

Principles and Techniques of DBMS 12

XML

Haopeng Chen

***RE**liable, **IN**telligent and **Scalable** Systems Group (**REINS**)*

Shanghai Jiao Tong University

Shanghai, China

<http://reins.se.sjtu.edu.cn/~chenhp>

e-mail: chen-hp@sjtu.edu.cn

- XML
 - Introducing XML
 - Parsing an XML Documents
 - Validating an XML Documents
 - Locating Information with Xpath
 - XQuery
 - Using Namespaces
 - Generating XML documents
 - XSL Transformations
- XML DBMS
 - Sedna

- "The Extensible Markup Language (XML) has replaced Java, Design Patterns, and Object Technology as the software industry's solution to world hunger."
 - Half-jokingly stated in the preface of the book Essential XML by Don Box et al. (Addison-Wesley Professional 2000)

- A property file contains a set of name/value pairs, such as
fontname=Times Roman
fontsize=12
windowsize=400 200
color=0 50 100
- Consider the fontname/fontsize entries in the example. It would be more object oriented to have a single entry:
font=Times Roman 12
But then parsing the font description gets ugly
- Property files have a single flat hierarchy
title.fontname=Helvetica
title.fontsize=36
body.fontname=Times Roman
body.fontsize=12

- Another shortcoming of the property file format is caused by the requirement that keys be unique.
menu.item.1=Times Roman
menu.item.2=Helvetica
menu.item.3=Goudy Old Style
- The XML format solves these problems
 - because it can express hierarchical structures
 - and thus is more flexible than the flat table structure of a property file.

```
<configuration>
  <title>
    <font>
      <name>Helvetica</name>
      <size>36</size>
    </font>
  </title>
  <body>
    <font>
      <name>Times Roman</name>
      <size>12</size>
    </font>
  </body>
```

```
<window>
  <width>400</width>
  <height>200</height>
</window>
<color>
  <red>0</red>
  <green>50</green>
  <blue>100</blue>
</color>
<menu>
  <item>Times Roman</item>
  <item>Helvetica</item>
  <item>Goudy Old Style</item>
</menu>
</configuration>
```

- Even though XML and HTML have common roots, there are important differences between the two.
 - Unlike HTML, XML is case sensitive.
 - For example, `<H1>` and `<h1>` are different XML tags
 - In XML, you can **never** omit an end tag. In HTML, you can omit end tags if it is clear from the context where a paragraph or list item ends.
 - such as `</p>` or `` tags
 - In XML, elements that have a single tag without a matching end tag must end in a `/`, as in ``.
 - That way, the parser knows not to look for a `` tag

- Even though XML and HTML have common roots, there are important differences between the two.
 - In XML, attribute values must be enclosed in quotation marks. In HTML, quotation marks are optional.
 - For example, `<applet code="MyApplet.class" width=300 height=300>` is legal HTML but not legal XML.
 - In XML, you have to use quotation marks: `width="300"`
 - In HTML, you can have attribute names without values, such as
 - `<input type="radio" name="language" value="Java" checked>`
 - In XML, all attributes must have values, such as
 - `checked="true"` or `checked="checked"`

- An XML document should start with a header such as
`<?xml version =“1.0”?>`

or

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

Strictly speaking, a header is optional, but it is highly recommended.

- The header can be followed by a document type definition (DTD), such as

`<!DOCTYPE web-app PUBLIC`

`“-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN”`

`“http://java.sun.com/j2ee/dtds/web-app_2_2.dtd”>`

- DTDs are an important mechanism to ensure the correctness of a document, but they are **not required**.

- Finally, the body of the XML document contains the root element, which can contain other elements.
- For example

```
<?xml version="1.0"?>
<!DOCTYPE configuration ...>
<configuration>
  <title>
    <font>
      <name>Helvetica</name>
      <size>36</size>
    </font>
  </title>
  ...
</configuration>
```
- An element can contain child elements, text, or both.

- XML elements can contain attributes, such as
`<size unit="pt">36</size>`
- There is some disagreement among XML designers about when to use elements and when to use attributes.
- For example, it would seem easier to describe a font as
``
than
``
 `<name>Helvetica</name>`
 `<size>36</size>`
``

- However, attributes are much less flexible.
- Suppose you want to add units to the size value. If you use attributes, then you must add the unit to the attribute value:

```
<font name="Helvetica" size="36 pt"/>
```

- Adding an attribute to the size element is much cleaner:

```
<font>
```

```
  <name>Helvetica</name>
```

```
  <size unit="pt">36</size>
```

```
</font>
```

- Elements and text are the "bread and butter" of XML documents.
- Here are a few other markup instructions that you might encounter:
 - Character references have the form `&#decimalValue;` or `&#xhexValue;`.
 - For example, the character `é` can be denoted with either of the following:
 - `é` or `Ù`
 - Entity references have the form `&name;`. The entity references
 - `<`, `>`, `&`, `"`; and `'`;
 - have predefined meanings: the less than, greater than, ampersand, quotation mark, and apostrophe characters.
 - CDATA sections are delimited by `<![CDATA[` and `]]>`. They are a special form of character data.
 - `<![CDATA[< & > are my favorite delimiters]]>`

- Here are a few other markup instructions that you might encounter:
 - Processing instructions are instructions for applications that process XML documents. They are delimited by `<?>` and `?>`, for example,
`<?xml-stylesheet href="mystyle.css" type="text/css"?>`
Every XML document starts with a processing instruction
`<?xml version="1.0"?>`
 - Comments are delimited by `<!-->` and `-->`, for example,
`<!-- This is a comment. -->`
Comments should not contain the string `--`.

- A parser is a program that
 - reads a file,
 - confirms that the file has the correct format,
 - breaks it up into the constituent elements,
 - and lets a programmer access those elements.
- The Java library supplies two kinds of XML parsers:
 - **Tree** parsers such as the Document Object Model (**DOM**) parser that read an XML document into a tree structure.
 - **Streaming** parsers such as the Simple API for XML (**SAX**) parser that generate events as they read an XML document.

- To read an XML document, you need a **DocumentBuilder** object, which you get from a **DocumentBuilderFactory**, like this:

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

- You can now read a document from a file:

```
File f = ...  
Document doc = builder.parse(f);
```

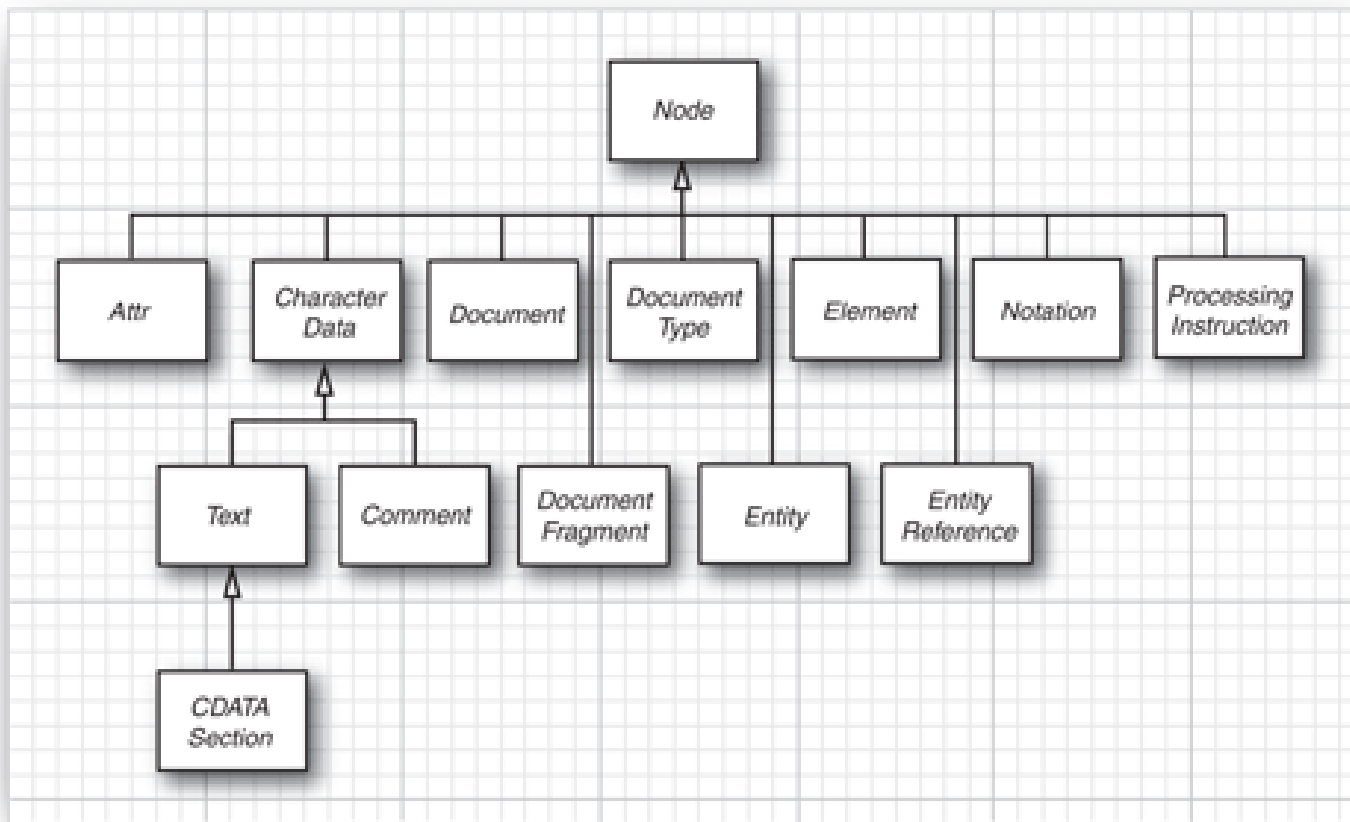
- Alternatively, you can use a URL:

```
URL u = ...  
Document doc = builder.parse(u);
```

- You can even specify an arbitrary input stream:

```
InputStream in = ...  
Document doc = builder.parse(in);
```


- The **Document** object is an in-memory representation of the tree structure of the XML document.



- You start analyzing the contents of a document by calling the **getDocumentElement** method. It returns the root element.

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

- For example, if you are processing a document

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

```
<font>
```

```
...
```

```
</font>
```

then calling **getDocumentElement** returns the font element.

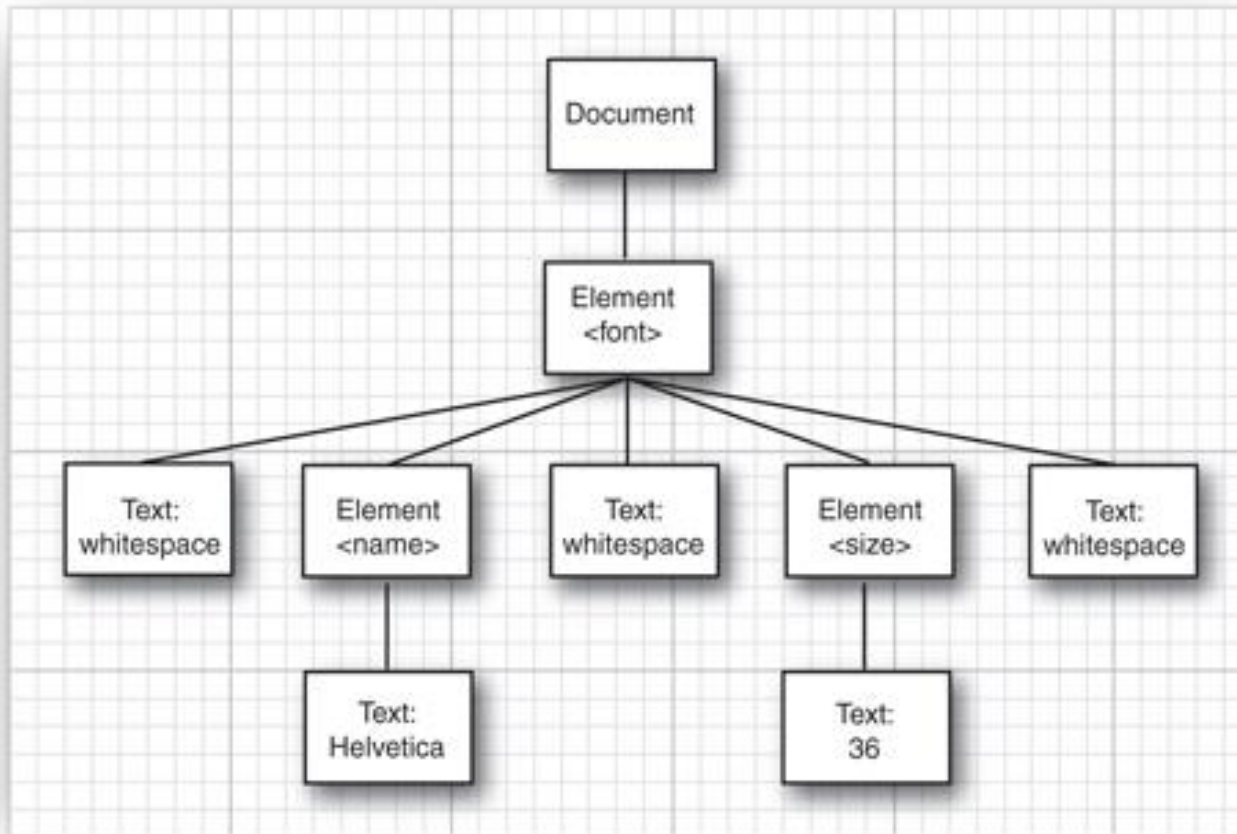
- The **getTagName** method returns the tag name of an element.
 - In the preceding example, **root.getTagName()** returns the string "font".

- To get the element's children, use the **getChildNodes** method.
 - That method returns a collection of type **NodeList**.
 - The **item** method gets the item with a given index,
 - and the **getLength** method gives the total count of the items.

- Therefore, you can enumerate all children like this:

```
NodeList children = root.getChildNodes();
for (int i = 0; i < children.getLength(); i++)
{
    Node child = children.item(i);
    ...
}
```

- Be careful when analyzing the children.
- Suppose, for example, that you are processing the document
``
 `<name>Helvetica</name>`
 `<size>36</size>`
``
- You would expect the font element to have two children, but the parser reports five:
 - The **whitespace** between `` and `<name>`
 - The **name** element
 - The **whitespace** between `</name>` and `<size>`
 - The **size** element
 - The **whitespace** between `</size>` and ``



- If you expect only subelements, then you can ignore the whitespace:

```
for (int i = 0; i < children.getLength(); i++)  
{  
    Node child = children.item(i);  
    if (child instanceof Element)  
    {  
        Element childElement = (Element) child;  
        ...  
    }  
}
```

Now you look at only two elements, with tag names **name** and **size**.

- You can use the `getFirstChild` method without having to traverse another `NodeList`.
- Then use the `getData` method to retrieve the string stored in the `Text` node.

```
for (int i = 0; i < children.getLength(); i++)
{
    Node child = children.item(i);
    if (child instanceof Element)
    {
        Element childElement = (Element) child;
        Text textNode = (Text) childElement.getFirstChild();
        String text = textNode.getData().trim();
        if (childElement.getTagName().equals("name"))
            name = text;
        else if (childElement.getTagName().equals("size"))
            size = Integer.parseInt(text);
    }
}
```

- You can also get the last child with the `getLastChild` method, and the next sibling of a node with `getNextSibling`.
- Therefore, another way of traversing a set of child nodes is

```
for (Node childNode = element.getFirstChild();  
     childNode != null;  
     childNode = childNode.getNextSibling())  
{  
    ...  
}
```


- To enumerate the attributes of a node, call the **getAttributes** method.
 - It returns a **NamedNodeMap** object that contains Node objects describing the attributes.

```
NamedNodeMap attributes = element.getAttributes();
for (int i = 0; i < attributes.getLength(); i++)
{
    Node attribute = attributes.item(i);
    String name = attribute.getNodeName();
    String value = attribute.getNodeValue();
    ...
}
```

- Alternatively, if you know the name of an attribute, you can retrieve the corresponding value directly:
`String unit = element.getAttribute("unit");`

- The SAX parser reports events as it parses the components of the XML input, but it does not store the document in any way
 - it is up to the event handlers whether they want to build a data structure.
 - In fact, the DOM parser is built on top of the SAX parser. It builds the DOM tree as it receives the parser events.
- The **ContentHandler** interface defines several callback methods that the parser executes as it parses the document. Here are the most important ones:
 - **startElement** and **endElement** are called each time a start tag or end tag is encountered.
 - **characters** is called whenever character data are encountered.
 - **startDocument** and **endDocument** are called once each, at the start and the end of the document.

- For example, when parsing the fragment

```
<font>
  <name>Helvetica</name>
  <size units="pt">36</size>
</font>
```
- The parser makes the following callbacks:
 - startElement, element name: font
 - startElement, element name: name
 - characters, content: Helvetica
 - endElement, element name: name
 - startElement, element name: size, attributes: units="pt"
 - characters, content: 36
 - endElement, element name: size
 - endElement, element name: font

- Here is how you get a SAX parser:

```
SAXParserFactory factory = SAXParserFactory.newInstance();  
SAXParser parser = factory.newSAXParser();
```
- You can now process a document:

```
parser.parse(source, handler);
```
- Here,
 - **source** can be a file, URL string, or input stream.
 - **handler** belongs to a subclass of **DefaultHandler**.
- The **DefaultHandler** class defines do-nothing methods for the four interfaces:
 - ContentHandler
 - DTDHandler
 - EntityResolver
 - ErrorHandler

```
DefaultHandler handler = new
DefaultHandler()
{
    public void startElement(String namespaceURI, String lname,
                            String qname, Attributes attrs)
        throws SAXException
    {
        if (lname.equalsIgnoreCase("a") && attrs != null)
        {
            for (int i = 0; i < attrs.getLength(); i++)
            {
                String aname = attrs.getLocalName(i);
                if (aname.equalsIgnoreCase("href"))
                    System.out.println(attrs.getValue(i));
            }
        }
    }
};
```

- The StAX parser is a "pull parser." Instead of installing an event handler, you simply iterate through the events, using this basic loop:

```
InputStream in = url.openStream();
XMLInputFactory factory = XMLInputFactory.newInstance();
XMLStreamReader parser = factory.createXMLStreamReader(in);
while (parser.hasNext())
{
    int event = parser.next();
    Call parser methods to obtain event details
}
```

- For example, when parsing the fragment

```
<font>
  <name>Helvetica</name>
  <size units="pt">36</size>
</font>
```

- The parser yields the following events:
 - START_ELEMENT, element name: font
 - CHARACTERS, content: white space
 - START_ELEMENT, element name: name
 - CHARACTERS, content: Helvetica
 - END_ELEMENT, element name: name
 - CHARACTERS, content: white space
 - START_ELEMENT, element name: size
 - CHARACTERS, content: 36
 - END_ELEMENT, element name: size
 - CHARACTERS, content: white space
 - END_ELEMENT, element name: font
- To analyze the attribute values, call the appropriate methods of the **XMLStreamReader** class. For example,
`String units = parser.getAttributeValue(null, "units");`
gets the units attribute of the current element.

- You also need to check whether the document contains the nodes that you expect.
- To specify the document structure
 - you can supply a **DTD** or an **XML Schema** definition.
 - A DTD or schema contains rules that explain how a document should be formed, by specifying the legal child elements and attributes for each element.

- For example, a DTD might contain a rule:

```
<!ELEMENT font (name,size)>
```

This rule expresses that a font element must always have two children, which are name and size elements.

- The XML Schema language expresses the same constraint as

```
<xsd:element name="font">
```

```
  <xsd:sequence>
```

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

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

```
  </xsd:sequence>
```

```
</xsd:element>
```

- There are several methods for supplying a DTD. You can include a DTD in an XML document like this:

```
<?xml version="1.0"?>  
<!DOCTYPE configuration [  
  <!ELEMENT configuration ...>  
  more rules  
  ...  
<configuration>  
  ...  
</configuration>
```

- Supplying a DTD inside an XML document is somewhat uncommon because DTDs can grow lengthy.
- It makes more sense to store the DTD externally. The **SYSTEM** declaration can be used for that purpose.
- You specify a URL that contains the DTD, for example:
`<!DOCTYPE configuration SYSTEM "config.dtd">`
or
`<!DOCTYPE configuration SYSTEM "http://myserver.com/config.dtd">`

- Rules for Element Content

Rule	Meaning
E^*	0 or more occurrences of E
E^+	1 or more occurrences of E
$E?$	0 or 1 occurrences of E
$E_1 E_2 \dots E_n$	One of E_1, E_2, \dots, E_n
E_1, E_2, \dots, E_n	E_1 followed by E_2, \dots, E_n
#PCDATA	Text
$(\#PCDATA E_1 E_2 \dots E_n)^*$	0 or more occurrences of text and E_1, E_2, \dots, E_n in any order (mixed content)
ANY	Any children allowed
EMPTY	No children allowed

- You also specify rules to describe the legal attributes of elements.
- The general syntax is
<!ATTLIST element attribute type default>

- Attribute Types and Defaults

Type	Meaning
CDATA	Any character string
(A ₁ A ₂ ... A _n)	One of the string attributes A ₁ A ₂ ... A _n
NMTOKEN, NMTOKENS	One or more name tokens
ID	A unique ID
IDREF, IDREFS	One or more references to a unique ID
ENTITY, ENTITIES	One or more unparsed entities

Default	Meaning
#REQUIRED	Attribute is required.
#IMPLIED	Attribute is optional.
A	Attribute is optional; the parser reports it to be A if it is not specified.
#FIXED A	The attribute must either be unspecified or A; in either case, the parser reports it to be A.

<!ELEMENT font (name,size)>

<!ELEMENT name (#PCDATA)>

<!ELEMENT size (#PCDATA)>

<!ELEMENT chapter
(intro,(heading,(para|image|table|note)+)+)>

<!ELEMENT para (#PCDATA|em|strong|code)*>

<!ATTLIST font style (plain|bold|italic|bold-italic) "plain">

<!ATTLIST size unit CDATA #IMPLIED>

- To reference a Schema file in a document, add attributes to the root element, for example:

```
<?xml version="1.0"?>  
<configuration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
    xsi:noNamespaceSchemaLocation="config.xsd">  
...  
</configuration>
```

- For example, here is an enumerated type:

```
<xsd:simpleType name="StyleType">  
    <xsd:restriction base="xsd:string">  
        <xsd:enumeration value="PLAIN" />  
        <xsd:enumeration value="BOLD" />  
        <xsd:enumeration value="ITALIC" />  
        <xsd:enumeration value="BOLD_ITALIC" />  
    </xsd:restriction>  
</xsd:simpleType>
```


- You can compose types into complex types, for example:

```
<xsd:complexType name="FontType">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="size" type="xsd:int"/>
    <xsd:element name="style" type="StyleType">
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:enumeration value="PLAIN" />
          <xsd:enumeration value="BOLD" />
          <xsd:enumeration value="ITALIC" />
          <xsd:enumeration value="BOLD_ITALIC" />
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>
```

- The XPath language makes it simple to access tree nodes.
- For example, suppose you have this XML document:

```
<configuration>
```

```
...
```

```
<database>
```

```
<username>dbuser</username>
```

```
<password>secret</password>
```

```
...
```

```
</database>
```

```
</configuration>
```

- You can get the database user name by evaluating the XPath expression

```
/configuration/database/username
```

- That's a lot simpler than the plain DOM approach:
 - Get the document node.
 - Enumerate its children.
 - Locate the database element.
 - Get its first child, the username element.
 - Get its first child, a Text node.
 - Get its data.

 - `/gridbag/row`
 - `/gridbag/row[1]`
 - `/gridbag/row[1]/cell[1]/@anchor`
 - `/gridbag/row/cell/@anchor`
 - `count(/gridbag/row)`

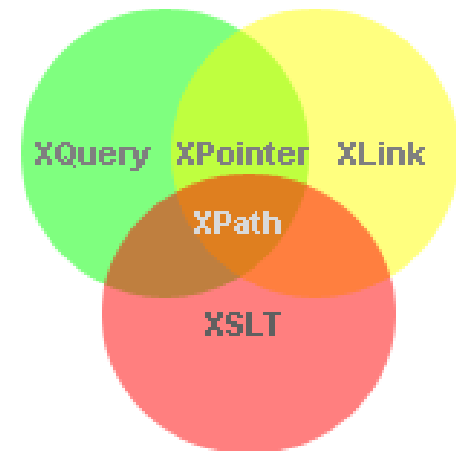
- Java SE 5.0 added an API to evaluate XPath expressions. You first create an XPath object from an **XPathFactory**:

```
XPathFactory xpfactory = XPathFactory.newInstance();  
path = xpfactory.newXPath();
```

- You then call the **evaluate** method to evaluate XPath expressions:

```
String username =  
path.evaluate("/configuration/database/username", doc);  
NodeList nodes = (NodeList) path.evaluate("/gridbag/row", doc,  
XPathConstants.NODESET);  
Node node = (Node) path.evaluate("/gridbag/row[1]", doc,  
XPathConstants.NODE);  
int count = ((Number) path.evaluate("count(/gridbag/row)", doc,  
XPathConstants.NUMBER)).intValue();  
  
result = path.evaluate(expression, node);
```

- XQuery is to XML what SQL is to database tables.
 - XQuery is designed to query XML data - not just XML files, but anything that can appear as XML, including databases.
 - XQuery is *the* language for querying XML data
 - XQuery for XML is like SQL for databases
 - XQuery is built on XPath expressions
 - XQuery is supported by all major databases
 - XQuery is a W3C Recommendation



- XQuery can be used to:
 - Extract information to use in a Web Service
 - Generate summary reports
 - Transform XML data to XHTML
 - Search Web documents for relevant information
- XQuery and XPath
 - XQuery 1.0 and XPath 2.0 share the same data model and support the same functions and operators.
 - If you have already studied XPath you will have no problems with understanding XQuery.

- Book.xml

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

```
<bookstore>
```

```
<book category="COOKING">
```

```
<title lang="en">Everyday Italian</title>
```

```
<author>Giada De Laurentiis</author>
```

```
<year>2005</year>
```

```
<price>30.00</price>
```

```
</book>
```

```
<book category="CHILDREN">
```

```
<title lang="en">Harry Potter</title>
```

```
<author>J K. Rowling</author>
```

```
<year>2005</year>
```

```
<price>29.99</price>
```

```
</book>
```

```
<book category="WEB">
```

```
<title lang="en">XQuery Kick Start</title>
```

```
<author>James McGovern</author>
```

```
<author>Per Bothner</author>
```

```
<author>Kurt Cagle</author>
```

```
<author>James Linn</author>
```

```
<author>Vaidyanathan Nagarajan</author>
```

```
<year>2003</year>
```

```
<price>49.99</price>
```

```
</book>
```

```
<book category="WEB">
```

```
<title lang="en">Learning XML</title>
```

```
<author>Erik T. Ray</author>
```

```
<year>2003</year>
```

```
<price>39.95</price>
```

```
</book>
```

```
</bookstore>
```

- **Functions**

- XQuery uses functions to extract data from XML documents.
- The `doc()` function is used to open the "books.xml" file:
`doc("books.xml")`

- **Path Expressions**

- XQuery uses path expressions to navigate through elements in an XML document.

`doc("books.xml")/bookstore/book/title`

- The XQuery above will extract the following:

`<title lang="en">Everyday Italian</title>`

`<title lang="en">Harry Potter</title>`

`<title lang="en">XQuery Kick Start</title>`

`<title lang="en">Learning XML</title>`

- Predicates

- XQuery uses predicates to limit the extracted data from XML documents.

- ```
doc("books.xml")/bookstore/book[price<30]
```

- The XQuery above will extract the following:

- ```
<book category="CHILDREN">  
  <title lang="en">Harry Potter</title>  
  <author>J K. Rowling</author>  
  <year>2005</year>  
  <price>29.99</price>  
</book>
```

- A namespace is identified by a Uniform Resource Identifier (URI), such as
<http://www.w3.org/2001/XMLSchema>
<http://www.uuid.org/>
`uuid:1c759aed-b748-475c-ab68-10679700c4f2`
`urn:com:books-r-us`
- Why use HTTP URLs for namespace identifiers? It is easy to ensure that they are unique.

- To specify the long names of namespace:
`<element xmlns="namespaceURI">`
 children
`</element>`
- A child can provide its own namespace, for example:
`<element xmlns="namespaceURI1">`
 `<child xmlns="namespaceURI2">`
 grandchildren
 `</child>`
 more children
`</element>`

- A better approach is
 - to build up a DOM tree with the contents of the document
 - and then write out the tree contents.

```
Document doc = builder.newDocument();
```

- Use the **createElement** method of the **Document** class to construct the elements of your document.

```
Element rootElement = doc.createElement(rootName);
```

```
Element childElement = doc.createElement(childName);
```

- Use the **createTextNode** method to construct text nodes:

```
Text textNode = doc.createTextNode(textContents);
```

- Add the **root** element to the document, and add the child nodes to their parents:

```
doc.appendChild(rootElement);  
rootElement.appendChild(childElement);  
childElement.appendChild(textNode);
```

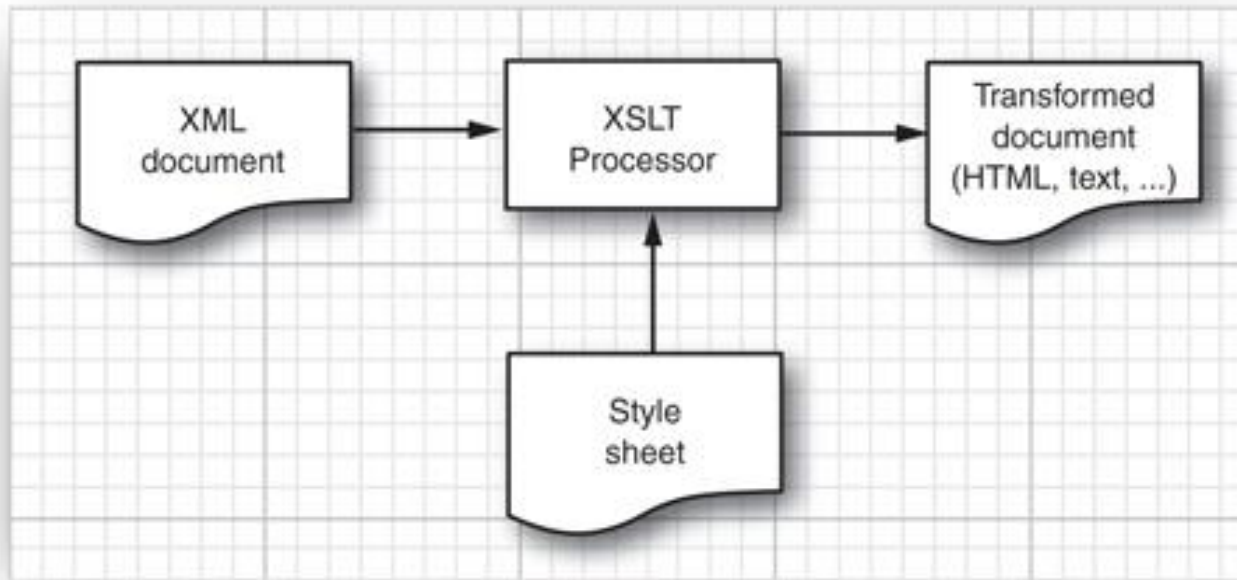
- As you build up the DOM tree, you may also need to set element attributes. Simply call the **setAttribute** method of the Element class:

```
rootElement.setAttribute(name, value);
```

- Somewhat curiously, the DOM API currently has no support for writing a DOM tree to an output stream.
- To overcome this limitation, we use the Extensible Stylesheet Language Transformations (XSLT) API.

```
Transformer t = TransformerFactory.newInstance().newTransformer();  
  
t.setOutputProperty(OutputKeys.DOCTYPE_SYSTEM, systemIdentifier);  
t.setOutputProperty(OutputKeys.DOCTYPE_PUBLIC, publicIdentifier);  
t.setOutputProperty(OutputKeys.INDENT, "yes");  
t.setOutputProperty(OutputKeys.METHOD, "xml");  
t.setOutputProperty("{http://xml.apache.org/xslt}indent-amount", "2");  
  
t.transform(new DOMSource(doc),  
           new StreamResult(new FileOutputStream(file)));
```

- The XSL Transformations (XSLT) mechanism allows you to
 - specify rules for transforming XML documents into other formats,
 - such as plain text, XHTML, or any other XML format.



- Here is a typical example.

```
<staff>
  <employee>
    <name>Carl Cracker</name>
    <salary>75000</salary>
    <hiredate year="1987" month="12" day="15"/>
  </employee>
  <employee>
    <name>Harry Hacker</name>
    <salary>50000</salary>
    <hiredate year="1989" month="10" day="1"/>
  </employee>
  <employee>
    <name>Tony Tester</name>
    <salary>40000</salary>
    <hiredate year="1990" month="3" day="15"/>
  </employee>
</staff>
```


- The desired output is an HTML table:

```
<table border="1">
<tr>
<td>Carl Cracker</td><td>$75000.0</td><td>1987-12-15</td>
</tr>
<tr>
<td>Harry Hacker</td><td>$50000.0</td><td>1989-10-1</td>
</tr>
<tr>
<td>Tony Tester</td><td>$40000.0</td><td>1990-3-15</td>
</tr>
</table>
```

- A style sheet with transformation templates has this form:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">
  <xsl:output method="html" />
  template1
  template2
  ...
</xsl:stylesheet>
```

- Here is a typical template for **employee** nodes:

```
<xsl:template match="/staff/employee">
  <tr><xsl:apply-templates/></tr>
</xsl:template>
```

- Here is a template for **name** nodes:

```
<xsl:template match="/staff/employee/name">  
  <td><xsl:apply-templates/></td>  
</xsl:template>
```

- Here is an example for **hiredate** nodes. :

```
<xsl:template match="/staff/employee/hiredate">  
  <td><xsl:value-of select="@year"/>-<xsl:value-of  
  select="@month"/>-<xsl:value-of select="@day"/></td>  
</xsl:template>
```

- It is extremely simple to generate XSL transformations in the Java platform.

```
File styleSheet = new File(filename);
```

```
StreamSource styleSource = new StreamSource(styleSheet);
```

```
Transformer t = TransformerFactory.newInstance().
```

```
    newTransformer(styleSource);
```

```
t.transform(source, result);
```

- Start and Shutdown Sedna
 - To start Sedna server go to INSTALL_DIR/bin and run:
 - `se_gov`
 - To shutdown Sedna server run:
 - `se_stop`
- Create and Run a Database
 - To create a database named testdb:
 - `se_cdb testdb`
 - To run the testdb database:
 - `se_sm testdb`
 - To shutdown the testdb database:
 - `se_smsd testdb`

- Access with Java API

```
import javax.xml.xquery.*;
import javax.xml.namespace.QName;
import net.xqj.sedna.SednaXQDataSource;
public class QuickStart {
    public static void main(String[] args) throws XQException {
        XQDataSource xqs = new SednaXQDataSource();
        xqs.setProperty("serverName", "localhost");
        xqs.setProperty("databaseName", "test");
        XQConnection conn = xqs.getConnection("SYSTEM", "MANAGER");
        XQPreparedExpression xqpe =
            conn.prepareExpression("declare variable $x as xs:string external; $x");
        xqpe.bindString(new QName("x"), "Hello World!", null);
        XQResultSequence rs = xqpe.executeQuery();
        while(rs.next())
            System.out.println(rs.getItemAsString(null));
        conn.close();
    }
}
```

- Core Java™ Volume II–Advanced Features, Eighth Edition
 - by Cay S. Horstmann; Gary Cornell
 - Publisher: Prentice Hall
 - Print ISBN-13: 978-0-13-235479-0
- Sedna XML Database
 - <http://www.sedna.org/>



Thank You!