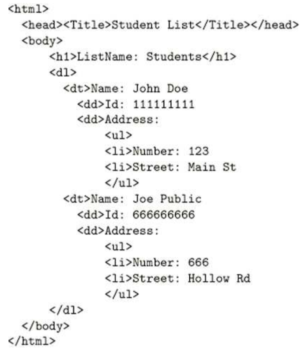
XML and Web Data

Facts about the Web

- Growing fast
- Popular
- Semi-structured data
  - – Data is presented for 'human'-processing
  - – Data is often 'self-describing' (including name of attributes within the data fields)

Figure 17.1
A student list in HTML.



```
<html>
  <head><Title>Student List</Title></head>
  <body>
      <h1>ListName: Students</h1>
      <dl>
        <dt>Name: John Doe
          <dd>Id: 111111111
          <dd>Address:
              <ul>
              <li>Number: 123
              <li>Street: Main St
              </ul>
        <dt>Name: Joe Public
          <dd>Id: 666666666
          <dd>Address:
              <ul>
              <li>Number: 666
              <li>Street: Hollow Rd
              </ul>
      </dl>
  </body>
</html>
```

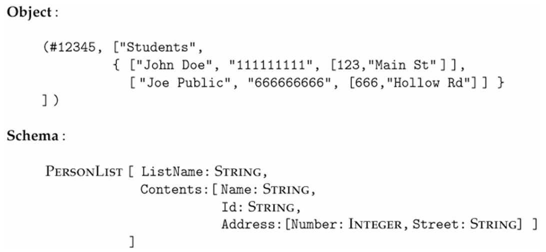
Students

Name	Id	Address	
		Number	Street
John Doe	111111111	123	Main St
Joe Public	666666666	666	Hollow Rd

Vision for Web data

- *Object-like* – it can be represented as a collection of objects of the form described by the conceptual data model
- *Schemaless* – not conformed to any type structure
- *Self-describing* – necessary for machine readable data

Figure 17.2
Student list in object form.



Object :

```
(#12345, ["Students",
          { ["John Doe", "111111111", [123,"Main St"] ],
            [ "Joe Public", "666666666", [666,"Hollow Rd"] ] }
 ] )
```

Schema :

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

XML – Overview

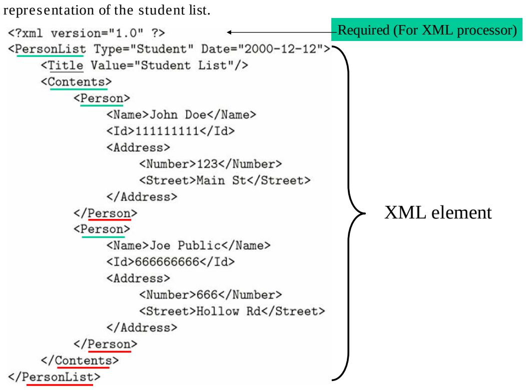
- Simplifying the data exchange between software agents
- Popular thanks to the involvement of W3C (World Wide Web Consortium – independent organization www.w3c.org)

XML – Characteristics

- Simple, open, widely accepted
- HTML-like (tags) but extensible by users (no fixed set of tags)
- No predefined semantics for the tags (because XML is developed not for the displaying purpose)
- Semantics is defined by *stylesheet* (later)

Figure 17.3

XML representation of the student list.



```
<?xml version="1.0" ?>
<PersonList Type="Student" Date="2000-12-12">
    <Title Value="Student List"/>
    <Contents>
        <Person>
            <Name>John Doe</Name>
            <Id>111111111</Id>
            <Address>
                <Number>123</Number>
                <Street>Main St</Street>
            </Address>
        </Person>
        <Person>
            <Name>Joe Public</Name>
            <Id>666666666</Id>
            <Address>
                <Number>666</Number>
                <Street>Hollow Rd</Street>
            </Address>
        </Person>
    </Contents>
</PersonList>
```

XML Documents

- User-defined tags:

  <tag> info </tag>
- Properly nested:<tag1>.. <tag2>…</tag1></tag2>

  is not valid
- Root element: an element contains all other elements
- Processing instructions <?command ….?>
- Comments <!--- comment --- >
- CDATA type
- DTD

XML element

- Begin with a *opening tag* of the form

  <XML_element_name>
- End with a *closing tag*

  </XML_element_name>
- The text between the beginning tag and the closing tag is called the *content* of the element

XML element

Attribute Value of the attribute

<PersonList Type="Student">

<Student StudentID="123">

<Name> <First>"XYZ"</First>

<Last>"PQR"</Last> </Name>

<CrsTaken CrsName="CS582" Grade="A"/>

</Student>

…

</PersonList>

Relationship between XML elements

- Child-parent relationship
  - Elements nested directly in an element are the children of this element (*Student* is a child of *PersonList*, *Name* is a child of *Student*, etc.)
- Ancestor/descendant relationship: important for querying XML documents (extending the child/parent relationship)

XML elements & Database Objects

- XML elements can be converted into objects by
  - considering the tag's names of the children as attributes of the objects
  - Recursive process

```
<Student StudentID="123">
   <Name> "XYZ  PQR" </Name>
   <CrsTaken>
    <CrsName>CS582</CrsName>
     <Grade>"A"</Grade>  </CrsTaken>
</Student>
```

Partially converted object

```
(#099,
   Name: "XYZ  PQR"
   CrsTaken:
   <CrsName>"CS582"</CrsName>
   <Grade>"A"</Grade>
)
```

XML elements & Database Objects

- Differences: Additional text within XML elements

```
<Student StudentID="123">
   <Name> "XYZ  PQR" </Name>
    has taken the following course
   <CrsTaken>
    Database management system II
    <CrsName>CS582</CrsName>
    with the grade
     <Grade>"A"</Grade>  </CrsTaken>
</Student>
```

XML elements & Database Objects

- Differences: XML elements are orderd

```
<CrsTaken>                                <CrsTaken>
   <CrsName>"CS582"</CrsName>                <Grade>"A"</Grade>
   <Grade>"A"</Grade>               ≠        <CrsName>"CS582"</CrsName>
</CrsTaken>                               </CrsTaken>
```

{#901, Grade: "A", CrsName: "CS582"}

XML Attributes

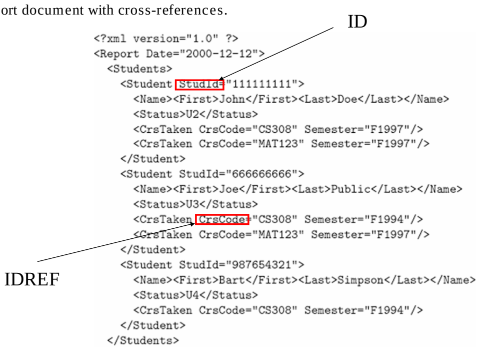
- Can occur within an element (arbitrary many attributes, order unimportant, same attribute only one)
- Allow a more concise representation
- Could be replaced by elements
- Less powerful than elements (only string value, no children)
- Can be declared to have unique value, good for integrity constraint enforcement (next slide)

XML Attributes

- Can be declared to be the type of ID, IDREF, or IDREFS
- ID: unique value throughout the document
- IDREF: refer to a valid ID declared in the same document
- IDREFS: space-separated list of strings of references to valid IDs

Figure 17.4A

A report document with cross-references.

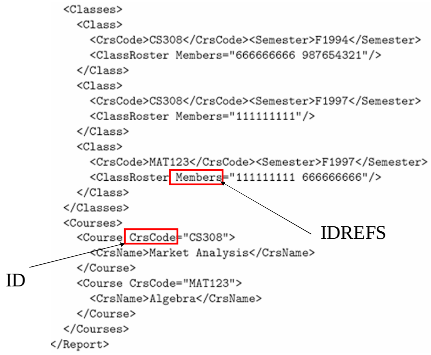


```
<?xml version="1.0" ?>
<Report Date="2000-12-12">
  <Students>
    <Student StudId="111111111">
      <Name><First>John</First><Last>Doe</Last></Name>
      <Status>U2</Status>
      <CrsTaken CrsCode="CS308" Semester="F1997"/>
      <CrsTaken CrsCode="MAT123" Semester="F1997"/>
    </Student>
    <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 on next slide) ➔

Figure 17.4B (continued)

A report document with cross-references.



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

Well-formed XML Document

- It has a root element
- Every opening tag is followed by a matching closing tag, elements are properly nested
- Any attribute can occur at most once in a given opening tag, its value must be provided, quoted

So far

- Why XML?
- XML elements
- XML attributes
- Well-formed XML document

Namespaces and DTD

Namespaces

- For avoiding naming conflicts
- Name of every XML tag must have two parts:
  - **namespace**: a string in the form of a **uniform resource identifier** (**URI**) or a **uniform resource locator** (**URL**)
  - **local name**: as regular XML tag but cannot contain ':'
- Structure of an XML tag:

namespace:local_name

Namespaces

- An XML namespace is a collection of names, identified by a URI reference, which are used in XML documents as element types and attribute names. XML namespaces differ from the "namespaces" conventionally used in computing disciplines in that the XML version has internal structure and is not, mathematically speaking, a set.

Source: www.w3c.org

Uniform Resource Identifier

- URI references which identify namespaces are considered identical when they are exactly the same character-for-character. Note that URI references which are not identical in this sense may in fact be functionally equivalent. Examples include URI references which differ only in case, or which are in external entities which have different effective base URIs.

Source: www.w3c.org

Namespace - 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>
```

Two namespaces are used: the two **URL**s

xmlns =   defined the default namespace,

xmlns:*toy* =   defined the second namespace

Namespace declaration

- Defined by

  xml : prefix = declaration

- Tags belonging to a namespace should be prefixed with “*prefix:”*
- Tags belonging to the default namespace do not need to have the prefix
- Have its own scope

Namespace declaration

```
<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 xmlns=“http://www.acmeinc.com/jp#supplies2”
         xmlns:toy=“http://www.acmeinc.com/jp#toys2”>
         <name> notebook </name>
         <feature> <toy:name>sticker</toy:name> </feature>
   </item>
</item>
```

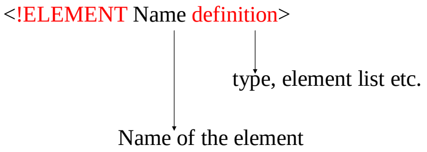
Document Type Definition

- Set of rules (by the user) for structuring an XML document
- Can be part of the document itself, or can be specified via a URL where the DTD can be found
- A document that conforms to a DTD is said to be *valid*
- Viewed as a *grammar* that specifies a legal XML document, based on the tags used in the document

DTD Components

- A **name** – must coincide with the tag of the root element of the document conforming to the DTD
- A set of **ELEMENT**s – one ELEMENT for each allowed tag, including the root tag
- **ATTLIST** statements – specifies the allow attributes and their type for each tag
- *, +, ? – like in grammar definition
  - * : zero or finitely many number
  - + : at least one
  - ? : zero or one

DTD Components – Element



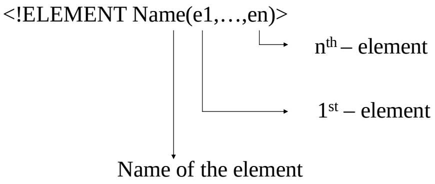
<!ELEMENT Name definition>

type, element list etc.

Name of the element

definition can be: EMPTY, (#PCDATA), or element list (e1,e2,…,en) where the list (e1,e2,…,en) can be shortened using grammar like notation

DTD Components – Element



<!ELEMENT Name(e1,…,en)>

nth – element

1st – element

Name of the element

<!ELEMENT PersonList (Title,Contents)>

<!ELEMENT Contents(Person *)>

DTD Components – Element

<!ELEMENT Name EMPTY>

no child for the element Name

<!ELEMENT Name (#PCDATA)>

value of Name is a character string

<!ELEMENT Title EMPTY>

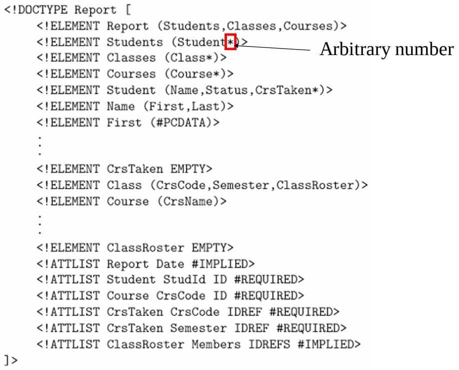
<!ELEMENT Id (#PCDATA)>

DTD Components – Attribute List

<!ATTLIST EName Att {Type} Property>
where

- Ename – name of an element defined in the DTD
- Att – attribute name allowed to occur in the opening tag of Ename
- {type} – might/might not be there; specify the type of the attribute (CDATA, ID, IDREF, IDREFS)
- Property – either #REQUIRED or #IMPLIED

Figure 17.5
A DTD for the report document in Figure 17.4.



```
<!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)>
    .
    .
    .
    <!ELEMENT ClassRoster EMPTY>
    <!ATTLIST Report Date #IMPLIED>
    <!ATTLIST Student StudId ID #REQUIRED>
    <!ATTLIST Course CrsCode ID #REQUIRED>
    <!ATTLIST CrsTaken CrsCode IDREF #REQUIRED>
    <!ATTLIST CrsTaken Semester IDREF #REQUIRED>
    <!ATTLIST ClassRoster Members IDREFS #IMPLIED>
]>
```

Arbitrary number

DTD as Data Definition Language?

- Can specify exactly what is allowed on the document
- XML elements can be converted into objects
- Can specify integrity constraints on the elements
- Is is good enough?

Inadequacy of DTP as a Data Definition Language

- Goal of XML: for specifying documents that can be exchanged and automatically processed by software agents
- DTD provides the possibility of querying Web documents but has many limitations (next slide)

Inadequacy of DTP as a Data Definition Language

- Designed without namespace in mind
- Syntax is very different than that of XML
- Limited basic types
- Limited means for expressing data consistency constrains
- Enforcing referential integrity for attributes but not elements
- XML data is ordered; not database data
- Element definitions are global to the entire document

XML Schema

XML Schema – Main Features

- Same syntax as XML
- Integration with the namespace mechanism (different schemas can be imported from different namespaces and integrated into one)
- Built-in types (similar to SQL)
- Mechanism for defining complex types from simple types
- Support keys and referential integrity constraints
- Better mechanism for specifying documents where the order of element types does not matter

XML Document and Schema

A document conforms to a schema is called an *instance* of this schema and is said to be *schema valid*.

XML processor does not check for schema validity

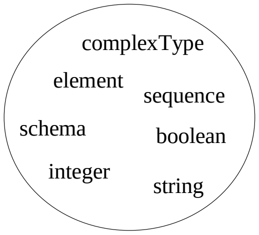
XML Schema and Namespaces

- Describes the structure of other XML documents
- Begins with a declaration of the namespaces to be used in the schema, including
  - http://www.w3.org/2001/XMLSchema
  - http://www.w3.org/2001/XMLSchema-instance
  - **targetnamespace** (user-defined namespace)

http://www.w3.org/2001/XMLSchema

- Identifies the names of tags and attributes used in a schema (names defined by the XML Schema Specification, e.g., *schema, attribute, element*)
- Understood by all schema aware XML processor
- These tags and attributes describe structural properties of documents in general

http://www.w3.org/2001/XMLSchema



complexType

element

sequence

schema

boolean

integer

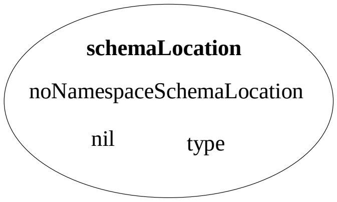
string

The names defined in XMLSchema

http://www.w3.org/2001/XMLSchema-instance

- Used in conjunction with the XMLSchema namespace
- Identifies some other special names which are defined in the XML Schema Specification but are used in the instance documents

http://www.w3.org/2001/XMLSchema-instance



schemaLocation

noNamespaceSchemaLocation

nil

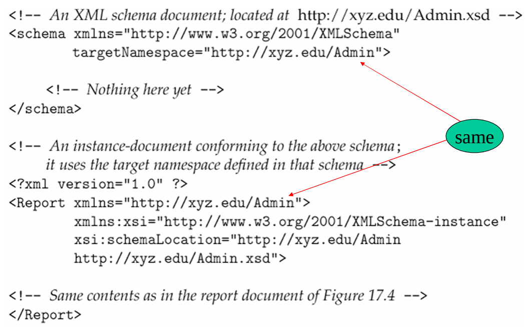
type

The names defined in XMLSchema-instance

Target namespace

- identifies the set of names defined by a particular schema document
- is an attribute of the *schema* element (**targetNamespace**) whose value is the name space containing all the names defines by the schema

Figure 17.6
Schema and an instance document.



```
<!-- An XML schema document; located at http://xyz.edu/Admin.xsd -->
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://xyz.edu/Admin">

     <!-- Nothing here yet -->
</schema>

<!-- An instance-document conforming to the above schema;
     it uses the target namespace defined in that schema -->
<?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 report document of Figure 17.4 -->
</Report>
```

Include statement

```
<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/CoursTypes.xsd"/>
 ….
</schema>
```

Include the schema in the location … to this schema
(good for combining)

Types

- Simple types (See Slides 56-68 of [RC])
  - Primitive
  - Deriving simple types
- Complex types

[RC] – Roger Costello's Slide on XML-Schema

Built-in Datatypes (From [RC])

• Primitive Datatypes	• Atomic, built-in
– string	– "**Hello World**"
– boolean	– {**true**, **false**}
– decimal	– **7.08**
– float	– **12.56E3**, **12**, **12560**, **0**, **-0**, **INF**, **-INF**, **NAN**
– double	– **12.56E3**, **12**, **12560**, **0**, **-0**, **INF**, **-INF**, **NAN**
– duration	– **P1Y2M3DT10H30M12.3S**
– dateTime	– format: *CCYY-MM-DDThh-mm-ss*
– time	– format: *hh:mm:ss.sss*
– date	– format: *CCYY-MM-DD*
– gYearMonth	– format: *CCYY-MM*
– gYear	– format: *CCYY*
– gMonthDay	– format: --*MM-DD*

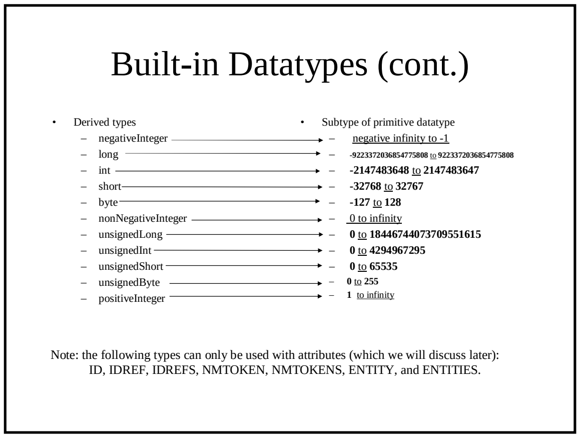
Note: 'T' is the date/time separator
INF = infinity
NAN = not-a-number

Built-in Datatypes (cont.)

• Primitive Datatypes	• Atomic, built-in
– gDay	– format: ---*DD* (note the 3 dashes)
– gMonth	– format: --*MM*--
– hexBinary	– a hex string
– base64Binary	– a base64 string
– anyURI	– **http://www.xfront.com**
– QName	– a namespace qualified name
– NOTATION	– a NOTATION from the XML spec

Built-in Datatypes (cont.)

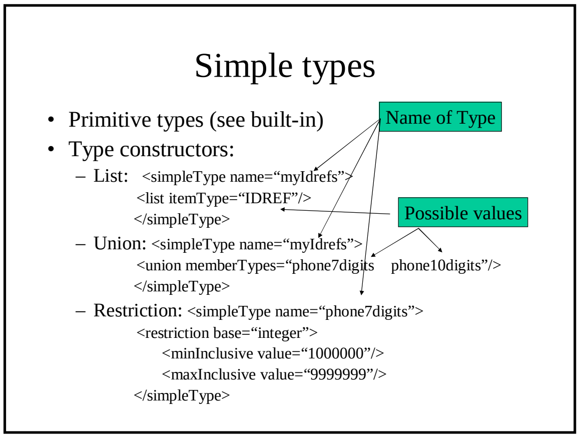
• Derived types	• Subtype of primitive datatype
– normalizedString	– A string without tabs, line feeds, or carriage returns
– token	– String w/o tabs, l/f, leading/trailing spaces, consecutive spaces
– language	– any valid xml:lang value, e.g., **EN**, **FR**, ...
– IDREFS	– must be used only with attributes
– ENTITIES	– must be used only with attributes
– NMTOKEN	– must be used only with attributes
– NMTOKENS	– must be used only with attributes
– Name	
– NCName	– **part** (no namespace qualifier)
– ID	– must be used only with attributes
– IDREF	– must be used only with attributes
– ENTITY	– must be used only with attributes
– integer	– **456**
– nonPositiveInteger	– negative infinity to 0



Built-in Datatypes (cont.)

Derived types	Subtype of primitive datatype
negativeInteger	negative infinity to -1
long	**-9223372036854775808** to **9223372036854775808**
int	**-2147483648** to **2147483647**
short	**-32768** to **32767**
byte	**-127** to **128**
nonNegativeInteger	0 to infinity
unsignedLong	**0** to **18446744073709551615**
unsignedInt	**0** to **4294967295**
unsignedShort	**0** to **65535**
unsignedByte	**0** to **255**
positiveInteger	**1** to infinity

Note: the following types can only be used with attributes (which we will discuss later):
ID, IDREF, IDREFS, NMTOKEN, NMTOKENS, ENTITY, and ENTITIES.

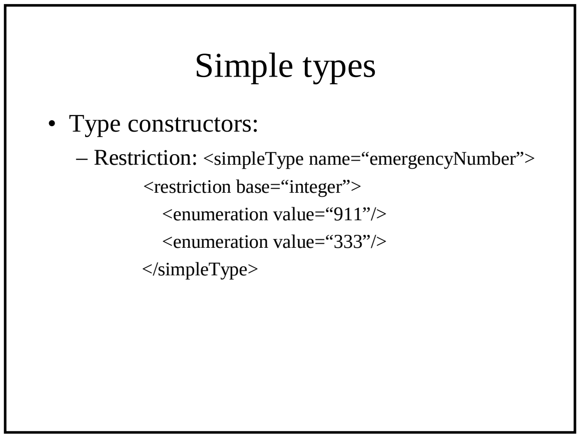


Simple types

- Primitive types (see built-in)
- Type constructors:
  - List: <simpleType name="myIdrefs">
    <list itemType="IDREF"/>
    </simpleType>
  - Union: <simpleType name="myIdrefs">
    <union memberTypes="phone7digits phone10digits"/>
    </simpleType>
  - Restriction: <simpleType name="phone7digits">
    <restriction base="integer">
    <minInclusive value="1000000"/>
    <maxInclusive value="9999999"/>
    </simpleType>

Name of Type

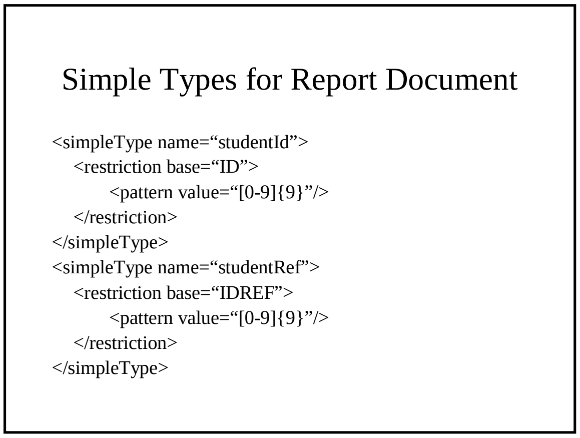
Possible values



Simple types

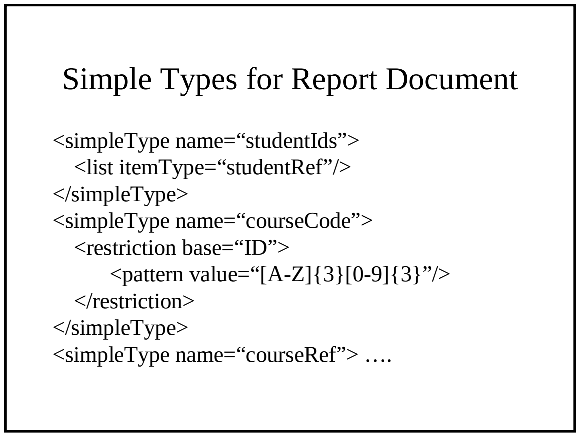
- Type constructors:
  - Restriction: <simpleType name="emergencyNumber">

```
<restriction base="integer">
   <enumeration value="911"/>
   <enumeration value="333"/>
</simpleType>
```



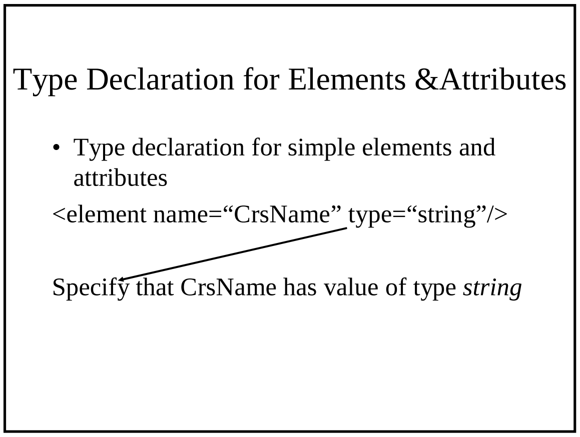
Simple Types for Report Document

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



Simple Types for Report Document

```
<simpleType name="studentIds">
   <list itemType="studentRef"/>
</simpleType>
<simpleType name="courseCode">
   <restriction base="ID">
         <pattern value="[A-Z]{3}[0-9]{3}"/>
   </restriction>
</simpleType>
<simpleType name="courseRef"> ….
```



Type Declaration for Elements &Attributes

- Type declaration for simple elements and attributes

<element name="CrsName" type="string"/>

Specify that CrsName has value of type *string*

Type Declaration for Elements &Attributes

- Type declaration for simple elements and attributes

```
<element name=“status” type=“adm:studentStatus”/>
```

Specify that status has value of type *studentStatus* that will be defined in the document

Example for the type studentStatus

```
<simpleType name=“studentStatus”>
   <restriction base=“string”>
         <enumeration value=“U1”/>
         <enumeration value=“U2”/>
         …
         <enumeration value=“G5”/>
   </restriction>
</simpleType>
```

Complex Types

- Use to specify the type of elements with children or attributes
- Opening tag: *complexType*
- Can be associated to a name in the same way a simple type is associated to a name

Complex Types

- Special Case: element with simple content and some attributes/no child with some attributes

```
<complexType name=“CourseTakenType”>
 <attribute name=“CrsCode” type=“adm:courseRef”/>
<attribute name=“Semester” type=“string”/>
</complexType>
```

Complex Types

- Combining elements into group -- <all>

```
<complexType name=“AddressType”>
<all>
   <element name=“StreetName” type=“string”>
   <element name=“StreetNumber” type=“string”>
   <element name=“City” type=“string”>
</all>
</complexType>
```

The three elements can appear in arbitrary order! (NOTE: <all> requires special care – it must occur after <complexType> - see book for invalid situation)

Complex Types

- Combining elements into group – <sequence>

```
<complexType name=“NameType”>
<sequence>
   <element name=“First” type=“string”>
   <element name=“Last” type=“string”>
</sequence>
</complexType>
```

The two elements must appear in order

Complex Types

- Combining elements into group – <choice>

```
<complexType name=“addressType”>
<choice>
   <element name=“POBox” type=“string”>
   <sequence><element name=“Name” type=“string”>
      <element name=“Number” type=“string”>
   </sequence>
</choice>  ….
</complexType>
```

Either POBox or Name and Number is needed

Complex Types

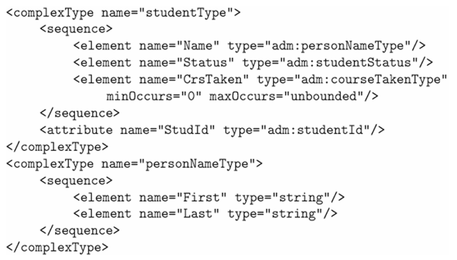
- Can also refer to local type like – allowing different elements to have children with the same name (next slides)

[studentType – courseType] both have the “Name” element

[studentType – personNameType] both have the “Name” element

```
<complexType name=“studentType”>
   <sequence>
        <element name=“Name” type=“…”>
        <element name=“Status” type=“…”>
         <element name=“CrsTaken” type=“…”>
   </sequence>
   <attribute name=“StudId” type=“…”>
</complexType>
<complexType name=“courseType”>
   <sequence>
        <element name=“Name” type=“…”>
   </sequence>
   <attribute name=“CrsCode” type=“…”>
</complexType>
```

Figure 17.7
Definition of the complex type 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>
```

Complex Types

- Importing schema: like *include* but does not require schemaLocation

instead of

<include schemaLocation=“http://xyz.edu/CoursTypes”/>

we can use

<import namespace=“http://xyz.edu/CoursTypes”/>

Complex Types

- Deriving new complex types by extension and restriction (for modifying imported schema)

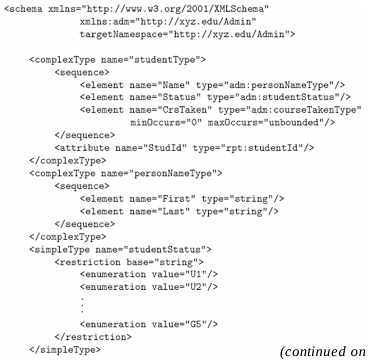
```
….
<import namespace=“http://xyz.edu/CoursTypes”/>
…..
<complexType name=“courseType”>
<complexContent> <extension base=“..”>
        <element name=“syllabus” type=“string”/>
                    </extension>
</complexContent></complexType>
```

The type that is going to be extended

A complete XML Schema for the Report Document

```
<schema xmlns=“http://www.w3.org/2001/XMLSchema”>
      xmlns:adm=“http://xyz.edu/Admin”
      targetNamespace=“http://xyz.edu/Admin”>
<include schemaLocation=“http://xyz.edu/StudentTypes.xsd”/>
<include schemaLocation=“http://xyz.edu/CourseTypes.xsd”/>
<element name=“Report” type=“adm:reportType”/>
<complexType name=“reportType”>
      <sequence>
            <element name=“Students” type=“adm:studentList”/>
            <element name=“Classes” type=“adm:classOfferrings”/>
            <element name=“Course” type=“adm:couseCatalog”/>
      </sequence>
</complexType>
<complexType name=“studentList”>
      <sequence>
            <element name=“Student> type=“adm:studentType”
                        minOccurs=“0” maxOccurs=“unbounded”/>
      </sequence>
</compleType>
</schema>
```

Figure 17.8A
Student types at http://xyz.edu/StudentTypes.xsd.

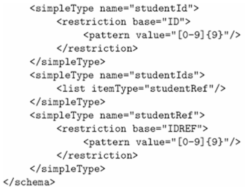


```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
            xmlns:adm="http://xyz.edu/Admin"
            targetNamespace="http://xyz.edu/Admin">

    <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="rpt:studentId"/>
    </complexType>
    <complexType name="personNameType">
        <sequence>
            <element name="First" type="string"/>
            <element name="Last" type="string"/>
        </sequence>
    </complexType>
    <simpleType name="studentStatus">
        <restriction base="string">
            <enumeration value="U1"/>
            <enumeration value="U2"/>
            .
            .
            .
            <enumeration value="G5"/>
        </restriction>
    </simpleType>
```

(continued on next slide) ➔

Figure 17.8B (continued)
Student types at http://xyz.edu/StudentTypes.xsd.



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

Integrity Constraints

- ID, IDREF, IDREFS can still be used
- Specified using the attribute xpath (next)
- XML keys, foreign keys
- Keys are associated with collection of objects not with types

Integrity Constraints - Keys

```
<key name=“PrimaryKeyForClass”>
  <selector xpath=“Classes/Class”/>
  <field    xpath=“CrsCode”/>
  <field    xpath=“Semester”/>
</key>
```

Collection of elements which are associated with the key

The key comprises of two elements (CrsCode and Semester) – both are children of Class

Integrity Constraints - Foreign key

```
<keyref name=“XXX” refer=“adm:PrimaryKeyForClass”>
  <selector xpath=“Students/Student/CrsTaken”/>
  <field    xpath=“@CrsCode”/>
  <field    xpath=“@Semester”/>
</keyref>
```

Source Collection: where the elements should satisfy the key specified by the “Prim … Class”

Figure 17.9

Course types at http://xyz.edu/CourseTypes.xsd.

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

<complexType name="courseTakenType">
    <attribute name="CrsCode" type="adm:courseRef"/>
    <attribute name="Semester" type="string"/>
</complexType>
<complexType name="courseType">
    <sequence>
        <element name="Name" type="string"/>
    </sequence>
    <attribute name="CrsCode" type="adm:courseCode"/>
</complexType>

<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>
</schema>
```

Complex type with only att's

Complex type with sequence

Simple type with restriction

Example of type definitions

Figure 17.10A

Part of a schema with a key and a foreign-key constraint.

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

  <complexType name="courseTakenType">
    <attribute name="CrsCode" type="adm:courseRef"/>
    <attribute name="Semester" type="string"/>
  </complexType>
  <complexType name="classType">
    <sequence>
      <element name="CrsCode" type="adm:courseCode"/>
      <element name="Semester" type="string"/>
      <element name="ClassRoster" type="adm:classListType"/>
    </sequence>
  </complexType>
```

Similarly to couseTakenType: type for classOfferings as a sequence of classes whose type is classType

lide) ➔

Figure 17.10B

Part of a schema with a key and a foreign-key constraint.

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

  <key name="PrimaryKeyForClass">
    <selector xpath="Classes/Class"/>
    <field xpath="CrsCode"/>
    <field xpath="Semester"/>
  </key>
  <keyref name="NoBogusTranscripts" refer="adm:PrimaryKeyForClass">
    <selector xpath="Students/Student/CrsTaken"/>
    <field name="@CrsCode"/>
    <field name="@Semester"/>
  </keyref>
</complexType>
</schema>
```

KEY: 2 children CrsCode and Semester of Class

FOREIGN KEY: 2 attributes CrsCode and Semester of CrsTaken

XML Query Languages

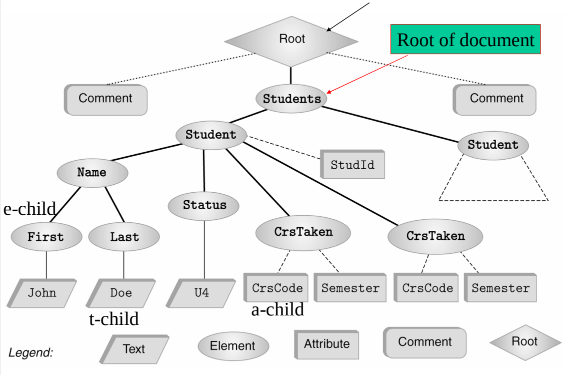
- Market, convenience, …
- XPath, XSLT, XQuery: three query languages for XML
- XPath – simple & efficient
- XSLT – full feature programming language, powerful query capabilities
- XQuery – SQL style query language – most powerful query capabilities

XPath

- Idea comes from *path expression* of OQL in object databases
- Extends the path expressions with query facilities by allowing search condition to occur in path expressions
- XPath data model: view documents as trees (see picture), providing operators for tree traversing, use *absolute* and *relative path expression*
- A XPath expression takes a document tree, returns a set of nodes in the tree

Figure 17.11

XPath document tree.


Root of XPath tree

Root of document

XPath Expression - Examples

/Students/Student/CrsTaken – returns the set of references to the nodes that correspond to the elements *CrsTaken*

First or **./First** refers to the node corresponds to the same child element *First* if the current position is *Name*

/Students/Student/CrsTaken/@CrsCode – the set of values of attributes CrsCode

/Students/Student/Name/First/text() – the set of contents of element *First*

Advanced Navigation

/Students/Student[1]/CrsTaken[2] – first *Student* node, second *CrsTaken* node

//CrsTaken – all CrsTaken elements in the tree (*descendant-or-self*)

Student/* - all e-children of the *Student* children of the current node

/Students/Student[search_expression] – all *Student* node satisfying the expressions; see what *search_expression* can be in the book!

XPointer

- Use the features of XPath to navigate within an XML document
- Syntax:

  someURL#xpointer*(XPathExpr1)*xpointer*(XPathExpr2)…*
- Example:

  http://www.foo.edu/Report.xml#xpointer(//Student[…])

XSLT

- Part of XSL – an *extensible stylesheet langage of XML*, a transformation language for XML: converting XML documents into any type of documents (HTML, XML, etc)
- A functional programming language
- XML syntax
- Provide instructions for converting/extracting information
- Output XML

XSLT Basics

- Stylesheet: specifies a transformation of one type of document into another type
- Specifies by a command in the XML document

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl"
        href="http://xyz.edu/Report/report.xsl"?>
<Report Date="2002-03-01"
….
</Report>
```

What parser should be used!

Location of the stylesheet

XSLT - Example

```
<?xml version="1.0"?>
<StudentList xmlns:xsl=
   "http://www.w3.org/1999/XSL/Transform"
   xsl:version="1.0">
<xsl:copy-of select= "//Student/Name"/>
</StudentList >
```

Result:

```
<StudentList>
<Name><First>John</First><Last>Doe</Last></Name>
<Name>…….</Name>
……
</ StudentList>
```

XSLT – Instructions

- copy-of
- if-then
- for-each
- value-of
- …..

XSLT – Instructions

```
<?xml version="1.0"?>
<StudentList xmlns:xsl=
    "http://www.w3.org/1999/XSL/Transform"
    xsl:version="1.0">
<xsl:for-each select= "//Student">
  <xsl:if test="count (CrsTaken) &gt; 1">
     <FullName> <xsl:value-of select="*/Last"/>,
                      <xsl:value-of select="*/First"/>
      </FullName>   </xsl:if>
</xsl:for-each>
</StudentList >
```

XSLT – Instructions

```
<?xml version="1.0"?>
<StudentList xmlns:xsl=
    "http://www.w3.org/1999/XSL/Transform"
    xsl:version="1.0">
<xsl:for-each select= "//Student">
  <xsl:if test="count (CrsTaken) &gt; 1">
     <FullName> <xsl:value-of select="*/Last"/>,
                      <xsl:value-of select="*/First"/>
      </FullName>   </xsl:if>
</xsl:for-each>
</StudentList >
```

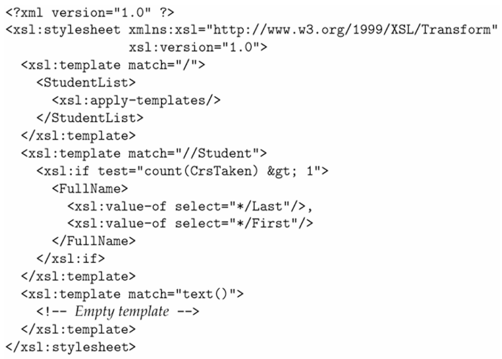
Result:

```
<StudentList>
   <FullName>
   John, Doe
   …..
   …..
  </FullName>
</StudentList>
```

XSLT – Template

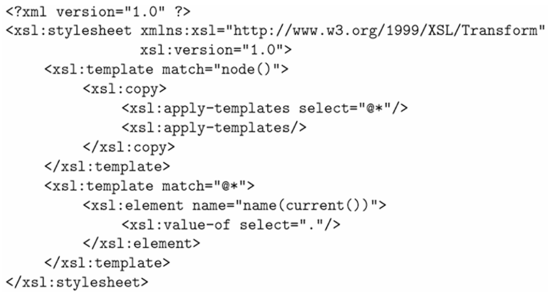
- Recursive traversal of the structures of the document
- Often defined recursively
- Algorithm for processing a XSLT template (book)

Figure 17.12
Recursive stylesheet.



```
<?xml version="1.0" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xsl:version="1.0">
  <xsl:template match="/">
    <StudentList>
      <xsl:apply-templates/>
    </StudentList>
  </xsl:template>
  <xsl:template match="//Student">
    <xsl:if test="count(CrsTaken) &gt; 1">
      <FullName>
        <xsl:value-of select="*/Last"/>,
        <xsl:value-of select="*/First"/>
      </FullName>
    </xsl:if>
  </xsl:template>
  <xsl:template match="text()">
    <!-- Empty template -->
  </xsl:template>
</xsl:stylesheet>
```

Figure 17.14
XSLT stylesheet that converts attributes into elements.



```
<?xml version="1.0" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xsl:version="1.0">
     <xsl:template match="node()">
          <xsl:copy>
               <xsl:apply-templates select="@*"/>
               <xsl:apply-templates/>
          </xsl:copy>
     </xsl:template>
     <xsl:template match="@*">
          <xsl:element name="name(current())">
               <xsl:value-of select="."/>
          </xsl:element>
     </xsl:template>
</xsl:stylesheet>
```

XQuery

- Syntax similar to SQL

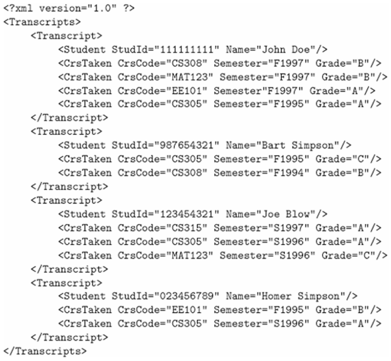
  FOR *variable declaration*

  WHERE *condition*

  RETURN *result*

Figure 17.15

Transcripts at http://xyz.edu/transcripts.xml.



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

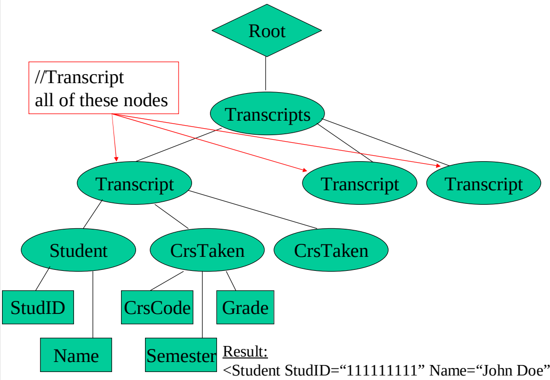
XQuery - Example

FOR $t IN
  document("http://xyz.edu/transcripts.xml")
  //Transcript
WHERE $t/CrsTaken/@CrsCode = "MA123"
RETURN $t/Student

Declare $t and its range

Find all transcripts containing "MA123"

Return the set of Student's elements of those transcripts



Result:
<Student StudID="111111111" Name="John Doe"/>
<Student StudID="123456789" Name="Joe Blow"/>

Putting it in well-formed XML

<StudentList>
(FOR $t IN
  document("http://xyz.edu/transcripts.xml")
  //Transcript
WHERE $t/CrsTaken/@CrsCode = "MA123"
RETURN $t/Student
)
</StudentList>

Figure 17.16

Construction of class rosters from transcripts: first try.

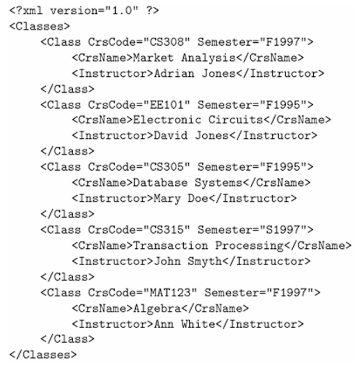
```
FOR $c IN distinct(document("http://xyz.edu/transcripts.xml")
                                     //CrsTaken)
RETURN <ClassRoster CrsCode=$c/@CrsCode Semester=$c/@Semester>
    (
        FOR $t IN document("http://xyz.edu/transcripts.xml")
                                     //Transcript
        WHERE $t/CrsTaken/@CrsCode = $c/@CrsCode
              AND $t/CrsTaken/@Semester = $c/@Semester
        RETURN
              $t/Student
              SORTBY($t/Student/@StudId)
    )
</ClassRoster>
SORTBY($c/@CrsCode)
```

For each class $c, find the students attending the class and output his information
=Ł output one class roster for each *CrsTaken* node Ł possibly more than one if different students get different grade

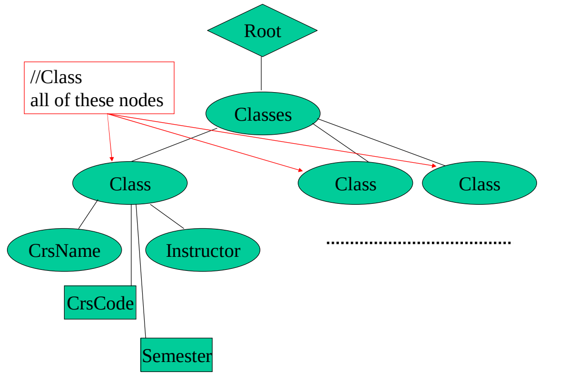
Fix ?

- Assume that the list of classes is available – write a different query
- Use the filter operation

Figure 17.17
Classes at http://xyz.edu/classes.xml.



```
<?xml version="1.0" ?>
<Classes>
    <Class CrsCode="CS308" Semester="F1997">
        <CrsName>Market Analysis</CrsName>
        <Instructor>Adrian Jones</Instructor>
    </Class>
    <Class CrsCode="EE101" Semester="F1995">
        <CrsName>Electronic Circuits</CrsName>
        <Instructor>David Jones</Instructor>
    </Class>
    <Class CrsCode="CS305" Semester="F1995">
        <CrsName>Database Systems</CrsName>
        <Instructor>Mary Doe</Instructor>
    </Class>
    <Class CrsCode="CS315" Semester="S1997">
        <CrsName>Transaction Processing</CrsName>
        <Instructor>John Smyth</Instructor>
    </Class>
    <Class CrsCode="MAT123" Semester="F1997">
        <CrsName>Algebra</CrsName>
        <Instructor>Ann White</Instructor>
    </Class>
</Classes>
```



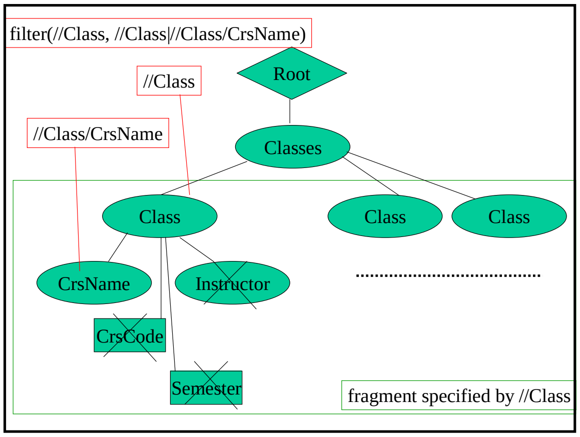
See Pg. 604 for XQuery (next slide)

```
FOR $c IN document("http://xyz.edu/classes.xml")//Class
RETURN
   <ClassRoster CrsCode=$c/@CrsCode Semester=$c/@Semester>
   $c/CrsName   $c/Instructor
   (FOR $t IN document("http://xyz.edu/transcripts.xml")//Transcript
        WHERE $t/CrsTaken/@CrsCode =  $c/@CrsCode
        RETURN $t/Student
        SORTBY($t/Student/@StudID)
    )
    </ClassRoster>
    SORTBY($c/@CrsCode)
```

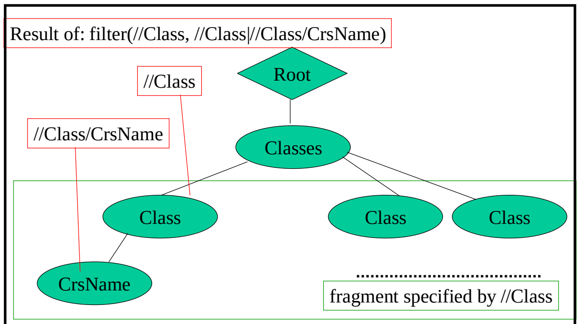
Give the "correct" result:
All ClassRoster, each only once

Filtering

- Syntax: filter(*argument1*, *argument2*)
- Meaning: return a document fragment obtained by
  - deleting from the set of nodes specified by *argument1* the nodes that do not occur in *argument2*
  - reconnecting the remaining nodes according to the child-parent relationship of the document specified by *argument1*



Result of: filter(//Class, //Class|//Class/CrsName)



Result:
<Class><CrsName>Market Analysis</CrsName></Class>
<Class><CrsName>Electronic Circuits </CrsName></Class>
…….

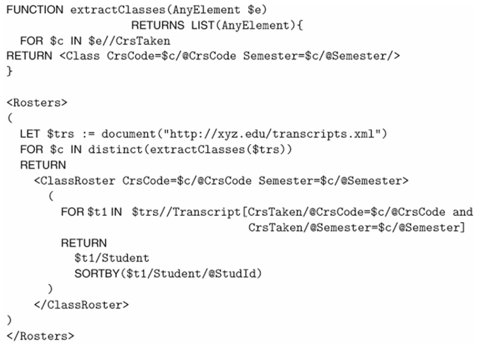
```
LET $trs:=document("http://xyz.edu/transcripts.xml")//Transcript
LET $ct:=$trs/CrsTaken
FOR $c IN  distinct(filter($ct, $ct|$ct/@CrsCode|$ct/@Semester))
RETURN
  <ClassRoster CrsCode=$c/@CrsCode Semester=$c/@Semester>
  (FOR $t IN $trs
      WHERE $t/CrsTaken/@CrsCode =  $c/@CrsCode AND
                 $t/CrsTaken/@Semester = $c/@Semester
       RETURN $t/Student
       SORTBY($t/Student/@StudID))
   </ClassRoster>
   SORTBY($c/@CrsCode)
```

Give the “correct” result:
All ClassRoster, each only once

Advances Features

- User-defined functions
- XQuery and Data types
- Grouping and aggregation

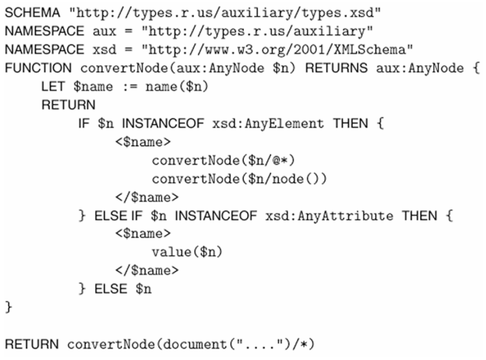
Figure 17.18
Class rosters constructed with user-defined functions.



```
FUNCTION extractClasses(AnyElement $e)
                  RETURNS LIST(AnyElement){
  FOR $c IN $e//CrsTaken
RETURN <Class CrsCode=$c/@CrsCode Semester=$c/@Semester/>
}

<Rosters>
(
  LET $trs := document("http://xyz.edu/transcripts.xml")
  FOR $c IN distinct(extractClasses($trs))
  RETURN
    <ClassRoster CrsCode=$c/@CrsCode Semester=$c/@Semester>
      (
        FOR $t1 IN  $trs//Transcript[CrsTaken/@CrsCode=$c/@CrsCode and
                                     CrsTaken/@Semester=$c/@Semester]
        RETURN
          $t1/Student
          SORTBY($t1/Student/@StudId)
      )
    </ClassRoster>
)
</Rosters>
```

Figure 17.19
XQuery transformation that does the same work as the stylesheet in Figure 17.14.



```
SCHEMA "http://types.r.us/auxiliary/types.xsd"
NAMESPACE aux = "http://types.r.us/auxiliary"
NAMESPACE xsd = "http://www.w3.org/2001/XMLSchema"
FUNCTION convertNode(aux:AnyNode $n) RETURNS aux:AnyNode {
     LET $name := name($n)
     RETURN
          IF $n INSTANCEOF xsd:AnyElement THEN {
               <$name>
                    convertNode($n/@*)
                    convertNode($n/node())
               </$name>
          } ELSE IF $n INSTANCEOF xsd:AnyAttribute THEN {
               <$name>
                    value($n)
               </$name>
          } ELSE $n
}

RETURN convertNode(document("....")/*)
```