Chapter – 5

Extensible Markup Language (XML)

INTRODUCTION:

XML stands for **E**xtensible **M**arkup **L**anguage. It is a text-based markup language derived from Standard Generalized Markup Language (SGML).

XML tags identify the data and are used to store and organize the data, rather than specifying how to display it like HTML tags, which are used to display the data. XML is not going to replace HTML in the near future, but it introduces new possibilities by adopting many successful features of HTML.

There are three important characteristics of XML that make it useful in a variety of systems and solutions:

- ☑ **XML is extensible:** XML allows us to create our own self-descriptive tags, or language, that suits our application.
- ☑ **XML carries the data, does not present it:** XML allows us to store the data irrespective of how it will be presented.
- ☑ **XML is a public standard:** XML was developed by an organization called the World Wide Web Consortium (W3C) and is available as an open standard.

SYNTAX:

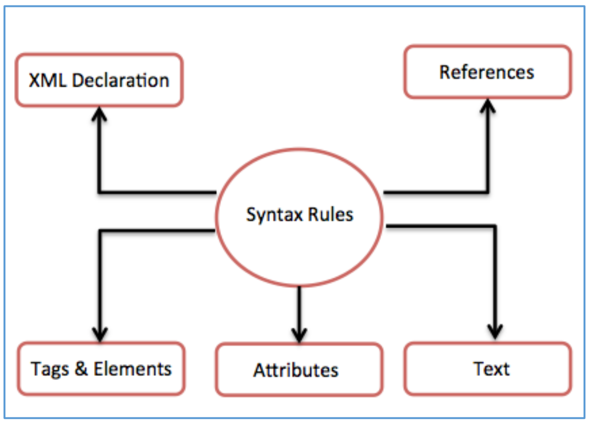
Following is a complete XML document:

```
<?xml version="1.0"?>
    <contact-info>
    <name>Ramesh Pandey</name>
    <company>Insoft</company>
    <phone>(+977) 9806587733 </phone>
</contact-info>
```

We can notice there are two kinds of information in the above example:

- ➢ markup, like <contact-info> and
- ➢ The text, or the character data, Insoft and (+977) 9806587733.

The diagram depicts the syntax rules to write different types of markup and text in an XML document.



Let us see each component of the above diagram in detail:

XML DECLARATION:

The XML document can optionally have an XML declaration. It is written as below:

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

Where *version* is the XML version and *encoding* specifies the character encoding used in the document.

SYNTAX RULES FOR XML DECLARATION:

- The XML declaration is case sensitive and must begin with "<?xml>" where "xml" is written in lower-case.
- If document contains XML declaration, then it strictly needs to be the first statement of the XML document.
- The XML declaration strictly needs be the first statement in the XML document.
- An HTTP protocol can override the value of *encoding* that you put in the XML declaration.

TAGS AND ELEMENTS:

An XML file is structured by several XML-elements, also called XML-nodes or XML-tags. XML-elements' names are enclosed by triangular brackets < > as shown below:

```
<element>
```

SYNTAX RULES FOR TAGS AND ELEMENTS

1. **Element Syntax:**

Each XML-element needs to be closed either with start or with end elements as shown below:

```
<element>....</element>
```

2. **Nesting Of Elements:**

An XML-element can contain multiple XML-elements as its children, but the children elements must not overlap. i.e., an end tag of an element must have the same name as that of the most recent unmatched start tag.

Following example shows incorrect nested tags:

```
<?xml version="1.0"?>
<contact-info>
<company>TutorialsPoint
<contact-info>
</company>
```

Following example shows correct nested tags:

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

```
<contact-info>
<company>TutorialsPoint</company>
<contact-info>
```

3. Root Element:

An XML document can have only one root element. For example, following is not a correct XML document, because both the x and y elements occur at the top level without a root element:

```
<x>...</x>
<y>...</y>
```

The following example shows a correctly formed XML document:

```
<root>
  <x>...</x>
  <y>...</y>
</root>
```

4. Case Sensitivity:

The names of XML-elements are case-sensitive. That means the name of the start and the end elements need to be exactly in the same case. For example **<contact-info>** is different from **<Contact-Info>**.

ATTRIBUTES:

An **attribute** specifies a single property for the element, using a name/value pair. An XML-element can have one or more attributes. For example:

```
<a href="http://www.tutorialspoint.com/">Tutorialspoint!</a>
```

Here *href* is the attribute name and *http://www.tutorialspoint.com/* is attribute value.

Syntax Rules for XML Attributes:

- Attribute names in XML (unlike HTML) are case sensitive. That is, *HREF* and *href* are considered two different XML attributes.
- Same attribute cannot have two values in a syntax. The following example shows incorrect syntax because the attribute *b* is specified twice: <a b="x" c="y" b="z">....</a>
- Attribute names are defined without quotation marks, whereas attribute values must always appear in quotation marks. Following example demonstrates incorrect xml syntax: <a b=x>....</a>
- In the above syntax, the attribute value is not defined in quotation marks.

XML REFERENCES:

References usually allow us to add or include additional text or markup in an XML document. References always begin with the symbol **"&",** which is a reserved character and end with the symbol **";"**. XML has two types of references:

1. **Entity References:**

An entity reference contains a name between the start and the end delimiters. For example **&amp;** where *amp* is *name*. The *name* refers to a predefined string of text and/or markup.

2. **Character References:**

These contain references, such as **&#65;** contains a hash mark ("#") followed by a number. The number always refers to the Unicode code of a character. In this case, 65 refers to alphabet "A".

XML TEXT:

- ➢ The names of XML-elements and XML-attributes are case-sensitive, which means the name of start and end elements need to be written in the same case.
- ➢ To avoid character encoding problems, all XML files should be saved as Unicode UTF-8 or UTF-16 files.
- ➢ Whitespace characters like blanks, tabs and line-breaks between XML-elements and between the XML-attributes will be ignored.
- ➢ Some characters are reserved by the XML syntax itself. Hence, they cannot be used directly. To use them, some replacement-entities are used, which are listed below:

Not Allowed Character	Replacement-Entity	Character Description
<	&lt;	less than
>	&gt;	greater than
&	&amp;	ampersand
'	&apos;	apostrophe
"	&quot;	quotation mark

XML USAGE:

- ☑ XML can work behind the scene to simplify the creation of HTML documents for large web sites.
- ☑ XML can be used to exchange the information between organizations and systems.
- ☑ XML can be used for offloading and reloading of databases.
- ☑ XML can be used to store and arrange the data, which can customize your data handling needs.
- ☑ XML can easily be merged with style sheets to create almost any desired output.
- ☑ Virtually, any type of data can be expressed as an XML document.

XML SPECIFICATION:

- ☑ **XML:**
  Defines the syntax of XML.
- ☑ **XLL (Extensible, Linking Language):**
  Defines a standard way to represent links between resources.

- ☑ **XSL (Extensible Style Language):**
  Will define a standard style sheet language for XML.
- ☑ **XUA (XML User Agent):**
  Will help standardize XML User Agents (like browser).

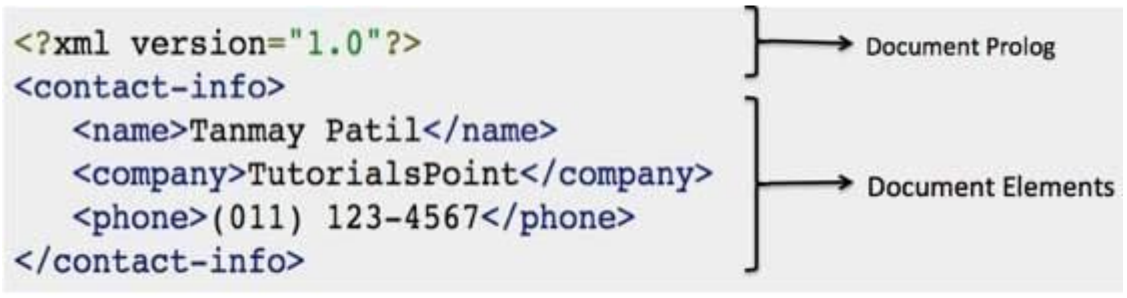
DOCUMENTS:

An XML *document* is a basic unit of XML information composed of elements and other markup in an orderly package. An XML *document* can contains wide variety of data. For example, database of numbers, numbers representing molecular structure or a mathematical equation.

Example:

```
<?xml version="1.0"?>
<contact-info>
  <name>Tanmay Patil</name>
  <company>TutorialsPoint</company>
  <phone>(011) 123-4567</phone>
</contact-info>
```

The following image depicts the parts of XML document.



```
<?xml version="1.0"?>
<contact-info>
   <name>Tanmay Patil</name>
   <company>TutorialsPoint</company>
   <phone>(011) 123-4567</phone>
</contact-info>
```

Document Prolog Section:

The **document prolog** comes at the top of the document, before the root element. This section contains:

- ➢ XML declaration
- ➢ Document type declaration

Document Elements Section:

Document Elements are the building blocks of XML. These divide the document into a hierarchy of sections, each serving a specific purpose. You can separate a document into multiple sections so that they can be rendered differently, or used by a search engine. The elements can be containers, with a combination of text and other elements.

STRUCTURE OF XML:

According to the specification, XML documents have both logical and physical structure. A document is built up from storage units called entities. They can contain parsed or unparsed data. Parsed entities contain characters that formed either character data or markups. Markups are used to encode the logical and physical structure of the document. Both structures are subject of limitations, well-formalness and validity.

☑ Logical Structure:

The document logical structure consists of declarations, elements, comments, processing instructions and character references. The UML diagram in figure below illustrates this:

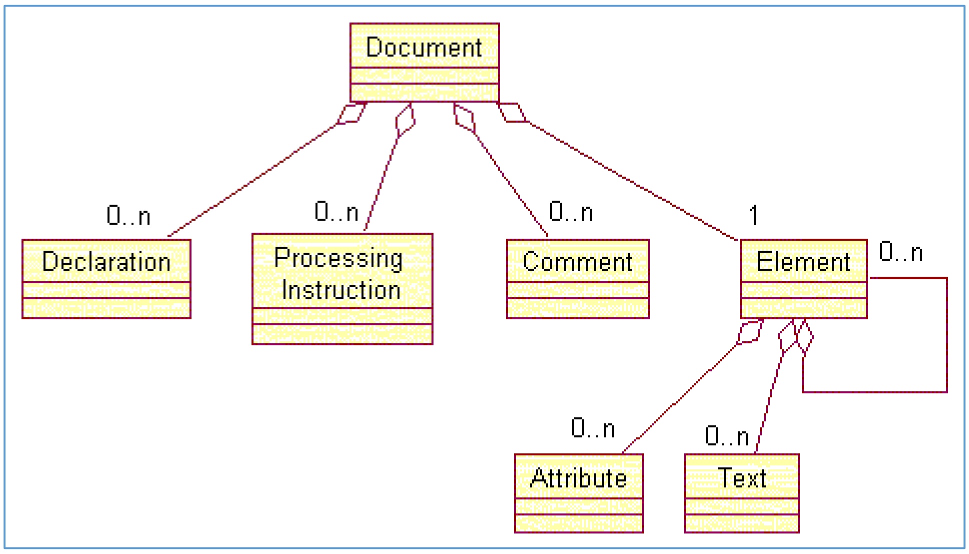


Fig: XML Document Logical Structure

Every well-formed document contains one or more elements that form a tree hierarchy. Consequently, there is exactly one element at the highest level of the hierarchy that serves as a root for the tree.

The allowed locations of processing instructions, comments and declarations regarding to the element hierarchy are given by the grammar rules in the specification.

The document logical structure can vary in the level of details and structure. This depends on the needs of the concrete application. For instance text nodes can be further decomposed into characters. Also, declarations can be represented either as a monolithic item or as a list of single declarations.

At the moment the specifications in W3C uses several logical models. The most important are:

- **DOM (Document Object Model)**, which provides logical model for XML and HTML documents and a set of interfaces to it. DOM has 3 variations: leve1, 2 and 3. They provide different level of details on the document content.
- **XML Information Set** defines abstract data model of the document information content. Its purpose is to serve as a common set of terms used by the other specifications;
- **Models** used by XPath and XPointer specifications. These models treat an XML document as a tree of nodes. There are mappings from them to XML Information Set.

XML processor takes the responsibility to provide interface to the logical structure. Furthermore, applications can extend it and provide new features. However, before doing that certain recognition of document physical structure is required followed by reading and processing of declared entities. This is a process that depends on the parser and therefore the resulted logical structure exposed to the application can vary.

☑ Physical Structure:

This section describes the entity concept and presents three classifications of entities as they are defined in the specification. For each entity type examples show the syntax for entity declaration and usage. Finally, the section gives some remarks about the process of entity expansion.

Entities:

Every XML document is composed of storage units called entities. An entity has a name and content. The name is used to form a reference to the entity. There are two exceptions of entities without names, the document entity and the part of the DTD that is not contained in the document (so called external subset).

An entity can contain references to other entities. There is a special entity called document entity or root that serves as a main storage unit. XML processors always start document processing from that unit, which can contain the whole document.

Entities can take different forms. They can form a separate text file with XML data identified by an URI and obtained and processed by the parser. Processing usually results in inclusion of the text in the place of reference. Or they can be files that contain any kind of resources including non-textual objects. In this case the entity content is not retrieved and processed by the parser. The application is informed about the resource and can take some actions. Also, entities can be defined as named strings inside another entity and referred to from several places.

Entity Types:

We have three classifications of entities:

1. ***<u>Parsed And Unparsed Entities:</u>***

According to the specification parsed entities contain text that is intended to be processed by the parser and is considered as an integral part of the document. Unparsed entities are resources that can be of any type including text objects.

The main difference between parsed and unparsed entities is in the treatment taken by the parser. The XML parser never processes unparsed entities. Instead, their presence is reported to the application.

The following are two declarations of parsed entities:

```
<!ENTITY full_name “XML Technology in E_Commerce” >  ------------ (1)
<!ENTITY short_name “XTEC” > -------------------- (2)
```

In this examples the strings **full_name** and **short_name** are entity names and the text in quotations is the entity value. Here two entities take the form of named strings.

These entities can be used in the document by using references to them:

```
<courses> ------------------------- (3)
    <course>
```

```
            <title>&full_name;</title>
            <abrev>&short_name;</abrev>
        </course>
    </courses>
```

The entity reference is made up by the entity name delimited with & and ; characters. In the example above if the name of the course is used several times across the document, it can be declared only once as an entity value and a reference to it can be used multiple times. The value will be included by replacing the reference. This process is performed by the parser and is called expansion of the entity reference. If a change of the course name is required, it is localized at a single place, the entity value. This mechanism makes the document content more manageable and reduces the potential for errors.

In the example above the entities take the form of a string declared in some storage unit, usually a text file. There is no separate storage unit for them. It is possible to declare an entity whose value is contained outside of the storage object that contains the declaration, e.g. in a separate file. This type of entity is called external. In (4) and (5) an example of external entity declaration is shown:

<!ENTITY course_objectives SYSTEM “http://trese.cs.utwente.nl/courses/xtec/objectives.xml” > --------------------- (4)

This declaration uses system identifier, which is an URI after the SYSTEM keyword that can be used to retrieve the entity resource. There is another variant based on a public identifier:

<!ENTITY course_objectives PUBLIC “-//Twente University//XTEC Objectives//EN” “http://trese.cs.utwente.nl/courses/xtec/objectives.xml” > --------------------- (5)

The mechanism of public identifiers is inherited from SGML. It defines a set of names within an organization but the names may be invisible outside.

The exact syntax rules for these two forms are in the XML specification.

Entity reference syntax to such kind of entities is the same, but the treatment by the XML parser is different. There can be an expansion of the reference, but this depends on the type of the parser. Detailed discussion about parser types (validating and non-validating) is contained in the specification. Generally, non-validating parsers are not required to process external parsed entities.

The usage of external entities allows for modularization of XML documents and independent authoring by multiple authors.

Unparsed entities are declared in the following way:

<!ENTITY logo SYSTEM “http://trese.cs.utwente.nl/courses/xtec/logo.gif” NDATA gif> ------ (6)

The form with public identifier is also possible.

This example entity declares an image resource in GIF format. The entity is recognized as unparsed by the NDATA keyword followed by a notation name, in this case “gif”. XML processor must inform the application about the entity identifiers and notation name. Notation carries out information about the format of an unparsed entity. Notations are declared like that:

<!NOTATION gif SYSTEM "GIF" > -------------------- (7)

According to the validity constraints notation names must be declared.

XML specification doesn't specify notation semantics. It is assumed that the application will handle the entity on the base of the notation name, but it also can take another specific handling.

Unparsed entities are only used by name. It can be the value of attributes with type ENTITY or ENTITIES.

2. ***General And Parameter Entities:***

The distinction between these types lies in the scope of their usage. Parameter entities are always parsed and used only in the DTD part of the XML document, whereas the general entities are used in the document content. The entities of these types are declared and used differently.

Assume that in the DTD we have several elements that share a common set of attributes. XML specification does not allow single attribute definition that can be referred to multiple times in the context of different elements. Attribute definition is always attached to an element declaration. Consequently, if several elements share a common attribute the attribute definition will be repeated for each element. The mechanism of parameter entities provides a work around for the problem. Instead of repeating the definition of attributes we can declare a parameter entity like this:

```
<!ENTITY % common_attr  "id ID --------------------- (8)
#REQUIRED meta CDATA
#IMPLIED time CDATA
#IMPLIED">
```

and use it for more than one element:

```
<!ELEMENT el1 ……..> ----------------------------- (9)
<!ATTLIST    el1 %common_attr;>
<!ELEMENT el2 ……..>
<!ATTLIST    el2 %common_attr;>
…………………………………..
…………………………………..
```

Parameter entity declaration includes % character before the name and an entity reference uses % and ; as delimiters. Since the context of both types is different, we can have two entities with the same name, but one as general and one as parameter:

```
<!ENTITY % common_attr  "id ID ---------------- (10)
#REQUIRED meta CDATA
#IMPLIED time CDATA
#IMPLIED">
<!ENTITY common_attr "Example of general and parameter entities with the same names">
```

3. ***Internal and External Entities:***

For internal entities there is no separate physical storage object. In the above examples internal entities are (1), (2), (8) and (10). Internal entities are always parsed.

If an entity is not internal it is an external entity like (4), (5) and (6). It can be noticed that the presence of SYSTEM denotes an entity as external. NDATA keyword marks an external entity as unparsed.

ELEMENTS:

XML elements can be defined as building blocks of an XML. Elements can behave as containers to hold text, elements, attributes, media objects or all of these.

Each XML document contains one or more elements, the scope of which are either delimited by start and end tags, or for empty elements, by an empty-element tag.

Syntax

```
<element-name attribute1 attribute2>
     ....content
</element-name>
```

Where

- ☑ element-name is the name of the element. The name its case in the start and end tags must match.
- ☑ attribute1, attribute2 are attributes of the element separated by white spaces. An attribute defines a property of the element. It associates a name with a value, which is a string of characters. An attribute is written as:
  name = "value"
  name is followed by an = sign and a string value inside double(" ") or single(' ') quotes.

EMPTY ELEMENT:

An empty element (element with no content) has following syntax:

```
<name attribute1 attribute2.../>
```

Example:

```
<?xml version="1.0"?>
<contact-info>
  <address category="residence">
    <name>Tanmay Patil</name>
    <company>TutorialsPoint</company>
    <phone>(011) 123-4567</phone>
  <address/>
</contact-info>
```

NAMING RULES:

XML elements must follow these naming rules:

- Element names are case-sensitive i.e. Address is not same as address.
- Element names must start with a letter (alphabets) or underscore (_)
- Element names cannot start with the letters xml (or XML, or Xml, etc)

- Element names can contain letters, digits, hyphens, underscores, and periods (.)
- Element names cannot contain spaces.

Any name can be used, no words are reserved (except xml).

Best Naming Practices:

- Create descriptive names, like this: <person>, <firstname>, <lastname>.
- Create short and simple names, like this: <book_title> not like this: <the_title_of_the_book>.
- Avoid "-". If you name something "first-name", some software may think you want to subtract "name" from "first".
- Avoid ".". If you name something "first.name", some software may think that "name" is a property of the object "first".
- Avoid ":". Colons are reserved for namespaces (more later).

Non-English letters like éòá are perfectly legal in XML, but watch out for problems if your software doesn't support them.

Naming Styles:

There are no naming styles defined for XML elements. But here are some commonly used:

Style	Example	Description
Lower case	<firstname>	All letters lower case
Upper case	<FIRSTNAME>	All letters upper case
Underscore	<first_name>	Underscore separates words
Pascal case	<FirstName>	Uppercase first letter in each word
Camel case	<firstName>	Uppercase first letter in each word except the first

If we choose a naming style, it is good to be consistent! XML documents often have a corresponding database. A common practice is to use the naming rules of the database for the XML elements.

Element Content Models:

To declare the syntax of an element in a DTD, we use the <!ELEMENT> element like this: <!ELEMENT*name content_model*>. In this syntax, *name* is the name of the element we're declaring and *content_model* is the content model of the element. A *content model* indicates what content the element is allowed to have, for example, we can allow child elements or text data, or we can make the element empty by using the EMPTY keyword, or we can allow any content by using the ANY keyword.

```
<!DOCTYPE document [
<!ELEMENT document (employee)*>
   .
   .
   .
]>
```

This <!ELEMENT> element not only declares the <document> element, but it also says that the <document> element may contain <employee> elements. When we declare an element in this way, we also specify what contents that element can legally contain; the syntax for doing that is a little involved. The following sections dissect that syntax, taking a look at how to specify the content model of elements, starting with the least restrictive content model of all—ANY, which allows any content at all.

1. **Handling Any Content:**

If we give an element the content model ANY, that element can contain any content, which means any elements and/or any character data. What this really means is that we're turning off validation for this element because the contents of elements with the content model ANY are not even checked. Here's how to specify the content model ANY for an element named <document>:

```
<!DOCTYPE document [
<!ELEMENT document ANY>
  .
  .
  .
]>
```

As far as the XML validator is concerned, this just turns off validation for the <document> element. It's usually not a good idea to turn off validation, but we might want to turn off validation for specific elements, for example, if we want to debug a DTD that's not working. It's usually far preferable to actually list the contents we want to allow in an element, such as any possible child elements the element can contain.

2. **Specifying Child Elements:**

We can specify what child elements an element can contain in that element's content model. For example, we can specify that an element can contain another element by explicitly listing the name of the contained element in parentheses, like this:

```
<!DOCTYPE document [
<!ELEMENT document (employee)*>
  .
  .
  .
]>
```

This specifies that a <document> element can contain <employee> elements. The * here means that a <document> element can contain any number (including zero) <employee> elements. (We'll talk about what other possibilities besides * are available in a few pages.) With this line in a DTD, we can now start placing an <employee> element or elements inside a <document> element, this way:

```
<?xml version = "1.0" standalone="yes"?>
<!DOCTYPE document [
<!ELEMENT document (employee)*>
]>
<document>
 <employee>
  .
  .
  .
 </employee>
</document>
```

Note, however, that this is no longer a valid XML document because we haven't specified the syntax for individual <employee> elements. Because <employee> elements can contain <name>, <hiredate>, and <projects> elements, *in that order*, we can specify a content model for<employee> elements this way:

```
<?xml version = "1.0" standalone="yes"?>
<!DOCTYPE document [
<!ELEMENT document (employee)*>
<!ELEMENT employee (name, hiredate, projects)>
<!ELEMENT name (lastname, firstname)>
<document>
 <employee>
  <name>
   <lastname>Kelly</lastname>
   <firstname>Grace</firstname>
  </name>
  <hiredate>October 15, 2005</hiredate>
  <projects>
   <project>
    <product>Printer</product>
    <id>111</id>
    <price>$111.00</price>
   </project>
   <project>
    <product>Laptop</product>
    <id>222</id>
    <price>$989.00</price>
   </project>
  </projects>
 </employee>
</document>
```

Listing multiple elements in a content model this way is called creating a *sequence*. We use commas to separate the elements we want to have appear, and then the elements have to appear in that sequence in our XML document. For example, if we declare this sequence in the DTD:

```
<!ELEMENT employee (name, hiredate, projects)>
```

then inside an <employee> element, the <name> element must come first, followed by the <hiredate>element, followed by the <projects> element, like this:

```
<employee>
  <name>
    <lastname>Kelly</lastname>
    <firstname>Grace</firstname>
  </name>
  <hiredate>October 15, 2005</hiredate>
  <projects>
    <project>
      <product>Printer</product>
      <id>111</id>
      <price>$111.00</price>
    </project>
    <project>
      <product>Laptop</product>
      <id>222</id>
      <price>$989.00</price>
    </project>
  </projects>
</employee>
```

This example introduces a whole new set of elements <name>, <hiredate>, <lastname>, and so on that don't contain other elements at all—they contain text. So how can we specify that an element contains text? Read on.

3. Handling Text Content:

In the preceding section's example, the <name>, <hiredate>, and <lastname> elements contain text data. In DTDs, non-markup text is considered parsed character data (in other words, text that has already been parsed, which means the XML processor shouldn't touch that text because it doesn't contain markup). In a DTD, we refer to parsed character data as #PCDATA. Note that this is the only way to refer to text data in a DTD we can't say anything about the actual format of the text, although that might be important if we're dealing with numbers. In fact, this lack of precision is one of the reasons that XML schemas were introduced.

Here's how to give the text-containing elements in the PCDATA content model example:

```
<?xml version = "1.0" standalone="yes"?>
<!DOCTYPE document [
<!ELEMENT document (employee)*>
<!ELEMENT employee (name, hiredate, projects)>
<!ELEMENT name (lastname, firstname)>
<!ELEMENT lastname (#PCDATA)>
<!ELEMENT firstname (#PCDATA)>
<!ELEMENT hiredate (#PCDATA)>
<!ELEMENT projects (project)*>
<!ELEMENT project (product,id,price)>
<!ELEMENT product (#PCDATA)>
<!ELEMENT id (#PCDATA)>
<!ELEMENT price (#PCDATA)>
]>
<document>
 <employee>
  <name>
   <lastname>Kelly</lastname>
   <firstname>Grace</firstname>
  </name>
  <hiredate>October 15, 2005</hiredate>
  <projects>
   <project>
    <product>Printer</product>
    <id>111</id>
    <price>$111.00</price>
   </project>
   <project>
    <product>Laptop</product>
    <id>222</id>
    <price>$989.00</price>
   </project>
  </projects>
 </employee>
</document>
```

4. Specifying Multiple Child Elements:

There are a number of options for declaring an element that can contain child elements. We can declare the element to contain a single child element:

```
<!ELEMENT document (employee)>
```

We can declare the element to contain a list of child elements, in order:

```
<!ELEMENT document (employee, contractor, partner)>
```

We can also use symbols with special meanings in DTDs, such as *, which means "zero or more of," as in this example, where we're allowing zero or more <employee> elements in a <document> element:

```
<!ELEMENT document (employee)*>
```

There are a number of other ways of specifying multiple children by using symbols. (This syntax is actually borrowed from regular expression handling in the Perl language, so if we know that language, we have a leg up here.) Here are the possibilities:

- **x+**—Means x can appear one or more times.
- **x***—Means x can appear zero or more times.
- **x?**—Means x can appear once or not at all.
- **x, y**—Means x followed by y.
- **x | y**—Means x *or* y—but not both.

The following sections take a look at these options.

5. **Allowing One or More Children:**

We might want to specify that a <document> element can contain between 200 and 250 <employee>elements, and if we do, we're out of luck with DTDs because DTD syntax doesn't give us that kind of precision. On the other hand, we still do have some control here; for example, we can specify that a <document> element must contain one or more <employee> elements if we use a + symbol, like this:

```
<!ELEMENT document (employee)+>
```

Here, the XML processor is being told that a <document> element has to contain at least one <employee> element.

6. **Allowing Zero or More Children:**

By using a DTD, we can use the * symbol to specify that we want an element to contain any number of child elements that is, zero or more child elements. We saw this in action earlier, when we specified that the <document> element may contain <employee> elements in the above example:

```
<!ELEMENT document (employee)*>
```

7. **Allowing Zero or One Child:**

When using a DTD, we can use ? to specify zero or one child elements. Using ? indicates that a particular child element *may* be present once in the element we're declaring, but it need not be. For example, here's how to indicate that a <document> element may contain zero or one <employee> elements:

```
<!ELEMENT document (employee)?>
```

ELEMENT OCCURRENCE INDICATORS:

There are seven indicators:

- ☑ **ORDER INDICATORS:**

Order indicators are used to define the order of the elements.

1. **All Indicator:**

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

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

Note: When using the <all> indicator we can set the <minOccurs> indicator to 0 or 1 and the <maxOccurs> indicator can only be set to 1 (the <minOccurs> and <maxOccurs> are described later).

2. **Choice Indicator:**

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

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

3. **Sequence Indicator:**

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

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

☑ Occurrence Indicators:

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

Note: For all "Order" and "Group" indicators (any, all, choice, sequence, group name, and group reference) the default value for maxOccurs and minOccurs is 1.

1. **maxOccurs Indicator:**

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

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

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

2. **minOccurs Indicator:**

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

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

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

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

A working example:

An XML file called "Myfamily.xml":

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

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

<person>
<full_name>Stale Refsnes</full_name>
</person>
</persons>
```

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

Here is the schema file "family.xsd":

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs=http://www.w3.org/2001/XMLSchema elementFormDefault="qualified">
<xs:element name="persons">
<xs:complexType>
<xs:sequence>
<xs:element name="person" maxOccurs="unbounded">
<xs:complexType>
<xs:sequence>
<xs:element name="full_name" type="xs:string"/>
<xs:element name="child_name" type="xs:string" minOccurs="0" maxOccurs="5"/>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:schema>
```

☑ Group Indicators:

Group indicators are used to define related sets of elements.

1. Element Groups:

Element groups are defined with the group declaration, like this:

```
<xs:group name="groupname">
...
```

```
</xs:group>
```

We must define an all, choice, or sequence element inside the group declaration. The following example defines a group named "persongroup" that defines a group of elements that must occur in an exact sequence:

```
<xs:group name="persongroup">
<xs:sequence>
<xs:element name="firstname" type="xs:string"/>
<xs:element name="lastname" type="xs:string"/>
<xs:element name="birthday" type="xs:date"/>
</xs:sequence>
</xs:group>
```

After we have defined a group, we can reference it in another definition, like this:

```
<xs:group name="persongroup">
<xs:sequence>
<xs:element name="firstname" type="xs:string"/>
<xs:element name="lastname" type="xs:string"/>
<xs:element name="birthday" type="xs:date"/>
</xs:sequence>
</xs:group>
<xs:element name="person" type="personinfo"/>
<xs:complexType name="personinfo">
<xs:sequence>
<xs:group ref="persongroup"/>
<xs:element name="country" type="xs:string"/>
</xs:sequence>
</xs:complexType>
```

2. Attribute Groups:

Attribute groups are defined with the attributeGroup declaration, like this:

```
<xs:attributeGroup name="groupname">
...
</xs:attributeGroup>
```

The following example defines an attribute group named "personattrgroup":

```
<xs:attributeGroup name="personattrgroup">
<xs:attribute name="firstname" type="xs:string"/>
<xs:attribute name="lastname" type="xs:string"/>
<xs:attribute name="birthday" type="xs:date"/>
</xs:attributeGroup>
```

After we have defined an attribute group, we can reference it in another definition, like this:

```
<xs:attributeGroup name="personattrgroup">
<xs:attribute name="firstname" type="xs:string"/>
<xs:attribute name="lastname" type="xs:string"/>
<xs:attribute name="birthday" type="xs:date"/>
</xs:attributeGroup>

<xs:element name="person">
<xs:complexType>
<xs:attributeGroup ref="personattrgroup"/>
</xs:complexType>
</xs:element>
```

CDATA:

The term CDATA means, Character Data. CDATA are defined as blocks of text that are not parsed by the parser, but are otherwise recognized as markup.

The predefined entities such as &lt;, &gt;, and &amp; require typing and are generally difficult to read in the markup. In such cases, CDATA section can be used. By using CDATA section, you are commanding the parser that the particular section of the document contains no markup and should be treated as regular text.

Syntax

```
<![CDATA[
  characters with markup
]]>
```

The above syntax is composed of three sections:

- **CDATA Start section -** CDATA begins with the nine-character delimiter **<![CDATA[**
- **CDATA End section -** CDATA section ends with]**]>** delimiter.
- **CData section -** Characters between these two enclosures are interpreted as characters, and not as markup. This section may contain markup characters (<, >, and &), but they are ignored by the XML processor.

Example

The following markup code shows example of CDATA. Here, each character written inside the CDATA section is ignored by the parser.

```
<script>
<![CDATA[
  <message> Welcome to TutorialsPoint </message>
]] >
</script >
```

In the above syntax, everything between <message> and </message> is treated as character data and not as markup.

CDATA Rules:

The given rules are required to be followed for XML CDATA:

- CDATA cannot contain the string "]]>" anywhere in the XML document.
- Nesting is not allowed in CDATA section.

DTD (DOCUMENT TYPE DECLARATION):

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

```
<!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 external source.

Syntax

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

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

Example

```
<?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 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 we 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:

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

Where *file-name* is the file with *.dtd* extension.

Example:

```
<?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** are as shown:

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

Types:

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

1. **System Identifiers:**

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

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

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

2. **Public Identifiers:**

Public identifiers provide a mechanism to locate DTD resources and are written as below:

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

As we 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.*

Limitation of DTD:

- DTDs do not have built-in datatypes.
- DTDs do not support user-derived datatypes.
- DTDs allow only limited control over cardinality (the number of occurrences of an element within its parent).
- DTDs do not support Namespaces or any simple way of reusing or importing other schemas.

DOCUMENT TYPE DEFINITION (DTD):

A specification for a SGML or HTML document that specifies structural elements and markup definitions that can be used to create documents that describe content. The DTD can put constraints on the occurrence and content of elements and other details of the document structure

SCHEMA:

XML Schema is commonly known as XML Schema Definition (XSD). It is used to describe and validate the structure and the content of XML data. XML schema defines the elements, attributes and data types. Schema element supports Namespaces. It is similar to a database schema that describes the data in a database.

Syntax:

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

Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="contact">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="name" type="xs:string" />
      <xs:element name="company" type="xs:string" />
      <xs:element name="phone" type="xs:int" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:schema>
```

The basic idea behind XML Schemas is that they describe the legitimate format that an XML document can take.

Definition Types:

We can define XML schema elements in following ways:

1. **Simple Type:**

Simple type element is used only in the context of the text. Some of predefined simple types are: xs:integer, xs:boolean, xs:string, xs:date. For example:

```
<xs:element name="phone_number" type="xs:int" />
```

2. **Complex Type:**

A complex type is a container for other element definitions. This allows us to specify which child elements an element can contain and to provide some structure within your XML documents. For example:

```
<xs:element name="Address">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="name" type="xs:string" />
          <xs:element name="company" type="xs:string" />
      <xs:element name="phone" type="xs:int" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

In the above example, *Address* element consists of child elements. This is a container for other <xs:element> definitions, that allows to build a simple hierarchy of elements in the XML document.

3. **Global Types:**

With global type, we can define a single type in our document, which can be used by all other references. For example, suppose we want to generalize the *person* and *company* for different addresses of the company. In such case, we can define a general type as below:

```
<xs:element name="AddressType">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="name" type="xs:string" />
```

```
                <xs:element name="company" type="xs:string" />
            </xs:sequence>
        </xs:complexType>
    </xs:element>
```

Now let us use this type in our example as below:

```
    <xs:element name="Address1">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="address" type="AddressType" />
                    <xs:element name="phone1" type="xs:int" />
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="Address2">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="address" type="AddressType" />
                    <xs:element name="phone2" type="xs:int" />
            </xs:sequence>
        </xs:complexType>
    </xs:element>
```

Instead of having to define the name and the company twice (once for *Address1* and once for *Address2*), we now have a single definition. This makes maintenance simpler, i.e., if you decide to add "Postcode" elements to the address, you need to add them at just one place.

ATTRIBUTES:

Attributes in XSD provide extra information within an element. Attributes have *name* and *type* property as shown below:

```
    <xs:attribute name="x" type="y"/>
```

XSL (EXTENSIBLE STYLE SHEET LANGUAGE):

XSL is a language for expressing style sheets. An XSL style sheet is, like with CSS, a file that describes how to display an XML document of a given type. XSL shares the functionality and is compatible with CSS2 (although it uses a different syntax). It also adds:

- A transformation language for XML documents: XSLT. Originally intended to perform complex styling operations, like the generation of tables of contents and indexes, it is now used as a general purpose XML processing language. XSLT is thus widely used for purposes other than XSL, like generating HTML web pages from XML data.

- Advanced styling features, expressed by an XML document type which defines a set of elements called Formatting Objects, and attributes (in part borrowed from CSS2 properties and adding more complex ones.

ASSOCIATING CSS STYLE SHEET WITH XML:

- XML has emerged as a "universal" data format in a variety of application areas.
- Style sheets are an essential step in XML deployment to define the presentation of XML documents.
- The association consists of inserting the XML processing instruction at the top of the document, before the root element of the XML document and after the XML prolog.
- The processing instruction has two required attributes type and href which respectively specify the type of stylesheet (Internet Media Type text/css) and its address (path).
  <?xml-stylesheet type="text/css" href="foo.css"?>

Example:

First create a file .xml

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/css" href="style.css"?>
<PU>
    <Centers>
    <Center>
            <Code>001</Code>
            <Location id="Pkr">Pokhara</Location>
            <Phone>344333</Phone>
            <Email>pkr@webcom.com</Email>
    </Center>
    <Center>
            <Code>003</Code>
            <Location id="Ktm">Kathmandu</Location>
            <Phone>3444433</Phone>
            <Email>ktm@webcom.com</Email>
    </Center>
    <Center>
            <Code>004</Code>
            <Location id="Tnu">Tanahu</Location>
            <Phone>342223</Phone>
            <Email>tanahu@webcom.com</Email>
    </Center>
    </Centers>
</PU>
```

Then create style.css file:

```
PU {
        font-family: verdana;
        color:brown;
      }
    Center {
        background:yellow;
        display:block;
        margin:5px;
      }
    Location {
        font-size:large;
        display:block;
      }
    Email {
        font-size:small;
        display:block;
      }
```

XML PROCESSORS:

1. DOM:

The Document Object Model (DOM) is an application programming interface (API) for HTML and XML documents. It defines the logical structure of documents and the way a document is accessed and manipulated.

XML DOM is a standard object model for XML. XML documents have a hierarchy of informational units called nodes; DOM is a standard programming interface of describing those nodes and the relationships between them.

As XML DOM also provides an API that allows a developer to add, edit, move or remove nodes at any point on the tree in order to create an application.

Example:

The following example (sample.htm) parses an XML document ("address.xml") into an XML DOM object and then extracts some information from it with JavaScript:

```
<!DOCTYPE html>
<html>
  <body>
    <h1>TutorialsPoint DOM example </h1>
    <div>
      <b>Name:</b> <span id="name"></span><br>
      <b>Company:</b> <span id="company"></span><br>
      <b>Phone:</b> <span id="phone"></span>
    </div>
    <script>
```

```
      if (window.XMLHttpRequest)
      {// code for IE7+, Firefox, Chrome, Opera, Safari
        xmlhttp = new XMLHttpRequest();
      }
      else
      {// code for IE6, IE5
        xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
      }
      xmlhttp.open("GET","/xml/address.xml",false);
      xmlhttp.send();
      xmlDoc=xmlhttp.responseXML;

      document.getElementById("name").innerHTML=
      xmlDoc.getElementsByTagName("name")[0].childNodes[0].nodeValue;
      document.getElementById("company").innerHTML=
      xmlDoc.getElementsByTagName("company")[0].childNodes[0].nodeValue;
      document.getElementById("phone").innerHTML=
      xmlDoc.getElementsByTagName("phone")[0].childNodes[0].nodeValue;
    </script>
  </body>
</html>
```

Contents of **address.xml** are as below:

```
<?xml version="1.0"?>
<contact-info>
  <name>Tanmay Patil</name>
  <company>TutorialsPoint</company>
  <phone>(011) 123-4567</phone>
</contact-info>
```

Now let us keep these two files **sample.htm** and **address.xml** in the same directory **/xml** and execute the **sample.htm** file by opening it in any browser.

<u>Advantages</u>

- XML DOM is language and platform independent.
- XML DOM is travesible - Information in XML DOM is organized in a hierarchy which allows developer to navigate around the hierarchy looking for specific information.
- XML DOM is modifiable - It is dynamic in nature providing developer a scope to add, edit, move or remove nodes at any point on the tree.

<u>Disadvantages</u>

- It consumes more memory (if the XML structure is large) as program written once remains in memory all the time until and unless removed explicitly.
- Due to the larger usage of memory its operational speed, compared to SAX is slower.

2. SAX (Simple API for XML):

SAX (Simple API for XML) is an event-driven online algorithm for parsing XML documents, with an API developed by the XML-DEV mailing list. SAX provides a mechanism for reading data from an XML document that is an alternative to that provided by the Document Object Model (DOM). Where the DOM operates on the document as a whole, SAX parsers operate on each piece of the XML document sequentially.

A SAX parser only needs to report each parsing event as it happens, and normally discards almost all of that information once reported (it does, however, keep some things, for example a list of all elements that have not been closed yet, in order to catch later errors such as end-tags in the wrong order).

Advantages

- It is simple and memory efficient.
- It is very fast and works for huge documents.

Disadvantages

- It is event-based so its API is less intuitive.
- Clients never know the full information because the data is broken into pieces.