

XML (part 1)

Chapitre 2: XML

Outline

1. Background
2. Structure of XML Data
3. XML Document Schema
 - 3.1. DTD
 - 3.2. XML Schema
4. XML Query Languages
 - 4.1. XPath
 - 4.2. XQuery
5. Application Program Interface

1. Background

- **eXtensible Markup Language (XML)** has its roots as a document markup language
 - tags enclosed in angle-brackets < >, used in pairs, <tag> and </tag> delimit start and end portion of document
 - derived from SGML, but simpler to use
 - defined by the WWW consortium (www.w3.org)
- XML (unlike HTML) does not have a predefined set of tags
 - set may be specialized as needed
 - key to XML's major role in **data representation and data exchange** (HTML used for document formatting)

1. Background

- Example: bank application, account & customer information as part of XML document

```
<bank>
  <account>
    <account-nb>A-101</account-nb>
    <branch-name>Downtown</branch-name>
    <balance>500</balance>
  </account>
  ...
  <customer>
    <customer-name>Johnson</customer-name>
    <customer-city>Palo Alto</customer-city>
  </customer>
  ...
```

1. Background

```
< depositor>
    < account-nb>A-101</ account-nb>
    < customer-name>Johnson</ customer-name>
</ depositor>
< depositor>
    < account-nb>A-102</ account-nb>
    < customer-name>Johnson</ customer-name>
    < access-date>11/08/2004</ access-date>
</ depositor>
...
</ bank>
```

1. Background

■ XML representation vs. storage of data in DB:

Cons:

- inefficient since tag names repeated throughout the doc.

Pros: when used to exchange data

- presence of tags makes the doc. **self-describing**
no need to consult the schema to understand the meaning of the doc.
- format of document not rigid
- variety of tools: browser softwares, DB tools

2. Structure of XML Data

- Fundamental construct in XML doc: **element**
element = start and end tags, and text between them (content)
- XML doc. must have a single **root** element
encloses all other elements in the doc.
ex: bank element
- Elements must **nest** properly
ex: <account>...<balance>...</balance>...</account>

2. Structure of XML Data

- Text may be mixed with the subelements of an element (**mixed content**)

<account>

 This account is seldom used.

 <account-nb>A-102</account-nb>

 <branch-name>Perryridge</branch-name>

 <balance>400</balance>

</account>

2. Structure of XML Data

- Ability to nest elements within other elements provides an alternative way to represent the information
 - ex: account elements nested within customer elements
 - easy to find all accounts of a customer
 - stores account elements redundantly if shared
 - nested representation widely used in XML data interchange applications to avoid joins

2. Structure of XML Data

```
<bank-1>
  <customer>
    <customer-name>Johnson</customer-name>
    <customer-city>Palo Alto</customer-city>
    <account>
      <account-nb>A-101</account-nb>
      <branch-name>Downtown</branch-name>
      <balance>500</balance>
    </account>
    ...
  </customer>
  ...
</bank-1>
```

2. Structure of XML Data

- An element may have **attributes**
 - **name = value** pairs before ">" in start-tag
 - attributes are strings, do not contain markup
 - attributes appear only once in a given tag, unlike subelements which may be repeated

```
<account account-type="savings">  
  <account-nb>A-102</account-nb>  
  <branch-name>Perryridge</branch-name>  
  <balance>400</balance>  
</account>
```

2. Structure of XML Data

- Distinction between subelement and attribute:
 - in doc. context: important
 - attribute is text that does not appear in the printed document
 - in DB context: less relevant
 - choice of representing data as attribute or subelement is arbitrary
 - attributes should be used to describe how to interpret data elements, and not contain data elements
- An **empty** element:
 - contains no subelements or text
 - `<element></element>`, or abbreviated as `<element/>`
 - can have attributes

2. Structure of XML Data

■ Namespace mechanism

- allows an organization to specify globally unique names to be used as element tags
- prepend each tag with the namespace
- doc. can have more than one namespace in the root element

```
<bank xmlns:FB="http://www.FirstBank.com/Def">
  ...
  <FB:branch>
    <FB:branchname>Downtown</FB:branchname>
    <FB:branchcity>Brooklyn</FB:branchcity>
  </FB:branch>
  ...
</bank>
```

2. Structure of XML Data

- First line of an XML document (.xml): **XML declaration**

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

- XML is case-sensitive

`<address>`, `<Address>`, `<ADDRESS>` are different

- Element names:

- may only start with either a letter or underscore
- then mixture of letter, number, underscore, dot, hyphen
- spaces not allowed in element names

- **Comments:**

- inserted between '`<!--`' and '`-->`', no nested comments

2. Structure of XML Data

■ Tools:

- ❑ Code in XML document must be processed by a small application called **parser**
 - ex: Microsoft XML parser (MSXML 4.0)
- ❑ XML **editors** (syntax checker and schema validator)
 - ex: XMLSpy, XMLwriter
- ❑ Internet Explorer Tools for Validating XML and Viewing XSLT Output
 - download iexmltls.exe, install (readme.txt in IEXMLTLS folder)
 - right-click on browser window: drop down menu with item 'Validate XML' (works only with DTDs)

3. XML Document Schema

- DBs have schemas
- XML doc. can be created with/without a schema
 - schema useful when XML doc. must be processed automatically
 - several schema formalisms: DTD (part of XML standard), XML Schema (more recently defined), ...
 - parser checks whether an XML doc. **conforms** to the associated schema

3. XML Document Schema

- An XML document is **well-formed** if:
 - it starts with an XML declaration
 - it has a root element
 - all tags have start and end tags, or end with ">"
 - tags nest properly
- An XML document is **valid** if:
 - it is well-formed
 - it conforms to the associated schema (if any)

3.1. DTD

- **Document Type Definition (DTD):**
 - ❑ optional part of an XML document (structure of the doc.)
 - ❑ list of rules: pattern of subelements that appear within an element
 - ❑ DTD can be given at the beginning of a document:
`<!DOCTYPE root-name [rules]>`
or in a separate file (.dtd), and referenced in the doc:
`<!DOCTYPE root-name SYSTEM "location">`
ex: `<!DOCTYPE bank SYSTEM "bank.dtd">`

3.1. DTD

- **Element declaration**: regular expression for the subelements of the element
 - **sequence** ‘,’ (order), **choice** ‘|’
 - **+**: one or more, *****: zero or more
 - ?**: zero or one, **none**: one

3.1. DTD

```
<!DOCTYPE bank [  
  <!ELEMENT bank (account | customer | depositor)+>  
  <!ELEMENT account (account-nb, branch-name, balance)>  
  <!ELEMENT customer (customer-name, customer-city)>  
  <!ELEMENT depositor (account-nb, customer-name,  
                        access-date?)>  
  <!ELEMENT account-nb (#PCDATA)>  
  <!ELEMENT branch-name (#PCDATA)>  
  <!ELEMENT balance (#PCDATA)>  
  <!ELEMENT customer-name (#PCDATA)>  
  <!ELEMENT customer-city (#PCDATA)>  
  <!ELEMENT access-date (#PCDATA)>  
]>
```

3.1. DTD

- **(#PCDATA)**: text data (parsed character data)

EMPTY: element has no content

ANY: no constraints on the subelements

- if the declaration of an element is missing in the DTD, it is of type ANY

- Mixed content:

<!ELEMENT student (**#PCDATA | name | id**)*>

3.1. DTD

- Attributes also declared in a DTD
 - no order (unlike subelements)
 - **Attribute declaration**: type declaration + default declaration
- Type declaration:
 - **CDATA** (character data, string)
 - enumeration type (list of values)
 - **ID** (unique identifier)
 - a value that occurs on an ID attribute must not occur on *any other* ID attribute in the *same* document
 - at most one attribute of type ID for an element
 - **IDREF** and **IDREFS** (reference to element(s))
 - must take a value taken on *some* ID attribute in the same document
 - IDREFS: a list of references separated by space

3.1. DTD

■ Default declaration:

- ❑ **#IMPLIED**: optional, with no default value
- ❑ **defaultValue**: optional, with a default value
- ❑ **#REQUIRED**: a value must be specified for the attribute in each element
- ❑ **#FIXED defaultValue**: constant value

<!ATTLIST depositor access-date CDATA #IMPLIED>

<!ATTLIST account account-type (checking | savings) "checking">

3.1. DTD

```
<!DOCTYPE bank-2 [  
  <!ELEMENT bank-2 (account | customer)+>  
  <!ELEMENT account (branch-name, balance)>  
    <!ATTLIST account  account-nb ID #REQUIRED>  
  <!ELEMENT customer (customer-name, customer-city)>  
    <!ATTLIST customer  
      customer-id ID #REQUIRED  
      accounts IDREFS #IMPLIED>  
  ... declarations for branch-name, balance, customer-name,  
  ... customer-city  
>
```

3.1. DTD

```
<bank-2>
  <account account-nb="A-101">
    <branch-name>Downtown</branch-name>
    <balance>500</balance>
  </account>
  <account account-nb="A-102">
    <branch-name>Perryridge</branch-name>
    <balance>400</balance>
  </account>
  <customer customer-id="C-10" accounts="A-101 A-102">
    <customer-name>Johnson</customer-name>
    <customer-city>Palo Alto</customer-city>
  </customer>
</bank-2>
```

3.1. DTD

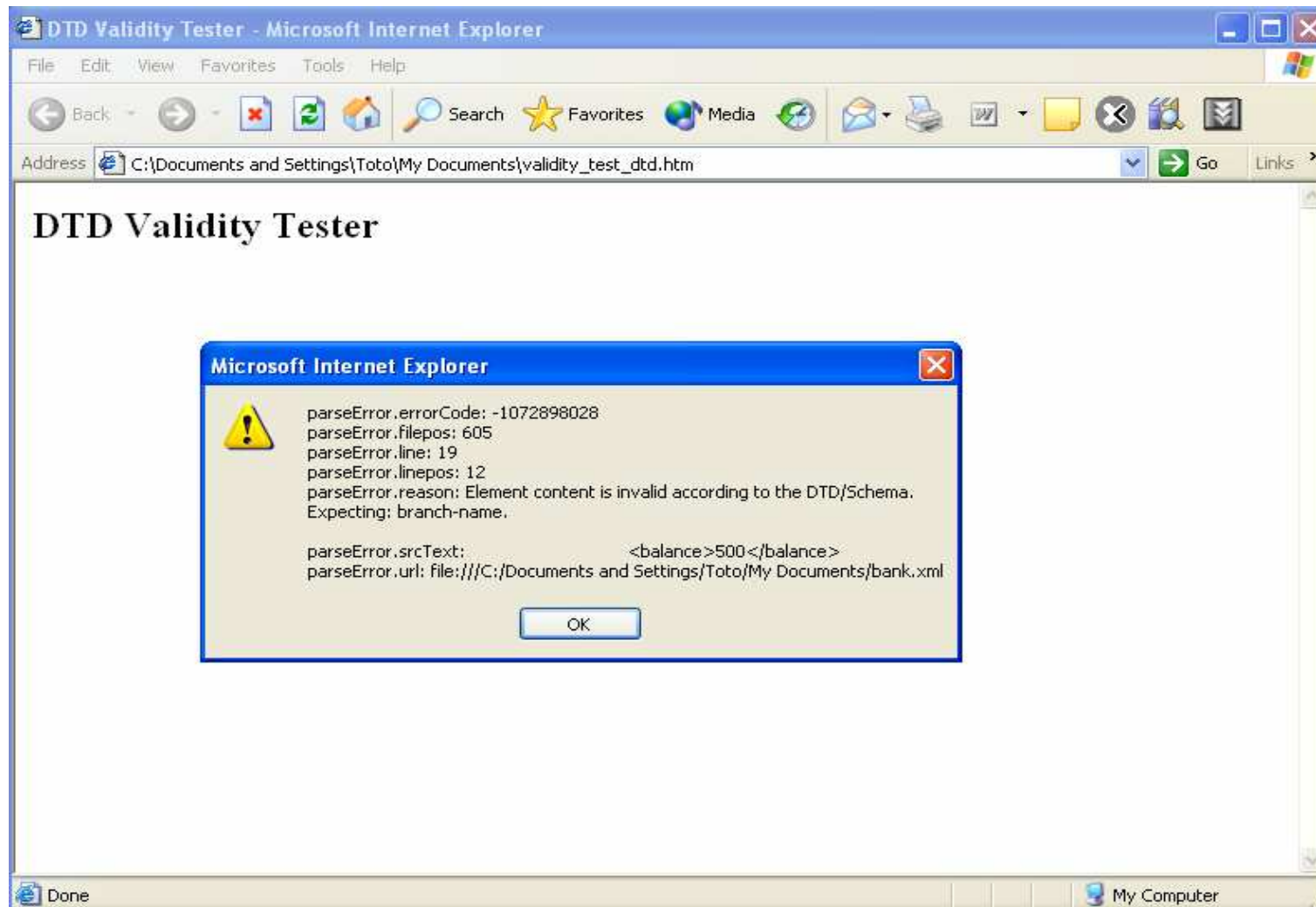
- Limitations of DTDs as schema mechanism:
 - ❑ text elements (PCDATA) and attributes (CDATA) can not be further typed
 - ex: balance cannot be constrained to a positive number
 - ❑ difficult to specify unordered sets of subelements
 - ❑ lack of typing in IDs and IDREFs:
 - no way to specify the type of element to which an IDREF(S) attribute refers
 - ex: does not prevent the accounts attribute of a customer element from referring to customers

3.1. DTD

- Script to test the validity of an XML document:

```
<HTML>
  <HEAD>
    <TITLE>DTD Validity Tester</TITLE>
    <SCRIPT LANGUAGE="JavaScript" FOR="window" EVENT="ONLOAD">
      Document = dsoTest.XMLDocument;
      message = "parseError.errorCode: " + Document.parseError.errorCode + "\n"
        + "parseError.filepos: " + Document.parseError.filepos + "\n"
        + "parseError.line: " + Document.parseError.line + "\n"
        + "parseError.linepos: " + Document.parseError.linepos + "\n"
        + "parseError.reason: " + Document.parseError.reason + "\n"
        + "parseError.srcText: " + Document.parseError.srcText + "\n"
        + "parseError.url: " + Document.parseError.url;
      alert (message);
    </SCRIPT>
  </HEAD>
  <BODY>
    <!-- set SRC to the URL of the XML document you want to check: -->
    <XML ID="dsoTest" SRC="bank.xml"></XML>
    <H2>DTD Validity Tester</H2>
  </BODY>
</HTML>
```

3.1. DTD



3.2. XML Schema

- More sophisticated schema language. Its benefits over DTDs are:
 - ❑ it is itself written in XML
 - ❑ it allows specific types such as numeric types, or lists
 - ❑ it allows types to be restricted, ex. specify min. and max. values
 - ❑ it allows user-defined types to be created
 - ❑ it allows complex types to be extended by using a form of inheritance
 - ❑ it allows uniqueness and foreign key constraints
 - ❑ it is integrated with namespaces to allow different parts of a doc. to conform to different schemas
 - ❑ it is a superset of DTD

... Price paid for these features: XML Schema significantly more complicated than DTD

3.2. XML Schema

- **XML Schema document** = XML file (.xsd):

XML Schema language itself has a schema located at <http://www.w3.org/2001/XMLSchema>

→ Assign this URL to a namespace (ex: 'xs') in the root element *schema*, and use the elements and attributes of this namespace.

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

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
```

... schema definitions added here

```
</xs:schema>
```

3.2. XML Schema

■ XML instance document:

XML Schema declaration used in the root element of the document (unlike the DTD declaration that has a separate `<!DOCTYPE>` tag)

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

```
<bank
```

```
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
  xsi:noNamespaceSchemaLocation="bank.xsd">
```

```
  ...
```

```
</bank>
```

3.2. XML Schema

- Element declaration: 2 types
 - Simple type
 - Complex type
- Simple type:
 - large number of built-in data types: string, integer, positiveInteger, decimal, boolean, date, time, ...
`<xs:element name="balance" type="xs:decimal"/>`

3.2. XML Schema

- built-in data types can be customized by:

- restricting the range

```
<xs:simpleType name="MySmallInteger">  
  <xs:restriction base="xs:integer">  
    <xs:minInclusive value="1"/>  
    <xs:maxInclusive value="10"/>  
  </xs:restriction>  
</xs:simpleType>  
  
<xs:element name="quantity" type="MySmallInteger"/>
```

- restricting the number of digits

3.2. XML Schema

- restricting the string length

```
<xs:simpleType name="postcodeType">  
  <xs:restriction base="xs:string">  
    <xs:minLength value="6"/>  
    <xs:maxLength value="10"/>  
  </xs:restriction>  
</xs:simpleType>  
  
<xs:element name="postcode" type="postcodeType"/>
```

- restricting the patterns

3.2. XML Schema

- restricting the possible values

```
<xs:simpleType name="colorType">  
  <xs:restriction base="xs:string">  
    <xs:enumeration value="white"/>  
    <xs:enumeration value="black"/>  
  </xs:restriction>  
</xs:simpleType>  
  
<xs:element name="color" type="colorType"/>
```

3.2. XML Schema

- not necessary to name a custom data type (if used only once) → **anonymous** (created within the `<xs:element>` and `</xs:element>` tags)

```
<xs:element name="color">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:enumeration value="white"/>
      <xs:enumeration value="black"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

3.2. XML Schema

- **Complex type**: 4 kinds

- Element-only
- Text-only
- Empty
- Mixed-content

- **Complex - sequence**

```
<xs:complexType name="customerType">
  <xs:sequence>
    <xs:element name="customer-name" type="xs:string"/>
    <xs:element name="customer-city" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
```

3.2. XML Schema

- Complex - **choice**

```
<xs:complexType name="telephoneType">
  <xs:choice>
    <xs:element name="officeNumber" type="xs:string"/>
    <xs:element name="homeNumber" type="xs:string"/>
  </xs:choice>
</xs:complexType>
```

- Complex - **all**: each element appears once in any order

```
<xs:complexType name="customerType">
  <xs:all>
    <xs:element name="customer-name" type="xs:string"/>
    <xs:element name="customer-city" type="xs:string"/>
  </xs:all>
</xs:complexType>
```

3.2. XML Schema

■ Referencing elements

Both simple type and complex type elements that are declared just within the `<xs:schema>` tags are available globally

→ can be referenced by assigning their name to 'ref' attribute

```
<xs:complexType name="bankType">
  <xs:choice>
    <xs:element ref="customer"/>
    <xs:element ref="account"/>
  </xs:choice>
</xs:complexType>
```

3.2. XML Schema

■ Element occurrences

- ❑ attributes '**minOccurs**' and '**maxOccurs**' are used to specify how many times an element, sequence, set of choices, or group may appear
- ❑ values:
 - any positive integer
 - maxOccurs can take the keyword '**unbounded**' (unlimited occurrences)
 - minOccurs can take the value '0' (optional)
 - default value of both attributes is '1'

3.2. XML Schema

```
<xs:complexType name="bankType">  
  <xs:choice maxOccurs="unbounded">  
    <xs:element ref="customer"/>  
    <xs:element ref="account"/>  
  </xs:choice>  
</xs:complexType>
```

3.2. XML Schema

■ Text-only elements

- ❑ Text-only element without attributes: defined as simple type
- ❑ Text-only element with attributes: defined as complex type
 - definition contained in a 'simpleContent' element (within the 'complexType' element)
 - complex type based on any XMLSchema type (string, integer,...) → 'base' attribute of an 'extension' element
 - each attribute defined with an 'attribute' element (within the 'extension' element)

3.2. XML Schema

```
<xs:complexType name="priceType">
  <xs:simpleContent>
    <xs:extension base="xs:decimal">
      <xs:attribute name="currency"
        type="xs:string"/>
      <xs:attribute name="offer"
        type="xs:boolean"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
```

3.2. XML Schema

■ Empty elements with attributes

Any element declaration that does not assign any data type is considered an empty element

→ empty elements will always have attributes (within the 'complexType' element)

```
<xs:element name="info">
  <xs:complexType>
    <xs:attribute name="price" type="xs:decimal"/>
    <xs:attribute name="currency" type="xs:string"/>
  </xs:complexType>
</xs:element>
```

3.2. XML Schema

- **Mixed-content elements**

boolean attribute 'mixed' in the 'complexType' element

- **Anonymous complex types**

if a complex type will not be reused, define an un-named complex type within the element declaration (see previous slide)

3.2. XML Schema

■ Requiring attributes

- **'use'** attribute in the 'attribute' element
- values: 'required', 'optional' (default value), 'prohibited'

■ Predefining attribute values

- **'fixed'** or **'default'** attributes in the 'attribute' element

```
<xs:attribute name="customer-id" type="xs:string"  
              use="required"/>
```

```
<xs:attribute name="currency" type="xs:string"  
              default="euro"/>
```

3.2. XML Schema

■ Constraints: unique, key, keyref

□ Uniqueness

- selector: element which has the uniqueness constraint
- field: element or attribute whose value will be checked for uniqueness
- only one xs:selector element, one or more xs:field elements
- location of the xs:unique element gives the context node in which the constraint holds (scope)

```
<xs:element name="bank">
  ...
  <!-- after the complexType element-->
  <xs:unique name="cons1">
    <xs:selector xpath="./account"/>
    <xs:field    xpath="account-nb"/>
  </xs:unique>
</xs:element>
```

3.2. XML Schema

■ Key

- similar to xs:unique except that the value has to be non null

```
<xs:key name="cons2">  
  <xs:selector xpath="./account"/>  
  <xs:field    xpath="account-nb"/>  
</xs:key>
```

■ Reference

- xs:keyref allows to define a reference to a xs:key or a xs:unique

```
<xs:element name="bank">  
  ...  
  <xs:keyref name="cons3" refer="cons2">  
    <xs:selector xpath="./depositor"/>  
    <xs:field    xpath="account-nb"/>  
  </xs:keyref>  
</xs:element>
```

3.2. XML Schema

- Script to test the validity of an XML document:

```
<HTML>
  <HEAD>
    <TITLE>XML Schema Validity Tester</TITLE>
    <SCRIPT LANGUAGE="JavaScript" FOR="window" EVENT="ONLOAD">
      /* set XMLFileURL to the URL of the XML document: */
      var XMLFileURL = "BankBis.xml";

      /* if the XML document's root element belongs to a namespace, set XMLNamespaceName to the
         namespace name; otherwise, set it to an empty string: */
      var XMLNamespaceName = "";

      /* set SchemaFileURL to the URL of the XML schema file: */
      var SchemaFileURL = "BankBis.xsd";

      /* create a new, empty Document node: */
      Document = new ActiveXObject ("Msxml2.DOMDocument.4.0");

      /* store the URL of the XML schema file in a new XMLSchemaCache object, and then assign that
         object to the Document node's 'schemas' property: */
      XMLSchemaCache = new ActiveXObject ("Msxml2.XMLSchemaCache.4.0");
      XMLSchemaCache.add (XMLNamespaceName, SchemaFileURL);
      Document.schemas = XMLSchemaCache;
```

3.2. XML Schema

...cont'd

```
    /* cause subsequent call to 'load' method to load the XML document synchronously: */
    Document.async = false;

    /* load the XML document into the Document node: */
    Document.load (XMLFileURL);

    /* display error information: */
    message = "parseError.errorCode: " + Document.parseError.errorCode + "\n"
        + "parseError.filepos: " + Document.parseError.filepos + "\n"
        + "parseError.line: " + Document.parseError.line + "\n"
        + "parseError.linepos: " + Document.parseError.linepos + "\n"
        + "parseError.reason: " + Document.parseError.reason + "\n"
        + "parseError.srcText: " + Document.parseError.srcText + "\n"
        + "parseError.url: " + Document.parseError.url;
    alert (message);
</SCRIPT>
</HEAD>
<BODY>
    <H2>XML Schema Validity Tester</H2>
</BODY>
</HTML>
```

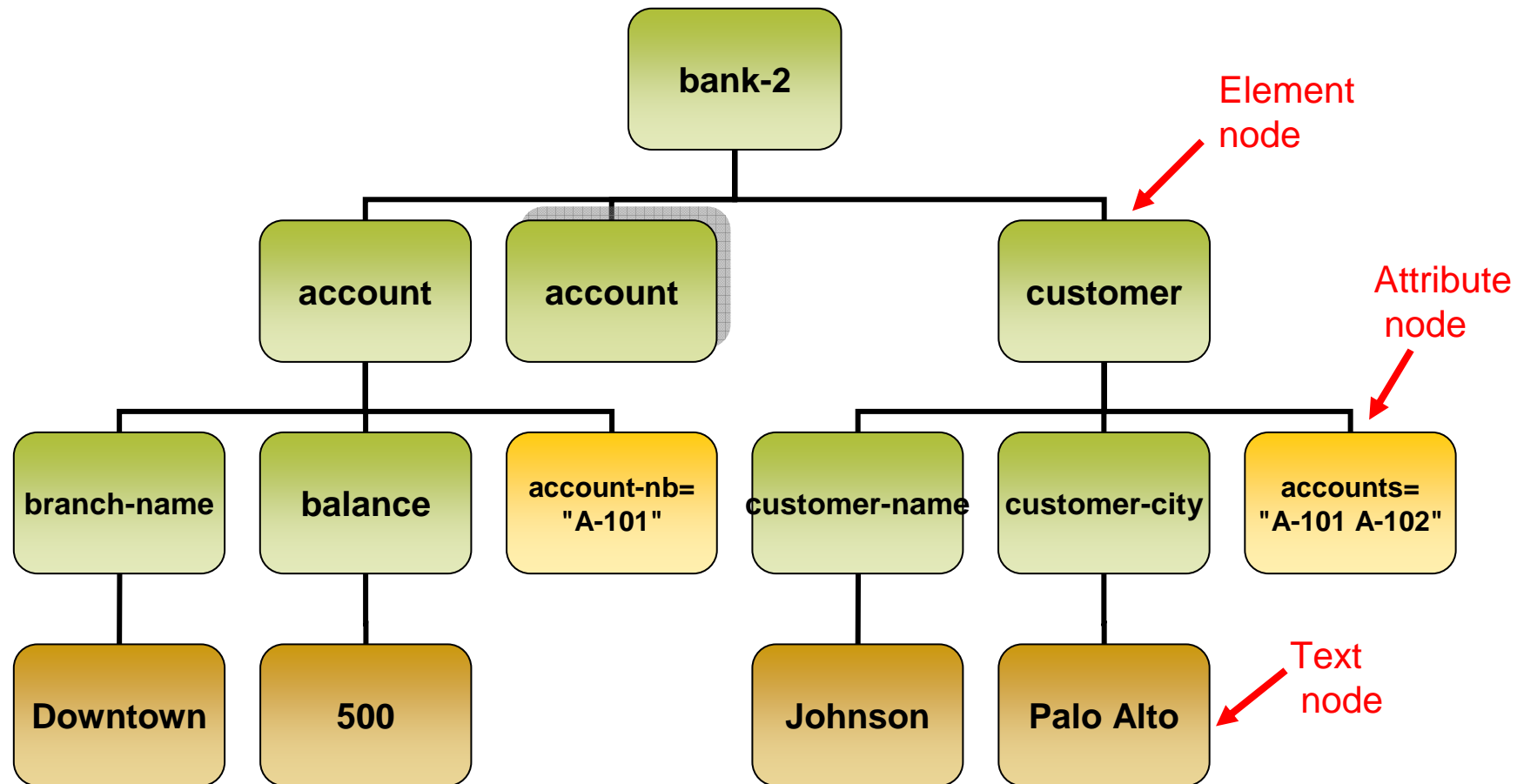
3.2. XML Schema



4. XML Query Languages

- Output of an XML query is XML data
- Several query languages, ex:
 - XPath: language for path expressions, building block for XQuery
 - XQuery: standard for querying XML data
- **Tree model** of XML data is used in these languages
XML document = tree:
 - nodes correspond to elements and attributes
 - an element node can have children nodes for the subelements and attributes
 - each node (attribute or element node), other than the root element node, has a parent node (element node)
 - ordering of children
 - text content of an element in a text node (child of the element node)

4. XML Query Languages



4.1. XPath

■ Path expression:

- ❑ sequence of location steps separated by '/'
- ❑ result of a path expression: set of nodes

(see files bank.xml p.4, bank1.xml p.10, bank2.xml p.25)

/bank-2/account/branch-name

returns:

<branch-name>Downtown</branch-name>

<branch-name>Perryridge</branch-name>

/bank-2/account/branch-name/text()

returns the same names without the enclosing tags:

Downtown

Perryridge

4.1. XPath

- Like a directory hierarchy
 - ❑ initial '/': root of the document (abstract root above <bank-2>)
 - ❑ path expression evaluated from left to right.
As it is evaluated, the result at any point is a set of nodes from the document
 - ❑ x/* : returns all children of x

- Attributes may be accessed using '@'
/bank-2/account/@account-nb

4.1. XPath

■ Selection predicates

- ❑ may follow any step in a path, in square brackets

`/bank-2/account[balance > 400]`

`/bank-2/account[balance > 400]/@account-nb`

returns the account number of accounts with
balance > 400

`/bank-2/account[@account-nb = 'A-101']/balance`

returns the balance of the account having number A-101

`/bank-2/account[balance]`

tests the existence of a subelement (list it without any
comparison operator)

4.1. XPath

- ❑ logical operators **and**, **or**, and **not(..)** can be used in predicates
 - `/bank-2/account[balance > 400 and branch-city = 'Downtown']`
- ❑ **functions** can be used in predicates, such as:
 - testing the position of the current node in the sibling order
 - `/bank-2/account[position() = 2]`
 - `/bank-2/account[2]`
 - both return the second account
 - counting the number of nodes matched
 - `/bank-1/customer[count(account) > 2]`
 - returns the customers with more than 2 accounts

4.1. XPath

- From the current node, several directions along which a step in a path may proceed
 - ex:
 - `/` : child
 - `..` : parent
 - `//` : descendants
 - ancestors, siblings
 - an Xpath expression can skip multiple levels by using `//`
`/bank-2//name`
 - finds any 'name' element anywhere under the 'bank-2' element, regardless of the element in which it is contained
- Ability to find the required data without a full knowledge of the schema

4.1. XPath

- Other examples:
 - `/bank/*/account-nb`
finds the account-nb of any child of the 'bank' element
 - `/bank//*`
finds any element anywhere under the 'bank' element
- Function `id('A-101')`
returns the element (if any) with an attribute of type ID and value "A-101"
- Operator `|`
allows expression results to be unioned (cannot be nested inside other operators)
`/bank-2/account[2] | /bank-2/customer`

4.1. XPath

- Script to run a query:

```
<html>
  <body>
    <script type = "text/vbscript">
      set xmlDoc = CreateObject("MSXML2.DOMDocument.4.0")
      xmlDoc.async = "false"
      xmlDoc.load("bank.xml")
      path = "/bank/depositor[customer-name='Johnson']/account-nb" // your query
      set nodes = xmlDoc.selectNodes(path)

      for each x in nodes
        document.write("<xmp>")
        document.write(x.xml)
        document.write("</xmp>")
      next
    </script>
  </body>
</html>
```

4.2. XQuery

- Developed by W3C
 - derived from Quilt, an XML query language, which included features from XPath, XQL and XML-QL
- XQuery: FLWOR (flower) expressions
 - **for**: variables that range over the results of XPath expressions. If more than 1 variable, Cartesian product of values taken by the variables (like from in SQL)
 - **let**: allows complicated expressions to be assigned to variable names (for simplicity)
 - **where**: tests on data given by the 'for' clause (like where in SQL)
 - **order by**: sorts the result (like order by in SQL)
 - **return**: construction of result in XML

4.2. XQuery

- Example: return the branch names of accounts with balance > 300

(see files bank.xml p.4, bank1.xml p.10, bank2.xml p.25)

```
for $x in doc("bank.xml")/bank/account  
let $bname := $x/branch-name  
where $x/balance > 300  
return <name>{$bname/text()} </name>
```

equivalent to: (selections in XPath expressions, without 'let')

```
for $x in doc("bank.xml")/bank/account[balance > 300]  
return <name>{$x/branch-name/text()} </name>
```

returns:

```
<name>Downtown</name>  
<name>Perryridge</name>
```

4.2. XQuery

- Results can be **sorted** (ascending order by default)

```
for $a in doc("bank.xml")/bank/account  
order by $a/balance descending  
return $a
```

note: {} are needed around variables in the 'return' clause only if there are tags or more than one variable

- Path expressions may return a multiset (repeated nodes), the function **distinct-values** removes duplicates

```
for $c in distinct-values(doc("bank.xml")/bank/  
                           depositor/customer-name)  
return $c
```

4.2. XQuery

- **Joins** specified like in SQL:

```
for   $d in doc("bank.xml")/bank/depositor,  
       $a in doc("bank.xml")/bank/account,  
       $c in doc("bank.xml")/bank/customer  
where $a/account-nb = $d/account-nb and  
       $c/customer-name = $d/customer-name  
return <cust-acct> {$c} {$a} </cust-acct>
```

returns ...

4.2. XQuery

```
<cust-acct>
  <customer>
    <customer-name>Johnson</customer-name>
    <customer-city>Palo Alto</customer-city>
  </customer>
  <account>
    <account-nb>A-101</account-nb>
    <branch-name>Downtown</branch-name>
    <balance>500</balance>
  </account>
</cust-acct>
<cust-acct>
  <customer>
    <customer-name>Johnson</customer-name>
    <customer-city>Palo Alto</customer-city>
  </customer>
  <account>
    <account-nb>A-102</account-nb>
    <branch-name>Perryridge</branch-name>
    <balance>400</balance>
  </account>
</cust-acct>
```

4.2. XQuery

- Example: for each customer, list his/her name and accounts (in bank1.xml)

```
for $c in doc("bank1.xml")/bank-1/customer  
return
```

```
  <cust>  
    {$c/customer-name}  
    {$c/account/account-nb}  
  </cust>
```

returns:

```
<cust>  
  <customer-name>Johnson</customer-name>  
  <account-nb>A-101</account-nb>  
  <account-nb>A-102</account-nb>  
</cust>
```

4.2. XQuery

- FLWOR expressions can be **nested** in the return clause
Generates element nestings that do not appear in the source document,
ex: bank doc. → bank1 doc. where account elements nested within
customer elements

```
<bank-1>
{
  for $c in doc("bank.xml")/bank/customer
  return
    <customer>
      {$c/*}
      {
        for $d in doc("bank.xml")/bank/depositor
          [customer-name = $c/customer-name],
          $a in doc("bank.xml")/bank/account[account-nb = $d/account-nb]
          return $a
      }
    </customer>
}
</bank-1>
```

4.2. XQuery

- XQuery provides **built-in functions**
 - Conversion functions: XQuery uses the type system of XML Schema, and provides functions to convert between types.
ex: **number(x)** converts a string to a number
 - String functions

```
for $n in doc("bank.xml")/bank/customer/customer-name  
where contains($n/text(), "John")  
return $n
```
- XQuery supports **user-defined functions**

4.2. XQuery

- XQuery provides **aggregate functions** (e.g. sum, count)

- example:

```
for $c in doc("bank1.xml")/bank-1/customer
let $b := $c/account/balance
return
  <cust>
    {$c/customer-name}
    <minbalance> {min($b/text())} </minbalance>
  </cust>
```

- no group by but aggregated queries can be written by using nested FLWOR expressions

4.2. XQuery

- Other features:
 - **if-then-else** clause can be used within the 'return' clause.
ex: for each customer, list his/her name and first account, and an empty 'more' element if he/she has additional accounts

```
<result>
{
  for $c in doc("bank1.xml")/bank-1/customer
  return
    <customer>
      {$c/customer-name}
      {
        for $a in $c/account[position() < 2]
        return $a
      }
      {
        if (count($c/account) > 1)
        then <more/>
        else ()
      }
    </customer>
}
</result>
```

4.2. XQuery

- existential and universal quantification can be used in predicates in the 'where' clause: **some**, **every**
where some \$e in path satisfies P
- 'for' and 'let' clauses both bind variables, but are different:
 - let \$s := (<one/>, <two/>, <three/>)
return <out> {\$s} </out>
returns: <out> <one/> <two/> <three/> </out>
... the variable \$s is bound to the result of the expression
 - for \$s in (<one/>, <two/>, <three/>)
return <out> {\$s} </out>
returns:
 <out> <one/> </out>
 <out> <two/> </out>
 <out> <three/> </out>
... the variable \$s iterates over the given expression

4.2. XQuery

- Run a query:
 - you can use Qexo, the GNU kawa implementation of XQuery
 - download the file: kawa-1.7.90.jar
 - write your query in a file *myfile.xql*
 - type the command:

```
java -jar kawa-1.7.90.jar --xquery -f myfile.xql
```
 - some missing features (order by, distinct-values)

5. Application Program Interface

- Application Program Interface
 - to access and manipulate a document
 - a programmer can navigate a document's structure, and add, modify or delete its elements
 - several APIs:
 - **DOM**: Document Object Model, tree-based model
 - **SAX**: Simple API for XML, event-based model
 - DOM and SAX are standard APIs that can be used with any programming language, such as JavaScript, Perl, Java, C++

5. Application Program Interface

- DOM
 - treats an XML document as a tree
 - each tree node is an object, with the top level 'Document' object
- Java DOM API (package **org.w3c.dom**):
 - Interfaces including:
 - **Node**
 - **Element**, **Attr** which inherit from Node
 - Node interface:
 - getFirstChild(), getNextSibling(), getParentNode(), getAttributes(),...
methods to navigate the DOM tree (starting with the root node)
 - getNodeName(), getNodeTypes(), getNodeValue(),...
methods to return the name, type and value of a node
 - setNodeValue(String nodeValue), appendChild(Node newChild),
removeChild(Node oldChild),...
methods to update the document

References

- ❑ Database System Concepts, by A. Silberschatz, H. Korth, S. Sudarshan, 4th edition, McGraw-Hill, 2001
- ❑ XML in easy steps, by M. McGrath, Computer Step, 2002
- ❑ Beginning XML, by D. Hunter, K. Cagle, C. Dix et al., 2nd edition, Wrox Press, 2003

- ❑ Tutorials: www.w3schools.com

- ❑ www.w3.org/XML
W3C recommendation, 1.0 (3rd edition), feb. 2004
- ❑ www.w3.org/XML/Schema
W3C recommendation, 1.0 (2nd edition), oct. 2004
- ❑ www.w3.org/TR/xpath
W3C recommendation, 1.0, nov. 1999. Working draft, XPath 2.0, oct. 2004
- ❑ www.w3.org/TR/xquery
working draft, XQuery 1.0, oct. 2004
www.w3.org/TR/xquery-use-cases
- ❑ java.sun.com/j2se/1.5.0/docs/guide/plugin/dom/
- ❑ www.gnu.org/software/qexo