Unit 9: XML and Java

XML technology has become a buzz word everywhere in the IT community. Ever since its inception, XML technology has made leaps and bounds and completely changed the way enterprise computing is done. Its adoption by the software industry is such that you'll hardly find any software application that doesn't use XML.

What is XML?

XML stands for "eXtensible Markup Language". The notion of XML is based on something called XML Document. XML is a simple text-based language which was designed to store and transport data in plain text format.

What is an XML Document?

- We all know that there are several ways for storing data and information. For instance, we use a text file to store the data line by line, we use database where tables are used to store data etc.
- XML document is just another way for storing data and information but uses a tree like structure.
- An XML document/file comprises of several tags that represent the data or information. These tags are also referred to as nodes. Following is how a XML tag looks like:

  ```
  <lastName>John</lastName>
  ```

  In the above tag, **lastName** is the name of the tag. Every tag has a beginning and an end. The beginning of the tag is denoted by the tag name in between '<' and '>' and the end of the tag is denoted with the tag name in between '</' and '>' symbols. All the characters or text in between represent the tag data

- The above tag is the simplest and smallest tag you can see in an XML document. However, we can also have complex tags which are nothing but tags within tags as shown below:

  ```
  <name>

  <firstName>John</firstName>

  <lastName>Smith</lastName>

  </name>
  ```

  Here, name is a complex tag that contains two simple tags namely **firstName** and **lastName**. The good thing with XML is that, the tag names can be anything. However, there is just one rule that we need to strictly follow. Rule: **Every tag that is opened must be closed.**

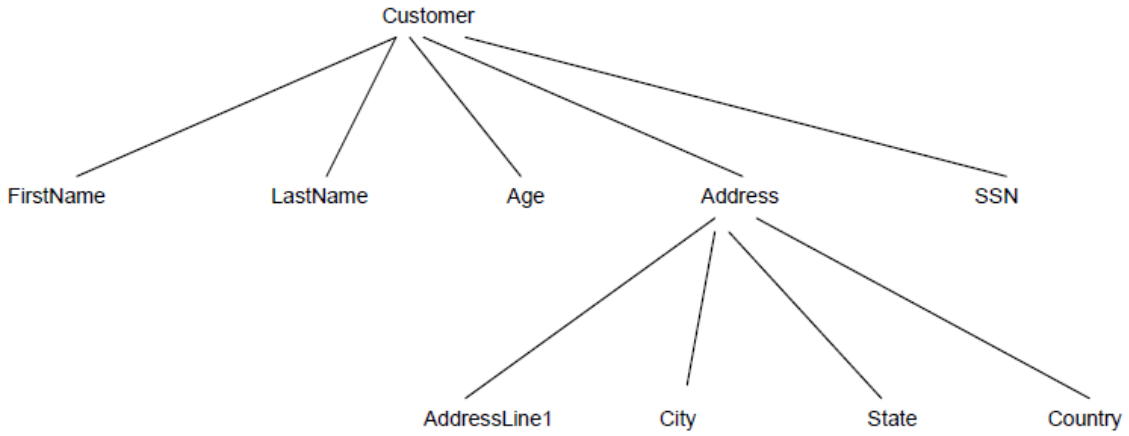
- If the above rule is followed, then we say that the XML document as **well formed**. Following shows a well formed XML document:

General rules:

- XML documents must have a root element
- XML elements must have a closing tag
- XML tags are case sensitive
- XML elements must be properly nested
- XML attribute values must be quoted

```
<?xml version="1.0"?>
        <customer>
            <firstName>John</firstName>
            <lastName>Smith</lastName>
            <age>20</age>
            <ssn>23324</ssn>
            <address>
                <addressline1>Apt 2222</addressline1>
                <city>Columbus</city>
                <state>OH</state>
                <country>USA</country>
            </address>
        </customer>
```

The above XML represents customer information. If you notice, every tag has a closing tag which is why we say that it is well formed. Don't you see a tree like representation here? I am sure you did. In this document, customer is the root node. This has five child nodes namely **firstName**,**lastName**,**age,ssn** and **address**. The **address** child node in turn has 4 child nodes. There is no limit on the number of child nodes a node can have. Following figure shows the tree representation of the above xml document:



- In above xml document,

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

  We call this s as the prolog of the XML document. It represents the version of the XML we are using.

- In XML, a node can also have attributes as shown below:
  `<customer email = "john.smith@abc.com">`

  In the above node, email is the attribute of tag whose value is john.smith@abc.com.

- A tag can have any number of attributes as shown below:

```
<book author="john" isbn="HG76876" pages="200">
    <publisher>Wrox</publisher>
</book>
```

- An XML document is saved with the extension ".xml". One simple way of verifying whether an XML document is well-formed or not is by opening the xml file using any browser like Google Chrome. If the document is well-formed, you'll see the complete XML in the browser.

  - If we fail to close some tag, the browser will display an error as shown below indicating that the XML document is not well formed. The error will also list the element name where the violation occurred.

- XML is a markup language.
- XML was released in late 90's. It was created to provide an easy to use and store self-describing data. XML became a W3C Recommendation on February 10, 1998
- XML is a tag based language like HTML.
- Unlike HTML, XML tags are not predefined.
- We can define our own tags which is why it is called extensible language.
- XML tags are designed to be self-descriptive.
- XML is designed to store and transport data.
- XML is W3C Recommendation for data storage and data transfer.
- XML is not a replacement for HTML.
- XML is designed to carry data, not to display data.
- XML is platform independent and language independent.
- XML was designed to be both human- and machine-readable.

XML declaration

XML declaration contains details that prepare an XML processor to parse the XML document. It is optional, but when used, it must appear in the first line of the XML document.

Syntax

Following syntax shows XML declaration −

```
<?xml
   version = "version_number"
   encoding = "encoding_declaration"
   standalone = "standalone_status"
?>
```

Each parameter consists of a parameter name, an equals sign (=), and parameter value inside a quote. Following table shows the above syntax in detail −

Parameter	Parameter_value	Parameter_description
Version	1.0	Specifies the version of the XML standard used.
Encoding	UTF-8, UTF-16, ISO-10646-UCS-2, ISO-10646-UCS-4, ISO-8859-1 to ISO-8859-9, ISO-2022-JP, Shift_JIS, EUC-JP	It defines the character encoding used in the document. UTF-8 is the default encoding used.
Standalone	yes or no	It informs the parser whether the document relies on the information from an external source, such as external document type definition (DTD), for its content. The default value is set to no. Setting it to yes tells the processor there are no external declarations required for parsing the document.

Why XML is important?

The main reason why XML has tasted unprecedented success is because it is 100% **language independent and platform independent.**

It can be used to represent any complex data or information without any limitations on size with user defined tags. The only rule is that the document must be well formed. This is the reason why XML though very simple, is yet so powerful.

Since enterprise applications involve complex data representation and processing, the flexibility that XML offers make it an ideal candidate for using in such situations. J2EE extensively uses XML in various ways for simplifying things.

Features and Advantages of XML

XML is widely used in the era of web development. It is also used to simplify data storage and data sharing.

The main features or advantages of XML are given below.

- **XML separates data from HTML**

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

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

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

- **XML simplifies data sharing**

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

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

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

- **XML simplifies data transport**

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

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

- **XML simplifies Platform change**

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

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

- **XML increases data availability**

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

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

- **XML can be used to create new internet languages**

  A lot of new Internet languages are created with XML.

  Here are some examples:

  Extensible Hypertext Markup Language (**XHTML)**

  WSDL for describing available web services

  WAP and **WML** as markup languages for handheld devices

  RSS languages for news feeds

  RDF and **OWL** for describing resources and ontology

  SMIL for describing multimedia for the web

XML Validation

As we know, XML is used to represent data or information. Whenever we have any data, the first thing we need to do before processing the data is to verify whether the data is valid or not, right? So the question is how to validate the data represented by XML document?

- An XML document with correct syntax is called "Well Formed"
- A "well formed" XML document is not the same as a "valid" XML document.
- A "valid" XML document must be well formed. In addition, it must conform to a document type definition.
- There are two different document type definitions that can be used with XML:
  - DTD
  - XML Schema

Document Type Definition (DTD)

- DTD stands for Document Type Definition. It defines the legal building blocks of an XML document. It is used to define document structure with a list of legal elements and attributes.

- Its main purpose is to define the structure of an XML document. It contains a list of legal elements and define the structure with the help of them.
- 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.
- A DTD basically defines the following things:
  - ✓ The order of elements in the XML
  - ✓ The valid list child elements of a particular element
  - ✓ The list of valid attributes for a particular element
- All the validations defined using DTD are stored in a file with the extension ".dtd" and is referenced by the XML document.
- A DTD defines the validation rules for elements in an XML document using the ELEMENT declaration.
- The syntax for the element declarations in a DTD is shown below:
  <!ELEMENT element name content-model>
- The **content-model** basically tells the type of content the element can have. There are four types of content models an element can have as listed below:
  1. **EMPTY:** This indicates the element will not have any content As an example, look at the following DTD for an element, and the element usage in the XML based on the DTD declaration

     DTD

     ```
     <!ELEMENT format EMPTY>
     ```

     XML

     ```
     <format></format> or <format/>
     ```

  2. **ANY:** This indicates that the element can have anything
  3. **Children Only:** This indicates that the element can only have child elements in the specified order. Look at the following DTD and equivalent XML.

     DTD

     ```
     <!ELEMENT account (accountNumber, accountBalance) >
     ```

     XML

     ```
     <account>
     <accountNumber>1234</accountNumber>
     <accountBalance>100.23</accountBalance>
     </account>
     ```

  4. **Text with mixed children:** This indicates that an element can have text as well as specified children in any order. See the following DTD and equivalent XML.

DTD

```
<!ELEMENT description (#PCDATA|b|code)* >
```

where asterisk(*) indicates that the elements in parenthesis can occur 0 or more times within description. Following lists the various symbols.

+ means 1 or more

? means 0 or 1

*** means 0 or more**

Cardinality of A DTD Element

An element's cardinality defines how many times it will appear within a content model. DTDs allow four indicators for cardinality:

```
[none]: only once
?: 0 or 1
+: 1 or more
*: 0 or more
```

XML

```
<description>
     This is a test <b> description </b>
     The child elements can be <code> in </code> in any
     <b>order</b>
</description>
```

Using the above declarations, following is the DTD for the customer data we defined before.

```
<!ELEMENT customer (firstName,lastName,age,ssn,address) >
<!ELEMENT firstName #PCDATA >
<!ELEMENT lastName #PCDATA >
<!ELEMENT age #PCDATA >
<!ELEMENT ssn #PCDATA >
<!ELEMENT address (addressLine1, city, state, country) >
<!ELEMENT addressLine1#PCDATA >
<!ELEMENT city #PCDATA >
<!ELEMENT state #PCDATA >
<!ELEMENT country #PCDATA >
```

The above DTD tells that the **customer** element should have the five child nodes namely **firstName,lastName,age,ssn,address** in the same order. It then defines the content types of every element. All the text elements are defined as PCDATA which stands for Parsed Character DATA. We'll see what parsing is in the next section. The **address** element in turn defines its child elements along with the order of the elements

As we learned before, an XML element can also have attributes. Let's see how we can define the validation rules for attributes using DTD. Following is the syntax for attribute declarations in a DTD

```
<! ATTLIST element-name
(attribute-name attribute-type default-declaration) *
>
```

Consider the following DTD definition for an element named book.

```
<!ELEMENT book (author, publisher) >
<! ATTLIST book
isbn CDATA #REQUIRED
pages CDATA #IMPLIED
>
```

The above declaration tells that book element can have two attributes namely **isbn** which is required and pages attribute which is optional. So, the xml will look as shown below:

XML

```
<book isbn="SD34324" pages="100">
<author>James</author>
<publisher>BPB</publisher>
</book>
```

The attribute-type can be one of the following:

Type	Description
CDATA	The value is character data
(*en1*|*en2*|..)	The value must be one from an enumerated list
ID	The value is a unique id
IDREF	The value is the id of another element
IDREFS	The value is a list of other ids
NMTOKEN	The value is a valid XML name
NMTOKENS	The value is a list of valid XML names
ENTITY	The value is an entity
ENTITIES	The value is a list of entities

NOTATION	The value is a name of a notation
xml:	The value is a predefined xml value

The attribute-value can be one of the following:

Value	Explanation
value	The default value of the attribute
#REQUIRED	The attribute is required
#IMPLIED	The attribute is optional
#FIXED *value*	The attribute value is fixed

Types of DTD

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

1. **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: <!DOCTYPE root-element [element-declarations]>`

2. **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.

   "Private" External DTDs:

   Private external DTDs are identified by the keyword SYSTEM, and are intended for use by a single author or group of authors.

   ```
   <!DOCTYPE root_element SYSTEM "DTD_location">
   ```

   "Public" External DTDs:

   Public external DTDs are identified by the keyword PUBLIC and are intended for broad use. The "DTD_location" is used to find the public DTD if it cannot be located by the "DTD_name".

   ```
   <!DOCTYPE root_element PUBLIC "DTD_name" "DTD_location">
   ```

   where:

   `DTD_location`: relative or absolute URL
   `DTD_name`: follows the syntax:

   ```
   "prefix//owner_of_the_DTD//
   description_of_the_DTD//ISO639_language_identifier"
   ```

   The following prefixes are allowed in the DTD name:

Prefix:	Definition:
ISO	The DTD is an ISO standard. All ISO standards are approved.
+	The DTD is an approved non-ISO standard.
-	The DTD is an unapproved non-ISO standard.

XML document with an 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>Bimal </name>
   <company>OurTech</company>
   <phone>1234567</phone>
</address>
```

How to link .dtd and .xml file

Syntax:

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

Once we have the DTD in a file say **customer.dtd**, we finally need to link it with the XML file using the DOCTYPE element as shown below:

```
<!DOCTYPE customer SYSTEM "customer.dtd">

<?xml version="1.0" ?>

<customer>

…….

</customer>
```

- It's important that you have the DOCTYPE element before the XML prolog. Once the XML is linked with the DTD, our applications can start validating the XML document before processing it.

A simple xml file (note.xml)

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

A DTD file (note.dtd)

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

A Reference to a DTD

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

<!DOCTYPE note SYSTEM
"https://www.w3schools.com/xml/note.dtd">

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

- Though DTD provides a simple way of validating an XML document, **it has certain limitations listed below.**
    - ♦ It can only validate the order of the elements. It cannot validate the list of valid values an element can have.
    - ♦ Less flexible. A DTD cannot be extended using inheritance.
    - ♦ No support for validating numeric and boolean data

To overcome the above limitations, a new validation scheme is created which is called as XML Schema.

XML 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.
- An XML schema is used to define the structure of an XML document. It is like DTD but provides more control on XML structure
- Unlike a DTD, an XML Schema is itself an XML document with elements, attributes etc.
- XML Schemas overcame all the limitations of DTD and had now become the standard for validating XML documents.
- The good thing about XML Schema is that it is closely associated with Object Oriented data models.
- One of the major concerns with DTD is the lack of support of various data types. There is no way that using a DTD we can validate the type of data an element can have. This is where schema comes in real handy.
- It contains several built in primitive data types such as string, integer, float etc for validating the data. XML Schema can also be used to build complex data types using simple data types.
    - **XML Schema Data types:**
      There are two types of data types in XML schema.
        1. **Simple Type** allows us to have text-based elements. It contains less attributes, child elements, and cannot be left empty.
        2. **Complex Type** allows us to hold multiple attributes and elements. It can contain additional sub elements and can be left empty.
- **Simple type** element is used only in the context of the text. Some of the predefined simple types are: xs:integer, xs:boolean, xs:string, xs:date.
- Some important simple data types are listed in the following table.

Data Type	Description
`string`	Used for text data
`boolean`	Used for boolean data (True/False)
`float`	Used for 32 bit decimal numbers
`double`	Used for 64 bit decimal numbers

- Using the above data types, an element named score will be defined as shown below:

```
<xsd:element name="score" type="xsd:int"/>
```

- Following are some of the examples using different data types:

```
<xsd:element name="firstName" type="xsd:string" />

<xsd:element name="expiration" type="xsd:date"/>

<xsd:element name="financialIndicator" type="xsd:boolean/>
```

- All the above defines data types for simple elements. However, using XML schema we can build complex data structures from simple data structures. This is where XML schema exhibits its true power. It allows us build complex data structures by defining the order of elements, valid values for different elements and so on.
- 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 our 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.

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

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

XSD Indicators

There are seven indicators:

Order indicators: *used to define the order of the elements.*

all	*specifies that the child elements can appear in any order, and that each child element must occur only once*
choice	*specifies that either one child element or another can occur*
sequence	*specifies that the child elements must appear in a specific order*

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

maxOccurs: *specifies the maximum number of times an element can occur*

minOccurs: *specifies the minimum number of times an element can occur*

Group indicators: *are used to define related sets of elements.*

Group name

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

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

attributeGroup name:

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

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

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

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

The **note** element is a complex type because it contains other elements. The other elements (to, from, heading, body) are simple types because they do not contain other elements

The important thing to note in the above schema document is the **<xs:schema>** element which defined the namespace for all the elements in the schema. A namespace uses a prefixing strategy that associates element names with a **Uniform Resource Identifier (URI)** where the actual vocabulary for the elements will be defined. A namespace is defined using the attribute **xmlns:xs**. This tells the schema to look for the vocabulary for all the elements prefixed with **xs** at the resource specified by the **URI**. You are free to use which ever prefix you want. Namespaces come very handy especially when combining more than one XML document together resulting in namespace collisions

A Reference to an XML Schema

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

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

Syntax of XSD Explained :

An XML XSD is kept in a separate document and then the document can be linked to an XML document to use it.

The basic syntax of a XSD is as follows −

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

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

   targetNamespace = "http://www.tutorialspoint.com"

   xmlns = "http://www.tutorialspoint.com" elementFormDefault =
"qualified">

   <xs:element name = 'class'>

      <xs:complexType>

         <xs:sequence>

            <xs:element name = 'student' type = 'StudentType'
minOccurs = '0'

               maxOccurs = 'unbounded' />

         </xs:sequence>

      </xs:complexType>
```

```
   </xs:element>
   <xs:complexType name = "StudentType">
      <xs:sequence>
         <xs:element name = "firstname" type = "xs:string"/>
         <xs:element name = "lastname" type = "xs:string"/>
         <xs:element name = "nickname" type = "xs:string"/>
         <xs:element name = "marks" type =
"xs:positiveInteger"/>
      </xs:sequence>
      <xs:attribute name = 'rollno' type =
'xs:positiveInteger'/>
   </xs:complexType>
</xs:schema>
```

<Schema> Element

Schema is the root element of XSD and it is always required.

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

The above fragment specifies that elements and data types used in the schema are defined in http://www.w3.org/2001/XMLSchema namespace and these elements/data types should be prefixed with xs. It is always required.

```
targetNamespace = "http://www.tutorialspoint.com"
```

The above fragment specifies that elements used in this schema are defined in http://www.tutorialspoint.com namespace. It is optional.

```
xmlns = "http://www.tutorialspoint.com"
```

The above fragment specifies that default namespace is http://www.tutorialspoint.com.

```
elementFormDefault = "qualified"
```

The above fragment indicates that any elements declared in this schema must be namespace qualified before using them in any XML Document. It is optional

Referencing Schema

Take a look at the following Referencing Schema –

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

<class xmlns = "http://www.tutorialspoint.com"

   xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"

   xsi:schemaLocation = "http://www.tutorialspoint.com
student.xsd">

   <student rollno = "393">

      <firstname>Dinkar</firstname>

      <lastname>Kad</lastname>

      <nickname>Dinkar</nickname>

      <marks>85</marks>

   </student>

   <student rollno = "493">

      <firstname>Vaneet</firstname>

      <lastname>Gupta</lastname>

      <nickname>Vinni</nickname>

      <marks>95</marks>

   </student>

   <student rollno = "593">

      <firstname>Jasvir</firstname>

      <lastname>Singh</lastname>

      <nickname>Jazz</nickname>

      <marks>90</marks>

   </student>

</class>
```

```
xmlns = "http://www.tutorialspoint.com"
```

The above fragment specifies default namespace declaration. This namespace is used by the schema validator check that all the elements are part of this namespace. It is optional.

```
xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
```

```
xsi:schemaLocation = "http://www.tutorialspoint.com
student.xsd">
```

After defining the **XMLSchema-instance** xsi, use **schemaLocation** attribute. This attribute has two values, namespace and location of XML Schema, to be used separated by a space. It is optional.

XML Parsing

- Representing data using XML is one side of the coin. Though XML provides infinite flexibility to represent data of any complexity, it's of no use if we cannot read the data back from XML file. This is where parsing an XML document comes into picture.
- Parsing an XML document is nothing but reading the data back from the document. This is the other side of XML.
- The application that parses an XML document is called an XML parser.
- Parser can read, validate and parse the xml document.
- The purpose of an XML parser is to make some interfaces available to an application so that it can modify and read the contents of an XML document. The generation of these interfaces is based on two XML standards namely **SAX** and **DOM**.
- Parsing means "reading" the XML file/string and getting its content according to the structure, usually to use them in a program.
- The Java API for XML Processing (JAXP) is for processing XML data using applications written in the Java programming language. JAXP provides three types of parsers:
    - Simple API for XML Parsing **(SAX),**
    - Document Object Model **(DOM),** and
    - Streaming API for XML **(StAX)**.

  It provides different packages containing different classes and interfaces for processing xml documents.

DOM

- DOM stands for Document Object Model.
- In this model, an XML document is represented as a tree of nodes.
- A parser based on this model, can traverse the through the nodes to read the data or modify the data by removing the nodes. In DOM parsing, the entire XML must be loaded into the memory before reading or modifying the document. Because of this reason, DOM based parsing should be used only when the XML document is small enough not to cause any memory issues.
- The Document Object Model (DOM) is an official recommendation of the World Wide Web Consortium (W3C). It defines an interface that enables programs to access and update the style, structure, and contents of XML documents. XML parsers that support DOM implement this interface.
- We should use a DOM parser when −
    - We need to know a lot about the structure of a document.
    - We need to move parts of an XML document around (we might want to sort certain elements, for example).
    - We need to use the information in an XML document more than once.
- When we parse an XML document with a DOM parser, we get back a tree structure that contains all of the elements of your document. The DOM provides a variety of functions we can use to examine the contents and structure of the document.
- In DOM Parsing, the entire XML is loaded in the memory as a tree of nodes. We call this entire tree as the Document. Once the XML is loaded, we can traverse through the entire tree and get the information about every node. In DOM, there are different types of nodes corresponding to the component types of an XML document. These are:
    - Element Node
    - Attribute Node
    - CDATA Node etc.
- The DOM defines several Java interfaces. Here are the most common interfaces −
    - **Node** − The base datatype of the DOM.
    - **Element** − The vast majority of the objects you'll deal with are Elements.
    - **Attr** − Represents an attribute of an element.
    - **Text** − The actual content of an Element or Attr.
    - **Document** − Represents the entire XML document. A Document object is often referred to as a DOM tree.
- When we are working with DOM, there are several methods we'll use often −

  `Document.getDocumentElement()` − Returns the root element of the document.

  `Node.getFirstChild()` − Returns the first child of a given Node.

  `Node.getLastChild()` − Returns the last child of a given Node.

`Node.getNextSibling()` − These methods return the next sibling of a given Node.

`Node.getPreviousSibling()` − These methods return the previous sibling of a given Node.

`Node.getAttribute(attrName)` − For a given Node, it returns the attribute with the requested name.

The most important interfaces we use in DOM parsing are the Document and Node. The basic process to implement DOM parsing is shown below:

- Import XML-related packages.
- Get an instance of **DocumentBuilderFactory** class.
- Create a **DocumentBuilde**r object using the factory class.
- Invoke the **parse()** method on the builder object by passing the XML file. This method then returns a Document object.
- Use the Document object to process the XML.
    - ✓ Extract the root element
    - ✓ Examine attributes
    - ✓ Examine sub-elements

Import XML-related packages

```
import org.w3c.dom.*;

import javax.xml.parsers.*;

import java.io.*;
```

Get an instance of DocumentBuilderFactory class and Create a DocumentBuilder

```
DocumentBuilderFactory factory =

DocumentBuilderFactory.newInstance();

DocumentBuilder builder = factory.newDocumentBuilder();
```

Create a Document from a file or stream using parse() method

```
StringBuilder xmlStringBuilder = new StringBuilder();
xmlStringBuilder.append("<?xml version="1.0"?> <class> </class>");
ByteArrayInputStream input = new ByteArrayInputStream(
xmlStringBuilder.toString().getBytes("UTF-8"));
Document doc = builder.parse(input);
```

Extract the root element

```
Element root = document.getDocumentElement();
```

Examine attributes

```
//returns specific attribute
getAttribute("attributeName");


//returns a Map (table) of names/values
getAttributes();
```

Examine sub-elements

```
//returns a list of subelements of specified name
getElementsByTagName("subelementName");


//returns a list of all child nodes
getChildNodes();
```

Working Example

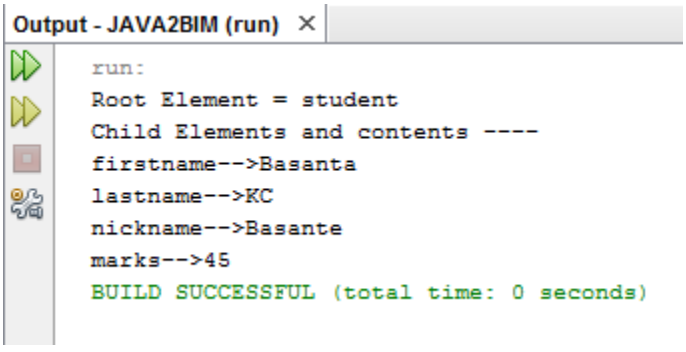
File : student.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
It is a simple xml file
-->
<student>
      <firstname>Basanta</firstname>
      <lastname>KC</lastname>
      <nickname>Basante</nickname>
      <marks>45</marks>
   </student>
```

File: DomParserDemo.java

```
package xmlandjava;
import java.io.File;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
public class DomParserDemo {
    public static void main(String[] args) {
      try {
         File inputFile = new File("student.xml");
         DocumentBuilderFactory dbFactory;
         dbFactory= DocumentBuilderFactory.newInstance();
```

```
            DocumentBuilder dBuilder =
dbFactory.newDocumentBuilder();
            Document doc = dBuilder.parse(inputFile);
            Element elmnt = doc.getDocumentElement();
            System.out.println("Root Element =
"+elmnt.getNodeName());
            NodeList nodes = elmnt.getChildNodes();
            System.out.println("Child Elements and contents ----");
            for (int i =0 ; i<nodes.getLength();i++){
                Node node = nodes.item(i);
                if(node.getNodeType()==Node.ELEMENT_NODE)
                {
                    String n = node.getNodeName();
                    String content = node.getTextContent();
                    System.out.println(n + "-->"+content);
                }
            }
        }catch(Exception e ){
            e.printStackTrace();
        }}
}
```



SAX

- SAX is abbreviated for Simple API for XML.
- SAX parsing is based on event model in which sequences of events are generated for each and every tag in the XML document.
- Based on the type of events, the application can take appropriate action. The various events a SAX based parser generates are:
  - ✓ **startDocument:** This event indicates the beginning of the XML document
  - ✓ **startElement** : This event indicates the beginning of a tag or a node
  - ✓ **characters** : This represents the data within the tag
  - ✓ **endElement** : This indicates the end of the tag
  - ✓ **endDocument :** This event indicates the end of the XML document
- SAX based parsing can be used only for reading the data from XML and not for modifying the contents. Moreover, SAX parsing is more efficient in situations where the XML document is huge and we only want to extract a small piece of data from it.
- Unlike a DOM parser, a SAX parser creates no parse tree. SAX is a streaming interface for XML, which means that applications using SAX receive event notifications about the XML document being processed an element, and attribute, at a time in sequential order starting at the top of the document, and ending with the closing of the ROOT element.
  - → Reads an XML document from top to bottom, recognizing the tokens that make up a well-formed XML document.
  - → Tokens are processed in the same order that they appear in the document.
  - → Reports the application program the nature of tokens that the parser has encountered as they occur.
  - → The application program provides an "event" handler that must be registered with the parser.
  - → As the tokens are identified, callback methods in the handler are invoked with the relevant information.

We should use a SAX parser when −

- We can process the XML document in a linear fashion from top to down.
- The document is not deeply nested.
- We are processing a very large XML document whose DOM tree would consume too much memory. Typical DOM implementations use ten bytes of memory to represent one byte of XML.
- The problem to be solved involves only a part of the XML document.

ContentHandler Interface

This interface specifies the callback methods that the SAX parser uses to notify an application program of the components of the XML document that it has seen.

- **`void startDocument()` –** Called at the beginning of a document.
- **`void endDocument()` –** Called at the end of a document.
- **`void startElement(String uri, String localName, String qName, Attributes atts)` –** Called at the beginning of an element.
- **`void endElement(String uri, String localName,String qName)`** − Called at the end of an element.
- **`void characters(char[] ch, int start, int length)`** − Called when character data is encountered..
- **`void skippedEntity(String name)`** − Called when an unresolved entity is encountered.

Attributes Interface

This interface specifies methods for processing the attributes connected to an element.

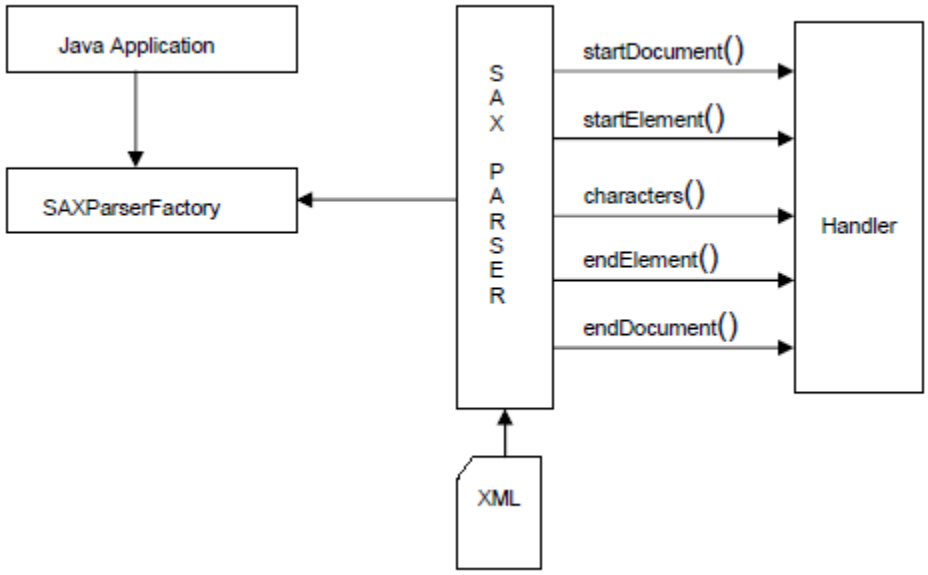
`int getLength()` − Returns number of attributes.

`String getQName(int index)`

`String getValue(int index)`

`String getValue(String qname)`

Following figure gives a complete idea of SAX Parsing



From the above figure, we observe the following things:

- Java application uses a **SAXParserFactory** to get a **SAXParser**
- The **SAXParser** takes the XML document to parse, and begins generating the events and send them to a handler that handles the events.

Steps for parsing a document with SAX Parser

1. **Write a handler class to handle events**.

   To write a handler class, we need to do the following 2 things.

   a) The class must extend the **DefaultHandler** class.
   b) The handler class should override the methods namely *startDocument(), startElement()* etc. These methods are known as callback methods that get invoked automatically when the parsing begins.

2. **Write a class to initiate parsing with the following steps**
   a. Get an instance of **SAXParserFactory** class
   b. Get a **SAXParser** from the factory class
   c. Invoke the **parse()** method by passing the XML file and a reference to handler object.

Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
file: employees.xml
-->
<employees>
  <employee emptype = "fulltime">
      <empid> 360</empid>
      <empname> Urmila </empname>
      <empsalary> 98000 </empsalary>
  </employee>
</employees>
```

```
//File: MyHandler.java
package xmlandjava;
import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.DefaultHandler;

public class MyHandler extends DefaultHandler{
   boolean bId = false;
   boolean bName = false;
   boolean bSalary = false;
   @Override
   public void startDocument(){
       System.out.println("\n Parsing is satarted");
   }
```

```
   @Override
   public void startElement(String uri,
   String localName, String qName, Attributes attributes) throws
SAXException{
      System.out.println("Start Element :" + qName);
      if (qName.equalsIgnoreCase("employee")) {
         String empType = attributes.getValue("emptype");
         System.out.println("Employee Type  : " + empType);
      } else if (qName.equalsIgnoreCase("empid")) {
         bId = true;
      } else if (qName.equalsIgnoreCase("empname")) {
         bName = true;
      } else if (qName.equalsIgnoreCase("empsalary")) {
         bSalary = true;
       }
   }
@Override
   public void endElement(String uri,
      String localName, String qName) throws SAXException {
         System.out.println("End Element :" + qName);
   }
   @Override
   public void characters(char ch[], int start, int length)
throws SAXException {
          if (bId) {
               System.out.println("Employee ID : " + new
String(ch, start, length));
               bId = false;
          }
          if (bName) {
```

```
                System.out.println("Employee Name : " + new
String(ch, start, length));
                bName = false;
            }
            if (bSalary) {
                System.out.println("Employee Salary : " + new
String(ch, start, length));
                bSalary = false;
            }
    }
    @Override
    public void endDocument(){
        System.out.println("\n Parsing is completed");
    }
}
```

```
//File: SaxParserDemo.java
package xmlandjava;
import java.io.File;
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;
public class SaxParserDemo {
    public static void main(String[] args) {
      try {
         File inputFile = new File("employees.xml");
         SAXParserFactory factory =
SAXParserFactory.newInstance();
         SAXParser saxParser = factory.newSAXParser();
         MyHandler myhandler = new MyHandler();
```

```
            saxParser.parse(inputFile, myhandler);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```