in the head of the document or in an external CSS file, creates styling rules for the first, div tag on the page, known as or container div tag:

```
#container {
        width: 780px;
        background: #FFFFFF;
        margin: 0 auto;
        border: 1px solid #000000;
        text-align: left;
}
```

The #container rule styles the container div tag to have a width of 780 pixels, a white background, no margin (from the left side of the page), a solid, black, 1 pixel border, and text that is aligned left. Figure 2 shows the results of applying the rule to the container div tag.

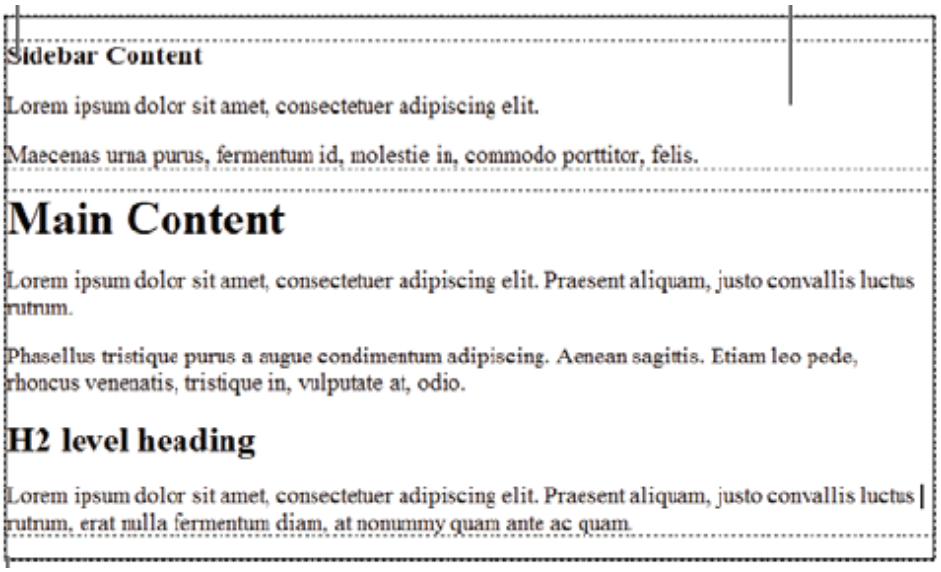


Figure. Container div tag, 780 pixels, no margin A. Text aligned left B. White background C. 1-pixel solid black border

The next CSS rule creates styling rules for the sidebar div tag:

```
#sidebar {
        float: left;
        width: 200px;
        background: #EBEBEB;
        padding: 15px 10px 15px 20px;
}
```

The #sidebar rule styles the sidebar div tag to have a width of 200 pixels, a gray background, a top and bottom padding of 15 pixels, a right padding of 10 pixels, and a left padding of 20 pixels. (The default order for padding is top-right-bottom-left.) Additionally, the rule positions the sidebar div tag with float: left—a property that pushes the sidebar div tag to the left side of the container div tag. Figure 3 shows the results of applying the rule to the sidebar div tag.

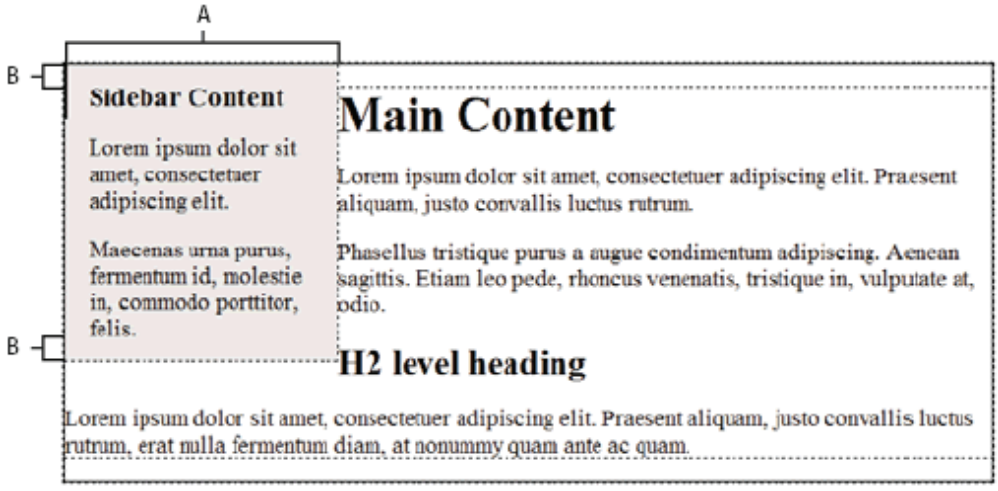


Figure. Sidebar div, float left A. Width 200 pixels B. Top and bottom padding, 15 pixels

Last, the CSS rule for the main container div tag finishes the layout:

```
#mainContent {
        margin: 0 0 0 250px;
        padding: 0 20px 20px 20px;
}
```

The #mainContent rule styles the main content div with a left margin of 250 pixels, which means that it places 250 pixels of space between the left side of the container div, and the left side of the main content div. Additionally, the rule provides for 20 pixels of spacing on the right, bottom, and left sides of the main content div. Figure 4 shows the results of applying the rule to the mainContent div.

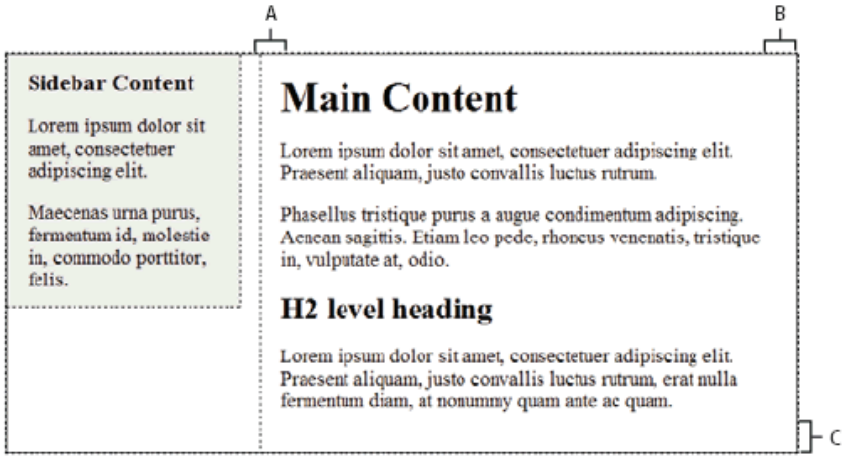


Figure. Main Content div, left margin of 250 pixels A. 20 pixels left padding B. 20 pixels right padding C. 20 pixels bottom padding

The complete code looks as follows:

```
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
<title>Untitled Document</title>
<style type="text/css">
#container {
        width: 780px;
        background: #FFFFFF;
        margin: 0 auto;
        border: 1px solid #000000;
        text-align: left;
}
```

```
#sidebar {
        float: left;
        width: 200px;
        background: #EBEBEB;
        padding: 15px 10px 15px 20px;
}
#mainContent {
        margin: 0 0 0 250px;
        padding: 0 20px 20px 20px;
}
</style>

</head>
<body>
<!--container div tag-->
<div id="container">
        <!--sidebar div tag-->
        <div id="sidebar">
   <h3>Sidebar Content</h3>
   <p>Lorem ipsum dolor sit amet, consectetuer adipiscing elit.</p>
   <p>Maecenas urna purus, fermentum id, molestie in, commodo  porttitor, felis.</p>
        </div>
        <!--mainContent div tag-->
        <div id="mainContent">
   <h1> Main Content </h1>
  <p>Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Praesent aliquam,  justo convallis luctus rutrum.</p>
  <p>Phasellus tristique purus a augue condimentum adipiscing. Aenean  sagittis. Etiam leo pede, rhoncus venenatis, tristique in, vulputate at, odio.</p>
   <h2>H2 level heading </h2>
```

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Praesent aliquam, justo convallis luctus rutrum, erat nulla fermentum diam, at nonummy quam ante ac quam.

</body>

Note: The above example code is a simplified version of the code that creates the two-column fixed left sidebar layout when you create a new document using the predesigned layouts that come with Dreamweaver.

Create a page with a CSS layout

When creating a new page in Dreamweaver, you can create one that already contains a CSS layout. Dreamweaver comes with over 30 different CSS layouts that you can choose from. Additionally, you can create your own CSS layouts and add them to the configuration folder so that they appear as layout choices in the New Document dialog box.

To create a page with a CSS layout:

1. Select File > New.
2. In the New Document dialog box, select the Blank Page category. (It's the default selection.)
3. For Page Type, select the kind of page you want to create.
4. For Layout, select the CSS layout you want to use. You can choose from 16

   Different layouts. The Preview window shows the layout and gives a brief description of the selected layout.

   The predesigned CSS layouts provide the following types of columns:

1. Fixed Column width is specified in pixels. The column does not resize based on the size of the browser or the site visitor's text settings.
2. Liquid Column width is specified as a percentage of the site visitor's browser width. The design adapts if the site visitor makes the browser wider or narrower, but does not change based on the site visitor's text settings.
3. Select a document type from the DocType pop up menu.
4. Select a location for the layout's CSS from the Layout CSS in pop up menu.
5. Add To Head: Adds CSS for the layout to the head of the page you're creating.
6. Create New File: Adds CSS for the layout to a new external CSS stylesheet and attaches the new stylesheet to the page you're creating.
7. Link To Existing File: Lets you specify an existing CSS file that already contains the CSS rules needed for the layout. This option is particularly useful when you want to use the same CSS layout (the CSS rules for which are contained in a single file) across multiple documents.

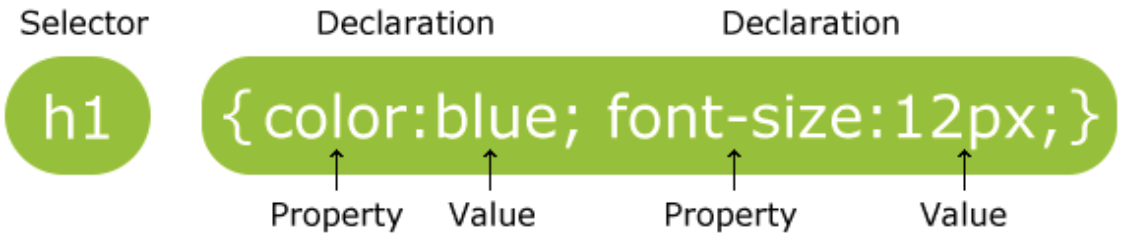
Do one of the following:

1. If you selected Add to Head from the Layout CSS in pop up menu (the default option), click Create.
2. If you selected Create New File from the Layout CSS pop up menu, click Create, and then specify a name for the new external file in the Save Style Sheet File As dialog box.
3. If you selected Link to Existing File from the Layout CSS in pop up menu, add the external file to the Attach CSS file text box by clicking the Add Style Sheet icon, completing the Attach External Style Sheet dialog box, and clicking OK. When you're finished, click Create in the New Document dialog box.
4. When you put the layout CSS in a new file or link to an existing file, Dreamweaver automatically links the file to the HTML page you're creating..

8.5 CSS Syntax and Classes

CSS Syntax

A CSS rule has two main parts: a selector, and one or more declarations:



The selector is normally the HTML element you want to style.

Each declaration consists of a property and a value.

The property is the style attribute you want to change. Each property has a value.

CSS Example

A CSS declaration always ends with a semicolon, and declaration groups are surrounded by curly brackets:

p {color:red;text-align:center;}

To make the CSS more readable, you can put one declaration on each line, like this:

Example

p

{

color:red;

text-align:center;

}

In CSS, classes allow you to apply a style to a given class of an element. To do this, you link the element to the style by declaring a style for the class, then assigning that class to the element.

CSS Class Syntax

You declare a CSS class by using a dot (.) followed by the class name. You make up the class name yourself. After the class name you simply enter the properties/values that you want to assign to your class.

.class-name { property:value; }

If you want to use the same class name for multiple elements, but each with a different style, you can prefix the dot with the HTML element name.

html-element-name.class-name { property:value; }

CSS Class Example

<head>

<style type="text/css">

div.css-section { border:1px dotted red; }

p.css-section { color:green; }

</style>

</head>

<body>

CSS Class

CSS classes can be very useful

</body>

8.6 Introduction to XSL

In computing, the term Extensible Stylesheet Language (XSL) is used to refer to a family of languages used to transform and render XML documents.

Historically, the XSL Working Group in W3C produced a draft specification under the name XSL, which eventually split into three parts:

1. XSL Transformation (XSLT): is an XML language for transforming XML documents
2. XSL Formatting Objects (XSL-FO): an XML language for specifying the visual formatting of an XML document
3. the XML Path Language (XPath): a non-XML language used by XSLT, and also available for use in non-XSLT contexts, for addressing the parts of an XML document.

8.7 Summary

The unit explain about the importance of style sheet and how does style sheet integrate with XML. In the unit we have concepts related to page layout of web page. The unit also provide an abstract method for our coding using the classes in style sheet.

8.8 Self Assessment Question

1. What do you mean by CSS? Define the various types with syntax?
2. How can we create a class and integrate it with our web page?
3. Define the page layout with possible syntaxes?
4. Define XML with style sheet with example?

Unit - 9 : XML Formatting with XSL - FO

Structure of the Unit:

9.0 Objective
9.1 Introduction
9.2 What is XSL - FO
9.3 XSL Document Structure and Area
9.4 Page and Region
9.5 Block, Line and Inline Areas
9.6 List, Tables
9.7 Outputting Formatted Contents, Output and Flow
9.8 Summary
9.9 Self Assessment Question

9.0 Objective

This unit is basically design to create a web page with help of formatted contents. To enable this unit have set of commands and tags belong to different section of our page and their separation.

Unit include the topic -

* XSL- FO
* XSL – FO areas and Structure

9.1 Introduction

Unlike the combination of HTML and CSS, XSL-FO is a unified presentational language. It has no semantic markup in the way it is meant in HTML. And, unlike CSS which modifies the default presentation of an external XML or HTML document, it stores all of the document's data within itself.

9.2 What is XSL - FO

XSL Formatting Objects, or XSL-FO, is a markup language for XML document formatting which is most often used to generate PDFs. XSL-FO is part of XSL (Extensible Stylesheet Language), a set of W3C technologies designed for the transformation and formatting of XML data.

The general idea behind XSL-FO's use is that the user writes a document, not in FO, but in an XML language. XHTML, DocBook, and TEI are all possible examples. Then, the user obtains anXSLT transform, either by writing one themselves or by finding one for the document type in question. This XSLT transform converts the XML into XSL-FO.

Once the XSL-FO document is generated, it is then passed to an application called an FO processor. FO processors convert the XSL-FO document into something that is readable, printable or both. The most common output of XSL-FO is a PDF file or as PS, but some FO processors can output to other formats like RTF files or even just a window in the user's GUI displaying the sequence of pages and their contents.

9.3 XSL Document Structure and Area

XSL-FO documents are XML documents, but they do not have to conform to any DTD or schema. Instead, they conform to a syntax defined in the XSL-FO specification.

XSL-FO documents contain two required sections. The first section details a list of named page layouts. The second section is a list of document data, with markup, that uses the various page layouts to determine how the content fills the various pages.

Page layouts define the properties of the page. They can define the directions for the flow of text, so as to match the conventions for the language in question. They define the size of a page as well as the margins of that page. More importantly, they can define sequences of pages that allow for effects where the odd and even pages look different. For example, one can define a page layout sequence that gives extra space to the inner margins for printing purposes; this allows more space to be given to the margin where the book will be bound.

The document data portion is broken up into a sequence of flows, where each flow is attached to a page layout. The flows contain a list of blocks which, in turn, each contain a list of text data, inline markup elements, or a combination of the two. Content may also be added to the margins of the document, for page numbers, chapter headings and the like.

Blocks and inline elements function in much the same way as for CSS, though some of the rules for padding and margins differ between FO and CSS. The direction, relative to the page orientation, for the progression of blocks and inlines can be fully specified, thus allowing FO documents to function under languages that are read different from English. The language of the FO specification, unlike that of CSS 2.1, uses direction-neutral terms like start and end rather than left and right when describing these directions.

XSL-FO's basic content markup is derived from CSS and its cascading rules. As such, many attributes in XSL-FO propagate into the child elements unless explicitly overridden.

Capabilities of XSL-FO

Lists

An XSL-FO list is, essentially, two sets of blocks stacked side by side. An entry consists of a block on the "left", or start inline direction, and a block sequence on the "right", or end inline direction. The block on the left is conceptually what would be the number or bullet in a list. However, it could just as easily be a string of text, as one might see in a glossary entry. The block on the right works as expected. Both of these blocks can be block containers, or have multiple blocks in a single list entry.

Numbering of XSL-FO lists, when they are numbered, is expected to be done by the XSLT, or whatever other process, that generated the XSL-FO document. As such, number lists are to be explicitly numbered in XSL-FO.

Pagination controls

The user can specify Widow and Orphan for blocks or for the flow itself, and allow the attributes to cascade into child blocks. Additionally, blocks can be specified to be kept together on a single page. For example, an image block and the description of that image can be set to never be separated. The FO

processor will do its best to adhere to these commands, even if it requires creating a great deal of empty space on a page.

Footnotes

The user can create footnotes that appear at the bottom of a page. The footnote is written, in the FO document, in the regular flow of text at the point where it is referenced. The reference is represented as an inline definition, though it is not required. The body is one or more blocks that are placed by the FO processor to the bottom of the page. The FO processor guarantees that wherever the reference is, the footnote cited by that reference will begin on the same page. This will be so even if it means creating extra empty space on a page.

Tables

An FO table functions much like an HTML/CSS table. The user specifies rows of data for each individual cell. The user can, also, specify some styling information for each column, such as background color. Additionally, the user can specify the first row as a table header row, with its own separate styling information.

The FO processor can be told exactly how much space to give each column, or it can be told to auto-fit the text in the table.

XSL-FO Areas

The XSL formatting model defines a number of rectangular areas (boxes) to display output.

All output (text, pictures, etc.) will be formatted into these boxes and then displayed or printed to a target media.

We will take a closer look at the following areas:

* Pages
* Regions
* Block areas
* Line areas
* Inline areas

9.4 Page and Region

XSL-FO uses page templates called "Page Masters" to define the layout of pages.

XSL-FO Page Templates

XSL-FO uses page templates called "Page Masters" to define the layout of pages. Each template must have a unique name:

```
<fo:simple-page-master master-name="intro">
  <fo:region-body margin="5in" />
</fo:simple-page-master>
```

```
<fo:simple-page-master master-name="left">
  <fo:region-body margin-left="2in" margin-right="3in" />
</fo:simple-page-master>
<fo:simple-page-master master-name="right">
  <fo:region-body margin-left="3in" margin-right="2in" />
</fo:simple-page-master>
```

In the example above, three <fo:simple-page-master> elements, define three different templates. Each template (page-master) has a different name.

The first template is called "intro". It could be used as a template for introduction pages.

The second and third templates are called "left" and "right". They could be used as templates for even and odd page numbers.

XSL-FO Page Size

XSL-FO uses the following attributes to define the size of a page:

* page-width defines the width of a page
* page-height defines the height of a page

XSL-FO Page Margins

XSL-FO uses the following attributes to define the margins of a page:

* margin-top defines the top margin
* margin-bottom defines the bottom margin
* margin-left defines the left margin
* margin-right defines the right margin
* margin defines all four margins

XSL-FO Page Regions

XSL-FO uses the following elements to define the regions of a page:

* region-body defines the body region
* region-before defines the top region (header)
* region-after defines the bottom region (footer)
* region-start defines the left region (left sidebar)
* region-end defines the right region (right sidebar)

Note that the region-before, region-after, region-start, and region-end is a part of the body region. To avoid text in the body region to overwrite text in these regions, the body region must have margins at least the size of these regions.

9.5 Block, Line and Inline Areas

Block Area Attributes

Blocks are sequences of output in rectangular boxes:

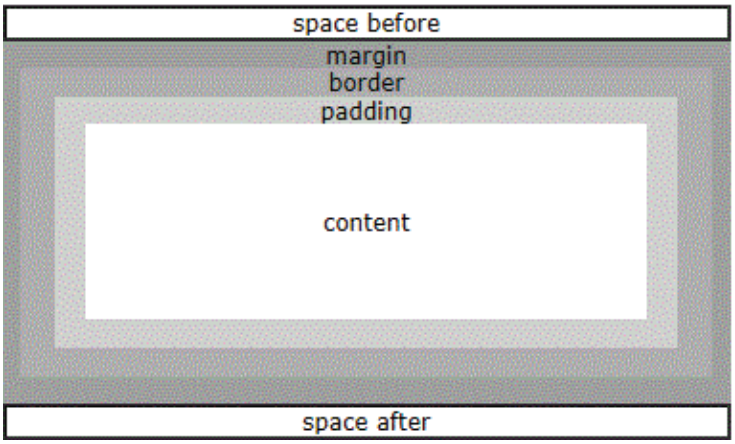
<fo:block border-width="1mm">

This block of output will have a one millimeter border around it.

</fo:block>

Since block areas are rectangular boxes, they share many common area properties:

* space before and space after
* margin
* border
* padding



The **space before** and **space after** is the empty space separating the block from the other blocks.

The **margin** is the empty area on the outside of the block.

The **border** is the rectangle drawn around the external edge of the area. It can have different widths on all four sides. It can also be filled with different colors and background images.

The **padding** is the area between the border and the content area.

The **content** area contains the actual content like text, pictures, graphics, or whatever.

Block Margin

* margin
* margin-top
* margin-bottom
* margin-left
* margin-right

Block Border

Border style attributes:

* border-style
* border-before-style
* border-after-style
* border-start-style
* border-end-style
* border-top-style (same as border-before)
* border-bottom-style (same as border-after)
* border-left-style (same as border-start)
* border-right-style (same as border-end)

Border color attributes:

* border-color
* border-before-color
* border-after-color
* border-start-color
* border-end-color
* border-top-color (same as border-before)
* border-bottom-color (same as border-after)
* border-left-color (same as border-start)
* border-right-color (same as border-end)

Border width attributes:

* border-width
* border-before-width
* border-after-width
* border-start-width
* border-end-width
* border-top-width (same as border-before)
* border-bottom-width (same as border-after)
* border-left-width (same as border-start)
* border-right-width (same as border-end)

Block Padding

* padding
* padding-before
* padding-after
* padding-start
    * padding-end
    * padding-top (same as padding-before)
    * padding-bottom (same as padding-after)
    * padding-left (same as padding-start)
    * padding-right (same as padding-end)

Block Background

    * background-color
    * background-image
    * background-repeat
    * background-attachment (scroll or fixed)

Block Styling Attributes

Blocks are sequences of output that can be styled individually:

```
<fo:block font-size="12pt" font-family="sans-serif">
  This block of output will be written in a 12pt sans-serif font.
</fo:block>
```

Font attributes:

* font-family
* font-weight
* font-style
* font-size
* font-variant

Text attributes:

* text-align
* text-align-last
* text-indent
* start-indent

* end-indent
* wrap-option (defines word wrap)
* break-before (defines page breaks)
* break-after (defines page breaks)

SL-FO Line and Inline areas

Text inside the Block area is defined with Line area. Line areas are generated by the same formatting object that generate their parent. Line areas don't have borders and padding. These areas contain XSL-FO Inline areas.

Inline areas specify attributes for pieces of Lines. inline content is defined via **<fo:inline>** element. To generate an inline reference area use **<fo:inline-content>** element.

The following attribute may be used with <fo:inline>:

* border, padding, margin and background properties
* baseline-shift
* alignment-adjust
* alignment-baseline
* color
* line-height
* text-decoration
* id, etc.

9.6 List, Tables

XSL-FO List Blocks

There are four XSL-FO objects used to create lists:

* fo:list-block (contains the whole list)
* fo:list-item (contains each item in the list)
* fo:list-item-label (contains the label for the list-item - typically an <fo:block> containing a number, character, etc.)
* fo:list-item-body (contains the content/body of the list-item - typically one or more <fo:block> objects)

An XSL-FO list example:

<fo:list-block>

<fo:list-item>

<fo:list-item-label>

```
    <fo:block>*</fo:block>
  </fo:list-item-label>
  <fo:list-item-body>
    <fo:block>Volvo</fo:block>
  </fo:list-item-body>
</fo:list-item>
<fo:list-item>
  <fo:list-item-label>
    <fo:block>*</fo:block>
  </fo:list-item-label>
  <fo:list-item-body>
    <fo:block>Saab</fo:block>
  </fo:list-item-body>
</fo:list-item>
</fo:list-block>
```

XSL-FO Tables

The XSL-FO table model is not very different from the HTML table model.

There are nine XSL-FO objects used to create tables:

* fo:table-and-caption
* fo:table
* fo:table-caption
* fo:table-column
* fo:table-header
* fo:table-footer
* fo:table-body
* fo:table-row
* fo:table-cell

XSL-FO uses the **<fo:table-and-caption>** element to define a table. It contains a **<fo:table>** and an optional **<fo:caption>** element.

The **<fo:table>** element contains optional **<fo:table-column>** elements, an optional **<fo:table-header>** element, a **<fo:table-body>** element, and an optional **<fo:table-footer>** element. Each of these elements has one or more **<fo:table-row>** elements, with one or more **<fo:table-cell>**elements:

```
<fo:table-and-caption>
<fo:table>
<fo:table-column column-width="25mm"/>
<fo:table-column column-width="25mm"/>
<fo:table-header>
  <fo:table-row>
    <fo:table-cell>
      <fo:block font-weight="bold">Car</fo:block>
    </fo:table-cell>
    <fo:table-cell>
      <fo:block font-weight="bold">Price</fo:block>
    </fo:table-cell>
  </fo:table-row>
</fo:table-header>
<fo:table-body>
  <fo:table-row>
    <fo:table-cell>
      <fo:block>Volvo</fo:block>
    </fo:table-cell>
    <fo:table-cell>
      <fo:block>$50000</fo:block>
    </fo:table-cell>
  </fo:table-row>
  <fo:table-row>
    <fo:table-cell>
      <fo:block>SAAB</fo:block>
    </fo:table-cell>
    <fo:table-cell>
      <fo:block>$48000</fo:block>
    </fo:table-cell>
  </fo:table-row>
```

```
</fo:table-body>
</fo:table>
</fo:table-and-caption>
```

9.7 Outputting Formatted Contents, Output and Flow

XSL-FO Page, Flow, and Block

"Blocks" of content "Flows" into "Pages" and then to the output media.

XSL-FO output is normally nested inside <fo:block> elements, nested inside <fo:flow> elements, nested inside <fo:page-sequence> elements:

```
<fo:page-sequence>
  <fo:flow flow-name="xsl-region-body">
    <fo:block>
      <!-- Output goes here -->
    </fo:block>
  </fo:flow>
</fo:page-sequence>
```

XSL-FO Example

It is time to look at a real XSL-FO example:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
<fo:layout-master-set>
  <fo:simple-page-master master-name="A4">
    <fo:region-body />
  </fo:simple-page-master>
</fo:layout-master-set>
<fo:page-sequence master-reference="A4">
  <fo:flow flow-name="xsl-region-body">
    <fo:block>Hello W3Schools</fo:block>
  </fo:flow>
</fo:page-sequence>
</fo:root>
```

XSL-FO Page Sequences

XSL-FO uses <fo:page-sequence> elements to define **output pages.**

Each **output page** refers to a page master which defines **the layout.**

Each **output page** has a <fo:flow> element defining **the output.**

Each **output page** is printed (or displayed) **in sequence.**

XSL-FO Flow

XSL-FO pages are filled with content from the <fo:flow> element.

The <fo:flow> element contains all the elements to be printed to the page.

When the page is full, the same page master will be used over (and over) again until all the text is printed.

Where To Flow?

The <fo:flow> element has a "flow-name" attribute.

The value of the flow-name attribute defines where the content of the <fo:flow> element will go.

The legal values are:

* xsl-region-body (into the region-body)
* xsl-region-before (into the region-before)
* xsl-region-after (into the region-after)
* xsl-region-start (into the region-start)
* xsl-region-end (into the region-end)

9.8 Summary

The unit is about to design formatted page. In the unit we have various tags related to set the margin of page, design block, list and tabular data and also the unit have formats to set the flow of formatted data.

9.9 Self Assessment Question

1. What are the attributes to set the margin of page?
2. Define the differences and similarities between XSL-FO List and Table?
3. Define the page and region areas of XSL-FO?
4. Define the XSL-FO document structure?

Unit - 10 : XSLT Transformation with XSLT

Structure of the Unit:

10.0 Objective
10.1 Introduction
10.2 What is XSLT
10.3 XSLT Template
10.4 Details of XSLT Notation
10.5 Value of Tag
10.6 Choosing Specific Element
10.7 Condition Statements
10.8 Sorting
10.9 Xquery in XSLT
10.10 Summary
10.11 Self Assessment Question

10.0 Objective

The unit cover topic related to transformation of XML in various fields. The unit also provide description about the tags to access data, perform selection and choose appropriate data.

10.1 Introduction

Typically, input documents are XML files, but anything from which the processor can build an XQuery and XPath Data Model can be used, for example relational database tables, or geographical information systems.

10.2 What is XSLT

XSLT (Extensible Stylesheet Language Transformations) is a language for transforming XML documents into other XML documents, or other objects such as HTML for web pages, plain text or into XSL Formatting Objects which can then be converted to PDF, PostScript and PNG.

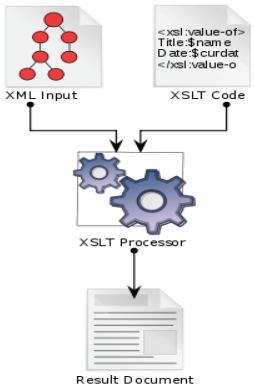


Diagram of the basic elements and process flow of Extensible Stylesheet Language Transformations.

The XSLT processor takes one or more XML sources, plus one or more XSLT stylesheet modules, and

processes them with the XSLT template-processing engine (the processor) to produce an output document. The XSLT stylesheet contains a collection of template rules: instructions and other directives that guide the processor in the production of the output document.

10.3 XSLT Template

Template rule processing

The XSLT language is declarative: rather than listing an imperative sequence of actions to perform in a stateful environment, template rules only define how to handle a node matching a particular XPath-like pattern, if the processor should happen to encounter one, and the contents of the templates effectively comprise functional expressions that directly represent their evaluated form: the result tree, which is the basis of the processor's output.

The processor follows a fixed algorithm: assuming a stylesheet has already been read and prepared, the processor builds a source tree from the input XML document. It then starts by processing the source tree's root node, finding in the stylesheet the best-matching template for that node, and evaluating the template's contents. Instructions in each template generally direct the processor either to create nodes in the result tree, or to process more nodes in the source tree in the same way as the root node. Output derives from the result tree.

The purpose of XSLT is to help transform an XML document into something new. To transform an XML document, XSLT must be able to do two things well:

1. Find information in the XML document.
2. Add additional text and/or data. In a previous example, we added HTML tags.

Both of these items are taken care of with the very important XSL elementxsl:template.

Template

To find information in an XML document you will need to use xsl:template'smatch attribute. It is in this attribute that you use your knowledge of XPath to find information in your XML document.

We will be using class.xml as our example XML document.

XML Code:

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

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

<class>

    <student>Jack</student>

    <student>Harry</student>

    <student>Rebecca</student>

    <teacher>Mr. Bean</teacher>

</class>
```

If we wanted to find student elements, we would set the match attribute to a simple XPath expression: student

Every time we find a student, let's print something out. The text we want printed must go between the opening and closing tags of <xsl:template>. Let's have it print out, "Found a learner!"

The following XSLT code will find student elements in the XML and output, "Found a learner!" for each student element. This example displays both thefinding and the adding text functionality of XSLT!

10.4 Details of XSLT Notation

[DEFINITION: An XSLT element is an element in the XSLT namespace whose syntax and semantics are defined in this specification.] For a non-normative list of XSLT elements.

In this document the specification of each XSLT element is preceded by a summary of its syntax in the form of a model for elements of that element type. The meaning of syntax summary notation is as follows:

* An attribute that is REQUIRED is shown with its name in bold. An attribute that may be omitted is shown with a question mark following its name.
* An attribute that is deprecated is shown in a grayed font within square brackets.
* The string that occurs in the place of an attribute value specifies the allowed values of the attribute. If this is surrounded by curly brackets ({...}), then the attribute value is treated as an attribute value template, and the string occurring within curly brackets specifies the allowed values of the result of evaluating the attribute value template. Alternative allowed values are separated by |. A quoted string indicates a value equal to that specific string. An unquoted, italicized name specifies a particular type of value.

  In all cases where this specification states that the value of an attribute MUST be one of a limited set of values, leading and trailing whitespace in the attribute value is ignored. In the case of an attribute value template, this applies to the effective value obtained when the attribute value template is expanded.
* Unless the element is REQUIRED to be empty, the model element contains a comment specifying the allowed content. The allowed content is specified in a similar way to an element type declaration in XML; sequence constructor means that any mixture of text nodes, literal result elements, extension instructions, and XSLT elements from the instruction category is allowed; other-declarations means that any mixture of XSLT elements from the declaration category, other than xsl:import, is allowed, together with user-defined data elements.
* The element is prefaced by comments indicating if it belongs to the instruction category or declaration category or both. The category of an element only affects whether it is allowed in the content of elements that allow a sequence constructor or other-declarations.

Example: Syntax Notation

This example illustrates the notation used to describe XSLT elements.

<!-- Category: instruction -->

<xsl:example-element

select = expression

debug? = { "yes" | "no" }>

<!-- Content: ((xsl:variable | xsl:param)*, xsl:sequence) -->

</xsl:example-element>

This example defines a (non-existent) element xsl:example-element. The element is classified as an instruction. It takes a mandatory select attribute, whose value is an XPathexpression, and an optional debug attribute, whose value MUST be either yes or no; the curly brackets indicate that the value can be defined as an attribute value template, allowing a value such as debug="{$debug}", where the variable debug is evaluated to yield "yes" or "no" at run-time.

The content of an xsl:example-element instruction is defined to be a sequence of zero or more xsl:variable and xsl:param elements, followed by an xsl:sequence element.

[ERR XTSE0010] A static error is signaled if an XSLT-defined element is used in a context where it is not permitted, if a REQUIRED attribute is omitted, or if the content of the element does not correspond to the content that is allowed for the element.

Attributes are validated as follows. These rules apply to the value of the attribute after removing leading and trailing whitespace.

* [ERR XTSE0020] It is a static error if an attribute (other than an attribute written using curly brackets in a position where an attribute value template is permitted) contains a value that is not one of the permitted values for that attribute.
* [ERR XTDE0030] It is a non-recoverable dynamic error if the effective value of an attribute written using curly brackets, in a position where an attribute value template is permitted, is a value that is not one of the permitted values for that attribute. If the processor is able to detect the error statically (for example, when any XPath expressions within the curly brackets can be evaluated statically), then the processor may optionally signal this as a static error.

10.5 Value of Tag

The <xsl:value-of> element can be used to extract the value of an XML element and add it to the output stream of the transformation:

Example

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

<xsl:stylesheet version="1.0"

xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">

<html>

```
<body>
<h2>My CD Collection</h2>
<table border="1">
  <tr bgcolor="#9acd32">
    <th>Title</th>
    <th>Artist</th>
  </tr>
  <tr>
    <td><xsl:value-of select="catalog/cd/title"/></td>
    <td><xsl:value-of select="catalog/cd/artist"/></td>
  </tr>
</table>
</body>
</html>
</xsl:template>
```

10.6 Choosing Specific Element

The XSL <xsl:for-each> element can be used to select every XML element of a specified node-set:

Example

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
  <html>
  <body>
  <h2>My CD Collection</h2>
  <table border="1">
    <tr bgcolor="#9acd32">
      <th>Title</th>
      <th>Artist</th>
    </tr>
```

```
<xsl:for-each select="catalog/cd">
<tr>
<td><xsl:value-of select="title"/></td>
<td><xsl:value-of select="artist"/></td>
</tr>
</xsl:for-each>
</table>
</body>
</html>
</xsl:template>
</xsl:stylesheet>
```

Filtering the Output

We can also filter the output from the XML file by adding a criterion to the select attribute in the <xsl:for-each> element.

<xsl:for-each select="catalog/cd[artist='Bob Dylan']">

Legal filter operators are:

* = (equal)
* != (not equal)
* < less than
* > greater than

10.7 Condition Statements

To put a conditional if test against the content of the XML file, add an <xsl:if> element to the XSL document.

Syntax

```
<xsl:if test="expression">
...some output if the expression is true...
</xsl:if>
```

Where to Put the <xsl:if> Element

To add a conditional test, add the <xsl:if> element inside the <xsl:for-each> element in the XSL file:

Example

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

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
  <html>
  <body>
  <h2>My CD Collection</h2>
  <table border="1">
    <tr bgcolor="#9acd32">
      <th>Title</th>
      <th>Artist</th>
    </tr>
    <xsl:for-each select="catalog/cd">
      <xsl:if test="price &gt; 10">
        <tr>
          <td><xsl:value-of select="title"/></td>
          <td><xsl:value-of select="artist"/></td>
        </tr>
      </xsl:if>
    </xsl:for-each>
  </table>
  </body>
  </html>
</xsl:template>
</xsl:stylesheet>
```

The <xsl:choose> element is used in conjunction with <xsl:when> and <xsl:otherwise> to express multiple conditional tests.

The <xsl:choose> Element

Syntax

```
<xsl:choose>
  <xsl:when test="expression">
    ... some output ...
```

```
  </xsl:when>
  <xsl:otherwise>
    ... some output ....
  </xsl:otherwise>
</xsl:choose>
```

Where to put the Choose Condition

To insert a multiple conditional test against the XML file, add the <xsl:choose>, <xsl:when>, and <xsl:otherwise> elements to the XSL file:

Example

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
  <html>
  <body>
  <h2>My CD Collection</h2>
  <table border="1">
    <tr bgcolor="#9acd32">
      <th>Title</th>
      <th>Artist</th>
    </tr>
    <xsl:for-each select="catalog/cd">
    <tr>
      <td><xsl:value-of select="title"/></td>
      <xsl:choose>
        <xsl:when test="price &gt; 10">
          <td bgcolor="#ff00ff">
          <xsl:value-of select="artist"/></td>
        </xsl:when>
        <xsl:otherwise>
          <td><xsl:value-of select="artist"/></td>
```

```
      </xsl:otherwise>
    </xsl:choose>
  </tr>
  </xsl:for-each>
 </table>
 </body>
 </html>
</xsl:template>
</xsl:stylesheet>
```

10.8 Sorting

The <xsl:sort> element is used to sort the output.

Note: <xsl:sort> is always within <xsl:for-each> or <xsl:apply-templates>.

Syntax

```
<xsl:sort select="expression"
lang="language-code"
data-type="text|number|qname"
order="ascending|descending"
case-order="upper-first|lower-first"/>
```

Attributes

Attribute	Value	Description
Select	XPath-expression	Optional. Specifies which node/node-set to sort on
Lang	language-code	Optional. Specifies which language is to be used by the sort
data-type	text	Optional. Specifies the data-type of the data to be sorted. De fault numberis "text" qname
Order	ascending	Optional. Specifies the sort order. Default is "ascend ing"
	descending	
case-order	upper-first	Optional. Specifies whether upper- or lowercase let ters are to be lower-first ordered first

10.9 Xquery in XSLT

XML makes it possible to store content in a standardized format that can be converted to a variety of output media using a broad choice of technologies. Most of these technologies build on related W3C standards with both commercial and open source tools support. XSLT lets you define a transformation

of a set of documents into a particular format, so that three XSLT stylesheets could create published products from the same content for three different media. The XQuery language lets you pull subsets of XML content from huge reositories, so that XML data bases that sup port XQuery can (among other things) provide dynamic publications custom ized for different customers.

The classes in this course will show you what you need to put XSLT and XQuery to work, as we look at effi cient and effective development practices, how to write test-driven XSLT applications, and where XSLT, XQuery, and related technologies best fit into the application architecture of a larger system.

Because the "Hands-on Introduction to XML" course will provide intro ductory material on XSLT and XQuery, classes in this course will focus on help ing exist ing XSLT and/or XQuery developers get the most out of their code and their development time.

Query ing XML Data bases with XQuery

This class will provide an overview of the capabilities and use cases of XML data bases, examining some of the data base products that support XML and how they are being used. It will then cover the role of XQuery among other XML technologies in the querying of XML databases.

As a group, the class will build a simple search application using XQuery and an XML database (eXist). This will provide an opportunity for attendees to learn the syntax and capabilities of XQuery, as well as see it in action. Major features of the XQuery language such as FLWOR expressions, XMLconstructors, and user-defined functions will be explained.

Developing and Testing in XSLT

Unit tests, profiling, debugging and, increasingly, test-driven development are part of the bread and butter of working with other programming languages but are not always so with XSLT or XQuery. In test-driven devel opment, which is a fundamental part of agile approaches to soft ware development, the developers write tests that describe the desired behaviour of their application, then write code that meets the tests. This style of devel opment keeps code focused, avoids breaking existing code and facilitates refactoring.

In this session, Jeni Tennison and Tony Graham will describe both the state of the art in testing and debugging XSLT and XQuery and how test-driven development applies to XSLT and XQuery develop-ment. In particu lar, they will focus on the use of the XSpec testing framework.

Trends in XSLT/XQuery

XSLT 3.0 and XQuery 3.0 are going to come with a lot of new, powerful features. Expected to be released by the end of the year (XSLT morelikely in 2013), this version 3.0 will bring those new tools to the XML developers.

XPath 3.0 itself comes with one of the most useful of them: function items and higher-order functions; that is, the ability to manipulate functions and pass them around, calling them dynamically and define new anonymous inline functions. XQuery 3.0 has numerous additions on its own, among them Florent will introduce grouping, windowing, try/catch, and private functions. For XSLT 3.0, the biggest addi-tion is the suport for streaming transformation; Florent will also introduce the packages, the evaluation of dynamically computed XPath expres sions, the extension of template rules to atomic val ues, the try/catch mechanism and the new xsl:assert instruc tion. As the final specifications have not been released yet, those features might still change, even though some are very stable now. This introduction will give

you an over view of the new fea tures that you will be able to use soon. Processors even implement some of them already!

The class will also cover XProc, the W3C XML Pipeline language for describing operations to be per formed on XML documents. Released in May 2010, XProc was the missing piece in the XML stack. Every time you used more than one XML technology, you had to use another language, like Java or .NET, in order to glue them together. You then had to know the APIs very well and to be careful in connected all the pieces together, again and again. XProc allows you to describe your processing as a network of steps,

10.10 Summary

To perform XSLT we have various elements related to various requirements. To perform selection we have conditional statements and their tags to process user's requirements. If we want to select and also access data from elements then we have value of tag.

10.11 Self Assessment Question

1. Define the template in XSLT?
2. How can we use the conditional statements in XSLT?
3. How can we access data from an elements? Define with an example?
4. Define the purpose of sorting with its syntax and attributes?

Unit - 11 : XLink and XPointer

Structure of the Unit:

11.0 Objective
11.1 Introduction
11.2 Linking in XML
11.3 XLInking and XPointer Syntax
11.4 Summary
11.5 Self Assessment Question

11.0 Objective

The unit about to explain the use of XLinking in our XML page also this unit provide the various syntax related to XPointers.

11.1 Introduction

XLink defines a standard way of creating hyperlinks in XML documents.

XPointer allows the hyperlinks to point to more specific parts (fragments) in the XML document.

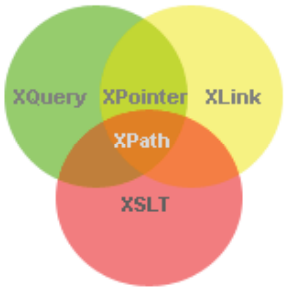
11.2 Linking in XML

Before you continue you should have a basic understanding of the following:

* HTML / XHTML
* XML / XML Namespaces
* XPath

If you want to study these subjects first, find the tutorials on our Home page.

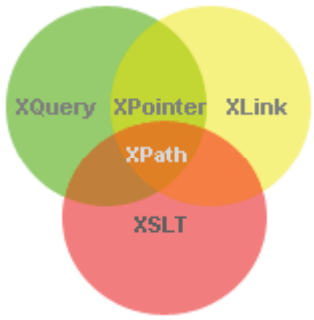
What is XLink?



* XLink is short for XML Linking Language
* XLink is used to create hyperlinks in XML documents
* Any element in an XML document can behave as a link
* XLink supports simple links (like HTML) and extended links (for linking multiple re sources together)

* With XLink, the links can be defined outside the linked files
* XLink is a W3C Recommendation

What is XPointer?



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

11.3 XLInking and XPointer Syntax

XLink Syntax

In HTML, we know that the <a> element defines a hyperlink. However, this is not how it works with XML. In XML documents, you can use whatever element names you want - therefore it is impossible for browsers to predict what hyperlink elements will be called in XML documents.

The solution for creating links in XML documents was to put a marker on elements that should act as hyperlinks.

Below is a simple example of how to use XLink to create links in an XML document:

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

<homepages xmlns:xlink="http://www.w3.org/1999/xlink">

<homepage xlink:type="simple"

xlink:href="http://www.w3schools.com">Visit W3Schools</homepage>

<homepage xlink:type="simple"

xlink:href="http://www.w3.org">Visit W3C</homepage>

</homepages>
```

To get access to the XLink attributes and features we must declare the XLink namespace at the top of the document.

The xlink:type and the xlink:href attributes in the <homepage> elements define that the type and href attributes come from the xlink namespace.

The xlink:type="simple" creates a simple, two-ended link (means "click from here to go there"). We will look at multi-ended (multidirectional) links later.

XPointer Syntax

In HTML, we can create a hyperlink that either points to an HTML page or to a bookmark inside an HTML page (using #).

Sometimes it is more useful to point to more specific content. For example, let's say that we want to link to the third item in a particular list, or to the second sentence of the fifth paragraph. This is easy with XPointer.

If the hyperlink points to an XML document, we can add an XPointer part after the URL in the xlink:href attribute, to navigate (with an XPath expression) to a specific place in the document.

11.4 Summary

In this module we have representation to understand how to link pages and sections of pages for communication of data. The unit also have xpointer to control the navigation of data as data indicator.

11.5 Self Assessment Question

1. What do you mean by XLink? Define with the help of an example?
2. Define the differences between XLink and XPointer?

Unit - 12 : XForms

Structure of the Unit:

12.0 Objective
12.1 Introduction
12.2 Why XForm
12.3 XForm Structure and Syntax
12.4 Selecting and Controlling Input Xform
12.5 Summary
12.6 Self Assessment Question

12.0 Objective

The unit is about to design a form with the help of XML and also define how to perform selection and input of data using XForm.

12.1 Introduction

XFA (also known as XFA forms) stands for XML Forms Architecture, a family of proprietary XML specifications that was suggested and developed by JetForm to enhance the processing of web forms. It can be also used in PDF files starting with PDF 1.5 specification. XML Forms Architecture was not standardized as an ISO standard

12.2 Why XForm

XFA's main extension to XML are computationally active tags. In addition, all instances created from a given XFA form template keep the specification of data capture, rendering, and manipulation rules from the original. Another major advantage of XFA is that its data format allows compatibility with other systems, and with changes to other technology, applications and technology standards.

According to JetForm's submission to the World Wide Web Consortium, "XFA addresses the needs of organizations to securely capture, present, move, process, output and print information associated with electronic forms.

XForms provides specific benefits when used on mobile devices:

* User interfaces using XForms require fewer round trips with the server and are in that sense more self contained than user interfaces using HTML 4 forms.

* Capabilities of mobile devices vary greatly; consequently the amount of the work involved in generating different user interfaces for different devices is of particular concern in the mobile world. XForms has been designed from the ground up to allow forms to be described inde pendently of the device, which reduces the amount of work required to target multiple devices.

* XForms reduces the need for JavaScript, which is particularly interesting as JavaScript support varies greatly on mobile devices and cannot be widely relied upon. This also allows systems on which JavaScript is disabled for security concerns to continue to operate flawlessly.

Differences from HTML forms

In contrast to the original HTML forms, the creators of XForms have used a model–view–controller approach. The model consists of one or more XForms models describing form data, constraints upon that data, and submissions. The view describes what controls appear in the form, how they are grouped together, and what data they are bound to. CSS can be used to describe a form's appearance.

An XForms document can be as simple as an HTML form (by only specifying the submission element in the model section, and placing the controls in the body), but XForms includes many advanced features. For example, new data can be requested and used to update the form while it is running, much like using XMLHttpRequest/AJAX except without scripting. The form author can validate user data against XML Schema data types, require certain data, disable input controls or change sections of the form depending on circumstances, enforce particular relationships between data, input variable length arrays of data, output calculated values derived from form data, prefill entries using an XML document, respond to actions in real time (versus at submission time), and modify the style of each control depending on the device they are displayed on (browser versus mobile versus text only, etc.). There is often no need for any scripting with languages such as JavaScript.

12.3 XForm Structure and Syntax

XForms are an application of XML [XML 1.0], and have been designed for use within other XML vocabularies, in particular XHTML [XHTML 1.0]. This chapter discusses some of the high-level features of XForms that allow this specification to be used with other document types.

The XForms Namespace

The XForms namespace has the URI: http://www.w3.org/2001/06/xforms. Future revisions are expected to use a different identifier. This document uses the convention of an xform: prefix to represent elements and attributes that are part of the XForms Namespace.

XForms Processors must use the XML namespaces mechanism [XML Names] to recognize elements and attributes from this namespace. Except where specifically allowed by the Schema for XForms, foreign-namespaced elements are not allowed as content of elements in the XForms namespace. Foreign-namespaced attributes are, however, allowed on any XForms element. The XForms Processor must ignore any foreign-namespaced elements or attributes that are unrecognized.

XForms Elements

xform

The xform element is used as a container for other XForms elements, and can serve as the root element of a standalone document or be embedded in other document types such as XHTML. A single containing document may contain any number of xform elements.

Editorial note

under discussion are XLink attributes on the xform element. These are: xlink:type="extended" and xlink:role="http://www.w3.org/2001/06/xforms" - and they should be defaulted or even fixed in the Schema/DTD.

xmlns = namespace-identifier - Optional standard XML attribute for identifying an XML namespace. It is often useful to include this standard attribute at this point.

id = xsd:ID - Optional unique identifier used to refer to this particular xform element.

model

The model element is used to define the XForms Model. The content of the XForms Model may be defined inline or obtained from a external URI.

id = xsd:ID - Optional unique identifier.

xlink:href = xsd:anyURI - Optional link to an externally defined XForms Model.

Editorial note

As above, we need to find a place to discuss the defaulted attributes. Here they are xlink:role="http://www.w3.org/2001/06/xforms-model" xlink:type="locator"

instance

The instance element is used to define initial instance data. The instance data may be defined inline or obtained from a external URI.

id = xsd:ID - Optional unique identifier.

xlink:href = xsd:anyURI - Optional link to externally defined instance data

The content of the instance element is arbitrary XML in any namespace other than the XForms namespace. Authors must ensure that proper namespace declarations are used for content within theinstance element.

Editorial note

As above, we need to find a place to discuss the defaulted attributes. Here they are xlink:role="http://www.w3.org/2001/06/xforms-instance" xlink:type="locator"

Issue (issue-schemalocation):

Should a schemaLocation attribute, linking the instance data with a schema definition, be present here?

submitInfo

The submitInfo element provides information on how and where to submit the instance data.

id = xsd:ID - Optional unique identifier.

xlink:href = xsd:anyURI - Required destination for submitted instance data.

method = xsd:string - Optional indicator to provide details on the submit protocol. With HTTP, the default is "POST".

Issue (submit-method-values):

The possible values for method, and their respective meanings, still need to be defined.

bind

The bind element represents a connection between the different parts of XForms.

id = xsd:ID - Required unique identifier.

ref = XForms binding expression - A link to an externally defined XForms Model.

12.4 Selecting and Controlling Input Xform

xforms:select

Sets the choices that are displayed by a checkgroup or list when the user can select one or more of the choices. When the form is processed, an individual check or cell item is automatically generated to represent each choice.

Syntax

The xforms:select option has two different syntaxes, depending on whether the choices are included in the option itself, or whether the choices are included in the data model and linked by the option.

Table . xforms select parameters

Parameter	Type	Description
single node binding	string	see Single node binding.
Style	string	sets to one of the following values: * full for a checkgroup * compact for a list item Default: compact.
label text	string	sets the text for a label that is displayed at the top of thecheckgroup. Leave this blank to display no label. If the item also has a label option, it will override this setting.
label for choice	string	sets the text that is displayed for the choice.
value for choice	string	sets the value that is stored if the user selects this choice.
XFDL Options	XFDL	optionsadds specific XFDL options to the item represented by the choice. For example, you might want to add a type option to the choices in a list, so that the cells that are generated by those choices trigger actions.

xforms:switch

Divides a portion of the form into sets of items, and then controls which set is shown to the user. For example, you may have a form page with Basic and Advanced settings, and may only want to show one type of settings to the user at any given time.

The switch option uses the xforms:switch element to group the items into sets. Each set can contain any number of XFDL items, which are written normally as children of the xforms:switch.

To change which set of items is displayed, you must use the xforms:toggle action.

Syntax

```
<xforms:switch id="name" single_node_binding xfdl:state="state">
  <xforms:case1 id="name" selected="boolean"
    ...XFDL items...
  </xforms:case1>
  ...
  <xforms:casen id="name" selected="boolean"
    ...XFDL items...
  </xforms:casen>
</xforms:switch>
```

xforms:input

Links a field, combobox or check box to an element in the data model so that they share data. However, the xforms:input only support a single line of data.

For example, if you added an xforms:input option to a field in your form, you could use that option to link the field to a name element in your data model. Once linked, any changes made to the data in one would be reflected by the other.

This option is only available if you are using an XForms data model.

Syntax

```
<xforms:input single_node_binding>
  <xforms:label>label text</xforms:label>
  Alert Setting
  Hint Setting
  Help Setting
</xforms:input>
```

Table. xforms input parameters

Parameter	Type	Description
single node binding	string	See Single node binding.
label text	string	Sets the text for the item's built-in label, as well as the accessibility message for the item. Although the xforms:label tag must appear, you can use an empty string if you do not want to set the label.

		If the item also has a label or acclabel option, they will override this setting.
Alert, Hint, Help Setting	metadata	see Metadata sub-options.

Available in

check, combobox, field, custom

Example

The following code shows an XForms model that contains a name, age, and birth date element:

```
<xformsmodels>
  <xforms:model>
    <xforms:instance id="test">
      <testmodel>
        <name></name>
        <age></age>
        <birthdate></birthdate>
      </testmodel>
    </xforms:instance>
  </xforms:model>
</xformsmodels>
```

Using that data model, the following code links a field to the name element in the data model, so that they share data:

```
<field sid="nameField">
  <xforms:input ref="name">
    <xforms:label>Name:</xforms:label>
  </xforms:input>
</field>
```

Once you have a basic field, you can add a help message to it. In the following example, the <xforms:hint> element is used to provide some simple help for the user:

```
<field sid="nameField">
  <xforms:input ref="name">
    <xforms:label>Name:</xforms:label>
    <xforms:hint>Enter your full name.</xforms:hint>
```

```
    </xforms:input>
  </field>
```

12.5 Summary

In the module Xform tag is use with different attributes like select, input and many more to perform the application of insertion and selection of data.

12.6 Self Assessment Question

1. Define the document structure of XForm with suitable syntaxes?
2. How to insert data in Xform? Define with an example?
3. Define the various syntaxes to perform the selection of data in XForm?

Unit - 13 : Applying XML

Structure of the Unit:

13.0 Objective
13.1 Introduction
13.2 XML and Web Service
13.3 XML and HTML
13.4 XML and eCommerce
13.5 ebXML and SOAP
13.6 XML Database
13.7 Storing Binary Data in XML
13.8 Summary
13.9 Self Assessment Question

13.0 Objective

The unit provide detail about how to create a web service using the XML and current areas for XML in commercial fields. Unit also provide the concepts to perform data manipulation using XML.

13.1 Introduction

XML Web services are the fundamental building blocks in the move to distributed computing on the Internet. Open standards and the focus on communication and collaboration among people and applications have created an environment where XML Web services are becoming the platform for application integration. Applications are constructed using multiple XML Web services from various sources that work together regardless of where they reside or how they were implemented.

13.2 XML and Web Service

There are probably as many definitions of XML Web Service as there are companies building them, but almost all definitions have these things in common:

- XML Web Services expose useful functionality to Web users through a standard Web protocol. In most cases, the protocol used is SOAP.
- XML Web services provide a way to describe their interfaces in enough detail to allow a user to build a client application to talk to them. This description is usually provided in an XML document called a Web Services Description Language (WSDL) document.
- XML Web services are registered so that potential users can find them easily. This is done with Universal Discovery Description and Integration (UDDI).

I'll cover all three of these technologies in this article but first I want to explain why you should care about XML Web services.

One of the primary advantages of the XML Web services architecture is that it allows programs written in different languages on different platforms to communicate with each other in a standards-based way. The other significant advantage that XML Web services have over previous efforts is that they work with standard Web protocols—XML, HTTP and TCP/IP. A significant number of companies already have a

Web infrastructure, and people with knowledge and experience in maintaining it, so again, the cost of entry for XML Web services is significantly less than previous technologies.

We've defined an XML Web service as a software service exposed on the Web through SOAP, described with a WSDL file and registered in UDDI. The next logical question is. "What can I do with XML Web services?" The first XML Web services tended to be information sources that you could easily incorporate into applications—stock quotes, weather forecasts, sports scores etc. It's easy to imagine a whole class of applications that could be built to analyze and aggregate the information you care about and present it to you in a variety of ways; for example, you might have a Microsoft® Excel spreadsheet that summarizes your whole financial picture—stocks, 401K, bank accounts, loans, etc. If this information is available through XML Web services, Excel can update it continuously. Some of this information will be free and some might require a subscription to the service. Most of this information is available now on the Web, but XML Web services will make programmatic access to it easier and more reliable.

Exposing existing applications as XML Web services will allow users to build new, more powerful applications that use XML Web services as building blocks. For example, a user might develop a purchasing application to automatically obtain price information from a variety of vendors, allow the user to select a vendor, submit the order and then track the shipment until it is received. The vendor application, in addition to exposing its services on the Web, might in turn use XML Web services to check the customer's credit, charge the customer's account and set up the shipment with a shipping company.

In the future, some of the most interesting XML Web services will support applications that use the Web to do things that can't be done today. For example, one of the services that XML Web Services would make possible is a calendar service. If your dentist and mechanic exposed their calendars through this XML Web service, you could schedule appointments with them on line or they could schedule appointments for cleaning and routine maintenance directly in your calendar if you like. With a little imagination, you can envision hundreds of applications that can be built once you have the ability to program the Web.

13.3 XML and HTML

XML and HTML are both designed for different purposes. Although they have some similarities in markup syntax but they are created for different types of goals. XML is not at all the replacement of HTML but is complement to html.

Here is the list of comparison between XML and HTML given below:

1. XML is designed to carry the data while html is to display the data and focus on how the data looks.
2. XML does not do anything but it is used to structure the data, store the data and transport the data while HTML is used to display the data only.
3. XML is cross platform, hardware and software independent tool to carry data from one source to another destination.
4. XML is self descriptive. The DTD or schema describes what and how to use tags and elements in an xml document.
5. XML does not have predefined tags while HTML has. XML lets you invent your own tags while html gives you predefined tags and you have to use them only.

6. XML is extensible as you can add your own tags to extend the xml data.
7. XML is a complement to HTML not the replacement. Most likely use of xml is to transfer the data between incompatible systems as xml is cross platform. XML has now established a strong foundation in the world of web development.
8. XML can separate the data from html. While working on html, instead of storing data in html you can store the data in a separate xml file to make the html code cleaner and easier. Now you can concentrate on working in html rather than storing data. It also eliminates the need to change in html when xml data is changed.
9. XML tags are case sensitive while html tags are not.
10. Attribute values must always be quoted in xml while its not the case with html.

13.4 XML and eCommerce

- Internet first gave people easy access to information
    - e.g., share prices
- then enabled commerce between people and businesses
    - e.g., trading shares using on-line stockbroker
- finally it will enable *integrated e-commerce* through application programs exchanging information
    - trading shares from within a personal finance application running on a home computer
- XML is key enabler of integrated e-commerce

The Significance of XML

- XML seen as a *universal*, *open*, *readable* representation for *software integration* and *data exchange*
- IBM, Microsoft, Oracle and Sun have built XML into database, authoring, and publishing products
- "XML will be the lingua franca of the web," Steve Ballmer (CEO, Microsoft) at *ACM1: Beyond Cyberspace* conference (March 2001)
- XML is central to Microsoft's .NET strategy (software as a service)
- "This will be a much bigger deal" than Java, Ballmer added

Electronic Data Interchange (EDI)

- *Electronic Data Interchange* (*EDI*) used by companies to integrate applications with those of their trading partners
- EDI has been used for over 20 years
- so e-commerce is not a new phenomenon
- usage of EDI limited to large corporations because of high costs and complexity
- advent of the Internet and availability of free or cheap software

- affordable way for customers and small-to-medium enterprises to exchange information electronically

E-Commerce Paths to Profitability

From Products to Services

- software as a service to which customers can subscribe (e.g., .NET)
- music recording industry being forced to offer subscription-based services in reaction to peer-to-peer music sharing systems
- using loyalty cards and electronic purchase tracking to offer enhanced services to customers
- also moving to software as a rentable service in B2B: design of software becomes more customer-centric

Integrating e-commerce vocabularies

- each business has its own *vocabulary* for describing its data and processes
    - describe what terms mean, how they relate to one another, and when they are valid
- each application has a schema based on the vocabulary
- data interchange requires *mapping* between schemas
    - may be able to define new schema
    - may have to use an existing schema from elsewhere
- will still have to map to/from trading partners' schemas
    - e.g., phone numbers as day and evening, or

o as work, home and mobile

- as well as own legacy systems

Integrating legacy data

- *legacy data*: data generated and used by processes that don't use the organisation's current technology
- e.g., Y2K-compliant date exchange
- legacy COBOL structure for date such as 102103:
- 01 POLICY-RECORD
- ...
- 05 POLICY-ISSUED
- 10 MM PIC 9(2)
- 10 DD PIC 9(2)
- 10 YY PIC 9(2)

- ...
- XML representation of Y2K-compliant date:
- <policy>
- ...
- <issue-date format=”ISO-8601">20031021</issue-date>
- ...
- </policy>
- transform legacy to XML:
- POLICY-ISSUED => issue-date
- format ISO-8601 <= CC&&YY&&MM&&DD

  where && means concatenation and CC is a calculated century prefix

BizTalk, eCo and XML.org

- BizTalk framework
  - loose grouping of many XML technologies by Microsoft
  - describes how to publish schemas in XML and to integrate programs using XML messages
  - includes a vocabulary for wrapping XML documents in an “envelope” which manages message routing and security
  - no pre-defined document types such as purchase order
- eCo framework
  - developed by CommerceNet, a business consortium
  - allows e-commerce systems to describe themselves, their services and their interoperability requirements
  - will take account of and complement other specifications
- XML.org registry
  - aids interoperability by publishing a range of specifications, schemas and vocabularies
  - attempts to prevent the duplication or overlapping of work that already exists

13.5 ebXML and SOAP

ebXML

- initiative undertaken by
  - UN/CEFACT: United Nations body for Trade Facilitation and Electronic Business
  - OASIS: Organisation for the Advancement of Structured Information Standards

- vision is a global electronic marketplace, where enterprises of any size can
    - find each other electronically
    - conduct business by exchanging XML-based messages
    - use off-the-shelf business applications

Open Financial Exchange (OFX)

- a technical specification created by Intuit, CheckFree and Microsoft
- allows financial institutions to communicate account transactions between themselves and their clients
- originated as an SGML application
- supported by accounting packages such as *Microsoft Money* and Intuit's *Quicken*
- supports 4 kinds of services
    - banking
    - bill presentation
    - bill payment
    - investment

ebXML Protocol Layer

The ebXML protocol layer provides the ability to send and receive messages via the Internet according to the ebXML Message Service specifications for transport, message packaging, and security. The ebXML 1.0 and 2.0 message service specifications are independent of the communication protocol used. Oracle WebLogic Integration supports the HTTP(S) communication protocol.

ebXML Business Messages

A business message is the basic unit of communication between trading partners. Business messages are exchanged as part of a conversation. The roles in a conversation are implemented by business processes, which choreograph the exchange of business messages.

An ebXML business message contains one XML business document and one or more attachments. An ebXML message is a communication protocol-independent MIME/Multipart message envelope, referred to as a *message package*. All message packages are structured in compliance with the SOAP Messages with Attachments specification.

Logical MIME Parts of an ebXML Business Message

The message package shown in the preceding figure illustrates the following logical MIME parts:

Header Container—Logical container in which one SOAP 1.1-compliant message is stored. This SOAP message is an XML document; its root element is the SOAP Envelope, which, in turn, contains the following elements:

SOAP Header—Contains ebXML-specific header elements, including the

ebXML MessageHeader element that specifies details such as from and to business IDs, service that relates to the business process, and action that relates to a node in the business process. The SOAP Header is a generic mechanism for adding features to a SOAP message.

SOAP Body—Container for message service handler control data and information related to the payload parts of the message.

Payload Container—Zero or more payloads. Each payload can contain XML or non-XML (binary) data.

13.6 XML Database

An **XML database** is a data persistence software system that allows data to be stored in XML format. These data can then be queried, exported and serialized into the desired format. XML databases are usually associated withdocument-oriented databases.

Two major classes of XML database exist

1. **XML-enabled**: these may either map XML to traditional database structures (such as a relational database, accepting XML as input and rendering XML as output, or more recently support native XML types within the traditional database. This term implies that the database processes the XML itself (as opposed to relying on middleware).
2. **Native XML** (NXD): the internal model of such databases depends on XML and uses XML documents as the fundamental unit of storage, which are, however, not necessarily stored in the form of text files.

Rationale for XML in databases

O'Connell gives one reason for the use of XML in databases: the increasingly common use of XML for datatransport, which has meant that "data is [sic] extracted from databases and put into XML documents and vice-versa".t may prove more efficient (in terms of conversion costs) and easier to store the data in XML format.

XML Enabled databases

XML enabled databases typically offer one or more of the following approaches to storing XML within the traditional relational structure:

1. XML is stored into a CLOB
2. XML is 'shredded' into a series of Tables based on a Schema
3. XML is stored into a native XML Type as defined by the ISO

RDBMS that support the ISO XML Type are:

1. IBM DB2 (pureXML
2. Microsoft SQL Server
3. Oracle Database
4. PostgeSQL

13.7 Storing Binary Data in XML

Storing binary data in XML

When you do need to include some binary data in an XML document, you'll need to make sure it won't trip up the XML parser. If the data happens to be text, you can dump it into a CDATA section and be done with it, but true binary data needs to be encoded in a safe and recoverable manner.

Luckily the MIME standards define a safe encoding scheme that's well-supported, base64. The base64 encoding makes binary data approximately 137% its original size so you're trading off additional storage space (and a little processing throughput) for the ability to embed the binary data in your XML document.

Typically you'd want to indicate the encoding and original file name in your XML.

One example of a base64-encoded file inside an XML document

```
<?xml version="1.0" encoding="UTF-8"?>
<sample>
  <description>
  An embedded image file.
  </description>
    <image name="stop.png" encoding="base64"
    source="FamFamFam"
    href="http://www.famfamfam.com/lab/icons/silk/">
iVBORw0KGgoAAAANSUhEUgAAABAAAAAQ
CAYAAAAf8/9hAAAABGdBTUEAAK/INwWK
6QAAABl0RVh0U29mdHdhcmUAQWRvYmUg
SW1hZ2VSZWFkeXHJZTwAAAJOSURBVDjL
pZI9T1RBFIaf3buAoBgJ8rl6QVBJVNDC
ShMLOhBj6T+wNUaDjY0WmpBIgYpAjL/A
ShJ+gVYYYRPIony5IETkQxZ2770zc2fG
YpflQy2MJzk5J5M5z/vO5ESstfxPxA4e
rL4Zuh4pLnoaiUZdq7XAGKzRJVbIBZ3J
PLJaD9c/eCj/CFgZfNl5qK5q8EhTXdxx
LKgQjAFr0NK0ppOpt9n51D2gd2cmsvOE
lVcvOoprKvuPtriNzsY8rH+H0ECoQEg4
WklY1czP8akZby51p6G3b6QAWBl43llS
VTlUfuZE3NmYh9Vl0HkHSuVq4ENFNWFd
C+uJ5JI/9/V2Y//rkShA1HF6yk/VxJ0f
07CcgkCB7+fSC8Dzcy7mp4l9/khlUzwe
caI9hT+wRrsOISylcsphCFLl1RXIvBMp
```

YDZJrKYRjHELACNEgC/KCQQofWBQ5nuV

64UAP8AEfrDrQEiLlJD18+p7BguwfAoB

UmKEsLsAGZSiFWxtgWWP4gGAkuB5YDRW

ylKAKIDJZBa1H8Kx47C1Cdls7qLnQTZf

fQ+20lB7EiU1ent7sQBQ6+vdq2PJ5dC9

ABW1sJnOQbL5Qc/HpNOYehf/4lW+jY4v

h2tr3fsWafrWzRtlDW5f9aVzjUVj72Fm

CqzBypBQCKzbjLp8jZUPo7OZyYm7bYkv

w/sAAFMd7V3lp5sGqs+fjRcZhVYKY0xu

pwysfpogk0jcb5ucffbbKu9Esv1Kl1N2

+Ekk5rg2DIXRmog1Jdr3F/Tm5mO0edc6

MSP/CvjX+AV0DoH1Z+D54gAAAABJRU5E

rkJggg==

</image>

</sample>

In a machine-generated XML document, you can leave out the white space, and run the entire base64-encoded file together without newline characters.

Avoiding the issue

The best way to deal with binary data in XML is to avoid it entirely. As you've seen in HTML, referring to an external file in a standardized way works well. This is a great option when you have some way for the client application to get at the external file. In the case of HTML', the browser just makes another HTTP request to get the data included through elements like <img>.

By not including the binary data directly in the XML, you avoid potentially wasteful text encodings and make it possible to implement other enhancements, such as the image caching most people love in their Web browsers.

13.8 Summary

The module have set of methods which will help us to create xml page with data in it I various form. Using this module we can also connect the SOAP with our native application.

13.9 Self Assessment Question

1. How can we integrate the XML with SOAP?
2. Define the database integration with XML?

Unit - 14 : Parsing XML in JAVA

Structure of the Unit:

14.0 Objective
14.1 Introduction
14.2 What is Parser
14.3 Using an XML Parser
14.4 Parser and XML Generator
14.5 Accessing DOM
14.6 Using SAX
14.7 Summary
14.8 Self Assessment Question

14.0 Objective

Using the unit we can create application to use the DOM.

14.1 Introduction

An XML parser is the component that deciphers the XML code. Without a parser, your code cannot be understood. Computers require instruction. An XML parser provides vital information to the program on how to read the file. Parsers come in multiple formats and styles. They can be free-standing software, libraries, modules and classes.

14.2 What is Parser

Where Do I Find a Parser?

Most of the time, you won't need to find a parser. Modern day browsers have built-in XML parsers. When you open Firefox or Internet Explorer, the program supplies the parser for you. This is true for development platforms as well, such as Java or Delphi. Unless you are building your own software, there is no need to worry about finding an XML parser.

How Does a Parser Work?

Parsers can be a bit complex. The goal is to transform XML into a readable code. Computer hardware only knows one language. It is the software that turns all the different languages into a workable format. Software is smart, but computers by themselves are ignorant. When presented with a bunch of XML code, computer hardware has no clue what it means. Parsers convert that code into something the hardware will recognize.

What are Validating XML Parsers?

Parsers come in two flavors, ones that validate and ones that don't. When you write XML code, you need to follow the rules. This means root elements, nesting and a declaration statement. A non-validating parser will give the code a quick check to make sure you have all the basics. A validating parser looks deeper. Parsers that validate compare a set of specific rules for each XML file, such as a DTD or schema. With these rules in hand, it goes through the XML and make decisions about default values and validates data types.

What are Standalone XML Parsers?

This is a parser that works outside of any other program. Standalone parsers are exactly what they sound like; separate packages that only parse XML. More often than not, you will find little need for a standalone parser. They may come in handy if you need to parse code locally, or without an editor or server package. Programmers that design software look for standalone parsers to view their code. Beyond that, they serve little purpose because most editing software will have a parser built-in.

14.3 Using an XML Parser

The XML is based on text, so one might think that it would be dead easy to take any XML input and have it converted in the way one wants.

Unfortunately, that is wrong. If you think you'll be able to parse a XML file with your own home grown parser you did overnight, think again, and look at the XML specs closely. It's as complex as the CGI specs, and you'll never want to waste precious time trying to do something that will surely end up wrong anyway. Most of the background discussions on why you have to use CGI.pm instead of your own CGI-parser apply here.

<CHATTER><INFO site="http://perlmonks.org" sitename="Perl Monks">

Rendered by the Chatterbox XML Ticker</INFO>

<message author="OeufMayo" time="20010228112952">

test</message>

<message author="deprecated" time="20010228113142">

pong</message>

<message author="OeufMayo" time="20010228113153">

/me test again; :)</message>

<message author="OeufMayo" time="20010228113255">

please note the use of HTML

tags</message>

</CHATTER>

Let's assume we want to output this file in a readable way (though it'll still be barebone). It doesn't handles links and internal HTML entities. It only gets the CB ticker, parses it and prints it, you have to launch it again to follow the wise meditations and the brilliant rethoric of the other fine monks present at the moment.

```
1  #!/usr/bin/perl -w
2  use strict;
3  use XML::Parser;
4  use LWP::Simple;  # used to fetch the chatterbox ticker
```

```
5
6  my $message;      # Hashref containing infos on a message
7
8  my $cb_ticker = get("http://perlmonks.org/index.pl?node=chatterbox+
+xml+ticker");
9  # we should really check if it succeeded or not
10
11  my $parser = new XML::Parser ( Handlers => {   # Creates our parse
+r object
12                          Start   => \&hdl_start,
13                          End     => \&hdl_end,
14                          Char    => \&hdl_char,
15                          Default => \&hdl_def,
16                        });
17  $parser->parse($cb_ticker);
18
19  # The Handlers
20  sub hdl_start{
21      my ($p, $elt, %atts) = @_;
22      return unless $elt eq 'message';  # We're only interrested in
+what's said
23      $atts{'_str'} = '';
24      $message = \%atts;
25  }
26
27  sub hdl_end{
28      my ($p, $elt) = @_;
29      format_message($message) if $elt eq 'message' && $message && $
+message->{'_str'} =~ /\S/;
30  }
31
```

```
32  sub hdl_char {
33      my ($p, $str) = @_;
34      $message->{'_str'} .= $str;
35  }
36
37  sub hdl_def { }  # We just throw everything else
38
39  sub format_message { # Helper sub to nicely format what we got fro
+m the XML
40      my $atts = shift;
41      $atts->{'_str'} =~ s/\n//g;
42
43      my ($y,$m,$d,$h,$n,$s) = $atts->{'time'} =~ m/^(\d{4})(\d{2})(
+\d{2})(\d{2})(\d{2})(\d{2})$/;
44
45      # Handles the /me
46      $atts->{'_str'} = $atts->{'_str'} =~ s/^\/me// ?
47      "$atts->{'author'} $atts->{'_str'}"   :
48      "<$atts->{'author'}>: $atts->{'_str'}";
49      $atts->{'_str'} = "$h:$n " . $atts->{'_str'};
50      print "$atts->{'_str'}\n";
51      undef $message;
52  }
```

Step-by-step code walkthrough:

Lines 1 to 4

Initialisation of the basics needed for this snippet, XML::Parser, of course, and LWP::Simple to get the chatterbox ticker.

Line 8

LWP::Simple get the requested URL, and put the content of the page in the $cb_ticker scalar.

Lines 11 to 16

The most interesting part, no doubt. We create here a new XML::Parser object. The Parser can come in

different styles, but when you have to deal with simple data, like the CB ticker, the Handlers way is the easiest (see also the Subs style, as it is really close to this one).

For this object, we define four handlers subs, each representing a different state in the parsing process.

- The 'Start' handler is called whenever a new element (or tag, HTML-wise) is found. The sub given is called with the expat object, the name of the element, and a hash containing all the atrributes of this element.
- The 'End' is called whenever an element is closed, and is called with the same parameters as the 'Start', minus the attributes.
- The 'Char' handler is called when the parser finds something which is not mark-up (in our case, the text enclosed in the <message> tag).
- Finally, the 'Default' handler is called, well, by default, when anything else matching the three other handlers is called.

Line 17

The line that does all the magic, parsing and calling all your subs for you at the right moment.

Lines 20-25: the Start handler

We only want to deal with the <message> elements (those containing what it is being said in the Chatterbox) so we'll happily skip every other element.

We got a hash with the attributes of the element, and we're going to use this hash to store the string that will contain the text to be displayed in the $atts{'_str'}

Lines 27-30: the End handler

Once we've reached the end of a message element, we format all the info we have gathered and prints them via the format_message sub.

Lines 32-35: the Char handler

This sub gets all the strings returned by the parser and appends it to the string to be finally displayed

Line 37: the Default handler

It does nothing, but it doesn't have to figure out what to do with this!

Lines 39-52

This subroutine mangles all the info we got from the XML file, with bad regexes and all, and prints the formatted text in a hopefully readable way. Please note that XML::Parser handled all of the decoding of the < and > entities that were included in the original XML file

14.4 Parser and XML Generator

XML parser generator that took an XML description of how you'd like the native-language data structures to look and where in the XML it could find the values for those data structures. The Java code-base for this was ugly, ugly, ugly. I tried several times to clean it up into something publishable. I tried to clean it up several times so that it could actually generate the parser it used to read the XML description file.

Alas, the meta-ness, combined with the clunky Java code, kept me from completing the circle.

Fast forward to last week. Suddenly, I have a reason to parse a wide variety of XML strings in Objective C. I certainly didn't want to pull out the Java parser generator and try to beat it into generating Objective C, too. That's fortunate, too, because I cannot find any of the copies (in various states of repair) that once lurked in ~/src.

What's a man to do? Write it in Lisp, of course.

Example

Here's an example to show how it works. Let's take some simple XML that lists food items on a menu:

```
<menu>
    <food name="Belgian Waffles" price="$5.95" calories="650">
        <description>two of our famous Belgian Waffles with plenty of real maple syrup</description>
    </food>
    <!— ... more food entries, omitted here for brevity ... —>
</menu>
```

We craft an XML description of how to go from the XML into a native representation:

```
<parser_generator root="menu" from="/menu">
  <struct name="food item">
    <field type="string" name="name" from="@name" />
    <field type="string" name="price" from="@price" />
    <field type="string" name="description" from="/description/." />
    <field type="integer" name="calories" from="@calories" />
  </struct>
   <struct name="menu">
    <field name="menu items">
      <array>
        <array_element type="food item" from="/food" />
      </array>
    </field>
  </struct>
</parser_generator>
```

Now, you run the parser generator on the above input file:

```
% sh parser-generator.sh —language=lisp \
                    —types-package menu \
                    —reader-package menu-reader \
                    —file menu.xml
```

This generates two files for you: types.lisp and reader.lisp. This is whattypes.lisp looks like:

```
(defpackage :menu
  (:use :common-lisp)
  (:export #:food-item
           #:name
           #:price
           #:description
           #:calories
          #:menu
           #:menu-items))
 (in-package :menu)
 (defclass food-item ()
  ((name :initarg :name :type string)
   (price :initarg :price :type string)
   (description :initarg :description :type string)
   (calories :initarg :calories :type integer)))
 (defclass menu ()
  ((menu-items :initarg :menu-items :type list :initform nil)))
```

I will not bore you with all of reader.lisp as it's 134 lines of code you never had to write. The only part you need to worry about is the parsefunction which takes a stream for or pathname to the XML and returns an instance of the menu class. Here is a small snippet though:

```
;;; ============================================================
;;; food-item struct
;;; ============================================================
(defmethod data progn ((handler sax-handler) (item food-item) path value)
  (with-slots (name price description calories) item
    (case path
```

```
(:|@name| (setf name value))

(:|@price| (setf price value))

(:|/description/.| (setf description value))

(:|@calories| (setf calories (parse-integer value))))))
```

Where it's at

I currently have the parser generator generating its own parser (five times fast). I still have a little bit more that I'd like to add to include assertions for things like the minimum number of elements in an array or the minimum value of an integer. I also have a few kinks to work out so that you can return some type other than an instance of a class for cases like this where the menuclass just wraps one item.

14.5 Accessing DOM

The DOM is a W3C (World Wide Web Consortium) standard.

The DOM defines a standard for accessing documents like XML and HTML:

"The W3C Document Object Model (DOM) is a platform and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure, and style of a document."

The DOM is separated into 3 different parts / levels:

- Core DOM - standard model for any structured document
- XML DOM - standard model for XML documents
- HTML DOM - standard model for HTML documents

The DOM defines the **objects and properties** of all document elements, and the **methods**(interface) to access them.

Most browsers have a built-in XML parser to read and manipulate XML.

The parser converts XML into a JavaScript accessible object (the XML DOM).

XML Parser

The XML DOM contains methods (functions) to traverse XML trees, access, insert, and delete nodes.

However, before an XML document can be accessed and manipulated, it must be loaded into an XML DOM object.

An XML parser reads XML, and converts it into an XML DOM object that can be accessed with JavaScript.

Most browsers have a built-in XML parser.

Load an XML Document

The following JavaScript fragment loads an XML document ("book.xml"):

Example

```
• if (window.XMLHttpRequest)
  {
  xhttp=new XMLHttpRequest();
  }
else // IE 5/6
  {
  xhttp=new ActiveXObject("Microsoft.XMLHTTP");
  }
xhttp.open("GET","books.xml",false);
xhttp.send();
xmlDoc=xhttp.responseXML;
```

- **Load an XML String**
- The following code loads and parses an XML string:
- Example

```
• if (window.DOMParser)
  {
  parser=new DOMParser();
  xmlDoc=parser.parseFromString(text,"text/xml");
  }
else // Internet Explorer
  {
  xmlDoc=new ActiveXObject("Microsoft.XMLDOM");
  xmlDoc.async=false;
  xmlDoc.loadXML(text);
  }
```

Access Across Domains

For security reasons, modern browsers do not allow access across domains.

This means, that both the web page and the XML file it tries to load, must be located on the same server.

The examples on W3Schools all open XML files located on the W3Schools domain.

If you want to use the example above on one of your web pages, the XML files you load must be located on your own server.

14.6 Using SAX

SAX (**Simple API for XML**) is an event-basedsequintial access parser API developed by the XML-DEV mailing list for XML documents. SAX provides a mechanism for reading data from an XML document that is an alternative to that provided by the Document Object Model (DOM). Where the DOM operates on the document as a whole, SAX parsers operate on each piece of the XML document sequentially

Definition

Unlike DOM, there is no *formal* specification for SAX. The Java implementation of SAX is considered

to be normative.. SAX processes documents state-dependently, in contrast toDOM which is used for state-independent processing of XML documents.

Benefits

SAX parsers have some benefits over DOM-style parsers. A SAX parser only needs to report each parsing event as it happens, and normally discards almost all of that information once reported (it does, however, keep some things, for example a list of all elements that have not been closed yet, in order to catch later errors such as end-tags in the wrong order). Thus, the minimum memory required for a SAX parser is proportional to the maximum depth of the XML file (i.e., of the XML tree) and the maximum data involved in a single XML event (such as the name and attributes of a single start-tag, or the content of a processing instruction, etc.).

This much memory is usually considered negligible. A DOM parser, in contrast, typically builds a tree representation of the entire document in memory to begin with, thus using memory that increases with the entire document length. This takes considerable time and space for large documents (memory allocation and data-structure construction take time). The compensating advantage, of course, is that once loaded *any* part of the document can be accessed in any order.

Drawbacks

The event-driven model of SAX is useful for XML parsing, but it does have certain drawbacks.

Virtually any kind of XML validation requires access to the document in full. The most trivial example is that an attribute declared in the DTD to be of type IDREF, requires that there be an element in the document that uses the same value for an ID attribute. To validate this in a SAX parser, one must keep track of all ID attributes (any one of them *might* end up being referenced by an IDREF attribute at the very end); as well as every IDREF attribute until it is resolved. Similarly, to validate that each element has an acceptable sequence of child elements, information about what child elements have been seen for each parent, must be kept until the parent closes.

Additionally, some kinds of XML processing simply require having access to the entire document. XSLT and Xpath, for example, need to be able to access any node at any time in the parsed XML tree. Editors and browsers likewise need to be able to display, modify, and perhaps re-validate at any time. While a SAX parser may well be used to construct such a tree initially, SAX provides no help for such processing as a whole.

processing with SAX

A parser that implements SAX (i.e., *a SAX Parser*) functions as a stream parser, with an event-driven API The user defines a number of callback methods that will be called when events occur during parsing. The SAX events include (among others):

- XML Text nodes
- XML Element Starts and Ends
- XMLprocessing Instructioins
- XML Comments

Some events correspond to XML objects that are easily returned all at once, such as comments. However, XML *elements* can contain many other XML objects, and so SAX represents them as does XML itself: by one event at the beginning, and another at the end. Properly speaking, the SAX interface does not deal in *elements*, but in *events* that largely correspond to *tags*. SAX parsing is unidirectional; previously parsed data cannot be re-read without starting the parsing operation again.

There are many SAX-like implementations in existence. In practice, details vary, but the overall model is the same. For example, XML attributes are typically provided as extreme name and value arguments passed to element events, but can also be provided as separate events, or via a hash or similar collection of all the attributes. For another, some implementations provide "Init" and "Fin" callbacks for the very start and end of parsing; others don't. The exact names for given event types also vary slightly between implementations.

Given the following XML document:

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

<DocumentElement param="value"**>**

<FirstElement>

¶ Some Text

</FirstElement>

<?some_pi some_attr="some_value"**?>**

<SecondElement param2="something"**>**

Pre-Text **<Inline>**Inlined text**</Inline>** Post-text.

</SecondElement>

</DocumentElement>

14.7 Summary

To use the parser with java we have various syntaxes to implement it in XML. The parser also provide facility to integrate itself with DOM. There are some merits and demerits of parser which we discuss in this module.

14.8 Self Assessment Question

1. Define the steps to use parser?
2. How can we integrate DOM with parser?

Unit - 15 : Security in XML

Structure of the Unit:

15.0 Objective
15.1 Introduction
15.2 Security Overview
15.3 Canonicalization
15.4 XML Encryption
15.5 XML Digital Signature
15.6 XKMS Structure
15.7 Guidelines for Signing XML Document
15.8 Summary
15.9 Self Assessment Question

15.0 Objective

To use and successfully implement the XML it must be bugg free, so we have to mention the XML security method in this unit

15.1 Introduction

The extensible markup language (XML) is a markup language promoted by the World Wide Web consortium (W3C). XML overcomes the limitations of hypertext markup language (HTML) and represents an important opportunity to solve the problem of protecting information distributed on the Web, with the definition of access restrictions directly on the structure and content of the document. This chapter summarizes the key XML security technologies and provides an

15.2 Security Overview

overview of how they fit together and with XML. It should serve as a roadmap for future research and basis for further exploration of relevant scientific literature and standard specifications. An essential requirement of new Internet-wide security standards is that they apply to content created using extensible markup language (XML) .XML has been adopted widely for a great variety of applications and types of content. Examples of XML-based markup languages are security assertion markup language (SAML) Traditionally, XML security has developed along two distinct though related lines of research, corresponding to two facets of the XML security notion. The frst facet defnes XML security as a set of security techniques (access control , diferential encryption , digital signature) tightly coupled with XML to maintain the main features of the XML semi-structured data model while adding to it all necessary security capabilities. This is especially important in XML-based protocols, such as SOAP, which are explicitly designed to allow intermediary processing and modification of messages XML security relies on some legacy security algorithms and tools, but the actual formats used to implement security requirements are specifcally aimed at XML applications, supporting common XML technical approaches for managing content, such as specifying content with uniform resource identifier strings (URIs) or using other XML standard definitions like XPath and XQuery for locating portions of XML content. A second important facet of XML security deals with models and languages specifying and exchanging access control policies to generic resources, which may or may not comply with the XML data model.

XML appears in fact a natural choice as the basis for the common security policy language, due to the ease with which its syntax and semantics can be extended and the widespread support that it enjoys from all the main platform and tool vendors. To this purpose, several proposals have been introduced for access control to distributed heterogeneous resources from multiple sources. One of the most important XML-based language is extensible access control markup language (XACML), a language for defining rules and policies for controlling access to information. Another security aspect that needs to be taken into consideration is the secure and selective dissemination of XML documents. Often, XML documents contain information with different level of sensitivity, which has to be shared by user communities and managed according to access control policies.

15.3 Canonicalization

In computer science, **canonicalization** (abbreviated **c14n**, where 14 represents the number of letters between the C and the N; also sometimes **standardization** or **normalization**) is a process for converting data_ that has more than one possible representation into a "standard", "normal", or cononical form This can be done to compare different representations for equivalence, to count the number of distinct data tructures, to improve the efficiency of various algorithms by eliminating repeated calculations, or to make it possible to impose a meaningful sorting order.

Web servers

Canonicalization of filenames is important for computer security. For example, a web server may have a security rule stating "only execute files under the cgi directory (C:\inetpub\wwwroot\cgi-bin)". The rule is enforced by checking that the path starts with "C:\inetpub\wwwroot\cgi-bin\", and if it does, the file is executed.

Should file "C:\inetpub\wwwroot\cgi-bin\..\..\..\Windows\System32\cmd.exe" be executed? No, because this trick path goes back up the directory hierarchy (through use of the '..' path specifier), not staying within cgi-bin. Accepting it at face value would be an error due to failure to canonicalize the filename to the unique (simplest) representation, namely:

Unicode

Varible length encoding in the Unicode standard, in particular UTF-8, have more than one possible encoding for most common characters. This makes string validation more complicated, since every possible encoding of each string character must be considered. A software implementation which does not consider all character encodings runs the risk of accepting strings considered invalid in the application design, which could cause bugs or allow attacks. The solution is to allow a single encoding for each character. Canonicalization is then the process of translating every string character to its single allowed encoding. An alternative is for software to determine whether a string is canonicalized, and then reject it if it is not. In this case, in a client/server context, the canonicalization would be the responsibility of the client.

Search engines and SEO

In web search and search engine optimization (SEO),URL cononicalization deals with web content that has more than one possible URL. Having multiple URLs for the same web content can cause problems for search engines - specifically in determining which URL should be shown in search results

XML

A Canonical XML document is by definition an XML document that is in XML Canonical form, defined by The canonical XML specification. Briefly, canonicalization removes whitespace within tags, uses particular character encodings, sorts namespace references and eliminates redundant ones, removes XML and DOCTYPE declarations, and transforms relative URIs into absolute URIs.

Simple example: Given two versions of the same XML:

* “<node1>Data</node1> <node2>Data</node2>”

* “<node1>Data</node1> <node2>Data</node2>”

Note the extra spaces in the samples, the canonicalized version of these two might be:

* “<node1>Data</node1><node2>Data</node2>”

Note that the spaces are removed — this is one thing a canonicalizer does. A real canonicalizer may make other changes as well.

A full summary of canonicalization changes is listed below:

* The document is encoded in UTF-8

* Line breaks normalized to #xA on input, before parsing

* Attribute values are normalized, as if by a validating processor

* Character and parsed entity references are replaced

15.4 XML Encryption

XML encryption [6] can be used to encrypt arbitrary data. As for XML signature, the main advantage given by XML encryption is that it supports the

<patient>

<patientId>123a45d</patientId>

<diagnosis id=”Diagnosis001">

<EncryptedData Type=”http://www.w3.org/2001/04/xmlenc#Element”

xmlns=”http://www.w3.org/2001/04/xmlenc#”>

<EncryptionMethod Algorithm=’http://www.w3.org/2001/04/xmlenc#tripledes-cbc”/>

<ds:KeyInfo xmlns:ds=”http://www.w3.org/2000/09/xmldsig#”>

...

</ds:KeyInfo>

<CipherData>

<CipherValue>H343HJS90F</CipherValue>

</CipherData>

</EncryptedData>

</PaymentInfo>

</diagnosis>

</patient>

Fig. 6.2. An example of XML encryption

encryption of speci?c portions of an XML document rather than the complete document. This feature is particularly important in a business scenario,

where di?erent remote parties cooperate to provide a service. A consequence

of partial encryption is also support for multiple encryptions. For instance, in

a health-care scenario, when a patient goes to a hospital for a visit, her record

contains both doctor's diagnosis and information for billing payment. In this

case payment information must not be seen by a doctor and diagnosis must

not be seen by the billing administrator. This requirement can be obtained by

encrypting the two types of information using a di?erent encryption key. XML

encryption supports encryption at di?erent granularity levels: document, element, and element-content level. As an example, suppose that we need to

encrypt the diagnosis speci?ed within a patient record. Figure 6.2 illustrates

the XML encryption, where the content of the diagnosis element has been

replaced by the EncryptedData element with attribute Type, which speci?es

the type of the encrypted data .

15.5 XML Digital Signature

An XML signature is a digital signature obtained by applying a digital signature operation to arbitrary data. The concept of a digital signature is not new and several technologies have already been presented to the community . However, while the existing technologies allow us to sign only a whole XML document, XML signature provides a means to sign a portion of a document. This functionality is very important in a istributed multi party environment, where the necessity to sign only a portion of a document arises whenever changes and additions to the document are required. For instance, consider a patient record stored in a hospital repository. This record can contain several entries (diagnoses) coming from several doctors. Each doctor wants to take responsibility only over her diagnosis. In this case, every additional diagnosis added to the patient record must be singularly signed. This important feature is supported by XML signature. The extensible nature of XML also allows support for multiple signatures inside the same document. It is also important to highlight that the possibility of signing online a portion of a document and inserting the signature inside the document avoids the development of ad hoc methods to manage persistent signatures, and provides a ?exible mechanism to sign and preserve part of the document. The data to be signed are first digested (a digest is a fixed-length representation of a resource and is

created using, for example, a hash function such as SHA-1) and the resulting value is placed in an element, called DigestValue, together with other information. This element is then digested and cryptographically signed. An XML signature is inserted in the signature element and it can be associated with the data objects in three different ways:

<patient>

<patientId>123a45d</patientId>

<diagnosis id="Diagnosis001">...</diagnosis>

<Signature Id="Signature001" xmlns="http://www.w3.org/2000/09/xmldsig#">

<SignedInfo>

<CanonicalizationMethod Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315"/>

<SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#dsa-sha1"/>

<Reference URI="#Diagnosis001">

<Transforms>

<Transform Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315"/>

</Transforms>

<DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>

<DigestValue>dh5gf68fhgfjt7FHfdgS55FghG=</DigestValue>

</Reference>

</SignedInfo>

<SignatureValue>MC0CFFrVLtRlk=...</SignatureValue>

<KeyInfo>...</KeyInfo>

</Signature>

</patient>

Fig. 6.1. An example of internal XML detached signature to be signed; (ii) enveloped signature, where the signature is a child element of the data to be signed; (iii) detached signature, where the signature element and the signed data objects are separated. Figure 6.1 illustrates an example of internal detached signature, where a doctor's diagnosis (element diagnosis) is signed. As is visible from this example, the signature element is inserted within the XML document as a sibling of the signed element. The signature element contains three subelements: SignedInfo, SignatureValue, and KeyInfo.The required SignedInfo element contains the information signed and has three subelements: the required Canonicalization Method element de- ?nes the algorithm used to canonicalize the SignedInfo element before it is signed or validated; the required SignatureMethod element specifies the digital signature algorithm used to generate the signature (DSA-SHA1, in our example); one or more Reference elements identify the data that is digested via a URI. The Reference element contains: an option Transforms element that in turn contains a list of one or more Transform elements describing a transformation algorithm used to transform the data before

they are digested;

the DigestMethod element speci?es the method used to generate the digest

value reported in the DigestValue element.

The SignatureValue element contains the signature value computed over

the SignedInfo element.

Finally, the KeyInfo element indicates the key that must be used for signature validation.

15.6 XKMS Structure

XML Key Management Specification (XKMS) uses the web services framework to make it easier for developers to secure inter-application communication using public key infrastructure (PKI).XML Key Management Specification is a protocol developed by W3C which describes the distribution and registration of public keys. Services can access an XKMS compliant server in order to receive updated key information for encryption and authentication.

Architecture

XKMS consists of two parts:

XKISS

XML Key Information Service Specification

XKRSS

XML Key Registration Service Specification

The XKISS service specification is concerned with management of the public component of a public key pair. The XKRSS is concerned with management of private keys. In both cases the goal of XKMS is to allow all the complexity of traditional PKI implementations to be offloaded from the client to an external service. While this approach was originally suggested by Diffie and Hellman in their New Directions paper this was generally considered impractical at the time leading to commercial development focusing on the certificate based approach proposed by Loren Kohnfelder. Development history The team that developed the original XKMS proposal submitted to the W3C included Warwick Ford, Phillip Hallam-Baker (editor) and Brian LaMacchia. The architectural approach is closely related to the MIT PGP Key server originally created and maintained by Brian LaMacchia. The realization in XML is closely related to SAML, the first edition of which was also edited by Hallam-Baker.

At the time XKMS was proposed no security infrastructure was defined for the then entirely new SOAP protocol for Web Services. As a result a large part of the XKMS specification is concerned with the definition of security 'bindings' for specific Web Services protocols.

XKMS Structure

On the whole the XKMS specifies the protocols for distributing and registering public keys. This is suitable for use in conjunction with the planned standard for XML signature and as an ddditional standard for XML encryption.

The structure of XKMS contains two sections:

o XML Key Information Service Specification (X-KISS)

o XML Key Registration Service Specification (X-KRSS)

XML Key Information Service Specification

X-KISS characterizes a protocol for a reliance service. It helps in managing the public-key information contained in documents that confirm to the XML signature specification. The basic objective of this protocol design is that relieving the XML programmers from the complex task of writing the code to process the XML signature ds:KeyInfo element. Essentially PKI may be ased upon a different specification such as X.509, the international standard for public-key certificates or Pretty Good Privacy (PGP), the widely available public key encryption system. Any trust policy can be utilized along with the XML signature specification.

When ever, a person is signing a document it is not necessary to specify any key information except that the value for the element . The value includes the key name, certificate name, key identifier and so on. Otherwise a link may be provided to a location which contains the required KeyInfo details.

XML Key Registration Service Specification

The Registration of the public key information is done through the protocol X-KRSS specifies. Once the key is registered it can be used along with other web services. The same protocol may be also used for recovery of the private keys. Since the protocol provides for authentication of the applicant, the key pair public key and private key may be generated by the applicant. This is the proof of possession of the private key. A means of communicating the private key to the client is provided if the private key is generated by the registration service.

15.7 Guidelines for signing XML Document

Signing of XML documents needs care, since any change in the document like introduction of white space, change of case tend to change the signature.

The following two points to be kept in mind when going for signing the document:

1. Content Presentation techniques may introduce changes

2. Transformation may alter the content

XML relies on transformations and substitutions during the processing of XML documents. For example, if an XML document includes an embedded style sheet or references to an external style sheet, the transformed document should be represented to the user rather than the document without the style sheet. In this case, the signer should be careful to sign not only the original XML but also the other information that may affect the presentation.

While due consideration is not been given for handling the original and transformed document, it will return a different result than intended. As in any security infrastructure, the security of an overall system will depend on the security and integrity of procedures and personnel as well as procedural enforcement.

15.8 Summary

To use and successfully implement the XML it must be bugg free, so we have to mention the XML security method in this module. Varible length encoding in the Unicode standard, in particular UTF-8, have more than one possible encoding for most common characters.A Canonical XML document is by definition an XML document that is in XML Canonical form

15.9 Self Assessment Question

1. What is Canonicalization ?
2. What type of security is avalible in XML ?
3. Explain the concept of digital signature .
4. Explain XKMS structure .
5. What is sigining explain in detail.