XML

Paul Fodor

CSE316: Fundamentals of Software Development

Stony Brook University

http://www.cs.stonybrook.edu/~cse316

XML

- XML – eXtensible Markup Language
  - Markup language (no execution semantics!)
  - Designed to describe the structure of data
  - XML documents contain user specified 'tags' (making it look similar to HTML but it is not!)
  - Suitable for semistructured data and has become a standard:
    - Easy to describe object-like data
    - Self-describing
    - Doesn't require a schema (but can be provided optionally)
      - DTDs – an older way to specify schema
      - XML Schema – a newer, more powerful (and much more complex!) way of specifying schema
  - Query and transformation languages:
    - XPath
    - XSLT
    - XQuery

Overview of XML

- Like HTML, but any number of different tags can be used (up to the document author) – extensible
- Unlike HTML, no semantics behind the tags
  - For instance, HTML's <table>…</table> means: render contents as a table; in XML: doesn't mean anything special
  - Some semantics can be specified using XML Schema (types); some using stylesheets (browser rendering)
- Unlike HTML, is intolerant to bugs:
  - Browsers will render buggy HTML pages
  - *XML processors* are not supposed to process buggy XML documents

XML File Format

- XML typically contains UTF-8 text
- Must contain a line providing information on the XML version to use:

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

  - This must be the first line in the file!

XML tags

- Elements must [almost] always have a start and end tag

  <tag> …. </tag>

  - Some elements can be fully self contained and end with a '/>'

  <tag/>

XML Example

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

<PersonList Type="Student"  Date="2002-02-02" >
    <Title  Value="Student List" />
    <Person>
    … … …
    </Person>
    <Person>
    … … …
    </Person>
</PersonList>
```

attributes

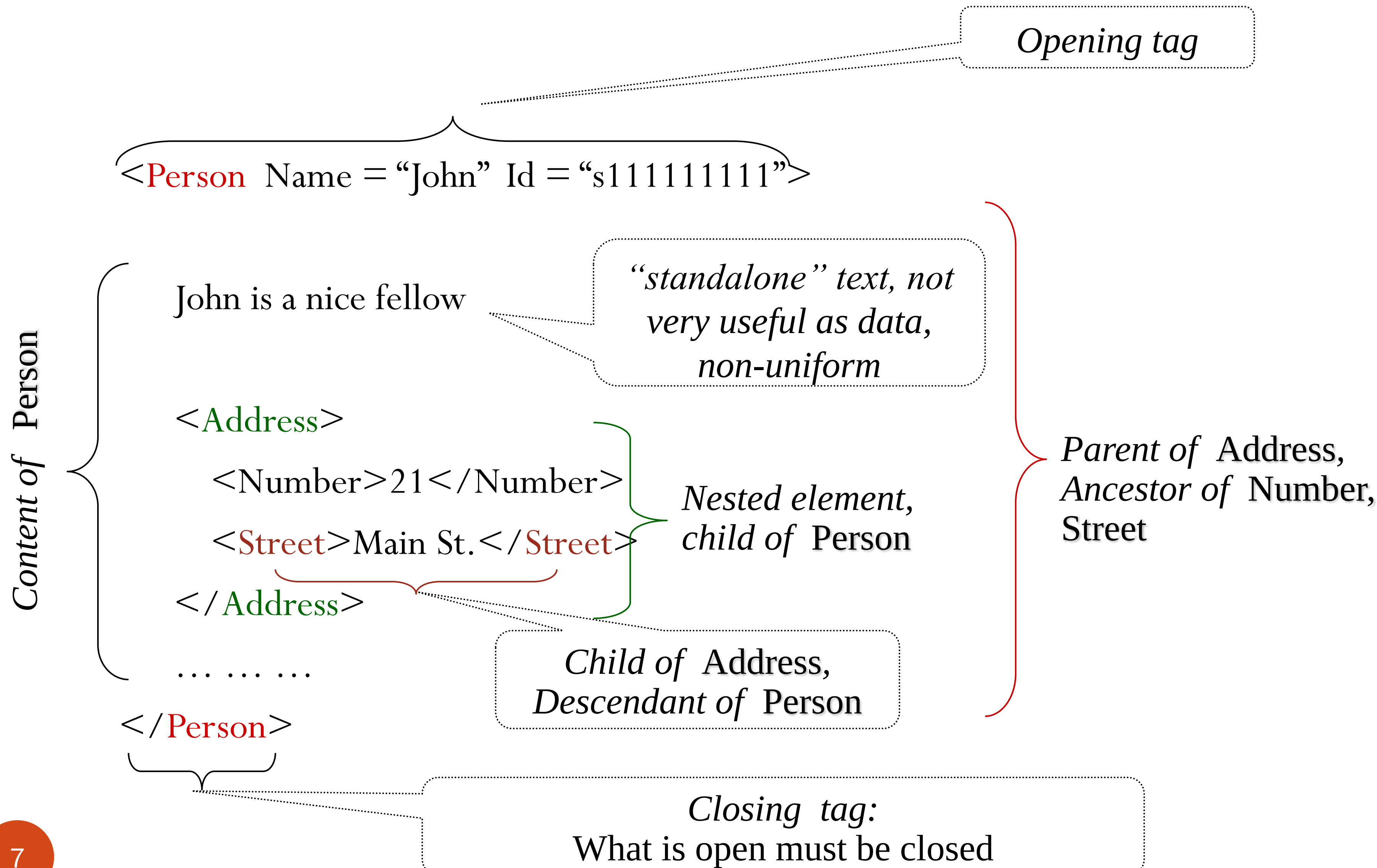
elements

Root element

Element (or tag) names

- Elements are nested
- Root element contains all others

More Terminology



XML : Example - cars

```
<?xml version='1.0' encoding='UTF-8'?>
<?xml-stylesheet type='text/xsl' href='./cars.xsl' ?>
<carlist>
  <car>
    <make>Ford</make>
    <model>Custom</model>
    <year>1969</year>
    <color>Aquamarine</color>
    <engine>V6</engine>
    <transmission>Manual 3-spd, column</transmission>
  </car>
  <car>
    <make>Ford</make>
    <model>E350 Econoline Van</model>
    <year>1997</year>
    <color>Red</color>
    <engine>Triton V10</engine>
    <transmission>Automatic</transmission>
  </car>
  <car>
    <make>Nissan</make>
    <model>Sentra</model>
    <year>1991</year>
    <color>Black</color>
    <engine>V4</engine>
    <transmission>Manual 4spd</transmission>
  </car>
  <car>
    <make>Nissan</make>
    <model>Sentra</model>
    <year>1998</year>
    <color>Metalic Blue</color>
    <engine>V4</engine>
    <transmission>Automatic</transmission>
  </car>
  <car>
    <make>Nissan</make>
    <model>Sentra</model>
    <year>2009</year>
    <color>Metalic Grey</color>
    <engine>V4</engine>
    <transmission>Manual 6 spd</transmission>
  </car>
</carlist>
```

XML – Special characters

- Some characters are used in strings in attributes
  - These avoid to be considered as part of tags in a document
    - & -- &
    - ' -- '
    - > -- >
    - < -- <
    - ' -- "
  - Example:

```
<xsl:if test='year &gt; 1990'>
   <tr>
     <td><xsl:value-of select='make' /></td>
     <td><xsl:value-of select='model' /></td>
    </tr>
</xsl:if>
```

Well-formed XML Documents

- Must have a *root element*
- Every *opening tag* must have matching *closing tag*
- Elements must be *properly nested*
  - <foo><bar></foo></bar> is a no-no
- An *attribute* name can occur *at most once* in an opening tag. If it occurs,
  - It *must have an explicitly specified value* (Boolean attrs, like in HTML, are not allowed)
  - The value *must be quoted* (with " or ')
- *XML processors are not supposed to try and fix ill-formed documents (unlike HTML browsers)*

Identifying and Referencing with Attributes

- An attribute can be declared (in a DTD – see later) to have type:
  - *ID* – unique identifier of an element
    - If attr1 & attr2 are both of type ID, then it is illegal to have <something attr1="*abc*"> … <somethingelse attr2="*abc*"> within the same document
  - *IDREF* – references a unique element with matching ID attribute (in particular, an XML document with IDREFs is not a tree)
    - If attr1 has type ID and attr2 has type IDREF then we can have: <something attr1="*abc*"> … <somethingelse attr2="*abc*">
  - *IDREFS* – a list of references, if attr1 is ID and attr2 is IDREFS, then we can have
    - <something attr1="*abc*">…<somethingelse attr1="*cde*">… <someotherthing attr2="*abc cde*">

Example: Report Document with Cross-References

```
<?xml version="1.0" ?>
<Report Date="2002-12-12">
  <Students>
    <Student StudId="s111111111">
      <Name><First>John</First><Last>Doe</Last></Name>
 <Status>U2</Status>
      <CrsTaken CrsCode="CS308" Semester="F1997" />
      <CrsTaken CrsCode="MAT123" Semester="F1997" />
    </Student>
    <Student StudId="s666666666">
      <Name><First>Joe</First><Last>Public</Last></Name>
 <Status>U3</Status>
      <CrsTaken CrsCode="CS308" Semester="F1994" />
      <CrsTaken CrsCode="MAT123" Semester="F1997" />
    </Student>
    <Student StudId="s987654321">
      <Name><First>Bart</First><Last>Simpson</Last></Name>
 <Status>U4</Status>
      <CrsTaken CrsCode="CS308" Semester="F1994" />
    </Student>
  </Students>
...... continued ... ...
```

ID (callout pointing to StudId)

IDREF (callout pointing to CrsCode)

Report Document

```
<Classes>
  <Class>
    <CrsCode>CS308</CrsCode> <Semester>F1994</Semester>
    <ClassRoster Members="s666666666  s987654321" />
  </Class>
  <Class>
    <CrsCode>CS308</CrsCode> <Semester>F1997</Semester>
    <ClassRoster Members="s111111111" />
  </Class>
  <Class>
    <CrsCode>MAT123</CrsCode> <Semester>F1997</Semester>
    <ClassRoster Members="s111111111  s666666666" />
  </Class>
</Classes>
```

IDREFS

...... *continued* ... ...

Report Document

ID

```
<Courses>
    <Course CrsCode = “CS308” >
        <CrsName>Market Analysis</CrsName>
    </Course>
    <Course CrsCode = “MAT123” >
        <CrsName>Market Analysis</CrsName>
    </Course>
</Courses>
</Report>
```

XML Namespaces

- A mechanism to prevent name clashes between components of same or different documents
- Namespace declaration
  - *Namespace* – a symbol, typically a URL (*doesn't need to point to a real page*)
  - *Prefix* – an abbreviation of the namespace, a convenience; works as an alias
  - Actual name (element or attribute) – *prefix*:*name*
  - Declarations/prefixes have *scope* similarly to begin/end
- Example:

```
<item  xmlns = "http://www.acmeinc.com/jp#supplies"
       xmlns:toy = "http://www.acmeinc.com/jp#toys">
   <name>backpack</name>
   <feature>
      <toy:item><toy:name>cyberpet</toy:name></toy:item>
   </feature>
</item>
```

Default namespace

toy *namespace*

reserved keyword

Namespaces (cont'd.)

- Scopes of declarations are color-coded:

```
<item xmlns="http://www.foo.org/abc"
      xmlns:cde="http://www.bar.com/cde">
   <name>...</name>
   <feature>
      <cde:item><cde:name>...</cde:name><cde:item>
   </feature>
   <item  xmlns="http://www.foobar.org/"
          xmlns:cde="http://www.foobar.org/cde" >
      <name>...</name>
      <cde:name>...</cde:name>
   </item>
</item>
```

New default; overshadows old default

Redeclaration of cde*; overshadows old declaration*

Namespaces (cont'd.)

- xmlns="http://foo.com/bar" ***doesn't*** mean there is a document at this URL: using URLs is just a convenient convention; and a namespace is just an identifier
- Namespaces aren't part of XML 1.0, but all XML processors understand this feature now
- A number of prefixes have become "standard" and some XML processors might understand them without any declaration. E.g.,
  - **xs** for http://www.w3.org/2001/XMLSchema
  - **xsl** for http://www.w3.org/1999/XSL/Transform
  - Etc.

Document Type Definition (DTD)

- A *DTD* is a grammar specification for an XML document
- DTDs are optional – don’t need to be specified
  - If specified, DTD can be part of the document (at the top); or it can be given as a URL
- A document that conforms (i.e., parses) w.r.t. its DTD is said to be *valid*
  - XML processors are not required to check validity, even if DTD is specified
  - But they are required to test well-formedness

DTDs (cont'd)

- DTD specified as part of a document:

```
<?xml  version="1.0" ?>
<!DOCTYPE  Report [
    … … …
]>
<Report> … … … </Report>
```

- DTD specified as a standalone thing

```
<?xml  version="1.0" ?>
<!DOCTYPE  Report  "http://foo.org/Report.dtd">
<Report> … … … </Report>
```

DTD Components

<!ELEMENT *elt-name*

(…*contents*…)/EMPTY/ANY >

Element's contents

<!ATTLIST *elt-name* *attr-name*

An attr for elt

CDATA/ID/IDREF/IDREFS

Type of attribute

#IMPLIED/#REQUIRED

Optional/mandatory

>

- Can define other things, like macros (called *entities* in the XML jargon)

DTD Example

```
<!DOCTYPE  Report [
    <!ELEMENT  Report  (Students, Classes, Courses)>
    <!ELEMENT  Students  (Student*)>
    <!ELEMENT  Classes  (Class*)>
    <!ELEMENT  Courses  (Course*)>
    <!ELEMENT  Student  (Name, Status, CrsTaken*)>
    <!ELEMENT  Name  (First,Last)>
    <!ELEMENT  First  (#PCDATA)>
    … … …
    <!ELEMENT  CrsTaken  EMPTY>
    <!ELEMENT  Class (CrsCode,Semester,ClassRoster)>
    <!ELEMENT  Course  (CrsName)>
    … … …
    <!ATTLIST  Report  Date  CDATA #IMPLIED>
    <!ATTLIST  Student  StudId  ID  #REQUIRED>
    <!ATTLIST  Course  CrsCode  ID  #REQUIRED>
    <!ATTLIST  CrsTaken  CrsCode  IDREF  #REQUIRED>
    <!ATTLIST  ClassRoster  Members  IDREFS  #IMPLIED>
]>
```

Zero or more

Has text content

Empty element, no content

Same attribute in different elements

Limitations of DTDs

- Doesn't understand namespaces
- Very limited assortment of data types (just strings)
- Very weak w.r.t. consistency constraints (ID/IDREF/IDREFS only)
- Can't express unordered contents conveniently
- All element names are global: can't have one Name type for people and another for companies:

```
<!ELEMENT  Name  (Last, First)>
<!ELEMENT  Name  (#PCDATA)>
```

both can't be in the same DTD

XML Schema

- Came to rectify some of the problems with DTDs
- Advantages:
  - Integrated with namespaces
  - Many built-in types
  - User-defined types
  - Has local element names
  - Powerful key and referential constraints
- Disadvantages:
  - Unwieldy – much more complex than DTDs

Schema Document and Namespaces

```
<schema
   xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://xyz.edu/Admin">
     … … …
</schema>
```

- Uses standard XML syntax.
- http://www.w3.org/2001/XMLSchema – namespace for keywords used in a schema document (*not* an instance document), e.g., "*schema*", *targetNamespace*, etc.
- targetNamespace – names the namespace defined by the above schema.

Instance Document

- Report document whose structure is being defined by the earlier schema document

```
<?xml  version = "1.0" ?>
<Report  xmlns="http://xyz.edu/Admin"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xsi:schemaLocation="http://xyz.edu/Admin
                                 http://xyz.edu/Admin.xsd" >
      …
</Report>
```

Default namespace for instance document

Namespace for XML Schema names that occur in instance documents rather than their schemas

Schema namespace

Schema location

- xsi:schemaLocation says: the schema for the namespace http://xyz.edu/Admin is found in http://xyz.edu/Admin.xsd
- Document schema & its location are <u>not binding</u> on the XML processor; it can decide to use another schema, or none at all

Building Schemas from Components

```
<schema  xmlns="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://xyz.edu/Admin" >
    <include  schemaLocation="http://xyz.edu/StudentTypes.xsd">
    <include  schemaLocation="http://xyz.edu/ClassTypes.xsd">
    <include  schemaLocation="http://xyz.edu/CourseTypes.xsd">
    … … …
</schema>
```

- <include…> works like #include in the C language
  - Included schemas must have the same targetNamespace as the including schema
- schemaLocation – tells where to find the piece to be included

Simple Types

- *Primitive types*: *decimal*, *integer*, *Boolean*, *string*, ID, IDREF, etc. (defined in XMLSchema namespace)
- *Type constructors*: *list* and *union*
  - A simple way to derive types from primitive types (disregard the namespaces for now):

```
<simpleType name="myIntList">
    <list itemType="integer" />
</simpleType>

<simpleType name="phoneNumber" >
    <union memberTypes="phone7digits phone10digits" />
</simpleType>
```

Deriving Simple Types by Restriction

```
<simpleType name="phone7digits">
   <restriction base="integer">
         <minInclusive value="1000000"/>
         <maxInclusive value="9999999"/>
    </restriction>
</simpleType>
<simpleType name="emergencyNumbers">
   <restriction base="integer">
         <enumeration value="911"/>
         <enumeration value="333"/>
   </restriction>
</simpleType>
```

- Has more type-building primitives (see textbook and specs)

Some Simple Types Used in the Report Document

```
<simpleType name="studentId" >
        <restriction base="ID" >
                <pattern value="s[0-9]{9}" />
        </restriction>
</simpleType>
<simpleType name="studentIds" >
        <list itemType="adm:studentRef" />
</simpleType>

<simpleType name="studentRef" >
        <restriction base="IDREF" >
                <pattern value="s[0-9]{9}" />
        </restriction>
</simpleType>
```

targetNamespace = http://xyz.edu/Admin
xmlns:adm= http://xyz.edu/Admin

XML ID types always start with a letter

Prefix for the target namespace

Simple Types for Report Document (contd.)

```
<simpleType name="courseCode" >
        <restriction base="ID" >
                <pattern value="[A-Z]{3}[0-9]{3}" />
        </restriction>
</simpleType>
<simpleType name="courseRef" >
        <restriction base="IDREF" >
                <pattern value="[A-Z]{3}[0-9]{3}" />
        </restriction>
</simpleType>
<simpleType name="studentStatus" >
        <restriction base="string" >
                <enumeration value="U1" />
                … … …
                <enumeration value="G5" />
        </restriction>
</simpleType>
```

Schema Document That Defines Simple Types

```
<schema xmlns=“http://www.w3.org/2001/XMLSchema”
        xmlns:adm=“http://xyz.edu/Admin”
        targetNamespace=“http://xyz.edu/Admin”>
  … … …
  <element name=“CrsName” type=“string”/>
  <element name=“Status” type=“adm:studentStatus” />
  … … …
  <simpleType name=“studentStatus” >
      … … …
  </simpleType>
</schema>
```

element declaration using derived type

Complex Types

- Allows the definition of element types that have complex internal structure
- Similar to class definitions in object-oriented databases
  - Very verbose syntax
  - Can define both child elements and attributes
  - Supports ordered and unordered collections of elements

Example: studentType

```
<element name="Student" type="adm:studentType" />
<complexType name="studentType" >
    <sequence>
        <element name="Name" type="adm:personNameType" />
        <element name="Status" type="adm:studentStatus" />
        <element name="CrsTaken" type="adm:courseTakenType"
                minOccurs="0" maxOccurs="unbounded" />
    </sequence>
    <attribute name="StudId" type="adm:studentId" />
</complexType>
<complexType name="personNameType" >
    <sequence>
        <element name="First" type="string" />
        <element name="Last" type="string" />
    </sequence>
</complexType>
```

Compositors: Sequences, Sets, Alternatives

- *Compositors*:
  - *sequence*, *all*, *choice* are required when element has at least 1 child element (= *complex content*)
- sequence
- all – can specify sets of elements
- choice – can specify alternative types

Sets

- Suppose the order of components in addresses is unimportant:

```
<complexType name="addressType" >
        <all>
                <element name="StreetName" type="string" />
                <element name="StreetNumber" type="string" />
                <element name="City" type="string" />
        </all>
</complexType>
```

- *Problem*: all comes with a host of awkward restrictions. For instance, cannot occur inside a sequence; only sets of elements, not bags.

Alternative Types

- Assume addresses can have P.O.Box or street name/number:

```
<complexType name="addressType">
    <sequence>
        <choice>
            <element name="POBox" type="string" />
            <sequence>
                <element name="Name" type="string" />
                <element name="Number" type="string" />
            </sequence>
        </choice>
        <element name="City" type="string" />
    </sequence>
</complexType>
```

This or that

Local Element Names

- A DTD can define only global element name:
  - Can have at most one <!ELEMENT foo …> statement per DTD
- In XML Schema, names have scope like in programming languages – the nearest containing complexType definition
  - Thus, can have the same element name (e.g., *Name),* within different types and with different internal structures

Local Element Names: Example

```
<complexType name="studentType">
    <sequence>
        <element name="Name" type="adm:personNameType" />
        <element name="Status" type="adm:studentStatus" />
        <element name="CrsTaken" type="adm:courseTakenType"
                         minOccurs="0" maxOccurs="unbounded" />
    </sequence>
    <attribute name="StudId" type="adm:studentId" />
</complexType>
<complexType name="courseType">
    <sequence>
        <element name="Name" type="string" />
    </sequence>
    <attribute name="CrsCode" type="adm:courseCode" />
</complexType>
```

Same element name,
different types,
inside different complex types

Importing XML Schemas

- Import is used to share schemas developed by different groups at different sites
- Include vs. import:
  - *Include*:
    - Included schemas are usually under the control of the same development group as the including schema
    - Included and including schemas must have the same target namespace (because the text is physically included)
    - schemaLocation attribute required
  - *Import*:
    - Schemas are under the control of different groups
    - Target namespaces are different
    - The import statement must tell the importing schema what that target namespace is
    - schemaLocation attribute optional

Import of Schemas (cont'd)

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
          targetNamespace="http://xyz.edu/Admin"
          xmlns:reg="http://xyz.edu/Registrar"
          xmlns:crs="http://xyz.edu/Courses" >
    <import namespace="http://xyz.edu/Registrar"
            schemaLocation="http://xyz.edu/Registrar/StudentType.xsd" />
    <import namespace="http://xyz.edu/Courses" />
        … … …
        … … …
</schema>
```

Prefix declarations for imported namespaces

required

optional

Extension and Restriction of Base Types

- Mechanism for modifying the types in imported schemas
- Similar to subclassing in object-oriented languages
- *Extending* an XML Schema type means adding elements or adding attributes to existing elements
- *Restricting* types means tightening the types of the existing elements and attributes (i.e., replacing existing types with subtypes)

Type Extension: Example

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:xyzCrs="http://xyz.edu/Courses"
        xmlns:fooAdm="http://foo.edu/Admin"
        targetNamespace="http://foo.edu/Admin" >
    <import namespace="http://xyz.edu/Courses" />

    <complexType name="courseType" >
        <complexContent>
            <extension base="xyzCrs:courseType">
                <element name="syllabus" type="string" />
            </extension>
        </complexContent>
    </complexType>
    <element name="Course" type="fooAdm:courseType" />
    … … …
</schema>
```

Extends by adding

Defined

Used

Type Restriction: Example

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
            xmlns:xyzCrs="http://xyz.edu/Courses"
            xmlns:fooAdm="http://foo.edu/Admin"
            targetNamespace="http://foo.edu/Admin" >
    <import namespace="http://xyz.edu/Courses" />

    <complexType name="studentType" >
          <complexContent>
             <restriction base="xyzCrs:studentType" >
                   <sequence>
                      <element name="Name" type="xyzCrs:personNameType" />
                      <element name="Status" type="xyzCrs:studentStatus" />
                      <element name="CrsTaken" type="xyzCrs:courseTakenType"
                                    minOccurs="0" maxOccurs="60" />
                   </sequence>
                   <attribute name="StudId" type="xyzCrs:studentId" />
             </restriction>
          </complexContent>
    </complexType>
    <element name="Student" type="fooAdm:studentType" />
```

Must repeat the original definition

Tightened type: the original was "unbounded"

Structure of an XML Schema Document

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:adm="http://xyz.edu/Admin"
        targetNamespace="http://xyz.edu/Admin">
   <element name="Report" type="adm:reportType" />

   <complexType name="reportType" >
       … … …
   </complexType>
   <complexType name=… >
       … … …
   </complexType>
   … … …
</schema>
```

Root type

Root element

Definition of root type

Definition of types mentioned in the root type; Types can also be included or imported

Anonymous Types

- So far all types were *named*
  - Useful when the same type is used in more than one place
- When a type definition is used exactly once, *anonymous* types can save space

"element" *used to be empty element – now isn't*

No type name

```
<element name="Report">
  <complexType>
      <sequence>
         <element name="Students" type="adm:studentList" />
         <element name="Classes" type="adm:classOfferings" />
         <element name="Courses" type="adm:courseCatalog" />
      </sequence>
  </complexType>
</element>
```

Integrity Constraints in XML Schema

- A DTD can specify only very simple kinds of key and referential constraint; only using attributes
- XML Schema also has ID, IDREF as primitive data types, but these can also be used to type elements, not just attributes
- In addition, XML Schema can express complex key and foreign key constraints

Schema Keys

- A *key* in an XML document is a sequence of components, which might include elements and attributes, which uniquely identifies document components in a *source collection* of objects in the document
- *Issues*:
  - Need to be able to identify that source collection
  - Need to be able to tell which sequences form the key
- For this, XML Schema uses *XPath* – a simple XML query language.

Basic XPath – for Key Specification

```
<Offerings>        -- current reference point
   <Offering>
       <CrsCode Section="1">CS532</CrsCode>

       <Semester><Term>Spring</Term><Year>2002</Year></Semest
   er>
   </Offering>
   <Offering>
       <CrsCode Section="2">CS305</CrsCode>
       <Semester><Term>Fall</Term><Year>2002</Year></Semester>
   </Offering>
</Offerings>
```

Offering/CrsCode/@Section – *selects occurrences of attribute* Section *within* CrsCode *within* Offerings

Offering/CrsCode – *selects all* CrsCode *element occurrences within* Offerings

Offering/Semester/Term –*all* Term *elements within* Semester *within* Offerings

Offering/Semester/Year –*all* Year *elements within* Semester *within* Offerings

Keys: Example

```
<complexType name="reportType">
  <sequence>
    <element name="Students" … />
    <element name="Classes" >
      <complexType>
        <sequence>
          <element name="Class" minOccurs="0" maxOccurs="unbounded" >
            <sequence>
              <element name="CrsCode" … />
              <element name="Semester" … />
              <element name="ClassRoster" … />
            </sequence>
          </element>
        </sequence>
      </complexType>
      … … key specification goes here – next slide … …
    </element>
    <element name="Courses" … />
  <sequence>
</complexType>
```

Example (cont'd)

- A key specification:

```
<key name="PrimaryKeyForClass">
    <selector xpath="Class" />

    <field xpath="CrsCode" />
    <field xpath="Semester" />
</key>
```

Defines ***source collection*** *of objects to which the key applies. The XPath expression is relative to element to which the key is local*

Fields that form the key.
The XPath expression is relative to the source collection of objects specified in selector.
So, CrsCode *is actually* Classes/Class/CrsCode

field *must return exactly one value per object specified by* selector

Foreign Keys

- Like the REFERENCES clause in SQL, but more involved
- Need to specify:
  - *Foreign key*:
    - *Source collection* of objects
    - Fields that form the foreign key
  - *Target key*:
    - A previously defined *key* (or *unique*) specification, which is comprised of:
      - *Target collection* of objects
      - Sequence of fields that comprise the key

Foreign Key: Example

```
<keyref name="NoEmptyClasses" refer="adm:PrimaryKeyForClass" >
    <selector xpath="Student/CrsTaken" />
    <field xpath="@CrsCode" />
    <field xpath="@Semester" />
</keyref>
```

Target key

Source collection

Fields of the foreign key. XPath expressions are relative to the source collection

XML Query Languages

- XPath – core query language. Very limited, a selection operator. Very useful, though: used in XML Schema, XSLT, XQuery, many other XML standards
- XSLT – a functional style document transformation language. Very powerful, very complicated
- XQuery – W3C standard. Very powerful, fairly intuitive, SQL-style.
- SQL/XML – attempt to marry SQL and XML, part of SQL:2003.

Why Query XML?

- Need to extract parts of XML documents
- Need to transform documents into different forms
- Need to relate – join – parts of the same or different documents

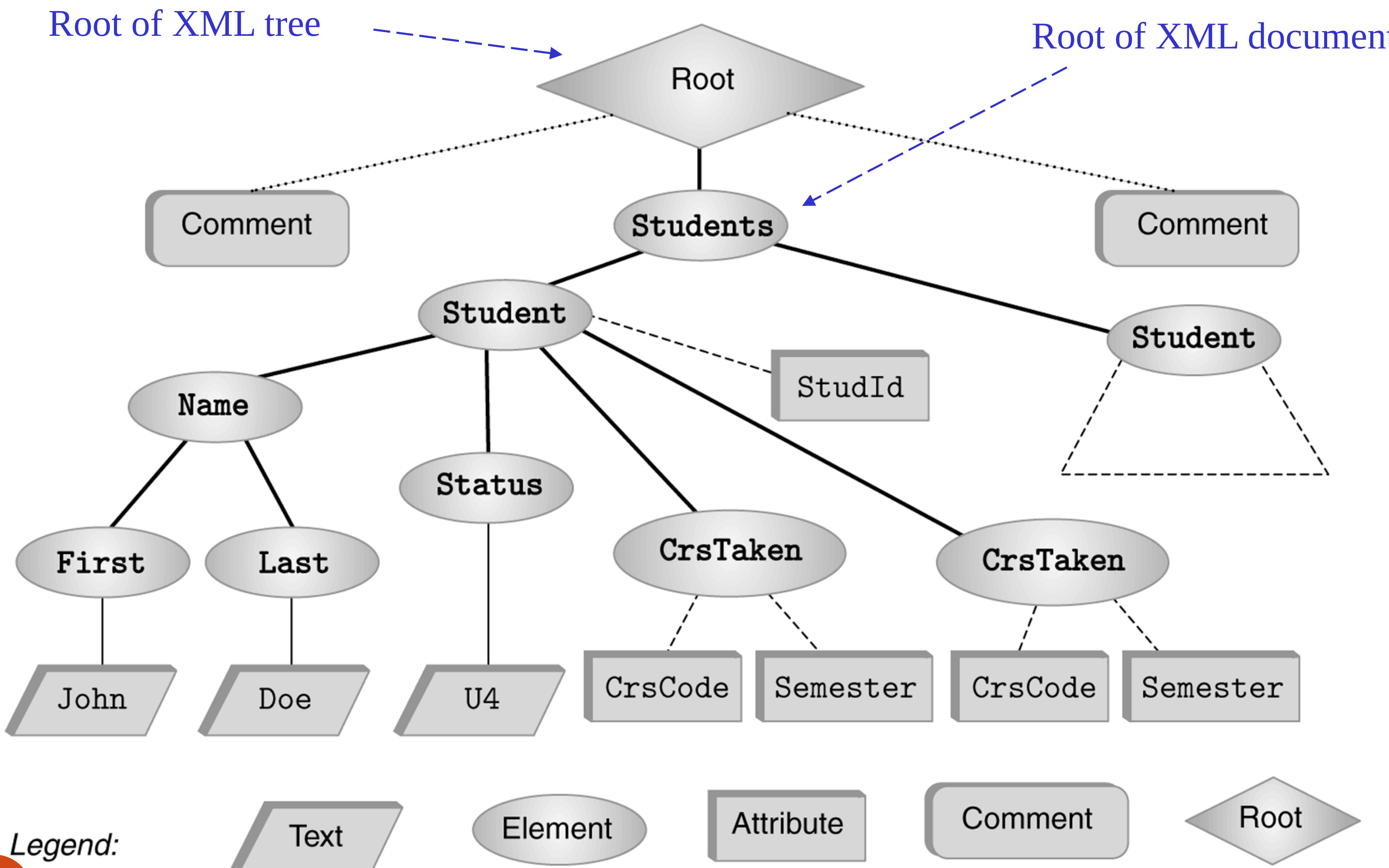
XPath

- Analogous to path expressions in object-oriented languages (e.g., OQL)
- Extends path expressions with query facility
- XPath views an XML document as a tree
  - Root of the tree is a ***new*** node, which doesn't correspond to anything in the document
  - Internal nodes are elements
  - Leaves are either
    - Attributes
    - Text nodes
    - Comments
    - Other: processing instructions, etc.

XML Example

```
<?xml  version="1.0" ?>
<!-- Some  comment -->
<Students>
        <Student StudId="111111111" >
                <Name><First>John</First><Last>Doe</Last></Name>
                <Status>U2</Status>
                <CrsTaken  CrsCode="CS308"  Semester="F1997" />
                <CrsTaken  CrsCode="MAT123"  Semester="F1997" />
        </Student>
        <Student StudId="987654321" >
                <Name><First>Bart</First><Last>Simpson</Last></Name>
                <Status>U4</Status>
                <CrsTaken  CrsCode="CS308"  Semester="F1994" />
        </Student>
</Students>
```

XPath Document Tree



Terminology

- *Parent/child* nodes, as usual
- Child nodes (that are of interest to us) are of types *text*, *element*, *attribute*
  - We call them *t-children*, *e-children*, *a-children*
  - Also, *et-children* are child-nodes that are either elements or text, *ea-children* are child nodes that are either elements or attributes, etc.
- Ancestor/descendant nodes – as usual in trees

XPath Basics

- An XPath expression takes a document tree as input and returns a multi-set of nodes of the tree
- Expressions that *start* with / are *absolute path expressions*
  - Expression / – returns root node of XPath tree
  - /Students/Student – returns all Student-elements that are children of Students elements, which in turn must be children of the root
  - /Student – returns empty set (no such children at root)

XPath Basics (cont'd)

- *Current* (or *context* node) – exists during the evaluation of XPath expressions (and in other XML query languages)
- . – denotes the current node; .. – denotes the parent
  - foo/bar – returns all bar-elements that are children of foo nodes, which in turn are children of the current node
  - ./foo/bar – same
  - ../abc/cde – all cde e-children of abc e-children of the <u>parent</u> of the current node
- Expressions that don't start with / are *relative* (to the current node)

Attributes, Text, etc.

Denotes an attribute

- /Students/Student/@StudentId – returns all StudentId a-children of Student, which are e-children of Students, which are children of the root
- /Students/Student/Name/Last/text( ) – returns all t-children of Last e-children of …
- /comment( ) – returns comment nodes under root

Overall Idea and Semantics

> This is called *full* syntax.
> We used *abbreviated* syntax before.
> Full syntax is better for describing meaning. Abbreviated syntax is better for programming.

- An XPath expression is:
  locationStep1/locationStep2/ …
- *Location step*:
  Axis::nodeSelector[predicate]
- Navigation *axis*:
  - *child, parent*
  - *ancestor, descendant, ancestor-or-self, descendant-or-self*
- *Node selector*: node name or wildcard; e.g.,
  - ./child::Student (we used ./Student, which is an abbreviation)
  - ./child::* – any e-child (abbreviation: ./*)
- *Predicate*: a selection condition; e.g.,
  Students/Student[CourseTaken/@CrsCode = "CS532"]

XPath Semantics

- The meaning of the expression locationStep1/locationStep2/… is the set of all document nodes obtained as follows:
  - Find all nodes reachable by locationStep1 from the current node
  - For each node *N* in the result, find all nodes reachable from *N* by locationStep2; take the union of all these nodes
  - For each node in the result, find all nodes reachable by locationStep3, etc.
  - The value of the path expression on a document is the set of all document nodes found after processing the last location step in the expression

Algorithm

- locationStep1/locationStep2/…:
  - Find all nodes specified by locationStep1
  - For each such node N:
    - Find all nodes specified by locationStep2 using N as the current node
    - Take union
  - For each node returned by locationStep2 do the same
- locationStep = axis::node[predicate]
  - Find all nodes specified by axis::node
  - Select only those that satisfy predicate

Navigation Primitives

- 2^{nd} CrsTaken child of 1^{st} Student child of Students:

  /Students/Student[1]/CrsTaken[2]

- All last CourseTaken elements within each Student element:

  /Students/Student/CrsTaken[last( )]

Wildcards

- Wildcards are useful when the exact structure of document is not known
- *Descendant-or-self* axis, // : allows to descend down any number of levels (including 0)
  - //CrsTaken – all CrsTaken nodes under the root
  - Students//@Name – all Name attribute nodes under the elements Students, who are children of the current node
- The * wildcard:
  - * – any element: Student/*/text()
  - @* – any attribute: Students//@*

XPath Queries (selection predicates)

- Location step = Axis::nodeSelector[predicate]
- Predicate:
  - XPath expression = const | built-in function | XPath expression
  - XPath expression
  - built-in predicate
  - a Boolean combination thereof
- Axis::nodeSelector[predicate] $\subseteq$ Axis::nodeSelector but contains only the nodes that satisfy predicate
- Built-in predicate: special predicates for string matching, set manipulation, etc.
- Built-in function: large assortment of functions for string manipulation, aggregation, etc.

XPath Queries – Examples

- Students who have taken CSE532:

  //Student[CrsTaken/@CrsCode="CSE532"]

  True if : "CSE532" ∈ //Student/CrsTaken/@CrsCode

- Complex example:

  //Student[Status="U3" and starts-with(.//Last, "A")
  and contains(.//@CrsCode, "ESE")
  and not(.//Last = .//First)]

- Aggregation: sum( ), count( )

  //Student[sum(.//@Grade) div count(.//@Grade) > 3.5]

Xpath Queries (cont'd)

- Testing whether a subnode exists:
  - //Student[CrsTaken/@Grade] – students who have a grade (for some course)
  - //Student[Name/First **or** CrsTaken/@Semester **or** Status/text() = "U4"] – students who have either a first name or have taken a course in some semester or have status U4
- Union operator, | :

  //CrsTaken[@Semester="F2001"] | //Class[Semester="F1990"]

  - union lets us define *heterogeneous* collections of nodes

XPointer

- XPointer = URL + XPath
- Syntax:

  url # xpointer (XPathExpr1) xpointer (XPathExpr2) …

  - Follow *url*
  - Compute XPathExpr1
    - Result non-empty? – return result
    - Else: compute XPathExpr2; and so on
- Example: you might click on a link and run a query against your Registrar's database

  http://yours.edu/Report.xml#xpointer(
  //Student[CrsTaken/@CrsCode="CS532"
  and CrsTaken/@Semester="S2012"])

XLS: XML Stylesheets

- Powerful programming language, uses *functional programming paradigm*
  - Used to describe how to style data from an XML document
- Originally designed as a stylesheet language: this is what “S”, “L”, and “T” stand for
  - The idea was to use it to display XML documents by transforming them into HTML

XML – Stylesheets - Transformations

- XSL Transformations use XPATH to find data and then convert the input data to some output format
- XSL documents should start with the following boilerplate code:

```
<?xml version='1.0' encoding='UTF-8'?>
<xsl:stylesheet version='1.0' xmlns:xsl='http://www.w3.org/1999/XSL/Transform'>
```

Stylesheets - Examples

- Here is part of an XML file holding information on cars

```
<?xml version='1.0' encoding='UTF-8'?>
<?xml-stylesheet type='text/xsl' href='./cars.xsl' ?>

<carlist>
  <car>
    <make>Ford</make>
    <model>Custom</model>
    <year>1969</year>
    <color>Aquamarine</color>
    <engine>V6</engine>
    <transmission>Manual 3-spd, column</transmission>
  </car>
  <car>
      …
</carlist>
```

This references the stylesheet file to use
When loading this XML file

This is tags and data
designed by the user

XSL Selection/Flow-of-control

- XSL functions for selection and repetition:
  - **<xsl:for-each >** – repeats output generation for each copy of a specific tag
  - **<xsl:if > -** uses a test condition to determine whether to expand output
  - **<xsl:choose > -** uses a value test to determine which of several output templates to expand
  - **<xsl:sort >** - Sorts data by a specific tagged field
  - **generate-id()** – Generates ids that can be used as anchors and jump targets

XSL <xsl:stylesheet >

- Other than leading <?xml …> tag, this encapsulates the entire stylesheet
- Example:

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

...
</xsl:stylesheet>
```

- This holds the version of XSL as well as a URL of the information page on XSL Transform at W3C

XLS <xsl:output >

- This tag indicates the output format
- Example:

```
<xsl:output method='html' />
```

- The html method is the default and technically, this output statement is not needed for html.
  - However, it is good style to include it

XLS <xsl:template >

- This tag describes a section where data will be substituted and output generated
- Example:

```
<xsl:template match='/'>

<xsl:for-each select='carlist/car'>
<xsl:value-of select='make'/>,
<xsl:value-of select='model'/>
</xsl:for-each>

</xsl:template>
```

The 'match' parameter indicates the starting tag of the tree which the template searches. Here it is the entire XML document from the root '/'

XSL <xsl:value-of >

- This extracts a value from a specific data item
- Example:

This extracts the 'make' element from the current enclosing element

<p>Car: <i><xsl:value-of select=*'make'*/>,
<xsl:value-of select=*'model'/></i>.</p>*

This selects the 'model' element from the current enclosing element

The two data items are enclosed in <i> and <b> html tags

XSL <xsl:text >

- This allows embedding of literal text inside a document's output
- Example:

<p>Some misc text<xsl:text>   :   </xsl:text>More text.*</p>*

Despite html markup, this adds exactly 3 spaces, a colon and 3 more spaces.

XSL <xsl:for-each >

- This tag iterates through a collection of like tags and generates output for each based on the enclosing template code
- Example:

```
<xsl:for-each select='carlist/car'>
   <h3><a name='{generate-id(model)}'>
          <xsl:value-of select='model' /></a></h3>
      <xsl:value-of select='current()'/>
      <p>Car: <i><b>
          <xsl:value-of select='make'/>,
          <xsl:value-of select='model'/></b></i>. Built in
          <xsl:value-of select='year'/>. It had (a/an)
          <xsl:value-of select='transmission'/> transmission and a
         <xsl:value-of select='engine'/> engine.
       </p><br />

</xsl:for-each>
```

Current() selects all the elements in the current <car> stanza

Each of these are from the current element inside <carlist><car> … </car>

XSL <xsl:if >

- This can be used to restrict which stanzas are displayed
- Example:

This test restricts the car stanzas being displayed to those with a year greater than 1990

```
<xsl:if test='year &gt; 1990'>
    <tr>
      <td><xsl:value-of select='make' /></td>
      <td><xsl:value-of select='model' /></td>
     </tr>
</xsl:if>
```

XSL <xsl:choose > <xsl:when > <xsl:otherwise>

- This tag set allows simulation of a 'case' or 'switch' statement
- Example:

```
<xsl:choose>
    <xsl:when test='year &gt; 2000'>
        // stuff
    </xsl:when>
    <xsl:when test='year &lt; 2000'>
        // other stuff
    </xsl:when>
    <xsl:otherwise>
        // And now for something completely different!
    </xsl:otherwise>
</xsl:choose>
```

This is the default in case no other condition matches.

XSL <xsl:sort >

- This tag allows you to sort the stanzas based on a specific value within the stanza
- Example:

```
<xsl:sort select='year' order='ascending' data-type='number' />
```

- This sorts the car data by ascending manufacture year

XSL generate-id

- This is a function that allows you to generate anchor ids for various keywords
- Example:

```
<xsl:for-each select='carlist/car'>
    <a href='#{generate-id(model)}'><xsl:value-of select='model' /></a><br />
</xsl:for-each>
```

Note that this goes in its own <for-each > tag set!

```
<?xml version='1.0' encoding='UTF-8'?>
<xsl:stylesheet version='1.0' xmlns:xsl='http://www.w3.org/1999/XSL/Transform'>
<xsl:output method='html' />
<xsl:template match='/'>
<html>
  <head>
    <title>My Cars</title>
  </head>
  <body>
    Version: <xsl:value-of select='system-property('xsl:version')' /><br />
    Vendor: <xsl:value-of select='system-property('xsl:vendor')' /><br />
    Vendor URL: <xsl:value-of select='system-property('xsl:vendor-url')' /><br />
   <xsl:for-each select='carlist/car'>
       <a href='#{generate-id(model)}'>
         <xsl:value-of select='model' /></a><br />
     </xsl:for-each>
```

```
<xsl:for-each select='carlist/car'>
        <xsl:sort select='year' order='descending' data-type='number' />
    <h3><a name='{generate-id(model)}'>
    <xsl:value-of select='model' /></a></h3>
     <xsl:value-of select='current()'/>
      <p>Car: <i><b>
          <xsl:value-of select='make'/>,
          <xsl:value-of select='model'/></b></i>. Built in
          <xsl:value-of select='year'/>. It had (a/an)
          <xsl:value-of select='transmission'/> transmission and a
          <xsl:value-of select='engine'/> engine.
       </p><br />
    </xsl:for-each>
  </body>
 </html>
</xsl:template>
</xsl:stylesheet>
```

file:///C:/Users/cars.xml

Version: 1
Vendor: Microsoft
Vendor URL: http://www.microsoft.com
Custom
E350 Econoline Van
Sentra
Sentra
Sentra

Sentra

Nissan Sentra 2009 Metalic Grey V4 Manual 6 spd

Car: ***Nissan, Sentra***. Built in 2009. It had (a/an) Manual 6 spd transmission and a V4 engine.

Sentra

Nissan Sentra 1998 Metalic Blue V4 Automatic

Car: ***Nissan, Sentra***. Built in 1998. It had (a/an) Automatic transmission and a V4 engine.

E350 Econoline Van

Ford E350 Econoline Van 1997 Red Triton V10 Automatic

Car: ***Ford, E350 Econoline Van***. Built in 1997. It had (a/an) Automatic transmission and a Triton V10 engine.

XSL Utility Functions

- **translate()** – Converts characters from one set to those in a different set
- **round()** – Does standard mathematical rounding
- **floor()** – Converts to nearest whole number below given value
- **ceil()** – Converts to nearest whole number above given value
- **position()** – Returns position of the element within a list of like elements
- **last()** – Returns the number of the last element in a list (so the count of elements)
- **format-number()** – Formats a numerical value
- **substring-before()** – Extracts text before a given character
- **contains()** – Determines if one string is contained inside another
- **sum()** – Add numeric values from a set of elements
- **count()** – Count the number of elements (of same name)

XSL

- Example:

```
Math stuff: <br />
Round PI : <xsl:value-of select='round(3.14)' /><br />
Floor PI : <xsl:value-of select='floor(3.14)' /><br />
Ceiling PI : <xsl:value-of select='ceiling(3.14)' /><br />
```

Round PI : 3
Floor PI : 3
Ceiling PI : 4

XSL

- Example:

```
<xsl:value-of select='position()' />.
<xsl:value-of select='translate(make, 'abcdefghijklmnopqrstuvwxyz','ABCDEFGHIJKLMNOPQRSTUVWXYZ')' />
<xsl:text>  :  </xsl:text>
<xsl:value-of select='translate(model, 'abcdefghijklmnopqrstuvwxyz','ABCDEFGHIJKLMNOPQRSTUVWXYZ')' />
<br />
Node position: <xsl:value-of select='position()' /> out of <xsl:value-of select='last()' /><br /><br />
```

1. FORD : CUSTOM

Node position: 1 out of 5

XSL

- Example:

Year: <xsl:value-of select='year' /> when average year is
<xsl:value-of select='format-number(sum(/carlist/car/year) div count(/carlist/car), '####.##')' />

Year: 1969 when average year is 1992.8

XQuery – XML Query Language

- Integrates XPath with earlier proposed query languages: XQL, XML-QL
- SQL-style, not functional-style
- Much easier to use as a query language than XSLT
- Can do pretty much the same things as XSLT amd more, but typically easier
- 2004: XQuery 1.0

Consider transcript.xml

```
<Transcripts>
   <Transcript>
         <Student StudId="111111111" Name="John Doe" />
         <CrsTaken  CrsCode="CS308"  Semester="F1997"  Grade="B" />
         <CrsTaken  CrsCode="MAT123"  Semester="F1997"  Grade="B" />
         <CrsTaken  CrsCode="EE101"  Semester="F1997"  Grade="A" />
         <CrsTaken  CrsCode="CS305"  Semester="F1995"  Grade="A" />
   </Transcript>
   <Transcript>
         <Student StudId="987654321"  Name="Bart Simpson" />
         <CrsTaken  CrsCode="CS305"  Semester="F1995"  Grade="C" />
         <CrsTaken  CrsCode="CS308"  Semester="F1994"  Grade="B" />
   </Transcript>
```

… … *cont'd* … …

transcript.xml (cont'd)

<Transcript>

<Student StudId="123454321" Name="Joe Blow" />

<CrsTaken CrsCode="CS315" Semester="S1997" Grade="A" />

<CrsTaken CrsCode="CS305" Semester="S1996" Grade="A" />

<CrsTaken CrsCode="MAT123" Semester="S1996" Grade="C" />

</Transcript>

<Transcript>

<Student StudId="023456789" Name="Homer Simpson" />

<CrsTaken CrsCode="EE101" Semester="F1995" Grade="B" />

<CrsTaken CrsCode="CS305" Semester="S1996" Grade="A" />

</Transcript>

</Transcripts>

XQuery Basics

- General structure:

  FOR *variable declarations*
  WHERE *condition*
  RETURN *document*

  } *XQuery expression*

- Example:

  (: *students who took* MAT123 :) — *comment*

```
FOR $t IN doc("http://xyz.edu/transcript.xml")//Transcript
WHERE $t/CrsTaken/@CrsCode = "MAT123"
RETURN $t/Student
```

- Result:

```
<Student StudId="111111111" Name="John Doe" />
<Student StudId="123454321" Name="Joe Blow" />
```

XQuery Basics (cont'd)

- Previous query doesn't produce a well-formed XML document; the following does:

```
<StudentList>
{
    FOR  $t IN  doc("transcript.xml")//Transcript
    WHERE  $t/CrsTaken/@CrsCode = "MAT123"
    RETURN  $t/Student
}
</StudentList>
```

Query inside XML

- FOR binds $t to Transcript elements one by one, filters using WHERE, then places Student-children as *e*-children of StudentList using RETURN

Consider transcript.xml

```
<Transcripts>
  <Transcript>
      <Student StudId="111111111" Name="John Doe" />
      <CrsTaken CrsCode="CS308" Semester="F1997" Grade="B" />
      <CrsTaken CrsCode="MAT123" Semester="F1997" Grade="B" />
      <CrsTaken CrsCode="EE101" Semester="F1997" Grade="A" />
      <CrsTaken CrsCode="CS305" Semester="F1995" Grade="A" />
  </Transcript>
  <Transcript>
      <Student StudId="987654321" Name="Bart Simpson" />
      <CrsTaken CrsCode="CS305" Semester="F1995" Grade="C" />
      <CrsTaken CrsCode="CS308" Semester="F1994" Grade="B" />
  </Transcript>
```

… … *cont'd* … …

transcript.xml (cont'd)

```
<Transcript>
    <Student StudId="123454321" Name="Joe Blow" />
    <CrsTaken CrsCode="CS315" Semester="S1997" Grade="A" />
    <CrsTaken CrsCode="CS305" Semester="S1996" Grade="A" />
    <CrsTaken CrsCode="MAT123" Semester="S1996" Grade="C" />
</Transcript>
<Transcript>
    <Student StudId="023456789" Name="Homer Simpson" />
    <CrsTaken CrsCode="EE101" Semester="F1995" Grade="B" />
    <CrsTaken CrsCode="CS305" Semester="S1996" Grade="A" />
</Transcript>

</Transcripts>
```

Result

<StudentList>

<Student StudId="111111111" Name="John Doe" />

<Student StudId="123454321" Name="Joe Blow" />

<StudentList>

Document Restructuring with XQuery

- *Reconstruct lists of students taking each class using the* Transcript *records*:

```
FOR $c IN distinct-values(doc("transcript.xml")//CrsTaken)
RETURN
   <ClassRoster  CrsCode = {$c/@CrsCode}
                Semester = {$c/@Semester}>
   {
       FOR $t IN doc("transcript.xml")//Transcript
       WHERE  $t/CrsTaken/[@CrsCode = $c/@CrsCode  and
                           @Semester = $c/@Semester]
       RETURN  $t/Student
                ORDER BY $t/Student/@StudId
   }
   </ClassRoster>
   ORDER BY  $c/@CrsCode
```

Query inside RETURN – similar to query inside SELECT

Consider transcript.xml

```
<Transcripts>
  <Transcript>
      <Student StudId=“111111111” Name=“John Doe” />
      <CrsTaken CrsCode=“CS308” Semester=“F1997” Grade=“B” />
      <CrsTaken CrsCode=“MAT123” Semester=“F1997” Grade=“B” />
      <CrsTaken CrsCode=“EE101” Semester=“F1997” Grade=“A” />
      <CrsTaken CrsCode=“CS305” Semester=“F1995” Grade=“A” />
  </Transcript>
  <Transcript>
      <Student StudId=“987654321” Name=“Bart Simpson” />
      <CrsTaken CrsCode=“CS305” Semester=“F1995” Grade=“C” />
      <CrsTaken CrsCode=“CS308” Semester=“F1994” Grade=“B” />
  </Transcript>
```

… … *cont’d* … …

transcript.xml (cont’d)

```
<Transcript>
    <Student StudId=“123454321” Name=“Joe Blow” />
    <CrsTaken CrsCode=“CS315” Semester=“S1997” Grade=“A” />
    <CrsTaken CrsCode=“CS305” Semester=“S1996” Grade=“A” />
    <CrsTaken CrsCode=“MAT123” Semester=“S1996” Grade=“C” />
</Transcript>
<Transcript>
    <Student StudId=“023456789” Name=“Homer Simpson” />
    <CrsTaken CrsCode=“EE101” Semester=“F1995” Grade=“B” />
    <CrsTaken CrsCode=“CS305” Semester=“S1996” Grade=“A” />
</Transcript>

</Transcripts>
```

Result

```
<ClassRoster  CrsCode=“CS305”  Semester=“F1995”>
        <Student StudId=“111111111”  Name=“John Doe” />
        <Student StudId=“987654321”  Name=“Bart Simpson” />
</ClassRoster>
<ClassRoster  CrsCode=“CS305”  Semester=“F1995”>
        <Student StudId=“111111111”  Name=“John Doe” />
        <Student StudId=“987654321”  Name=“Bart Simpson” />
</ClassRoster>
<ClassRoster  CrsCode=“CS308” Semester=“F1994”>
        <Student StudId=“987654321”  Name=“Bart Simpson” />
</ClassRoster>
<ClassRoster  CrsCode=“CS308” Semester=“F1997”>
        <Student StudId=“111111111”  Name=“John Doe” />
</ClassRoster>
<ClassRoster  CrsCode=“EE101”  Semester=“F1997”>
        <Student StudId=“111111111”  Name=“John Doe” />
</ClassRoster>
…
```

Document Restructuring (cont'd)

- *Output elements have the form*:

  <ClassRoster CrsCode="CS305" Semester="F1995">
  <Student StudId="111111111" Name="John Doe" />
  <Student StudId="987654321" Name="Bart Simpson" />
  </ClassRoster>

- *Problem*: the above element <u>will be output *twice*</u> – once when $c is bound to

  <CrsTaken CrsCode="CS305" Semester="F1995" Grade="A" />

  John Doe's

  and once when it is bound to

  Bart Simpson's

  <CrsTaken CrsCode="CS305" Semester="F1995" Grade="C" />

The statement distinct-values( ) won't eliminate transcript records that refer to same class BECAUSE the grades are different!!!

Document Restructuring (cont'd)

- *Solution*: instead of

```
FOR $c IN distinct-values(doc("transcript.xml")//CrsTaken)
```

use

```
FOR $c IN doc("classes.xml")//Class
```

where classes.xml lists course offerings (course code/semester) *explicitly* (no need to extract them from transcript records).

Then $c is bound to each class exactly once, so each class roster will be output exactly once

http://xyz.edu/classes.xml

```
<Classes>
    <Class  CrsCode=“CS308”  Semester=“F1997” >
            <CrsName>SE</CrsName> <Instructor>Adrian Jones</Instructor>
    </Class>
    <Class  CrsCode=“EE101”  Semester=“F1995” >
            <CrsName>Circuits</CrsName> <Instructor>David Jones</Instructor>
    </Class>
    <Class  CrsCode=“CS305”  Semester=“F1995” >
            <CrsName>Databases</CrsName> <Instructor>Mary Doe</Instructor>
    </Class>
    <Class  CrsCode=“CS315”  Semester=“S1997” >
            <CrsName>TP</CrsName> <Instructor>John Smyth</Instructor>
    </Class>
    <Class  CrsCode=“MAR123”  Semester=“F1997” >
            <CrsName>Algebra</CrsName> <Instructor>Ann White</Instructor>
    </Class>
</Classes>
```

Document Restructuring (cont'd)

- *More problems*: the above query will list classes with no students.
- Reformulation that avoids this:

Test that classes aren't empty

```
FOR $c IN doc("classes.xml")//Class
WHERE doc("transcripts.xml")
                //CrsTaken[@CrsCode = $c/@CrsCode
                        and @Semester = $c/@Semester]
RETURN
   <ClassRoster CrsCode = {$c/@CrsCode} Semester = {$c/@Semester}> {
      FOR $t IN doc("transcript.xml")//Transcript
      WHERE $t/CrsTaken[@CrsCode = $c/@CrsCode and
                        @Semester = $c/@Semester]
      RETURN $t/Student ORDER BY $t/Student/@StudId
   } </ClassRoster>
ORDER BY $c/@CrsCode
```

XQuery Semantics

- So far the discussion was informal
- XQuery *semantics* defines what the expected result of a query is
- Defined analogously to the semantics of SQL

XQuery Semantics (cont'd)

- *Step 1*: Produce a list of bindings for variables
  - The FOR clause binds each variable to a *list* of nodes specified by an XQuery expression.
    The expression can be:
    - An XPath expression
    - An XQuery query
    - A function that returns a list of nodes
  - End result of a FOR clause:
    - Ordered list of tuples of document nodes
    - Each tuple is a binding for the variables in the FOR clause

XQuery Semantics (cont'd)

Example (bindings):

- Let FOR declare $A and $B
- Bind $A to document nodes {v,w}; $B to {x,y,z}
- Then FOR clause produces the following list of bindings for $A and $B:
  - $A/v, $B/x
  - $A/v, $B/y
  - $A/v, $B/z
  - $A/w, $B/x
  - $A/w, $B/y
  - $A/w, $B/z

XQuery Semantics (cont'd)

- *Step 2*: filter the bindings via the WHERE clause
  - Use each tuple binding to substitute its components for variables; retain those bindings that make WHERE true
  - Example: WHERE $A/CrsTaken/@CrsCode = $B/Class/@CrsCode

Binding: $A/w, where w = <CrsTaken CrsCode="CS308"…/>
$B/x, where x = <Class CrsCode="CS308" … />

Then w/CrsTaken/@CrsCode = x/Class/@CrsCode, so the WHERE condition is satisfied & binding retained

XQuery Semantics (cont'd)

- *Step 3*: Construct result
  - For each retained tuple of bindings, instantiate the RETURN clause
  - This creates a fragment of the output document
  - Do this for each retained tuple of bindings in sequence

User-defined Functions

- Can define functions, even recursive ones
- Functions can be called from within an XQuery expression
- Body of function is an XQuery expression
- Result of expression is returned
  - Result can be a primitive data type (integer, string), an element, a list of elements, a list of arbitrary document nodes, …

XQuery Functions: Example

- Count the number of *e*-children recursively:

```
DECLARE FUNCTION countNodes($e AS element()) AS integer {
    RETURN
        IF empty($e/*) THEN 0
        ELSE
            sum(FOR $n IN $e/* RETURN countNodes($n))
             + count($e/*)
}
```

Function signature

XQuery expression

Built-in functions sum, count, empty

Class Rosters Using Functions

```
DECLARE FUNCTION extractClasses($e AS element()) AS element()* {
    FOR $c IN $e//CrsTaken
    RETURN <Class CrsCode={$c/@CrsCode} Semester={$c/@Semester} />
}
<Rosters>
    FOR $c IN
        distinct-values(FOR $d IN doc(“transcript.xml”) RETURN extractClasses($d) )
    RETURN
        <ClassRoster CrsCode = {$c/@CrsCode} Semester = {$c/@Semester} >
        {
          LET $trs := doc(“transcript.xml”)
            FOR $t IN $trs//Transcript[CrsTaken/@CrsCode=$c/@CrsCode and
                                       CrsTaken/@Semester=$c/@Semester]
            RETURN $t/Student
          ORDER BY $t/Student/@StudId
        }
        </ClassRoster>
</Rosters>
```

Converting Attributes to Elements with XQuery

```
DECLARE FUNCTION convertAttributes($a AS attribute()) AS element() {
   RETURN element {name($a)} {data($a)}
}
DECLARE FUNCTION convertElement($e AS node()) AS element() {
   RETURN   element {name($e)}
          {
           { FOR $a IN $e/@* RETURN convertAttribute ($a) } ,
           IF empty($e/*) THEN $e/text( )
           ELSE { FOR $n IN $e/* RETURN convertElement($n) }
          }
}

RETURN convertElement(doc("my-document")/*)
```

Computed element

Concatenate results

The actual query: Just a RETURN statement!!

Integration with XML Schema and Namespaces

- Let type *sometype* be defined in http://types.r.us/types.xsd:

```
IMPORT SCHEMA  namespace trs = "http://types.r.us"
                              AT  "http://types.r.us/types.xsd";
DECLARE NAMESPACE  foo = "http://foo.org/foo";
DECLARE FUNCTION   local:doSomething($x  AS  trs:sometype)
                                                 AS  xs:string  {
FOR  $i  IN  fn:doc(...);
      ... ... ...
RETURN
   <foo:something>
           .........
   </foo:something>
}
```

Namespace

Location

Predefined namespace

Another namespace

Built-in functions have predefined namespaces

Local functions have predefined namespace

Grouping and Aggregation

- Does not use separate grouping operator
  - Subqueries inside the RETURN clause obviate this need
- Uses built-in aggregate functions count, avg, sum, etc. (some borrowed from XPath)

Aggregation Example

- *Produce a list of students along with the number of courses each student took*:

```
FOR $t IN fn:doc(“transcripts.xml”)//Transcript,
        $s IN $t/Student
LET $c := $t/CrsTaken
RETURN
      <StudentSummary StudId = {$s/@StudId} Name = {$s/@Name}
                      TotalCourses = {fn:count(fn:distinct-values($c))} />
ORDER BY StudentSummary/@TotalCourses
```

- The *grouping effect* is achieved because $c is bound to a *new* set of nodes for *each* binding of $t

Quantification in XQuery

- XQuery supports explicit quantification: SOME (∃) and EVERY (∀)
- *Example*:

```
FOR $t IN fn:doc("transcript.xml")//Transcript
WHERE SOME $ct IN $t/CrsTaken
            SATISFIES $ct/@CrsCode = "MAT123"
RETURN $t/Student
```

"Almost" equivalent to:

```
    FOR $t IN fn:doc("transcript.xml")//Transcript,
        $ct IN $t/CrsTaken
WHERE $ct/@CrsCode = "MAT123"
RETURN $t/Student
```

- *Not equivalent, if students can take same course twice*!

Implicit Quantification

- Note: in SQL, variables that occur in FROM, but not SELECT are implicitly quantified with $\exists$
- In XQuery, variables that occur in FOR, but not RETURN are similar to those in SQL. However:
  - In XQuery variables are bound to document nodes
    - Two nodes may look textually the same (e.g., two different instances of the same course element), but they are still different nodes and thus different variable bindings
    - Instantiations of the RETURN expression produced by binding variables to <u>different nodes</u> are output <u>even if these instantiations are textually identical</u>
  - In SQL a variable can be bound to the same value only once; identical tuples are not output twice (in theory)

  - *This is why the two queries in the previous slide are not equivalent*

Quantification (cont'd)

- Retrieve all classes (from classes.xml) where each student took MAT123
  - Hard to do in SQL (before SQL-99) because of the lack of explicit quantification

```
FOR $c IN fn:doc(classes.xml)//Class
LET $g := {          (: Transctipt records that correspond to class $c :)
   FOR $t IN fn:doc("transcript.xml")//Transcript
   WHERE $t/CrsTaken/@Semester = $c/@Semester
                AND $t/CrsTaken/@CrsCode = $c/@CrsCode
   RETURN $t
         }
WHERE EVERY $tr IN $g SATISFIES
                    NOT fn:empty($tr[CrsTaken/@CrsCode="MAT123])
RETURN $c ORDER BY $c/@CrsCode
```