Chapter 3
The Client Tier

Representing Content - XML

- XML: e**X**tensible **M**arkup **L**anguage.
- XML is a standard for creating markup languages which describe the structure of data.
- It is a widely used system for defining data formats.
- XML is cross-platform, independent tool for exchanging data.
- XML is a **framework** for defining markup languages:
  1. There is **no fixed collection of markup tags** - we may define our own tags, tailored for our kind of information
  2. Each XML language is targeted at its own application domain, but the languages will share many features
  3. There is a common set of **generic tools** for processing documents
  4. is a markup language which relies on the concept of rule-specifying tags and the use of a tag-processing application that knows how to deal with the tags.

XML Basics:

- Developed by W3C (World Wide Web Consortium).
- It is the pared-down version of SGML (Standardized Generalized Markup Language).
- Designed especially for web documents
- Allows you to create your own language for displaying documents.
- "***XML is not the replacement of HTML.***"
- XML requires a processing application. There is no XML browser in market yet.
- XML isn't about display – it's about Structure.
- XML file should be saved with the extension "**.xml**".
- You can use any text editor to write XML code. And the output can be viewed in any browser.
- Every XML page consists of processing instruction at the very first line.
- There should be a root element in every XML document, which should be unique.

Official Goals of XML

"Self-education, the only kind of education that exists."

Rules for writing XML:

- The first line must be **<?XML version = “1.0”?>**.
- Tags are enclosed in angle brackets.
- Tags are case sensitive and must have a matching closing tag.
- Tags may contain attributes in the form **name = “value”**.
- Tags may contain text, other tags, or both. Tags content lies between the starting and ending tag.

Advantages of XML:

- The first benefit of XML is that because you are writing your own markup language, you are not restricted to a limited set of tags defined by proprietary vendors.
- With XML, you can create your own set of tags at your own pace.
- XML allows every person/organization to build their own tag library which suits their needs perfectly.
- XML allows you to define all sorts of tags with all sorts of rules, such as tags representing business rules or tags representing data description or data relationships.
- With XML, GUI is extracted. Thus, changes to display do not require futzing with the data. Instead separate style-sheet will specify a table display or a list display.
- Searching the data is easy and efficient. Search-engine can simply parse the description-bearing tags rather than muddling in the data. Tags provide the search engines with the intelligence they lack.
- Complex relationships like tree and inheritance can be communicated.
- The code is much more legible to the person coming into the environment with no prior knowledge.

Disadvantages of XML:

- XML requires a processing application.
- There are no XML browsers on the market yet. Thus, XML documents must either be converted into HTML before distribution or converting it to HTML on-the-fly by middleware.
- Barring translation, developers must code their own processing applications.
- XML isn't about display -- it's about structure.
- This has implications that make the browser question secondary. So the whole issue of what is to be displayed and by what means is intentionally left to other applications.

XML Declaration:

- To begin an XML document, it is a good idea to include the XML declaration as the very first line of the document.
- The XML declaration is a processing instruction that notifies the processing agent that the following document has been marked up as an XML document.
- The XML declaration looks as following:

  <?xml version = "1.0"?>
- In its full regalia, the XML declaration might look like the following:

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

Elements:

- Elements are the basic unit of XML content.
- Syntactically, an element consists of a start tag, and an end tag, and everything in between.
- *Example:*

  <NAME>Frank Lee</NAME>
- All XML documents must have at least one root element to be well formed. The root element, also often called the document tag, must follow the prolog (XML declaration plus DTD) and must be a nonempty tag that encompasses the entire document.
- XML defines the text between the start and end tags to be "character data" and the text within the tags to be "markup".

WELL-FORMED XML

A well-formed XML Document has to follow several generic rules associated with element and attributes.

Rules for Elements:

- Every starting tag must have a matching end tag.
- Tags can’t overlap.
- XML documents can have only one root element.
- Element names must XML naming conventions.

- XML is case sensitive.
- XML will keep white space in your text.

Attributes:

- Attributes are properties that are associated with an element.
- An attribute specification is the "*name-value*" pair that is associated with the element.
- Adding attribute provides an alternative way to include information in an element.
- Like XML tags, attributes are also not restricted to store the type of information.
- They are attached with a starting tag.
- Attributes must have values.
- We can use double quote (" ") or single quote (' ') to delineate attribute values.
- *Example:*

  <ELEMENT_NAME attribute_name = "value"> or

  < ELEMENT_NAME attribute_name = 'value'>

Why use Attribute?

- Attributes provide meta-data that may not be relevant to most applications dealing with our XML.
- Attributes are easier to use, they don't require nesting and you don't have to worry about crossed tags.
- Attributes takes much less space than elements.

Rules for Creating Attributes:

- The name must begin with a letter or an underscore ('_'), followed by '0' or some letters, digits, periods ('**.**'), hyphens('-').
- The XML specification states that attribute names beginning with the prefix "xml" (in any combination of uppercase or lowercase letters) are "reserved for standardization". Although IE-5 doesn't enforce this restriction, its better not to use this prefix to avoid future problems.
- A particular attribute name can appear only once in the same start-tag or empty tag. Example:

  <ANIMATION FileName="a.ani" **FileName**="b.ani"**>** Some Text **</ANIMATION>**

- The value of attribute must be delimited using either single quotes (‘ ‘) or double quotes (“ “).
- The value of attribute cannot contain the same quote character used to delimit it.
- The value of attribute cannot include the ‘<’ character and ‘&’ character except to begin a character or entity reference.

Comments:

- Comments provide a way to insert text that isn’t really part of the document, but rather is intended for people who are reading the XML source itself.
- Comments Start with the String <!- - and end with the string - ->
- <u>*Example:*</u>

```
<NAME NickName = “Maddy”>
        <FIRST-NAME> John <FIRST-NAME>
        <MIDDLE-NAME> </MIDDLE-NAME>
                <!- - John lost his Middle Name in Fire - - >
        <LAST-NAME> Doe </LAST-NAME>
</NAME>
```

- <u>***Points to Ponder:***</u>

≢ You cannot have comment inside a tag.

<MIDDLE-NAME> </MIDDLE-NAME<!- - John lost his Middle Name in Fire - - > **>**

≢ You can’t use ‘- -‘ character inside a comment.

Escaping Characters:

- There are some reserved characters that you can’t include in your PCDATA because they are used in XML syntax.
- For Example:
  ‘**<**’ and ‘**&**’ characters are example of those characters.

<COMPARISON> 6 is < 7 & 7 > 6 **</COMPARISION>**

- To escape such characters, you simply can use &lt; and &amp; characters.

- It automatically un-escapes the characters for you when it displays the document.

Character Reference:

- The strings such as &#NNN; (where 'NNN' is the Unicode numbers), used to escape the reserved words of XML is known as C*haracter Reference*.
- It can also be &#Xnnn; with 'X' preceding the number and 'nnn' is a hexadecimal number.

Entity Reference:

- The strings such as &lt; used to escape the reserved words of XML is known as *Entity Reference*.
- The Entity Reference can be like &gt; &amp; &lt; etc.
- You can either use character reference or entity reference to escape the characters.

Processing Instructions:

- A processing instruction is a bit of information meant for the application using the XML document.
- That is, they are not really of interest to the XML parser.
- Instead, the instructions are passed intact straight to the application using the parser.
- The application can then pass this on to another application or interpret it itself.
- All processing instructions follow the generic format of:

  <?NAME_OF_APPLICATION Instructions?>
- Example of processing instructions:

```
<?XML version = "1.0"?>
  <doc>
          <?JavaScript function funct1 (y){ ?>
          <?JavaScript var x;?>
          <?JavaScript x = y * 10;?>
          <?JavaScript return (x);?>
          <?JavaScript }?>
  </doc>
```

Document Type Definitions

What is DTD?

- DTD defines the rules that set out how the document should be structured, what elements should be included, what kind of data may be included and what default values to use.
- Multiple documents and applications can share DTDs.
- DTDs use a formal grammar to describe the structure and syntax of an XML document, including the permissible values for much of that document's content.
- DTDs:
  1. Provide a formal and complete definition of an XML vocabulary.
  2. Are shareable descriptions of the structure of an XML documents.
  3. Are a way to validate specific instances of XML documents and constraints.
  4. Are restricted to one DTD per document instance.
  5. Specifies the validity of each tag.

Internal Vs. External DTD

- DTD may be divided into two parts: the internal subset and the external subset.
- These subsets are relative to the document instance.
- The internal subset is a portion of the DTD including within the document.
- The external subset is a portion of declarations that are located in a separate document.
- A DTD might be contained entirely within the document, with no external subset, or a document may simply refer to an external subset and contain no DTD declaration of its own.
- In many cases, DTD may use a combination of both.
- DTD declarations in the internal subset have priority over those in the external subset.

Associating a DTD with an XML document

- Each XML document can be associated with one, and only one DTD using single DOCTYPE declaration.
- DTDs are linked to XML documents using markup called the Document Type Definitions.

- This declaration is commonly referred to as "the DOCTYPE declaration" to differentiate it from a DTD.

The Document Type (DOCTYPE) Declarations:

- A document type declaration is placed in an XML document's prolog to say what DTD that document adheres to.
- It also specifies which element is the root element of the document.
- A document type *declaration* is not the same thing as a document type *definition*.
- A document type declaration must contain or refer to a document type definition, but a document type definition never contains a document type declaration.
- A document type declaration begins with <!DOCTYPE and ends with a >.
- A document type declaration has this basic form:

  <!DOCTYPE *name_of_root_element*
  SYSTEM "*URL of the external DTD subset*" [
  internal DTD subset
  ]>

- Here *name_of_root_element* is simply the name of the root element.
- The SYSTEM keyword indicates that what follows is a URL where the DTD is located.
- The square brackets enclose the internal subset of the DTD—that is, those declarations included inside the document itself.
- The DOCTYPE declaration consists of:

1. The usual XML tag delimiters ("<" and "?").
2. The exclamation mark ("!") that signifies a special XML declaration.
3. The DOCTYPE keyword.
4. The name of the document element (document_element).
5. One of two legal source keywords.
6. One of two DTD locations to associate an external DTD subset within a document.
7. Some additional declarations referring to the internal subset of the DTD.

Validating Against a DTD

- To be considered *valid,* an XML document must satisfy four criteria:

  1. It must be well formed.

  2. It must have a document type declaration.

  3. Its root element must be the one specified by the document type declaration.

  4. It must satisfy all the constraints indicated by the DTD specified by the document type declaration.

The Document Element Name

- The first variable of any DOCTYPE declaration is the name of the document element.
- This is required to be the root element of XML document.
- **Example:**

```
<?XML version = “1.0”?>
<!DOCTYPE Employee………..>



<Employee>



</Employee>
```

Basic DTD Declarations

- DTD declarations are delimited with the usual XML tag delimiters (“<” and “>”).
- Like DOCTYPE declarations, all DTD declarations are indicated by the use of the exclamation mark (“!”) followed by a keyword, and its specific parameters

  <! Keyword parameter1, paramenter2, …………, parameterN>

- There are four basic keywords used in DTD declarations
  1. ELEMENT
  2. ATTLIST
  3. ENTITY
  4. NOTATION

Element Type (ELEMENT) Declarations:

- Elements are described using the element type declaration.
- This declaration can have one of two different forms depending on the value of the category parameter

<!ELEMENT name Category>
<!ELEMENT name (Content_Model)>

Element content Categories

- There are 5 categories of element content:

Content Category	**Description**
ANY	Element type may contain any well formed XML data.
EMPTY	Element type may contain any text or child elements- only elements attributes are permitted.
Element	Element type contains only child elements no additional text is permitted.
Mixed	Element type may contain text and/or child element.
PCDATA	Element type may contain text (character data) only.

Content Models

- Content models are used to describe the structure and content of a given element type.
- The content may be:
  1. Character data (PCDATA content).
  2. One or more child element types (element-only content).
  3. A combination of the two (mixed content).

- The key difference between element content and mixed content is the use of the **#PCDATA** keyword.
- If present, the content model is either mixed or PCDATA.
- The absence of this keyword indicates element-only content.

Cardinality

- Cardinality operators define how many child elements may appear in a content model.
- There are four cardinality operators:

Operators	Description
[none]	The absence of a cardinality operator character indicates that one, and only one, instance of child element is allowed (required).
?	Zero or one element – optional singular element.
*	Zero or more element – optional element(s).
+	One or more child elements – required element(s).

/************** Example of Cardinality Operators ***************/

<!ELEMENT PersonName**(**
(Mr./Miss) **?**, First_Name **+**, Middle_Name *, Last_Name**)>**

The Attribute (ATTLIST) Declarations

- Attributes can be used to describe the meta-data or properties of the associated element.
- Element attributes are described using the attribute list declarations, also called ATTLIST declarations.
- This declaration has the usual DTD declarations format, using the ATTLIST keyword plus zero or more attribute definitions.

<! ATTLIST element_name attrName attrType attrDefault defaultValue>

The value is a predefined xml value. The default-value can be one of the following:

'Value'	-	The default value of the attribute
#REQUIRED	-	The attribute is required
#IMPLIED	-	The attribute is not required

#FIXED value - The attribute value is fixed

/************* Example of Attribute List Declarations ************/

```
<!ELEMENT Book EMPTY>
<!ATTLIST Book
   isbn     CDATA  #REQUIRED
   title    CDATA  #REQUIRED
   author  CDATA  #REQUIRED
   price   CDATA  #IMPLIED (optional)
>
```

Attribute Types

- There are 10 different types of attributes defined in XML 1.0 recommendation.

Attribute	Description
CDATA	Character Data (simple text string)
Enumerated values (Choice list)	Attribute must be one of a series that is explicitly defined in DTD.
ID	Attribute value is the unique identifier for this element instance
IDREF	A reference to the element with an ID attribute that has the same value as that of IDREF
IDREFS	A list of IDREFs delimited by white space
NMTOKEN	A name token – a text string that confirms to the XML name rules
NMTOKENS	A list of NMTOKENs delimited by white spaces
ENTITY	The name of a pre-defined entity
ENTITIES	A list of ENTITY name delimited by white spaces
NOTATION	Attribute value must be a notation type that is explicitly declared elsewhere in the DTD

XML Elements vs. Attributes

In XML, there are no rules about when to use attributes, and when to use child elements.

Use of Elements vs. Attributes

Data can be stored in child elements or in attributes.

```
<person gender="female">
  <firstname>Anna</firstname>
  <lastname>Smith</lastname>
</person>

<person>
  <gender>female</gender>
  <firstname>Anna</firstname>
  <lastname>Smith</lastname>
</person>
```

In the first example gender is an attribute. In the last, gender is a child element. Both examples provide the same information. There are no rules about when to use attributes, and when to use child elements. My experience is that attributes are handy in HTML, but in XML you should try to avoid them. Use child elements if the information feels like data.

I like to store data in child elements.

The following three XML documents contain exactly the same information:

A date attribute is used in the first example:

```
<note date="12/11/2002">
        <to>Tove</to>
        <from>Jani</from>
        <heading>Reminder</heading>
        <body>Don't forget me this weekend!</body>
</note>
```

A date element is used in the second example:

```
<note>
        <date>12/11/2002</date>
        <to>Tove</to>
        <from>Jani</from>
        <heading>Reminder</heading>
        <body>Don't forget me this weekend!</body>
</note>
```

An expanded date element is used in the third:

```
<note>
<date>
  <day>12</day>
  <month>11</month>
  <year>2002</year>
</date>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

Should you avoid using attributes?

Some of the problems with attributes are:

- attributes cannot contain multiple values (child elements can)
- attributes are not easily expandable (for future changes)
- attributes cannot describe structures (child elements can)
- attributes are more difficult to manipulate by program code
- attribute values are not easy to test against a DTD

DTD – Entities

Entities are variables used to define shortcuts to standard text or special characters. Entity references are references to entities. Entities can be declared internal or external.

An Internal Entity Declaration

Syntax

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

Example DTD:

```
<!ENTITY writer "Donald Duck.">
<!ENTITY copyright "Copyright W3Schools.">
```

XML example:

```
<author>&writer;&copyright;</author>
```

Note: An entity has three parts: an ampersand (&), an entity name, and a semicolon (;).

An External Entity Declaration

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

DTD Example:

```
<!ENTITY writer SYSTEM "http://www.w3schools.com/entities.dtd">
<!ENTITY copyright SYSTEM "http://www.w3schools.com/entities.dtd">
```

XML example:

```
<author>&writer;&copyright;</author>
```

/**************************** Example of DTD ***************************/**

```
<!DOCTYPE Employees[
<!ELEMENT Employees (employee +)>
<!ELEMENT employee (Name +, Position +, Address +)>
<!ATTLIST employee id ID #REQUIRED>
<!ELEMENT Name (First +, Middle ?, Last +)>
<!ELEMENT First (#PCDATA)>
<!ELEMENT Middle(#PCDATA)>
<!ELEMENT Last (#PCDATA)>
<!ELEMENT Position (#PCDATA)>
<!ELEMENT Address (Street ?, City +, State +, Zip *)>
<!ELEMENT Street (#PCDATA)>
<!ELEMENT City (#PCDATA)>
<!ELEMENT State (#PCDATA)>
<!ELEMENT Zip (#PCDATA)>
]>
```

Limitations of DTD

- Some limitations of DTD include:

1. DTD are not extensible, unlike XML itself.
2. Only one DTD may be associated with each XML document.
3. DTDs do not work well with XML namespaces.
4. Supports very weak data typing.
5. Limited content model descriptions.
6. No object oriented type object inheritance.
7. A document can override / ignore an external DTD using internal subset.
8. Non-XML syntax.
9. No DOM support.
10. Relatively few older, more expensive tools.
11. Very limited support to modularity and reuse.
12. Too simple ID attribute mechanism (no points to requirements, uniqueness scope etc.)

DTD - Examples from the Cloud

TV Schedule DTD

```
<!DOCTYPE TVSCHEDULE [
<!ELEMENT TVSCHEDULE (CHANNEL+)>
<!ELEMENT CHANNEL (BANNER,DAY+)>
<!ELEMENT BANNER (#PCDATA)>
<!ELEMENT DAY (DATE,(HOLIDAY|PROGRAMSLOT+)+)>
<!ELEMENT HOLIDAY (#PCDATA)>
<!ELEMENT DATE (#PCDATA)>
<!ELEMENT PROGRAMSLOT (TIME,TITLE,DESCRIPTION?)>
<!ELEMENT TIME (#PCDATA)>
<!ELEMENT TITLE (#PCDATA)>
<!ELEMENT DESCRIPTION (#PCDATA)>
<!ATTLIST TVSCHEDULE NAME CDATA #REQUIRED>
<!ATTLIST CHANNEL CHAN CDATA #REQUIRED>
<!ATTLIST PROGRAMSLOT VTR CDATA #IMPLIED>
<!ATTLIST TITLE RATING CDATA #IMPLIED>
<!ATTLIST TITLE LANGUAGE CDATA #IMPLIED>
]>
```

Newspaper Article DTD

```
 <!DOCTYPE NEWSPAPER [
<!ELEMENT NEWSPAPER (ARTICLE+)>
<!ELEMENT ARTICLE (HEADLINE,BYLINE,LEAD,BODY,NOTES)>
<!ELEMENT HEADLINE (#PCDATA)>
<!ELEMENT BYLINE (#PCDATA)>
<!ELEMENT LEAD (#PCDATA)>
<!ELEMENT BODY (#PCDATA)>
<!ELEMENT NOTES (#PCDATA)>
<!ATTLIST ARTICLE AUTHOR CDATA #REQUIRED>
<!ATTLIST ARTICLE EDITOR CDATA #IMPLIED>
<!ATTLIST ARTICLE DATE CDATA #IMPLIED>
<!ATTLIST ARTICLE EDITION CDATA #IMPLIED>
<!ENTITY NEWSPAPER "Vervet Logic Times">
<!ENTITY PUBLISHER "Vervet Logic Press">
<!ENTITY COPYRIGHT "Copyright 1998 Vervet Logic Press">
]>
```

Product Catalog DTD

```
<!DOCTYPE CATALOG [
<!ENTITY AUTHOR "John Doe">
<!ENTITY COMPANY "JD Power Tools, Inc.">
<!ENTITY EMAIL "jd@jd-tools.com">
<!ELEMENT CATALOG (PRODUCT+)>
<!ELEMENT PRODUCT
(SPECIFICATIONS+,OPTIONS?,PRICE+,NOTES?)>
<!ATTLIST PRODUCT
NAME CDATA #IMPLIED
CATEGORY (HandTool|Table|Shop-Professional) "HandTool"
PARTNUM CDATA #IMPLIED
PLANT (Pittsburgh|Milwaukee|Chicago) "Chicago"
INVENTORY (InStock|Backordered|Discontinued) "InStock">

<!ELEMENT SPECIFICATIONS (#PCDATA)>
<!ATTLIST SPECIFICATIONS
WEIGHT CDATA #IMPLIED
POWER CDATA #IMPLIED>
<!ELEMENT OPTIONS (#PCDATA)>
<!ATTLIST OPTIONS
FINISH (Metal|Polished|Matte) "Matte"
ADAPTER (Included|Optional|NotApplicable) "Included"
CASE (HardShell|Soft|NotApplicable) "HardShell">
<!ELEMENT PRICE (#PCDATA)>
<!ATTLIST PRICE
MSRP CDATA #IMPLIED
WHOLESALE CDATA #IMPLIED
STREET CDATA #IMPLIED
SHIPPING CDATA #IMPLIED>
        <!ELEMENT NOTES (#PCDATA)>
]>
```

-wts

XML Schema (XSD)

What is Schema?

- XML Schema is an XML-based alternative to DTDs.
- An XML Schema describes the structure of an XML document.
- The XML Schema language is also referred to as XML Schema Definition (XSD).

The purpose of an XML Schema is to define the legal building blocks of an XML document, just like a DTD.

An XML Schema defines:

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

'XML Schemas are the Successors of DTDs'

We think that very soon XML Schemas will be used in most Web applications as a replacement for DTDs. Here are some reasons:

- XML Schemas are extensible to future additions
- XML Schemas are richer and more powerful than DTDs
- XML Schemas are written in XML
- XML Schemas support data types

- XML Schemas support namespaces
- XML Schema is a W3C Standard

'XML Schema became a W3C Recommendation 02. May 2001.' 'XML Schemas are much more powerful than DTDs.'

XML Schemas Support Data Types

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

With support for data types:

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

'XML Schemas use XML Syntax'.

Another great strength about XML Schemas is that they are written in XML. *Some benefits of that XML Schema are written in XML:*

- You don't have to learn a new language
- You can use your XML editor to edit your Schema files
- You can use your XML parser to parse your Schema files
- You can manipulate your Schema with the XML DOM
- You can transform your Schema with XSLT
- XML Schemas Secure Data Communication

When sending data from a sender to a receiver, it is essential that both parts have the same "expectations" about the content.

With XML Schemas, the sender can describe the data in a way that the receiver will understand.

A date like: "03-11-2004" will, in some countries, be interpreted as 3, November and in other countries as 11, March. However, XML elements with a data type like this: <date type="date">2004-03-11</date> Ensures a mutual understanding of the content, because the XML data type "date" requires the format "YYYY-MM-DD".

'XML Schemas are extensible, because they are written in XML.'

With an extensible Schema definition you can:

- Reuse your Schema in other Schemas
- Create your own data types derived from the standard types
- Reference multiple schemas in the same document

Well-Formed XML

A well-formed XML document is a document that conforms to the XML syntax rules, like:

- it must begin with the XML declaration
- it must have one unique root element
- start-tags must have matching end-tags
- elements are case sensitive
- all elements must be closed
- all elements must be properly nested
- all attribute values must be quoted

- entities must be used for special characters

Even if documents are well-formed they can still contain errors, and those errors can have serious consequences.

XML Namespace

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.

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

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

There are really two fundamental needs for namespaces:

1. To disambiguate between two elements that happen to share the same name
2. To group elements relating to a common idea together

[*SELF STUDY FOR MORE*]

A Simple XML Document

Look at this simple XML document called "note.xml":

```
<?xml version="1.0"?>
<note>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

A DTD File

The following example is a DTD file called "note.dtd" that defines the elements of the XML document above ("note.xml"):

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

The first line defines the note element to have four child elements: "to, from, heading, body".
Line 2-5 defines the to, from, heading, body elements to be of type "#PCDATA".

An XML Schema

The following example is an XML Schema file called "note.xsd" that defines the elements of the XML document above ("note.xml"):

```
 <?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.w3schools.com"
xmlns="http://www.w3schools.com"
elementFormDefault="qualified">
<xs:element name="note">
   <xs:complexType>
     <xs:sequence>
          <xs:element name="to" type="xs:string"/>
          <xs:element name="from" type="xs:string"/>
          <xs:element name="heading" type="xs:string"/>
          <xs:element name="body" type="xs:string"/>
     </xs:sequence>
   </xs:complexType>
</xs:element>
</xs:schema>
```

The note element is a complex type because it contains other elements. The other elements (to, from, heading, body) are simple types because they do not contain other elements. You will learn more about simple and complex types in the following chapters.

A Reference to a DTD

This XML document has a reference to a DTD:

```
<?xml version="1.0"?>
<!DOCTYPE note SYSTEM "http://www.w3schools.com/dtd/note.dtd">
<note>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

A Reference to an XML Schema

This XML document has a reference to an XML Schema:

```
<?xml version="1.0"?>
<note
xmlns="http://www.w3schools.com"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.w3schools.com note.xsd">
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

'The <schema> element is the root element of every XML Schema.'

The <schema> Element

The <schema> element is the root element of every XML Schema:

```
<?xml version="1.0"?>
<xs:schema>
```

```
...
</xs:schema>
```

The <schema> element may contain some attributes. A schema declaration often looks something like this:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.w3schools.com"
xmlns="http://www.w3schools.com"
elementFormDefault="qualified">
...
</xs:schema>
```

The following fragment:xmlns:xs="http://www.w3.org/2001/XMLSchema"

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

This fragment:targetNamespace="http://www.w3schools.com"

Indicates that the elements defined by this schema (note, to, from, heading, body.) come from the "http://www.w3schools.com" namespace.

This fragment:xmlns="http://www.w3schools.com"

Indicates that the default namespace is "http://www.w3schools.com".

This fragment:elementFormDefault="qualified"

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

Referencing a Schema in an XML Document

This XML document has a reference to an XML Schema:

```
<?xml version="1.0"?>
<note xmlns="http://www.w3schools.com"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.w3schools.com note.xsd">
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

The following fragment:xmlns="http://www.w3schools.com"

Specifies the default namespace declaration. This declaration tells the schema-validator that all the elements used in this XML document are declared in the "http://www.w3schools.com" namespace.

Once you have the XML Schema Instance namespace available:
xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance you can use the schemaLocation attribute.

And next is the location of the XML schema to use for that namespace:
xsi:schemaLocation=http://www.w3schools.com note.xsd

What is a Simple Element?

A simple element is an XML element that can contain only text. It cannot contain any other elements or attributes.

You can add restrictions (facets) to a data type in order to limit its content, or you can require the data to match a specific pattern.

Defining a Simple Element

The syntax for defining a simple element is:

```
<xs:element name="xxx" type="yyy"/>
```

Where xxx is the name of the element and yyy is the data type of the element.

The most common data types are:

- xs:string
- xs:decimal
- xs:integer
- xs:boolean
- xs:date
- xs:time

Example:

Here are some XML elements:

```
<lastname>Refsnes</lastname>
<age>36</age>
<dateborn>1970-03-27</dateborn>
```

And

Here are the corresponding simple element definitions:

```
<xs:element name="lastname" type="xs:string"/>
<xs:element name="age" type="xs:integer"/>
<xs:element name="dateborn" type="xs:date"/>
```

Default and Fixed Values for Simple Elements

Simple elements may have a default value OR a fixed value specified. A default value is automatically assigned to the element when no other value is specified.

In the following example the default value is "red":

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

A fixed value is also automatically assigned to the element, and you cannot specify another value.

In the following example the fixed value is "red":

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

What is an Attribute?

Simple elements cannot have attributes. If an element has attributes, it is considered to be of a complex type. But the attribute itself is always declared as a simple type.

How to Define an Attribute?

'All attributes are declared as simple types.'

The syntax for defining an attribute is:

```
<xs:attribute name="xxx" type="yyy"/>
```

Where xxx is the name of the attribute and yyy specifies the data type of the attribute.

XML Schema has a lot of built-in data types. The most common types are:

- xs:string
- xs:decimal
- xs:integer
- xs:boolean
- xs:date
- xs:time

Here is an XML element with an attribute:

```
<lastname lang="EN">Smith</lastname>
```

And here is the corresponding attribute definition:

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

Default and Fixed Values for Attributes

Attributes may have a default value OR a fixed value specified.

In the following example the default value is "EN":

```
<xs:attribute name="lang" type="xs:string" default="EN"/>
```

In the following example the fixed value is "EN":

```
<xs:attribute name="lang" type="xs:string" fixed="EN"/>
```

Optional and Required Attributes

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

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

Restrictions on Content (Facet)

With XML Schemas, you can also add your own restrictions to your XML elements and attributes. These restrictions are called facets. Restrictions are used to define acceptable values for XML elements or attributes. Restrictions on XML elements are called facets.

Restrictions on Values

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

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

Restrictions on a Set of Values

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

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

Restrictions on a Series of Values

To limit the content of an XML element to define a series of numbers or letters that can be used, we would use the pattern constraint.

The example below defines an element called "letter" with a restriction. The only acceptable value is ONE of the LOWERCASE letters from a to z:

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

The next example defines an element called "initials" with a restriction. The only acceptable value is THREE of the UPPERCASE letters from a to z:

```
<xs:element name="initials">
<xs:simpleType>
  <xs:restriction base="xs:string">
    <xs:pattern value="[A-Z][A-Z][A-Z]"/>
  </xs:restriction>
</xs:simpleType>
</xs:element>
```

The next example also defines an element called "initials" with a restriction. The only acceptable value is THREE of the LOWERCASE OR UPPERCASE letters from a to z:

```
<xs:element name="initials">
<xs:simpleType>
  <xs:restriction base="xs:string">
    <xs:pattern value="[a-zA-Z][a-zA-Z][a-zA-Z]"/>
  </xs:restriction>
</xs:simpleType>
</xs:element>
```

The next example defines an element called "choice" with a restriction. The only acceptable value is ONE of the following letters: x, y, OR z:

```
<xs:element name="choice">
<xs:simpleType>
  <xs:restriction base="xs:string">
    <xs:pattern value="[xyz]"/>
  </xs:restriction>
</xs:simpleType>
</xs:element>
```

The next example defines an element called "prodid" with a restriction. The only acceptable value is FIVE digits in a sequence, and each digit must be in a range from 0 to 9:

```
<xs:element name="prodid">
<xs:simpleType>
  <xs:restriction base="xs:integer">
    <xs:pattern value="[0-9][0-9][0-9][0-9][0-9]"/>
  </xs:restriction>
</xs:simpleType>
</xs:element>
```

Other Restrictions on a Series of Values

The example below defines an element called "letter" with a restriction. The acceptable value is zero or more occurrences of lowercase letters from a to z:

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

The next example also defines an element called "letter" with a restriction. The acceptable value is one or more pairs of letters, each pair consisting of a lower case letter followed by an upper case letter. For example, "sToP" will be validated by this pattern, but not "Stop" or "STOP" or "stop":

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

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

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

The next example defines an element called "password" with a restriction. There must be exactly eight characters in a row and those characters must be lowercase or uppercase letters from a to z, or a number from 0 to 9:

```
<xs:element name="password">
<xs:simpleType>
  <xs:restriction base="xs:string">
    <xs:pattern value="[a-zA-Z0-9]{8}"/>
  </xs:restriction>
</xs:simpleType>
</xs:element>
```

Restrictions on Length

To limit the length of a value in an element, we would use the length, maxLength, and minLength constraints. *This example defines an element called "password" with a restriction. The value must be exactly eight characters:*

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

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

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

Restrictions for Data types

- ***Enumeration -*** *Defines a list of acceptable values.*
- ***length*** *- Specifies the exact number of characters or list items allowed. Must be equal to or greater than zero.*
- ***maxExclusive -*** *Specifies the upper bounds for numeric values (the value must be less than this value).*
- ***maxInclusive -*** *Specifies the upper bounds for numeric values (the value must be less than or equal to this value).*
- ***maxLength -*** *Specifies the maximum number of characters or list items allowed. Must be equal to or greater than zero.*
- ***minExclusive -*** *Specifies the lower bounds for numeric values (the value must be greater than this value).*
- ***minInclusive -*** *Specifies the lower bounds for numeric values (the value must be greater than or equal to this value).*

- ***minLength*** *- Specifies the minimum number of characters or list items allowed. Must be equal to or greater than zero.*
- ***pattern*** *- Defines the exact sequence of characters that are acceptable.*

XSD Complex Elements
A complex element contains other elements and/or attributes.

What is a Complex Element?
A complex element is an XML element that contains other elements and/or attributes.
There are four kinds of complex elements:

1. empty elements
2. elements that contain only other elements
3. elements that contain only text
4. elements that contain both other elements and text

'Each of these elements may contain attributes as well!'

Examples of Complex Elements
A complex XML element, "product", which is empty:

```
<product pid="1345"/>
```

A complex XML element, "employee", which contains only other elements:

```
<employee>
        <firstname>John</firstname>
        <lastname>Smith</lastname>
</employee>
```

A complex XML element, "food", which contains only text:

```
<food type="dessert">Ice cream</food>
```

A complex XML element, "description", which contains both elements and text:

```
<description> It happened on <date lang="norwegian">03.03.99</date> ....</description>
```

How to Define a Complex Element

Look at this complex XML element, "employee", which contains only other elements:

```
<employee>
        <firstname>John</firstname>
        <lastname>Smith</lastname>
</employee>
```

We can define a complex element in an XML Schema two different ways:

The "employee" element can be declared directly by naming the element, like this:

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

The child elements, "firstname" and "lastname", are surrounded by the <sequence> indicator. This means that the child elements must appear in the same order as they are declared.

XSD Complex Empty Elements

An empty complex element cannot have contents, only attributes.

Complex Empty Elements

An empty XML element: <product prodid="1345" />

The "product" element above has no content at all. To define a type with no content, we must define a type that allows only elements in its content, but we do not actually declare any elements, like this:

```
<xs:element name="product">
  <xs:complexType>
    <xs:complexContent>
```

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

In the example above, we define a complex type with a complex content. The **complexContent** element signals that we intend to restrict or extend the content model of a complex type, and the restriction of integer declares one attribute but does not introduce any element content.

However, it is possible to declare the "product" element more compactly, like this:

```
<xs:element name="product">
  <xs:complexType>
    <xs:attribute name="prodid" type="xs:positiveInteger"/>
  </xs:complexType>
</xs:element>
```

XSD Complex Type - Elements Only

An "elements-only" complex type contains an element that contains only other elements.

An XML element, "person", that contains only other elements:

```
<person>
<firstname>John</firstname>
<lastname>Smith</lastname>
</person>
```

You can define the "person" element in a schema, like this:

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

Here is an example:

```
<shoesize country="france">35</shoesize>
```

The following example declares a complexType, "shoesize". The content is defined as an integer value, and the "shoesize" element also contains an attribute named "country":

```
<xs:element name="shoesize">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base="xs:integer">
        <xs:attribute name="country" type="xs:string" />
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>
```

XSD Complex Types with Mixed Content
A mixed complex type element can contain attributes, elements, and text.

Complex Types with Mixed Content

An XML element, "letter", that contains both text and other elements:

```
<letter>
Dear Mr.<name>John Smith</name>.
Your order <orderid>1032</orderid>
will be shipped on <shipdate>2001-07-13</shipdate>.
</letter>
```

The following schema declares the "letter" element:

```
<xs:element name="letter">
  <xs:complexType mixed="true">
```

```
      <xs:sequence>
        <xs:element name="name" type="xs:string"/>
        <xs:element name="orderid" type="xs:positiveInteger"/>
        <xs:element name="shipdate" type="xs:date"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
```

Note: To enable character data to appear between the child-elements of "letter", the mixed attribute must be set to "true".

XSD Complex Types Indicators

We can control HOW elements are to be used in documents with indicators.

Order indicators:

1. All
2. Choice
3. Sequence

Occurrence indicators:

1. maxOccurs
2. minOccurs

All Indicators

The <all> indicator specifies that the child elements can appear in any order, and that each child element must occur only once:

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

When using the <all> indicator you can set the <minOccurs> indicator to 0 or 1 and the <maxOccurs> indicator can only be set to 1.

Choice Indicator

The <choice> indicator specifies that either one child element or another can occur:

```
<xs:element name="person">
 <xs:complexType>
  <xs:choice>
   <xs:element name="employee" type="employee"/>
   <xs:element name="member" type="member"/>
  </xs:choice>
 </xs:complexType>
</xs:element>
```

Sequence Indicator

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

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

Occurrence Indicators

Occurrence indicators are used to define how often an element can occur.

For all default value for maxOccurs and minOccurs is 1.

maxOccurs Indicator

The <maxOccurs> indicator specifies the maximum number of times an element can occur:

```
<xs:element name="person">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="full_name" type="xs:string"/>
      <xs:element name="child_name" type="xs:string" maxOccurs="10"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

The example above indicates that the "child_name" element can occur a minimum of one time (the default value for minOccurs is 1) and a maximum of ten times in the "person" element.

minOccurs Indicator

The <minOccurs> indicator specifies the minimum number of times an element can occur:

```
<xs:element name="person">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="full_name" type="xs:string"/>
      <xs:element name="child_name" type="xs:string"
      maxOccurs="10" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

The example above indicates that the "child_name" element can occur a minimum of zero times and a maximum of ten times in the "person" element.

To allow an element to appear an unlimited number of times, use the maxOccurs="unbounded" statement.

A working example: **An XML file called "Myfamily.xml":**

```
<?xml version="1.0" encoding="ISO-8859-1"?>
        <persons xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="family.xsd">
```

```
    <person>
        <full_name>Hege Refsnes</full_name>
        <child_name>Cecilie</child_name>
    </person>
    <person>
        <full_name>Tove Refsnes</full_name>
        <child_name>Hege</child_name>
        <child_name>Stale</child_name>
        <child_name>Jim</child_name>
        <child_name>Borge</child_name>
    </person>
    <person>
      <full_name>Stale Refsnes</full_name>
    </person>
    </persons>
```

The XML file above contains a root element named "persons". Inside this root element we have defined three "person" elements. Each "person" element must contain a "full_name" element and it can contain up to five "child_name" elements.

Here is the schema file "family.xsd":

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
<xs:element name="persons">
 <xs:complexType>
  <xs:sequence>
   <xs:element name="person" maxOccurs="unbounded">
    <xs:complexType>
     <xs:sequence>
      <xs:element name="full_name" type="xs:string"/>
      <xs:element name="child_name" type="xs:string"
      minOccurs="0" maxOccurs="5"/>
     </xs:sequence>
    </xs:complexType>
   </xs:element>
   </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:schema>
```

An XSD Example

This chapter will demonstrate how to write an XML Schema. You will also learn that a schema can be written in different ways.

An XML Document

Let's have a look at this XML document called "shiporder.xml":

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<shiporder orderid="889923"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="shiporder.xsd">
 <orderperson>John Smith</orderperson>
 <shipto>
  <name>Ola Nordmann</name>
  <address>Langgt 23</address>
  <city>4000 Stavanger</city>
  <country>Norway</country>
 </shipto>
 <item>
  <title>Empire Burlesque</title>
  <note>Special Edition</note>
  <quantity>1</quantity>
  <price>10.90</price>
 </item>
 <item>
  <title>Hide your heart</title>
  <quantity>1</quantity>
  <price>9.90</price>
 </item>
</shiporder>
```

The XML document above consists of a root element, "shiporder", that contains a required attribute called "orderid". The "shiporder" element contains three different child elements: "orderperson", "shipto" and "item". The "item" element appears twice, and it contains a "title", an optional "note" element, a "quantity", and a "price" element.

Create an XML Schema

Here is the complete listing of the schema file called "shiporder.xsd":

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="shiporder">
 <xs:complexType>
  <xs:sequence>
   <xs:element name="orderperson" type="xs:string"/>
   <xs:element name="shipto">
    <xs:complexType>
     <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="address" type="xs:string"/>
      <xs:element name="city" type="xs:string"/>
      <xs:element name="country" type="xs:string"/>
     </xs:sequence>
    </xs:complexType>
   </xs:element>
   <xs:element name="item" maxOccurs="unbounded">
    <xs:complexType>
     <xs:sequence>
      <xs:element name="title" type="xs:string"/>
      <xs:element name="note" type="xs:string" minOccurs="0"/>
      <xs:element name="quantity" type="xs:positiveInteger"/>
      <xs:element name="price" type="xs:decimal"/>
     </xs:sequence>
    </xs:complexType>
   </xs:element>
  </xs:sequence>
  <xs:attribute name="orderid" type="xs:string" use="required"/>
 </xs:complexType>
</xs:element>
</xs:schema>
```

XSD Date and Time Data Types

Date and time data types are used for values that contain date and time.

Date Data Type

The date data type is used to specify a date. The date is specified in the following form "YYYY-MM-DD" where:

YYYY indicates the year

MM indicates the month

DD indicates the day

Note: All components are required!

The following is an example of a date declaration in a schema:

```
<xs:element name="start" type="xs:date"/>
```

An element in your document might look like this:

```
<start>2002-09-24</start>
```

Time Data Type

The time data type is used to specify a time. The time is specified in the following form "hh:mm:ss" where:

hh indicates the hour

mm indicates the minute

ss indicates the second

Note: All components are required!

The following is an example of a time declaration in a schema:

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

An element in your document might look like this:

```
<start>09:00:00</start>
```

Or

It might look like this:

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

DateTime Data Type

The dateTime data type is used to specify a date and a time. The dateTime is specified in the following form "YYYY-MM-DDThh:mm:ss" where:

YYYY indicates the year

MM indicates the month

DD indicates the day

T indicates the start of the required time section

hh indicates the hour

mm indicates the minute

ss indicates the second

Note: All components are required!

The following is an example of a dateTime declaration in a schema:

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

An element in your document might look like this:

```
<startdate>2002-05-30T09:00:00</startdate>
```

Or

It might look like this:

```
<startdate>2002-05-30T09:30:10.5</startdate>
```

Duration Data Type

The duration data type is used to specify a time interval.

The time interval is specified in the following form "PnYnMnDTnHnMnS" where:

P indicates the period (required)

nY indicates the number of years

nM indicates the number of months

nD indicates the number of days

T indicates the start of a time section (required if you are going to specify hours, minutes, or seconds)

nH indicates the number of hours

nM indicates the number of minutes

nS indicates the number of seconds

The following is an example of a duration declaration in a schema:

```
<xs:element name="period" type="xs:duration"/>
```

An element in your document might look like this:

```
<period>P5Y</period>
```

The example above indicates a period of five years.

Or

It might look like this:

```
<period>P5Y2M10D</period>
```

The example above indicates a period of five years, two months, and 10 days.

Or

It might look like this:

```
<period>P5Y2M10DT15H</period>
```

The example above indicates a period of five years, two months, 10 days, and 15 hours.

Or

It might look like this :

```
<period>PT15H</period>
```

The example above indicates a period of 15 hours.

XSD Numeric Data Types
Decimal data types are used for numeric values.

Decimal Data Type

The decimal data type is used to specify a numeric value. The following is an example of a decimal declaration in a schema:

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

An element in your document might look like this:

```
<prize>999.50</prize>
```

Note: The maximum number of decimal digits you can specify is 18.

Integer Data Type

The integer data type is used to specify a numeric value without a fractional component. The following is an example of an integer declaration in a schema:

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

An element in your document might look like this:

```
<prize>999</prize>
```

Or

```
<prize>+999</prize>
```

Or

```
<prize>-999</prize>
```

Or

```
<prize>0</prize>
```

negativeInteger - An integer containing only negative values (.., -2, -1.)

nonNegativeInteger - An integer containing only non-negative values (0, 1, 2, ..)

nonPositiveInteger - An integer containing only non-positive values (.., -2, -1, 0)

positiveInteger - An integer containing only positive values (1, 2, ..)

Boolean Data Type

The boolean data type is used to specify a true or false value. The following is an example of a boolean declaration in a schema:

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

An element in your document might look like this:

```
<prize disabled="true">999</prize>
```

Legal values for boolean are true, false, 1 (which indicates true), and 0 (which indicates false).

AnyURI Data Type

The anyURI data type is used to specify a URI. The following is an example of an anyURI declaration in a schema:

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

An element in your document might look like this:

```
<pic src="http://www.w3schools.com/images/smiley.gif" />
```

Note: If a URI has spaces, replace them with %20.

Displaying XML with CSS

With CSS (Cascading Style Sheets) you can add display information to an XML document. It is possible to use CSS to format an XML document.

Sample XML File

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/css" href="cd_catalog.css"?>
<CATALOG>
 <CD>
  <TITLE>Empire Burlesque</TITLE>
  <ARTIST>Bob Dylan</ARTIST>
  <COUNTRY>USA</COUNTRY>
  <COMPANY>Columbia</COMPANY>
  <PRICE>10.90</PRICE>
  <YEAR>1985</YEAR>
 </CD>
 <CD>
  <TITLE>Hide your heart</TITLE>
  <ARTIST>Bonnie Tyler</ARTIST>
  <COUNTRY>UK</COUNTRY>
  <COMPANY>CBS Records</COMPANY>
  <PRICE>9.90</PRICE>
  <YEAR>1988</YEAR>
 </CD>
.
.
.
</CATALOG>
```

cd_catalog.css

```
CATALOG { background-color: #ffffff; width: 100%; }
CD { display: block; margin-bottom: 30pt; margin-left: 0; }
```

```
TITLE { color: #FF0000; font-size: 20pt; }
ARTIST { color: #0000FF; font-size: 20pt; }
COUNTRY,PRICE,YEAR,COMPANY { display: block; color: #000000; margin-left: 20pt; }
```

'All CSS properties can be used for formatting. REFER 'CSS' FOR MORE.'

Formatting XML with CSS is not the most common method. W3C recommends using XSLT instead.

XSL for XML: a happy marriage!

XSL - stands for eXtensible Stylesheet Language. The World Wide Web Consortium (W3C) started to develop XSL because there was a need for an XML-based Style sheet Language.

CSS = HTML Style Sheets

HTML uses predefined tags and the meanings of the tags are well understood. The <table> element in HTML defines a table - and a browser knows how to display it. Adding styles to HTML elements is simple. Telling a browser to display an element in a special font or color is easy with CSS.

XSL = XML Style Sheets

XML does not use predefined tags (we can use any tag-names we like), and the meaning of these tags is not well understood. A <table> element could mean an HTML table, a piece of furniture, or something else - and a browser does not know how to display it.

XSL describes how the XML document should be displayed! XSL is more Than a Style Sheet Language

1. XSLT - a language for transforming XML documents
2. XPath - a language for navigating in XML documents
3. XSL-FO - a language for formatting XML documents

XSLT - the language for transforming XML documents. XSLT is a language for transforming XML documents into XHTML documents or to other XML documents. XPath is a language for navigating in XML documents.

What is XSLT?

- XSLT stands for XSL Transformations
- XSLT is the most important part of XSL
- XSLT transforms an XML document into another XML document
- XSLT uses XPath to navigate in XML documents
- XSLT is a W3C Recommendation
- XSLT = XSL Transformations

XSLT is used to transform an XML document into another XML document, or another type of document that is recognized by a browser, like HTML and XHTML. Normally XSLT does this by transforming each XML element into an (X) HTML element.

With XSLT you can add/remove elements and attributes to or from the output file. You can also rearrange and sort elements, perform tests and make decisions about which elements to hide and display, and a lot more.

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

How does it Works?

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

XSLT is a W3C Recommendation; it became a W3C Recommendation 16. November 1999.

XSLT Browsers

Nearly all-major browsers have support for XML and XSLT.

- Mozilla Firefox
- Google Chrome
- Safari
- Netscape

- Opera
- Internet Explorer

Correct Style Sheet Declaration

The root element that declares the document to be an XSL style sheet is

<xsl:stylesheet> or <xsl:transform>.

Note: <xsl:stylesheet> and <xsl:transform> are completely synonymous and either can be used!

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

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

OR

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

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

The **xmlns:xsl="http://www.w3.org/1999/XSL/Transform"** points to the official W3C XSLT namespace. If you use this namespace, you must also include the attribute **version="1.0".**

Start with a Raw XML Document

We want to transform the following XML document ("cdcatalog.xml") into XHTML:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<catalog>
  <cd>
    <title>Empire Burlesque</title>
    <artist>Bob Dylan</artist>
    <country>USA</country>
    <company>Columbia</company>
    <price>10.90</price>
    <year>1985</year>
  </cd>
```

```
.
.
.
</catalog>
```

Then you create an XSL Style Sheet ("cdcatalog.xsl") with a transformation template:

```
<?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 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>
```

Add the XSL style sheet reference to your XML document ("cdcatalog.xml"):

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="cdcatalog.xsl"?>
<catalog>
  <cd>
    <title>Empire Burlesque</title>
    <artist>Bob Dylan</artist>
    <country>USA</country>
    <company>Columbia</company>
    <price>10.90</price>
    <year>1985</year>
  </cd>
.
.
.
</catalog>
```

The <xsl:template> Element

The <xsl:template> element is used to build templates. The match attribute is used to associate a template with an XML element. The match attribute can also be used to define a template for the entire XML document. The value of the match attribute is an XPath expression (i.e. match="/" defines the whole document).

```
<?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>.</td>
      <td>.</td>
    </tr>
  </table>
 </body>
 </html>
</xsl:template>
</xsl:stylesheet>
```

Since an XSL style sheet is an XML document itself, it always begins with the XML declaration:

<?xml version="1.0" ?>.

The next element, <xsl:stylesheet>, defines that this document is an XSLT style sheet document (along with the version number and XSLT namespace attributes).

The <xsl:template> element defines a template. The match="/" attribute associates the template with the root of the XML source document. The content inside the <xsl:template> element defines some HTML to write to the output. The last two lines define the end of the template and the end of the style sheet.

The result of the transformation above will look like this:

```
My CD Collection
Title          Artist
.              .
```

The result from this example was a little disappointing, because no data was copied from the XML document to the output.

XSLT <xsl:value-of> Element

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

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

Note: The value of the select attribute is an XPath expression. An XPath expression works like navigating a file system; where a forward slash (/) selects subdirectories.

The result of the transformation above will look like this:

My CD Collection

Title	**Artist**
Empire Burlesque	Bob Dylan

XSLT <xsl:for-each> Element

The <xsl:for-each> element allows you to do looping in XSLT.

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

```
<?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>
```

The value of the select attribute is an XPath expression. An XPath expression works like navigating a file system; where a forward slash (/) selects subdirectories.

The result of the transformation above will look like this:

My CD Collection

Title	**Artist**
Empire Burlesque	Bob Dylan
Hide your heart	Bonnie Tyler
Greatest Hits	Dolly Parton
Still got the blues	Gary More
Eros	Eros Ramazzotti
One night only	Bee Gees

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)

&lt; less than

&gt; greater than

```
<?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[artist='Bob Dylan']">
    <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>
```

The result of the transformation above will look like this:

```
My CD Collection
Title               Artist
Empire Burlesque    Bob Dylan
```

XSLT <xsl:sort> Element

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

To sort the output, simply add an <xsl:sort> element inside the <xsl:for-each> element in the XSL file:

```
<?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:sort select="artist"/>
     <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>
```

The select attribute indicates what XML element to sort on.

The result of the transformation above will look like this:

My CD Collection

Title	**Artist**
Romanza	Andrea Bocelli
One night only	Bee Gees
Empire Burlesque	Bob Dylan
Hide your heart	Bonnie Tyler
The very best of	Cat Stevens
Greatest Hits	Dolly Parton
Sylvias Mother	Dr.Hook
…	…

XSLT <xsl:if> Element

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:

```
<?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 value of the required test attribute contains the expression to be evaluated.

The code above will only output the title and artist elements of the CDs that has a price that is higher than 10.

The result of the transformation above will look like this:

My CD Collection

Title	**Artist**
Empire Burlesque	Bob Dylan
Still got the blues	Gary Moore
One night only	Bee Gees
Romanza	Andrea Bocelli
Black Angel	Savage Rose

XSLT <xsl:choose> Element

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

Syntax

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

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

```
<?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>
```

The code above will add a pink background-color to the "Artist" column WHEN the price of the CD is higher than 10.

The result of the transformation will look like this:

My CD Collection

Title	**Artist**
Empire Burlesque	Bob Dylan
Hide your heart	Bonnie Tyler
Greatest Hits	Dolly Parton
Still got the blues	Gary Moore
Eros	Eros Ramazzotti
One night only	Bee Gees
Sylvias Mother	Dr.Hook
Maggie May	Rod Stewart
Romanza	Andrea Bocelli

ASSIGNMENT 7

XML DOM

What is the 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.

The XML DOM defines a standard way for accessing and manipulating XML documents. The DOM presents an XML document as a tree-structure. For anyone working with XML, knowing the XML DOM is a must.

The XML DOM is:

- A standard object model for XML
- A standard programming interface for XML
- Platform- and language-independent
- A W3C standard

The XML DOM defines the **objects and properties** of all XML elements, and the **methods** (interface) to access them. In other words: **The XML DOM is a standard for how to get, change, add, or delete XML elements.**

In the DOM, everything in an XML document is a node.

DOM Nodes

According to the DOM, everything in an XML document is a **node**.

The DOM says:

- The entire document is a document node
- Every XML element is an element node
- The text in the XML elements are text nodes
- Every attribute is an attribute node
- Comments are comment nodes

DOM Example

Look at the following XML file (books.xml):

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<bookstore>
 <book category="cooking">
  <title lang="en">Everyday Italian</title>
  <author>Giada De Laurentiis</author>
  <year>2005</year>
  <price>30.00</price>
 </book>
 <book category="children">
  <title lang="en">Harry Potter</title>
  <author>J K. Rowling</author>
  <year>2005</year>
  <price>29.99</price>
 </book>
 <book category="web">
  <title lang="en">XQuery Kick Start</title>
  <author>James McGovern</author>
  <author>Per Bothner</author>
  <author>Kurt Cagle</author>
  <author>James Linn</author>
  <author>Vaidyanathan Nagarajan</author>
  <year>2003</year>
  <price>49.99</price>
 </book>
 <book category="web" cover="paperback">
  <title lang="en">Learning XML</title>
  <author>Erik T. Ray</author>
  <year>2003</year>
  <price>39.95</price>
 </book>
</bookstore>
```

The root node in the XML above is named <bookstore>. All other nodes in the document are contained within <bookstore>.

The root node <bookstore> holds four <book> nodes.

The first <book> node holds four nodes: <title>, <author>, <year>, and <price>, which contains one text node each, "Everyday Italian", "Giada De Laurentiis", "2005", and "30.00".

Text is Always Stored in Text Nodes

A common error in DOM processing is to expect an element node to contain text.

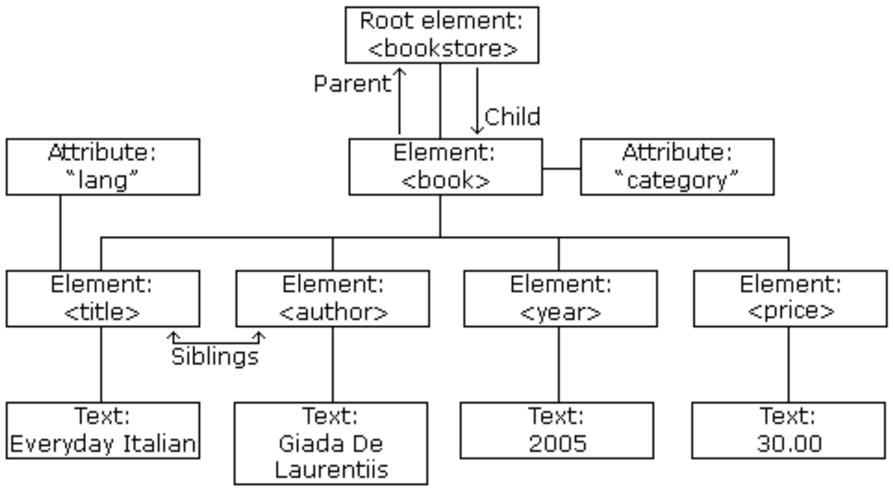
However, the text of an element node is stored in a text node. In this example: **<year>2005</year>**, the element node <year>, holds a text node with the value "2005". "2005" is **not** the value of the <year> element!

The XML DOM views an XML document as a node-tree. All the nodes in the tree have a relationship to each other.

The XML DOM Node Tree

The XML DOM views an XML document as a tree-structure. The tree structure is called a **node-tree.** All nodes can be accessed through the tree. Their contents can be modified or deleted, and new elements can be created.

The node tree shows the set of nodes, and the connections between them. The tree starts at the root node and branches out to the text nodes at the lowest level of the tree:



The image above represents the XML file books.xml.

Node Parents, Children, and Siblings

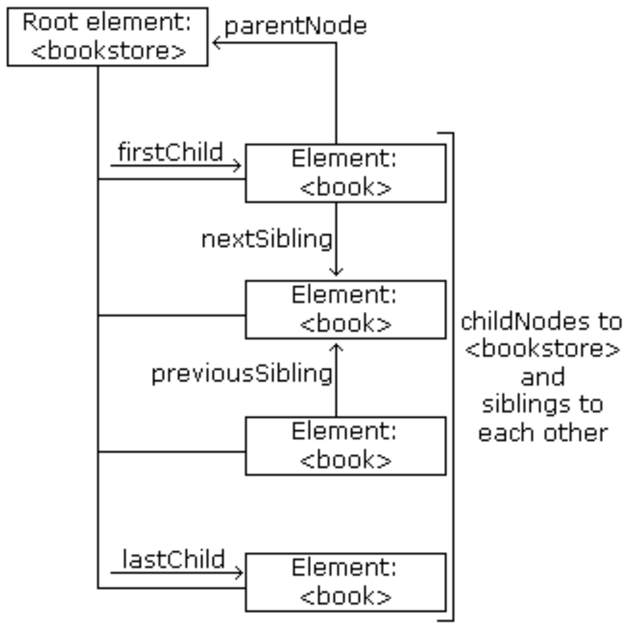
The nodes in the node tree have a hierarchical relationship to each other.

The terms parent, child, and sibling are used to describe the relationships. Parent nodes have children. Children on the same level are called siblings (brothers or sisters).

- In a node tree, the top node is called the root
- Every node, except the root, has exactly one parent node
- A node can have any number of children
- A leaf is a node with no children

- Siblings are nodes with the same parent

The following image illustrates a part of the node tree and the relationship between the nodes:



Because the XML data is structured in a tree form, it can be traversed without knowing the exact structure of the tree and without knowing the type of data contained within.

You will learn more about traversing the node tree in a later chapter of this tutorial.

First Child - Last Child

Look at the following XML fragment:

```
<bookstore>
  <book category="cooking">
    <title lang="en">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year>
    <price>30.00</price>
  </book>
</bookstore>
```

In the XML above, the <title> element is the first child of the <book> element, and the <price> element is the last child of the <book> element. Furthermore, the <book> element is the parent node of the <title>, <author>, <year>, and <price> elements.

[**DETAIL ON XML DOM** - http://goo.gl/eXCbG]

WHY XML Parser?

We need XML parser because we do not want to do everything in our application from scratch, and we need some "helper" programs or libraries to do something very low-level but very necessary to us. These low-level but necessary things include checking the well-formed-ness, validating the document against its DTD or schema (just for validating parsers), resolving character reference, understanding CDATA sections, and so on. XML parsers are just such "helper" programs and they will do all these jobs. With XML parsers, we are shielded from a lot of these complexities and we could concentrate ourselves on just programming at high-level through the API's implemented by the parsers, and thus gain programming efficiency.

SAX Parser

SAX is the Simple API for XML, originally a Java-only API. SAX was the first widely adopted API for XML in Java, and is a "de facto" standard. The current version is SAX 2.0.1, and there are versions for several programming language environments other than Java.

SAX (Simple API for XML) is an event-based sequential 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.

DOM parsers Vs. SAX parsers

- A DOM parser creates a tree structure in memory from the input document and then waits for requests from client. But a SAX parser does not create any internal structure. Instead, it takes the occurrences of components of an input document as events, and tells the client what it reads as it reads through the input document.
- A DOM parser always serves the client application with the entire document no matter how much is actually needed by the client. But a SAX parser serves the client application always only with pieces of the document at any given time.
- With DOM parser, method calls in client application have to be explicit and forms a kind of chain. But with SAX, some certain methods (usually overridden by the client) will be invoked automatically (implicitly) in a way, which is called "callback" when some certain events occur. These methods do not have to be called explicitly by the client, though we could call them explicitly.

In the following cases, using SAX parser is advantageous than using DOM parser.

- The input document is too big for available memory (actually in this case SAX is your only choice)
- You can process the document in small contiguous chunks of input.
- You do not need the entire document before you can do useful work
- You just want to use the parser to extract the information of interest, and all your computation will be completely based on the data structures created by yourself. Actually in most of our applications, we create data structures of our own which are usually not as complicated as the DOM tree.

In the following cases, using DOM parser is advantageous than using SAX parser.

- Your application needs to access widely separately parts of the document at the same time.
- Your application may probably use an internal data structure, which is almost as complicated as the document itself.
- Your application has to modify the document repeatedly.
- Your application has to store the document for a significant amount of time through many method calls.

XML 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:

- XML Text nodes
- XML Element Starts and Ends
- XML Processing Instructions
- 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 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.

Example

```
<?xml version="1.0" encoding="UTF-8"?>
        <DocumentElement param="value">
                <FirstElement>&#xb6; Some Text</FirstElement>
                <?some_pi some_attr="some_value"?>
                <SecondElement param2="something">
                        Pre-Text <Inline>Inlined text</Inline> Post-text.
                </SecondElement>
        </DocumentElement>
```

This XML document, when passed through a SAX parser, will generate a sequence of events like the following:

- XML Element start, named *DocumentElement*, with an attribute *param* equal to "value"
- XML Element start, named *FirstElement*
- XML Text node, with data equal to "¶ Some Text" (note: certain white spaces can be changed)
- XML Element end, named *FirstElement*
- Processing Instruction event, with the target *some_pi* and data *some_attr="some_value"* (the content after the target is just text; however, it is very common to imitate the syntax of XML attributes, as in this example)
- XML Element start, named *SecondElement*, with an attribute *param2* equal to "something"
- XML Text node, with data equal to "Pre-Text"
- XML Element start, named *Inline*
- XML Text node, with data equal to "Inlined text"
- XML Element end, named *Inline*
- XML Text node, with data equal to "Post-text."
- XML Element end, named *SecondElement*
- XML Element end, named *DocumentElement*

[SRC: Wiki, W3S and more]