XML: eXtensible Markup Language

Slides are based on slides from

Database System Concepts
©Silberschatz, Korth and Sudarshan
See www.db-book.com for conditions on re-use

Many examples are from www.w3schools.com

http://www.w3schools.com/xml/
http://www.w3schools.com/dtd/
http://www.w3schools.com/schema/

Datamodels

- RDBMS: atomic elements stored in tables
- OODBMS (ODL): data stored in classes. (references, class extents)

- Data models describe the form in which data can be stored and operated.
- Data models must be generic. (want to apply a DBMS for lots of different databases)

- Must provide enough structure to provide operations (such as queries)

Datamodel for Files?

- Files: sequence of Bytes.
- Very generic.

- Files are used to store information.
- Files are often used to exchange data
  - between businesses
  - between departments
  - between databases

- Hundreds of different file formats
  - (.doc, .rtf, .pdf, and 100s none of us will ever hear of)

- Each of these file formats needs tools to process them.
  - validation, translate into object structure, add data into database, extract information, translate into another file format.

- Need to implement these tools from scratch for each of them.

Solution: XML

- XML acts like a data model for files.

- Allows to describe file formats.

- There are lots of tools which allow to process XML files.
  - validators
  - parser
  - query languages

- Instead of designing a new file formal from scratch, we base it on XML. Then we have all these tools available.

- Can store data in XML files and query them.

XML Example

```
<bank>
   <account>
      <account-number> A-101     </account-number>
      <branch-name>      Downtown </branch-name>
      <balance>       500         </balance>
   </account>
   <depositor>
      <account-number> A-101    </account-number>
      <customer-name> Johnson </customer-name>
   </depositor>
</bank>
```

start tag

end tag

- **human and machine readable**
- **self documenting**
- **based on nested tags**
- **Unlike in HTML, you can define your own tags**

Structure of XML Data

- **Tag**: label in angle brackets for a section of data
- **Element**: section of data beginning with start tag <*tagname*> and ending with matching end tag </*tagname*>

```
<tagname>
   <!-- comment, place the content of your
        element here -->
</tagname>
```

- Can invent our own tags.

```
<ca-super-tag>  </ca-super-tag>
```

- Every document must have a single top-level element

Structure of XML Data

- Elements must be properly nested.

```
<bank>
    <customer>
        <customer-name>    Hayes  </customer-name>
        <customer-street> Main  </customer-street>
        <customer-city>    Harrison  </customer-city>
        <account>
            <account-number> A-102 </account-number>
            <branch-name>    Perryridge </branch-name>
            <balance>        400 </balance>
        </account>
        <account>
            <account-number> A-205 </account-number>
            <branch-name>    Oxford street </branch-name>
            <balance>        329 </balance>
        </account>
    </customer>
</bank>
```


Improper Nested Elements

```
<bank>
    <customer>
        <customer-name>    Hayes  </customer-name>
        <customer-street> Main  </customer-street>
        <customer-city>    Harrison  </customer-city>
        <account>
            <account-number> A-102 </account-number>
            <branch-name>    Perryridge </branch-name>
            <balance>        400 </balance>
        <account>
            <account-number> A-205 </account-number>
            <branch-name>    Oxford street </branch-name>
            <balance>        329 </balance>
        </account>
</bank2>
```

Attributes

- Elements can have **attributes**

```
<account acct-type = "checking" >
    <account-number> A-102   </account-number>
    <branch-name> Perryridge </branch-name>
    <balance>      400       </balance>
</account>
```

- Attributes are specified by *name=value* pairs inside the starting tag of an element
- An element may have several attributes, but each attribute name can only occur once

```
<account  acct-type = "checking"  monthly-fee= "5">
```


Attributes vs. Subelements

- Distinction between subelement and attribute
  - Same information can be represented in two ways
    -
```
<account  account-number = "A-101" >
   ...
</account>
```
    -
```
<account>
    <account-number> A-101 </account-number>
    ...
</account>
```
  - **Good style: use attributes for identifiers of elements (see later), and use subelements for content.**

More on XML Syntax

- Elements without subelements can be abbreviated by ending the start tag with a /> and deleting the end tag

<account number="A-101" branch="Perryridge" balance="200" />

<emptyElement/>

bad style! should use subelements here. But we needed an example

- To store string data that may contain tags, without the tags being interpreted as subelements, use CDATA as below
  - <![CDATA[<account>]]>
    - Here, <account> is treated as just strings

Well-formed Documents

- A document which conforms to the XML syntax is called **well-formed**.
  - all tag properly nested
  - single top level element
  - ...
- How to check this?
  - load the XML-file in your web browser.
  - Use NetBeans
  - many more tools ...

Namespaces

- XML is often use for data exchange between organizations.
- Same tag name may have different meaning in different organizations, causing confusion on exchanged documents.
- Need *unique* names for elements.
- Better solution: use **unique-name:element-name.**
- **unique-name is called namespace.**
- Trick to obtain unique **unique-names**: use URIs.
- Example: University of Liverpool defines a `student` tag: `http://www.liv.ac.uk:student`

Namespaces

- Avoid using long unique names all over document by using defining a short namespace ***prefix*** within a document.
- FB:branch is just an abbreviation to make the file more readable.

```
<bank Xmlns:FB='http://www.FirstBank.com'>
        …
    <FB:branch>
        <FB:branchname> Downtown </FB:branchname>
        <FB:branchcity> Brooklyn </FB:branchcity>
    </FB:branch>
  …
</bank>
```

- The full (qualified) name of the `branch` element is

  `http://www.firstbank.com:branch`

Default Namespaces

```
<bank Xmlns:FB="http://www.FirstBank.com"
      Xmlns="http://www.gov.uk">
      ...
    <FB:branch>
        <FB:branchname> Downtown </FB:branchname>
        <FB:branchcity> Brooklyn </FB:branchcity>
    </FB:branch>
    <prime-minister> Tony Blair </prime-minister>
    ...
</bank>
```

qualified name is
http://www.gov.uk:prime-minister

XML Document Schema

- **When we build a database, we first design the database schema.**
- **Database schemas constrain what information can be stored in the database.**
  - names of tables and attributes
  - data types of attributes,
  - the order they appear in the table
- **By requiring that the data conforms to the schema, we give data a meaning.**
- **XML documents are not required to have a schema.**
- **In practice, they are very important.**
  - e.g. when defining a new *file format* based on XML
  - or for data exchange

XML Schema Mechanisms

- **Document Type Definition (DTD)**
  - Widely used
- **XML Schema**
  - Newer, increasing use
- A document that confirms with its DTD or XML schema is called **valid**.
- There exist tools to check whether a XML-file is valid.
  - www.xmlvalidation.com
  - NetBeans
  - many others
- A validator checks also whether the XML-file is well-formed.

Note.xml with DTD included

```
<?xml version="1.0"?>
<!DOCTYPE note [
   <!ELEMENT note (to, from, heading, body)>
   <!ELEMENT to (#PCDATA)>
   <!ELEMENT from (#PCDATA)>
   <!ELEMENT heading (#PCDATA)>
   <!ELEMENT body (#PCDATA)>
]>
<note>
    <to>      Tove    </to>
    <from>    Jani     </from>
    <heading> Reminder </heading>
    <body>Don't forget me this weekend!</body>
</note>
```

DTD

Document Type Definition (DTD)

- The type of an XML document can be specified using a DTD
- DTD constraints structure of XML data
  - What elements can occur
  - What attributes can/must an element have
  - What subelements can/must occur inside each element, and how many times.
- DTD does not constrain data types
  - All values represented as strings in XML
- DTD syntax
  - `<!ELEMENT element (subelements-specification) >`
  - `<!ATTLIST element (attributes) >`

Element Specification in DTD

- Subelements can be specified as
  - names of elements, or
  - #PCDATA (parsed character data), i.e., character strings
  - EMPTY (no subelements) or ANY (anything can be a subelement)
- Example

  <! ELEMENT depositor (customer-name account-number)>
  <! ELEMENT customer-name (#PCDATA)>
  <! ELEMENT account-number (#PCDATA)>
- Subelement specification may have regular expressions

  <!ELEMENT bank ((account | customer | depositor)+)>
  - Notation:
    - "|" - alternatives
    - "+" - 1 or more occurrences
    - "*" - 0 or more occurrences

Bank DTD

```
<!ELEMENT bank ( ( account | customer | depositor)+)>
<!ELEMENT account (branch-name, balance)>
<!ELEMENT branch-name (#PCDATA)>
<!ELEMENT balance (#PCDATA)>
<!ELEMENT customer
      (customer-name, customer-street, customer-city)>
<!ELEMENT customer-name (#PCDATA)>
<!ELEMENT customer-street (#PCDATA)>
<!ELEMENT customer-city (#PCDATA)>
<!ELEMENT depositor (customer-name, account-number)>
```


Attribute Specification in DTD

- Attribute specification : for each attribute
  - Name
  - Type of attribute
    - CDATA
    - ID (identifier) or IDREF (ID reference) or IDREFS (multiple IDREFs)
      - more on this later
  - Whether
    - mandatory (#REQUIRED)
    - has a default value (value),
    - or neither (#IMPLIED)
- Examples
  - <!ATTLIST account acct-type CDATA “checking”>
  - <!ATTLIST customer
    customer-id ID # REQUIRED
    accounts IDREFS # REQUIRED >

IDs and IDREFs

- An element can have at most one attribute of type ID
- The ID attribute value of each element in an XML document must be distinct
  - Thus the ID attribute value is an object identifier
- An attribute of type IDREF must contain the ID value of an element in the same document
- An attribute of type IDREFS contains a set of (0 or more) ID values. Each ID value must contain the ID value of an element in the same document

Note.xml with DTD included

```
<?xml version="1.0"?>
<!DOCTYPE note [
   <!ELEMENT note (to, from, heading, body)>
   <!ELEMENT to (#PCDATA)>
   <!ELEMENT from (#PCDATA)>
   <!ELEMENT heading (#PCDATA)>
   <!ELEMENT body (#PCDATA)>
]>
<note>
    <to>      Tove    </to>
    <from>    Jani     </from>
    <heading> Reminder </heading>
    <body>Don’t forget me this weekend!</body>
</note>
```

Note.xml with a reference to DTD

note.dtd

```
<!ELEMENT note (to, from, heading, body)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)>
```

note.xml

```
<?xml version="1.0"?>
<!DOCTYPE note SYSTEM
"c:\comp302\note.dtd">
<note>
    <to>      Tove    </to>
    <from>    Jani     </from>
    <heading> Reminder </heading>
    <body>Don't forget me this
weekend!</body>
</note>
```

Reference to DTD in c:\comp302\note.dtd

Note.xml with a reference to DTD

note.dtd

```
<!ELEMENT note (to, from, heading, body)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)>
```

note.xml

```
<?xml version="1.0"?>
<!DOCTYPE note SYSTEM
"http://www.csc.liv.ac.uk/~christoph/note.dtd">
<note>
    <to>      Tove    </to>
    <from>    Jani     </from>
    <heading> Reminder </heading>
    <body>Don't forget me this weekend!</body>
</note>
```

refernce to DTD at URL

XML data with ID and IDREF attributes

```
<bank2>
   <account account-number="A-401" owners="C100 C102">
      <branch> Downtown </branch>
      <balance>       500 </balance>
   </account>
   <account account-number="A-402" owners="C102">
      <branch> Downtown </branch>
      <balance>  8000 </balance>
   </account>
   <customer customer-id="C100" accounts="A-401">
      <customer-name>   Joe       </customer-name>
      <customer-street>  Monroe   </customer-street>
      <customer-city>     Madison </customer-city>
   </customer>
   <customer customer-id="C102" accounts="A-401 A-402">
      <customer-name>   Mary     </customer-name>
      <customer-street>   Erin      </customer-street>
      <customer-city>      Newark </customer-city>
   </customer>
</bank2>
```


Bank DTD with Attributes

```
<!ELEMENT bank2 ( ( account | customer )+)>
<!ELEMENT account (branch, balance)>
<!ATTLIST account
    account-number ID          #REQUIRED
    owners         IDREFS      #REQUIRED>
<!ELEMENT customer
      (customer-name, customer-street, customer-city)>
<!ATTLIST customer
      customer-id     ID          #REQUIRED
      accounts        IDREFS      #REQUIRED>
<!ELEMENT branch (#PCDATA)>
<!ELEMENT balance (#PCDATA)>
<!ELEMENT customer-name (#PCDATA)>
<!ELEMENT customer-street (#PCDATA)>
<!ELEMENT customer-city (#PCDATA)>
```

Limitations of DTDs

- No typing of text elements and attributes
  - All values are strings, no integers, reals, etc.
- Difficult to specify unordered sets of subelements
  - Order is usually irrelevant in databases
  - (A | B)* allows specification of an unordered set, but
    - Cannot ensure that each of A and B occurs only once
- IDs and IDREFs are untyped
  - The *owners* attribute of an account may contain a reference to another account, which is meaningless
    - *owners* attribute should ideally be constrained to refer to customer elements

XML Schema

- XML Schema is a more sophisticated schema language which addresses the drawbacks of DTDs. Supports
  - Typing of values
    - E.g. integer, string, etc
    - Also, constraints on min/max values
  - User defined types
  - Is itself specified in XML syntax, unlike DTDs
    - More standard representation, but verbose
  - Is integrated with namespaces
  - Many more features
    - List types, uniqueness and foreign key constraints, inheritance ..
- BUT: significantly more complicated than DTDs, not yet widely used.

Note.xsd (xml schema definition)

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           targetNamespace="http://www.liv.ac.uk">
        <xs:element name="note">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="to" type="xs:string"/>
                <xs:element name="from" type="xs:string"/>
                <xs:element name="heading" type="xs:string"/>
                <xs:element name="body" type="xs:string"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```


Note.xml

```
<?xml version="1.0"?>
<note    xmlns="http://www.liv.ac.uk"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.liv.ac.uk note.xsd">

    <to>      Tove</to>
    <from>    Jani</from>
    <heading> Reminder </heading>
    <body>Don't forget me this weekend!</body>
</note>
```

XML schema for http://www.liv.ac.uk namespace (same path as xml file)

Schema could be stored on the web:

```
xsi:schemaLocation=
"http://www.liv.ac.uk http://www.liv.ac.uk/~christoph/note.xsd"
```

Can we translate XML to a RDBMS?

```
<bank2>
  <account a-number="A-401" a-owners="C100 C102">
    <a-branch> Downtown </a-branch>
    <a-balance>        500 </a-balance>
  </account>
  <account a-number="A-402" a-owners ="C102">
    <a-branch> Downtown </a-branch>
    <a-balance>  8000 </a-balance>
  </account>
  <customer c-id="C100" c-accounts="A-401">
    <c-name>   Joe       </c-name>
    <c-street>  Monroe   </c-street>
    <c-city>     Madison </c-city>
  </customer>
  <customer c-id="C102" c-accounts="A-401 A-402">
    <c-name>   Mary     </c-name>
    <c-street>   Erin      </c-street>
    <c-city>      Newark </c-city>
  </customer>
</bank2>
```


XML to RDBMS Mapping

```
<bank2>
  <account a-number="A-401" a-owners="C100 C102">
    <a-branch> Downtown </a-branch>
    <a-balance>        500 </a-balance>
  </account>
  <account a-number="A-402" a-owners ="C102">
    <a-branch> Downtown </a-branch>
    <a-balance>  8000 </a-balance>
  </account>
  <customer c-id="C100" c-accounts="A-401">
    <c-name>   Joe       </c-name>
    <c-street>  Monroe   </c-street>
    <c-city>     Madison </c-city>
  </customer>
  <customer c-id="C102" c-accounts="A-401 A-402">
    <c-name>   Mary     </c-name>
    <c-street>   Erin      </c-street>
    <c-city>      Newark </c-city>
  </customer>
</bank2>
```

Accounts

a-number	a-branch	a-balance
A-401	Downtown	500
A-402	Downtown	8000

Customers

c-id	c-name	c-street	c-city
C100	Joe	Monroe	Madison
C102	Mary	Erin	Newark

Owners

a-number	customer
A-401	C100
A-401	C102
A-402	C102

Do not need a **accounts** relation, because it is ***inverse*** to owners.

Could a DTD be useful here?

Mapping based on DTD

```
<!ELEMENT bank2 ( ( account | customer )+)>
<!ELEMENT account (a-branch, a-balance)>
  <!ATTLIST account
      a-number        ID          #REQUIRED
      a-owners        IDREFS      #REQUIRED>
  <!ELEMENT a-branch  (#PCDATA)>
  <!ELEMENT a-balance (#PCDATA)>

<!ELEMENT customer (c-name, c-street, c-city)>
  <!ATTLIST customer
              c-id          ID       #REQUIRED
          c-accounts        IDREFS   #REQUIRED>
  <!ELEMENT         c-name (#PCDATA)>
  <!ELEMENT       c-street (#PCDATA)>
  <!ELEMENT         c-city (#PCDATA)>
```

- Relations: ID and IDREFs behave like keys in RDBMS.
  - accounts(a-number, a-branch, a-balance)
  - customers(c-id, c-name, c-street, c-city)
  - owners(c-id, a-num)

FOREIGN KEY c-id REFERENCES customer(c-id)
FOREIGN KEY a-num REFERENCES accounts(a-number)

XML Mapping: Example 2

```
<bank>
    <customer>
        <c-name>   Hayes      </c-name>
        <c-street> Main       </c-street>
        <c-city>   Harrison   </c-city>
        <c-phone>  0151- 5464 </c-phone>
        <c-phone>  0151- 6578 </c-phone>
        <account>
            <a-number> A-102         </a-number>
            <a-branch> Perryridge    </a-branch>
            <balance>  400           </balance>
        </account>
        <account>
            <a-number> A-205         </a-number>
            <a-branch> Oxford street </a-branch>
            <balance>  329           </balance>
        </account>
    </customer>
    <customer>
         <c-name>   Fox      </c-name>
            ...
    </customer>
</bank>
```

XML Mapping: Example 2

```
<bank>
    <customer>
        <c-name>   Hayes      </c-name>
        <c-street> Main       </c-street>
        <c-city>   Harrison   </c-city>
        <c-phone>  0151- 5464 </c-phone>
        <c-phone>  0151- 6578 </c-phone>
        <account>
            <a-number> A-102         </a-number>
            <a-branch> Perryridge    </a-branch>
            <balance>  400           </balance>
        </account>
        <account>
            <a-number> A-205         </a-number>
            <a-branch> Oxford street </a-branch>
            <balance>  329           </balance>
        </account>
    </customer>
    <customer>
            ...
    </customer>
</bank>
```

Assume every account belongs to exactly one customer

```
<!ELEMENT bank (customer+)>
<!ELEMENT customer (c-name,c-street,c-city,
                    c-phone*,account+)>
<!ELEMENT   c-name (#PCDATA)>
<!ELEMENT c-street (#PCDATA)>
<!ELEMENT   c-city (#PCDATA)>
<!ELEMENT  c-phone (#PCDATA)>
<!ELEMENT account (a-number,a-branch,
                   a-balance)>
<!ELEMENT a-number (#PCDATA)>
<!ELEMENT a-branch (#PCDATA)>
<!ELEMENT a-balance (#PCDATA)>
```


XML Mapping: Example 2

```
<bank>
    <customer>
        <c-name>   Hayes      </c-name>
        <c-street> Main       </c-street>
        <c-city>   Harrison   </c-city>
        <c-phone>  0151- 5464 </c-phone>
        <c-phone>  0151- 6578 </c-phone>
        <account>
            <a-number> A-102         </a-number>
            <a-branch> Perryridge    </a-branch>
            <balance>  400           </balance>
        </account>
        <account>
            <a-number> A-205         </a-number>
            <a-branch> Oxford street </a-branch>
            <balance>  329           </balance>
        </account>
    </customer>
    <customer>
            ...
    </customer>
</bank>
```

Customers

c-id	c-name	c-street	c-city
C100	Hayes	Main	Harrison
C101	Fox	...	...

C-phone

c-id	phone
C100	0151-5464
C100	0151-6578

Accounts

a-number	a-branch	a-balance	a-owner
A-102	Perryridge	500	C100
A-205	Oxford street	8000	C100

Assuming that every account belongs to exactly 1 customer

Mapping XML Data to Relations

- **Create a relation for each element type that appears more than once within its parent.**
- The attributes of the relation are:
  - Introduce an id attribute if needed: serves as primary key.
  - For #PCDATA subelements, store the text as attribute value.
  - A relation attribute corresponding to each element attribute.
  - For complex subelements, break subelement in multiple attributes. (e.g. address(street, city, zip) -> addr-street, addr-city, addr-zip)
  - A parent-id attribute to keep track of parent element (foreign key). (not needed for children of top-level element)

Mapping XML Data to Relations

- Benefits:
  - Efficient storage
  - All advantages of RDBMS
    - Recovery, Transactions
  - Can translate XML queries into SQL, execute efficiently, and then translate SQL results back to XML
  - Can output results of SQL queries as XML documents.
- Can generate DTDs and XML schemas from relational schemas and vise versa.
- Need to know DTD or XML schema

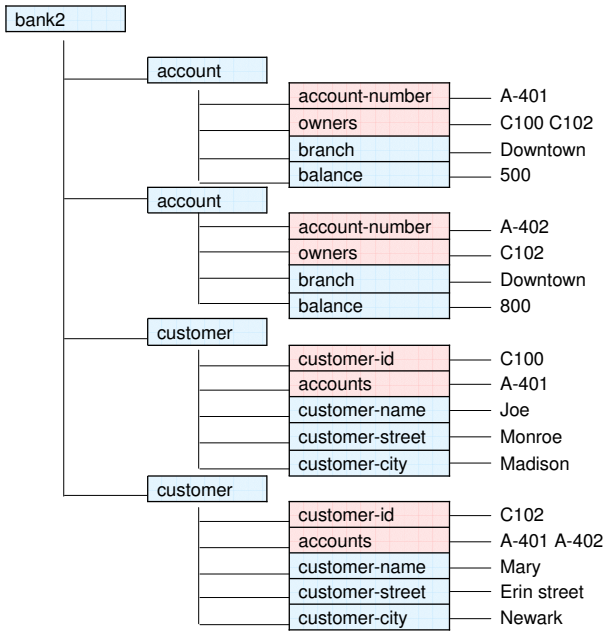
Querying and Transforming XML Data

- Translation of information from one XML schema to another
- Querying on XML data
- Above two are closely related, and handled by the same tools
- Standard XML querying/translation languages
  - XPath
    - Simple language consisting of path expressions
  - XSLT
    - Simple language designed for translation from XML to XML and XML to HTML
  - XQuery
    - An XML query language with a rich set of features
- Wide variety of other languages have been proposed, and some served as basis for the Xquery standard
  - XML-QL, Quilt, XQL, …

Tree Structure of XML Documents

```
<bank2>
  <account account-number="A-401" owners="C100 C102">
    <branch> Downtown </branch>
    <balance>        500 </balance>
  </account>
  <account account-number="A-402" owners="C102">
    <branch> Downtown </branch>
    <balance>  8000 </balance>
  </account>
  <customer customer-id="C100" accounts="A-401">
    <customer-name>   Joe       </customer-name>
    <customer-street>  Monroe   </customer-street>
    <customer-city>     Madison </customer-city>
  </customer>
  <customer customer-id="C102" accounts="A-401 A-402">
    <customer-name>   Mary     </customer-name>
    <customer-street>   Erin     </customer-street>
    <customer-city>      Newark </customer-city>
  </customer>
</bank2>
```


Tree Structure of XML Documents



Tree Model of XML Data

- Query and transformation languages are based on a **tree model** of XML data.
- An XML document is modeled as a tree, with **nodes** corresponding to elements and attributes
  - The top-level element is the root of the tree.
  - Element nodes have children nodes, which can be attributes or subelements.
  - Text in an element is modeled as a text node child of the element.
  - Children of a node are ordered according to their order in the XML document.
  - Element and attribute nodes (except for the root node) have a single parent, which is an element node.
- We use the terminology of nodes, children, parent, siblings, ancestor, descendant, etc., which should be interpreted in the above tree model of XML data.

XPath

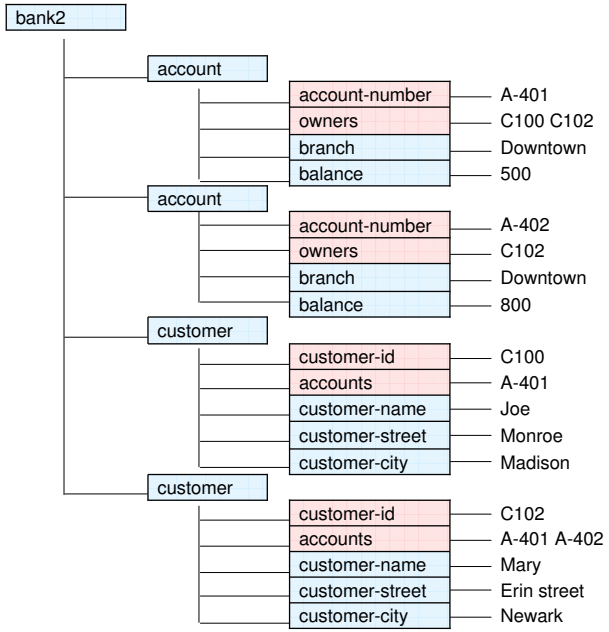
- XPath is used to address (select) parts of documents using **path expressions**
- A path expression is a sequence of steps separated by "/"
  - Think of file names in a directory hierarchy
- Result of path expression: set of values that along with their containing elements/attributes match the specified path
- E.g. /bank-2/customer/customer-name evaluated on the bank-2 data we saw earlier returns

  <customer-name>Joe</customer-name>

  <customer-name>Mary</customer-name>
- E.g. /bank-2/customer/customer-name/text( )

  returns the same names, but without the enclosing tags

X path query example: /bank2/account



Idea: query elements from the XML document by specifying their access path in the tree

Returns all elements reachable from the root via a path /bank2/account

X path query example: /bank2/account

```
<bank2>
   <account account-number="A-401" owners="C100 C102">
      <branch> Downtown </branch>
      <balance>        500 </balance>
   </account>
   <account account-number="A-402" owners="C102">
      <branch> Downtown </branch>
      <balance>  8000 </balance>
   </account>
    <customer customer-id="C100" accounts="A-401">
      <customer-name>  Joe       </customer-name>
      <customer-street> Monroe  </customer-street>
      <customer-city>    Madison </customer-city>
    </customer>
    <customer customer-id="C102" accounts="A-401 A-402">
      <customer-name>  Mary     </customer-name>
      <customer-street>  Erin     </customer-street>
      <customer-city>     Newark </customer-city>
    </customer>
</bank2>
```


XPath (Cont.)

- The initial "/" denotes root of the document (above the top-level tag)
- Path expressions are evaluated left to right
  - Each step operates on the set of instances produced by the previous step
- Selection predicates may follow any step in a path, in [ ]
  - E.g. /bank-2/account[balance > 400]
    - returns account elements with a balance value greater than 400
    - /bank-2/account[balance] returns account elements containing a balance subelement
- Attributes are accessed using "@"
  - E.g. /bank-2/account[balance > 400]/@account-number
    - returns the account numbers of those accounts with balance > 400
  - IDREF attributes are not dereferenced automatically (more on this later)

Functions in XPath

- XPath provides several functions
  - The function count() at the end of a path counts the number of elements in the set generated by the path
    - E.g. /bank-2/account[customer/count() > 2]
      - Returns accounts with > 2 customers
  - Also function for testing position (1, 2, ..) of node w.r.t. siblings
- Boolean connectives and and or and function not() can be used in predicates
- IDREFs can be referenced using function id()
  - id() can also be applied to sets of references such as IDREFS and even to strings containing multiple references separated by blanks
  - E.g. /bank-2/account/id(@owner)
    - returns all customers referred to from the owners attribute of account elements.

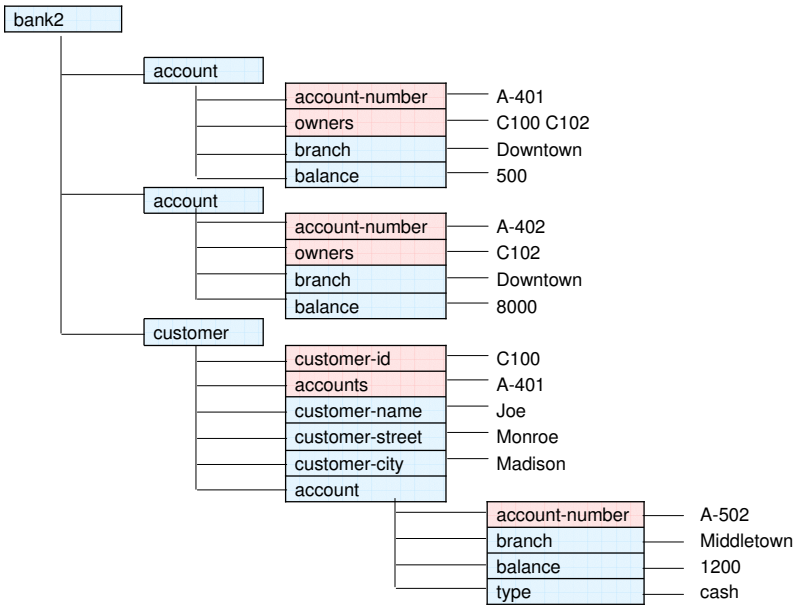
More XPath Features

- Operator "|" used to implement union
  - E.g. /bank-2/account/id(@owner) | /bank-2/loan/id(@borrower)
    - gives customers with either accounts or loans
    - However, "|" cannot be nested inside other operators.
- "//" can be used to skip multiple levels of nodes
  - E.g. /bank-2//customer-name
    - finds any customer-name element *anywhere* under the /bank-2 element, regardless of the element in which it is contained.
- A step in the path can go to:

  parents, siblings, ancestors and descendants

  of the nodes generated by the previous step, not just to the children
  - "//", described above, is a short from for specifying "all descendants"
  - ".." specifies the parent.
  - We omit further details,

Tree Structure of XPath Example



XQuery

- XQuery is a general purpose query language for XML data
- FLOWR: XQuery uses a
  for … let … order by ... where .. return …
  syntax
  for ⇔ SQL from
  let ⇔ ??
  order by ⇔ SQL order by
  where ⇔ SQL where
  return ⇔ SQL select
- The output of an XQuery is an XML document
- for $x in ... return <c> {$x} </c>: for produces a c-element for every value of $x.
- let $x:= ... return <c> {$x} </c>: let produces only one c-element, containing the sequence for $x values.
- see examples.

FLWR Syntax in XQuery

- For clause uses XPath expressions, and variable in for clause ranges over values in the set returned by XPath
- Simple FLWR expression in XQuery
  - find all accounts with balance > 400, with each result enclosed in an <account-number> .. </account-number> tag

```
for     $x in /bank-2/account
let     $acctno := $x/@account-number
where $x/balance > 400
return <account-number> $acctno </account-number>
```

- Let clause not really needed in this query, and selection can be done In XPath. Query can be written as:

```
for $x in /bank-2/account[balance>400]
return <account-number> $x/@account-number
                                    </account-number>
```


Path Expressions and Functions

- Path expressions are used to bind variables in the for clause, but can also be used in other places
  - E.g. path expressions can be used in **let** clause, to bind variables to results of path expressions
- The function distinct( ) can be used to remove duplicates in path expression results
- The function **document(name)** returns root of named document
  - E.g. document("bank-2.xml")/bank-2/account
- Aggregate functions such as sum( ) and count( ) can be applied to path expression results
- XQuery does not support group by, but the same effect can be got by nested queries, with nested FLWR expressions within a **result** clause
  - More on nested queries later

Joins

- Joins are specified in a manner very similar to SQL

```
for $a  in  /bank/account,
    $c  in  /bank/customer,
    $d  in  /bank/depositor
where   $a/account-number = $d/account-number
    and $c/customer-name = $d/customer-name
return <cust-acct> $c $a </cust-acct>
```

- The same query can be expressed with the selections specified as XPath selections:

```
for $a in /bank/account
    $c in /bank/customer
    $d in /bank/depositor[
              account-number = $a/account-number and
              customer-name  = $c/customer-name]
return <cust-acct> $c $a</cust-acct>
```


Changing Nesting Structure

- The following query converts data from the flat structure for bank information into the nested structure used in bank-1

```
<bank-1>
    for $c in /bank/customer
    return
      <customer>
        $c/*
        for $d in /bank/depositor[customer-name = $c/customer-name],
            $a in /bank/account[account-number=$d/account-number]
        return $a
      </customer>
</bank-1>
```

- $c/* denotes all the children of the node to which $c is bound, without the enclosing top-level tag
- Exercise for reader: write a nested query to find sum of account balances, grouped by branch.

XQuery Path Expressions

- $c/text() gives text content of an element without any subelements/tags
- XQuery path expressions support the "–>" operator for dereferencing IDREFs
  - Equivalent to the id( ) function of XPath, but simpler to use
  - Can be applied to a set of IDREFs to get a set of results
  - June 2001 version of standard has changed "–>" to "=>"