Approaches to mapping

- XSLT (+DTD)
- Simple mapping example (books)
- RDBMS views
- CORBA IDL

'Web Services Made Easier', Sun Microsystems Technical White Paper, http://java.sun.com/xml/webservices.pdf

The Java Web Services Tutorial, http://java.sun.com/webservices/tutorial.html

XML Document Example

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

<!DOCTYPE AddressList SYSTEM
"AddressList.dtd" >

<!-- Simple Address Example -->

<AddressList>
  <Address Name="Fred Bloggs" >
    <Work>
      <Street>70 Symonds St</Street>
      <City>Auckland</City>
    </Work>
    <Work>
      <Street>38 Princes St</Street>
      <City>Auckland</City>
    </Work>
  </Address>
```

```
<Address Name="Myra Smith" >
    <Work>
      <Street>55 The Terrace</Street>
      <City>Wellington</City>
    </Work>
    <Home>
      <Street>18 Adams Terrace</Street>
      <City>Wellington</City>
    </Home>
  </Address>
</AddressList>
```


XML Document Structure

- The XML Declaration
- Document Type Declaration
- Document Body
  - Elements
  - Attributes
  - Character Data
  - Comments

The XML Declaration - Processing instructions

- `<?xml version="1.0" encoding="ISO-8859-1" standalone="yes" ?>`
  - Processing instructions for the application consuming the document `<? target processing_instructions ?>`
  - Identifies as an XML file and specifies version conformance
  - Encoding to specify character set used in document
  - `standalone="yes"`
    - No external markup declarations which affect the XML information

Document Type Declaration

- `<!DOCTYPE doc_name SYSTEM "{uri}">`
  - `<!DOCTYPE ADDRESSLIST SYSTEM "AddressList.dtd">`
  - `doc_name` must be the root ELEMENT of the DTD
  - SYSTEM indicates the DTD is at the given URI
- `<!DOCTYPE doc_name PUBLIC "{catalog id}">`
- `<!DOCTYPE doc_name PUBLIC "{catalog id}" "{uri}">`
  - `<!DOCTYPE PERSON PUBLIC "//DSTC/PERSON">`
  - PUBLIC indicates the application knows where to find the DTD
- Internal specification

```
<!DOCTYPE PERSON [
  <!ELEMENT PERSON (name, address, phone+, company?)>
  …
]>
```


XML and DTDs

- Define an instance of XML language (vocabulary)
- Good points
  - Define document organisation in an easily shared manner
  - Understand full structure for further manipulation
  - Validating parser can ensure correctness
  - Can define required and optional information
- Disadvantages
  - Different syntax from rest of XML
  - Validating parser required to read another file
  - Complexity of parsing with DTD is increased

XML DTD

- Identifies instance of XML language
- Meta information about a document's contents
  - Valid elements
  - Valid attribute names and values
  - Nesting structure allowed
- DTD usually a separate document
- DTD describes syntax of document - not semantics
  - XMLSchema is the preferred method to describe complex data structures as it provides fine-grain control of structural specification of a schema (in an object-oriented manner).

XML DTD Example

```
<!ELEMENT catalogue (publisherDetails, (publication)+) >
<!ELEMENT publisherDetails (publisherName, phone?, fax?, email?) >
<!ELEMENT publication (title, creator, subject?, description?, cost) >
<!ELEMENT publisherName (#PCDATA) >
<!ELEMENT phone (#PCDATA) >
<!ELEMENT fax (#PCDATA) >
<!ELEMENT email (#PCDATA) >
<!ELEMENT title (#PCDATA) >
<!ELEMENT creator (#PCDATA) >
<!ELEMENT subject (#PCDATA) >
<!ELEMENT description (#PCDATA) >
<!ELEMENT cost (#PCDATA) >
```

XML Document

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE catalogue SYSTEM "BookBroker.dtd" >
<catalogue>
  <publisherDetails>
    <publisherName>Amorzon</publisherName>
    <phone>83068</phone>
    <fax>82651</fax>
    <email>trebor@cs.auckland.ac.nz</email>
  </publisherDetails>
  <publication>
    <title>A Reader in Planning Theory</title>
    <creator>Faludi, A</creator>
    <cost>15.99</cost>
  </publication>
  <publication>
    <title>Gender, Planning and the Policy Process</title>
    <creator>LITTLE, JO</creator>
    <description>Planning has a central essential legitimacy in addressing
social goals.</description>
    <cost>14.99</cost>
  </publication>
```

Element Declarations

- Map to tags in the final document
- `<!ELEMENT name content-model >`
  - content-model specifies terminal and non-terminal content
  - `?` optional (0 or 1)
  - `*` 0 or more
  - `+` 1 or more
  - `( a | b )` either a or b but not both
  - `( a , b )` a followed by b

Element Declarations

- No look-ahead in processors so content-model must be parsable without back tracking
  - `(a, b, c, d) | (b, c, d) | (c, d) | (d)`
  - `(a, b, c, d) | (a, b, c) | (a, b) | (a)`
- `#PCDATA` for terminal content
  - Parsed character data, allows text and markup
- `EMPTY` for no content
  - `<name></name>` or `<name />`
- `ANY` to match any content

DTD with attribute

```
<!ELEMENT AddressList (Address)* >

<!ELEMENT Address (Work|Home)+ >
<!ATTLIST Address Name CDATA #REQUIRED >

<!ELEMENT Work (Street, City) >
<!ELEMENT Home (Street, City) >

<!ELEMENT Street (#PCDATA) >
<!ELEMENT City (#PCDATA) >
```

```
<AddressList>
  <Address Name="Fred Bloggs" >
    <Work>
      <Street>70 Symonds St</Street>
      <City>Auckland</City>
    </Work>
    <Work>
      <Street>38 Princes St</Street>
      <City>Auckland</City>
    </Work>
  </Address>
```

Attribute Declarations

- Container for attributes associated with an element
- `<!ATTLIST element_name (att_name type default)+ >`
- Attribute types
  - `CDATA` character data (string)
  - `ID` unique ID within the document
  - `IDREF` a reference to a unique ID within the document
  - `IDREFS` a list of references to unique IDs
  - `ENTITY` a reference to an entity within the document
  - `ENTITIES` a list of references to entities
  - `NMTOKEN` a valid XML name token
  - `NMTOKENS` a list of valid XML name tokens
  - `NOTATION` an enumerated reference to a list of notation data types
  - `(value1 | value2 | …)` an enumerated list of possible values

Attribute Declarations

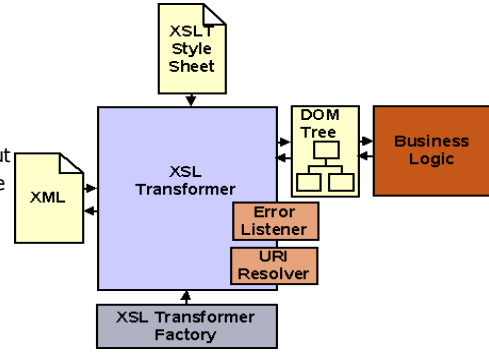
- Default types
  - `#REQUIRED` - must be specified for the element
  - `#IMPLIED` - attribute may not be specified, application will be able to calculate a value
  - `"default_value"` - if attribute is not specified then use this value
  - `#FIXED "constant_value"` - attribute will contain this value if specified
- References have some conventions
  - `<node ID="node101" >This is 101</node>`
  - `<start ref="node101" >`
  - also evolving standards Xlink, Xpointer
- Why use Entities rather than Attributes?

XSL/XSLT

- Extensible Stylesheet Language (XSL) and XSL Transformations (XSLT)
- XSL is a formatting language, for converting XML documents into formatted documents (building upon style sheets)
- JAXP includes XSLT implementation as part of javax.xml.transform package (actually wraps the Xalan XSLT implementation)

XSLT

- Basic approach, transform from DOM to DOM using XSL stylesheet to specify the transformation
- Resultant DOM represents formatted document which is then walked to produce output
- Some implementations handle SAX inputs directly (so don't need a DOM)



XML Example

- Coffee price list and DTD (from "Web Services Made Easier")

```
<priceList>
  <coffee>
    <name>Mocha Java</name>
    <price>11.95</price>
  </coffee>
  <coffee>
    <name>Sumatra</name>
    <price>12.50</price>
  </coffee>
</priceList>

<!ELEMENT priceList (coffee)+>
<!ELEMENT coffee (name, price) >
<!ELEMENT name (#PCDATA) >
<!ELEMENT price (#PCDATA) >
```


XSL Basic Approach

- XSL uses a rule-based template matching approach
- XSL uses a XML encoding so it has a tagged structure (which makes it difficult to read)
- Example with the coffee price list DTD from the web services paper:

```
<!ELEMENT priceList (coffee)+>
<!ELEMENT coffee (name, price) >
<!ELEMENT name (#PCDATA) >
<!ELEMENT price (#PCDATA) >
```


XSL Rules

- XSL is a rule-based language. Rules (template rules) have:
  - A match pattern, to match against XML elements specified as an Xpath expression
  - A template which specifies the form of the document to produce if an element matches
  - A template may cause further rules to be applied

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="name">  Matches elements with tag name
    <tr><td>                   Constructs a html table row
      <xsl:apply-templates/>   Apply a stylesheet to bits of name element
                               Result goes in this place
    </td></tr>                 Completes the html table row
  </xsl:template>
```


XSL for Coffee Pricelist

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="priceList">
    <html><head>Coffee Prices</head>
      <body>
        <table>
          <xsl:apply-templates />
        </table>
      </body>
    </html>
  </xsl:template>
  <xsl:template match="name">
    <tr><td>
      <xsl:apply-templates />
    </td></tr>
  </xsl:template>
  <xsl:template match="price">
    <tr><td>
      <xsl:apply-templates />
    </td></tr>
  </xsl:template>
</xsl:stylesheet>
```

Application to an example

```
<priceList>
  <coffee>
    <name>Mocha Java</name>
    <price>11.95</price>
  </coffee>
  <coffee>
    <name>Sumatra</name>
    <price>12.50</price>
  </coffee>
</priceList>
```

```
<html><head>Coffee Prices</head>
  <body>
    <table>
      <tr><td>
        Mocha Java
      </td></tr>
      <tr><td>
        11.95
      </td></tr>
      <tr><td>
        Sumatra
      </td></tr>
      <tr><td>
        12.50
      </td></tr>
    </table>
  </body>
</html>
```

Xpath and More Complex Matching

- See the handout from Java Web Services Tutorial for a more complete description of Xpath expressions
  - "/" The root element
  - "/priceList/name" name elements of priceList
  - "SECT|PARA|NOTE" Only SECT, PARA, or NOTE elements
  - "LIST/@type" The type attribute of LIST elements
- Using these can pull a XML structure apart and reorder the results to give a very different tree shape as a result

Phases

- What does it mean for an application to be XML-based or to be a Web Service?
- Typically three phases
  - XML input processing
    - Parsing and validating
    - Recognising/searching/extracting information
    - Binding information to business objects
  - Business logic
    - Processing information
  - XML output processing
    - Constructing a model of document to be produced
    - Applying XSLT or directly serialising to XML

Processing Models

- SAX
  - Serial access with the Simple API for XML
  - Parser generates events as it encounters tokens (callback)
  - Need to do everything in a single cycle
  - Low memory use
- DOM
  - Document Object Model
  - Constructs a parse tree of objects
  - Can walk through a tree multiple times extracting information
    - Ie random access but more memory intensive
  - Also JDOM – DOM tuned for Java – different and simpler construction and access protocol

Processing Models

- XSLT
  - Extensible Stylesheet Language Transformations
  - Higher level approach
    - Codes transformations as rules
    - Condition patterns specified using Xpath expressions
    - Little Java coding needed – a scripting approach
  - XSLT is itself an XML-based grammar (as is Xpath)
- JAXB
  - New Java API for XML/Java Binding
  - Produces object structure (as does DOM) but has compiler that generates classes based on XML DTD
    - Children and attributes accessible as properties
    - Can subclass to provide behaviour

Comparison

Processing Phase	SAX	DOM	XSLT
XML output processing			
Constructing	No default support but can be done by generating a properly balanced sequence of method calls to event handlers	Implicitly part of the model: API factory methods	Implicitly part of the model: XSL statements
Serializing	No default support but can be done with a custom event handler	Implementation specific support, or through XSLT identity transformation	Implicitly part of the model: XSL output method statement

Comparison

Processing Phase	SAX	DOM	XSLT
XML input processing			
Parsing and validating	Built in	Built in or based on SAX	Based on SAX or DOM
Recognizing/ searching	Catching events with event handlers	Searching the tree with tree walkers	Xpath patterns
Extracting	Catching events	Getting attribute values, node content: API methods	Getting attribute values, node contents: Xpath statements
Mapping/ binding	Creating business objects from the extracted information	Creating business objects from the extracted information	If ever, through DOM or SAX (pipelining)

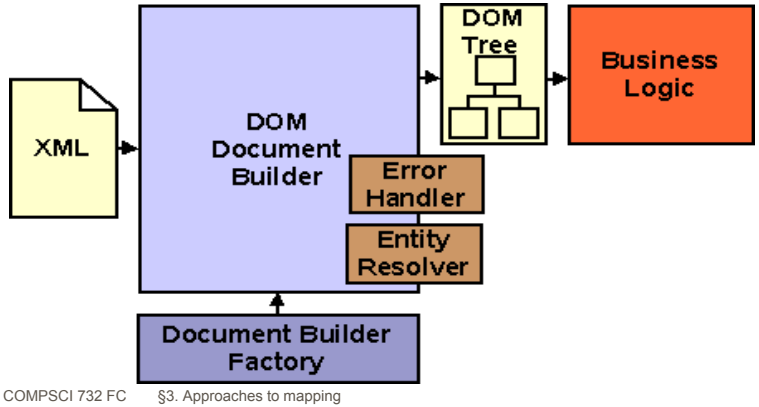
DOM

- Document Object Model (DOM)
  - The DOM specification defines how a XML document can be represented as a hierarchical object structure
  - Also specifies mechanisms for accessing elements within the tree
    - Allows for complex processing/manipulation of the document
  - More memory expensive than SAX as the whole document is in memory (but memory is cheap)
- Sun's JAXP includes a DOM implementation with an API defined in javax.xml.parsers

DOM Construction

- See example in "Web Services Made Easier", pg 6
- DocumentBuilderFactory instance created
- Configuration variables set
- DocumentBuilder instance created using newDocumentBuilder()
- DocumentBuilder object's parse() method used to read in the XML document and construct the parse tree
- You then use the Node access methods to traverse or manipulate the tree
  - Can access by tree walk or by search on tag name

DOM Processing



DOM API

- The DOM API defines interfaces for each of the entities of a XML document
- org.w3c.dom.Node interface: a single node in the document tree
  - Defines methods to access, insert, remove, replace the child nodes
  - Defines methods to access the parent node
  - Defines methods to access the document
- org.w3c.dom.Document interface is a Node that represents the entire XML document
- org.w3c.dom.Element interface is a Node that represents a XML element
- org.w3c.dom.Text interface is a Node that represents the textual content of a XML document

Invoking a DOM parser using JAXP

```
import javax.xml.parsers.*;
import org.xml.sax.*
import org.w3c.dom.*;
import java.io.*;

DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
Document document = builder.parse("priceList.xml");
```

DOM Manipulation

```
Node rootNode = document.getDocumentElement();
NodeList list = document.getElementsByTagName("coffee");

// Loop through the list.
for (int i=0; i < list.getLength(); i++) {
  thisCoffeeNode = list.item(i);
  Node thisNameNode = thisCoffeeNode.getFirstChild();
  if (thisNameNode == null) continue;
  if (thisNameNode.getFirstChild() == null) continue;
  if (! thisNameNode.getFirstChild() instanceof org.w3c.dom.Text) continue;
  String data = thisNameNode.getFirstChild().getNodeValue();
  if (! data.equals("Mocha Java")) continue;
  //We're at the Mocha Java node. Create and insert the new
  //element.
  …
}
```


DOM Manipulation

```
    …
    //We're at the Mocha Java node. Create and insert the new
    //element.
    Node newCoffeeNode = document.createElement("coffee");

    Node newNameNode = document.createElement("name");
    Text tnNode = document.createTextNode("Kona");
    newNameNode.appendChild(tnNode);

    Node newPriceNode = document.createElement("price");
    Text tpNode = document.createTextNode("13.50");
    newPriceNode.appendChild(tpNode);

    newCoffeeNode.appendChild(newNameNode);
    newCoffeeNode.appendChild(newPriceNode);

    rootNode.insertBefore(newCoffeeNode, thisCoffeeNode);
    break;
  }
```


Outputting XML

- Can generate XML document from a DOM using a Transformer
- Eg suppose coffee processor modified to output results as a new XML document

```
Document document = builder.parse("priceList.xml");
// code that modifies the DOM in here

TransformerFactory transFactory = TransformerFactory.newInstance();
Transformer transformer = transFactory.newTransformer();
DOMSource source = new DOMSource(document);
File newXML = new File("newPriceList.xml");
FileOutputStream fos = new FileOutputStream(newXML);
StreamResult result = new StreamResult(fos);
transformer.transform(source, result);
```



```
<priceList>
  <coffee>
    <name>Kona</name>
    <price>13.50</price>
  </coffee>
  <coffee>
    <name>Mocha Java</name>
    <price>11.95</price>
  </coffee>
  <coffee>
    <name>Sumatra</name>
    <price>12.50</price>
  </coffee>
</priceList>
```

RDBMS views

- Allow database information to be accessed (and sometimes modified) in different forms
- Based on SELECT statement

```
CREATE VIEW titles_view AS
SELECT title, type, price, pubdate FROM titles
```

  - Allows any alternate structure possible through selections, joins, orderings, grouping, and calculations
  - However, to be updatable there are severe restrictions
    - No aggregate functions, grouping, unions, distincts, derived columns (calculations)
    - Insert and update can only reference columns from one table when a join is utilised
    - Delete can only work on views based on one table

RDBMS view example

```
CREATE VIEW publication_view AS
SELECT title, creator AS author, isbn, subject AS classification, description,
  tableOfContents AS contents, cost AS price
  FROM publication
```

```
CREATE VIEW publication_view AS
SELECT title, creator AS author, isbn, subject AS classification, description,
  tableOfContents AS contents, cost/0.5855 AS price
  FROM publication
```


CORBA IDL

- IDL: Interface Description Language
- CORBA IDL is a language-independent interface specification (declarative)
- Consists of modules, interfaces, types (structs, enumerated, ints, reals, strings etc.)
- Also might include exceptions, references to other IDL module specifications
- C++/Java-like syntax, but limited number of types available

IDL Components

- Types
  - Basic types
  - Named types
  - Enumerations
  - Structures
  - Unions
  - Arrays
  - Sequences
  - Recursive structures
- Constants
  - Allow expressions
- Interfaces (are a type)
  - Contain Operations
    - Return result type
    - Operation name
    - Zero or more parameters
      - in, out, inout
- User exceptions
- System exceptions
- Attributes
- Modules
- Forward declarations
- Inheritance

IDL Types Examples

```
typedef long Millimeter;
enum WallTypes { interior, exterior, trombe, underground };
struct WallInfo {
   WallTypes type;
   Millimeter height;
   Millimeter width;
}
union WallAtts switch (WallTypes) {
   case trombe:                        struct Node {
      long glazingArea;                   long value;
   case underground:                      sequence<Node> children;
      Millimeter soilDepth;            };
}
typedef WallInfo RectangularRoom[4];
typedef sequence<WallInfo> GeneralRoom;
```


IDL Interfaces Examples

```
module Building {  // like a Java package
  interface Wall {
    exception Incomplete { string missingAtts };
    // attribute definitions here…
    long wallArea() raises(Incomplete);
    void setHeight(in Millimeter newHeight);
    void setWidth(in Millimeter newWidth);
    …
  }
  interface TrombeWall : Wall {
    void setGlazingArea(in long newArea);
    …
  }
  interface Room {
    boolean fixWalls(inout sequence<Wall> wallPieces);
  }
}
```


XSLT, VIEW, IDL

- Allow for the transformation of data in one representation into a new representation
- Limitations on the types of transforms supported
- XSLT and IDL are uni-directional
- RDBMS VIEW is bi-directional in very constrained circumstances

- What can we do which is better than this?