Java Programming

Java & XML

Introduction to XML

XML (eXtensible Markup Language), like HTML, is a *markup language* for marking up the structure of a text document. It is a subset of Standard General Markup Language (SGML). XML is not a programming language like Java or C#. It is developed and maintained by World-Wide Web Consortium (W3C) @ www.w3c.org. (W3C also maintains specifications for HTTP, HTML, XHTML, CSS, among others.)

XML is a family of technologies, which includes:

- XML: the markup language itself.
- DTD (Document Type Definition) and XML Schema: define the structure of an XML document,
- XSL (XML Style sheet Language), XSLT (XSL Transformation), XSL-FO (XSL Formatting Objects): for presentation.
- XPath, XLink, XPointer, XQuery: for locating, linking and query document.
- DOM (Document Object Model), SAX (Simple API for XML): XML parser.
- XQL (XML Query Language), XSQL: for querying databases.
- and more...

I assume that you have some basic understanding of the HTML.

Why XML?

The HTML's original objective of letting the document author to focus on the contents of the document and leave the actual appearance of the document to the browser, has gone out of control. Many HTML documents have more markup tags than the contents. Worse still, many of the markup tags are dealing with the appearance of the document (e.g., <font>) rather than the contents (e.g., <h1>).

HTML has grown into a huge and complex language, with more than hundred markup tags in its latest version. On one hand, despite these many tags, specific applications (such as e-commerce and Mathematical formula) are asking for more tags, On the other hand, many tags are not used frequently by many applications and can be removed. Furthermore, many of the HTML tags (e.g., `<font>`, `<span>`, `<div>`) are meant for presentation rather than the contents.

Objectives of XML

XML aims to:

- Focus on the content rather than the appearance of the documents.
- Resolve the conflicting demands on tags: on one hand, specialized applications need more tags; on the other hand, many tags are not frequently used and can be removed.

XML adapts the following principles to meet the above objectives:

- XML has no pre-defined tags: The authors of the documents creates their own tags to suit their applications. Hence, XML is flexible and extensible.
- XML has strict syntax: HTML is sloppy and loose in syntax. HTML browsers need to correct sloppy HTML scripts, resulting in complex and heavy browser. By tightening the syntax, XML browser is smaller, lighter and faster.

Applications for XML

XML is useful for these applications:

- Data exchange between computer systems: XML is platform- and computer-language-neutral and text-based, which greatly facilitates exchanging of data between two computer systems. For example, two e-commerce partners can use an agree-upon XML format to exchange purchase orders and invoices electronically, and directly fed into their computer systems.
- Data storage: Unlike databases which is platform- and language-dependent, XML provide a platform-neutral mean for data storage.
- Specialized publishing: XML can be used for marking up documents for specialized applications, such as e-commerce, scientific documents, Mathematical formula, e-books, among others.

XML Documents

XML (eXtensible Markup Language) is a *markup language* for marking up the structure of a text document. Unlike HTML:

- XML has no pre-defined tags. You create your own tags for your specific applications. XML tag names are supposed to be self-describing.
- XML's tags are used to mark the *meaning* of the content, rather than the appearance or presentation of the content.
- XML has strict syntax rules, and is case sensitive.

Like HTML, XML uses *markup tags* to markup so-called *elements*. There are two kinds of elements:

1. Non-empty element (or container element): The content is enclosed within a pair of matching start-tag and end-tag, in the form of `<element-name>contents</element-name>`. The end-tag is preceded by a forward slash "`/`" and has the same name as the start-tag. For example, `<title>Java for dummies</title>`.

2. Empty element (or standalone element): Empty element has no data content. You can use a shorthand notation by closing the start tag with a forward slash "/" and omit the end tag, in the form of `<empty-element-name />`. Note that it has the same meaning as `<empty-element-name></empty-element-name>`. For examples, `<out_of_print>`, `<preferred_customer>`.

An XML document exhibits a *tree* structure. It has one (and only one) *root element*, and the elements are properly nested. XML documents are text-based, human readable, and are meant to be self-describing. The following is an example of an XML document for a bookstore:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- bookstore.xml -->
<bookstore>
  <book ISBN="0123456001">
    <title>Java For Dummies</title>
    <author>Tan Ah Teck</author>
    <category>Programming</category>
    <year>2009</year>
    <edition>7</edition>
    <price>19.99</price>
  </book>
  <book ISBN="0123456002">
    <title>More Java For Dummies</title>
    <author>Tan Ah Teck</author>
    <category>Programming</category>
    <year>2008</year>
    <price>25.99</price>
  </book>
  <book ISBN="0123456010">
    <title>The Complete Guide to Fishing</title>
    <author>Bill Jones</author>
    <author>James Cook</author>
    <author>Mary Turing</author>
    <category>Fishing</category>
    <category>Leisure</category>
    <language>French</language>
    <year>2000</year>
    <edition>2</edition>
    <price>49.99</price>
  </book>
</bookstore>
```

```
XML declaration
XML comment
Root element start-tag (one and only one root)
Child element start-tag (with an attribute in name="value" pair)


   (proper nesting of child elements)



Child element end-tag
Second child element start-tag




Second child element end-tag
Third child element start-tag









Third child element end-tag
Root element end-tag
```

In the above example, `<bookstore>` is the *root* element. It has one child element `<book>`, which in turn has several children (`<title>`, `<author>`, `<category>`, etc). Some of the elements such as `<author>` may appear more than once to carry multiple values. The element start-tag may contain so-called *attributes* in the form of `attribute_name="attribute_value"` pairs. The following figure shows the appearance of opening an XML document on a web browser. Observe the tree structure. You can click on the "+" and "-" sign to expand and collapse a portion of the tree.

Another example of an XML address book is as follows:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- addressbook.xml -->
<address_book>
  <person>
     <name>Tan Ah Teck</name>
     <tel type="mobile" preferred="yes">98888888</tel>
     <tel type="home">68888888</tel>
     <email preferred="yes">tan@abc.com</email>
     <email>teck@xyz.com</email>
     <address type="home">1 Happy Ave</address>
  </person>
  <person>
     <name>Mohd Ali</name>
     <tel type="mobile" preferred="yes">97777777</tel>
     <tel type="office">67777777</tel>
```

```
    <email>ali@abc.com</email>
  </person>
</address_book>
```

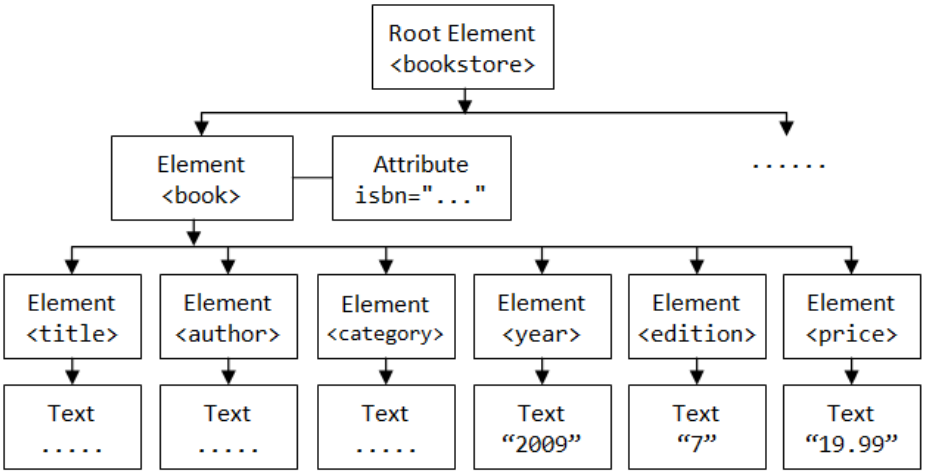
XML Syntax

There are currently two versions of XML specifications: XML 1.0 and XML 1.1, maintained by W3C.

- *Element* is the basic unit of an XML document. Each XML element must have a *start-tag* and *end-tag*. The tags are enclosed in angle brackets, e.g., `<title>...</title>`. Unlike HTML, closing tag is mandatory. *Empty-element tag* (or *standalone tag*) must be properly closed, e.g., `<out_of_print />`.
- An XML element includes its start-tag, enclosing character data and/or child elements, and the end-tag.
- The element's name can contain letters, numbers, and other Unicode characters, but NOT white spaces. The name must start with a letter, underscore "_", or colon ":", but cannot start with certain reserved words such as `xml`.
- Each XML document must have one (and only one) *root* element.
- XML elements must be *properly nested*. For example, `<book><title>...</book></title>` is incorrectly nested.
- XML is *case sensitive*. For example, `<book>` and `<Book>` are considered two different tags.
- The start-tag may contain *attributes* in the form of `attribute_name="attribute_value"` pairs. Attributes are used to provide extra information about the element. Unlike HTML, the `attribute_value` of an XML attribute must be properly quoted (either in double quotes or single quotes).
- Certain characters, such as `<`, `>`, which are used in XML syntax, must be replaced with so-called *entity references* in the form of `&name;`. XML has five pre-defined entity references: `&lt;` (<), `&gt;` (>), `&amp;` (&), `&quot;` ("), and `&apos;` (').
- XML comment takes the form of `<!-- comment texts -->`, which is the same as HTML.
- Unlike HTML, white spaces in the text are preserved. New-line is represented by a Line Feed (`LF`) character (`0AH`).

Well-Form XML Documents

An XML document is *well-formed*, if its structure meets the XML specification, i.e., it is syntactically correct. A well-formed XML document exhibits a tree-like structure, and can be processed by an XML processor. For example, the tree structure of the "bookstore.xml" is as follows:



Structure of XML Documents

An XML document comprises of the following basic units:

- Element: includes the start-tag, the enclosing character data and/or nested elements, and the end-tag.
- Attribute: defined in the start-tag to provide extra information about the element, in the form of `attribute_name="attribute_value"`.
- Entities References: in the form of `&name;`, e.g., `&lt;` (<), `&gt;` (>), `&amp;` (&), `&quot;` ("), and `&apos;` (').
- Character References: in the form of `&#decimal-number;` or `&#xhex-code;` for replacing any Unicode character, e.g., both `&#169;` and `&#xA9;` can be used for copyright symbol ©.
- PCDATA (Parsed Character Data): Text between start-tag and end-tag that will be examined by the parser for entity references and nested elements.
- CDATA (Character Data): Text between start-tag and end-tag that will NOT be examined by the parser for entity references and nested tags.

CDATA Section

As mentioned, special characters (such as "<", ">", "&") must be referenced through pre-defined entities (such as `&lt;`, `&gt;`, `&apos;`). If your texts contain many special characters, it is cumbersome to replace all of them. Instead a "CDATA (character data) section" can be used.CDATA sections are delimited by "`<[CDATA[`" and "`]]>`". The XML processor ignores all the markup within the CDATA section except "`]]>`". CDATA sections are used for program codes, such as JavaScript or Perl codes.

Processing Instruction (PI)

Processing Instruction (PI) tells an application to perform a specific task. It is a *command* to XML parser or an application program that uses the XML document. PI can be used for inserting non-XML statements, such as scripts, into the document, to be passed to an application for processing.

A PI begins with a "`<?`" and ends with "`?>`". For example,

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

is a PI (in this case, a XML instruction) that attaches a XSL style sheet to the XML document for layout processing.

The XML Declaration (the first line of an XML document) is also a processing instruction.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

XML is Extensible

XML is classified as an *extensible language* as it does not have a pre-defined set of tags. You can create and extend your own tags to suit your application.

You can also expand on an existing set of tags without breaking your existing applications. For example, in the bookstore example, we can define more tags such as `<number_of_pages>`, `<weight>`, `<dimension>` for shipping purpose, without breaking the existing applications.

Best Practices

Naming Convention:

- Names should be self-described.
- Names shall be nouns, and may consist of a few words. Use underscore "_" to join the words, e.g., `<first_name>`, `<last_name>`.
- Avoid colon ":" character, which is reserved for namespace. Avoid dot ".", which could be confused with object property. Avoid dash "-", which could be confused with subtract operation.

You can use either element or attribute to carry information. In the above example, `title` could be an element, or an attribute inside the `book` element like `ISBN`. Generally, try to avoid attributes, as attributes are harder to read, cannot carry multiple values, and not easily expandable.

HTML vs. XML

- XML defines data, HTML defines both data and presentation.
- XML is case sensitive, HTML is not.
- XML has strict syntax, HTML's syntax is loose and sloppy.
  - An XML element must begin with a start-tag and end with a end-tag. Empty element's tag must be closed with a forward slash "/". HTML's end-tag may be omitted.
  - XML elements must be properly nested within the root element.
  - XML document must have one (and only one) root element
  - XML elements must be properly nested.
  - XML attribute values must be properly quoted.
  - The same attribute can not appear more than once in the same element.

XML DTD & Schema

Document Type Definition (DTD) and Schema are techniques for defining the *structure* of a specific type of XML documents, via a list of legal elements and attributes. It is a formal description of the structure of an XML document, i.e., which elements are allowed, which elements must be present, which elements are optional, the sequence and relationship of the elements.

Document Type Definition (DTD)

Document Type Definition (DTD) is used to define the structure of an XML document. It describes the objects (such as elements, attributes, entities) and the relationship of the objects. It specifies a set of constraints and establishes the trees that are acceptable in an XML document.

A DTD can be declared inside an XML document (i.e., *inline*), or referenced as an *external* file.

An *inline* DTD is wrapped in a `DOCTYPE` declaration, and has the following syntax:

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

For example,

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- bookstore-inlineDTD.xml -->
<!DOCTYPE bookstore [
  <!ELEMENT bookstore (book+)>
  <!ELEMENT book (title, author+, category*, language?, year?, edition?, price)>
    <!ATTLIST book ISBN CDATA #REQUIRED>
  <!ELEMENT title    (#PCDATA)>
  <!ELEMENT author   (#PCDATA)>
  <!ELEMENT category (#PCDATA)>
  <!ELEMENT language (#PCDATA)>
  <!ELEMENT year     (#PCDATA)>
  <!ELEMENT edition  (#PCDATA)>
  <!ELEMENT price    (#PCDATA)>
]>
<bookstore>
  <book ISBN="0123456001">
    <title>Java For Dummies</title>
    <author>Tan Ah Teck</author>
    <category>Programming</category>
    <year>2009</year>
    <edition>7</edition>
    <price>19.99</price>
  </book>
```

```
  <book ISBN="0123456002">
    <title>More Java For Dummies</title>
    <author>Tan Ah Teck</author>
    <category>Programming</category>
    <year>2008</year>
    <price>25.99</price>
  </book>
  <book ISBN="0123456010">
    <title>The Complete Guide to Fishing</title>
    <author>Bill Jones</author>
    <author>James Cook</author>
    <author>Mary Turing</author>
    <category>Fishing</category>
    <category>Leisure</category>
    <language>French</language>
    <year>2000</year>
    <edition>2</edition>
    <price>49.99</price>
  </book>
</bookstore>
```

A DTD can also be stored in an external file. An XML document can reference an external DTD via the following syntax:

```
<!DOCTYPE root-element SYSTEM "DTD-filename">
```

For example,

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- bookstore-externalDTD.xml -->
<!DOCTYPE bookstore SYSTEM "bookstore.dtd">
<bookstore>
  <book ISBN="0123456001">
    <title>Java For Dummies</title>
    <author>Tan Ah Teck</author>
    <category>Programming</category>
    <year>2009</year>
    <edition>7</edition>
    <price>19.99</price>
  </book>
  <book ISBN="0123456002">
    <title>More Java For Dummies</title>
    <author>Tan Ah Teck</author>
    <category>Programming</category>
    <year>2008</year>
    <price>25.99</price>
  </book>
  <book ISBN="0123456010">
    <title>The Complete Guide to Fishing</title>
    <author>Bill Jones</author>
    <author>James Cook</author>
    <author>Mary Turing</author>
    <category>Fishing</category>
    <category>Leisure</category>
    <language>French</language>
    <year>2000</year>
    <edition>2</edition>
    <price>49.99</price>
  </book>
</bookstore>
```

The referenced external DTD "bookstore.dtd" is as follows:

```
<!ELEMENT bookstore (book+)>
<!ELEMENT book (title, author+, category*, language?, year?, edition?, price)>
  <!ATTLIST book ISBN CDATA #REQUIRED>
<!ELEMENT title    (#PCDATA)>
<!ELEMENT author   (#PCDATA)>
<!ELEMENT category (#PCDATA)>
<!ELEMENT language (#PCDATA)>
<!ELEMENT year     (#PCDATA)>
<!ELEMENT edition  (#PCDATA)>
<!ELEMENT price    (#PCDATA)>
```

DTD Syntax

XML's DTD is inherited from SGML's DTD. Hence it has its own syntax, which is different from XML's syntax. A DTD consists of declarations (for element, attributes and so on) such as:

- `<!DOCTYPE ...>`: Document Type declaration.
- `<!ELEMENT ...>`: Element declaration.
- `<!ATTLIST ...>`: Attribute List declaration for an element.
- `<!ENTITY ...>`: Entity declaration.
- `<!NOTATION ...>`: Define Notation for an external entity.

Element Declaration

"Element" is the basic building block of XML documents. The structure of elements can be defined in DTD using the following syntax:

```
// Declaring "element" in DTD
<!ELEMENT element-name (element-content)>



<!ELEMENT element-name category>
```

```
// Examples
<!ELEMENT title (#PCDATA)>              // contain parsed character data
<!ELEMENT name (first_name, last_name)> // contain child elements
<!ELEMENT person (name, address?, email+, hobby*)>
        // name (one), ? (zero or one), + (one or more), * (zero or more)
<!ELEMENT message (#PCDATA|(head, body))>   // | (or)

<!ELEMENT out_of_print EMPTY>   // an empty-element
<!ELEMENT message ANY>          // combination of all
```

Category (or special content)

- `#PCDATA` (Parsed Character Data): texts that will be examined for entity references and tags.
- `EMPTY`: Empty element (for leaf element only).
- `ANY`: unrestrictive.

Occurrence Indicators:

- "`+`": one or more occurrences.
- "`*`": zero or more occurrences.
- "`?`": zero or exactly one occurrence.
- No occurrence indicator: exactly one.

Connector:

- "`,`": indicate the sequence of the child elements.
- "`|`": choices (or) - choose only one of them.

Attribute List Declaration

An element start-tag may contain "attribute(s)" to provide additional information for the element. The structure of attributes can be declared in DTD as follows:

```
// Declaring "attribute" in DTD
<!ATTLIST element-name
  attribute-1-name attribute-1-type default
  attribute-2-name attribute-2-type default
  ...
>
// default
default-value|#REQUIRED|#IMPLIED|#FIXED value
```

```
// Examples
<!ATTLIST payment mode CDATA "cash">
<!ATTLIST trade action (buy|sell) #REQUIRED>  // enumeration type
<!ATTLIST person
    email CDATA #REQUIRED
    handphone CDATA #REQUIRED
>
```

Attribute types:

- `CDATA` (Character Data): text strings that will not be parsed for entity references and tags.
- `ID`: an unique identifier.
- `IDREF`, `IDREFS`: reference(s) to a previously defined ID.
- `ENTITY`, `ENTITIES`: external entity(entities).
- `NMTOKEN`, `NMTOKENS`: word(s) not containing spaces.
- Enumeration: list of `NMTOKEN` separated by "`|`".

Default:

- `#REQUIRED`: must be provided in the document.
- `#IMPLIED`: use the application default.
- `#FIXED` *value*: must use this *value*.
- A literal default value.

Attribute List declaration can appear anywhere in the DTD. For readability, it is best to follow the element declaration.

Entity Declaration

A DTD "entity" is a *variable* for defining *replacement text* or *special characters*. Once an entity is defined, you can use the *entity reference*, in the form of `&entity-name;` to obtain the value of the variable. Entities can be declared inline or external.

```
// Inline "entity" declaration
<!ENTITY entity-name "entity-value">
// External "entity" declaration
<!ENTITY entity-name SYSTEM "url">
```

```
// Examples
<!ENTITY author "Tan Ah Teck">  // In XML documents, entity referenced as &author;

<!ENTITY author SYSTEM "http://www.xyz.com/entities.dtd">
```

Valid XML Document

A well-formed XML document is *valid* if it meets the constraints spelled out in a DTD (Document Type Definition) or an XML Schema, imposed by a specific application.

Usage of DTD

DTD defines the structure of a certain type of XML documents, which could facilitate exchanging of documents between computer systems electronically. It also helps in standardizing a certain class of documents.

Limitations of DTD

- DTD has its own syntax (which is inherited from SGML DTD) and requires a dedicate processing tool to process the content. It does not use XML syntax and XML processor.
- DTD does not support object-oriented concepts such as hierarchies and inheritance.
- DTD's data type is limited to text string; and does not support other data types like number, date etc.
- DTD does not support namespaces.
- DTD's occurrence indicator is limited to 0, 1 and many; cannot support a specific number such as 8.

XML Schema

An XML Schemas is developed by W3C, which overcomes the limitation of DTD and meant to replace DTD. In brief, the XML Schema:

- is a well-formed XML document, which uses XML syntax.
- is object-oriented, support concepts like inheritance.
- supports namespaces.
- supports more data type.
- more element occurrence indicators.

[MORE]

XML Namespaces

XML is extensible. Namespace is needed to avoid naming conflict, when reusing XML elements. For example, two companies may have the same element <address>, which carries different contents. To differentiate them, either force the companies to use different names (impossible!) or use a namespace prefix to identify the companies (or applications), e.g., <abc:address> and <xyz:address>.

XML namespace associates a prefix to a unique URL. (URL is based on Internet domain name which is guaranteed to be unique among organizations.) E.g.,

```
xmlns:abc="http://www.abc.com/XSL/1.0"
```

The namespace prefix can be treated as a shorthand for a unique URL, to ensure uniqueness in naming and avoid naming conflict. The URL needs not be physically present.

For example,

```
<?xml version="1.0"?>
<book_review
  xmlns:abc="http://abc.com/rating/v10"
  xmlns:xyz="http://xyz.com/book/rating"
  xmlns="http://mydotcom.com/rating/book">
  <book title="XML for dummies">
    <abc:rating>5</abc:rating>
    <xyz:rating>Excellent</xyz:rating>
    <rating>0.7</rating>
  <book>
     ...
</book_review>
```

Three namespaces are used to distinguish the same element `<rating>`. The third `xmlns` declaration is for the so-called *default namespace*, i.e., those elements without a prefix.

The namespace is valid within the element where it is declared (includes all the child elements it contains). For example,

```
<?xml version="1.0"?>
<book_review xmlns="http://my.com/rating/book">
  ......
  <abc:rating xmlns:abc="http://abc.com/rating/v10">
  5
  </abc:rating>
  <xyz:rating xmlns:xyz="http://xyz.com/rating/book">
  Excellent
  </xyz:rating>
  <rating>0.7</rating>
  ......
</book_review>
```

XML Style Sheets

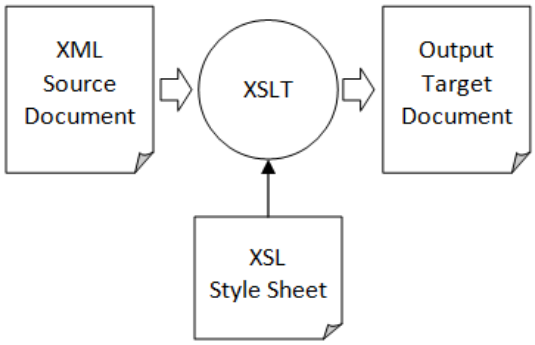
XML focus on the content of the document and gives no clue on the appearance or presentation. Style sheets can be used to provide styling information for displaying XML documents. Different style sheets can be applied to the same XML document for display on different platform or devices (desktop browser, PDA, mobile phone).

W3C has developed two style sheet standards, that can be used with XML documents:

- Cascading Style Sheet (CSS): originally used to support HTML, has been extended to support XML.
- XML Style Language (XSL): supports advanced styling for XML documents, such as creating a table of contents. XSL is organized in two parts: XSLT (XSL Transformation) and XSLFO (XSL Formatting Objects).

XSL Transformation (XSLT)

XSL Transformation (XSLT) is a text-based transformation process that merges a textual XML source document with a XSL style sheet to procedure a target document.



XSL Style Sheet

An XSL style sheet is a well-formed XML document. The root element `<xsl:stylesheet>` declares two namespaces: `xsl` for the XSL vocabulary and default for the target HTML (note: w3 and not w3c!), as follows:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns="http://www.w3.org/TR/REC-html40"
  version="1.0">
    ......
</xsl:stylesheet>
```

The bulk of the style sheet is a list of XSL template for matching with the source element and produce the target document. For example,

```
<xsl:template match="address-book/person">
  <H2><xsl:apply-templates/></H2>
</xsl:template>
```

An XSL template consists of two parts:

- A matching PATH in the match attribute, and
- The action to be taken upon a successful match in the content of the template.

XPath (XML Path Language)

The matching criteria is specified using XPath (XML Path Language). XPath specification defines how a specific item within an XML document can be located. XPath's syntax is in line with the tree structure of the XML document. It lists the elements along the path of a tree, separated by the path separator "`/`". For example, "`/bookstore/book/title`", "`/addressbook/person/name`".

To match the root, use:

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

There two two types of path: absolute and relative. Absolute path (such as "`/bookstore/book/title`") begins from the root "/". Relative path (such as "`person/email`") is relative to the current element being processed.

"`//`" can be used to indicate all children and grandchildren. For example, "`bookstore//title`" matches "`bookstore/title`", "`bookstore/book/title`", "`bookstore/book/chapter/title`".

To match any element, wildcard character "`*`" can be used. For example,

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

To match a few elements, you can combine the paths with the "`|`" (or) character. For example,

```
<xsl:template match="author|title|category">......</xsl:template>
```

To match on attribute, use the syntax:

```
element[@attribute-name='attribute-value']
```

For example:

```
match="email[@preferred='yes']"
```

The attribute-value is optional. If omitted, it matches if the attribute is present. For example,

```
match="email[@preferred]"
```

[TODO] more

Parsing XML Documents

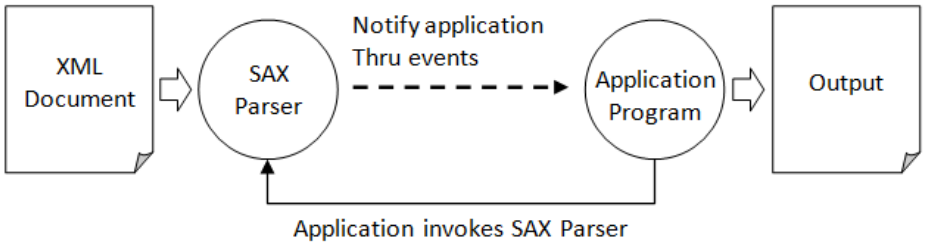
To process the data contained in XML documents, you need to write a application program (in a programming language such as Java, JavaScript). The program makes use of an XML parser to tokenize and retrieve the data/objects in the XML documents. An *XML parser* is the software that sits between the application and the XML documents to shield the application developer from the intricacies of the XML syntax. The parser reads a raw XML document, ensures that is well-formed, and may validate the document against a DTD or schema.

There are two standard APIs for parsing XML documents:

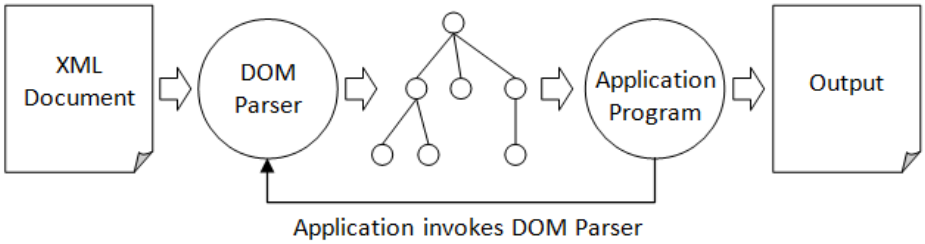
1. SAX (Simple API for XML)

2. DOM (Document Object Model)

SAX is an *event-driven* API. The SAX API defines a number of callback methods, which will be called when events occur during parsing. The SAX parser reads an XML document and generate events as it finds elements, attributes, or data in the document. There are events for document start, document end, element start-tags, element end-tags, attributes, text context, entities, processing instructions, comments and others.



DOM is an *object-oriented* API. The DOM parser explicitly builds an object model, in the form of a tree structure, to represent an XML document. Your application can then manipulate the nodes in the tree. DOM is a platform- and language-independent interface for processing XML documents. The DOM API defines the mechanism for querying, traversing and manipulating the object model built.



The JAXP (Java APIs for XML Processing) provides a common interface for creating, parsing and manipulating XML documents using the standard SAX, DOM and XSLTs.

SAX (Simple API for XML)

Let us begin with an example. Below is a simple SAX parser program to display all the books in the "`bookstore.xml`".

```
import java.io.File;
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;
import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.DefaultHandler;

/**
 * Use SAX Parser to display all books: isbn, title and authors.
 */
public class SAXParserBookstore {
   private String currentElement;
   private int bookCount = 1;

   // Constructor
   public SAXParserBookstore() {
      try {
         SAXParserFactory factory = SAXParserFactory.newInstance();
         SAXParser saxParser = factory.newSAXParser();
         saxParser.parse(new File("bookstore.xml"), new MyHandler());
      } catch (Exception e) {
         e.printStackTrace();
      }
   }

   // Entry main method
   public static void main(String args[]) {
      new SAXParserBookstore();
   }

   /*
    * Inner class for the Callback Handlers.
    */
   class MyHandler extends DefaultHandler {
      // Callback to handle element start tag
      @Override
      public void startElement(String uri, String localName, String qName,
            Attributes attributes) throws SAXException {
         currentElement = qName;
         if (currentElement.equals("book")) {
            System.out.println("Book " + bookCount);
            bookCount++;
            String isbn = attributes.getValue("ISBN");
            System.out.println("\tISBN:\t" + isbn);
         }
      }

      // Callback to handle element end tag
```

```
      @Override
      public void endElement(String uri, String localName, String qName)
            throws SAXException {
         currentElement = "";
      }

      // Callback to handle the character text data inside an element
      @Override
      public void characters(char[] chars, int start, int length) throws SAXException {
         if (currentElement.equals("title")) {
            System.out.println("\tTitle:\t" + new String(chars, start, length));
         } else if (currentElement.equals("author")) {
            System.out.println("\tAuthor:\t" + new String(chars, start, length));
         }
      }
   }
}
```

The expected output is as follows:

```
Book 1
        ISBN:   0123456001
        Title:  Java For Dummies
        Author: Tan Ah Teck
Book 2
        ISBN:   0123456002
        Title:  More Java For Dummies
        Author: Tan Ah Teck
Book 3
        ISBN:   0123456010
        Title:  The Complete Guide to Fishing
        Author: Bill Jones
        Author: James Cook
        Author: Mary Turing
```

JDK provides a SAX parser (called `SAXParser`) and also a DOM builer (called `DocumentBuilder`) in package `javax.xml.parsers`.

To use the `SAXParser`, you need to first create a `SAXParserFactory`, and then obtain a `SAXParser` from the factory. You can then use the `parse()` method to parse an XML document. The `parse()` methods requires two arguments, the XML document to be parsed (in `File`, `InputStream`, `InputSource`, or as a `String`), and a so-called *callback handler*.

```
SAXParserFactory factory = SAXParserFactory.newInstance();   // Create a SAX parser factory
SAXParser saxParser = factory.newSAXParser();                // Obtain a SAX parser
saxParser.parse(new File("bookstore.xml"), new MyHandler()); // Parse the given XML document using the callback handler
```

SAX is an event-driven API. It defines a set of *callback handler methods* that will be invoked when events occur during parsing. JDK provides a `DefaultHandler` class (in package `org.xml.sax.helpers`). You can subclass this `DefaultHandler` and override the callback handler methods to implement your programming logic. In the above example, the subclass of `DefaultHandler` is programmed as an inner class to access the private variables of the outer class.

The commonly-used callback methods are:

```
// Callback to handle the start/end of the document
void startDocument()
void endDocument()
// Callback to handle the element start/end tag
void startElement(String uri, String localName, String qName, Attributes attributes)
void endElement(String uri, String localName, String qName)
// Callback to handle the data inside an element
void characters(char[] chars, int start, int length)
```

In the `startElement()` callback methods, you can access the attributes via the `Attributes` argument. The commonly-used methods of the `Attributes` (in package `org.xml.sax`) are:

```
int getLength()             // Return the number of attributes
String getValue(int qName)  // Return the value of the attribute with the qualified name, or null
String getValue(int index)  // Return the value of the attribute at the index, or null if the index is out of range
String getQName(int index)  // Return the qualified name of the attribute at the index
```

In the above example, I used a private variable calld `currentElement` to maintain the element currently processed. It is set to the qualified name in the `startElement()` method, and cleared in the `endElement()` method.

You can use the `characters()` callback handler method to retrieve the text data within an element. The data is kept in a `char` array, which can be converted to a `String`. You may need to trim the leading and trailing whitespaces of the string. In the above example, I used the `currentElement` to selectively process data of a certain element in the `characters()`.

DOM (Document Object Model)

DOM is a platform- and language-independent API for processing XML documents. The DOM parser loads the XML document, builds an object model in the memory, in the form of a tree comprised of nodes. The DOM API defines the mechanism for querying, traversing the tree; and adding, modifying and deleting the elements and nodes.

```
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.NodeList;
/**
```

```
 * Use DOM Parser to display all books: isbn, title and authors.
 */
public class DOMParserBookStore {
   public static void main(String[] args) throws Exception {
      DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
      DocumentBuilder docBuilder = factory.newDocumentBuilder();
      File file = new File("bookStore.xml");
      Document doc = docBuilder.parse(file);

      // Get a list of all elements in the document
      // The wild card * matches all tags
      NodeList list = doc.getElementsByTagName("*");
      int bookCount = 0;
      for (int i = 0; i < list.getLength(); i++) {
         // Get the elements book (attribute isbn), title, author
         Element element = (Element)list.item(i);
         String nodeName = element.getNodeName();
         if (nodeName.equals("book")) {
            bookCount++;
            System.out.println("BOOK " + bookCount);
            String isbn = element.getAttribute("ISBN");
            System.out.println("\tISBN:\t" + isbn);
         } else if (nodeName.equals("title")) {
            System.out.println("\tTitle:\t"
                  + element.getChildNodes().item(0).getNodeValue());
         } else if (nodeName.equals("author")) {
            System.out.println("\tAuthor:\t"
                  + element.getChildNodes().item(0).getNodeValue());
         }
      }
   }
}
```

The output shall be the same as the previous example using SAX parser.

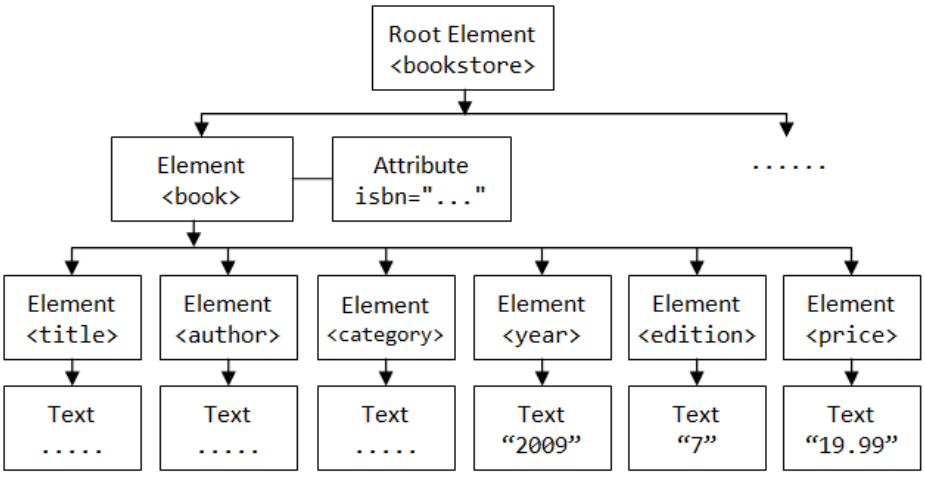
Here is another version of the program to perform the same purpose.

```
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.NodeList;
/**
 * Use DOM Parser to display all books: isbn, title and authors.
 */
public class DOMParserBookStore1 {
   public static void main(String[] args) throws Exception {
      DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
      DocumentBuilder docBuilder = factory.newDocumentBuilder();
      File file = new File("bookStore.xml");
      Document doc = docBuilder.parse(file);

      // Get a list of all <book> elements in the document
      NodeList bookNodes = doc.getElementsByTagName("book");
      for (int i = 0; i < bookNodes.getLength(); i++) {
         Element bookElement = (Element)bookNodes.item(i); // <book> element
         System.out.println("BOOK " + (i+1));
         String isbn = bookElement.getAttribute("ISBN"); // <book> attribute
         System.out.println("\tISBN:\t" + isbn);

         // Get the child elements <title> of <book>, only one
         NodeList titleNodes = bookElement.getElementsByTagName("title");
         Element titleElement = (Element)titleNodes.item(0);
         System.out.println("\tTitle:\t" + titleElement.getTextContent());

         // Get the child elements <author> of <book>, one or more
         NodeList authorNodes = bookElement.getElementsByTagName("author");
         for (int author = 0; author < authorNodes.getLength(); author++) {
            Element authorElement = (Element)authorNodes.item(author);
            System.out.println("\tAuthor:\t" + authorElement.getTextContent());
         }
      }
   }
}
```

We first get a new instance of `DocumentBuilderFactory`, and then obtain an instance of `DocumentBuilder` from the factory (in package `javax.xml.parsers`). After than, we can use the `parse()` method to parse an XML document (as a `File`, `InputStream`, `InputSource`, or `String`) and build a DOM tree to represent the XML document. The `parse()` method returns `Document` object (of package `org.w3c.dom`). Check the API for package `org.w3c.dom` for the various classes used in DOM, such as `Element`, `Node`, `NodeList`, `Text`, etc.

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder docBuilder = factory.newDocumentBuilder();
Document doc = docBuilder.parse(new File("bookStore.xml"));
```

The DOM tree for the bookstore example is illustrated below:



You can use the following method to get the root element of the document:

```
Element root = doc.getDocumentElement();   // return the root element
```

You can search element by tag-names as follows:

```
NodeList bookNodes = doc.getElementsByTagName("book"); // return all the book elements as NodeList
NodeList allNodes = doc.getElementsByTagName("*");     // return all the elements as NodeList,
                                                       //   wild card * matches all elements
```

The `org.w3c.dom.Node` interface defines constants for various type of nodes, such as `Node.ELEMENT_NODE`, `Node.ATTRIBUTE_NODE`, `Node.COMMENT_NODE`, `Node.ENTITY_NODE`, `Node.ENTITY_REFERENCE_NODE`, `Node.PROCESSING_INSTRUCTION_NODE`, `Node.TEXT_NODE`, etc.

Each of the units, such as `Element`, `Comment`, `Entity`, are sub-interface of `Node`, and can be upcast to `Node`.

XML Transformation

[TODO]

Validating XML Documents using DOM Parser

The following program illustrates how to use the DOM parser to validate an XML document against a DTD.

```
import java.io.File;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.transform.OutputKeys;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import org.w3c.dom.Document;
import org.xml.sax.ErrorHandler;
import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;

/**
 * Validate XML document specified via DTD using DOM Parser
 */
public class DOMDTDValidation {
   public static void main(String args[]) {
      try {
         DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
         // Enable the document validation as the document is being parsed.
         factory.setValidating(true);
         DocumentBuilder builder = factory.newDocumentBuilder();

         builder.setErrorHandler(new ErrorHandler() {  // anonymous inner class
            //Ignore the fatal errors
            @Override
            public void fatalError(SAXParseException exception) throws SAXException {}

            // Report validation errors
            @Override
            public void error(SAXParseException e) throws SAXParseException {
               System.out.print("Error at line " + e.getLineNumber() + ": ");
               System.out.println(e.getMessage());
            }

            // Report warnings
            @Override
            public void warning(SAXParseException e) throws SAXParseException {
               System.out.print("Warning at line " + e.getLineNumber() + ": ");
               System.out.println(e.getMessage());
            }
         });

         // External DTD validation
```

```
            Document xmlDocument = builder.parse(new File("bookstore-externalDTD.xml"));
            DOMSource source = new DOMSource(xmlDocument);
            StreamResult result = new StreamResult(System.out);
            TransformerFactory transformerFactory = TransformerFactory.newInstance();
            Transformer transformer = transformerFactory.newTransformer();
            transformer.setOutputProperty(OutputKeys.DOCTYPE_SYSTEM, "bookstore.dtd");
            transformer.transform(source, result);

            // Internal DTD validation
            xmlDocument = builder.parse(new File("bookstore-inlineDTD.xml"));
            source = new DOMSource(xmlDocument);
            result = new StreamResult(System.out);
            transformer.transform(source, result);
         } catch (Exception e) {
            e.printStackTrace();
         }
      }
   }
```

REFERENCES & RESOURCES

- XML and Related Technologies Specifications @ www.w3c.org.
- W3Schools' XML Tutorial @ http://www.w3schools.com.
- Rose India's Programming Tutorial @ http://www.roseindia.net.

Latest version tested: JDK 1.6
Last modified: March, 2009