

## Introduction to XML

Reading Material

## Table of Contents

Table of Contents .....	2
Unit Content and Learning Objectives .....	3
1. The Need for XML .....	4
2. XML Concepts and Definitions .....	5
What is XML?.....	5
Uses of XML.....	6
3. XML Syntax.....	8
4. The Content of an XML Document.....	11
5. The Structure of an XML Document .....	13
6. What is "Well-formed" XML.....	14
7. Entities.....	15
8. Elements or Attributes? .....	16
9. Namespaces .....	17
10. Style Sheets .....	19
11. The eXtensible Stylesheet Language (XSL).....	20
CASE STUDY: Formatting a Hospital Document from XML Data .....	20
12. The XML Document Tree and the Document Object Model .....	23
13. XML and Object-Oriented Programming .....	25
What are Objects, Classes and Interfaces? .....	25
14. XML Validation .....	26
Well-formed vs. Valid .....	26
15. XML Schema.....	27
16. Access to the Content of an XML document.....	32
17. XPath .....	33
What is XPath?.....	33
Location Paths .....	33
Axes .....	34
Node tests.....	38
Predicates .....	40
XPath and the Nodes.....	43
18. Transformations .....	55
XSLT.....	55
Unit Summary and Conclusion.....	62

## Unit Content and Learning Objectives

Units 1 of this Course introduced you to the need and creation of health information standards as well as standardized use of the language ("Controlled Vocabularies") used to communicate healthcare information.

Units 2 cover two important tools for creating and operating healthcare information systems:

- The Unified Modeling Language (UML)
- eXtensible Markup Language (XML)

The eXtensible Markup Language (XML) plays a fundamental role in the interchange of HL7 data since it is the native implementation mechanism ("wire format") for the HL7 CDA, Version 3 and FHIR standards as well as an alternative syntax for Version 2 exchanges.

In the HL7 domain, standards using XML include:

- The V2.x messaging specification can use XML syntax ("V2.xml") instead of | and ^ delimiters
- The Version 3 Implementation Technology Specification (ITS) is expressed in XML
- The Clinical Document Architecture (CDA) standard is expressed as a set of XML Schemas and the implementation guides are a set of Schematron documents and/or XSL style sheets
- The Structured Product Labeling (SPL) standard for the pharmaceutical industry and the FDA uses XML
- The Conformance Profiles used to define DTD or Schemas to verify the conformity of applications with messaging profiles

For these reasons, we must master the following tools to understand, generate and validate HL7 V2.XML, HL7 V3 Messages or CDA Documents:

- XML
- XML Schema
- XPath
- XSL
- Schematron

Therefore, this Unit will provide you with a basic understanding of these XML tools.

## 1. The Need for XML

The growth and power of the Internet is based on its capability to allow individuals to communicate with one another on an international scale. This strength is made possible using standard technologies and protocols such as the Internet Protocol (IP), Hypertext Transfer Protocol (HTTP) and Hypertext Markup Language (HTML); all of which can and have been widely implemented because they are platform independent.

Over time, Internet technologies have been used more and more to communicate between or link applications together. Common examples are electronic payments, health and medical information exchange and other commercial transactions. It is because of this kind of interchange that the Internet has become a backbone network for business-to-business ("B2B") communications and between organizations (organization-to-organization).

**HTML** is an example of a widely used syntax for information interchange. However, even though it is very useful for information presentation, it does not effectively manage the interchange of data for automatic processing.



*For effective information interchange between applications a language is needed that describes the data in the exchange and makes this description available to systems in a standard format so any application, now or in the future, can understand and process the data properly.*

In the "Introduction to Vocabularies" Unit, it was demonstrated that data is only useful if the context of use is understood. The expression "screen" needs the context, e.g. "computing", "classroom", "cinema" or "sun protection". Similarly, it is futile to say that the price of a product is "4", unless the currency is known: Yen, Euro, Peso, US Dollar, etc.

Usually, applications handling data contain business rules or conditions determining the context of data use. However, when data is being exchanged between different applications, there is always the risk of miscommunication.

Internal to an organization, this problem can be solved, to some degree, by using data dictionaries or master files. However, for the effective exchange of data between different organizations, it is critical that we clearly define and expose both the structure and context of data.

For today's business to effectively manage data it must fulfill several requirements:

- Data must be readable by human beings and by machines
- Data content and structure must be defined
- Data relationships must be defined
- Data structure must be separated from data presentation
- Data structure must be open and extensible

***XML fulfills all of these requirements!***

## 2.XML Concepts and Definitions

### What is XML?

XML is the acronym for eXtensible Markup Language. It is related to the Hyper Text Markup language (HTML) which was derived from the (Standard Generalized Markup Language (SGML - ISO Standard 8879:1986). SGML was defined by ISO in 1986 for providing a unique language to represent pre-existing formatting types for electronic text. However, its complexities contributed to its limited adoption.

XML is a simplified subset of SGML, retaining its advantages while being simpler to learn and use. It was approved in 2001 as a global standard and is formally supported by the World Wide Web Consortium (W3C).

With XML documents can be created which contain not only data, but also the definition, meaning and structure of that data as well.

**XML is written entirely as printable text, without binary components.** This makes it independent of platforms and therefore an ideal tool for systems interoperability. It is especially capable of transmitting information on the Web using the standard hypertext transfer protocol (HTTP).

**XML supports data structure definition in an open, self-descriptive way.** It allows easy transmission of data with the certainty that the receiver can consistently process it.

When used to describe and structure information **XML can be used as a data definition language.** XML can describe data components, records and even complex information structures such as a commercial invoice or a medical report.

**XML is a markup language** that uses tags, in a way similar to that of HTML, for defining data structures. The important difference between XML and HTML is that custom tags can be defined in XML. This is quite different from HTML where tags are fixed and predefined, and only describe how data will be displayed but not its context or structure. XML provides a full description of a document's content and structure. The XML markup tags allow applications to understand the data as well as process and display it correctly.

The following HTML fragment states that a table will be shown with two rows and two columns, with text in each of its four cells. Even though a human could deduce the informative meaning, there is no structure that allows one to determine it:

```
<Table>
  <TR>
    <TD>name</TD><TD>age</TD>
  </TR>
  <TR>
    <TD>Alice</TD><TD>42</TD>
  </TR>
</Table>
```

name	age
Alice	42

The following XML fragment contains the data for the name and age of a person:

```
<Person>
  <Name>Alice</Name>
  <Age>42</Age>
</Person>
```

Data in XML is structured in a simple format that can easily be parsed by applications.



*XML is a standard syntax for building text documents using tags to define data structures.*

*XML is extensible, because it is not limited to a predefined set of tags; instead its tag set is open and can be extended as needed.*

Information contained in documents, whichever kind they are, consists of three main elements: the data, its structure and its presentation.

For example, a word processing document contains words, punctuation marks, spaces, etc. among its data; paragraphs, titles, tables, etc. in its structure and contains fonts, colors paper size, etc. establishing its presentation. However, these elements are often combined in such a way as to make them almost impossible to parse in any reliable and reproducible way. It is very difficult to create an application capable of extracting the contents of a text processor document and understanding its semantic meaning.

**The goal of XML is to separate these elements.** It considers the structure of the document as important as the content of the document. Presentation information is kept separate from structural information and content.

## Uses of XML

XML has many uses and applications. Its use has grown significantly in the last few years; both within systems and applications and in the communication between different organizations. Here are some examples of the most common areas of use:

- **Business Process Automation:** The highly structured nature of XML makes it suitable for data interchange mechanisms in loosely coupled systems; particularly in electronic commerce ("Business-to-Business - B2B) using the Internet as the transport means. It provides the functionality formerly provided by far more complex and expensive methods, such as EDI (Electronic Data Interchange).
- **Information Distribution:** The proper and timely distribution of information is important in every business environment. The World Wide Web provides a convenient infrastructure for transmitting information. Nevertheless, HTML is severely limited due to its inability to represent data meaning. XML overcomes these limitations and together with its associated style sheets (XSL),

which describe how information will be presented, which supports truly useful data distribution both within and between organizations.

- **Application and Data Integration:** Even within the same organization, it can be difficult, slow and expensive to integrate applications that differ in structure, data processing and platform. XML facilitates the integration process allowing basic communication between such applications. An application can parse incoming XML data as well as outputting any resulting data in the XML format. Since XML processing is both simple and supported by most platforms, it is already perceived as a universal interface between systems and has become a standard means for systems integration.
- **Web Services:** Going beyond integration, systems have increasingly become a composite of different services. These systems may be delivered by different systems that may even be dispersed across the globe. The natural link between these services and the consumers is the internet. These services are known as Web Services and **XML is the common language that allows web services-enabled applications to understand each other and work together.**

In this context, it is worth mentioning that in client/server implementations, based on binary protocols such as RPC, XML is considered a simple and useful standardization option.

### 3.XML Syntax

XML documents are composed of tags and text. The tags define data elements and the text is the actual data content of the document. Tags also establish the data context.

The basic information unit of XML is the **element**. Its basic syntax is:

Opening tag	Element content	Closing tag
<code>&lt;Name&gt;</code>	<code>Alice Miller</code>	<code>&lt;/Name&gt;</code>



*Note: In all the XML examples in this Unit, color, indenting, spaces, tabs, new lines and other text formatting is used only for humans reading or editing the XML document, but has no relevance to XML!*

*The content of an XML document could be one very long string of characters!*

In the above example, the element contains the name of person. An element begins with an **opening tag**, `<Name>`, and ends with a **closing tag**, `</Name>`. The textual content of the element (its value or "element content") is enclosed between the opening and closing tags. Tags provide the name of the element. The content of the closing tag is the same as that of the opening tag, except it is preceded by a slash `/`. XML is case sensitive, so capitalization must be the same for both the opening and closing tags.

Using the same syntax, the following XML element contains an age of a person:

```
<Age>42</Age>
```

XML elements can be nested one inside another, creating a hierarchy of related information. This principle of elements and their nesting is the same for all XML documents, no matter how complex or long they might be.

In the following example, the name and age of a person is grouped within a `<Person>` element:

```
<Person>
  <Name>Alice Miller</Name>
  <Age>42</Age>
</Person>
```

This simple syntax gives XML its power, flexibility and openness.

This nesting of elements within elements within elements, all of them having the same structure, reduces the complexity to an **extremely simple recursive structure**, which can be managed with a small set of rules no matter how complex the content may become.

Often, **XML documents include groups of repetitive elements**. For example, an organization may have an XML document containing a list of employees (persons), with the age of each person:



```
<Persons>
  <Person>
    <Name>Alice Miller</Name>
    <Age>42</Age>
  </Person>
  <Person>
    <Name>Dorian Smith</Name>
    <Age>35</Age>
  </Person>
  <Person>
    <Name>Harrison Jones</Name>
    <Age>21</Age>
  </Person>
</Persons>
```

This example uses an element called **<Persons>** to group the members. There are also three **<Person>** elements with each person having a **<Name>** element and an **<Age>** element.

**An XML document has a tree structure.** An element (node) referred to as the "parent" can contain other elements referred to as "children". This nesting can repeat to any desired depth, resembling a tree's branches. Every element contained by another is the descendant of the parent, and every element containing it is its ancestor. This structure is referred to as recursive.

The following example demonstrates this recursive structure with elements containing elements, which in some instances have the same name and structure:

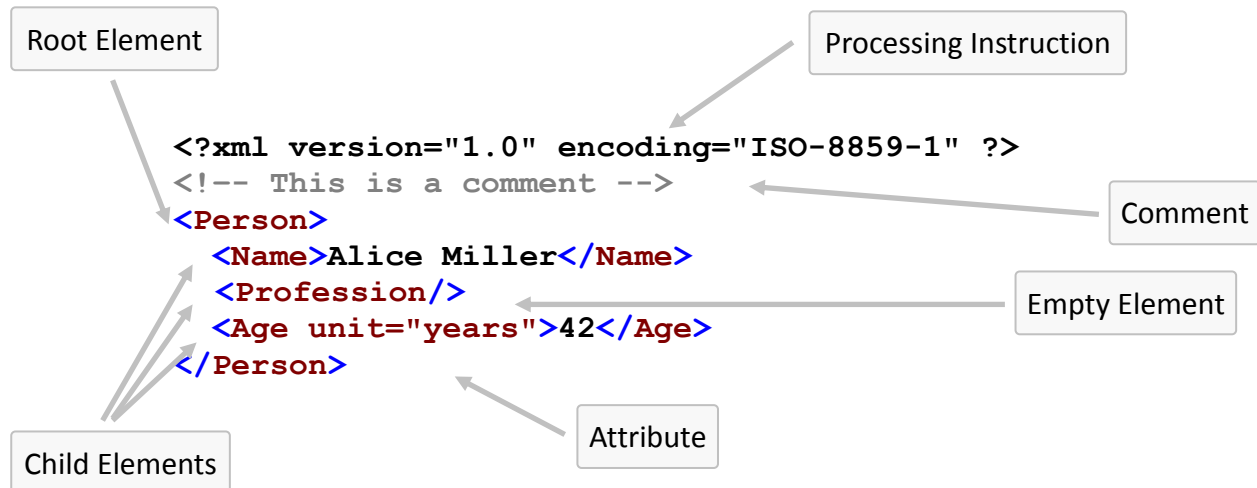
```
<Persons>
  <Person>
    <Name>Alice Miller</Name>
    <Age>42</Age>
    <Family>
      <Person>
        <Name>Anthony Miller</Name>
        <Age>23</Age>
      </Person>
      <Person>
        <Name>Anna Miller</Name>
        <Age>21</Age>
      </Person>
    </Family>
  </Person>
  <Person>
    <Name>Dorian Smith</Name>
    <Age>35</Age>
    <Family>
      <Person>
        <Name>David Smith</Name>
        <Age>15</Age>
      </Person>
    </Family>
  </Person>
</Persons>
```

```
    </Family>
  </Person>
  <Person>
    <Name>Harrison Jones</Name>
    <Age>21</Age>
  </Person>
</Persons>
```

## 4. The Content of an XML Document

Elements are the core component of an XML document; however, tag names are not the only forms of information that can be passed within the element.

The following diagram shows several of the main components of an XML document:



Processing instructions provide information to the XML processor necessary to correctly interpret the content of the document. They begin with a less than symbol and a question mark (`<?`) and they finish with a question mark and a greater than symbol (`?>`).



*Every XML document must begin with the following processing instruction, before any other character or white space. This is a special processing instruction known as an XML declaration:*

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

Processing instructions can be used for other purposes such as defining the encoding or character set for the document, or identifying a style sheet to be used to transform the document into a Web browser presentation.

Comments can be placed anywhere in an XML document except inside an element (e.g. between the opening and closing tags). Comments begin with a “less than” symbol, an exclamation point and one or two dashes (`<!--`); they end with two dashes and a “greater than” symbol (`-->`). Comments may include several lines. Because it serves to end a comment, the string `--` (e.g. two dashes) cannot be used elsewhere in a comment.

**Elements are the basic components of an XML document.** They have two main forms: one enclosing text and/or child elements, and another being completely empty. The following is an example of an **element containing text**; the `<Name>` element contains a person's name:

**`<Name>Alice Miller</Name>`**

The closing tag is required; it cannot be omitted.

**Child elements** are those enclosed by other elements, providing the hierarchical nature of XML documents.

In the following example, the **<Name>** element is a child of the **<Person>** element:

```
<Person>
  <Name>Alice Miller</Name>
</Person>
```

An **empty element** does not contain either text data or child elements:

```
<Name></Name>
```

An empty element can be represented using the full syntax or we can use an abbreviated syntax consisting of a single tag with the element name followed by a slash:

```
<Name/>
```

**Attributes** are used to provide information about the content of an element. They must be defined inside the opening tag of the element and consist of the name of the attribute, an equal sign, and the value of the attribute that is always enclosed between quotation marks.

In the following example, we can see an **<Age>** element containing a value of 42; the **unit** attribute specifies that the age is expressed in years:

```
<Age unit="years">42</Age>
```

## 5. The Structure of an XML Document

The World-Wide-Web Consortium (W3C) has defined a small set of rules defining the structure of a well-formed XML document. The basic structure consists of a **prologue** and a **root element**.

The **prologue** contains metadata applicable to the remainder of the document, such as the version of the XML standard it conforms to and the encoding or character set used.



*The term "**metadata**" is used to mean information about data (e.g. "data about data"). Metadata provides context for data being exchanged between applications, allowing the recipient to understand the data received from the sender in its correct meaning.*

One of the main parts of the prologue is identification of the **Document Type Definition (DTD)** or **XML Schema** that describes the structure of the document in terms of how element and attributes relate to each other.<sup>1</sup> DTDs and XML Schemas will be described in more detail later in this Unit.

The following example shows a prologue stating that the XML version is 1.0, the encoding is UTF-8 (an encoding commonly used for English language documents) and a reference to a DTD:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE persons SYSTEM "persons.dtd">
...
```

The following example prologue includes a reference to a XML Schema instead of a DTD:

```
<?xml version="1.0" encoding="UTF-8" ?>
<person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="persons.xsd">
...
```

The **root element** contains all the other elements and attributes of the document. All the data, provided they are represented by elements or attributes, are its descendants. There can only be one root element in any XML document. From a formal syntactical point of view, the entire content of the document consists only of the root element.

In the following example, the root element, which follows the prologue, is **<person>**:

```
<?xml version="1.0" encoding="UTF-8" ?>
<Person>
  <Name>Alice Miller</Name>
  <Profession/>
  <Age unit="years">42</Age>
</Person>
```

---

<sup>1</sup> Both DTDs and XML Schemas can define XML documents, but Schemas are the current preference and are replacing DTDs.

## 6. What is "Well-formed" XML

Every XML document must be "**well-formed**". That is to say, it must comply with a series of formal rules. If not in conformance to these rules, the document cannot be processed by XML parsers.

To be well-formed, an XML document must comply with the following rules:

- **There can only be one root element in an XML document** and it must be unique. All the other elements must be defined within the root element.
- **Each opening tag of an element must have a corresponding closing tag.** For empty elements, the abbreviated form can be used.
- **XML is case sensitive**; capitalization must be consistent. If the opening tag is `<Person>`, a closing tag of `</person>` would not be acceptable, it must be `</Person>`.
- **Elements must be correctly nested without overlapping.** Each element must be completely enclosed within its parent element.

The following XML example is correctly structured, because the tags are nested:

```
<aaaa>
  <bbbb> </bbbb>
</aaaa>
```

The following XML example is **incorrect**, because the tags "overlap":

```
<aaaa>
  <bbbb> </aaaa>
</bbbb>
```

- **The value of an attribute must be enclosed by quotation marks.** Either single or double quotation marks can be used. If a single quotation mark is used at the start of an attribute value, a single quotation mark must be used at the end of the attribute value.
- **Attributes within the same element must have unique names.** If it is required to represent repetitive information, nested elements must be used instead of attributes. An Attribute name cannot be repeated within the same element.

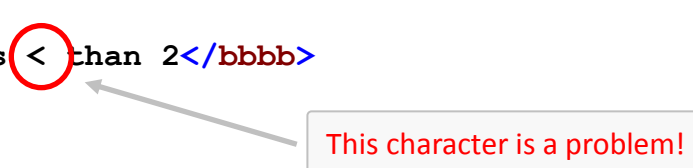
In addition to the requirement that every XML document be well-formed, it should also adhere to a certain **XML Schema** or **DTD**. If that is the case, the document is said to be **valid**.

## 7.Entities

The rules that establish an XML document as well-formed, raise certain disadvantages when we have to include characters in the document that are defined as delimiters by XML.

For example, if the text "1 is < than 2" is included in an element, then the following XML fragment is not considered well-formed, since the "<" in the text string represents a delimiter that does not have a corresponding closing delimiter:

```
<aaaa>
<bbbb>1 is < than 2</bbbb>
</aaaa>
```



This character is a problem!

**Entities** are used to resolve such situations. They are alternative representations of the XML delimiters and other special characters using a standardized nomenclature that allows these characters to be used as text in XML.

Entities used for XML documents are the same as those used in HTML, since both markup languages were derived from SGML. The following are the commonly used entities:

Character	XML Entity	Description
&	<code>&amp;amp;</code>	Ampersand
<	<code>&amp;lt;</code>	Less than
>	<code>&amp;gt;</code>	Greater than
'	<code>&amp;apos;</code>	Single quotation
"	<code>&amp;quot;</code>	Double quotation

Using an XML entity, the example above can be written in a well-formed manner:

```
<aaaa>
<bbbb>1 is &lt; than 2</bbbb>
</aaaa>
```

This XML is now well-formed and if viewed in a browser, would be displayed as:

```
<aaaa>
  <bbbb>1 is < than 2</bbbb>
</aaaa>
```

## 8.Elements or Attributes?

When defining the structure of an XML document, the document designer must first determine how best to represent the data. Elements or attributes can be used as there is no fixed rule for choosing one over the other and the design decision often depends on what we are attempting to convey. The following guidelines can be useful:

Elements are best used for:

- Core data fields
- Repetitive data
- Hierarchical sets of data
- Ordered data

Typically, the columns of a table or a database map well into elements.

We can use attributes for:

- Metadata of an element's content (to give more information such as currency etc.)
- Enumerated values

However, there are no fixed rules on which to choose.



## 9.Namespaces

The **grammar** of a formal language is a set of rules that defines which elements can be used and how elements relate to each other. Several grammars exist for XML documents, depending on the purpose of the document. These grammars are contained in **DTDs** and **XML Schemas**.

Besides these standard grammars, others can be created as needed, because XML is indeed open and flexible. For example, a custom Schema can be created that establishes certain elements, what their attributes are, which of them can be child elements, which are obligatory, etc. Based on this Schema, well-formed documents can be created that is valid since they satisfy the grammar defined by the Schema.



*However, with the advantage of flexibility comes the difficulty of ambiguity: many elements and attributes can be defined having the same name but different meaning, according to the Schema being used, and eventually it may be necessary to use more than one Schema simultaneously in a certain document or application.*

For example, XML documents can contain elements and attributes that have the same name, but are used differently depending on the business unit. For example, the accounting department of a hospital could use an element called "amount" to represent the size of a payment to be made. At the same time, the hospital pharmacy could use "amount" to indicate the volume of glucose fluid used for a drug infusion.

To distinguish those two elements with obviously very different meanings, XML **Namespaces** are used. They specify the context of use of a certain name, and therefore its meaning, resolving name collisions. Namespaces are identified by means of Universal Resource Identifiers (URI), which can take their form from a Uniform Resource Locator (URL) or Universal Resource Name (URN). In fact, the site referenced by URI does not need to actually exist or have a specific content, it is only necessary that it be a unique string, which makes the URL format a good option.

The Namespace definition includes an abbreviation that will be used as a prefix for the elements. This becomes the local name for the namespace within the XML document.

The notation is to prefix each element with the name of the namespace e.g. "acc" for elements that are used by the accounting department and "pharm" for the hospital pharmacy, followed by the colon ":" character:

```
...
<acc:pay xmlns:acc="http://www.GoodHealth.org/Accounts">
  <acc:customer>Hospital Supplies Inc.</acc:customer>
  <acc:amount>12000</acc:amount>
</acc:pay>
...
<pharm:inf xmlns:pharm="http://www.GoodHealth.org/Pharmacy">
  <pharm:infusion>Glucose</pharm:infusion>
  <pharm:amount>2000</pharm:amount>
</pharm:inf>
...
```

In the first part of the previous example, we see the prefix **acc** and the name of the element **pay**. The attribute **xmlns** is the standard indication of a namespace followed by the prefix and the corresponding URI. This prefix will be used within the element tags to indicate that their content is specific to the namespace.

In the second part of the example, the element is associated with the different namespace **pharm**, as are its child elements. In this way, element names remain unique and the XML document can be processed correctly with each element retaining its respective meaning.

An alternative is to declare a default namespace, omitting the prefix in the declaration of the namespace. The name of the namespace is directly assigned to the attribute **xmlns**:

```
<pay xmlns="http://www.GoodHealth.org/Accounts">
  <customer>Hospital Supplies Inc.</customer>
  <amount>12000</amount>
</pay>
```

Using this approach, the element and all its child elements and attributes are automatically part of the default namespace URI. It is not necessary to qualify them with the prefix as described before.

Note that there are a number of predefined Namespaces that should not be used: extensible style sheet language (xsl), scalable vector graphics (svg), etc.

## 10. Style Sheets

XML is very useful for representing data in complex structures and adapting itself to different purposes. Nevertheless, the presentation aspect of an XML document is not very elegant. If an XML document is viewed in a basic text editor, it will be displayed similar to this:

```
<?xml version="1.0" encoding="UTF-8" ?>
<Person><Name>Alice Miller</Name>
  <Profession/>
  <Age unit="years">42</Age></Person>
```

It is not any better if the same XML document is viewed in a browser:

```
<?xml version="1.0" encoding="UTF-8"?>
<Person>
  <Name>Alice Miller</Name>
  <Profession/>
  <Age unit="years">42</Age>
</Person>
```

While the line feeds and tab indents provide some assistance for reading or editing the document, they are in fact, not necessary at all. XML documents could be presented as a single line of text and would still be processed successfully by a computer.

However, a human reader would surely prefer a better form of display! In order to make the information acceptable to a human user, **style sheets** can be applied to an XML document, in a similar way that HTML defines Web pages.

There are two methods for defining style sheets for XML documents:

- eXtensible Stylesheet Language (XSL)
- Cascading Style Sheets (CSS)

In this Unit, only the XSL method will be explained.

## 11. The eXtensible Stylesheet Language (XSL)

The preferred method for presenting XML documents to human viewers is to use the eXtensible Stylesheet Language (XSL).

In spite of extending their use to XML, Cascading Style Sheets (CSS) have a limited scope.

One of the most common uses of an XSL style sheet is to **translate an XML document into an HTML document**. In the process, the XML tags are translated as they identify the data itself, whereas HTML tags define how the data is displayed in a browser.

Another use of XSL is to translate a set of XML tags into **another XML document with a different format**. This use is probably more appropriate for Business-to-Business (B2B) or e-Commerce communication, where it is particularly useful to transform formats between different applications to enable their interoperability.

The XSL style sheets consist of a set of rules that are applied to the different elements and attributes of an XML document. XSL uses pattern syntax to select specific elements or attributes. Furthermore, it provides a broad set of instructions and functions that is essentially a complete language for programming these transformations.

For example, a pattern to match all of the elements **<person>** in a XML document can be specified that allows them to be displayed like an HTML table.

Some of the advantages of using XSL to apply display formatting to XML documents are:

- It defines rules to transform elements and attributes to a different format
- It is more powerful and flexible than CSS
- It supports adding new elements and attributes, instead of only applying style to existing content.
- It allows ordering of elements and attributes
- It allows the use of loops, decision structures and other computations that can provide very sophisticated formatting

As with CSS, an XML processing instruction in the prologue is used to associate an XSL style sheet to an XML document:

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

Only one XSL style sheet can be applied to a document at a time. If more than one were associated to an XML document, only the first one would be used.

When an XML document containing a link to an XSL style sheet is loaded in an XML parser, it acts on the processing instruction and loads the XSL style sheet. The parser then transforms the XML document following the rules defined in the XSL style sheet.

### CASE STUDY: Formatting a Hospital Document from XML Data

This example shows how a fragment of an XML document that has an XSL style sheet associated with it is presented in a Web browser. The XML document and the XSL style sheet have been taken

from a real application in Spanish, so ISO-8859-1 encoding has been applied and confidential information has been hidden):

XML Patient List Document:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<?xml-stylesheet type="text/xsl" href="Rpt_1.xslt"?>
  <SOL_PACIENTE CAJA="D8" DTFECHACOMPROBANTE="01-07-2004" EPISODIO="" IAREA="251" ICONCEPTO="10050" PACIENTE="V***** D*****" VALE_PEDIDO="7*****" VCDESCRIPCION="PHMETRIA CON DOS SENSORES"/>
  <SOL_PACIENTE CAJA="D8" DTFECHACOMPROBANTE="01-07-2004" EPISODIO="" IAREA="252" ICONCEPTO="701" PACIENTE="M***** C*****" VALE_PEDIDO="7****" VCDESCRIPCION="VIDEOENDOSCOPIA DIGESTIVA ALTA ELECTRONICA" />
  . . .
```

The XML Patient List as displayed in a Web Browser after applying the XSL Style Sheet:

Listado de Pacientes por Día						
Pacientes por día						
Id Areal	Descripción	Pacienteo	Episodio	Caja	Vale / Pedido	Fecha
251	PHMETRIA CON DOS SENSORES			D8		01-07-2004
252	VIDEOENDOSCOPIA DIGESTIVA ALTA ELECTRONICA			D8		01-07-2004
252	VIDEOENDOSCOPIA ALTA CON TEST DE UREASA			D8		02-07-2004
253	VIDEOCOLONOSCOPIA ELECTRONICA			A8		01-07-2004
253	VIDEOCOLONOSCOPIA DE COLON IZQUIERDO			D8		01-07-2004
253	VIDEOCOLONOSCOPIA ELECTRONICA			-		03-07-2004
253	VIDEOCOLONOSCOPIA DE COLON IZQUIERDO			-		03-07-2004
253	VIDEOCOLONOSCOPIA ELECTRONICA			-		03-07-2004

The XSL Style Sheet used to convert the XML data to the Display above:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="html" encoding="ISO-8859-1" indent="yes"/>
  <xsl:template match="/">
  <xsl:element name="HTML">
    . . .
  <xsl:element name="BODY">
  <xsl:attribute name="class">copy-mini</xsl:attribute>
  <table border="0" cellspacing="0" cellpadding="0" width="100%"
    align="center">
    <tr><td>
      . . .
    <td align="center">Listado de Pacientes por Día</td>
    . . .
  <xsl:apply-templates select="/RG/SOL_PACIENTE"/>
  . . .
```

```
    </td></tr>
</table>
...
</xsl:element>
</xsl:template>
<xsl:template match="/RG/SOL_PACIENTE">
<xsl:variable name="Clase">
    <xsl:call-template name="clase"/>
</xsl:variable>
<xsl:element name="tr">
<xsl:attribute name="class">
    <xsl:value-of select="$Clase" />
</xsl:attribute>
<td>
<xsl:value-of select="./@IAREA" />
    </td>
...
</xsl:template>
<xsl:template name="clase">
<xsl:choose>
    <xsl:when test="(position() mod 2) =
    1">celdaResultPrototipo2</xsl:when>
    <xsl:otherwise>celdaResultPrototipo1</xsl:otherwise>
</xsl:choose>
</xsl:template>
</xsl:stylesheet>
```

The XSL file above may seem indecipherable; it is in fact a programming paradigm different from those commonly used. While its detailed study is beyond the scope of this Course, an XSL style sheet is itself an XML document based on patterns (templates), by which sets of elements of the original XML document are selected and some action is taken on them.

## 12. The XML Document Tree and the Document Object Model

An XML document has a structure with "branches" and "leaves" like a tree. This "tree" is made up of nodes (elements and attributes) that may have child nodes. The nodes of this structure can be navigated in a recursive way or any particular node can be directly accessed.

The Document Object Model (DOM) is a W3C standard that defines a set of objects and interfaces by which the content of an XML document can be searched and manipulated. The DOM is independent of any platform and any particular programming language. It is a data model that represents XML as a tree of nodes.

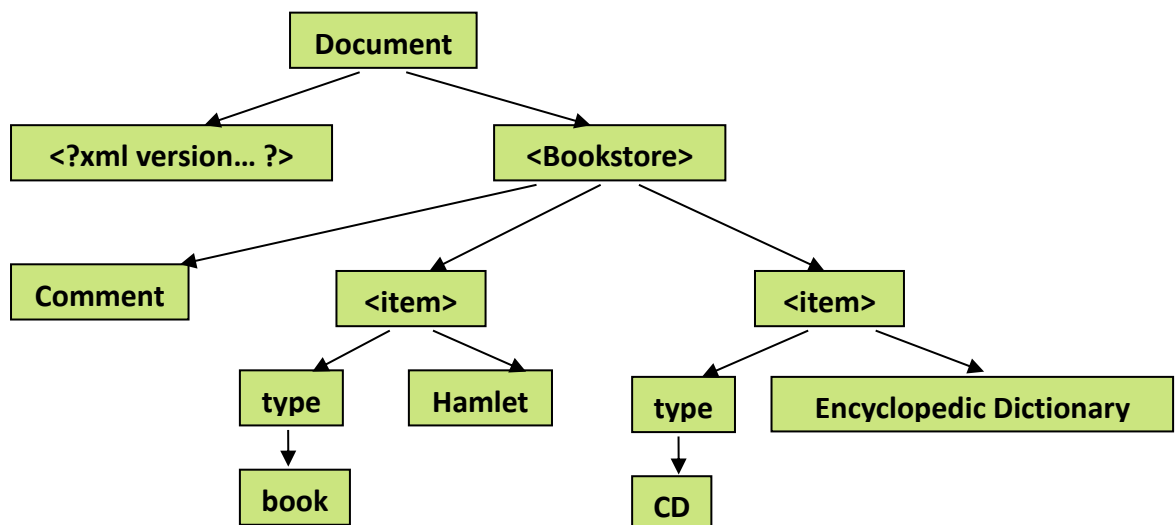
The DOM can be used to:

- navigate an XML tree
- obtain information (metadata) about any element, attribute, comment, processing instruction, text content, DTD or Schema in an XML document
- create new elements, attributes, comments, processing instructions or text contents or eliminate them
- create new XML documents

XML parsers are based on the DOM. In the following example, an XML document is converted into a DOM tree:

```
<?xml version="1.0" encoding="UTF-8"?>
<Bookstore>
  <!-- Comment -->
  <item type="book">Hamlet</item>
  <item type="CD">Encyclopedic Dictionary</item>
</Bookstore>
```

Every part of the document (processing instructions, elements, text, etc.) is represented as a node along with its parent/child relationship to other nodes. In the DOM tree diagram below, each of the green boxes represents a node of the XML document:



All the nodes descend from a root node, which is called the document object and is shown as the green box with the word *Document* in the above diagram. This node does not have a representation in the text of the XML document, because it is the XML document itself.

This root node has two child nodes, the XML declaration (the prologue mentioned previously) and the root element (the rectangle with the word *Bookstore*).

The element nodes can contain the following child nodes: comments, elements, attributes and one text node.

Attribute nodes can only contain a text node.

The DOM tree allows for the manipulation of XML documents. There are several tools to view and edit DOM trees; most modern browsers have built-in DOM viewing and editing functions.

These are some of the main objects within the hierarchy of the DOM (with their internal names shown in brackets):

**Document** - representing the complete XML document. It is the highest-level object in the hierarchy and is the access point to the entire document. It has methods to create elements (`createElement`), to create text nodes (`createTextNode`), to obtain elements by the name of its tag (`getElementByTagName`), etc.

**ProcessingInstruction** - representing and allowing access to a processing instruction.

**Element** - representing an element within the DOM tree. It allows access to the attributes and child nodes. It has methods to retrieve the value of the attributes (`getAttribute`), to assign value to them (`setAttribute`), to remove them (`removeAttribute`), to add child elements (`appendChild`), etc.

**Attr** - representing an attribute of an element. It has properties to access an element's name, value, etc.

**Text** - representing a text node and allowing the retrieval or modification of its content.

**Node** - representing a node of any type; element, attribute, text, etc. It has properties that allow for access to its name (`nodeName`), type (`nodeType`), value (`nodeValue`), child nodes (`childNodes`), parent node (`parentNode`), etc. and methods to add child nodes (`appendChild`), to clear a child node (`removeChild`), to select other nodes (`selectNodes` and `selectSingleNode`) using XPath expressions, which will be discussed later. Different node types (element, attribute, etc.) implement this interface. In order to use it, it is "cast" (that is, to assign its reference to a more specific data type) to the particular node type.

**NodeList** - representing a list of nodes and allows access to each particular iteration. This list can be formed by nodes that meet a certain characteristic, for example some XPath condition, as we will see later.

**Comment** - representing a comment.

**DocumentType** - representing the type of document, established by the DTD or Schema.



## 13. XML and Object-Oriented Programming

### What are Objects, Classes and Interfaces?



***Although object-oriented programming is not within the scope of this Course, "object oriented" (OO) has been repeatedly mentioned. This section is for students who have no knowledge of object-oriented programming. If you understand the fundamental OO concepts, please feel free to skip over this section.***

A class is like a blueprint used to construct a product. The class carries the definition of the characteristics (properties) and operation (methods) of the objects that are created from it. Each object is a different instance of the class with its own value in each of its properties, but it "performs" the same way as any other object of the same class.

In computer programs, objects of different classes are created to perform different parts of the function to be executed.

An interface is like the outer shell of an object, e.g. what is seen from the outside. This interface is the way the object communicates with other objects. Generally, the declaration of the set of properties and methods of a class is its interface, which is separate from the internal mechanism that makes it work. These internal mechanisms are not accessible from outside, they are implemented by each class in the interface.

Every class, and therefore the objects that are instances of it, has at least one interface. This is known as its default interface. In some cases, certain classes may implement a default interface and have additional interfaces supporting more methods or properties. There are different ways to do this (e.g. inheritance, secondary interfaces, etc.), but these details are outside the scope of this explanation. In summary, if a class implements all the methods and properties of an interface, it can be seen from the outside as if it has that interface and some other specialization. Therefore, it can be treated as either the more specialized class or the more "rudimentary" interface.

A DOM Node is an interface, that is to say, it is only the declaration of a set of methods and properties to define how a Node must be accessed from the outside but it is not a specification of its operation. The objects "Document", "Element" and Text implement the node interface; all have the Node's methods and properties along with the mechanism for the interface to work, in addition to their own methods and properties.

To summarize, any object of a class that implements an interface can be cast to this interface. This means that we can assign a reference of this object to one of the interface and from this moment on, we can use it as if it was specifically of that interface. The great advantage of this is that programs can be developed to handle nodes in some generic way, accessing them through the node interface, and then in run time, objects of the class elements, document, text, etc. will cast to node, simplifying development and making the objects that manipulate XML documents more capable and versatile.

## 14. XML Validation

XML is a very flexible syntax for describing the content of documents. Due to this flexibility, some mechanism is necessary to verify that a particular XML document type is well-formed and conforms to the format that has been selected. This mechanism is called validation.

Validation ensures that an XML document has the:

- Correct elements and attributes
- Correct relationships between elements and attributes
- Correct sequence and quantity of child elements
- Correct data types

Validation is a technical confirmation that a document fully conforms to a certain grammar and that it can be processed according to that grammar.

### Well-formed vs. Valid

Every XML document must be well-formed, e.g. it must follow the basic rules of the XML syntax. This does not imply any limitation on its content; it can have any legal combination of elements and/or attributes and can have any grammar. All parsers can process well-formed XML documents; they represent the starting point for any development of XML documents.



*A valid XML document is a well-formed document that also adheres to the grammar restrictions defined in a DTD or XML Schema that defines it. However, not every parser can validate an XML document nor will an XML document always require validation.*

There are two ways to define a grammar against which to validate an XML document: **Document Type Definition (DTD)** and **XML Schema**. XML Schema is the currently preferred approach and has largely replaced the original DTD.

## 15. XML Schema

The XML Schema specification was released as a World-Wide-Web Consortium (W3C) standard in 2002. It has quickly replaced the Document Type Definition (DTD) due to a number of important advantages. At present, XML Schema is the standard way to define a grammar for XML documents. Most parsers, on various platforms, are able to validate documents against Schemas.

The basic features of XML Schema are:

- is expressed as an XML document, eliminating the need to handle a completely different syntax.
- defines the elements, attributes and namespaces allowed in our XML document
- defines data types (primitive, simple or complex) for elements and attributes
- defines a minimum and maximum quantity for the elements
- defines the relationship between elements and attributes
- defines the order in which child elements must appear
- defines enumerations of possible values
- is a extensible definition
- has richer and more expressive semantics than the DTD
- can be applied to a complete XML document or just specific parts

The XML Schema is defined in a separate file with extension **".xsd"** and referenced by the XML instance document.

In order to apply it, a **Namespace** declaration is used; the prefix **"xsi"** is used to declare that the URL references a local Schema against which the document must be validated.

As with XML documents, using a namespace declaration allows the Schema to be applied only to an element and all its descendants. Therefore, a Schema can be applicable to the whole document if it is referenced by the root element or only to a specific element within the document.

The following XML document includes a Schema against which it is validated. It includes the namespace of the Schema that will be created in a file called **Persons.xsd**, in the same location as the XML document file:

```
<?xml version="1.0" encoding="UTF-8" ?>
<Persons xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="Persons.xsd">
  <Person>
    <Name>Dorian</Name>
    <LastName>Miller</LastName>
    <Document Type="PAS">8765432</Document>
    <Sons-Daughters>
      <Person>
        <Name>Judith</Name>
        <LastName>Miller</LastName>
        <Document Type="SSN">10987654</Document>
      </Person>
    </Person>
  </Person>
</Persons>
```

```
<Name>Herbert</Name>
<LastName>Marshall</LastName>
<Document Type="SSN">2345678</Document>
</Person>
<Person>
  <Name>Rupert</Name>
  <LastName>Marshall</LastName>
  <Document Type="SSN">12345678</Document>
</Person>
<Person>
  <Name>Alice</Name>
  <LastName>Marshall</LastName>
  <Document Type="SSN">17456789</Document>
</Person>
<Person>
  <Name>Harrison</Name>
  <LastName>Faber</LastName>
  <Document Type="SSN">37456789</Document>
</Person>
</Sons-Daughters>
</Person>
</Persons>
```

In the following, the Schema file Persons.xsd will be built step-by-step.

The Schema file Persons.xsd begins with an element `<xs:schema>`, of the namespace `xs`, as defined by the W3C and ends with the closing tag `</xs:schema>`:

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

The element `<xs:element>` is used to declare the valid elements that an XML document can contain. The value of its attribute `name` will be the name of the element in the XML document to be validated.

In the case of elements that will only have a child text node, its data type can be declared by means of the attribute `type`. The values of this attribute are defined by various W3C data types, `xs:string` for a character string, `xs:int` for integers, etc.

Thus, we could declare the element `<LastName>` of our XML document, in the following way:

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

Elements which do not have attributes or child elements are considered simple types (`<xs:simpleType>`) as opposed to complex types (`<xs:complexType>`), which have attributes or child elements. This declaration can be included as a child element of `<xs:element>`.

Within **<xs:complexType>** we can indicate the order in which the child elements will appear by means of the element **<xs:sequence>**. If we want to indicate that child elements exist but in no particular order, we can use **<xs:all>** instead of **<xs:sequence>**. Within this element, **<xs:element>** is included for each child element, indicating the name of the corresponding child element in its attribute **ref**. Complex types have to be declared somewhere in the Schema.

The element **<Person>** can be declared in the following way:

```
<xs:element name="Person">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="Name" />
      <xs:element ref="LastName" />
      <xs:element ref="Document" />
      <xs:element ref="Sons-Daughters" minOccurs="0" max-
Occurs="unbounded" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

The attributes **minOccurs** and **maxOccurs** respectively indicate the minimum and maximum number of allowed occurrences of the corresponding element. The value **unbounded** in **maxOccurs**, indicates that there is no limit to the number of occurrences.

Attributes of the elements of the XML document to be validated are declared by means of the element **<xs:attribute>**. It will have an attribute **name** with the name and an attribute **type** with its type. They may be included in a complex type following the element **<xs:sequence>**.

In order to declare elements that have a text node along with attributes or child elements, the element **<xs:simpleContent>** is used within **<xs:complexType>**. Within **<xs:simpleContent>** an element **<xs:extension>** is used to indicate the data type of the content by means of its attribute **base**. The declaration **<xs:extension>** is used for an element that "extends" this data type or conforms to a more complex type.

Therefore, the element **Document** of the previous XML document is declared as:

```
<xs:element name="Document"/>
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base="xs:int"/>
      <xs:attribute name="Type" use="required" type="xs:string"/>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>
```

Note the location of the element **<xs:attribute>**, within the element **<xs:extension>**. As previously mentioned, this indicates the extension of the type **xs:int** by means of the use of an attribute. The attribute **use** of the element **<xs:attribute>** indicates if it can and must be present.

Finally, the complete Schema for the XML document is:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Persons">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Person" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="Person">
    <xs:complexType>
      <xs:all>
        <xs:element ref="Name"/>
        <xs:element ref="LastName"/>
        <xs:element ref="Document"/>
        <xs:element ref="Sons-Daughters" minOccurs="0"/>
      </xs:all>
    </xs:complexType>
  </xs:element>
  <xs:element name="Name" type="xs:string"/>
  <xs:element name="LastName" type="xs:string"/>
  <xs:element name="Document">
    <xs:complexType>
      <xs:simpleContent>
        <xs:extension base="xs:int">
          <xs:attribute name="Type" use="required" type="xs:string"/>
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>
  <xs:element name="Sons-Daughters">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="Person" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

The structure of a Schema is quite complex, because it has a great level of detail and a recursive structure like XML. However, most of XML development tools can automatically generate an XML Schema from an XML document.

The element **<xs:include>** allows referencing another Schema, by means of the attribute **schemaLocation** containing its URL. This other Schema will be included by a parser at the time of validation; all definitions will be applied as if they were part of the base Schema.

When declaring a complex type by means of the element `<xs:complexType>`, an attribute **name**, can be included and assigned a name identifier. This name can be used in the attribute **type** of `<xs:element>`, declaring that these elements must have the structure defined by that complex type.

There is also a way to include information to document the Schema itself and to offer additional instructions for parsers or applications that process the Schema. This is done by means of the element `<xs:annotation>`. It can be included within the elements that require such information and can contain an element `<xs:documentation>`, comments to document the Schema, and an element `<xs:appinfo>`, that carries information for the application that processes the Schema.

This XML Schema overview is not exhaustive nor have all its elements been described as this is beyond the scope of this Course. Nevertheless, this introduction has provided the basics for understanding the use of XML Schema in the HL7 standards.

## 16. Access to the Content of an XML document

The expressions "parser" and or "to parse" have previously been mentioned in this Course. **Parsers are programs, libraries or components that allow us to work with XML documents.** They are implementations of the Document Object Model (DOM) and Simple API for XML (SAX).

Information systems that implement XML use an XML parser, either a third party product or developed internally. Parsers can be included in applications and transparently take care of (e.g. "encapsulate") all details of the XML documents. This makes the creation of applications that create and/or process XML documents much easier by substantially reducing the development time required.

Although they have functional differences according to their implementation, generally all parsers offer the same basic functions:

- They allow an XML document to be read and transform it into a DOM tree or SAX events.
- They allow access to the nodes by means of XPath expressions
- They allow the DOM to be manipulated, adding and removing nodes
- They can be controlled programmatically



## 17. XPath

### What is XPath?

One of the fundamental methods for the manipulation of an XML document is the XPath language. XPath is a standard language, defined by the W3C, used to locate nodes and data in an XML document.

When we work with XML, normally we need to select and manipulate certain nodes programmatically. XPath allows for the selection of either individual nodes or sets of nodes. Its data model maps an XML document to a tree of nodes: root, elements, attributes, text, etc. In order to make the selections, XPath uses expressions that recognize these nodes. These expressions can point the way to a certain node within the tree or can contain operators and functions to find the nodes that match certain criteria.

XPath is used in various aspects of XML document creations and processing<sup>1</sup>:

- eXtensible Stylesheet Language (XSL) uses XPath to select nodes that match a certain pattern and to apply a specified transformation to them.
- DOM parsers use XPath expressions to select or to filter nodes.
- The XML Pointer Language (XPointer) is a W3C standard that allows the creation of links in an XML document with nodes of another document, similar to HTML hyperlinks. XPath is used to define these links and also allows conditional linking according to the result of XPath expressions.<sup>2</sup>
- There are databases that support querying via XML and return the result as XML, e.g. Oracle and SQL Server. In these applications, we can also use XPath to select nodes of returned XML and to generate new documents.

The nodes in an XPath tree have four properties:

- The **node value**: a Unicode string with the content of the node. When the node is an element, this contains the concatenated value of all the child elements.
- The **node name** including the namespace, if any.
- The **parent node**, a reference to the containing node. The root node does not have a parent.
- A **list of child nodes**: a set of nodes (node-set) that are its direct children. The attributes, text nodes, processing instructions, namespaces and comments do not have child nodes.

### Location Paths

A location path is an XPath expression that contains a sequence of tests that locate specific nodes in an XML document. It selects nodes in a document hierarchically, similar to the path of a file in an operating system.

There are two types of location paths:

---

<sup>2</sup> More information on XPointer at <http://en.wikipedia.org/wiki/XPointer>

- **Relative:** defines the path through an XML document in relation to the present node, that is to say the one we selected and are working with. In relative location paths, the present node, also called the context node, is represented with a dot ("."). This path is useful when we have already located a particular node and we want to locate another near it such as a child or the parent.
- **Absolute:** defines the path through an XML document beginning from the root node of the document. It always begins with a slash ("/"). This path is useful when the present location of a node in the document is not important and we desire to start a new search from the root.

In both forms of location paths, the child nodes are represented with a slash followed by the name of the node being searched for ("/ChildNodeName").

A location path is made up of one or more location steps, separated by slashes. Each step returns a node-set as a result and is evaluated from left to right, refining the resulting node-set in each step.

The result of location path is called a node-set, which is the set of nodes that match the criteria of the expression. A node-set can contain one, many or no nodes.

These are the **absolute location paths** corresponding to the elements of an XML document:

Element	Absolute Location Path
<code>&lt;persons&gt;</code>	<code>/persons</code>
<code>&lt;person gender="F"&gt;</code>	<code>/persons/person</code>
<code>&lt;name&gt;Alice Miller&lt;/name&gt;</code>	<code>/persons/person/name</code>
<code>&lt;age&gt;42&lt;/age&gt;</code>	<code>/persons/person/age</code>
<code>&lt;/person&gt;</code>	<code>/persons/person</code>
<code>&lt;/persons&gt;</code>	<code>/persons</code>

Here is an example of **relative location paths** corresponding to the second element `<person>`:

Element	Absolute Location Path
<code>&lt;persons&gt;</code>	
<code>&lt;person gender="F"&gt;</code>	
<code>&lt;name&gt;Alice Miller&lt;/name&gt;</code>	<code>./name</code>
<code>&lt;age&gt;42&lt;/age&gt;</code>	<code>./age</code>
<code>&lt;/person&gt;</code>	
<code>&lt;/persons&gt;</code>	

## Axes

Each location path step specifies an axis, a node test and one or more optional predicates. The syntax is as follows:

**axis::test[predicate]**

The axis defines what type of search is to be made, for example the **child** axis for the child elements, the **attribute** axis for the attributes.

**Node test** specifies the names of the nodes or the type of nodes that are to be selected. Typically, it is the name of an element or attribute.

A predicate is a logical expression, with a Boolean result, that filters the node-set obtained from the node test. Each location path step can have zero or more predicates. Each one of them is enclosed in square brackets (`[]`) and filters the nodes that match the specified criteria.

Considering the previous XML example, the following location path, which is relative to the node `<persons>`, would select the child nodes of type `<person>`, which in turn, had a child node `<age>` with a value greater than 30:

```
child::person[age > 30]
```

XPath defines a number axes that allow us to look at an XML document in any way needed:

- **self**: contains only the context node (Can also be written as ".")
- **child**: contains the child nodes of the context node
- **parent**: contains the parent node of the context node (Can also be written as "..")
- **attribute**: contains the attributes of the context node (Can also be written as "@")
- **descendant**: contains the child nodes of the context node, plus their children, and so on
- **descendant-or-self**: contains the descendants and the context node itself (Can also as "///")
- **ancestor**: contains the parent node of the context node, plus its parent, and so on
- **ancestor-or-self**: contains the ancestors and the context node itself
- **following**: contains all the nodes to any depth that appear after the closing tag of the context node, except attributes nodes and namespace nodes.
- **following-sibling**: contains all the nodes that appear after the closing tag of the context node both at the same depth and in the same branch. We could call these "brothers". If the context node is an attribute or namespace node, the resulting node-set will be empty.
- **preceding**: contains all the nodes, at any depth, that appear before the opening tag of the context node, except ancestors nodes, attributes nodes and namespace nodes
- **preceding-sibling**: contains all the nodes that appear before the opening tag of the context node, at the same depth and in the same branch. We could also call these "brothers". If the context node is an attribute or namespace node, the result node-set will be empty.
- **namespace**: contains the namespace nodes of the context node

As shown in brackets in the above list, the most frequently used axes (*self*, *child*, *parent*, *descendant-or-self* and *attribute*) can be abbreviated for ease of use:

Axis	Unabbreviated example	Abbreviated
self	self::node()	.
child	child::name	name
parent	parent::node()	..
descendant-or-self	/descendant-or-self()/name	//name
attribute	attribute::gender	@gender

- The **self** axis is denoted by a dot (.)
- The **child** axis is the assumed default when only the node test is indicated
- The **parent** axis is denoted by two dots (..)
- The **attribute** axis is denoted with an at symbol (@) followed immediately by the node test
- The **descendant-or-self** axis is denoted with two slashes (//) followed immediately by the node test

Consider the following table:

```
<persons>
  <person gender="M">
    <name>
      Harrison Farrell
    </name>
    <age>35</age>
  </person>
  <person gender="F">
    <name>
      Christine Davis
    </name>
    <age>28</age>
  </person>
</persons>
```

If in the preceding example, the context node is the node **<persons>**.

The following location path uses the child axis in two successive steps:

**child::person/child::name**

From left to right, these steps are evaluated this way:

**child::person** obtains all the child nodes called **person** of the context node

**child::name** takes the node-set returned by the first step and from it, obtains all the child nodes called **name**

The equivalent abbreviated expression is as follows:

**person/name**

Either of the two expressions obtains the following nodes highlighted in color (the context node is **<persons>**):

```
<persons>
  <person gender="M">
    <name>
      Harrison Farrell
    </name>
  </person>
</persons>
```

```
</name>
<age>35</age>
</person>
<person gender="F">
  <name>
    Christine Davis
  </name>
  <age>28</age>
</person>
</persons>
```

Next, we'll examine the abbreviated and unabbreviated examples of absolute location paths using the attribute axis:

**child::persons/child::person/attribute::gender**

**/persons/person/@gender**

Either of the two expressions obtains the following nodes highlighted in color:

```
<persons>
  <person gender="M">
    <name>
      Harrison Farrell
    </name>
    <age>35</age>
  </person>
  <person gender="F">
    <name>
      Christine Davis
    </name>
    <age>28</age>
  </person>
</persons>
```

In the following example showing the use of the parent axis, let's assume that the context node is the one highlighted in green and the result is highlighted in yellow

**parent::node()/child::age**

**../age**

```
<persons>
  <person gender="M">
    <name>
      Harrison Farrell
    </name>
    <age>35</age>
  </person>
  <person gender="F">
    <name>
      Christine Davis
    </name>
    <age>28</age>
  </person>
</persons>
```

```
    </name>
    <age>35</age>
  </person>
  <person gender="F">
    <name>
      Christine Davis
    </name>
    <age>28</age>
  </person>
</persons>
```

The expression **node()** shown in the example, is a function that obtains a reference to the node itself. We will study the functions in more detail later.

If the context node is **<persons>**, the following example will select all the descendants (or the node itself if it matched) whose element name is **age**:

```
/descendant-or-self::age
```

```
//age
```

```
<persons>
  <person gender="M">
    <name>
      Harrison Farrell
    </name>
    <age>35</age>
  </person>
  <person gender="F">
    <name>
      Christine Davis
    </name>
    <age>28</age>
  </person>
</persons>
```

## Node tests

The node test follows the axis in the location path steps and provides for locating nodes by name and type.

This is the syntax to look for nodes by name:

```
axis::node-name[predicate]
```

XPath interprets the names in different ways, depending on the type of node indicated by the axis:

- If the axis is an attribute, the name is understood to be an attribute name. If we want to select all the attribute nodes of an element, we must use an asterisk (\*) instead of a specific name

- If the axis is namespace, the name is understood to be the prefix of a namespace. If we want to select all the namespace nodes, we must use an asterisk (\*) instead of a specific prefix
- For all the other axes, the name is understood to be an element name. XPath will select the elements within the specified axis whose names match those specified in the test. If we want to select all the element nodes, we must use an asterisk (\*) instead of a specific name

In order to select elements or attributes by name in a particular namespace, the prefix must be specified as follows:

**eje::prefix:node-name[predicate]**

And to select all the elements or attributes in a namespace:

**axis::prefix:\*[predicate]**

Let's review some examples and their explanation:

- **age** selects the **<age>** elements
- **@gender** selects the **gender** attributes
- **\*** selects all the elements
- **@\*** selects all the attributes

In the examples so far, elements and attributes have been tested by name. Sometimes it is useful to select nodes by their types instead of their names. The syntax is as follows:

**axis::node-type() [predicate]**

XPath has several functions for **node-type()**. These functions return the nodes according to their type:

- **node()** all the nodes in the specified axis
- **text()** all the text nodes in the specified axis
- **comment()** all the comment nodes in the specified axis
- **processing-instruction()** all the processing instruction nodes in the specified axis

Next we'll examine a test by node type example where all the text nodes of the XML document are selected (using the axis descendant-or-self and absolute location path):

**//text()**

```
<persons>
  <person gender="M">
    <name>
      Harrison Farrell
    </name>
    <age>35</age>
  </person>
  <person gender="F">
    <name>
      Christine Davis
    </name>
  </person>
</persons>
```

```
</name>  
<age>28</age>  
</person>  
</persons>
```

## Predicates

A predicate is a condition enclosed in square brackets ([]) that filters the node-set obtained with the node test. Each location path step can include predicates. Using predicates we can filter nodes by position, content or whether they contain other nodes or not. The predicate is evaluated successively for each node of the node-set to determine if it matches the condition.

In order to filter nodes by position, the following must be considered:

- Most of the XPath axes number the nodes following the document order. For example, in the child axis, the elements nearest the beginning of the document have a position smaller than those nearer to the end.
- Nodes can be filtered using either an ascending or descending axis. The descending axes are ancestor, ancestor-or-self, preceding and preceding-sibling. All other axes are ascending.
- Each node in a node-set has a position; the first position is 1.
- If a descending axis is used, the nodes are numbered inverse to their position in the document.
- If an ascending axis is used, the nodes are numbered in the same order as their position in the document.

Predicates can be used as follows:

- **person[1]** selects the first **<person>** child element of the context node.
- **person[last()]** selects the last **<person>** child element of the context node. The function **last()** returns the last existing position in a node-set.
- **persons/person[2]** selects the second **<person>** child element of the element **<persons>**

Predicates can be specified that filter nodes by their value. For this the equal operator (=) is used following the node to compare and then the search value enclosed in quotes (""):

- **person[@gender="F"]** selects the **<person>** child elements of the context node which have an attribute **gender** with a value equal to **F**.

In order to filter nodes by their presence, we indicate the element or attribute that the nodes seek must contain in the predicate:

- **person[name]** selects the **<person>** child elements of the context node which contain a child element **<name>**.
- **person[@gender]** selects the **<person>** child elements of the context node that contain an attribute **gender**.

We can define multiple predicates, each enclosed in their own set of brackets, to apply several filters to a node-set. The predicates are evaluated from left to right:



- **person[2][@gender]** selects the second **<person>** child element of the context node, **if** it contains an attribute **gender**.
- **person[@gender][2]** selects the second **<person>** child element of the context node **that** contains an attribute **gender**.

Different types of functions and operators can be used in the predicates, making their semantics more powerful.

The node-set is an XPath unique data type. It contains all the nodes that match the location path criteria. We can use the union operator and node-set functions to combine nodes.

The union operator is the vertical bar (|) and it indicates that the nodes matching any one of the conditions on both its sides will be selected:

- **name | age** selects all the **<name>** and **<age>** child elements of the context node.
- **(man | woman)/@age** selects all the **<man>** and **<woman>**, child elements of the context node, which contain an attribute **age**.

Node-set functions either receive a node-set type parameter or return a node-set or information about a particular node within a node-set:

- **position()** returns a node's position in a node-set (beginning at 1).
- **last()** returns the position of the last node of the node-set.
- **count()** returns the number of nodes in a node-set.
- **local-name()** returns the name of a node without including its namespace.
- **namespace-uri** returns the URI of the namespace of a node name.
- **name()** returns the namespace prefix and the name of a node.

For example, the predicate **person[position() = last()]** will return the **<person>** child element of the context node, which appears in the last place in the document order.

In the predicates we can include logical operators, which evaluate true or false:

- **=** equal
- **!=** different
- **>** greater than
- **>=** greater than or equal
- **<** lesser than
- **<=** lesser than or equal
- **and** logical conjunction
- **or** logical disjunction

For example, the predicate **person[age < 30 or @gender = "F"]** will select the **<person>** child elements of the context node, which have an **<age>** element having a value less than 30 or (that is to say also) those that have a gender attribute with a value of **F**.

The logical functions are:

- **true()** returns true
- **false()** returns false

- **boolean()** turns its argument to true or false: a node-set is true unless it is empty, a number is true unless it is zero, a character string is true unless it is empty.
- **not()** returns the logical negation of its argument

For example, the predicate **person[not(@gender)]** will select the **<person>** child elements of the context node, which do not have a **gender** attribute.

The precedence of the operators is as follows:

- **()** grouping
- **[]** filters
- **/ //** operations of path
- **< <= > >=** comparisons
- **= !=** equality or inequality
- **|** union
- **not()** negation
- **and** conjunction
- **or** disjunction

For numerical operations, XPath supports the floating point numerical data type with the following operators and numerical functions:

- **-** negation operator
- **+** addition operator
- **-** subtraction operator
- **\*** multiplication operator
- **div** division operator
- **mod** module operator, remainder of the whole division
- **number()** function that converts data to its numerical value. It returns NaN (not a number) if the argument is not a numerical value.
- **floor()** function that returns the maximum whole number lesser than or equal to the argument
- **ceiling()** function that returns the minimum whole number greater than or equal to the argument
- **round()** function that returns the rounded value of the argument
- **sum()** function that adds the argument node-set

For example, the predicate **account[sum(amount) > 0]** will select the **<account>** child elements of the context node, which in turn have **<amount>** child elements whose sum is greater than zero.

For character strings, the XPath **string** data type represents a sequence of zero or more characters. Next we'll review some of the functions for string manipulation:

- **string()** converts the argument to a character string
- **string-length()** returns the length of a character string
- **concat()** concatenates two or more character strings
- **starts-with()** returns true if the first parameter begins with the second one
- **contains()** returns true if the first parameter contains the second one

- **substring()** returns part of the string that it receives as first parameter, starting at the position indicated in the second parameter (beginning in 1) and of a length specified in the third parameter
- **normalize-space()** removes leading and trailing spaces from a string and converts multiple consecutive spaces into a single space

The XPath expression **string-length(person[1]/name)** will return the length of the text in the **<name>** child element of the first **<person>** child element of the context node.

The XPath expression **concat(@LastName, ", ", @Name)** will return a string with the value of the attribute **LastName** followed by a comma, a blank space and finally the value of the attribute **Name**.

As you can see, XPath is a powerful and simple language allowing us to access and manipulate the content of a XML document.

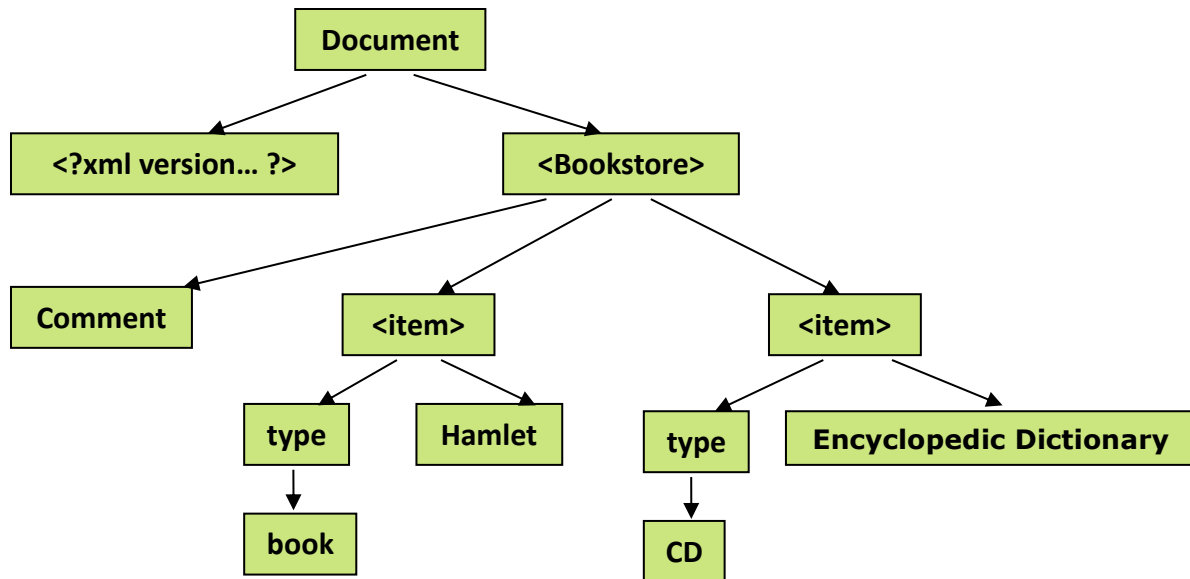
As with any language, there is a series of "good practices" to be considered in order to optimize its use and performance:

- Always use the abbreviated syntax, when available, since it is easier to understand and maintain
- Avoid expansive selections like **// (descendant-or-self)** where all the nodes in all the branches of the document must be visited. Take advantage of the knowledge of the structure of the document with which you are working to make XPath expressions as precise as possible.
- Avoid unnecessary use of the function **count()**, because it requires a loop through all the node-set. To verify if a node-set is empty, for example, use the function **last()**.
- Use **local-name()** or **namespace-URI()** according to the need to determine the name of a node. The function **name()** is not recommended, since it returns both the namespace prefix and the name, and there may be different namespaces in a document.

## XPath and the Nodes

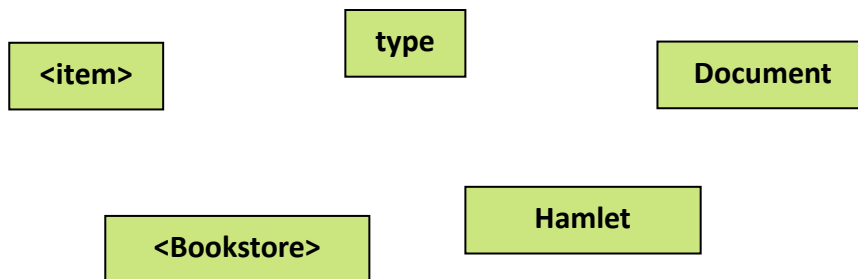
Based on the above introduction to the XPath language, here are more examples of node selection:

```
<?xml version="1.0" encoding="UTF-8"?>
<Bookstore>
  <!-- Comment -->
  <item type="book">
    Hamlet
  </item>
  <item type="CD">
    Encyclopedic Dictionary
  </item>
</Bookstore>
```



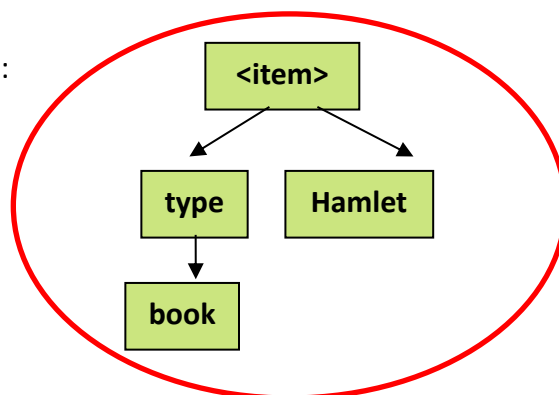
*Each of the green rectangles is one node.*

These then are nodes:



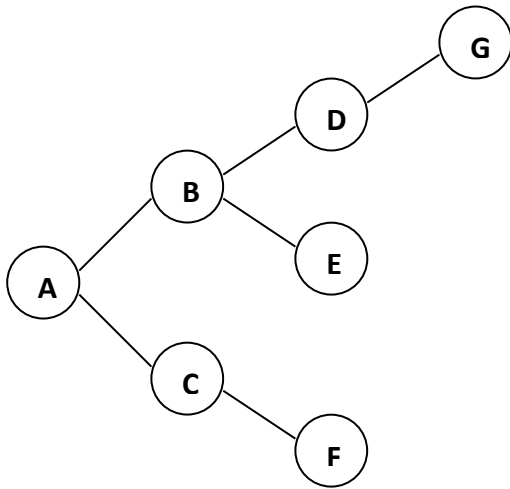
Etc.

But this is **not** a node:

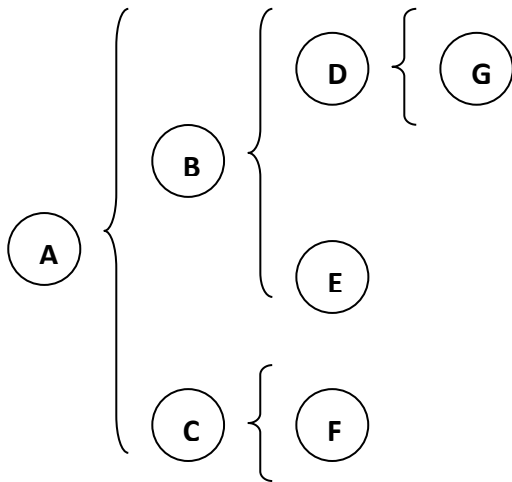


It is an element and its descendants. It is a **node-set** of several nodes.

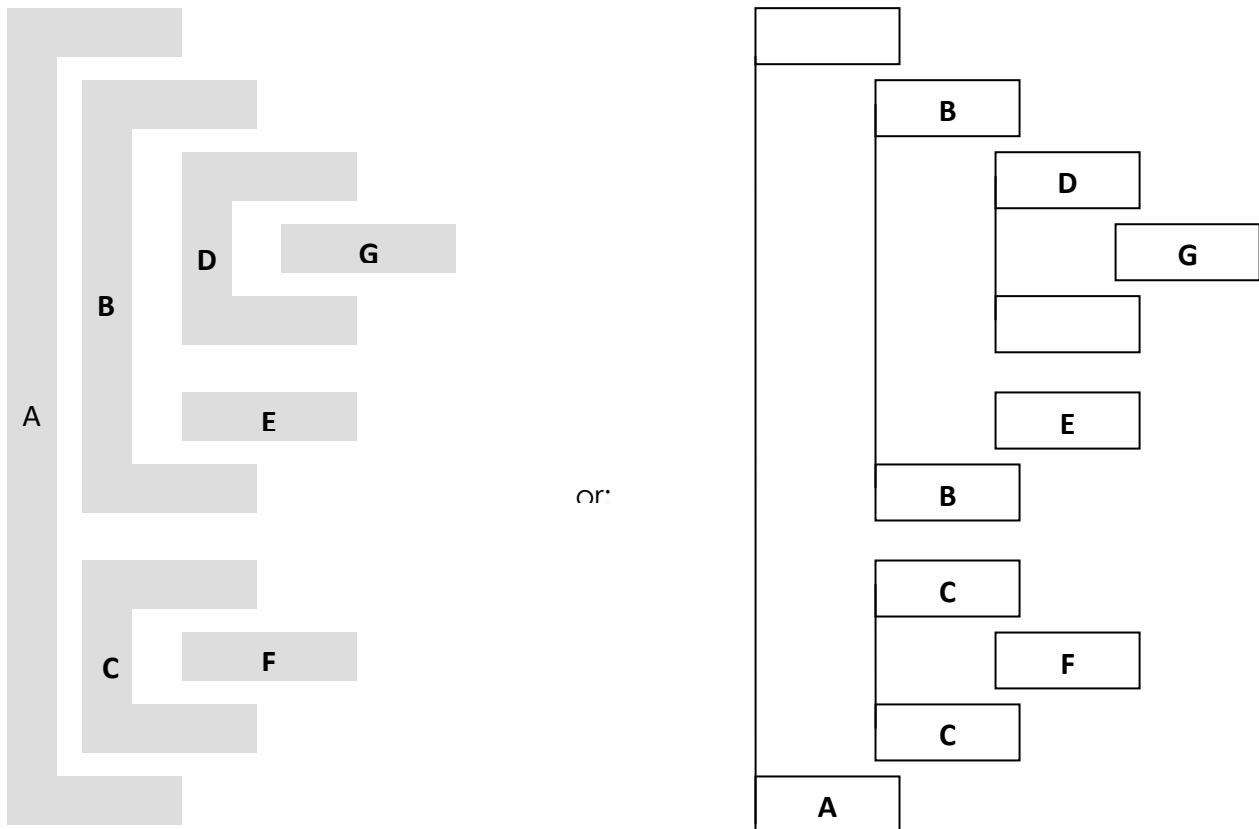
A tree structure is made up of nodes and united by connectors indicating the hierarchic relationship of the nodes:



The following diagram represents the same structure:



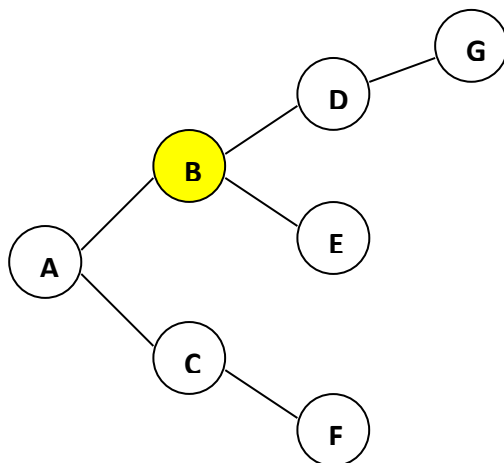
Which is equivalent to:



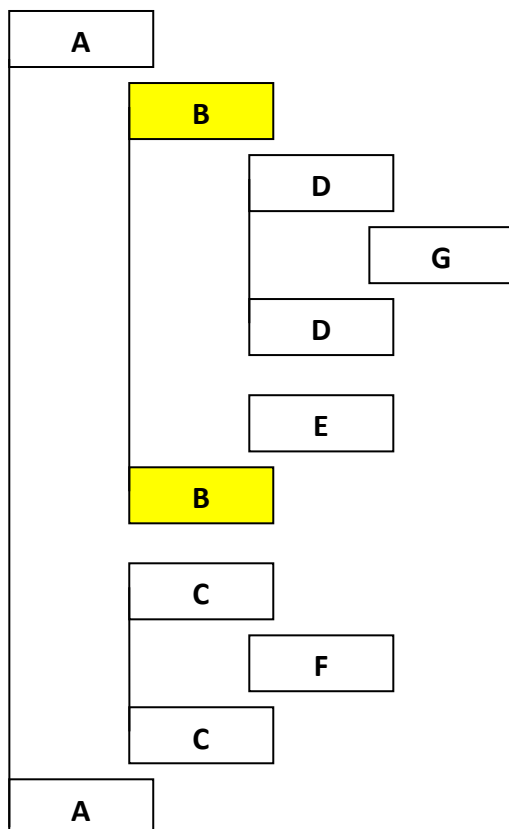
It should be obvious now how we can write this structure in XML:

```
<A>
  <B>
    <D>
      <G/>
    </D>
  <E/>
</B>
<C>
  <F/>
</C>
</A>
```

Please refer back to the diagram of the circles and the connectors. If node **B** is selected, this is what the result will be:



The same can be done in the successive diagrams:



And therefore to:

<A>  
<B>  
<D>

<G/>  
</D>  
<E/>  
</B>  
<C>  
<F/>  
</C>  
</A>

Conclusion: when a node is selected, that node is selected and nothing else; its contents are another issue.

When selecting a set of nodes with an XPath expression, XPath selects **exclusively** the nodes that match the expression, **not** all their descendants. If, in the case of an element, the descendants of a node are contained within its opening and closing tags; they are not the node itself.

With a resulting node-set from an XPath expression, it is possible to visit each one of those nodes, but another XPath expression is required to access the descendants of each one. In this case, the expression will be relative to the current node, the context node.

Consider the following XML document:

```
<?xml version="1.0" encoding="UTF-8"?>
<Bookstore>
  <item type="book">
    <author>
      William Shakespeare
    </author>
    <title>
      Hamlet
    </title>
  </item>
  <item type="CD">
    <author>
      Various
    </author>
    <title>
      Encyclopedic Dictionary
    </title>
  </item>
</Bookstore>
```

The expression `/Bookstore/item`, will select these nodes:

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



```
<item type="book">
  <author>
    William Shakespeare
  </author>
  <title>
    Hamlet
  </title>
</item>
<item type="CD">
  <author>
    Various
  </author>
  <title>
    Encyclopedic Dictionary
  </title>
</item>
</Bookstore>
```

We can only indicate opening tags, if we prefer:

```
<?xml version="1.0" encoding="UTF-8"?>
<Bookstore>
  <item type="book">
    <author>
      William Shakespeare
    </author>
    <title>
      Hamlet
    </title>
  </item>
  <item type="CD">
    <author>
      Various
    </author>
    <title>
      Encyclopedic Dictionary
    </title>
  </item>
</Bookstore>
```

Note that the elements `<author>` or `<title>` are not selected, because although they are within the tags of the `<item>` elements they are descendants of them.

The expression `/Bookstore/item/title` will select (highlighting both tags):

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

```
<item type="book">
  <author>
    William Shakespeare
  </author>
  <title>
    Hamlet
  </title>
</item>
<item type="CD">
  <author>
    Various
  </author>
  <title>
    Encyclopedic Dictionary
  </title>
</item>
</Bookstore>
```

Pay special attention to the fact that only the `<title>` elements are selected and not the text they contain, because those are text nodes, children of the element nodes.

The expression `/Bookstore/item/*` will select (highlighting only the opening tag):

```
<?xml version="1.0" encoding="UTF-8"?>
<Bookstore>
  <item type="book">
    <author>
      William Shakespeare
    </author>
    <title>
      Hamlet
    </title>
  </item>
  <item type="CD">
    <author>
      Various
    </author>
    <title>
      Encyclopedic Dictionary
    </title>
  </item>
</Bookstore>
```

The expression `/Bookstore/*` will select:

```
<?xml version="1.0" encoding="UTF-8"?>
<Bookstore>
  <item type="book">
```

```
<author>
  William Shakespeare
</author>
<title>
  Hamlet
</title>
</item>
<item type="CD">
  <author>
    Various
  </author>
  <title>
    Encyclopedic Dictionary
  </title>
</item>
</Bookstore>
```

The expression `/Bookstore/item/@*` will select:

```
<?xml version="1.0" encoding="UTF-8"?>
<Bookstore>
  <item type="book">
    <author>
      William Shakespeare
    </author>
    <title>
      Hamlet
    </title>
  </item>
  <item type="CD">
    <author>
      Various
    </author>
    <title>
      Encyclopedic Dictionary
    </title>
  </item>
</Bookstore>
```

The expression `/Bookstore/item/*/text()` will select:

```
<?xml version="1.0" encoding="UTF-8"?>
<Bookstore>
  <item type="book">
    <author>
      William Shakespeare
    </author>
    <title>
```

```
    Hamlet
  </title>
</item>
<item type="CD">
  <author>
    Various
  </author>
  <title>
    Encyclopedic Dictionary
  </title>
</item>
</Bookstore>
```

OK, let's compare this case with one of the earlier expressions `/Bookstore/item/title`. Now we are actually selecting the text node within the element nodes.

Finally, the expression `/Bookstore// * | // @* | //text ()` will select:

```
<?xml version="1.0" encoding="UTF-8"?>
<Bookstore>
  <item type="book">
    <author>
      William Shakespeare
    </author>
    <title>
      Hamlet
    </title>
  </item>
  <item type="CD">
    <author>
      Various
    </author>
    <title>
      Encyclopedic Dictionary
    </title>
  </item>
</Bookstore>
```

Because we *specifically* requested element nodes, attribute nodes and text nodes.

Now let's select `/Bookstore/item[2]`:

```
<?xml version="1.0" encoding="UTF-8"?>
<Bookstore>
  <item type="book">
    <author>
      William Shakespeare
```

```
    </author>
    <title>
      Hamlet
    </title>
  </item>
  <item type="CD">
    <author>
      Various
    </author>
    <title>
      Encyclopedic Dictionary
    </title>
  </item>
</Bookstore>
```

Taking this as the context node, either by means of some library processing the DOM or within an XSL style sheet, as we will see later, we will be able to write XPath expressions relative to this node to select other nodes.

In the following examples, the context node will be highlighted in green and the nodes selected by the different XPath expressions, will be highlighted in yellow:

The expression `./author` will select:

```
<?xml version="1.0" encoding="UTF-8"?>
<Bookstore>
  <item type="book">
    <author>
      William Shakespeare
    </author>
    <title>
      Hamlet
    </title>
  </item>
  <item type="CD">
    <author>
      Various
    </author>
    <title>
      Encyclopedic Dictionary
    </title>
  </item>
</Bookstore>
```

The expression `./title/text()` will select:

```
<?xml version="1.0" encoding="UTF-8"?>
<Bookstore>
  <item type="book">
    <author>
```

```
    William Shakespeare
  </author>
  <title>
    Hamlet
  </title>
</item>
<item type="CD">
  <author>
    Various
  </author>
  <title>
    Encyclopedic Dictionary
  </title>
</item>
</Bookstore>
```

Other axes can be used to select other "relatives of the same family", for example `./parent::* /item [1] /@*` which results in the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<Bookstore>
  <item type="book">
    <author>
      William Shakespeare
    </author>
    <title>
      Hamlet
    </title>
  </item>
  <item type="CD">
    <author>
      Various
    </author>
    <title>
      Encyclopedic Dictionary
    </title>
  </item>
</Bookstore>
```

## 18. Transformations

### XSLT

As outlined above, the utility of XML is largely due to its flexibility and versatility. One of the enormous advantages that leverage these characteristics is the ability to use style sheets that provide a simple way to perform transformations on XML documents; to adapt the same source data to different needs.

XSL is the acronym for eXtensible Stylesheet Language. XSL style sheets are divided into two great groups: XSLT supports all types of transformations on the original XML document that is oriented to the presentation formatting of the XML data.

XSLT or **eXtensible Stylesheet Language Transformations** is a W3C standard programming language used to transform the structure of XML documents, going far beyond the simple presentation mode. XSLT allows us to transform the structure and even the content of an XML document resulting in another XML document, a HTML document, or a text document. The following should be noted:

- **XSLT are XML documents themselves.** They can be applied to an XML document for use by another application when receiving the document before processing its content, becoming a fundamental resource for application integration.
- **XSLT is a declarative language.** Its paradigm is conceptually different from other languages (procedural, OO, etc). Under this paradigm, the rules to apply are simply described and the processor determines the execution sequence.
- **A XSLT consists of a set of rules that determine how the XML document will be processed.** These rules can appear in any order, without concern for either the order of the source document or that of the output. By default, XSLT are processed according to the hierarchical order of the document.

The root element of the XSLT is `<xsl:stylesheet>`. The namespace `xsl` determines the elements of the language. Its declaration will be:

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  ...
</xsl:stylesheet>
```

XSLT is based on groups of rules or templates. The programming consists of defining these rules. They are used to determine what nodes in the source document must be transformed and to specify how those nodes will be transformed to produce the desired output document.

These rules consist of two parts:

- To define the transformation itself
- To define the set or sets of nodes the transformation will be applied to

In order to define a rule, the `<xsl:template>` element is used. This has an attribute `match` used to identify nodes in the source document, by means of XPath of course, on which the transformation

will be applied. Within this specific element, the transformation that will be performed to the nodes is defined, that is to say, the output template:

```
<xsl:template match=criteria>
    ... output template ...
</xsl:template>
```

A template for the root element of the document must always be specified:

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

We will use a simple XSLT to explain its main elements, using the following source XML document:

```
<?xml version="1.0" encoding="UTF-8"?>
<Employees>
  <Section>Integration Department</Section>
  <Person gender="M">
    <Name>Dorian</Name>
    <LastName>Miller</LastName>
    <Document Type="PAS">8765432</Document>
  </Person>
  <Person gender="M">
    <Name>Herbert</Name>
    <LastName>Marshall</LastName>
    <Document Type="SSI">2345678</Document>
  </Person>
  <Person gender="F">
    <Name>Alice</Name>
    <LastName>Marshall</LastName>
    <Document Type="SSI">17456789</Document>
  </Person>
  <Person gender="F">
    <Name>Christine</Name>
    <LastName>Davis</LastName>
    <Document Type="PAS">6123456</Document>
  </Person>
</Employees>
```

The `<xsl:output>` element allows specification the format of the document resulting from the transformation. Its attribute **method** indicates if it will be XML, HTML or flat text. The output character set (encoding) can also be indicated.

```
<xsl:output method="html" encoding="UTF-8"/>
```



Within an output template new elements can be created "from nothing" by simply including them in the template.

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

    <HTML/>

</xsl:template>
```

The' XSLT so far:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="html" encoding="UTF-8" />
  <xsl:template match="/">
    <HTML/>
  </xsl:template>
</xsl:stylesheet>
```

This SimpleXSL.xslt example can be made available in the same file location as the source XML document. The XSLT is assigned to the XML document as seen before:

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

The XML document can now be loaded into the Web browser to see the result. However, nothing will be visible since the transformation will give the following resulting document:

```
<HTML/>
```

Another way to create output elements is by means of the `<xsl:element>` element. It has an attribute **name** that allows the assigning of names. Its content is created as child nodes.

```
<xsl:element name="HTML">
  <xsl:element name="HEAD"/>
  <xsl:element name="BODY"/>
</xsl:element>
```

The simplest way to produce output from a node of the source document is to use the `<xsl:value-of>` element. This has an attribute **select** that is used to specify the value of output. Any valid XPath expression can be assigned to this attribute.

```
<xsl:value-of select="/Employees/Section"/>
```

After adding a heading, the sample XSLT will now be:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="html" encoding="UTF-8" />
  <xsl:template match="/">
    <xsl:element name="HTML">
      <xsl:element name="HEAD"/>
      <xsl:element name="BODY">
        <xsl:element name="H2">
          <xsl:value-of select="/Employees/Section"/>
        </xsl:element>
      </xsl:element>
    </xsl:element>
  </xsl:template>
</xsl:stylesheet>
```

The output that will be produced is:

```
<HTML>
  <HEAD/>
  <BODY>
    <H2>Integration Department</H2>
  </BODY>
</HTML>
```

In a Web browser, one would now see:

## Integration Department



*Note that the source code of the Web page will remain the source XML document. The Web browser performs the transformation in memory and displays the result, but it does not save the output document. There are several applications and libraries that have the capability to produce and persist the output document.*

Nodes not meeting the selection criteria of any of the templates will not appear in the output.

In the output templates, it is also possible to declare that other templates must be applied to certain sets of nodes. This is done by means of the **<xsl:apply-templates>** element. It has an attribute **select** where we indicate, by means of XPath, the sets of nodes the templates must be applied to. The XSLT should have some template in whose attribute **match** an equivalent expression appears. This way, it is as if this template "has been called", and the selected set of nodes "has been passed" as the argument. The expression of the attribute **select** can be more specific than that of the attribute **match**, thus a template can be used in different circumstances.

For this example, only one template to deal with the **<Person>** elements is defined:

```
<xsl:template match="//Person">
  <tr>
    <td>
      <xsl:value-of select="Name"/>
    </td>
    <td>
      <xsl:value-of select="LastName"/>
    </td>
  </tr>
</xsl:template>
```

With this template rows of a table are created.

Once a node is selected in the template (in this case each one of the **<Person>** elements which will be processed one by one), it becomes the context node. Then, the selection of other nodes will be relative to this context node. Thus, the XPath expressions **Name** and **LastName** are relative to the context element **<Person>**.

Each template is defined separately, similar in a way to a subroutine, and each template is defined for a certain set of nodes.

Applying this template to two different sets of nodes, separately:

```
<xsl:apply-templates select="/Employees/Person[@gender='F']"/>
```

and

```
<xsl:apply-templates select="/Employees/Person[@gender='M']"/>
```

After adding a few cosmetics, the XSLT will finally be:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="html" encoding="UTF-8"/>

  <xsl:template match="/">
    <xsl:element name="HTML">
      <xsl:element name="HEAD"/>
      <xsl:element name="BODY">
        <xsl:element name="H2">
          <xsl:value-of select="/Employees/Section"/>
        </xsl:element>
        <xsl:element name="H3">Ladies</xsl:element>
        <table border="1">
          <xsl:apply-templates select="/Employees/Person[@gender='F']"/>
        </table>
        <xsl:element name="H3">Gentlemen</xsl:element>
      </xsl:element>
    </xsl:element>
  </xsl:template>
```

```
<table border="1">
  <xsl:apply-templates select="/Employees/Person[@gender='M']"/>
</table>
</xsl:element>
</xsl:element>
</xsl:template>

<xsl:template match="//Person">
  <tr>
    <td>
      <xsl:value-of select="Name"/>
    </td>
    <td>
      <xsl:value-of select="LastName"/>
    </td>
  </tr>
</xsl:template>
</xsl:stylesheet>
```

The resulting document of the transformation will be:

```
<?xml version="1.0" encoding="UTF-8"?>
<HTML>
  <HEAD/>
  <BODY>
    <H2>Integration Department</H2>
    <H3>Ladies</H3>
    <table border="1">
      <tr>
        <td>Alice</td>
        <td>Marshall</td>
      </tr>
      <tr>
        <td>Cándida</td>
        <td>Malerva</td>
      </tr>
    </table>
    <H3>Gentlemen</H3>
    <table border="1">
      <tr>
        <td>Dorian</td>
        <td>Miller</td>
      </tr>
      <tr>
        <td>Herbert</td>
        <td>Marshall</td>
      </tr>
    </table>
  </BODY>
```

**</HTML>**

The document will now be displayed in a Web browser like this:

<b>Integration Department</b>	
<b>Ladies</b>	
Alice	Marshall
Christine	Davis
<b>Gentlemen</b>	
Dorian	Miller
Herbert	Marshall

The detailed study of XSLT and all its programming capabilities exceed the scope of this Course. Nevertheless, here are some of its capabilities:

- Elements can be filtered with any XPath expression
- Sets of elements can be ordered
- Calculations and functions can be applied to the content of the elements and/or attributes
- Variables can be defined and used
- Loops can be performed
- Decision structures can be used

A very important feature is that we can create new nodes, which did not exist in the source document:

- A single element can be mapped to multiple elements
- Names of elements and attributes can be created dynamically
- Elements or their content can be created as a result of applying functions or templates on other elements

## Unit Summary and Conclusion

XML is the ideal mechanism for sharing data between systems, being a global standard and supporting any format of information representation. Conceptually, it is simple and the mechanisms for document manipulation are powerful.

XML use has been growing rapidly and multiple tools exist across all platforms to facilitate its use.

XML has already become the common language between most heterogeneous systems and it is being used for more and more applications. It has even been adopted as the low-level support for other standards. Therefore, knowledge of its capability and implementation has become indispensable in the information systems environments.

In this Unit, we have been learning about generic XML. In other Units of this Course, we will study how to apply XML to HL7 Version 2.x messages; how CDA makes extensive use of XML, how XML is used to convey V3 messages and how XML is used in the new FHIR Standard.