M.SC (CS) II YEAR

XML

HTML	**XML**
HTML is an abbreviation for HyperText Markup Language.	XML stands for eXtensible Markup Language.
HTML was designed to display data with focus on how data looks.	XML was designed to be a software and hardware independent tool used to transport and store data, with focus on what data is.
HTML is a markup language itself.	XML provides a framework for defining markup languages.
HTML is a presentation language.	XML is neither a programming language nor a presentation language.
HTML is case insensitive.	XML is case sensitive.
HTML is used for designing a web-page to be rendered on the client side.	XML is used basically to transport data between the application and the database.
HTML has it own predefined tags.	While what makes XML flexible is that custom tags can be defined and the tags are invented by the author of the XML document.
HTML is not strict if the user does not use the closing tags.	XML makes it mandatory for the user the close each tag that has been used.
HTML does not preserve white space.	XML preserves white space.
HTML is about displaying data,hence static.	XML is about carrying information,hence dynamic.

HTML is an abbreviation for HyperText Markup Language. **HTML** was designed to display data with focus on how data looks. **XML** was designed to be a software and hardware independent tool used to transport and store data, with focus on what data is.**XML** provides a framework for defining markup languages.

What is the difference between XML and HTML?

▪

- XML is the acronym from Extensible Markup Language (meta-language of noting/marking). XML is a resembling language with HTML. It was developed for describing data.
- The XML tags are not pre-defined in XML. You will have to create tags according to your needs.
- XML is self descriptive.
- XML uses DDT principle (Defining the Document Type) to formally describe the data.
- The main difference between XML and HTML: XML is not a substitute for HTML.

XML and HTML were developed with different purposes:

- XML was developed to describe data and to focalize on what the data represent.
- HTML was developed to display data about to focalize on the way that data looks.
- HTML is about displaying data, XML is about describing information.
- XML is extensible.

The tags used to mark the documents and the structures of documents in HTML are pre-defined. The author of HTML documents can use only tags that were previously defined in HTML. The Standard XML gives you the possibility to define personal structures and tags.

XML is a complement of the HTML language

It is important to understand that XML is not a substitute for HTML. In the future development of the Web, XML will be the main language to describe the structure and the Web data, and the HTML language will be responsible for displaying the data.

XML in the future development of web

We participated to the evolution of XML since its appearance. It is amazing to observe its rapid evolution, and how fast it was adopted by the majority of software developers. We strongly believe that XML will become as important as HTML for the future web evolution, especially when it comes to data manipulation.

How can you use XML?

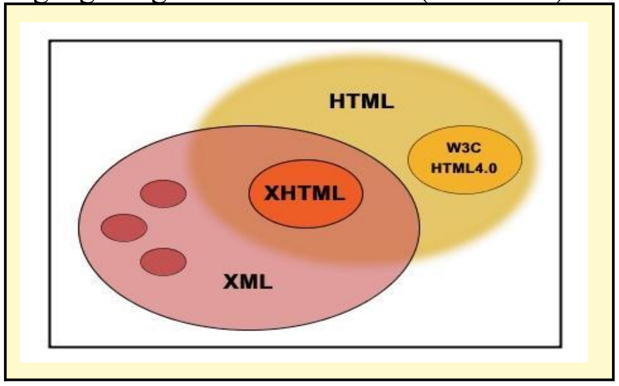
- XML can store data separately from HTML.
- XML can be used to store data inside the HTML documents.
- XML can be used as a format for exchanging information.
- XML can be used to store data in files and databases.

The HTML pages are used to display data. The data are sometimes stored in the interior of HTML pages. Using XML, you can store data in a separated file. This way, you can easily concentrate on using HTML for formatting and displaying, and you can also be certain that the modifications won’t bring any modifications to any HTML code.
XML can also store data inside the HTML documents. The XML data will store data in HTML documents as “data islands.” You can concentrate on HTML to format and display data.

In the real world, the calculation systems and the databases are containing data in incompatible formats. One of the most soliciting provocations for developers was exchanging data between incompatible systems over the internet. Converting XML data can reduce the complexity, and it can also create data that are easy to read by any kind of application.
XML can be used to store data in files or databases. You can write applications to store and recover information from the hard disks, and you can write generic application to display certain types of data.

Relation between XML and HTML

- XML is defined as a syntax
- HTML is defined as a vocabulary
- A given markup language might be either or both (or neither)



Comparison Chart

BASIS FOR COMPARISON	**XML**	**HTML**
Expands to	Extensible Markup Language	Hypertext Markup Language
Basic	Provides a framework for specifying markup languages.	HTML is predefined markup language.
Structural	Information Provided	Does not contain structural information
Language type	Case sensitive	Case insensitive
Purpose of the language	Transfer of information	Presentation of the data
Errors	Not allowed	Small errors can be ignored.
Whitespace	Can be preserved.	Does not preserve white spaces.

Closing tags	Compulsory to use closing tags.	Closing tags are optional.
Nesting	Must be properly done.	Not much

Goals for XML

The purpose of XML is to support dependable, long-term storage, transmission, conversion, and consumption of data.
To do this, XML:

- provides a flexible, well-defined structure for holding many different kinds of data (the 'tree', in short)
- methods of describing the data with element and attributes names and with schemas ('markup, in short')

The design goals for XML are:

1. XML shall be straightforwardly usable over the Internet.
2. XML shall support a wide variety of applications.
3. XML shall be compatible with SGML.
4. It shall be easy to write programs which process XML documents.
5. The number of optional features in XML is to be kept to the absolute minimum, ideally zero.
6. XML documents should be human-legible and reasonably clear.
7. The XML design should be prepared quickly.
8. The design of XML shall be formal and concise.
9. XML documents shall be easy to create.
10. Terseness in XML markup is of minimal importance.

WELL FORMED XML

An XML document is called well-formed if it satisfies certain rules, specified by The W3C(World Wide Web Consortium)
.
These rules are:

- A well-formed XML document must have a corresponding end tag for all of its start tags.
- Nesting of elements within each other in an XML document must be proper. For example, <tutorial><topic>XML</topic></tutorial> is a correct way of nesting but <tutorial><topic>XML</tutorial></topic> is not.
- In each element two attributes must not have the same value. For example, <tutorial id="001"><topic>XML</topic></tutorial> is right,but <tutorial id="001" id="w3r"><topic>XML</topic></tutorial> is incorrect.
- Markup characters must be properly specified. For example, <tutorial id="001"><topic>XML</topic></tutorial> is right, not <tutorial id="001" id="w3r"><topic>XML</topic></tutorial>.

- An XML document can contain only one root element. So, the root element of an xml document is an element which is present only once in an xml document and it does not appear as a child element within any other element.

Example of a Valid XML document

```
<?xml
version="1.0" ?>
<w3resource>
<design>
html
xhtml
css
svg
xml
</design>
<programming>
php
mysql
</programming>
</w3resource>
```

To check whether an XML document is well-formed or not, you have to use an XML parser. An XML parser is a Software Application, which introspects an XML document and stores the resulting output in memory. If an XML document is parsed by an XML parser and some irregularities are found, it generates some errors and it is then denoted as an XML file , which is not a well-formed XML document.
Well-formed XML is an XML document which follows some rules specified by W3C.

XML – DTDs

A Document Type Definition (DTD) is a document that describes the structure of an XML document, what elements and attributes it contains and what values they may have.
DTD's form part of the W3C's XML Standard, but are typically considered to be a separate schema technology and are not typically used in conjunction with other schema formats like XSD, RelaxNG etc.
A DTD document has its own syntax and grammar, the rules are simple, but with because definitions for elements and attributes are split, making sense of the document can be time consuming to process by hand. Furthermore if entity references are used, then a recursive pre-processing step may also need to be applied which can make it very difficult to interpret them by hand (see Tricky DTD Sample).
A DTD document can be embedded within an XML file or can exist on its own. Common definitions can also be broken out into files and included in order to increase re-use or maintainability.
The XML Document Type Declaration, commonly known as DTD, is a way to describe XML language precisely. DTDs check vocabulary and validity of the structure of XML documents against grammatical rules of appropriate XML language.

An XML DTD can be either specified inside the document, or it can be kept in a separate document and then liked separately.

Syntax

Basic syntax of a DTD is as follows −

```
<!DOCTYPE element DTD identifier
[
   declaration1
   declaration2
   ........
]>
```

In the above syntax,

- The **DTD** starts with <!DOCTYPE delimiter.
- An **element** tells the parser to parse the document from the specified root element.
- **DTD identifier** is an identifier for the document type definition, which may be the path to a file on the system or URL to a file on the internet. If the DTD is pointing to external path, it is called **External Subset.**
- **The square brackets [ ]** enclose an optional list of entity declarations called *Internal Subset*.

Internal DTD

A DTD is referred to as an internal DTD if elements are declared within the XML files. To refer it as internal DTD, *standalone* attribute in XML declaration must be set to **yes**. This means, the declaration works independent of an external source.

Syntax

Following is the syntax of internal DTD −

```
<!DOCTYPE root-element [element-declarations]>
```

where *root-element* is the name of root element and *element-declarations* is where you declare the elements.

Example

Following is a simple example of internal DTD −

```
<?xml version = "1.0" encoding = "UTF-8" standalone = "yes" ?>
<!DOCTYPE address [
   <!ELEMENT address (name,company,phone)>
   <!ELEMENT name (#PCDATA)>
   <!ELEMENT company (#PCDATA)>
   <!ELEMENT phone (#PCDATA)>
]>

<address>
   <name>Tanmay Patil</name>
   <company>TutorialsPoint</company>
   <phone>(011) 123-4567</phone>
</address>
```

Let us go through the above code −

Start Declaration − Begin the XML declaration with the following statement.

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

DTD − Immediately after the XML header, the *document type declaration*follows, commonly referred to as the DOCTYPE −

```
<!DOCTYPE address [
```

The DOCTYPE declaration has an exclamation mark (!) at the start of the element name. The DOCTYPE informs the parser that a DTD is associated with this XML document.
DTD Body − The DOCTYPE declaration is followed by body of the DTD, where you declare elements, attributes, entities, and notations.

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

Several elements are declared here that make up the vocabulary of the <name> document. <!ELEMENT name (#PCDATA)> defines the element *name*to be of type "#PCDATA". Here #PCDATA means parse-able text data.
End Declaration − Finally, the declaration section of the DTD is closed using a closing bracket and a closing angle bracket (**]>**). This effectively ends the definition, and thereafter, the XML document follows immediately.
Rules

- The document type declaration must appear at the start of the document (preceded only by the XML header) − it is not permitted anywhere else within the document.
- Similar to the DOCTYPE declaration, the element declarations must start with an exclamation mark.
- The Name in the document type declaration must match the element type of the root element.

External DTD

In external DTD elements are declared outside the XML file. They are accessed by specifying the system attributes which may be either the legal *.dtd* file or a valid URL. To refer it as external DTD, *standalone* attribute in the XML declaration must be set as **no**. This means, declaration includes information from the external source.
Syntax
Following is the syntax for external DTD −

```
<!DOCTYPE root-element SYSTEM "file-name">
```

where *file-name* is the file with *.dtd* extension.
Example
The following example shows external DTD usage −

```
<?xml version = "1.0" encoding = "UTF-8" standalone = "no" ?>
<!DOCTYPE address SYSTEM "address.dtd">
<address>
  <name>Tanmay Patil</name>
  <company>TutorialsPoint</company>
  <phone>(011) 123-4567</phone>
</address>
```

The content of the DTD file **address.dtd** is as shown −

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

Types

You can refer to an external DTD by using either **system identifiers** or **public identifiers**.

System Identifiers

A system identifier enables you to specify the location of an external file containing DTD declarations. Syntax is as follows −

```
<!DOCTYPE name SYSTEM "address.dtd" [...]>
```

As you can see, it contains keyword SYSTEM and a URI reference pointing to the location of the document.

Public Identifiers

Public identifiers provide a mechanism to locate DTD resources and is written as follows −

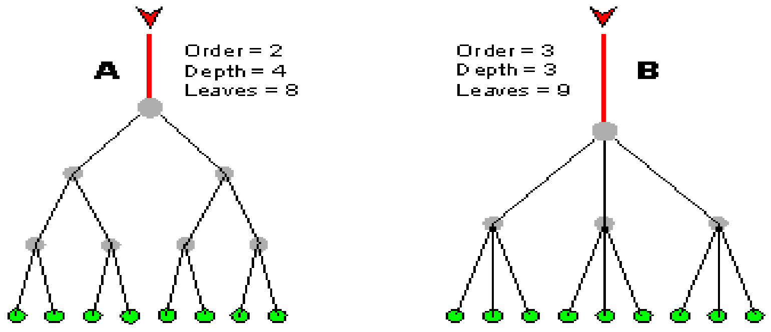
```
<!DOCTYPE name PUBLIC "-//Beginning XML//DTD Address Example//EN">
```

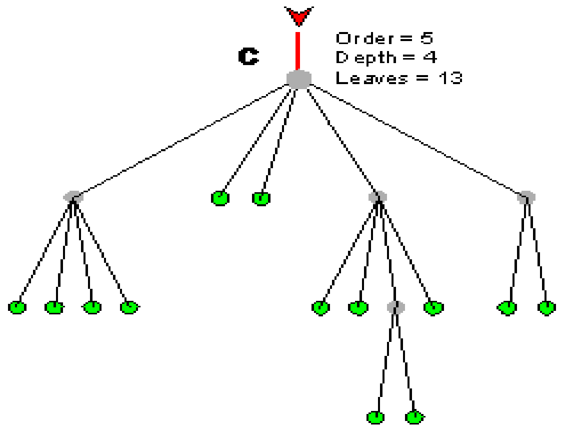
As you can see, it begins with keyword PUBLIC, followed by a specialized identifier. Public identifiers are used to identify an entry in a catalog. Public identifiers can follow any format, however, a commonly used format is called **Formal Public Identifiers, or FPIs**.

Tree structure in data organization

A tree structure is an algorithm for placing and locating files (called records or keys) in a database. The algorithm finds data by repeatedly making choices at decision points called nodes. A node can have as few as two branches (also called children), or as many as several dozen. The structure is straightforward, but in terms of the number of nodes and children, a tree can be gigantic.

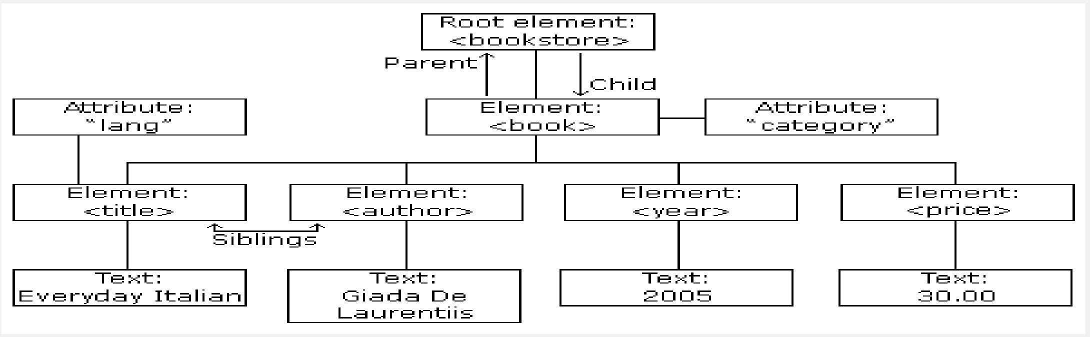
In a tree, records are stored in locations called leaves. This name derives from the fact that records always exist at end points; there is nothing beyond them. The starting point is called the root. The maximum number of children per node is called the order of the tree. The maximum number of access operations required to reach the desired record is called the depth. In some trees, the order is the same at every node and the depth is the same for every record. This type of structure is said to be balanced. Other trees have varying numbers of children per node, and different records might lie at different depths. In that case, the tree is said to have an unbalanced or asymmetrical structure.





The illustration shows three examples of tree structures. (Note that the portrayals are upside-down compared to real tree plants.) Structures A and B are balanced, and structure C is unbalanced. Roots are at the top, and are represented by red arrows and red lines. Nodes are shown as gray dots. Children are solid black lines. Leaves are at the bottom, and are represented by green dots. As the process moves toward the leaves and away from the root, children can branch out from a node, but children never merge into a node.
In a practical tree, there can be thousands, millions, or billions of nodes, children, leaves, and records. Not every leaf necessarily contains a record, but more than half do. A leaf that does not contain data is called a null. The trees shown here are simple enough to be rendered in two dimensions, but with some large databases, three dimensions are needed to clearly depict the structure.
XML documents form a tree structure that starts at "the root" and branches to "the leaves".

XML Tree Structure



```xml
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>
  <book category="cooking">
    <title lang="en">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year>
    <price>30.00</price>
  </book>
  <book category="children">
    <title lang="en">Harry Potter</title>
    <author>J K. Rowling</author>
    <year>2005</year>
    <price>29.99</price>
  </book>
  <book category="web">
    <title lang="en">Learning XML</title>
    <author>Erik T. Ray</author>
    <year>2003</year>
    <price>39.95</price>
  </book>
</bookstore>
```

XML Tree Structure

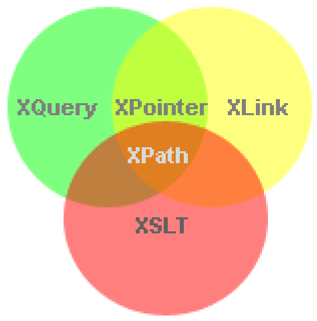
XML documents are formed as **element trees**.
An XML tree starts at a **root element** and branches from the root to **child elements**.
All elements can have sub elements (child elements):

```xml
<root>
  <child>
    <subchild>.....</subchild>
  </child>
</root>
```

The terms parent, child, and sibling are used to describe the relationships between elements.
Parents have children. Children have parents. Siblings are children on the same level (brothers and sisters).
All elements can have text content (Harry Potter) and attributes (category="cooking").

XPath

XPath is a major element in the XSLT standard.
XPath can be used to navigate through elements and attributes in an XML document.



- XPath stands for XML Path Language
- XPath uses "path like" syntax to identify and navigate nodes in an XML docum
- XPath contains over 200 built-in functions
- XPath is a major element in the XSLT standard
- XPath is a W3C recommendation

Searching and filtering with XPath

array **xpath** (string *path*)

The standard way for searching through XML documents for particular nodes is called XPath, and Sterling Hughes (the creator of the SimpleXML extension) described it saying it's "as important to XML as regular expressions are to plain text".

Fortunately for us, XPath is a darn sight easier than regular expressions for basic usage. That said, it might take a little while to get your head around all the possibilities it opens up to you!

Using the same employees.xml file, give this script a try:

```
<?php
   $xml = simplexml_load_file('employees.xml');

   echo "<strong>Using direct method...</strong><br />";
   $names = $xml->xpath('/employees/employee/name');
   foreach($names as $name) {
      echo "Found $name<br />";
   }
   echo "<br />";

   echo "<strong>Using indirect method...</strong><br />";
   $employees = $xml->xpath('/employees/employee');
   foreach($employees as $employee) {
      echo "Found {$employee->name}<br />";
   }
   echo "<br />";

   echo "<strong>Using wildcard method...</strong><br />";
   $names = $xml->xpath('//name');
   foreach($names as $name) {
      echo "Found $name<br />";
   }
?>
```

What that does is pull out names of employees in three different ways. The key real work is done in the call to the *xpath()* function. As you can see in the prototype, *xpath()* takes a query as its only parameter, and returns the result of that query.

The query itself has specialised syntax, but it's very easy. The first example says "Look in all the employees elements, find any employee elements in there, and retrieve all the names of them." It's very specific because only employees/employee/name is matched.
The second query matches all employee elements inside employees, but doesn't go specifically for the name of the employees. As a result, we get the full employee back, and need to print $employee->name to get the name.
The last one just looks for name elements, but note that it starts with "//" - this is the signal to do a global search for all name elements, regardless of where they are or how deeply nested they are in the document.
So, what we have here is the ability to grab specific parts of a document very easily, but that's really only the start of XPath's coolness. You see, you can also use it to filter your results according to any values you want. Try this script out:

```
<?php
   $xml = simplexml_load_file('employees.xml');

   echo "<strong>Matching employees with name 'Laura Pollard'</strong><br />";
   $employees = $xml->xpath('/employees/employee[name="Laura Pollard"]');

   foreach($employees as $employee) {
      echo "Found {$employee->name}<br />";
   }

   echo "<br />";

   echo "<strong>Matching employees younger than 54</strong><br />";
   $employees = $xml->xpath('/employees/employee[age<54]');

   foreach($employees as $employee) {
      echo "Found {$employee->name}<br />";
   }

   echo "<br />";

   echo "<strong>Matching employees as old or older than 48</strong><br />";
   $employees = $xml->xpath('//employee[age>=48]');

   foreach($employees as $employee) {
      echo "Found {$employee->name}<br />";
   }

   echo "<br />";

?>
```

Let's break that down to see how the querying actually works. The key part, is, of course, between the square brackets, [and]. The first query grabs all employees elements, then all

employee elements inside it, but then filters them so that only those that have a name that matches Laura Pollard. Once you get that, the other two are quite obvious: <, >, <=, etc, all work as you'd expect in PHP. Note that I slipped in a double slash in the last example to show you that the global search notation works here too.
You can grab only part of a query result by continuing on as normal afterwards, like this:

```
$ages = $xml->xpath('//employee[age>=48]/age');

foreach($ages as $age) {
   echo "Found $age ";
}
```

You can even run queries on queries, with an XPath search like this:

```
$employees = $xml->xpath('//employee[age>=49][name="Laura Pollard"]');
```

Going back to selecting various types of elements, you can use the | symbol (OR) to select more than one type of element, like this:

```
echo " Retrieving all titles and ages ";
$results = $xml->xpath('//employee/title|//employee/age');

foreach($results as $result) {
   echo "Found $result ";
}
```

That will output the following:

```
Found Chief Information Officer
Found 48
Found Chief Executive Officer
Found 54
```

You can, of course, combine all of this together to do search on more than one value, like this:

```
$names = $xml->xpath('//employee[age<40]/name|//employee[age>50]/name');

foreach($names as $name) {
   echo "Found $name ";
}
```

For maximum insanity, you can actually run calculations using XPath in order to get tighter control over your queries. For example, if you only wanted the names of employees who have an odd age (that is, cannot be divided by two without leaving a remainder) you would use an XPath query like this:

```
$names = $xml->xpath('//employee[age mod 2 = 1]/name');
```

Along with "mod" (equivalent to % in PHP) there's also "div" for division, + and - (same as PHP, except that - must always have whitespace either side of it as it may be confused with an element name), and *ceiling()* and *floor()* (equivalent to *ceil()* and *floor()* in PHP).