Describing, Transforming, and Querying Semistructured Data

1

What's in This Part?

- From relational model to OO model to semistructured data and XML
- XML & DTD – introduction
- XML Schema – user-defined data types, integrity constraints
- XPath & XPointer – core query language for XML
- XSLT – document transformation language
- XQuery – full-featured query language for XML
- SQL/XML – extension of SQL for XML

2

Why XML?

- XML is a standard for data exchange that is taking over the World
- All major database products have been retrofitted with facilities to store and construct XML documents
- There are already database products that are specifically designed to work with XML documents rather than relational or object-oriented data
- XML is closely related to object-oriented and so-called *semistructured* data

3

Problems with Flat Relations

Consider a relation

Person(*SSN*, *Name*, *PhoneN*, *Child*)

with:

- FD: *SSN* → *Name*
- Any person (identified by *SSN*) can have several phone numbers and children
- Children and phones of a person are not related to each other except through that person

4

An Instance of Person

SSN	*Name*	*PhoneN*	*Child*
111-22-3333	Joe Public	516-123-4567	222-33-4444
111-22-3333	Joe Public	516-345-6789	222-33-4444
111-22-3333	Joe Public	516-123-4567	333-44-5555
111-22-3333	Joe Public	516-345-6789	333-44-5555
222-33-4444	Bob Public	212-987-6543	444-55-6666
222-33-4444	Bob Public	212-987-1111	555-66-7777
222-33-4444	Bob Public	212-987-6543	555-66-7777
222-33-4444	Bob Public	212-987-1111	444-55-6666

redundancy, anomaly

5

Normalization removes redundancy:

Person1

SSN	*Name*
111-22-3333	Joe Public
222-33-4444	Bob Public

Phone

SSN	*PhoneN*
111-22-3333	516-345-6789
111-22-3333	516-123-4567
222-33-4444	212-987-6543
222-33-4444	212-135-7924

SSN	*Child*
111-22-3333	222-33-4444
111-22-3333	333-44-5555
222-33-4444	444-55-6666
222-33-4444	555-66-7777

ChildOf

6

But querying is still cumbersome:

Get the phone numbers of Joe's grandchildren.

Against the original relation: three cumbersome joins

```
SELECT  G.PhoneN
FROM    Person P, Person C, Person G
WHERE   P.Name = 'Joe Public'  AND
        P.Child = C.SSN  AND  C.Child = G.SSN
```

Against the decomposed relations is even worse: four joins

```
SELECT  N.PhoneN
FROM    Person1 P, ChildOf C, ChildOf G, Phone N
WHERE   P.Name = 'Joe Public'  AND  P.SSN = C.SSN  AND
        C.Child = G.SSN  AND  G.Child = N.SSN
```


Objects Allow Simpler Design

Schema:

Person(*SSN*: String,
Name: String,
PhoneN: **{**String**}**,
Child: **{**SSN**}**)

Set data types

No need to decompose in order to eliminate redundancy: the set data type takes care of this.

Object 1:
(111-22-3333,
"Joe Public",
{516-345-6789, 516-123-4567},
{222-33-4444, 333-44-5555}
)

Object 2:
(222-33-4444,
"Bob Public",
{212-987-6543, 212-135-7924},
{444-55-6666, 555-66-7777}
)

Objects Allow Simpler Queries

Schema (slightly changed):

Person(*SSN*: String,
Name: String,
PhoneN: {String},
Child: **{**Person**}**)

Set of persons

- Because the type of Child is the set of Person-*objects*, it makes sense to continue querying the object attributes in a **path expression**

Object-based query:

```
SELECT  P.Child.Child.PhoneN
FROM    Person P
WHERE   P.Name = 'Joe Public'
```

Path expression

- Much more natural!

ISA (or Class) Hierarchy

Person(*SSN*, *Name*)
Student(*SSN*,*Major*)
Query: *Get the names of all computer science majors*

Relational formulation:

```
SELECT  P.Name
FROM    Person P, Student S
WHERE   P.SSN = S.SSN  and S.Major = 'CS'
```

Object-based formulation:

```
SELECT  S.Name
FROM    Student S
WHERE   S.Major = 'CS'
```

Student-objects are also Person-objects, so they *inherit* the attribute Name

Object Methods in Queries

- Objects can have associated operations (methods), which can be used in queries. For instance, the method frameRange(from, to) might be a method in class Movie. Then the following query makes sense:

```
SELECT  M.frameRange(20000, 50000)
FROM    Movie M
WHERE   M.Name = 'The Simpsons'
```


The "Impedance" Mismatch

- One cannot write a complete application in SQL, so SQL statements are embedded in a host language, like C or Java.
- **SQL**: Set-oriented, works with relations, uses high-level operations over them.
- **Host language**: Record-oriented, does not understand relations and high-level operations on them.
- **SQL**: Declarative.
- **Host language**: Procedural.
- Embedding SQL in a host language involves ugly adaptors (cursors/iterators) – a direct consequence of the above mismatch of properties between SQL and the host languages. It was dubbed *"impedance" mismatch.*

Object Databases vs. Relational Databases

- *Relational*: set of relations; relation = set of tuples
- *Object*: set of classes; class = set of objects
- *Relational*: tuple components are primitive (int, string)
- *Object*: object components can be complex types (sets, tuples, other objects)
- *Unique features of object databases*:
  - Inheritance hierarchy
  - Object methods
  - In some systems (ODMG), the host language and the data manipulation language are the same

13

Semistructured Data

- A typical piece of data on the Web:

```
<dt>Name: John Doe
    <dd>Id: 111111111
    <dd>Address:  <ul>
                      <li>Number:  123
                      <li>Street:    Main
                  </ul>
</dt>
<dt>Name: Joe Public
    <dd>Id:  222222222
    … … … …
</dt>
```

HTML does not distinguish between attributes and values

14

Semistructured Data (cont'd.)

- To make the previous student list suitable for machine consumption on the Web, it should have these characteristics:
  - Be *object-like*
  - Be *schemaless* (not guaranteed to conform exactly to any schema, but different objects have some commonality among themselves)
  - Be *self-describing* (some schema-like information, like attribute names, is part of data itself)
- Data with these characteristics are referred to as *semistructured.*

15

What is Self-describing Data?

- Non-self-describing (relational, object-oriented):

Data part:

```
(#123,  ["Students",   {["John", 111111111, [123,"Main St"]],
                        ["Joe", 222222222, [321, "Pine St"]] }
        ] )
```

Schema part:

```
PersonList[ ListName: String,
            Contents: [ Name: String,
                        Id: String,
                        Address: [Number: Integer,  Street: String]  ]
          ]
```

16

What is Self-Describing Data? (contd.)

- *Self-describing*:
  - Attribute names embedded in the data itself, *but are distinguished* from values
  - Doesn't need schema to figure out what is what (but schema might be useful nonetheless)

```
(#12345,
   [ListName: "Students",
    Contents:  { [ Name: "John Doe",
                   Id:  "111111111",
                   Address: [Number: 123, Street: "Main St."] ] ,
                 [Name: "Joe Public",
                  Id:  "222222222",
                  Address: [Number: 321, Street: "Pine St."] ]  }
   ] )
```

17

XML – The De Facto Standard for Semistructured Data

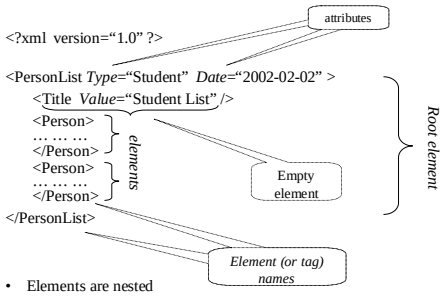
- XML: eXtensible Markup Language
  - Suitable for semistructured data and has become a standard:
    - Easy to describe object-like data
    - Self-describing
    - Doesn't require a schema (but can be provided optionally)
- We will study:
  - DTDs – an older way to specify schema
  - XML Schema – a newer, more powerful (and much more complex!) way of specifying schema
  - Query and transformation languages:
    - XPath
    - XSLT
    - Xquery
    - SQL/XML

18

Overview of XML

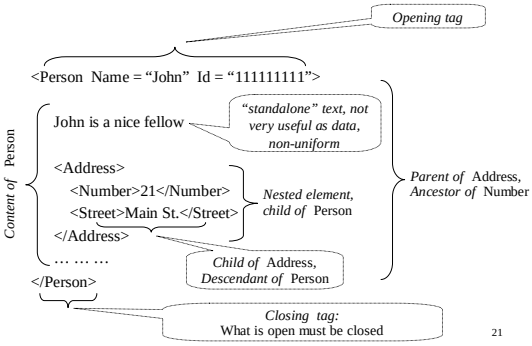
- Like HTML, but any number of different tags can be used (up to the document author) – extensible
- Unlike HTML, no semantics behind the tags
  - For instance, HTML's <table>…</table> means: render contents as a table; in XML: doesn't mean anything
  - Some semantics can be specified using XML Schema (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

Example



- Elements are nested
- Root element contains all others

More Terminology



Conversion from XML to Objects

- Straightforward:

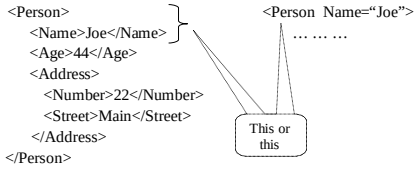
```
<Person Name="Joe">
    <Age>44</Age>
    <Address><Number>22</Number><Street>Main</Street></Address>
</Person>
```

Becomes:

```
(#345, [Name: "Joe",
        Age: 44,
        Address: [Number: 22, Street: "Main"]
] )
```


Conversion from Objects to XML

- Also straightforward
- Non-unique:
  - Always a question if a particular piece (such as Name) should be an element in its own right or an attribute of an element
  - *Example*: A reverse translation could give

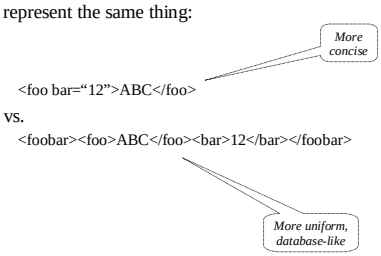


Differences between XML Documents and Objects

- XML's origin is document processing, not databases
  - Allows things like standalone text (useless for databases)
    <foo> Hello <moo>123</moo> Bye </foo>
  - XML data is ordered, while database data is not:
    <something><foo>1</foo><bar>2</bar></something>
    is different from
    <something><bar>2</bar><foo>1</foo></something>
    but these two complex values are same:
    [*something*: [*bar*:1, *foo*:2]]
    [*something*: [*foo*:2, *bar*:1]]

Differences between XML Documents and Objects (cont'd)

- Attributes aren't needed – just bloat the number of ways to represent the same thing:



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="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="666666666">
      <Name><First>Joe</First><Last>Public</Last></Name>  <Status>U3</Status>
      <CrsTaken CrsCode="CS308" Semester="F1994" />
      <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>
  …… continued … …
```

ID

IDREF

Report Document (cont'd.)

```
  <Classes>
    <Class>
      <CrsCode>CS308</CrsCode> <Semester>F1994</Semester>
      <ClassRoster Members="666666666 987654321" />
    </Class>
    <Class>
      <CrsCode>CS308</CrsCode> <Semester>F1997</Semester>
      <ClassRoster Members="111111111" />
    </Class>
    <Class>
      <CrsCode>MAT123</CrsCode> <Semester>F1997</Semester>
      <ClassRoster Members="111111111 666666666" />
    </Class>
  </Classes>
…… continued … …
```

IDREFS

Report Document (cont'd.)

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

ID

XML Namespaces

- A mechanism to prevent name clashes between components of same or different documents
- Namespace declaration
  - *Namespace* – a symbol, typically a URL
  - *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.,
  - **xsd** 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 >
- **<!**ATTLIST *elt-name attr-name*
  CDATA/ID/IDREF/IDREFS
  #IMPLIED/#REQUIRED
  >
- Can define other things, like macros (called *entities* in the XML jargon)

Element's contents

Type of attribute

Optional/mandatory

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

37

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

38

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

39

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.

40

Instance Document

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

Default namespace for instance document

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

```
<?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" >
    … same contents as in the earlier Report document …
</Report>
```

- 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 not binding on the XML processor; it can decide to use another schema, or none at all

41

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
  - Note: this schemaLocation keyword is from the XMLSchema namespace – different from xsi:schemaLocation in previous slide, which was in XMLSchema-instance namespace

42

Simple Types

- *Primitive types*: *decimal*, *integer*, *boolean*, *string*, ID, IDREF, etc.
- *Type constructors*: *list* and *union*
  - A simple way to derive types from primitive types:

```
<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="[0-9]{9}" />
      </restriction>
</simpleType>
<simpleType name="studentIds" >
      <list itemType="adm:studentRef" />
</simpleType>

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

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

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

Why is a namespace prefix needed here? (think)

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 -- have already seen
- all – can specify sets of elements
- choice – can specify alternative types

Sets

- Suppose the order of components in addresses is unimprotant:

```
<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

55

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

56

Extension and Resriction 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 (ie, replacing existing types with subtypes)

57

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

58

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>
    <element name="Student" type="fooAdm:studentType" />
  </complexType>
```

Must repeat the original definition

Tightened type: the original was "unbounded"

59

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

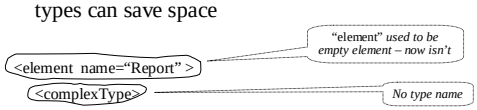
Definition of root type

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

60

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




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

61

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 (shown next)

62

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. (Much) more on XPath later

63

(Very) Basic XPath – for Key Specification

- Objects selected by the various XPath expressions are color coded

```
<Offerings>        – – current reference point
    <Offering>
        <CrsCode Section="1">CS532</CrsCode>
        <Semester><Term>Spring</Term><Year>2002</Year></Semester>
    </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

64

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>
```

65

Example (cont'd)

- A key specification for the previous document:

```
<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*

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

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

66

Foreign Keys

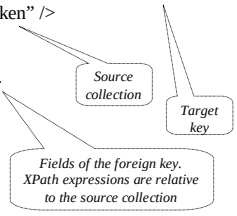
- Like the REFERENCES clause in SQL, but more involved
- Need to specify:
  - *Foreign key*:
    - *Source collection* of objects
    - Fields that form the foreign key
  - *Target key*:
    - A previously defined *key* (or *unique*) specification, which is comprised of:
      - *Target collection* of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

67

Foreign Key: Example

- Every class must have at least one student

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



The above keyref declaration is part of element declaration for Students

68