SIU CARBONDALE

SOUTHERN ILLINOIS UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

INTRODUCTION TO XML

Why Learn XML?

- HTML had been redefined as XHTML according to the rules of XML
- Many web applications are dealing with XML
- Cool web techniques like Ajax rely on XML
- The future "Semantic Web" and accompanying technologies are all centered around XML
- Already a **universally** accepted language for data *formatting* and data *exchange* on the web and elsewhere

Introduction

- **XML** – e**X**tensible **M**arkup **L**anguage
- Developed from SGML as a simplified version
- A ***meta****-markup* language
- Deficiencies of HTML and also SGML
  - Lax syntactical rules
  - Many complex features that are rarely used
- HTML is a specific markup language, while XML is used to define a markup language, e.g., XHTML
- Markup languages defined in XML are known as -- *XML applications* – so XHTML is also an application of XML
- XML documents/data can be written by hand or generated by computer
  - Useful for data exchange

The Syntax of XML

- Levels of syntax
  - *Well-formed documents* conform to basic XML rules
  - *Valid documents* are well-formed and also conform to a *schema* which defines details of the allowed contents (i.e., tags, attributes, entities)
- Well-formed XML documents
  - All tags must be closed -- empty tags self-closing
  - Nesting must be complete – no crossing allowed.
  - There is only one *root* tag containing all the other tags in a document
  - Attributes must have a value assigned, the value must be quoted
  - The characters <, >, & can only appear with their special meaning
  - Check http://www.w3.org/TR/2006/REC-xml-20060816/#sec-well-formed for more detailed explanation of the well-formedness

XML Document Structure

- Auxiliary files
  - Schema file
    - DTD or XML Schema
  - Style file
    - Cascading Style Sheets
    - XSLT – a sublanguage of XSL for Transformation
- Breaking file up
  - More manageable, space-saving, reuse, and less inconsistency
  - Using document entities
- Character data
  - <![CDATA …..]]> // can include “anything”, not to be parsed!

DTD -- Document Type Definitions

- A set of *declarations*
  - Define *tags*, *attributes*, *entities*
  - Specify the order and (nesting) composition of tags
  - Specify which attributes can be used with which tags
- General syntax
  - <!keyword …. >
  - Note, DTD itself is not XML!
    - This is why DTD is not usually recommended

Declaring Elements

- General syntax
  - <!ELEMENT *element-name content-description*)>
  - Content description specifies what tags may appear inside the named element and in what order and whether there may be any plain text in the content
- Think of a document in terms of a general tree structure
- A sequence of tags (order matters)
- Alternate tags (order does not matter)
- Multiplicity
  - +
  - *
  - ?
- #PCDATA – *parsable character data*, can include any char except for the special ones: < , >, and &

Declaring Attributes

- **General syntax**
  - **<!ATTLIST *element-name***

    (attribute-name attribute-type default-value?)+ >
- **Default values**
  - **A value –** a default value (in the true sense) to be used if none is specified by the user
  - **#FIXED value –** enforced value and never change
  - **#REQUIRED –** attribute value must be explicitly given by user
  - **#IMPLIED –** the attribute is optional, and no default value is given

Declaring Entities

- General Syntax
  - <!ENTITY [%] *entity-name “entity-value”*>
  - With %: a parameterized entity
  - Without %: a general entity
- Parameterized entities may only be referenced in the DTD
  - i.e., for DTD itself to use
- Form of remote entities
  - <!ENTITY *entity-name* SYSTEM *“file-location”*>
  - File-location URL can pint to anywhere on the Web
  - The entity is replaced with the content of the file at interpretation

A Sample DTD for “Parts”

The following file is called "parts.dtd".

```
<!ELEMENT PARTS (TITLE?, PART*)>
<!ELEMENT TITLE (#PCDATA)>
<!ELEMENT PART (ITEM, MANUFACTURER, MODEL, COST)+>
<!ATTLIST PART type (computer|auto|airplane) #IMPLIED>
<!ELEMENT ITEM (#PCDATA)>
<!ELEMENT MANUFACTURER (#PCDATA)>
<!ELEMENT MODEL (#PCDATA)>
<!ELEMENT COST (#PCDATA)>
```

Sample XML Document

```
<?xml version="1.0"?>
<!DOCTYPE PARTS SYSTEM "parts.dtd">
<?xml-stylesheet type="text/css" href="xmlpartsstyle.css"?>
<PARTS>
  <TITLE>Computer Parts</TITLE>
  <PART>
    <ITEM>Motherboard</ITEM>
    <MANUFACTURER>ASUS</MANUFACTURER>
    <MODEL>P3B-F</MODEL>
    <COST> 123.00</COST>
  </PART>
  <PART>
    <ITEM>Video Card</ITEM>
    <MANUFACTURER>ATI</MANUFACTURER>
    <MODEL>All-in-Wonder Pro</MODEL>
    <COST> 160.00</COST>
  </PART>
  <PART>
    <ITEMի inch Monitor</ITEM>
    <MANUFACTURER>LG Electronics</MANUFACTURER>
    <MODEL> 995E</MODEL>
    <COST> 290.00</COST>
  </PART>
…
</PARTS>
```

Internal and External DTDs

- An XML doc can either contain the DTD declarations directly or can refer to another (extern DTD) file
- Internal
  - <!DOCTYPE *root-element-name* [
    - *declarations*
  -]>
- External file
  - <!DOCTYPE *root-element-name* SYSTEM *"file-name"*>

Namespaces

- XML namespaces provide a simple method for qualifying element and attribute names used in XML docs by associating them with namespaces identified by URI references."
  - For more details, refer to:

    http://www.w3.org/TR/2006/REC-xml-names-20060816/
- An intended namespace needs be declared by

  <element xmlns[:prefix]="URI">
  - The prefix is used to qualify names belonging to the namespace
  - Multiple namespaces can be used in a single document
  - Default namespace does not have prefix given
- DTDs do not support namespaces very well

XML Schemas

- Schema is intended to replace DTD for the description of XML content models
- DTDs have several deficits
  - They do not use XML syntax
  - They do not support namespaces well
  - Data types cannot be strictly specified
    - Example: date vs. string

Schema Fundamentals

- An XML Doc conforming to a schema's rules is considered an *instance* of that schema
- Schema purposes
  - Stipulate the composition structure of valid instances
  - Define the data types of elements and attributes very specifically
- XML Schemas support namespaces
  - The XML Schema language itself is a set of XML tags
  - The application (or instances) is another set of tags
- Schema and instances can both be *uniformly processed* by the same XML parser!

Defining a Schema

- The root of an XML Schema document is the schema tag: <schema …>
- Attributes
  - The `xmlns` attributes introduce the namespaces used for the schema itself
  - The `targetNamespace` attribute declares the name of the *namespace being defined*
  - The `elementFormDefault` attribute with the value “qualified” to indicate to put *non-top-level elements* (i.e., nested elements) also into the target namespace

An Example of XML Schema

```
<?xml version = "1.0" encoding = "utf-8"?>
<xsd:schema
   xmlns:xsd = "http://www.w3.org/2001/XMLSchema"
   targetNamespace = "http://cs.uccs.edu/planeSchema"
   xmlns = "http://cs.uccs.edu/planeSchema"
   elementFormDefault = "qualified">
   <xsd:element name = "planes">
      <xsd:complexType>
         <xsd:all>
            <xsd:element name = "make"
                         type = "xsd:string"
                         minOccurs = "1"
                         maxOccurs = "unbounded" />
         </xsd:all>
      </xsd:complexType>
   </xsd:element>
</xsd:schema>
```

Note: **XMLSchema is** the XML Schema schema for XML Schemas

An Example Instance of the Schema

```
<?xml version="1.0" encoding="utf-8"?>
   <!--  planes1.xml
            A simple XML document for illustration -->
<planes
  xmlns = "http://cs.uccs.edu/planeSchema"
  xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation = "http://cs.uccs.edu/planeSchema/planes.xsd">
    <make> Cessna </make>
    <make> Piper </make>
    <make> Beechcraft </make>
</planes>
```

Note: namespace **XMLSchema-instance** defines several attributes for direct use in any XML documents

Overview of data types

- *Data Type Categories*

  1. Simple (strings only, no attributes and no nested elements)

  2. Complex (can have attributes and nested elements)

- XMLSchema have over 40 data types (predefined for you!)

  - Primitive: string, Boolean, float, …

  - Derived (predefined): byte, decimal, positiveInteger, …

  - User-defined – by specifying constraints on an existing type (called the *base* type)

  - Constrained types using constraints on *facets* of existing types, e.g., (totalDigits, maxInclusive, etc.)

- Both simple and complex types can be either named or anonymous

Simple Type Definition

- DTDs define only global elements (context is irrelevant)

- With XMLSchema, context is essential, and elements can be either:
  1. Local: appears **inside an element** that is a child of schema, or
  2. Global: appears **as a child** of schema

- Defining a simple element type:

  - Use the *element* tag and set the name and type attributes

    <xsd:element name = "bird" type = "xsd:string" />

  - An instance could be:

    <bird> Yellow-bellied sap sucker </bird>

Simple Type (cont')

- **Simple *User-Defined Types***
  - Defined in a simpleType element, using *facets* specified in the content of a restriction element
    - Facet values are specified with the value attribute

```
<xsd:simpleType name = "middleName" >
   <xsd:restriction base = "xsd:string">
      <xsd:maxLength value = "20" />
   </xsd:restriction>
</xsd:simpleType>
```

Complex Types

- **Four *Categories*** *of* *Complex Types*
  1. Element-only elements
  2. Text-only elements
  3. Mixed-content elements
  4. Empty elements

- *Element-only elements*
  - Defined with the **complexType** element
  - Use the **sequence** tag for nested elements that must be in a particular *order*
  - Use the **all** tag if the order is not important

Example

```
<xsd:complexType name = "sports_car">
   <xsd:sequence>
    <xsd:element  name = "make"
                  type = "xsd:string" />
    <xsd:element  name = "model"
                  type = "xsd:string" />
    <xsd:element  name = "engine"
                  type = "xsd:string" />
    <xsd:element  name = "year"
                  type = "xsd:string" />
   </xsd:sequence>
 </xsd:complexType>
```

Validating Instances of Schemas

- Various systems for validating instances against schemas
  - Online http://www.w3.org/2001/03/webdata/xsv
  - XML support libraries include validation: Xerces from Apache, Saxon, Altova XML tools
  - Some IDE's have automatic validation: Altova Spy, Eclipse with Oxygen, Eclipse with XML Buddy Pro

Displaying Raw XML Documents

- Plain XML documents are generally displayed **literally** by browsers
  - Don’t expect display is nicely formatted if no CSS
  - Firefox uses a default CSS if is no style provided by users
  - *WordPad* makes indented display as well
    - better than *Notepad*

Displaying XML Documents with CSS

- An ***xml-stylesheet*** **processing instruction** **(comes as a special tag)** can be used to associate a general XML document with a style sheet

```
<?xml-stylesheet type="text/css" href="planes.css">
```

- The style sheet selectors will specify tags (say ad) that appear in a particular document, e.g.,

```
ad {display: block; margin-top: 15px; color: blue;}
```

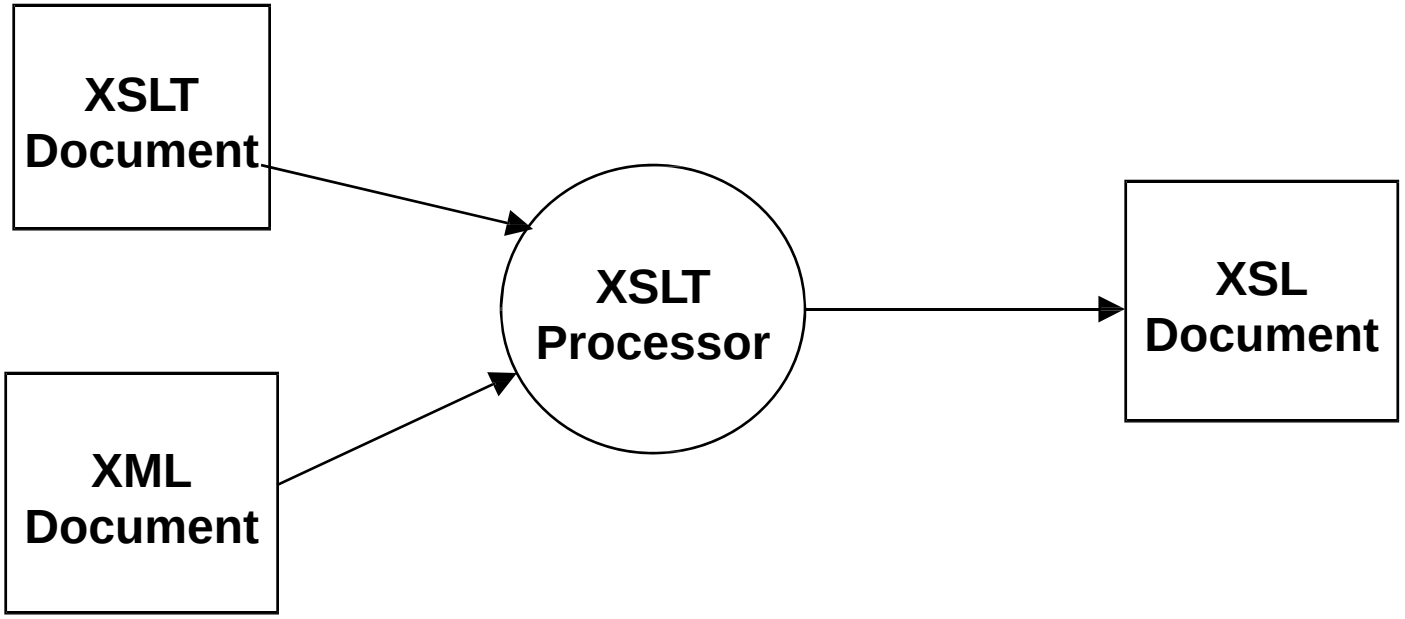
XSLT Style Sheets

- **XSLT** stands for ***XSL*** (eXtensible Stylesheet Language) **T**ransformations
- A family of specifications for processing XML documents
  - **XSLT:** specifies how to transform documents
  - **XPath:** specifies how to select parts of a document and compute values
  - **XSL-FO**: Extensible Stylesheet Language Formatting Objects, used for formatting XML data for output to screen, paper or other media.
- XSLT describes how to transform XML documents (conforming to one XSD) into other XML documents (conforming to another XSD, such as XHTML)
  - XSLT can be used to transform to non-XML documents as well

Overview of XSLT

- XSLT is a *functional-style* programming language (e.g., Lisp)
- Basic syntax is XML
  - There is some similarity to LISP and Scheme
- An XSLT processor takes an XML document as input and produces output based on the specifications of an XSLT document.

XSLT Processing



XSLT Structure

- An XSLT document contains templates
- XPath is used to specify patterns of elements to which the templates should apply
- The content of a template specifies how the matched element should be processed
- The XSLT processor will look for parts from the input document that match a template and apply the content of the template to the parts
- Two work models
  - **Template-driven** works with highly regular data
  - **Data-driven** works with more loosely structured data with a recursive structure (like XHTML documents)
  - We discuss only the template-driven model

XSL Transformations for Presentation

- One of the **most common** applications of XSLT is for display
- A XSLT style sheet can be associated with an XML document by using a **processor instruction**

  <?xml-stylesheet type="text/xsl" href="*stylesheet-ref*"?>

- The example xslplane.xml is an xml file with data about a single plane
  - The file is linked to the style sheet xslplane1.xsl that we will show later

XSLT Organization

- The root element **stylesheet**
  - Specifies namespaces for XSL and for non-XSLT elements included in the stylesheet

```
<xsl:stylesheet xmlns:xsl =
    "http://www.w3.org/1999/XSL/Format"
    xmlns = "http://www.w3.org/1999/xhtml">
```

- Then, elements from XSLT itself will have the prefix **xsl**
- And elements from XHTML will have no prefix (declared as the default namespace)

XSLT Templates

- There must be at least **one** template element in a style sheet
- The value of the `match` attribute is an **XPath expression** which specifies to which nodes the template applies
- Two standard choices for the `match` expression of the first template
  - '/' to match the root node of the entire document structure
  - '*root-tag*' to match the "root" element (the **current** root) to process
- Only the first template in an XSLT is applied automatically
- All other templates are applied only in response to `apply-templates` element which appears within the first template

XPath Basics and Node Selection

- An XPath expression beginning with a **/** specifies nodes in an absolute position relative to the document root node
- Otherwise, the expression specifies nodes relative to the ***current*** *node*, that is the node being processed before the matched node
- The XPath expression '**.**' refers to the current node
- The *apply-templates* tag uses the *select attribute* to choose which nodes should be matched to templates

Producing Transformation Output

- Elements not belonging to XSLT and other text will be **copied** over to the output when the containing template is applied
- The value-of tag causes the select attribute's value to be evaluated and the result be put into the output
  - The value of an element is the text contained in it, including sub-elements (if any)
  - The value of an attribute is plainly the value of it
- Example xslplane1.xsl transforms the xslplane.xml file into XHTML for display purposes

Processing *Repeated* Elements

- File xslplanes.xml contains data about **multiple** airplanes
- The style sheet xslplanes.xsl uses a for-each element to process each plane element in the source document
- A sort element could be included to sort output
  - The element

```
<xsl:sort select="year" data-type="number"/>
```

  - This example specifies sorting by year

XML Processors

- XML processors provide tools in programming languages to **read** in XML documents, **manipulate** them, and **write** them **out**

Purposes of XML Processors

- Four purposes
  - Check the basic syntax of the input document
  - Replace entities
  - Insert default values specified by schemas or DTD's
  - Validate the input document against the specified schemas (or DTD's)
- The basic structure of XML docs is simple and repetitive, so providing library support is reasonable
- Examples
  - Xerces-J from the Apache foundation provides library support for Java
  - Command line utilities are provided for checking *well-formedness* and validity
- Two different standards/models of processing XML files
  - **SAX** -- the Simple API for XML (SAX) is a publicly developed standard for the events-based parsing of XML documents.
  - **DOM** -- as you already know, must be fully loaded in memory**!**

Parsing

- As you know, the process of reading in a document and analyzing its structure is called ***parsing***
- The parser provides as output a structured view of the input document (the raw materials with structures hidden)

The SAX Approach

- With the SAX approach, an XML document is read in serially (as streams)
- As certain conditions, called **events**, are recognized, the event handlers are called for processing the watched elements
- The program using this approach sees only **part** of the document at a time
  - This is good and bad! (?)

The DOM Approach

- In the DOM approach, the parser produces a full in-memory representation of the **whole** input document
  - Because of the well-formedness rules of XML, the structure is a **tree**
- Advantages of DOM
  - Whole document can be restructured
  - Flexible access to any parts of the doc at any times
  - Processing can be **delayed** until the entire doc is checked/validated
- **A major disadvantage** is that a **very large** document may not fit in memory entirely

Web Services

- Allow **interoperation** among software components in *different* systems written in *different* languages
- We expect Servers to provide *software services* rather than just documents
- Remote Procedure Call are needed
  - DCOM and CORBA provide implementations
    - DCOM is Microsoft specific
    - CORBA is cross-platform

Web Service Protocols

- Three roles in web services
  - Service providers
  - Service requestors
  - Service registry (service broker)
- The *Web Services Definition Language* provides a standard way to describe services
  - The Universal Description, Discovery and Integration (UDDI) provides a standard **languag**e to provide information about services in response to a query (another one is WSDL**)**
  - SOAP is such a Web Service Protocol used to specify requests and responses

Lab

- **Part1:** Create a DTD for a catalog of cars where each car has the child elements `make`, `model`, `year`, `color`, `engine`, `number_of_doors`, `transmission_type` and `accessories`. The `engine` element has the child elements `number_of_cylinders` and `fuel_system`(carbureted or fuel injected). The `accessories` element has the attributes `radio`, `air_conditioning`, `power_windows`, `power_steering` and `power_brakes`, each of which is required and has the possible values `yes` and `no`. Entities must be declared for the names of popular car makes.
- **Part2:** Create an XML document with at least three instances of the `car` element defined in the DTD of above. Process this document using the DTD and produce a display of the raw XML document