XML Basics for Web Services

Worflows und Web Services
Kapitel 2

XML Origin and Usages

- Defined by the WWW Consortium (W3C)
- Originally intended as a document markup language not a database language
  - Documents have tags giving extra information about sections of the document
  - E.g. <title> XML </title>
    <slide> XML Origin and Usages </slide>
- Derived from SGML (Standard Generalized Markup Language), but simpler to use than SGML

XML Origin and Usages (cont.)

- XML documents are to some extent self-documenting
  - Tags can be used as metadata
  - 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>
```

XML Origin and Usages (cont.)

- XML is heavily used for data exchange
- Data interchange is critical in today's networked world
  - Examples:
    - Banking: funds transfer, Order processing
    - Scientific data: Chemistry (ChemML), Genetics (BSML, Bio-Sequence markup Language)
- XML has become the basis for new application-specific data interchange formats
- XML is a key technology for interoperation
- Each XML based standard defines what are valid elements, using
  - XML type specification languages to specify the syntax
    - DTD (Document Type Descriptors)
    - XML Schema
  - Plus textual descriptions of the semantics
- A wide variety of tools is available for parsing, browsing and querying XML documents/data

Describing XML Data: Basics - Elements

- **Tag:** label for a section of data
- **Element**: section of data beginning with <*tagname*> and ending with matching </*tagname*>
- Elements must be properly **nested**
  - Formally: every start tag must have a unique matching end tag, that is in the context of the same parent element.
- Every document must have a single top-level element
- Mixture of text with sub-elements is legal in XML
  - Example:

```
<account>
    This account is seldom used any more.
     <account-number> A-102</account-number>
     <branch-name> Perryridge</branch-name>
     <balance>400 </balance>
</account>
```

  - Useful for document markup, but discouraged for data representation

XML Document Example

```
<bank-1>
   <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>
   </customer>
     .
     .
</bank-1>
```

Describing XML Data: Attributes

- **Attributes:** can be used to describe elements
- 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
- Attribute names must be unique within the element

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

Attributes vs. Subelements

- Distinction between subelement and attribute
  - In the context of documents, attributes are part of markup, while subelement contents are part of the basic document contents
  - In the context of data representation, the difference is unclear and may be confusing
    - Same information can be represented in two ways
      - `<account  account-number = "A-101">  .... </account>`

      -
        ```
        <account>
           <account-number>A-101</account-number> …
         </account>
        ```
  - Suggestion: use attributes for identifiers of elements, and use subelements for contents

More on XML Syntax

- Shortcut: elements without subelements or text content can be abbreviated by ending the start tag with a /> and deleting the end tag
  - <account number="A-101" branch="Perryridge" balance="200" />
- To store string data that may contain tags, without the tags being interpreted as subelements, use CDATA as below
  - <![CDATA[<account> … </account>]]>
    - Here, <account> and </account> are treated as just strings

Namespaces

- A single XML document may contain elements and attributes defined for and used by multiple software modules
  - Motivated by modularization considerations, for example
- Name collisions have to be avoided
- Example:
  - The **Book** XSD contains a Title element for the title of a book
  - The **Person** XSD contains a Title element for an honorary title of a person
  - The **BookOrder** XSD reference both XSDs
- Namespaces specifies how to construct universally unique names

Namespaces (cont.)

- Namespace is a collection of names identified by a URI
- Namespaces are declared via a set of special attributes
  - These attributes are prefixed by xmlns - Example:
    <BookOrder xmlns:Customer="http://mySite.com/Person"
    xmlns:Item=http://yourSite.com/Book>
- Elements/attributes from a particular namespace are prefixed by the name assigned to the namespace in the corresponding declaration of the using XML document
  - ...Customer:Title='Dr'...
  - ...Item:Title='Introduction to XML'...

XML Document Schema

- Metadata and database schemas constrain what information can be stored, and the data types of stored values
- Metadata are very important for data exchange
  - Guarantees automatic and correct data interpretation
- XML documents are not required to have associated metadata/schema
- Two mechanisms for specifying XML schema
  - **Document Type Definition (DTD)**
    - Widely used
  - **XML Schema**
    - Newer, not yet widely used

Describing XML Data: DTD

- Type and structure of an XML document can be specified using a DTD
  - 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
    - "?" - 0 or 1 occurrence
    - "+" - 1 or more occurrences
    - "*" - 0 or more occurrences

Example: Bank DTD

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

Attribute Specification in DTD

- Attribute specification : for each attribute
  - Name
  - Type of attribute
    - CDATA
    - ID (identifier) or IDREF (ID reference) or IDREFS (multiple IDREFs)
  - Whether
    - mandatory (#REQUIRED)
    - 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

  ➔ 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
- IDs and IDREFs are untyped, unfortunately
  - Example below: 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

Example: Extended Bank DTD

- Bank DTD with ID and IDREF attribute types

```
<!DOCTYPE bank-2[
   <!ELEMENT account (branch-name, 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>
     … declarations for branch, balance, customer-name,
                     customer-street and customer-city
]>
```

XML data with ID and IDREF attributes

```
<bank-2>
    <account account-number="A-401" owners="C100 C102">
          <branch-name> Downtown </branch-name>
          <balance>500 </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>
</bank-2>
```

Describing XML Data: XML Schema

- XML Schema is closer to the general understanding of a (database) schema
- XML Schema supports
  - Typing of values
    - E.g. integer, string, etc
  - Constraints on min/max values
  - Typed references (for ID and IDREFS)
  - User defined types
  - Specified in XML syntax (unlike DTDs)
  - Integrated with namespaces
  - Many more features
    - List types, uniqueness and foreign key constraints, inheritance ..
- BUT: significantly more complicated than DTDs

XML Schema Structures

- **Datatypes (Part 2)**
  Describes Types of scalar (leaf) values
- **Structures (Part 1)**
  Describes types of complex values (attributes, elements)
  - Regular tree grammars
    repetition, optionality, choice recursion
- **Integrity constraints**
  Functional (keys) & inclusion dependencies (foreign keys)
- **Subtyping (akin to OO models)**
  Describes inheritance relationships between types
- **Supports schema reuse**

XML Schema Structures (cont.)

- Elements : tag name & simple or complex type
```
<xs: element name="sponsor" type="xsd: string"/>
<xs: element name="action" type="Action"/>
```
- Attributes : tag name & simple type
```
<xs: attribute name="date" type="xsd: date"/>
```
- Complex types
```
<xs: complexType name="Action">
    <xs: sequence>
        <xs: elemref name ="action-date"/>
        <xs: elemref name ="action-desc"/>
    </xs: sequence>
</xs: complexType>
```

XML Schema Structures (cont.)

- Sequence

```
<xs: sequence>
     <xs: element name="congress" type=xsd: string"/>
     <xs: element name="session" type=xsd: string"/>
</xs: sequence>
```

- Choice

```
<xs: choice>
     <xs: element name="author" type="PersonName"/>
     <xs: element name="editor" type="PersonName"/>
</xs: choice>
```

- Repetition

```
<xs: sequence minOccurs="1" maxOccurs="unbounded">
       <xs : element name ="section" type="Section"/>
</xs : sequence>
```

XML Schema Content Models

- Putting it all together

```
<element name="bill"> <complexType><sequence>
    <element name="action"><complexType><sequence>
        <element name="action-date">
                <attribute name="date" type="xsd: date"/>
         </element>
        <element name="action-desc" content="mixed">
            <element name="sponsor" type=xsd: string"/>
            <element name="co-sponsor" type=xsd: string" min="0"
              max="unbounded"/>
        </element>
    </sequence></complexType></element>
    <elementref name="legis-body"/>
    <element name="header" type="xsd: string"/>
    <sequence min="0"  max="unbounded">
      <elementref name ="section"/></sequence>. . .
```

XML Schema Version of Bank DTD

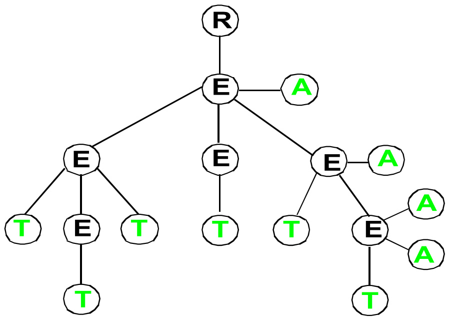
```
<xs:schema xmlns:xs=http://www.w3.org/2001/XMLSchema>
<xs:element name="bank" type="BankType"/>
<xs:element name="account">
    <xsd:complexType>
        <xsd:sequence>
            <xs:element name="account-number" type="xsd:string"/>
            <xs:element name="branch-name"     type="xsd:string"/>
            <xs:element name="balance"           type="xsd:decimal"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>                    ..... definitions of customer and depositor ....
<xs:complexType name="BankType">
    <xs:sequence>
        <xs:element ref="account"   minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="customer" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element ref="depositor" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
</xs:complexType>
</xs:schema>
```

Processing XML Data

- Querying XML data
- Translation of information from one XML schema to another
- 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
    - XQuery builds on experience with existing query languages: **XPath**, **Quilt**, XQL, XML-QL, Lorel, YATL, **SQL**, **OQL**, …

Tree Model of XML Data

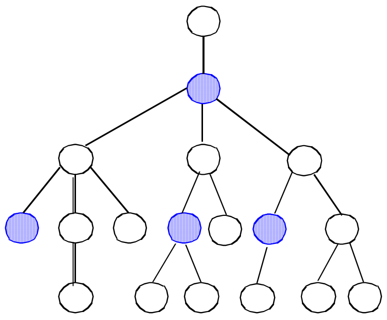
- Query and transformation languages are based on a **tree model** of XML data
- An XML document is modeled as a tree, with **nodes** corresponding to elements and attributes



- Several types of nodes:
  - Root, Element, Attribute,Text, Namespace, Comment, Processing Instruction

Processing XML Data: XPath

- XPath is used to address (select) parts of documents using path expressions
- XPath data model refers to a document as a tree of nodes
- An Xpath expression maps a node (the context node) into a set of nodes
- A path expression consists of one or more steps separated by "/"
- 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 returns
    - <customer-name> Joe </ customer-name>
    - < customer- name> Mary </ customer-name>
  - E.g.:/bank-2/customer/cust-name/text( ) returns the same names, but without the enclosing tags



XPath (cont.)

- The initial "/" denotes root of the document (above the top-level tag)
- In general, a step has three parts:
  - The ***axis*** (direction of movement: child, descendant, parent, ancestor, following, preceding, attribute, … - 13 axes in all -)
  - A ***node test*** (type and/or name of qualifying nodes)
  - Some ***predicates*** (refine the set of qualifying nodes)
- 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)

XPath (cont.)

- The following examples use XPath abbreviated notation:
  - Find the first item of every list that is under the context node
    .//list/item[1]
  - Find the "lang" attribute of the parent of the context node
    ../@lang
  - Find the last paragraph-child of the context node
    para[last()]
  - Find all warning elements that are inside instruction elements
    //instruction//warning
  - Find all elements that have an ID attribute
    //*[@ID]
  - Find names of customers who have an order with today's date
    //customer [order/date = today ( )] / name

- **XPath expressions use a notation similar to paths in a file system:**

/	means "child" or "root"
//	means "descendant"
.	means "self"
..	means "parent"
*	means "any"
@	means "attribute"

XPath (cont.)

- Node tests are most frequently element names, but other node tests are possible:
  - **name** – Matches <name> element nodes
  - ***** - Matches any element node
  - **namespace:name** – Matches <name> element nodes in the specified namespace
  - **namespace:*** - Matches any element node in the specified namespace
  - **comment()** – Matches comment nodes
  - **text()** – Matches text nodes
  - **processing-instruction()** – Matches processing instructions
  - **processing-instruction('target')** – Matches processing instructions with the specified target (<?target ...?>)
  - **node()** – Matches any node

XPath (cont.)

- 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(@owners)
    returns all customers referred to from the owners attribute of account elements

XPath (cont.)

- Operator "|" used to implement union
  - E.g. /bank-2/account/id(@owners) **|** /bank-2/loan/id(@borrower)
    returns customers with either accounts or loans
  - However, "|" cannot be nested inside other operators
  - In general, an XPath expression is a union of one or more paths
- "//" can be used to skip multiple levels of nodes
  - E.g. /bank-2//name
    returns any name element *anywhere* under the /bank-2 element, regardless of the element in which it is contained
- Using XPath in a URI
  - A **Universal Resource Identifier (URI)** identifies a document:
    - http://www.w3.org/XML/Query
- An Xpath expression can be appended to a URI to identify a specific part of the target document:
  - **E.g.** http://www.w3.org/XML/Query//figure[id="boat"]
  - The result is called an **XPointer**

XPath (cont.): Summary

- Strengths:
  - Compact and powerful syntax for navigating a tree,
    but not as powerfull as a regular-expression language
  - Recognized and accepted in XML community
  - Used in XML-related applications such as XPointer
- Limitations:
  - Operates on one document (no joins)
  - No grouping or aggregation
  - No facility for generating new output structures

Transforming XML Data: XSLT

- A **stylesheet** stores formatting options for a document, usually separately from document
  - E.g. HTML style sheet may specify font colors and sizes for headings, etc.
- The **XML Stylesheet Language (XSL)** was originally designed for generating HTML from XML
- XSLT is a general-purpose transformation language
  - Can translate XML to XML, and XML to HTML
- XSLT transformations are expressed using rules called **templates**
  - Templates combine selection using XPath with construction of results

Understanding A Template

- Most templates have the following form:

```
<xsl:template match="emphasis">
  <i><xsl:apply-templates/></i>
</xsl:template>
```

- The whole <xsl:template> element is a **template**
- The **match pattern** determines where this template applies
  - Xpath pattern
- **Literal result element(s)** come from non-XSL namespace(s)
- XSLT elements come from the XSL namespace

XQuery

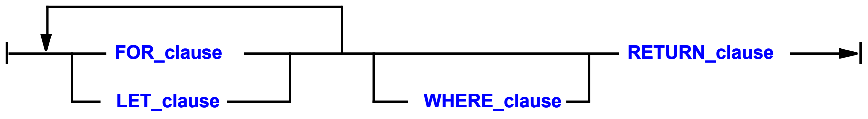
- XQuery is a general purpose query language for XML data
- Currently being standardized by the World Wide Web Consortium (W3C)
- Alpha version of XQuery engine available free from Microsoft
- XQuery is derived from
  - the **Quilt** ("Quilt" refers both to the origin of the language and to its use in "knitting " together heterogeneous data sources) query language, which itself borrows from
  - **XPath**: a concise language for navigating in trees
  - **XML-QL**: a powerful language for generating new structures
  - **SQL**: a database language based on a series of keyword-clauses: SELECT - FROM – WHERE
  - **OQL**: a functional language in which many kinds of expressions can be nested with full generality

XQuery: Language Requirements

- XML elements have attributes as well as content
- Content model of an XML element-type is much more flexible than in the relational setting (as described by an XML Schema)
  - Choices between alternative contents, variable numbers of repetitions
  - Mixed content (subelements mixed with text)

➔ An XML query language must be able to:
  - Query deeply nested and heterogeneous structures
  - Query metadata as well as user data
  - Search for objects by absolute and relative order
  - Preserve order of objects in input documents
  - Impose new ordering at multiple levels of output
  - Handle missing data and sparse data
  - Preserve or transform the structure of a document
  - Exploit references to unknown or heterogeneous types
  - Easily define recursive functions
  - Provide a very flexible data definition facility

XQuery: The General Syntax Expression FLWR



- FOR-clause
  iterates over a set of nodes (possibly specified by an XPath expression), binding a variable to the individual nodes in the set
- LET-clause
  binds a variable to the result of an expression
- WHERE-clause
  applies a predicate to filter the variables bound by FOR and LET
- RETURN-clause
  constructs the output
- Associations to SQl query expressions
  for ⇔ SQL **from**
  where ⇔ SQL **where**
  return ⇔ SQL **select**
  let allows temporary variables, and has no equivalent in SQL

XQuery: FLWR Syntax

- XQuery is a functional language
  - Every query is an expression
  - Expressions can be nested with full generality:
    - XPath expressions
    - Element constructors
    - FLWR expressions
- 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 and Where 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 can be used in various places, e.g.:
  - In the For clause to bind variables
  - In the Let clause to bind variables to results of path expressions
- $c/text() gives text content of an element without 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
    - E.g.: "List hobbies of Denver employees"

```
/emp[location = "Denver"]/@hobbies => *
```

Results may include <skiing>, <hiking>, etc.

- 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: 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>
```

XQuery: Changing Nesting Structure

- XML queries may need to transform their input document(s) into new hierarchic structures
- Example: inversion of a hierarchy

```
<book>
   <title>
   <author>
   <author>
</book>
<book>
   <title>
   <author>
   <author>
</book>
```

```
FOR $a IN distinct(//author)
RETURN
   <author>
      <name> { $a/text() } </name>
      { FOR $b IN //book[author = $a]
           RETURN $b/title
           SORTBY(.) }
   </author>
SORTBY(name)
```

```
<author>
   <name>
   <title>
   <title>
</author>
<author>
   <name>
   <title>
   <title>
</author>
```

Processing XML Data: Summary on XQuery

- XQuery on one slide: Forms of expressions:
  - Variables and constants: $x, 5
  - Operators and function calls: x + y, -z, foo(x, y)
  - Path expressions: /a//b[c = 5]
  - FLWR expressions: FOR ... LET ... WHERE ... RETURN
  - Element constructors: <a> ... </a>
  - Conditional expressions: IF ... THEN ... ELSE
  - Quantifiers: EVERY var IN expr SATISFIES expr
  - Sorted expressions: expr SORTBY (expr ASCENDING , ...)
  - Type test: expr INSTANCE OF type
  - Cast expression: CAST AS type (expr)
  - Typeswitch: branches on the dynamic type of an expr

Application Programming with XML

- Application needs to work with XML data/document
  - **Parsing** XML to extract relevant information
  - Produce XML
    - Write character data
    - Build internal XML document representation and **Serialize** it
  - Simple API for XML (SAX)
    - "Push" parsing (event-based parsing)
    - Parser sends notifications to application about the type of document pieces it encounters
    - Notifications are sent in "reading order" as they appear in the document
    - Preferred for large documents (high memory efficiency)
  - Document Object Model (DOM)
    - "One-step" parsing
    - Generates in-memory representation of the document (parse tree)
    - DOM specifies the types of parse tree objects, their properties and operations
      - Independent of programming language (uses IDL)
    - Bindings available to specific programming languages (e.g., Java)

DOM

- DOM structure is a hierarchy of nodes

Node type	**Contains**
Document	Element (maximum of one), ProcessingInstruction, Comment, DocumentType
DocumentFragment	Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
DocumentType	no children
EntityReference	Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
Element	Element, Text, Comment, ProcessingInstruction, CDATASection, EntityReference
Attr	Text, EntityReference
ProcessingInstruction	no children
Comment	no children
Text	no children
CDATASection	no children
Entity	Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
Notation	no children

DOM (cont.)

- Node interface

```
interface Node {
   ...
   readonly attribute DOMString nodeName;
   attribute DOMString nodeValue;
   readonly attribute unsigned short nodeType;
   readonly attribute Node parentNode;
   readonly attribute NodeList childNodes;
   readonly attribute Node firstChild;
   readonly attribute Node lastChild;
   readonly attribute Node previousSibling;
   readonly attribute Node nextSibling;
   readonly attribute NamedNodeMap attributes;
   readonly attribute Document ownerDocument;
   Node insertBefore(in Node newChild, in Node refChild)
             raises(DOMException);
   Node replaceChild(in Node newChild, in Node oldChild)
             raises(DOMException);
   Node removeChild(in Node oldChild) raises(DOMException);
   Node appendChild(in Node newChild) raises(DOMException);
   boolean hasChildNodes();
   Node cloneNode(in boolean deep);
};
```

DOM (cont.)

- Document interface

```
interface Document : Node {
   readonly attribute DocumentType doctype;
   readonly attribute DOMImplementation implementation;
   readonly attribute Element documentElement;
   Element createElement(in DOMString tagName)
   raises(DOMException);
   DocumentFragment createDocumentFragment();
   Text createTextNode(in DOMString data);
   Comment createComment(in DOMString data);
   CDATASection createCDATASection(in DOMString data)
             raises(DOMException);
   ProcessingInstruction createProcessingInstruction
     (in DOMString target, in DOMString data) raises(DOMException);
   Attr createAttribute(in DOMString name) raises(DOMException);
   EntityReference createEntityReference(in DOMString name)
             raises(DOMException);
   NodeList getElementsByTagName(in DOMString tagname);
};
```

XML Advantages for B2B

- Integrates data and meta-data (tags)
  - Self-describing
- XMLSchema, Namespaces
  - Defining valid document structure
  - Integrating heterogenous terminology and structures
- XML can be validated against schema (xsd, dtd) outside the application
- Many technologies exist for processing, transforming, querying XML documents
  - DOM, SAX, XSLT, Xpath, Xquery
- XML processing can easily handle schema heterogenity, schema evolution
  - Focus on known element tags, attributes, namespaces …
  - Powerful filter and transformation capabilities
- XML is independent of platforms, middleware, databases, applications …

Literature & Information

- Harold, E.R.:
  The XML Bible (2nd ed.),
  Hungry Minds, Inc., 2001
- Harold, E.R., Means, W.S.:
  XML in a Nutshell,
  O'Reilly, 2001

- www.w3.org/XML