31284 Web Services Development
32525 Web services technologies
31242/32549/AJPCPE Advanced Internet Programming

Introduction to XML

V7 March 2010
Agenda

• **XML Introduction**

• XML Namespaces

• DTD/XML Schema

• XML programming

• XSLT/XPATH

next weeks...
XML Introduction

• Meta-language (literally a language about languages)

• XML supports user-defined languages that add meaning to data
  – Human readable
  – Extensible & Vendor-independent
  – Widespread Industry Support
  – Form the basis for Web Services
  – Buzzword-compliant!
Key Uses Of XML

- **“Document centric”**
  - represents semi-structured information
  - eg: markup language - XHTML
  - eg: Document publishing (separate presentation and content)

- **“Data centric”**
  - represent highly structured information
  - eg: Data storage
  - eg: Data exchange
  - eg: Application configuration
Document centric XML

```xml
<?xml version="1.0" encoding="UTF-8"?>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>hello, World</title>
</head>
<body>
<h1>Hello, World</h1>
<p>This is a sample of XHTML, geez it looks just like HTML?</p>
</body>
</html>
```
<?xml version="1.0" encoding="UTF-8"?>
<addressBook>
  <entry list="personal">
    <name>Ms Smith</name>
    <address>1 Central Rd, Sydney</address>
    <phone>555 5555</phone>
  </entry>
  <entry list="business">
    <name>Mr Suit</name>
    <address>1 George St, Sydney</address>
    <phone>555 6666</phone>
  </entry>
</addressBook>
XML - Introduction

• XML stands for Extensible Markup Language

• XML is a tag based markup language, which appears superficially similar to other markup languages, such as HTML

• Unlike HTML, which has a fixed set of tags defined that you can use, XML allows you to use whatever tags you like – it is extensible
XML Technologies

XML Document

XML Document Object Model (DOM)

XML Streaming interface e.g. SAX

in memory

XPath (address parts of XML doc)

XLink XML doc linking

XQuery XML Query

XML Representations

XML Document file - Unicode

XSL: XML Transformation (uses XPath)

Text XML

Bits of XML doc

XML

HTML

XML

XML Schema Validate XML

(c) University of Technology, Sydney
What is XML?

• XML documents are plain text documents

• The basic rules of XML are quite simple.
  – Any document which obeys the basic rules of XML is said to be “well formed”

• A “well formed” XML document obeys rules of XML
  – every tag closed
  – no tags nested out of order
  – syntactically correct according to XML specification
    (look at http://www.w3.org/XML)
XML – Document Structure

- An XML document is a structured collection of text format markup tags

- Each tag either defines:
  - some information used to describe how the document is to be interpreted
  - some data contained within the document

- Tags in XML documents fall into the following categories:
  
  - **(1) Comments:**
    - XML, like Java and HTML, has a means of adding comments to a document
    
    <!-- comments in XML look like this -->
(2) XML declaration:
- Identifies the document as an XML document. Includes XML version (currently 1.0), encoding format, dependencies
- Must be at top of the XML document

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

(3) DTD declarations:
- DTD (Document Type Definition) specifies the semantic constraints

```xml
<!DOCTYPE addressBook SYSTEM "D:\projects\examples\xml\addressBook.dtd"
```
– **(4) Elements:**
  - Contain data values or other XML elements. Elements are defined between “<” and “>” characters
    
    ```xml
    <entry>
      <name>Mr Smith</name>
    </entry>
    ```

– **(5) Attributes:**
  - Attributes are name/value pairs associated with elements
    
    ```xml
    <entry list="personal">
    ```
- **(5) Entity references:**
  - Special identifiers that refer to a value
  - Defined between "&" and ";" characters
  - Some standard reserved entities
    - eg: `&lt;` represents "<" character
  - You can make your own eg: `&!ENTITY au Australia;`
  - Eg:
    - `<country>&au;</country>`
    - is the same as
    - `<country>Australia</country>`
(6) Processing instructions:
- Tell applications to perform some operation.
- Defined between "<?" and "?>" sequence
  
  ```xml
  <?OrderValidation creditCheck checkCustomer?>
  ```

(7) Unparsed character data:
- Tells XML processor not to parse quoted string
- Defined between `<![CDATA[` and `]]>` sequences
  
  ```xml
  <![CDATA[if (x > y) or (y < z)]]>
  ```
Agenda

• XML Introduction
• XML Namespaces
• DTD/XML Schema
• XML programming
• XSLT/XPATH
XML – Namespaces

- XML elements can belong to an XML Namespace
- XML Namespaces help avoid **clashes** in element names when documents of different types are combined

```java
class Author {
    String name;
    String title;
}

class Book {
    Author author;
    String title;
}
```

Solution: Qualify the name!

- Book’s title is **book**.title
- Author’s title is: **book**.author.title

Much like java packages!!
XML Namespaces

- Namespaces identify that elements in the document come from different sources
  - you can mix different namespaces in the same document
  - namespace appears as prefix on element/attribute

- At the top of each document, you need to declare the namespaces used by that document

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

tag is now `<xs:schema>`
XML – Namespaces

• Elements in an XML document can be marked with a namespace.
  – The namespace is a prefix to the element name – the format for the element tag is:
    
    ```xml
    <namespace : elementName>
    data
    </namespace : elementName>
    ```

• Namespaces are first declared at the root element containing the elements belonging to the name space.

• eg:
  ```xml
  <Address:addressBook
    xmlns:Address="http://www.xyz.com/addressBook”>
  ```
Example – XML Namespace

<?xml version="1.0" encoding="UTF-8"?>
<Address:addressBook
  xmlns:Address="http://www.zyx.com/addressBook">
<Address:addressBook>
  <Address:entry list="personal">
    <Address:name>Ms Smith</Address:name>
    <Address:address>1 Central Rd, Sydney</Address:address>
    <Address:phone>555 5555</Address:phone>
  </Address:entry>
  <Address:entry list="business">
    <Address:name>Mr Suit</Address:name>
    <Address:address>1 George St, Sydney</Address:address>
    <Address:phone>555 6666</Address:phone>
  </Address:entry>
</Address:addressBook>
Trees and XML

• A common way of processing XML documents is to read them into memory in a tree structure
  – arbitrary tree, not binary

• One way to process the in-memory XML document is to use a traversal algorithm

• While XML is good for tree-structured information, it is not so useful for representing the more general concept of graphs ...
XML and trees

- XML represents a tree structure
Trees - terminology

- Parent
- Child
- Siblings
- Ancestors
- Descendants

- Leaf
- Internal vertices
- Subtree

addressBook

entry

name address phone

entry

name address phone
• An abstract syntax representation of an XML document

```
<Document>
  Name: #document
  <Element>
    Name: addressBook
      <Attribute>
        Name: list
        Value: "personal"
      </Attribute>
      <Element>
        Name: entry
          <Element>
            Name: name
              <Text>
                Name: #text
                Value: "Ms Smith"
              </Text>
          </Element>
          <Element>
            Name: address
              <Text>
                Name: #text
                Value: "1 Central..."
              </Text>
          </Element>
          <Element>
            Name: phone
              <Text>
                Name: #text
                Value: "555 5555"
              </Text>
          </Element>
```

Tree Traversals

- Obviously when using XML, we need to systematically visit each vertex of a tree
- Three main ways to traverse:
  - preorder
  - inorder
  - postorder

- Which traversal to use depends on the trade-off of efficiency versus speed.

- We’ll talk about the different programming models later...
Preorder Traversal

1. Visit the root node \( r \).
2. Recursively visit each subtree according to their preorder sequence.

Nodes visited in preorder: \( a, b, e, j, k, n, o, p, f, c, d, g, l, m, h, i \)
Inorder Traversal

Inorder traversal

Step 1: Visit $T_1$ in inorder
Step 2: Visit $r$
Step 3: Visit $T_2$ in inorder
Step $n + 1$: Visit $T_n$ in inorder

j, e, n, k, o, p, b, f, a, c, l, g, m, d, h, i
Postorder Traversal

Step 1: Visit $T_1$ in postorder
Step 2: Visit $T_2$ in postorder
Step $n$: Visit $T_n$ in postorder

Postorder traversal

j, n, o, p, k, e, f, b, c, l, m, g, h, i, d, a
Exercise
Motivation

• XML is a highly structured language
• As a human, it's easy for you to see the structure:
  – Elements, Attributes, Entity references, Comments, Processing instructions, CDATA sections

• But to be any real use, XML has to be read by a computer application
  – not just read it as a single chunk of ASCII text
  – need to *parse* XML document to recognise the structure
Agenda

- XML Introduction
- XML Namespaces
- ➔ DTD/XML Schema
- XML programming
- XSLT/XPATH
**XML document structure**

- Often you need to add a *grammar* (Rules) to structure a XML document
- Want to define allowed tags, elements, attributes and their order and data types
- Use “*Document Type Definitions*” (DTD) or an XML Schema
- This allows validation of XML documents to check its conformity to the grammar expressed by the schema
Well-formed vs. Valid

- **Well-formed:**
  - XML document follows the basic syntax rules of XML

- **Valid** (or more precisely, schema-valid):
  - XML document follows the basic syntax rules of XML and also follows the rules in its associated DTD or XML Schema
DTDs

- DTD specifies the format
  (e.g. each book can have many authors, but only one publisher)

but
- Don’t have support for data-types (everything is text)
- Don’t look like XML
- Don’t have good support for namespaces
Comment: Ancient History?

- Why doesn’t DTD look like XML?
- Answer: *XML was derived from SGML (designed in the 1960’s to represent structured documents for publishing)*
- DTD’s are still commonly used!!!

- Roughly, DTDs are old-style XML Schema (in fact, XML Schema is quite new, so DTDs aren't such ancient history)
DTDs Example

<!– fax1_0.dtd file -->
<!ELEMENT fax (to, from, faxnumber, heading?, body?)>
<!ELEMENT to (#PCDATA) #REQUIRED>
<!ELEMENT from (#PCDATA) #REQUIRED>
<!ELEMENT faxnumber (#PCDATA) #REQUIRED>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)>

- #PCDATA means any PC character string
- #REQUIRED means a mandatory element
<?xml version="1.0"?>
<!DOCTYPE web-app PUBLIC "-//UTS//note DTD 1.0//EN" "http://learn.it.uts.edu.au/dsp/fax1_0.dtd">
<fax>
  <to>Robert</to>
  <from>Chris</from>
  <faxnumber>9514-9999</faxnumber>
  <heading>32525 DSP</heading>
  <body>Give everyone HD</body>
</fax>

• Note: You can also use
  <!DOCTYPE note SYSTEM "note.dtd"> for external DTD files
XML Schema

• XML Schema defines a sub-language of XML
  – ie: Like a DTD, adds more control over the syntax of your XML document

• Often represented as a separate XML document itself (usually with an extension of .xsd)
XML – XML Schema (cont)

• The schema definition itself allows the structure of elements and attributes to be defined, along with any constraints on these elements and attributes.

• An element definition looks like:
  `<element name="elementName"
     type="elementType"
     minOccurs="min times allowed to occur"
     maxOccurs="max times allowed to occur" />

• The XML schema specification defines some standard elementTypes.
• ➤ You can also define your own element types.
XML – XML Schema (cont)

• A complex type definition is of the following form:
  
  ```xml
  <complexType name="typeName">
    <![element definition>]
    <![element definition>]
  </complexType>
  ```

• An attribute definition is of the following form:
  
  ```xml
  <attribute name="attributeName"
    type="attributeType"
    [attribute options]>
  ```

• For further details about the full range of options available, refer to the XML Schema specification
Sample XML Schema

fax.xsd

```xml
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://learn.it.uts.edu.au/dsp"
    xmlns="http://learn.it.uts.edu.au/dsp"
    elementFormDefault="qualified">
  <xsd:element name="fax">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="from" type="xsd:string"/>
        <xsd:element name="to" type="xsd:string"/>
        <xsd:element name="faxnumber" type="xsd:string"/>
        <xsd:element name="heading" type="xsd:string"/>
        <xsd:element name="body" type="xsd:string"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

fax.xsd
**XML Schema data types**

- XML Schemas have predefined data types eg:

<table>
<thead>
<tr>
<th>Type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>boolean</td>
</tr>
<tr>
<td>integer, positiveInteger,</td>
<td>time, date, date-time,</td>
</tr>
<tr>
<td>negativeInteger</td>
<td>duration</td>
</tr>
<tr>
<td>base64Binary, hexBinary</td>
<td>Name</td>
</tr>
<tr>
<td>decimal</td>
<td>QName</td>
</tr>
<tr>
<td>anyURI</td>
<td>ID, IDREF</td>
</tr>
</tbody>
</table>
<?xml version="1.0"?>
<fax
    xmlns="http://learn.it.uts.edu.au/dsp"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://learn.it.uts.edu.au/dsp fax.xsd">
    <from>Wayne</from>  <to>Andrew</to>
    <faxnumber>9514-1807</faxnumber>
    <heading>Reminder</heading>
    <body>Don't forget the lecture!</body>
</fax>
Using XML Schema (2)

<?xml version="1.0"?>
<fax
    xmlns="http://learn.it.uts.edu.au/dsp"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://learn.it.uts.edu.au/dsp fax.xsd">
    
    <from>f3rj934#$%R</from>  <to>%TR$E#D</to>
    <faxnumber>go figure</faxnumber>
    <heading>Reminder</heading>
    <body>Don't forget the lecture!</body>

</fax>
XML Schemas & validity

- Example shown allows *string* elements.
  - But this doesn’t necessarily make sense
    - *eg:* faxnumber = “go figure”

- We can define our own data types that make sense to your schema
  - `simpleType`
    - derivative of base predefined data types
  - `complexType`
    - compound data type, like Javabean or C Structs
Defining custom data types

• Use regular expressions (patterns):

```
<xsd:simpleType name="nameType">
    <xsd:restriction base="xsd:string">
        <xsd:pattern value="[A-Z][a-z]*"/>
    </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="telnoType">
    <xsd:restriction base="xsd:string">
        <xsd:pattern value="[0-9]{4}-[0-9]{4}"/>
    </xsd:restriction>
</xsd:simpleType>
```

Chris

9514-4525
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://learn.it.uts.edu.au/dca"
    xmlns="http://learn.it.uts.edu.au/dca"
    elementFormDefault="qualified">
    <!-- Insert custom type definitions here -->
    <xsd:element name="fax">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="from" type="nameType"/>
                <xsd:element name="to" type="nameType"/>
                <xsd:element name="faxnumber" type="telnoType"/>
                <xsd:element name="heading" type="xsd:string"/>
                <xsd:element name="body" type="xsd:string"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
</xsd:schema>
Comment: regular expressions

- Most string pattern matching in file managers is based on regular expressions, e.g.
  - "*.txt" means "all files ending with .txt"

- String pattern matching in some text search tools are based on regular expressions, e.g. Unix grep, search-and-replace in Microsoft Word

- Syntax:
  - *string* matches the exact string “string”
  - [ABC] matches A or B or C only
  - [A-Z] matches A, B, C ... Z
  - . Matches any character
  - * means zero or more of the *preceding* pattern
  - {2} means exactly 2 of the *preceding* pattern
DTDs versus XML Schema

• DTDs are not as expressive as XML Schema

XML Schema allows a language designer to express more constraints than DTDs allow, e.g.

– "data types" associated with CDATA (xs:string, xs:integer, etc.)
– user-defined data types (simple types, complex types)
– regular expressions over CDATA (DTDs permit regular expressions over sequencing of elements, but not over CDATA content)
Agenda

- XML Introduction
- XML Namespaces
- DTD/XML Schema
- XML programming
- XSLT/XPATH

Next week...
XML Programming

• One of the benefits often stated for XML is that it is a language that is both:
  – human readable (more or less)
  – machine parseable

• XML is a context-free language

• Prior to XML, programmers would invent their own custom data representation languages
  – e.g. for saving info in configuration files
  – programmer must write a parser for their language
  – a benefit of XML is that the parsers already exist
XML parsers

- There are 3 three fundamental parser types.

1. **Document model** parser (**DOM**)
   - Reads entire document into data structure in memory.
   - Uses significantly more memory
   - Convenient as whole document can be accessed as data structure

2. **Push** parser (**SAX**)
   - Event driven.
   - As it parses document, notifies a listener object.
   - Not intuitive programming, cannot navigate document

3. **Pull** parser (**sTax/XMLPull**)
   - Reads a little bit of a document at once.
   - Your application drives the parser by requesting the next piece.
XML parsers

• These are **language-independent** API specs
  – There are DOM & SAX implementations for virtually every language e.g. Java, Javascript, .NET etc
  – E.g: Javascript DOM parser built into all modern browsers
    → how we get WEB 2.0 applications using AJAX (Asynchronous Javascript and XML)

• The Java API is called **JAXP** – Java API for XML Processing
  – javax.xml.* packages
• JAX Pax (Java XML Pack) is an all-in-one download of Java technologies for XML:
  – JAXP – Java API for Parsers
  – JAXM – Java API for Messaging
  – JAXB – Java API for Binding
  – JAXR – Java API for Registries
  – JAX-RPC Java API for XML RPC
  – JAX-WS – Java API for Web Services
JAXP

- APIs for accessing XML documents
  - **SAX**: Lightweight, event driven parser
    - "when you see element x do action y"
  - **DOM**: parses entire document into an object tree in memory

![Diagram of XML Document parsing with JAXP](image-url)
Document Object Model (DOM)

• DOM treats XML document as a **tree**

• Every tree **node** contains one of the components from XML structure (element node, text node, attribute node etc.)

• Parser creates a (tree) data structure in **memory**

• We get what we need by **traversing** the tree

• Allows to **create** and **modify** XML documents
DOM Tree

XML Document

<?xml version="1.0"?>

<addressbook>
  <person>
    <name>Nazmul Idris</name>
    <email>xml@java-xml.com</email>
  </person>

  <person>
    <name>John Doe</name>
    <email>john@doe.com</email>
  </person>

</addressbook>

Document object tree

document
  addressbook
    person
      name="Nazmul Idris"
      email="xml@java-xml.com"
    person
      name="John Doe"
      email="john@doe.com"
Sample DOM tree

〈Document〉
Name: #document

〈Element〉
Name: addressBook

〈Attribute〉
Name: list
Value: "personal"

〈Element〉
Name: entry

〈Element〉
Name: name

〈Text〉
Name: #text
Value: "Ms Smith"

〈Element〉
Name: address

〈Element〉
Name: phone

〈Text〉
Name: #text
Value: "1 Central..."

〈Text〉
Name: #text
Value: "555 5555"
Java DOM

- Use `DocumentBuilderFactory` to create a DOM parser
  ```java
  DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
  ```

- Use `DocumentBuilder` to read the XML document
  ```java
  DocumentBuilder builder = factory.newDocumentBuilder();
  Document document = builder.parse(...);
  ```
Simple DOM Example

public class EchoDOM{
    static Document document;
    public static void main(String argv[]){
        DocumentBuilderFactory factory =
            DocumentBuilderFactory.newInstance();
        try {
            DocumentBuilder builder =
                factory.newDocumentBuilder();
            document = builder.parse(
                new File("test.xml"));
        } catch (Exception e){
            System.err.println("Sorry, an error: " + e);
        }
        if(document!=null){
            scanDOMTree(document);
        }
    }
}
Java DOM processing

• Once we parse() a document this is now in memory
• We need to **TRAVERSE** the tree to read the document
• One technique is to use **Recursion**
  – i.e. The method will call itself
  – We then make decisions based on what the current node type is eg: DOCUMENT, ELEMENT, TEXT etc
Traverse DOM tree - 1

Scan(\textit{Document})

\begin{itemize}
  \item \texttt{Document}\ Name: \#document
  \item \texttt{Element}\ Name: addressBook
    \begin{itemize}
      \item \texttt{Element}\ Name: entry
        \begin{itemize}
          \item \texttt{Element}\ Name: name
            \begin{itemize}
              \item \texttt{Text}\ Name: \#text
                \begin{itemize}
                  \item Value: "Ms Smith"
                \end{itemize}
            \end{itemize}
          \item \texttt{Element}\ Name: address
            \begin{itemize}
              \item \texttt{Text}\ Name: \#text
                \begin{itemize}
                  \item Value: "1 Central..."
                \end{itemize}
            \end{itemize}
          \item \texttt{Element}\ Name: phone
            \begin{itemize}
              \item \texttt{Text}\ Name: \#text
                \begin{itemize}
                  \item Value: "555 5555"
                \end{itemize}
            \end{itemize}
        \end{itemize}
    \end{itemize}
\end{itemize}
Traverse DOM tree - 2

Scan(addressBook)

〈Document〉
Name: #document

〈Element〉
Name: addressBook

〈Attribute〉
Name: list
Value: "personal"

〈Element〉
Name: entry

〈Element〉
Name: name

〈Text〉
Name: #text
Value: "Ms Smith"

〈Element〉
Name: address

〈Text〉
Name: #text
Value: "1 Central..."

〈Element〉
Name: phone

〈Text〉
Name: #text
Value: "555 5555"

...
Traverse DOM tree - 3

Scan(entry)

〈Document〉
Name: #document

〈Element〉
Name: addressBook

〈Element〉
Name: entry

〈Attribute〉
Name: list
Value: "personal"

〈Element〉
Name: name

〈Text〉
Name: #text
Value: "Ms Smith"

〈Element〉
Name: address

〈Text〉
Name: #text
Value: "1 Central..."

〈Element〉
Name: phone

〈Text〉
Name: #text
Value: "555 5555"
Traverse DOM tree - 4

Scan(name)

 Dokument
Name: #document

 Element
Name: addressBook

 Element
Name: entry

 Attribute
Name: list
Value: "personal"

 Element
Name: name

 Text
Name: #text
Value: "Ms Smith"

 Text
Name: #text
Value: "1 Central..."

 Element
Name: phone

 Text
Name: #text
Value: "555 5555"
recursive tree processing

private static void scanDOMTree(Node node) {
    int type = node.getNodeType();
    switch (type) {
        case Node.ELEMENT_NODE:
            System.out.print("<" + node.getNodeName() + ">");
            NamedNodeMap attrs = node.getElementsByTagName();
            for (int i = 0; i < attrs.getLength(); i++) {
                Node attr = attrs.item(i);
                System.out.print(" " + attr.getNodeName() + "=" + attr.getNodeValue() + ");
            }
            NodeList children = node.getElementsByTagName();
            if (children != null) {
                int len = children.getLength();
                for (int i = 0; i < len; i++) {
                    scanDOMTree(children.item(i));
                }
            }
            System.out.println("</" + node.getNodeName() + ">");
            break;
    }
}
DOM nodes

• A "node" can be an element, attribute etc

// Check the type of node
if (n.getNodeType() == Node.ELEMENT_NODE) {
    // process an ELEMENT
    String elementName = n.getNodeName();
} else if (n.getNodeType() == Node.ATTRIBUTE_NODE) {
    // process an ATTRIBUTE
    String attributeName = n.getNodeName();
    String attributeValue = n.getNodeValue();
} else if (n.getNodeType() == Node.TEXT_NODE) {
    // process a TEXT node
    String textValue = n.getNodeValue();
} // etc.

• Your application should know what to expect
  – may not need to enumerate all cases
# DOM Node Types

<table>
<thead>
<tr>
<th>Interface</th>
<th>nodeName</th>
<th>nodeValue</th>
<th>attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attr</td>
<td>name of attribute</td>
<td>value of attribute</td>
<td>null</td>
</tr>
<tr>
<td>CDATASection</td>
<td>&quot;#cdata-section&quot;</td>
<td>content of the CDATA Section</td>
<td>null</td>
</tr>
<tr>
<td>Comment</td>
<td>&quot;#comment&quot;</td>
<td>content of the comment</td>
<td>null</td>
</tr>
<tr>
<td>Document</td>
<td>&quot;#document&quot;</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>DocumentFragment</td>
<td>&quot;#document-fragment&quot;</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>DocumentType</td>
<td>document type name</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>Element</td>
<td>tag name</td>
<td>null</td>
<td>NamedNodeMap</td>
</tr>
<tr>
<td>Entity</td>
<td>entity name</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>EntityReference</td>
<td>name of entity referenced</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>Notation</td>
<td>notation name</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>ProcessingInstruction</td>
<td>target</td>
<td>entire content excluding the target</td>
<td>null</td>
</tr>
<tr>
<td>Text</td>
<td>&quot;#text&quot;</td>
<td>content of the text node</td>
<td>null</td>
</tr>
</tbody>
</table>
Common DOM Methods

- **Node.getNodeType()** - the type of the underlying object, e.g. Node.ELEMENT_NODE
- **Node.getNodeName()** - value of this node, depending on its type, e.g. for elements it’s tag name, for text nodes always string #text
- **Node.getFirstChild()** and **getLastChild()** - the first or last child of a given node
- **Node.getNextSibling()** and **getPreviousSibling()** - the next or previous sibling of a given node
- **Node.getAttributes()** - collection containing the attributes of this node (if it is an element node) or null
Common DOM methods (2)

- **Node.getNodeValue()** - value of this node, depending on its type, e.g. value of an attribute but null in case of an element node
- **Node.getChildNodes()** - collection that contains all children of this node
- **Node.getParentNode()** - parent of this node
- **Element.getAttribute(name)** - an attribute value by name
- **Element.getElementsByTagName()** - collection of all descendant Elements with a given tag name
Common DOM methods (3)

- `Element.setAttribute(name, value)` - adds a new attribute, if an attribute with that name is already present in the element, its value is changed.
- `Attr.getValue()` - the value of the attribute.
- `Attr.getName()` - the name of this attribute.
- `Document.getDocumentElement()` - allows direct access to the child node that is the root element of the document.
- `Document.createElement(tagName)` - creates an element of the type specified.
Searching in DOM

• You can search for element tags:

    NodeList Document.getElementsByTagName()

    abook = mydoc.getElementsByTagName("addressbook")
    addr = abook.getElementsByTagName("address")

• Also can search by XML id attribute:

    Element person = mydoc.getElementById("1")
    would return the ONE element with the id="1"
    e.g. <person id="1">

• See also the javax.xml.xpath API (later...)

(c) University of Technology, Sydney
Create Elements/Attributes

- Create element; then append to parent

```java
// Create empty XML Document
Document docXMLDoc = domBuilder.newDocument();

// Create a person element
Element elmPerson = docXMLDoc.createElement("person");

// Create name attribute and set its value to “Jeff”
elmPerson.setAttribute("name", "Jeff");

// Attach person element to the XML Doc
docXMLDoc.appendChild(elmPerson);

Resulting XML:

<person name="Jeff"/>
```
SAX basics

• SAX is a very different kind of parser to DOM

• It relies heavily upon a content handler class that the programmer (you!) must define
  – the content handler class needs to implement certain methods, e.g.
    • `startDocument()` and `endDocument()`
    • `StartElement()` and `EndElement()`
    • `characters()`
    • ... others ...

  – as the SAX parser is reading in the XML document, when it encounters various kinds of markup, it will call your methods, defined in your content handler class
Simple API for XML (SAX)

- Event driven processing of XML documents
- Parser sends events to programmer’s code
- Programmer decides what to do with every event
- SAX parser doesn't create any objects at all, it simply delivers events
- But impossible to move backward in XML data
- Impossible to modify document structure
- Fastest and least memory intensive way of working with XML
Parsing XML document with SAX

• All examples for this part based on:
  http://java.sun.com/webservices/docs/1.3/tutorial/doc/JAXPSAX.html

• Import necessary classes:
  import org.xml.sax.*;
  import org.xml.sax.helpers.DefaultHandler;
  import javax.xml.parsers.SAXParserFactory;
  import javax.xml.parsers.SAXParser;
Create SAX Parser

- **SAXParserFactory** - creates an instance of the parser factory
  
  ```java
  SAXParserFactory factory = SAXParserFactory.newInstance();
  ```

- **SAXParser** - defines several kinds of `parse()` methods. Every parsing method expects an XML data source (file, URI, stream) and a `DefaultHandler` object (or object of any class derived from `DefaultHandler`)
  
  ```java
  SAXParser saxParser = factory.newSAXParser();
  saxParser.parse( new File("test.xml"), handler);
  ```
Handle SAX events

Among others:

- **startDocument** – receives notification of the beginning of a document
- **endDocument** – receives notification of the end of a document
- **startElement** – gives the name of the tag and any attributes it might have
- **endElement** – receives notification of the end of an element
- **characters** – parser will call this method to report each chunk of character data
SAX simple example

```xml
<?xml version="1.0"?>
<addressBook>
  <entry list="personal">
    <name>
      Ms Smith
    </name>
    ... & so on
  </entry>
</addressBook>
```

- startDocument()
- startElement(addressBook)
- startElement(entry)
  - Attributes[list]
- startElement(name)
- Characters(Ms Smith)
- endElement(name)
- endDocument()

**Caution:** your code MAY get extra characters() events due to line feeds, blank spaces etc
public class EchoSAX extends DefaultHandler
{
    public void startDocument() throws SAXException
    {//override necessary methods}
    public void endDocument() throws SAXException {...}
    public void startElement(...) throws SAXException{...}
    public void endElement(...) throws SAXException{...}
}
Overriding of methods

```java
public void startDocument() throws SAXException{
    System.out.println("DOCUMENT:");
}

public void endDocument() throws SAXException{
    System.out.println("END OF DOCUMENT:");
}

public void endElement(String namespaceURI, String sName,
                        String qName) throws SAXException {

    System.out.println("END OF ELEMENT: "+sName);
}
```
Parsing of the XML document

DefaultHandler handler = new EchoSAX();

// Use the default (non-validating) parser
SAXParserFactory factory = SAXParserFactory.newInstance();

// Set validation on
factory.setValidating(true);

// Parse the input
SAXParser saxParser = factory.newSAXParser();

saxParser.parse(new File("test.xml"), handler);
Notes on SAX

• SAX doesn't keep track of the document structure
  – That's up to the programmer
    • e.g. all textual data will be processed by the characters() method
    • There is nothing attached to the character data to indicate which element it belongs to
  – You have to keep track of "where you're up to"

• See the sample reading for one way to do this in a SAX content handler, using a stack
  • Philipp K. Janert, *Simple XML Parsing with SAX and DOM*, ONJava.com [Internet]. http://www.onjava.com/lpt/a/2452
StAX

- JSR173 "Streaming API for XML" (StAX)
- Similar to SAX, but the PROGRAMMER "pulls" each XML component and acts accordingly
- 2 versions – cursor API and iterative API
  - **Cursor API**: XMLStreamReader
    - Treats XML inputstream as set of tokens
    - You parse the tokens per element, attribute etc.
  - **Iterative API**: XMLEventReader
    - Parser treats XML stream as list of "XML Events". You then check each event
Using sTAX Cursor API

- Using Cursor API
  1. create a parser object
  2. create an input stream for the parser
  3. loop through each returned XML element using the `next()` method
  4. retrieve the tag name and value by using the `getLocalName()` and `getText()` methods
  5. retrieve attributes by using the `getAttributeName()` and `getAttributeValue()` methods
Example

```xml
<?xml version="1.0"?>
<catalog>
  <title id="1">
    <name>J2ME</name>
  </title>
</catalog>
```

START_DOCUMENT
START_TAG catalog
START_TAG title attribute: id = 1
START_TAG name
TEXT J2ME
END_TAG name
END_TAG title
END_TAG catalog
END_DOCUMENT
Sample code

• Implement as a simple loop:
  while (reader.hasNext()) {
    switch (reader.next()) {
      case START_ELEMENT: // <name>
        println(reader.getLocalName());
        // "name"
        println(reader.getText()); // "J2ME"
      case END_ELEMENT: // </name>
        & so on...
    }
  }
Creating XML with Stax

• Similar to XMLStreamReader, use XMLStreamWriter
• Just use methods such as
  – writeStartElement("name")
  – writeEndElement("name")
  – writeCharacters("J2ME")
• Finish off with
  – flush() & close()
## Which parser to use?

<table>
<thead>
<tr>
<th>DOM</th>
<th>SAX</th>
<th>StAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Want to read whole document AT ONCE</td>
<td>Interested in part of document only</td>
<td>Interested in part of document only</td>
</tr>
<tr>
<td>Controlled by parser library</td>
<td>Controlled by parser library</td>
<td>Programmer control</td>
</tr>
<tr>
<td>CPU/Memory Hog</td>
<td>lean</td>
<td>minimal. Can be used on mobile phones</td>
</tr>
<tr>
<td>Easy to code – single object</td>
<td>Hard to code &amp; debug – passed events to handle</td>
<td>Medium to code. Intuitive alternative to DOM/SAX</td>
</tr>
<tr>
<td>Can generate XML</td>
<td>Read-only</td>
<td>Can generate XML</td>
</tr>
</tbody>
</table>
JAXB

- Alternative to using a parser is to use JAXB (Java Architecture for XML Binding)
  - JSR31 for JAXB 1.0. JSR222 for JAXB 2.0
  - Uses XML schema to generate Java classes
  - Use `xjc` to create java classes from XML schema
  - Use `schemagen` to create XML schema from Java (need annotations `@XmlElement` tags)
  - You will see this being used later in web services tutorial
- Alternatives include `xmlBeans` (apache Axis)
JAXB 2.0

- JAXB applications need to runtime framework too....

(from Sun javaee 5 tutorial)
JAXB binding runtime

- Your application accesses JAXB objects like javabeans. Use getter/setters to access elements, attributes etc

- Use ObjectFactory to create the JAXB object

- Use UnMarshal class to read XML document

(from Sun javaee 5 tutorial)
JAXB Code sample

• From /pub/dca/lab05/sample2 directory...
• **xjc** sample2.xsd
  • → AddressList.java & ObjectFactory.java

```java
JAXBContext jc = JAXBContext.newInstance(ObjectFactory.class);
Unmarshaller u = jc.createUnmarshaller();
AddressList a = (AddressList) u.unmarshal(new FileInputStream("sample2.xml"));
// now you have a javabean containing address elements...
List<AddressList.Address> al = a.getAddress();
for (Address address : al) {
    System.out.println(address.getName());
}
```
Agenda

- XML Introduction
- XML Namespaces
- DTD/XML Schema
- XML programming
- XSLT/XPATH
XSLT & XPATH

- Often we need to transform XML to different formats
  - eg: from one Schema to another
  - eg: from XML to XHTML
  - eg: from XML to text
- Use Extensible Stylesheet Language Transformations (XSLT)

- This also requires using XPATH expressions to match elements and attributes
Formatted XML Data

My CD Collection

<table>
<thead>
<tr>
<th>Title</th>
<th>Artist</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empire Burlesque</td>
<td>Bob Dylan</td>
</tr>
<tr>
<td>Hide your heart</td>
<td>Bonnie Tyler</td>
</tr>
<tr>
<td>Greatest Hits</td>
<td>Dolly Parton</td>
</tr>
<tr>
<td>Still got the blues</td>
<td>Gary Moore</td>
</tr>
<tr>
<td>Bros</td>
<td>Eros Ramazzotti</td>
</tr>
<tr>
<td>One night only</td>
<td>Bee Gees</td>
</tr>
<tr>
<td>Sylvia Mother</td>
<td>Dr Hook</td>
</tr>
<tr>
<td>Maggie May</td>
<td>Rod Stewart</td>
</tr>
<tr>
<td>Romanza</td>
<td>Andrea Bocelli</td>
</tr>
<tr>
<td>When a man loves a woman</td>
<td>Percy Sledge</td>
</tr>
<tr>
<td>Black angel</td>
<td>Savage Rose</td>
</tr>
<tr>
<td>1999 Grammy Nominees</td>
<td>Many</td>
</tr>
<tr>
<td>For the good times</td>
<td>Kenny Rogers</td>
</tr>
<tr>
<td>Big Willie style</td>
<td>Will Smith</td>
</tr>
<tr>
<td>Tupelo Honey</td>
<td>Van Morrison</td>
</tr>
</tbody>
</table>
XSLT Formatting

XSL – Extensible Stylesheet Language

XML → XSLT processor → XML or XHTML
XSLT Tree conversion

- XSLT converts a source tree into a result tree
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

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

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

    ...

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

</xsl:stylesheet>
<title>Empire Burlesque</title>

element node

Text node

Name: title
Value: null

Name: #text
Value: "Empire Burlesque"
XML Document Tree

root

catalog

cd

title "Empire Burlesque"
artist "Bob Dylan"
country "USA"
company "Columbia"
price "10.90"
year "1985"

cd

title "Empire Burlesque"
artist "Bob Dylan"
Tree Walking

• If the current node has children, then the first child is chosen

• If the current node has following siblings, then the next sibling is chosen

• If the current node does not have a following sibling, then the following sibling with following siblings is chosen

• If no such ancestors exist, then the process is complete
XLST Tree Walking

root → catalog → cd

1 → 2 → 3 → cd → 4 → title "Empire Burlesque" → artist "Bob Dylan" → country "USA" → company "Columbia" → price "10.90" → year "1985" → title "Empire Burlesque" → artist "Bob Dylan" → cd → 16 → 17 → title "Empire Burlesque" → artist "Bob Dylan" → cd → 19 → 20 → title "Empire Burlesque" → artist "Bob Dylan"
XSL Templates

```xml
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

    <xsl:template match="/">
        Some Output
    </xsl:template>

</xsl:stylesheet>

The “root” element
```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    
    <xsl:template match="/">
        <html><body>
            <h2>My CD Collection</h2>
            <table border="1">
                <tr bgcolor="#9acd32">
                    <th>Title</th>
                    <th>Artist</th>
                </tr>
            </table>
        </body></html>
    </xsl:template>

</xsl:stylesheet>
Invoking another template (1)

```xml
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

    <xsl:template match="/">
        <html>
            <body>
                <h2>My CD Collection</h2>
                <table border="1">
                    <tr bgcolor="#9acd32">
                        <th>Title</th>
                        <th>Artist</th>
                    </tr>
                    <xsl:apply-templates/>
                </table>
            </body>
        </html>
    </xsl:template>
</xsl:stylesheet>
```
Invoking another template (2)

```xml
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  
  <xsl:template match="/">
    <html><body>
      <h2>My CD Collection</h2>
      <table border="1">
        <tr bgcolor="#9acd32">
          <th>Title</th>
          <th>Artist</th>
        </tr>
        <xsl:apply-templates/>
      </table>
    </body></html>
  </xsl:template>

  <xsl:template match="cd">
    <tr><xsl:apply-templates/></tr>
  </xsl:template>

</xsl:stylesheet>
```
Invoking another template (3)

...  
<xsl:apply-templates/>
</table>
</body></html>
</xsl:template>
<xsl:template match="cd">
  <tr><xsl:apply-templates/></tr>
</xsl:template>
<xsl:template match="title">
  <td><xsl:apply-templates/></td>
</xsl:template>
<xsl:template match="artist">
  <td><xsl:apply-templates/></td>
</xsl:template>
Hiding content

...  
   <xsl:apply-templates/>
   </table>
   </body></html>
</xsl:template>
<xsl:template match="cd">
   <tr><xsl:apply-templates/></tr>
</xsl:template>
<xsl:template match="title">
   <td><xsl:apply-templates/></td>
</xsl:template>
<xsl:template match="artist">
   <td><xsl:apply-templates/></td>
</xsl:template>
<xsl:template match="country"/>
<xsl:template match="company"/>
<xsl:template match="price"/>
<xsl:template match="year"/>
</xsl:stylesheet>
XSL hiding content

```xml
<xsl:apply-templates/>

<xsl:template match="cd">
  <tr><xsl:apply-templates/></tr>
</xsl:template>

<xsl:template match="title">
  <td><xsl:apply-templates/></td>
</xsl:template>

<xsl:template match="artist">
  <td><xsl:apply-templates/></td>
</xsl:template>

<xsl:template match="country"/>
<xsl:template match="company"/>
<xsl:template match="price"/>
<xsl:template match="year"/>
```
Extracting content

• You can extract the value of an element by using the `<xsl:value-of select="expression"`

```xml
<xsl:template match="title">
  <td>artist=<xsl:value-of select="/catalog/cd/artist"/>
</xsl:template>
```

• Expression can be any Xpath expression.

• Use @attribute to return attribute values
**XSLT logic elements**

- Like programming languages, XSLT has some control logic elements
  - `<xsl:for-each select="expression">`
    - repeating text
  `</xsl:for-each>`
  - `<xsl:if test="expression">`
    - content
  `</xsl:if>`
  - `<xsl:choose>`
    - `<xsl:when test="expression1">content1</xsl:when>`
    - ...
    - `<xsl:otherwise>content</xsl:otherwise>`
  `</xsl:choose>`
'Match' attribute

- The match attribute in the template element is an expression as defined by the XPath specification
  - http://www.w3.org/TR/xpath

- The patterns used in the match attribute are XPath expressions:

```xml
<xsl:template match="/catalog/cd/*"/>
```
XML – XPath

• XPath is a language for addressing parts of an XML document

• Uses a compact, non-XML syntax
  – So we can use XPath within URIs and XML attribute values

• Operates on the abstract, logical structure of an XML document

• Gets its name from its use of a path notation for navigating through the hierarchical structure of an XML document
XML – XPath (cont)

- XPath models an XML document as a tree of nodes.
- There are different types of nodes, including element nodes, attribute nodes and text nodes.
- It defines a way to compute a string-value for each type of node.
  - Some types of nodes also have names

- XPath fully supports XML Namespaces.
  - Name of a node is modeled as a pair, called a Data Model
  - Consists of (local part, (possibly null) namespace URI)
Example:

```
catalog/cd/company
```

Diagram representing a hierarchical structure with nodes for title, artist, country, company, price, year, and artist. The highlighted node is for the CD titled "Empire Burlesque" by Bob Dylan, published by Columbia in 1985 with a price of $10.90.
XML – XPath (cont)

- Main part of XPath is the **expression**.
- An expression is evaluated to return an object of type:
  - **node-set** (an unordered collection of nodes without duplicates)
  - **boolean** (true or false)
  - **number** (a floating-point number)
  - **string** (a sequence of UCS characters)

- Some examples of locations paths:
  - `child::para` selects the para element children of the context node
  - `child::*` selects all element children of the context node
  - `child::text()` selects all text node children of the context node
XPATH examples

- Select alternative elements:
  match="X | Y | Z"

- "match the elements X, Y and Z only"
XPATH examples

• Select alternative elements *example:*
  match="name | foreign"

<para>The <foreign>de facto</foreign> stylesheet language is <name>XSLT</name>.
</para>

• Result:
The *de facto* stylesheet language is *XSLT*
**XPath examples**

- **Specific parent:**
  
  match="P/X"

- “match X if it has a parent of P”
 XPath Examples

• **Specific Parent example:**
match="warning/para"

  `<para>A normal paragraph.</para>
  `<warning>
    `<para>Warning paragraph one.</para>
    `<para>Warning paragraph two.</para>
  </warning>
  `<para>Another normal paragraph.</para>`

  A normal paragraph.
  WARNING:
  *Warning paragraph one.*
  *Warning paragraph two.*
  Another normal paragraph.
XPath Examples

- **Specific ancestor:**
  
  match="A/P/X"

- “match X, but only when X has a parent of P, and P has a parent of A”
**XPath Examples**

- **Specific ancestor example:**
  
  match="intro/warning/para"

  ```xml
  <intro>
    <para>An introduction paragraph.</para>
    <warning>
      <para>Introduction warning.</para>
    </warning>
  </intro>
  
  An introduction paragraph.
  
  **Introduction warning.**
  ```
XPath Examples

• **Unknown Ancestry**
  – match="A/P/X"
  – match="A/Q/X"
  – match="A/R/X"

\[\text{match}="A/*/X"\]

• “match all X elements where A is their grandparent”
XPath Examples

• **Unknown Ancestry example:**
  match="intro/*/para"

  
  <intro>
    <para>An introduction paragraph.</para>
    <warning>
      <para>Introduction warning.</para>
    </warning>
    <note>
      <para>Introduction note.</para>
    </note>
  </intro>

  An introduction paragraph.
  *Introduction warning.*
  *Introduction note.*
XPath Examples

- Unknown ancestry examples:
  - match="intro/para"
  - match="intro/*/para"
  - match="intro/*//*/para"
  - match="intro/*//*//*/para"

match="intro//para"
XPath Examples

- **Specific Children – Predicate (search) Filter**
  
  match="X[C]"

- “match X when at least one of its children is C”
XPath Examples

• **Specific Siblings**
  match="P[S]/X"

• “matches all elements named X that have a sibling named S”
XPath Examples

• Attribute Context
  match="X[@A]"

• “matches X that have an attribute A”
  – eg:  <X A="something"/>
XPath Examples

• Attribute Value Context
match="X[@A='abc']"

• “matches X that have an attribute A with a value of abc”
  – eg:  <X A="abc"/>
XPath Examples

• Specific Child NOT present:
  match="X[not(C)]"

• “match X that do not have a child C”
XPath Examples

• **Specific Sibling NOT present**
  
  match="P[not(S)]/X"

```
<Z>
<X>
<P>
```

• “matches X that do **not** have a sibling S”
**XPath Examples**

- **Specific Attribute NOT present**
  
  match="X[not(@A)]"

- "matches X that do not have an attribute named A"
Priorities

- <template match="intro//para"> ... </template>
- <template match="warning//para"> ... </template>

- With priorities:
  - <template match="intro//para" priority="1"> ... </template>
  - <template match="warning//para" priority="2"> ... </template>
Default Priorities

- PRIORITY ‘0.5’
  <template match="warning/para">

- PRIORITY ‘0’
  <template match="para">
  <template match="x:para"> (ie: name space specific)

- PRIORITY ‘-0.25’
  <template match="x:*">

- PRIORITY ‘-0.5’
  <template match="*">
  <template match="node()">
Conflict Resolution

• When two or more templates remain applicable after lower priority matches, the template closest to the end of the stylesheet should be chosen.

However, not all XSLT processors might follow this standard behaviour.
**XPath/XSLT functions**

- You can define variables with XSLT.
- `<xsl:variable name="x" select="expression" />`

- You then refer to the variable as `<xsl:variable name="x" />`
  - Or from within any `<xsl:>` element as `${x}`

- XPath also provides many functions:
  - see [http://www.w3.org/TR/xpath](http://www.w3.org/TR/xpath)
  - see [www.w3schools.com/xpath/xpath_functions.asp](http://www.w3schools.com/xpath/xpath_functions.asp)
XPath/XSLT Functions

<intro>
  <para>Intro paragraph.</para>
  <warning>
    <para>Intro warning.</para>
  </warning>
</intro>

<xsl:template match="intro">
  There are <xsl:value-of select="count(para)"/>
  intro paragraphs.
  <xsl:apply-templates/>
</xsl:template>

There are 1 intro paragraphs.
XPath/XSLT Functions

• `<xsl:template match="intro//para/text()[1]">`
  `<b>`<xsl:value-of select="substring(.,1,1)"/>`</b>
  `<b>`<xsl:value-of select="substring(.,2)"/>`</b>
`</xsl:template>`

  <b>Intro paragraph.</b>
  <b>Intro warning.</b>
Using XSLT

• XSLT is a declarative language
  – ie: it doesn’t execute, isn’t compiled
• Instead, we call programs/methods which use an XSLT template to do the transformation
• We can link XSLT directly to XML file
  – → Most browsers will automatically transform it!
• In Java, we can use javax.xml.transform package
Most browsers will use `cdcatalog1.xsl` to transform this XML file.
Using Java to transform XML

try {
    // Load StreamSource objects with XML and XSLT files
    StreamSource xmlSource =
        new StreamSource(new File("input.xml"));
    StreamSource xsltSource =
        new StreamSource(new File("format.xslt"));
    // Create a StreamResult pointing to the output file
    StreamResult fileResult =
        new StreamResult(new FileOutputStream("output.xml"));
    // Load a Transformer object and perform the transformation
    TransformerFactory tfFactory =
        TransformerFactory.newInstance();
    Transformer tf = tfFactory.newTransformer(xsltSource);
    tf.transform(xmlSource, fileResult);
}
    catch (Exception e) { e.printStackTrace(); }
Using the XPATH API

• Java 5 also has javax.xml.xpath package:
  XPath xpath = XPathFactory.newInstance().newXPath();
  InputSource in_src = new InputSource("address.xml");
  NodeList nodes = (NodeList)
      xpath.evaluate("//address", input,
      XPathConstants.NODESET);

• Or to use with DOM document, just replace in_src with document

• Use the 3rd parameter to set the return value type
  - NODESET  → return NodeList
  - STRING
  - BOOLEAN
  - NUMBER
Using Java utility to transform..

- Apache Xalan XML parser has a utility to transform XML using XSL
  - (weblogic, websphere etc have this built in)

  - `java org.apache.xalan.xslt.Process`  
    - IN `addressbook.xml`  
    - XSL `address2html.xsl`  
    - OUT `address.html`

Alternatives:

- `msxsl.exe` on windows
- `xsltproc` on linux (from libxslt)
XML/XSLT on J2EE

- XML + XSL allows us to separate layout and content

Diagram:
- HTML
- Servlet/JSP
- Transform
- XSL
- XML
- EJB/Java/Web service

(c) University of Technology, Sydney
Programming XPATH

- Use java.xml.xpath package
- Use PathFactory to create an Xpath object
- Use evaluate() method to filter results
  - evaluate() can return String or NodeList
- Eg: if we have a DOM document, you can use:

  XPath xpath = PathFactory.newInstance().newXPath();

  NodeList nodes = (NodeList) xpath.evaluate("//cd", document, xpathConstants.NODESET);
Summary

• XML is a complex and rapidly growing field of technology

• XML is widely used already, and is becoming ubiquitous within e-commerce developments, particularly for B2B systems

• This lesson has introduced to a wide range of current XML specifications and technologies – you will need to do further study to investigate these areas in more detail
Debugging

• xsl:message("Now at somewhere")