Introduction to XML

Bebo White
bebo@slac.stanford.edu

InterLab 2006
FermiLab
October 2006

Tutorial Description

With your HTML knowledge, you have a solid foundation for working with markup languages. However, unlike HTML, XML is more flexible, allowing for custom tag creation. This course introduces the fundamentals of XML and its related technologies so that you can create your own markup language.

Topics*

- XML well-formed documents
- Validation concepts
- DTD syntax and constructs
- W3C Schema syntax and constructs
- XSL(T) syntax and processing
- XPath addressing language
- Development and design considerations
- XML processing model
- XML development and processing tools

* Tutorial plus references

What Is Markup?

- Information added to a text to make its structure comprehensible
- Pre-computer markup (punctuational and presentational)
  - Word divisions
  - Punctuation
  - Copy-editor and typesetters marks
  - Formatting conventions

Computer Markup (1/3)

- Any kind of codes added to a document
  - Typesetting (presentational markup)
    - Macros embedded in ASCII
    - Commands to define the layout
    - MS Word, TeX, RTF, Scribe, Script, nroff, etc.
      - *Hello* → **Hello**
      - /Hello/ → *Hello*
  - Declarative markup
    - HTML (sometimes)
    - XML

Computer Markup (2/3)

- Declarative markup (cont)
  - Names and structure
  - Framework for indirection
  - Finer level of detail (most human-legible signals are overloaded)
  - Independent of presentation (abstract)
  - Often called “semantic”

Computer Markup (3/3)

- Semantic Markup
  - Authors put annotations into their texts to help the publisher to understand what type of text this is (e.g. “this is a heading”)
  - Annotations are agreed between author and publisher
- Publisher decides on the layout
  - Descriptive markup
    - Describing content not the layout
  - Markup to support search in documents
    - Words in headings are more important than in footnotes
    - Markup for machines vs. markup for humans

Markup – ISO-Definitions

- **Markup** – Text that is added to the data of a document in order to convey information about it
- **Descriptive Markup** – Markup that describes the structure and other attributes of a document in a non-system-specific way, independently of any processing that may be performed on it
- **Processing Instruction (PI)** – Markup consisting of system-specific data that controls how a document is to be processed

Markup Language Features

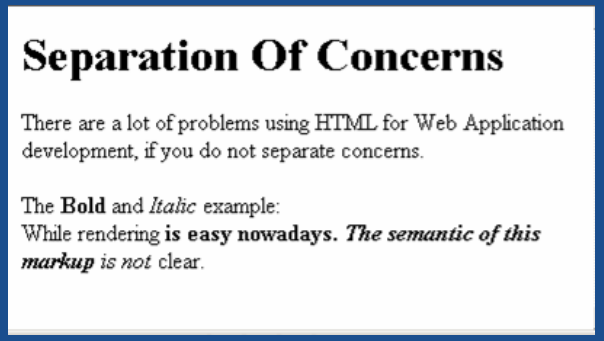
- Stylistic (appearance)
  - <I><B><U>
- Structural (layout)
  - <P>
<H2>
- Semantic (meaning)
  - <TITLE>
  - <META NAME=keywords CONTENT = " …... " >
- Functional (action)
  - <BLINK>
  - <A HREF = "[link]">Click here</A>

Hypertext Markup Language (HTML)

Hypertext Markup Language

- HTML – The Markup Language used to represent Web pages for viewing by people
  - Rendered and viewed in a Web Browser
  - Not extensible
- Documents
  - Easy to write – Markup your data with tags
  - Platform independent
  - Can contain links to Images, documents, and other pages
  - HTML is an application/instance of SGML **(Standard Generalized Markup Language, ISO 8879:1986 – used for defining Markup Languages)**
- For further information: http://www.w3.org/MarkUp/

Some Problems (1/2)



Some Problems (2/2)

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">
<html><head>
    <title>The Some Problems Example</title>
</head><body>
    <H1>Separation Of Concerns</h1>
    There are a lot of problems using
    HTML for <WebEngineering>Web Application development</WebEngineering>,
    if you do not separate concerns. <P>

    The <b>Bold</b> and <i>Italic</i> example: <br>
    While rendering <b> is easy nowadays.<i> The semantic of this markup </b> is
    not </i> clear.

</BODY></HTML>
```

- REMEMBER: Do not develop Applications in this manner!

Observations on HTML

- Powerful for Presentation (Focus on Client-Side)
  - Cascading Style Sheets (CSS)
  - Allows for dynamic behavior using scripting/ DHTML
  - Allows for proprietary extension (ActiveX, plug-ins, etc.).
- Easy to write and generate, **but**:
  - Difficult to parse
  - No support for extending semantics, e.g. using your own tags
- Difficult to apply disciplined approaches

eXtensible Markup Language (XML)

XML (1/2)

- The eXtensible Markup Language
- XML is a universal format for structured documents and data on the Web
- XML is a standard, interoperable way to describe data for flexible processing
  - Multi-format delivery
  - Schema-aware information retrieval
  - Transformation and dynamic data customization
  - Archival: standardized, self-describing

XML (2/2)

- http://www.w3.org/XML/
- XML looks like markup (e.g., HTML) **but** in this context the interpretation of data is the job of the application
- XML tags/elements/attributes are not predefined
- XML uses a Document Type Definition (DTD) or an XML Schema to describe data
- XML with a DTD or XML Schema is designed to be self-descriptive

XML History

- 1996 Development started
- 1997 Public Drafts
  - E.g. Provided in paper form at WWW6, Santa Clara, CA
- February, 1998 W3C REC
  - Based on experience: simplified form of SGML
  - XML derived from SGML – both are used for defining Markup Languages
  - XML = 80% of SGML´s capabilities, 20% of SGML´s complexity

The W3C Standards* Process

- World Wide Web Consortium (W3C)
- Development is organized into WGs.
  - Working Group (~10) - set agenda /decide
  - Special Interest Group (~100) - discuss/recommend
  - W3C members (~500) - vote
  - W3C Director (TimBL) - may veto
- The public--comment on public WDs; adopt/reject

XML Facts

- Important for Web development because it removes two constraints:
  - Dependence on a single, inflexible Document Type (HTML);
  - The complexity of full SGML, whose syntax allows many powerful but hard-to-program options
- XML was not designed to do anything
- XML is free and extensible
- XML complements (not replaces) HTML

XML and HTML

- XML was designed to “carry” data
- Two different goals:
  - XML – describe data and focus on what it is
  - HTML – display data and focus on how it looks

XML Characteristics

- **Well-Formed** – An XML document is well-formed if it complies to the following rules:
  - Elements have an open and close **tag:** <tag>content</tag>
  - Empty elements are closed by “ **/** ” e.g. <emptyelem/>
  - Attribute values are quoted
- **Valid** – An XML document is well-formed and if its content conforms to the rules in its document type definition or schema
  - Validity allows an application to make sure the XML data is complete, is formated properly, and has appropriate attribute values.

The Two Worlds of XML

- Markup of documents: the original
  - This perspective is our focus here
  - Document representation was the primary problem XML was created to solve
- Data exchange and protocol design
  - XML turned out to fill important gaps
  - Relational databases needed a way to share records and multi-table data
  - Protocol designers wanted a way to encapsulate structured data

The Two Worlds United

- Documents and “semi-structured” data share features
  - Hierarchical structure
  - String content
  - Variations in structure
- Their applications also share needs
  - Need for a lingua franca, independent of APIs
  - Ability to cope with international characters
  - “Fit” with WWW and HTTP.

XML is More General

- Tags label arbitrary information units
  - More suited to multiple purposes
  - “Looking right” is needed but not enough
- Supports custom information structures
  - If you have “price” or “procedure”, you can make a tag for it, and validate its usage
  - Can support many different information models
    - E.g., molecular models, vector graphics, etc.
- More “teeth” to enforce consistent syntax
  - Works hard to avoid semi-interoperable docs

Better Rendering than HTML

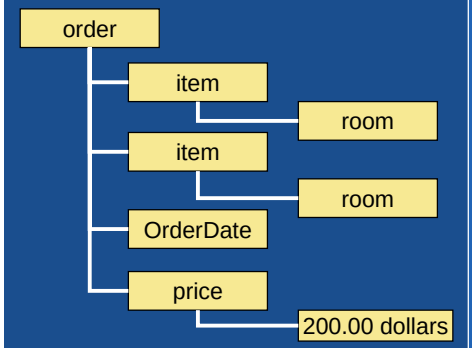
- Fully internationalized
  - Also better for visually-impaired users
- Supports multiple renderings
  - Customize to the user, time, situation, device
  - Separates formatting from structure
  - And processing other than rendering
- Large documents don’t break it
  - Easy to trade off server/client work
  - Artificial “next tiny bit” links no longer necessary
  - No searches that fail because big doc was split
- XHTML is XML-conforming flavor of HTML
  - Clean existing HTML is already close...

XML Treats Documents like Databases

- XML brings benefits of DBs to documents
  - Schema to model information directly
  - Formal validation, locking, versioning, rollback...
- But
  - Not all traditional database concepts map cleanly, because documents are fundamentally different in some ways

XML Example

- A way of representing information
- XML documents (application of XML) are composed of **elements** and *attributes*




```
<?xml version="1.0“encoding="ISO-8859-1” ?>
<order OrderID="10643">
    <item>
        <room id="Room10"/>
    </item>
    <item>
        <room id="Room11"/>
    </item>
    <OrderDate
        ts="2005-10-17T00:00:00"/>
    <price>200.00 dollars</price>
</order>
```

What is Structure

- To Relational Database theorists, structure is:
  - Tables with fixed sets of non-repeating named fields, that have little internal structure
  - E-R diagrams with fixed number of nodes
- Structured documents are different:
  - The order of SECs, Ps, etc. matters (a lot)
  - Many hierarchical layers (which text crosses)
  - Text/graphic data mixes with aggregate objects
  - Optional or repeatable sub-parts abound
  - Interaction with natural language phenomena
- These are very different requirements

When Structure is Essential

- Large scale data
- Data with individual parts you care about
  - (like price-tag, tool-list, citation, author,...)
- Need for good navigation tools
- Mission-critical information
- Information that must last
- Multi-author publishing process
- Multiple delivery media

What's the Difference?

- Without structure
  - Data conversion is far more expensive
  - Multi-platform and/or multi-media delivery require re-authoring and hand-work
  - Paper production is inconsistent
  - Late format changes are far more risky
  - Retrieval is prone to many false hits
- "Pay me now, or pay me later"

XML Design Principles

- Straightforwardly usable over the Internet
- Support for a wide variety of applications
- Compatible with SGML
- Make writing XML programs easy
- Avoid optional features
- Human-readable (if not terse) markup
- Formal and concise design
- Design produced quickly

Opportunities with XML

- Scalability and openness of Web solutions
- “Rich clients” for complex information
  - Dynamic user views
- XML as interprocess communication protocol for “data” (as opposed to “text”)
- eCommerce integration
- New methods of creation
  - Schema combination/composition
  - Free-form, schema-less data development

Web Usage

- XML works with familiar Web paradigms
  - Locations are expressed as URIs
  - High interoperability because of few options
  - Easily implementable and usable
  - Robust against network failures
  - Avoids serving schemas every time with documents
    - (but can do better validation anyway, when needed)

Some Additional XML Details

- Well-formedness
- Error handling
- Case sensitivity
- HTML compatibility

Well-formedness (1/2)

- Document has a single root element, and
- Elements nest properly
- Entities are whole subtrees (not `</P><P>`)
- No elements omission (close what you open)
- Attributes must be quoted
- < and & must always be escaped in some way
- A document can be well-formed (and parsable) whether or not it fits a given schema

Well-formedness (2/2)

```
<root>
  <child>
    <subchild>……</subchild>
  </child>
</root>
```

Tutorial Outline

- Part 1: The basics of creating an XML document
- Part 2: Developing constraints for a well formed XML document
- Part 3: XML and supplementary technologies

Part 1: Background for XML

- An eXtensible Markup Language (XML) document describes the *structure of data*
- XML and HTML have a similar syntax … both derived from SGML
- XML has no mechanism to specify the format for presenting data to the user
- An XML document resides in its own file with an '.xml' extension

Main Components of an XML Document

- Elements: <hello>
- Attributes: <item id="33905">
- Entities: < (<)
- Comments: <!– blah blah -->
- Advanced Components
  - CData Sections
  - Processing Instructions

The Basic Rules

- XML is case sensitive
- All start elements must have end elements
- Empty elements may be “self-closing” – e.g., <img…./>,
- Elements must be properly nested
- XML declaration is the first statement
- Every document must contain a root element
- Attribute values must have quotation marks
- Certain characters are reserved for parsing

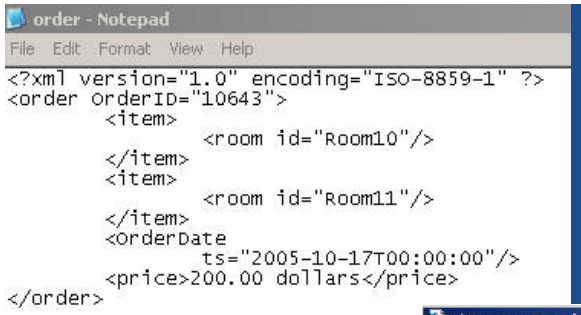
Common Errors for Element Naming

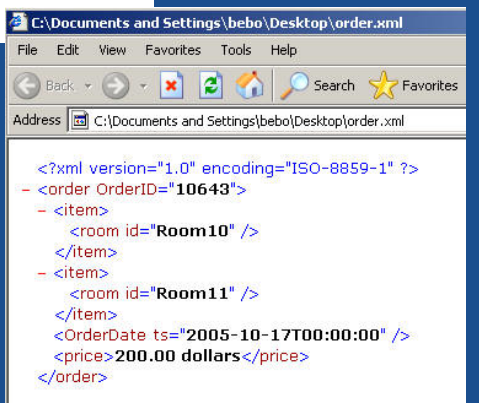
- Do not use white space when creating names for elements
- Element names cannot begin with a digit, although names can contain digits
- Only certain punctuation allowed – periods, colons, and hyphens

Try It!

- Open XML Editor (www.philo.de/xmledit)
- Peter’s XML Editor (www.iol.ie/~pxe)

- Notepad (but cannot check for well-formedness and validity





Elements vs. Attributes

- Data can be stored in child elements or in attributes

```
<person sex="female">
  <firstname>Hillary</firstname>
  <lastname>Clinton</lastname>
</person>
```

```
<person>
  <sex>female</sex>
  <firstname>Hillary</firstname>
  <lastname>Clinton</lastname>
</person>
```

Problems Using Attributes

- Attributes cannot contain multiple values (child elements can)
- Attributes are not easily expandable (for future modifications)
- Attributes cannot describe structures (child elements can)
- Attributes are more difficult to manipulate with program code
- Attribute values are not easily tested against DTDs or Schemas
- Metadata should be attributes; data should be elements

Part 2: Legal Building Blocks of XML

- A Document Type Definition (**DTD**) or XML Schema allows the developer to create a set of rules to specify legal content and place restrictions on an XML file
- If the XML document does not follow the rules contained within the DTD or Schema, a parser generates an error
- An XML document that conforms to the rules within a DTD or Schema is said to be **valid**

What are the Parts of an XML Document?

- The DTD
- Elements
- Attributes
- General entities
- Character references
- Comments
- Marked sections
- Processing instructions
- Notations
- Identifiers and catalogs

Error Handling

- "Draconian error handling"
  - Major errors cause processor to stop passing data in the "normal way"
- Fatal errors:
  - Ill-formed document
  - Certain entity references in incorrect places
  - Misplaced character-encoding declarations
- This helps save huge $ on error-recovery

Case Sensitivity

- HTML is
  - Case-insensitive for tag names: **<P> = <p>**
  - Case-sensitive for entity names: **< ≠ <**
- XML is case-sensitive for both!
  - Unicode standard advises against case-folding
  - Folding is not well-defined for all languages
    - Turkish has two lower-case i's, only one upper
    - In languages with no accented caps, can't reverse
    - Error-prone for programmers
- XHTML uses lower case

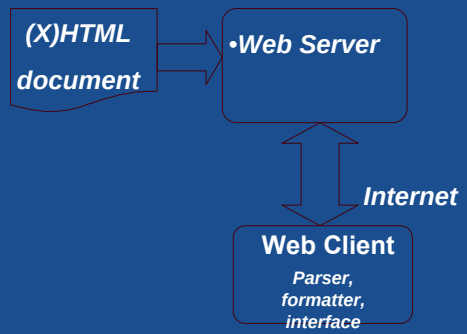
Practice Validating XHTML

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" lang="en">
```

XML System Architectures

An (X)HTML System

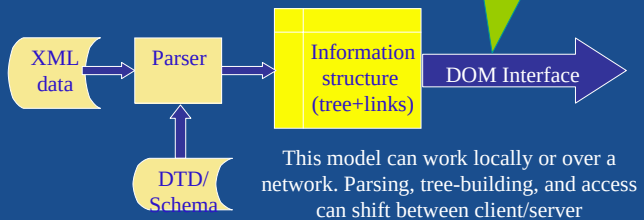


How Do You Get the Data?

Documents, stylesheets, and other data can *all* be expressed in XML.

But their information is accessed directly.

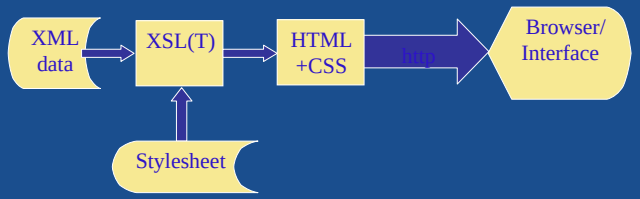
Any application can plug in via an API called "Document Object Model"



This model can work locally or over a network. Parsing, tree-building, and access can shift between client/server

Server-side XML Publishing

Server transforms to HTML/CSS;
Ship to client browser for display



Very common current strategy;
Leverages current technology

XML Everywhere

- XML separates representation from structure
  - So you can use the same parsers, network protocols, tree managers, and APIs to access documents, stylesheets, search and query, etc.
- XML allows separating application parts
  - So you can mix and match formatters, search engines, networks and protocols, etc.
- XML separates out semantics
  - So you can control style or search semantics without having to mangle your documents to do it

HTML Compatibility

- XHTML is an XML application
  - One schema among many (probably a popular one, of course)
- Web browser should start supporting generic XML regardless of tag-set.
  - Don’t hard-code sizes and names

Footnote

- XML is text, but isn't meant to be read
  - Applications can store their data or respond in Web-compliant style (Text) instead of binary format
- XML is verbose by design
  - Data + Markup is in most cases larger than a binary format – but disk space is cheap, HTTP supports compression on the fly (gzip)
- As XML defines Markup Languages…
  - XHTML, XSL, XForms, etc. are applications of XML

The XML “Alphabet Soup” (1/3)

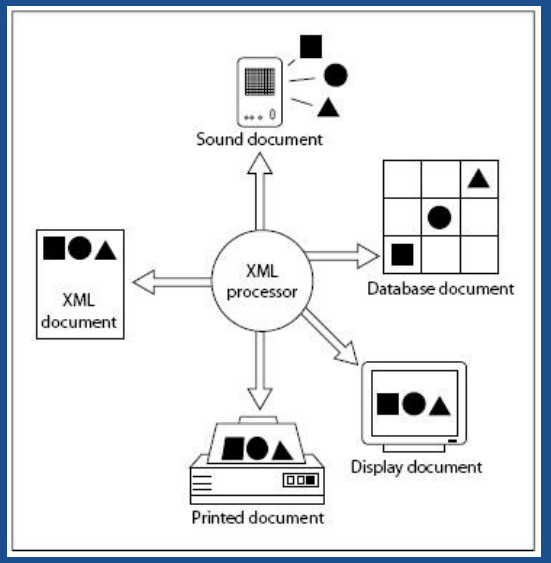
✓ **XML**	Extensible Markup Language	Defines XML documents
✓ **XSL**	Extensible Stylesheet Language	Language for expressing stylesheets; consists of XSL(T) and XSL-FO
✓ **XSL(T)**	XSL Transformations	Language for transforming XML documents
XSL-FO	XSL Formatting Objects	Language to describe precise layout of text on a page
Data Island	XML data embedded in a HTML page	
Data Binding	Automatic population of HTML elements from XML data	
Namespace	A collection of names, identified by a URI reference, which are used in XML documents	

The XML “Alphabet Soup”(2/3)

✓ **DTD**	Document Type Definition	Non-XML schema
✓ **DOM**	Document Object Model	API to read, create and edit XML documents; creates in-memory object model
✓ **SAX**	Simple API for XML	API to parse XML documents; event-driven
✓ **XML Schema (XSD)**	XML Schema Definition	an XML based alternative to DTD
✓ **XPath**	XML Path Language	A language for addressing parts of an XML document, designed to be used by both XSL(T) and XPointer
✓ **XPointer**	XML Pointer Language	Supports addressing into the internal structures of XML documents
✓ **XLink**	XML Linking Language	Describes links between XML documents
XQuery	XML Query Language (draft)	Flexible mechanism for querying XML data as if it were a database

The XML “Alphabet Soup”(3/3)

SOAP	Simple Object Access Protocol	A simple XML based protocol to let applications exchange information over HTTP
WSDL	Web Services Description Language	An XML-based language for describing Web services and how to access them
WAP	Wireless Application Protocol	The leading standard for information services on wireless terminals like digital mobile phones
WML	Wireless Markup Language	WAP uses the mark-up language WML (not HTML)



XML Information Set

- What data in an XML document “counts”?
  - Elements, attributes, content
  - Order and hierarchy of elements
  - No whitespace within tags
  - All whitespace within elements
  - Not which kind of quotes around attributes
- Required for interoperability
  - Applications must not count nodes differently
  - W3C “Document Object Model” is related
    - DOM is an API for XML, not an O.M.

Document Analysis

- Cycle of steps; repeat until out of time
- Identify project requirements/audience
- Using those, identify information items in the document that could be important
- Make sure you have a way to use that information
- Identify restrictions on those items
- Identify structural constraints that may be needed
- Identify non-semantic features that may be important for presentation, etc.

Project Requirements

- Know the audience/readers
- Know the authors
- Don’t forget the editorial/clerical staff
- These 3 groups are the experts, you are the detail person
- Don’t make a lifetime commitment to your processing model, but have one in mind; analysis without limitations is dangerous

Identifying Information Items

- This is pretty much a manual process
- Often best done with paper and highlighters and post-its
- In later stages, adding tags to a text transcript can be useful.
- The more documents you’ve looked at and thought about, the easier this becomes.

Issues to Think About

- Cross-references
- Structural divisions (headings, blurbs, ambiguities)
- Tradeoff between freedom and processing
- Normalization of data items
- What external data and catalogs may exist

Restrictions on Data Items

- Content model
- Data values (are there controlled or semi-controlled vocabularies?)
- Are there “authority files” for large open sets (like lists of authors)
- How variable is the content, and how realistic the idea to normalize it.

IE Data Islands

What is a Data Island?

- XML data embedded in an (X)HTML document
- Unique to IE
- Uses the “unofficial” <xml> element

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<CATALOG>
        <CD>
                <TITLE>Living With War</TITLE>
                <ARTIST>Neil Young</ARTIST>
                <COUNTRY>USA</COUNTRY>
                <COMPANY>Reprise</COMPANY>
                <PRICE>12.99</PRICE>
                <YEAR>2006</YEAR>
        </CD>
        <CD>
                <TITLE>Modern Times</TITLE>
                <ARTIST>Bob Dylan</ARTIST>
                <COUNTRY>USA</COUNTRY>
                <COMPANY>Sony</COMPANY>
                <PRICE>13.99</PRICE>
                <YEAR>2006</YEAR>
        </CD>
        <CD>
                <TITLE>A Bigger Bang</TITLE>
                <ARTIST>Rolling Stones</ARTIST>
                <COUNTRY>UK</COUNTRY>
                <COMPANY>Virgin</COMPANY>
                <PRICE>13.99</PRICE>
                <YEAR>2005</YEAR>
        </CD>
</CATALOG>
```

```
<html>
<body>
<h1>Some of Bebo's Favorite New CDs</h1>

<xml id="cdcat" src="cd_catalog.xml"></xml>

<table border="1" datasrc="#cdcat">
<tr>
<td><span datafld="ARTIST"></span></td>
<td><span datafld="TITLE"></span></td>
</tr>
</table>

</body>
</html>
```

DTD
(Document Type Definition)

Content Models

- These are modeled on regular expressions
- In DTD, each element has one content model for all time
- Similarly, each element has one set of attributes for all time
- Attributes and content models are completely independent

Ambiguity

- A content model is ambiguous if it contains an alternation (a | b) where the content models a and b cannot be distinguished by their first element.
- A content model is ambiguous if an optional occurrence indicator is followed by a submodel whose first element is not different.

Web-compliant Data Definitions

- Extensible Markup Language (XML) 1.0 (Third Edition)
  - W3C Recommendation 04 February 2004
  - http://www.w3.org/TR/REC-xml/
  - "The function of the markup in an XML document is to describe its storage and logical structure and to associate attribute-value pairs with its logical structures. XML provides a mechanism, the document type declaration, to define constraints on the logical structure and to support the use of predefined storage units."
  - **Document Type Declaration** – Contains or points to markup declarations that provide a grammar for a class of documents. This grammar is known as a document type definition, or DTD.
  - **Document Type Definition** – Set of markup declarations included in or referenced by an XML Document.
- Design using e.g. Diagramming Technique

Why Use a DTD?

- A single DTD ensures a common format for each XML document that references it
- An application can use a standard DTD to verify that data that it receives from the outside world is valid
- A description of legal, valid data further contributes to the interoperability and efficiency of using XML

XML 1.0 DTDs

- DTDs let you say:
  - What element types can occur and where
  - What attributes each element type can have
  - What notations are in use
  - What external entities can be referenced
- Standard DTDs exist in almost every domain
  - Some repositories exist, such as xml.org

XML Declaration

<?xml ?>

- Not required, but typically used
- Attributes include:
  - version
  - encoding – the character encoding in the document
  - standalone – if yes no external DTD required
- <?xml version=”1.0” encoding=”UTF-8”>
- <?xml version=”1.0” standalone=”yes”>

Document Type Definition

- **Document Type Definition Syntax**
  - **Document Type Definition** ::= XMLDecl? Misc* (**doctypedecl** Misc*)?
  - **XMLDecl** ::= '<?xml' VersionInfo EncodingDecl? SDDecl? S? '?>‘
  - **VersionInfo** ::= S 'version' Eq (""" VersionNum """ | "" VersionNum "")
  - **Eq** ::= S? '=' S?[26]
  - **VersionNum** ::= ([a-zA-Z0-9_.:] | '-')+
  - **Misc** ::= Comment | PI | S
  - S ::= White Space
- Example - <?xml version="1.0"?>

Document Type Definition

- **Document Type Definition Syntax**
  - Doctypedecl ::= '<!DOCTYPE' S Name (S ExternalID)? S? ('[' (Markupdecl | DeclSep)* ']' S?)? '>'
  - DeclSep ::= PEReference | S
  - Markupdecl ::= elementdecl | AttlistDecl | EntityDecl | NotationDecl | PI | Comment
- Types of Markup Declaration:
  - Element Type Declaration
  - Attribute-List Declaration
  - Entity Declaration
  - Notation Declaration

DOCTYPE

<!DOCTYPE …>

- Specify a DTD for the document
  - Refer to a DTD using a URI
  - Include a DTD inline as part of the document
- Example: Refer to a DTD
- <!DOCTYPE *order* SYSTEM "*http://a.b/order.dtd*">

Mixed Specification

- Mixed ::=
  '(' S? '#PCDATA' (S? '|' S? Name)* S? ')*'
  | '(' S? '#PCDATA' S? ')'
- Name must not appear more than once
- **Example**
  - **(#PCDATA)** – Only parsed Character Data allowed (= Text). Restricts all Child-Elements to be of Type Text.

Basic Operators

- Joining
  - Sequence **`a,b,c`**
  - Alternation **`a | b | c`**
- Grouping **`(a)`**
- Repetition
  - 0 or more **`a*`**
  - 1 or more **`a+`**
  - Optional **`a?`**

Data

- #PCDATA
- CDATA
- Element names
- Model groups
- Mixed content ***(#PCDATA | x | …)****
- ***ANY***
- ***EMPTY***

PCDATA

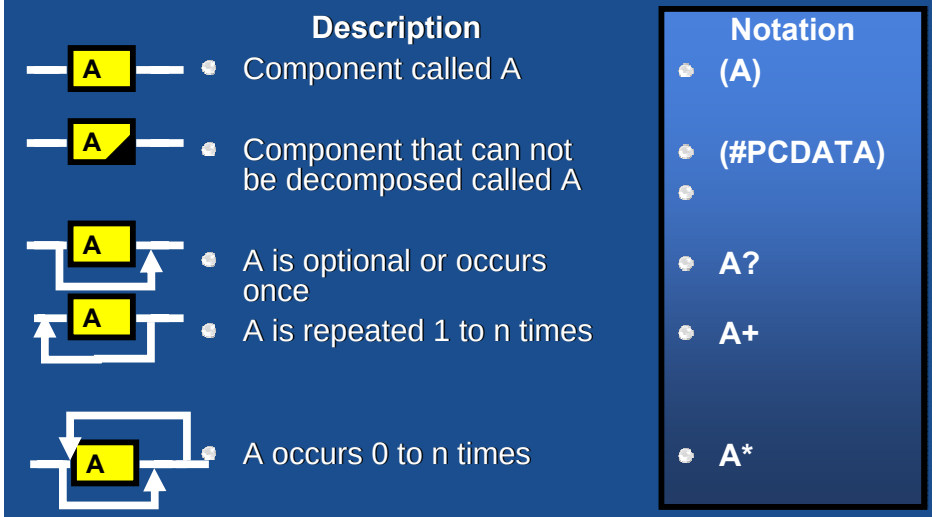
- Parsed character data
- Text occurring in a context in which markup and entity references may occur

CDATA

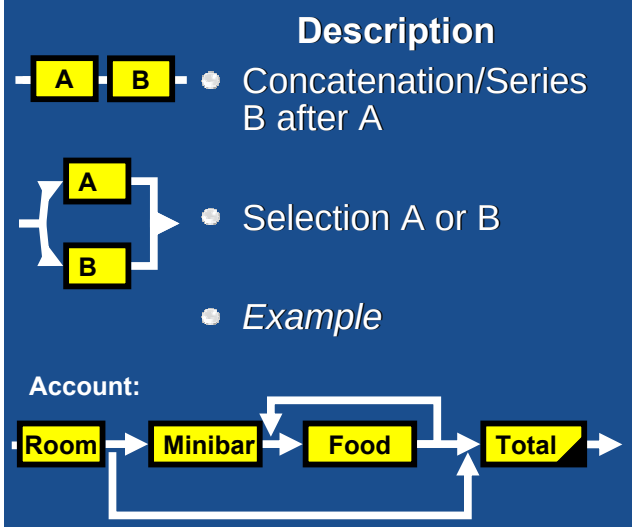
<![CDATA[ ]]>

- Allows to define special sections of character data, which the processor does not interpret as markup
- Anything inside is treated as plain text
- Example:
- <![CDATA[<ThisIsNoElement why="it is just data in a CDATA section"/>]]>

Diagramming Technique (1/2)



Diagramming Technique (2/2)



Description

- Concatenation/Series B after A
- Selection A or B
- *Example*

Notation

- **(A,B)**
- **(A|B)**
- ***Example***

Account= (Room, (Minibar,Food+)?, Total)

Total= (#PCDATA)

Declaration, Definition, Data

```
<?xml version="1.0"?>
<!DOCTYPE account [
  <!ELEMENT account (room,(minibar,food+)?,total)>
  <!ATTLIST account AccountID ID #REQUIRED>
  <!ELEMENT room EMPTY>
  <!ATTLIST room number NMTOKEN #REQUIRED>
 <!ELEMENT minibar EMPTY>
  <!ELEMENT food EMPTY>
  <!ATTLIST food price CDATA #REQUIRED>
<!ELEMENT total (#PCDATA)>
]>
<account AccountID="a3499bxdz">
        <room number="R101"/><minibar/>
        <food price="10.00"/>
        <food price="15.00"/>
<total>28.00
</total>
</account>
```

```
1
2 <!DOCTYPE account SYSTEM "account.dtd" >
3 <account AccountID="a3499bxdz">
4 <room number="R101"></room>
5 <minibar></minibar>
6 <food price="10.00"></food>
7 <food price="15.00"></food>
8 <total>28.00</total>
9 </account>
```

```
1 <!ELEMENT account (room,(minibar,food+)?,total)>
2   <!ATTLIST account AccountID ID #REQUIRED>
3   <!ELEMENT room EMPTY>
4   <!ATTLIST room number NMTOKEN #REQUIRED>
5  <!ELEMENT minibar EMPTY>
6   <!ELEMENT food EMPTY>
7   <!ATTLIST food price CDATA #REQUIRED>
8 <!ELEMENT total (#PCDATA)>
```

Cautions Concerning DTDs

- All element declarations begin with <!ELEMENT and end with >
- The ELEMENT declaration is *case sensitive*
- The programmer must declare all elements within an XML file
- Elements declared with the #PCDATA content model can not have children
- When describing sequences, the XML document must contain exactly those elements in exactly that order.

The DTD (schema)

- A DTD is a simple schema, based on SGML
- They consist of declarations for the parts:
  - `<!ELEMENT CHAP (TI, SEC*, SUM)>`
  - `<!ATTLIST P ID ID #IMPLIED>`
  - `<!ELEMENT P (#PCDATA)>`
- Can reference from DOCTYPE, or include:
-
```
<!DOCTYPE book SYSTEM “book.dtd” [
   <!ELEMENT P (#PCDATA)>…
]>
```
- Other schema languages are available
  - They use XML syntax (why not?)

Terminology (1/4)

- Element: a text feature distinguished by markup
- Tag: a string in angle brackets. <a> or . Two tags delimit an element
- Content: anything in an element (children in the parse tree) tags and characters between an element’s tags
- Attribute: a (name, value) pair associated with an element
- Element Type Name: a string like “p” or “img” that identifies the type of an element

Terminology (2/4)

- Entity: abstraction of an item of data storage.
- General entity: entity whose text is contained in its declaration.
- External entity: entity whose content is stored externally to its declaration
- Declaration: meta-markup that declares entities, content models, etc.
- Document instance: the tags and content in an XML document, not counting declarations

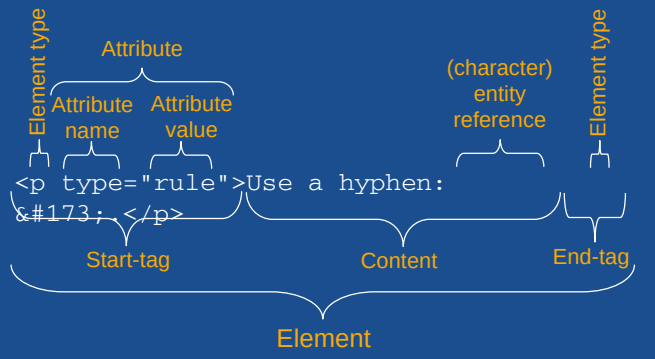
Terminology (3/4)

- Document Type declaration (DOCTYPE): declaration of root element of a document instance, can refer to:
- External subset: DTD (XML declarations) stored as an external entity.
- Internal subset: declarations contained within a DOCTYPE declaration. ATTLIST declarations must be parsed, and interpreted.

Terminology (4/4)

- Content Model: description of restrictions on the content of an element
- Model Group: content model subexpression in parentheses
- Repetition indicator: *, +, ?
- Prolog: All of the stuff before the document instance starts.

Anatomy of an Element



Elements

- Identify structural/semantic components
- Can (usually do) have children
- Represented by start-tags and end-tags:
  - <P>Hello, world.</P>
- Some elements are EMPTY
  - Special syntax so parser knows: <HR/>
- Schemas control what sub-element patterns can occur with any given type of element
- Order matters / Context does not

Element Type Declaration

- Allows to define name of an element and its Content Model
  - <!ELEMENT S Name S Content-Specification>
- Name is the element type being declared
- Content-Specification:
  - **ANY** – Any use (assumed when no content model is provided)
  - **EMPTY** – No sub-elements allowed
  - **Mixed** or **Children s**pecification

Children Specification

- Each name is the type of an element which may appear as a child, as described in the grammar:
- Syntax:
  - Children ::= (choice | seq) ('?' | '*' | '+')?
  - cp ::= (Name | choice | seq) ('?' | '*' | '+')?
  - choice ::= '(' S? cp (S? '|' S? cp)+ S? ')‘
  - seq ::= '(' S? cp (S? ',' S? cp)* S? ')'
- **Example**
  - **(room,total)** – Sequence of two elements of type room and total.

Attributes (1/2)

- Specify properties/characteristics of elements
  - That generally apply to the elements as wholes
- Values are atomic strings
  - Though applications may impose more structure
- Represented by assignments within start-tags:
  - `<P TYPE="SECRET" ID="FOO">`
- Schemas control what attributes can occur on any given type of element
- One special type: ID, unique per document
- Attributes are not ordered

Attributes (2/2)

- Data types
- Default values / omissability
- ***<!ATTLIST p***
  type (summary | body) “body”
  id ID #IMPLIED
  prefix CDATA “”>

Attribute-List Declaration

- **Attributes** – Used to associate name-value pairs with elements.
- Attribute-List Declaration defines
  - Attributes bound to an Element
  - Type Constraints for these Attributes
  - Default Values for Attributes
- Syntax:
  - AttlistDecl ::= '<!ATTLIST' S Name AttDef* S? '>'
  - AttDef ::= S AttributeName S AttType S DefaultDecl

Attribute Types (1/2)

- **String Type**
  - **CDATA** – Value is any literal string
- **Tokenized Types**
  - **ID** – Value must match name production and appear not more than once (only one ID per Element!)
  - **IDREF**, **IDREFS** – Value(s) must match ID attribute on some element in the document
  - **ENTITY**, **ENTITIES** – Value(s) must match name of unparsed entity
  - **NMTOKEN**, **NMTOKENS** – Values(s) must match NMToken production

Attribute Types (2/2)

- **Enumerated Types**
  - **(v1|…|vn)** – Value is one of the values provided in the declaration
- Example:
  - <!ATTLIST elemname myenumtype (true|false|dontknow) 'true'>

Attribute Defaults

- **Attribute Default** – Defines whether an attribute's presence is required and if not how to deal with it
- Syntax:
  - DefaultDecl ::= '#REQUIRED' | '#IMPLIED' | (('#FIXED' S)? AttValue)
  - #Required – Attribute must be specified
  - #Implied – Attribute is optional
  - #Fixed – Required attribute; value is specified in quotes
  - AttValue – Contains the declared default value

General Entities (1/2)

- A lexical mechanism for inclusion
  - But, constrained to including subtrees
  - This preserves fragment parsability
  - This allows lazy evaluation of structure nodes
- Also used for referring to graphic or other non-directly-XML data objects
- References occur in the document instance:
  - ***<PROCEDURE TYPE="REPAIR"> &warn37;&warn12;...</PROCEDURE>***
- Declarations associate the name with a URI or a “public identifier”

General Entities (2/2)

- Simple
  - ***<!ENTITY % ent "value">***
- External
  - ***<!ENTITY % include-file SYSTEM "http://www.w3.org//">***

Entity Examples

- Common Entity Declarations
  - <!ENTITY lt "<">
  - <!ENTITY gt ">">
  - <!ENTITY amp "&">
  - <!ENTITY apos "'">
  - <!ENTITY quot """>
- Character and Entity Reference
  - Character: <
  - Entity (Declaration above): < >
- Parameter-Entity Reference for order.dtd:
  - Declaration: <!ENTITY % minibar.items "book | cdrom">
  - Usage: <!ELEMENT item (%shop.items;)+>
  - Means: item = (book | cdrom) +

Entity Declaration

- **Entities** – define storage units of an XML-document. They are either parsed or unparsed.
  - Allow for better maintenance
  - Parsed entity – content is text replacement
  - Unparsed entity – a resource whose content may or may not be text (text may be other than XML)
- Very powerful tool for advanced usage, cf. XML Specification for full details.
- Cannot redefine predefined entities.

Parameter Entities

- Declaring
  - ***<!ENTITY % ent "value">***
  - ***<!ENTITY % include-file SYSTEM "http://www.w3.org//">***
- Using
  - ***%include-file;***
  - ***<![option [<!… optional declaration …>]]>***

Character References

- Can be used to obtain untype-able characters
  - Such as Kanji for users with English keyboards
- Map directly to a Unicode code point
- Represented much like entity references:
  - Decimal: ㋱
  - Hex: 뻯
- Schemas do not affect these

Comments (1/2)

<!-- a comment -->

- Contents are ignored by the XML processor
- Cannot come before the XML declaration
- Cannot appear inside an element tag
- May not include double hyphens

Comments (2/2)

- Can go most anywhere
  - (though not inside tags)
- Represented as:
  - `<!-- text of comment -->`
- Have simpler syntax than in SGML/HTML
  - Not `<!-- foo -- -- bar -->`
  - Not `<!-- foo -- >`
- Schemas can contain comments, too

Marked Sections

- Two purposes:
  - Escaping a lot of markup
  - Conditional inclusion
- In XML:
  - Escaping only in the document instance:
    - `<![CDATA[ <P>Hello</P> ]]>`
  - Conditional content only in schemas:
    - `<![IGNORE[ ... ]]>`
    - `<![INCLUDE[ ... ]]>`

Processing Instructions

- Form/example:
  - `<?target-name target-specific-stuff ?>`
  - `<?xmleditor insertionpoint?>`
- Used to insert instructions to processors
  - Not commonly needed
  - No way to escape "?>" inside
  - May declare targets in DTD as Notations
- One special one: to identify XML documents
  - `<?xml version="1.0"?>`

Processing Instructions

- Escape to procedural markup
  - *<!NOTATION my-app SYSTEM "http://my.com/">*
  - *<?my-app does something, anything …. ?>*
- Escape hatch
- Way to add declarations to XML in some cases
- Way to "pickle" application state in a document.

The "XML Declaration" PI

- At top of each XML document:
- ```
  <?XML version="1.0"
     standalone="yes"
     encoding="UTF-8"?>
  ```
- This marks the document as being XML
- "Encoding" can be double-checked
  - You can detect the encoding from the first few bytes, for many common ones (even EBCDIC)
  - MIME types also can signal encoding
  - (watch out if server re-encodes document)

Notations

- Used to name foreign data formats referenced
- Ties a notation name to a URI (presumably pointing to the format's specification)
- Entities can state their data's notation
- Processing instructions can (should) use them as target names
- Declared in the schema
  - ```
    <!NOTATION gif SYSTEM
    "http://specs.com/gif10.html">
    ```
- Can also use PUBLIC

Notations

- Declaring
  - *<!NOTATION blob SYSTEM "application/binary">*
- Using (to declare entity datatypes)
  - *<!ENTITY something SYSTEM http://blob.org/blobel*
    - *NDATA blob>*
- Using an NDATA entity
  - *<!ATTLIST img ref ENTITY #REQUIRED>*
  - *… in instance …*
  - *<img ref="something">*
- Or one can just use URIs and MIME types in software… less validation, more simplicity

Identifiers

- Used in entity declarations to state where the data to be included later can be found
- `<!ENTITY warning SYSTEM "http://www.warnsource.com/w993.xml" >`
- Uses a URI reference
  - Probably will later allow referencing subtrees directly by appending an XPointer
- Accommodates persistent naming schemes under development; but doesn't define one.

The Need For A Better DTD

- DTD in use for:
  - Sharing/Reuse many (!!) grammars
  - Validation by the parser
  - Defaulting of values
- Weaknesses of the concept:
  - DTD has a limited capability for specifying data types
  - DTD requires its own language
  - DTD provides incompatible set of data types with those found in databases
  - Example: DTD do not allow to specify element **day** and **month** of Type *Integer* and within a certain *Range*:
    <day>32</day><month>13</month>

Schema Languages

- 3 Leading contenders (all can win):
- XML Schema
  - Backed by the W3C
  - Very powerful
  - Very large + Complex theory
- Relax/NG
  - Backed by OASIS
  - Based on tree automata
  - Very small
- Schematron
  - Independent effort
  - Validation tool, not complete language

XML Schemas

XML Schemas

- XML Schema Definition Language (XSD)
  - http://www.w3.org/XML/Schema
  - XML Schemas provide a superset of the capabilities found in a DTD
- Motivation:
  - "While XML 1.0 supplies a mechanism, the Document Type Definition (DTD) for declaring constraints on the use of markup, automated processing of XML documents requires more rigorous and comprehensive facilities in this area. Requirements are for constraints on how the component parts of an application fit together, the document structure, attributes, data-typing, and so on."
- Notes:
  - W3C recommends "Schemas" as plural of schema
  - XDR (XML-Data Reduced) was an early attempt by Microsoft to define a Schema Language. XDR has been replaced by XSD

XML Schema Specification

- XML Schema Specification is partitioned into two parts
  - Part 1 specifies a language for defining composite types (called complex types) that describe the content model and attribute inventory of an XML element.
  - Part 2 specifies a set of built-in primitive types and a language for defining new primitive types (called simple types) in terms of existing types.
  - In addition to Parts 1 and 2, there is a primer to the XML Schema language known as Part 0 that provides an excellent overview of XML Schemas.
    - http://www.w3.org/TR/xmlschema-0/

XML Schema in short…

- XML - Meta-language for defining markup
- Schema - Formal specification of grammar for a language (in XML!!!!)
  - As such it inherits all the good "stuff", we know from XML
  - Useful for validation, interchange etc.
- XML Schema - Language for writing specifications

Solution: Namespaces

- **XML-Element written as <*nsname*:element>**
- Help avoid element collision
  - P – Paragraph in HTML
  - P – Person in Address-Book DTD
- Namespace declaration
  - Using the xmlns:*nsname*=*value* attribute
  - URI is recommended for *value*
- Can be an attribute of any element; the scope is inside the element's tags

Namespaces

- Helps to "uniquify" markup names
  - Colon delimiter allowed in names
  - `<cals:table>`
    `<html:table xyz:key="2">`
  - Attributes associate a prefix with a namespace URI
  - `<div xmlns:xhtml=`
    `"http://www.w3.org/1999/xhtml">`
    - Sets default for element and descendants

Namespaces: Declaration

- Declaration scopes in root element
  - <elem xmlns="uri1" *xmlns:ns2="uri2"* ... >
    <ns2:elem />
    </elem>
  - elem defines all namespaces
- Declaration after usage
  - <ns1:elem xmlns:ns1="uri1">
    <ns2:elem xmlns:ns2="uri2"/>
    </ns1:elem>

Things Namespace Almost Do

- Allow arbitrary mixing of DTDs /Schemas
- Provide a "type system" for referents of markup
- Allow automatic processing of foreign markup

Pros and Cons of Namespaces

- You can uniquely label element types in a global way
- You can must change the element name to take advantage of this
- Attempts to re-use large numbers of namespace-qualified elements are often clumsy/redundant
- Detection of a namespace is very easy
- There can only be one namespace for an instance of an element

Things are Confusing about Namespaces

- The URI reference in a namespace is just a string
- The URI reference in a namespace may not exist, it's just a string
- The URI reference in a namespace may exist and contain something irrelevant or unexpected: it's just a string
- Relative URI references in namespaces are well-defined, but don't do what you might expect, because they are just strings…
- Fragment identifiers are allowed in namespace URIs, if you want to use them.

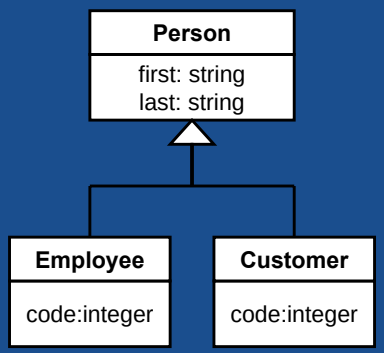
Example 1

- **An employee**

```
<person>
    <first>Peter</first>
    <last>Jones</last>
    <code>4711</code>
<person>
```

- **A customer**

```
<person>
    <first>Steve</first>
    <last>Smith</last>
    <code>0815</code>
<person>
```



Example 2

- **Porter (schemas applied)**

```
<P:person xmlns:P="urn:person">
    <P:first>Peter</P:first>
    <P:last>Jones</P:last>
  <E:employee xmlns:E="urn:employee">
      <E:code >4711</E:code>
  </E:employee>
</P:person>
```

- **Customer (schemas applied)**

```
<P:person xmlns:P="urn:person">
    <P:first>Steve</P:first>
    <P:last>Smith</P:last>
  <C:customer xmlns:C="urn:customer">
    <C:code >0815</C:code>
  </C:customer>
</P:person>
```

Schema Element

- Root Element of a Schema
  - <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
- XML Schema Namespace
  - Each Element in the schema is prefixed by xsd: (xsd is only a convention)
  - Namespace declaration xmlns:xsd="http://www.w3.org/2001/XMLSchema“
  - The same prefix also appears on the names of built-in simple types, e.g. xsd:string.
- Most notably Subelements:
  - element
  - complexType
  - simpleType

Complex Type Definitions (1/3)

- **Complex Types** – Allow child elements and may carry attributes
- Example: Element <USAddress> must consist of five Elements and one Attribute
  - <xsd:complexType name="USAddress" >
  - <xsd:sequence>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="street" type="xsd:string"/>
    <xsd:element name="city" type="xsd:string"/>
    <xsd:element name="state" type="xsd:string"/>
    <xsd:element name="zip" type="xsd:decimal"/>
    </xsd:sequence>
  - <xsd:attribute name="country" type="xsd:NMTOKEN" fixed="US"/>
  - </xsd:complexType>

Complex Type Definitions (2/3)

- Use of Complex Types - Example:
  - <xsd:complexType name="PurchaseOrderType">
  - <xsd:sequence>
    <xsd:element name="shipTo" **type="USAddress"**/>
    <xsd:element name="billTo" **type="USAddress"**/>
    <xsd:element **ref="comment"** minOccurs="0"/> <xsd:element name="items" type="Items"/> </xsd:sequence>
  - <xsd:attribute name="orderDate" type="xsd:date"/>
  - </xsd:complexType>
  - Use of shipTo and billTo Elements in XML requires that these Elements have the five Subelements as defined in USAddress
  - Ref Attribute (here its value is comment) indicates a Reference to elsewhere declared Element (global Element)
  - Comment Element here is optional due to Occurence Constraint

Complex Type Definitions (3/3)

- Element occurrence constraints
  - minOccurs
    - Required if value is 1
  - maxOccurs
    - Bound, e.g. value is 42
    - Unlimited if value is **unbound**
  - Default value for both attributes is 1
- Attributes occurrence constraints
  - Appear once or not at all
  - Constraints by use-attribute with value: required, optional or prohibited
  - Default value by default-attribute
- Example:
  - (minOccurs, maxOccurs) fixed, default = (0, 2) -, 37
    - Element may appear once, twice, or not at all
    - If the Element does not appear it is not provided; if it does appear and it is empty, its Value is 37; otherwise its value is that given

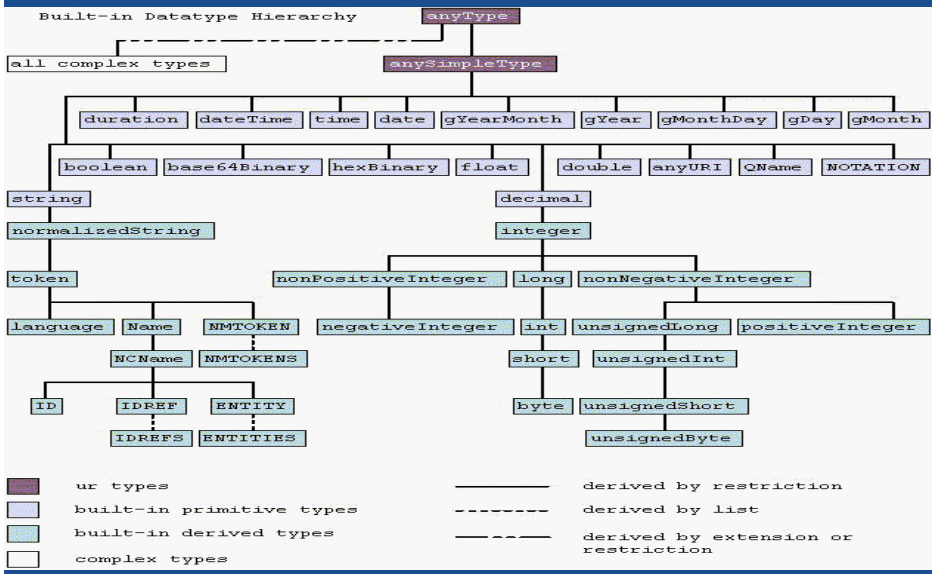
Simple Type Definitions (1/3)

- **Simple Types** – Cannot have element content and cannot carry attributes. XML Schema has more than 40 built in Simple Types, e.g. string, integer, boolean, time, dateTime, date, gMonth, anyURI, language
- Defining new Simple Types is allowed
  - Derive and restrict existing simple type
  - Define by **simpleType** element
  - Use **restriction** sub-element to define **Facets** that constrain the range of values

Simple Type Definitions (2/3)

- Example – Use of Facet called pattern
-
```
<xsd:simpleType name="SKU">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="\d{3}-[A-Z]{2}"/>
  </xsd:restriction>
</xsd:simpleType>
```
  - Derived from string value space
  - Facet: three Digits followed by a Hyphen followed by two upper-case ASCII Letters
- Other Facets available: Range, Enumeration, List, Union

Simple Type Definitions (3/3)



Element Content (1/2)

- Complex Types from Simple Types
  - **Example: <internationalPrice cur="EUR"> 423.46</internationalPrice>**
  -
```
<xsd:element name="internationalPrice">
  <xsd:complexType>
    <xsd:simpleContent>
      <xsd:extension base="xsd:decimal">
        <xsd:attribute name="cur" type="xsd:string"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
</xsd:element>
```

Element Content (2/2)

- Mixed Content
  - **Example:**
    <hello>Dear <name>Bebo White</name>.</hello>
  - <xsd:element name="hello">
    <xsd:complexType mixed="true">
    <xsd:sequence>
    <xsd:element name="name" type="xsd:string"/>
    </xsd:sequence>
    </xsd:complexType>
    </xsd:element>

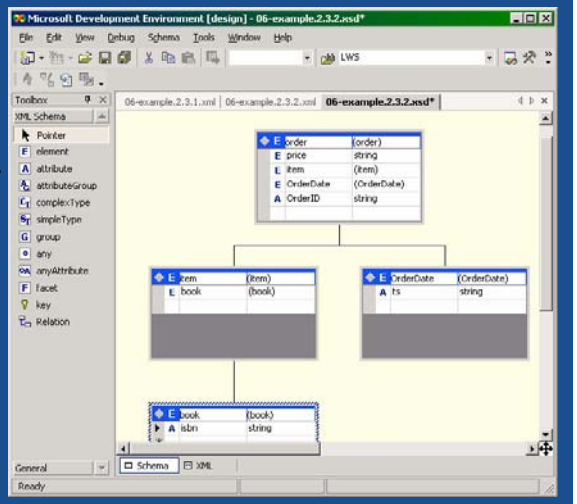
Power of XML Schema

- Defining Complex Types by group elements
  - E.g. sequence, choice, group, all
- Support for maintenance and evolution
  - Target Namespace
  - Schemas in multiple documents (include)
  - Deriving types by extension
  - Abstract elements and types (abstract="true")
  - Keys and references

Developing Schemas

- Use Tools:
- E.g. XMLSpy,
- XMLAuthority,
- Visual Studio.NET

Many other exist



XSL(T)

XSL (1/2)

- Extensible Stylesheet Language (XSL)
- Description of a transformation necessary
- “XSL is a language for expressing stylesheets. Given a class of structured documents or data files in XML, designers use an XSL stylesheet to express their intentions about how that structured content should be presented; that is, how the source content should be styled, laid out and paginated onto some presentation medium such as a window in a Web browser or a set of physical pages in a book, report, pamphlet, or memo.” (http://www.w3.org/TR/WD-xsl/)

XSL (2/2)

- Why Stylesheets?
  - separation of content (XML) from presentation (XSL)
- Why not just CSS for XML?
  - XSL is far more powerful:
    - selecting elements
    - transforming the XML tree
    - content based display (result may depend on actual data values)

Why Transform?

- Convert one schema to another
  - I say Level 1 Heading, you say Chapter
- Rearrange data for formatting
  - Present style languages can’t re-order or copy
    - “see section <xref sid=‘sec37’/>…”
- Project or select document portions

Some Special Transforms

- XML to HTML— for old browsers
- XML to LaT$_E$X—for T$_E$X layout
- XML to SVG—graphs, charts, trees
- XML to tab-delimited—for db/stat packages
- XML to plain-text—occasionally useful
- XML to FO—XSL formatting objects

Document Transformation

- The perspective is tree editing, not syntax
- Basic operations:
  - Changes to node properties
  - Structural rearrangement
  - Several models for this kind of task

XSL Tranformations – XSL(T)

- This specification defines the syntax and semantics of XSL(T), which is a language for transforming XML documents into other XML documents
- XSL specifies the styling of an XML document by using XSL(T) to describe how the document is transformed into another XML document that uses the formatting vocabulary
- A transformation expressed in XSL(T) describes rules for transforming a *Source Tree* into a *Result Tree*
- The transformation is achieved by associating patterns with templates. A pattern is matched against elements in the source tree. A template is instantiated to create a part of the result tree
- http://www.w3.org/TR/xslt

XSL(T) Overview

- XSL stylesheets are denoted in XML syntax
- XSL components:

  1. a language for transforming XML documents (XSL(T): integral part of the XSL specification)
  2. an XML formatting vocabulary (Formatting Objects: >90% of the formatting properties inherited from CSS)

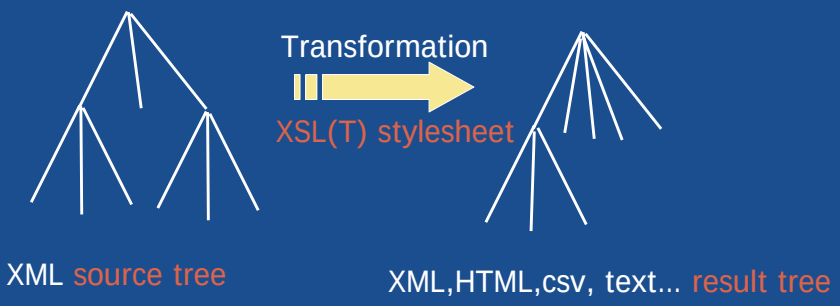
XSL(T) Processing Model (1/3)

- XSL(T) takes
  - A “source” XML document
  - A transform (XSL(T) program)
- XSL(T) applies templates to found nodes
  - (may delete or include the rest)
  - (may process in document or tree or any order)
- XSL(T) generates
  - A “result” XML or text document

XSL(T) Processing Model (2/3)

- XSL stylesheet: collection of template rules
- template rule: (pattern ⇒ template)
- main steps:
  - match pattern against source tree
  - instantiate template (replace current node "." by the template in the result tree)
  - select further nodes for processing
- control can be a mix of
  - recursive processing ("push": <xsl:apply-templates> ...)
  - program-driven ("pull": <xsl:foreach> ...)

XSL(T) Processing Model (3/3)



XSL(T) Elements (1/2)

- <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  - root element of an XSL(T) stylesheet "program"
- <xsl:template match=*pattern* name=*qname* priority=*number* mode=*qname*>
- *...template...*
- </xsl:template>
  - declares a rule: (*pattern* => *template*)
- <xsl:apply-templates select = *node-set-expression* mode = *qname*>
  - apply templates to selected children (default=all)
  - optional mode attribute
- <xsl:call-template name=*qname*>

XSL(T) Elements (2/2)

- Further XSL elements for ...
  - Numbering
    - <**xsl:number value**="position()" **format**="1 ">
  - Conditions
    - <**xsl:if test**="position() mod 2 = 0">
  - Repetition...

XSL(T) Processing Model

- Input in Form of a Tree
  - Recursive process
  - Checks for template when a new item is encountered
  - Transform source nodes into result nodes
  - Rearranges the items based on style sheet

Creating the Result Tree: Repetition

source (xml) | xpath console | stylesheet(xsl) | result | result(html)

```
<xsl:template match="/">
  <html>
    <head>
      <title>customers</title>
    </head>
    <body>
      <table>
    <tbody>
      <xsl:for-each select="customers/customer">
        <tr>
          <th>
        <xsl:apply-templates select="name"/>
          </th>
          <xsl:for-each select="order">
        <td>
          <xsl:apply-templates/>
        </td>
          ...
    </html>
  </xsl:template>
```

Creating the Result Tree: Sorting

source (xml) | xpath console | stylesheet(xsl) | result | result(html)

```
<xsl:template match="employees">
  <ul>
   <xsl:apply-templates select="employee">
            <xsl:sort select="name/last"/>
            <xsl:sort select="name/first"/>
        </xsl:apply-templates>
        </ul>
   </xsl:template>

   <xsl:template match="employee">
    <li>
      <xsl:value-of select="name/first"/>
      <xsl:text> </xsl:text>
      <xsl:value-of select="name/last"/>
    </li>
   </xsl:template>
```

XSL(T) Example

- XML to XML
- Takes one XML document as source tree
- Apply templates using XSL(T) stylesheet
- Transforms it into another XML document as a result tree (here the result tree element are conform to HTML element names;-)

Source Tree

fruit.xml - stylesheet01.xsl.xle - fruit.xml : stylesheet01.xsl

source (xml) | xpath console | stylesheet(xsl) | result | result(html)

```
<?xml version="1.0" ?>
<?xml-stylesheet type="text/xsl" href="blatz.xsl" ?>

<fruit_salad_ingredients>
  <fruit>
    <name>oranges</name>
  </fruit>
  <fruit>
    <name>pineapples</name>
  </fruit>
  <fruit>
    <name>starfruit</name>
  </fruit>
  <fruit>
    <name>watermelon</name>
  </fruit>
</fruit_salad_ingredients>
```

The Boilerplate

```
<xsl:stylesheet version="1.0" xmlns:xsl= "http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/|*|@*|text()">
    <xsl:copy-of>
      <xsl:apply-templates select="@*"/>
     <xsl:apply-templates/>
    </xsl:copy-of>
  </xsl:template>
</xsl:stylesheet>
```

From Copy to Transform

```
<?xml version="1.0"?>
<!-- Rename all p elements to para -->
<xsl:stylesheet xmlns:xsl = "http://www.w3.org/1999/XSL/Transform"
 <xsl:template match=/|*|@*|text()" priority="1">
    <xsl:copy>
      <xsl:apply-templates select="@*"/>
      <xsl:apply-templates/>
    </xsl:copy>
 </xsl:template>
 <xsl:template match="p" priority="2">
    <para>
    <xsl:apply-templates/>
    </para>
 </xsl:template>
</xsl:stylesheet>
```

XSL Style Sheet

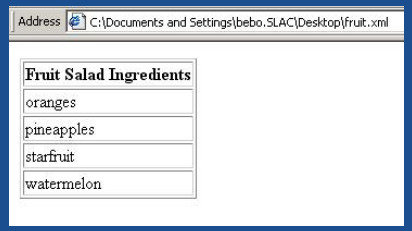
fruit.xml - blatz.xsl.xle - fruit.xml : blatz.xsl

source (xml) | xpath console | stylesheet(xsl) | result | result(html)

```
<?xml version="1.0" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
  <xsl:template match="/" >
    <html>
      <body>
        <table border="1">
          <th>Fruit Salad Ingredients</th>
          <!-- Display the name of each fruit-->
          <xsl:for-each select="/fruit_salad_ingredients/fruit">
            <tr>
              <td><xsl:value-of select="name"/></td>
            </tr>
          </xsl:for-each>
        </table>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

(blatz.xsl)

Results



(View source)

```
<?xml version="1.0" encoding="utf-8" ?>
<?xml-stylesheet type="text/xsl" href="blatz.xsl" ?>
<fruit_salad_ingredients>
    <fruit>
        <name>oranges</name>
    </fruit>
    <fruit>
        <name>pineapples</name>
    </fruit>
    <fruit>
        <name>starfruit</name>
    </fruit>
    <fruit>
        <name>watermelon</name>
    </fruit>
</fruit_salad_ingredients>
```

Models for Tree Editing

- Functional
- Rewrite rule-based
- Template-based
- Imperative

Functional Tree Rewriting

- Recursive processing
- Invoke start function at the root, construct a new tree
- Can think of this as “node functions”
- Result is “compositional” — substitution is generally nested
- Side effects often avoided: caching values, clarity.

Rule-based (Rewriting Systems)

- A transformation is defined by a list of pattern/result pairs
- Each is a piece of a tree with “holes” (variables)
- A match leads to replacement of the matched tree nodes by a result tree
- Variables shared between pattern and result allow preservation and rearrangement of arbitrary data
- Poweful, incremental, definitions; non-deterministic processing

Template-based Processing

- This is a model in which a pattern document is the starting point
- This model is very familiar from many web-based systems.
- It contains literal results interleaved with queries and sometimes imperative code
- Well-suited to repetitive or rigid structures
- Often requires extensions to deal with recursion and looping
- Frequently appropriate for database-style XML

Imperative

- Parser calls imperative code, which uses:
  - Stacks
  - Global variables
  - Explicit output commands
- Result is a side effect.
- Reasoning about the program may be hard, but creating it often starts out easily
- This approach makes it easy to create non-XML, or ill-formed XML documents

What's the Biggest Drawback to Tree Editing?

- Buffering!
  - You need a copy of the tree to edit
  - This means that it's very easy to build transformer for a document entirely in-memory
  - Doing this from secondary storage is fairly subtle, and has its own performance penalties
  - This is a complex speed/size/coding effort tradeoff
- This is one reason imperative approaches are sometimes appealing even to purists.

What Side are We On?

- XSL(T) falls squarely in the middle
- Styles of XSL(T) transform
  - Functional
  - Rule-based
  - Template-based
  - Imperative (although unusual)

XSL(T) and Transformation Styles

- Rule-based substitution (but results are like template languages)
- XPath addressing also looks like queries in traditional template languages
- Limited non-determinism
- Sufficient control over rule evaluation order that functional transformations are easy

Where does XSL(T) Fit?

- Dependencies
  - XML -> XPath -> XSL(T) -> XSL
- The WGs involved
  - XSL Working Group
    +XML Linking for XPath
- Status
  - Full W3C Recommendation, in wide use
  - http://www.w3.org/TR/xsl/

XML Documents as Trees of Nodes

- Root
- Elements
- Attributes
- Text Nodes (not characters)
- Namespaces
- Processing Instructions
- Comments

XML Document Order

- Root -- First
- Elements -- Occur in order of their starts
- Text Nodes -- As if children (leaves)
- Attributes, namespaces -- Attached to element, unordered
- PIs, comments -- Leaves like text nodes

XML Notions

- **XML declaration**: identifies a document as intending to conform to XML rules
- **DTD or schema**: rules for permissible elements and attributes for a genre
- **Well-formedness**: correct XML syntax, but maybe not valid to specified DTD
- **XML name**: token ok as element/attr names
- **Stylesheet PI**: links document to stylesheet

What's Inside an XSL(T) Transform?

- Any number of "templates"
- A template uses Xpath to match nodes
- Highest priority matching template selected
- Then the template takes over and generates:
  - Literal output XML (based on namespace)
  - Computational results (of XSL(T) functions)
  - Results of further template applications
  - Results of queries on the document
- Many options

What Goes in a Template?

- Literal XML to output
- "Pull" references to other content
- Instructions to generate more output
  - Setting and using variables
  - Invoking other templates like macros
  - Manually constructed XML constructs
  - Conditional instructions (if, choose, etc.)
  - Auto-numbering hacks

How Do You Apply One?

- Refer via "Stylesheet PI"
  - Defined in W3C "xml-stylesheet" rec
  - <?xml-stylesheet href="URI"
type=""
title=""
media=""
charset=""
alternate="yes" ?>
- Apply via standalone program
  - E.g. XT, Xalon, Saxon (see Web for latest versions)

Caveats

- Many constructs have extra options
- These are more constructs
- We will not cover all these
- For example:
  - <xsl:stylesheet id="ID"
    extension-element-prefixes="my-Fns"
    enclose-result-prefixes="html"
    version="1.0"
    xml:space="default">

Template Styles

- Push vs. Pull templates
- Or:
  - Fill-in-the-blanks
    - Looks like output document with pulls to merge
  - Navigation
    - Adds top-level <xsl:transform>, macros
  - Rule-based
    - Conceptually, a template for each elemet type
  - Computational
    - Gory processing to generate markup from none

At the Top Level

- Key thing: templates
- Also several option-settings:
  - <xsl:include> -- must be first
  - <xsl:import>
  - <xsl: strip-space> or <xsl:preserve-space>
  - <xsl:output>, <xsl:decimal-format>
  - <xsl:keys>, <xsl:namespace-alias>
  - <xsl:attribute-set>, <xsl:variable>, <xsl:param>
- Most of these are more advanced….

Anatomy of a Template

- XPath to select elements to apply template to
  - (this is where programming/scripting comes in)
- XML to output, for each instance selected
- Embedded within that output:
  - XSL(T) "instruction" elements
  - Literal output (including XML tags)
  - References to content to transclude
  - Place to put results of transforming the element's children (if desired)

Trivial Templates: Tag Renaming

- <xsl:template match="div[@type='idx']">
  <index>
  <xsl:apply-templates/>
  </index>
  </xsl:template>

- <xsl:template match="div1">

  <xsl:process-children/>

  </xsl:template>

Template Options

- Match = "xpath"
  - Which elements to apply template to
- Name = "qname"
  - Name a template for later reference
  - Mode -- (limit template to work in a certain named 'mode' -- more later)
- xml:space = "default|preserve"
  - Override inherited space-handling
- Priority="n" -- for conflicting rules

The Ultimate Default

- Elements are not copied
- Attribute values and text are copied,
- Thus a transform with no templates except for the root, strips markup from a document
  - <xsl:transform>
    <xsl:template match="//">
    </xsl:template>
    </xsl:transform>

Priority Example

- Delete all nested <list>s

```
<xsl:template match="list/list"
   priority="2">
   <!-- deleted nested list -->
</xsl:template>

<xsl:template match="list"
   priority="1">
   <list><xsl:apply-templates></list>
</xsl:template>
```

Template Priority

- Multiple templates may match an element
  - <template priority='3' match='h1'>
    <template priority='5' match="@class='big'">
    <template priority='9' match="h1[@id='S1']">
- Highest priority number wins
- Priorities are integers, including negative
- There are also default rules
  - All have priority -0.5 <= p <= +0.5

What Goes in a Template?

- Literal XML to output
- "Pull" references to other content
- Instructions to generate more output
  - Setting and using variables
  - Invoking other templates like macros
  - Manually constructed XML constructs
  - Conditional instructions (if, choose, etc.)
  - Auto-numbering hacks

Instructions: apply-templates

- <xsl:apply-templates select="xpath"
  mode="qname">
  - Main use (no attributes or content):
    - mark where to include result of processing children
  - select
    - Include certain children:
      - select="[secure='public']"
    - "Pull" (transclude) anything from elsewhere:
      - select="//[id='warning17']"
  - Mode: Apply only templates of this mode

Keeping Things in Variables

- 2 types (names are XML qnames):
  - Variables are assigned once and for all
  - Parameters can be overridden later
- Value types:
  - A template
  - The result of instantiating a template
  - Node-set, string, Boolean, or number
    - An RTF is a restricted type of node-set
- References: $varname

Setting XSL(T) Variables

- Default parameters declared at top level
  - <xsl:**param** name='p' select='s'/>

  or
  - <xsl:**param** name='p'>
    <template>…</template>
    </xsl:param>
- Override via similar xsl:with-param
- <xsl:**with-param** name='p'>
  <template>…</template>
  </xsl:param>

Instructions: call-template

- Invoke a template (like a subroutine)

```
<xsl:call-template name='t'>
  <xsl:with-param name='p'
    select='xpath'>
</xsl:call-template>
```

Using XSL(T) Variables

- Limited processing can be done on RTFs
  - Mainly string processing
- Embed variables via $varname
  - Can do for markup as well as content
  - Can process via functions

Data Handling via Functions

- For strings
- For numbers
- For truth values
- For XML information

String Values

- Anything can be cast to a string
  - Boolean: “true” or “false”
  - Numbers: To decimal
  - Nodes:
    - Root, Elements: character content of all descendants
    - Text nodes: the character content
    - Attributes: the attribute value
    - Comments, PIs: the character content
    - Namespaces: the namespace’s URI

For Strings

- String(object) -- explicit type-cast
- Concat(s1, s2, s3,…) -- concatenate
- Substring(s, offset, length)
  - Substring-after(s,s), Substring-before(s,s)
- Translate(s,from,to)
  - Substitute chars in ‘from’, with ones from ‘to’
- Normalize-space(s) -- delete extra whitespace
- Contains(s1,s2), starts-with(s1,s2)
  - Returns true or false
- String-length(s) -- length in characters

For Numbers and Logic

- Number
- Ceiling, Floor, Round, Sum

- Boolean
- True, False, Not

<xsl:value-of>

- <xsl:value-of select=“expr” disable-output-escaping=“yes|no”>
- Outputs the string value of the selected node(s).
- Any type can be cast to string.

<xsl:copy-of>

- <xsl:copy-of select="expr"/>
  - No content allowed
- Select attribute picks what to copy
  - Using the usual XPath method
- The result is copied
  - A node-set is copied (entire forest of subtrees)
  - An RTF is copied (likewise)
  - Anything else is cast to a string that is copied
- No processing is allowed enroute

<xsl:copy>

```
<xsl:copy use-attribute-sets
    ="qnames">
  <xsl:template>…</xsl:template>
 </xsl:copy>
```

- Generates the start- and end-tags
  - Does not include attributes or children
- May contain <xsl:apply-templates/> etc.

<xsl:if>

```
<xsl:if test="boolean-expr">
    <xsl:template>…
 </xsl:if>
```

- Applies the template only if the expression evaluates to true.
  - These can be nested
  - No 'else' construct
  - See also xsl:choose (=case or switch)
- E.g.: Test="@show='T'"

<xsl:choose>

- Like select/switch/case statement
  - Good for handling enumerated attributes

```
<xsl:choose>
 <xsl:when test="boolean-expr">
   <xsl:template>…
 </xsl:when>
 …
 <xsl:otherwise>
   <xsl:template>…
 </xsl:otherwise>
</xsl:choose>
```

<xsl:for-each>

- <xsl:for-each select="node-set-expr">
- May contain:
  - Xsl:sort -- any number of keys
  - Template
- Applies template to each node found
- <xsl:sort select="string-expr" lang="lg"
  data-type="text|number|qname"
  order="ascending|descending"
  case-order="upper-first|lower-first">

<xsl:apply-imports>

- Affects templates imported via xsl:import that would not otherwise by applied
- Imported templates have lowest priority
- Invoke from within a template

<xsl:variable>

- Declares a variable
  - Variables are scoped to where declared

<xsl:**variable** name="qname" select="expr">
<xsl:template>…

<xsl:message>

- Issues a message to the output
  - terminate='yes|no'
- Message is specified via contained template
  - Thus may include data from source

<xsl:fallback>

- Provides backup for when an instruction fails
  - Contains template to use
- Example:
  - trying to use an unknown extension instruction

<xsl:number>

- Used to generate auto-numbering

```
<xsl:number
    level="single|multiple|any"
    count="pattern" -- which nodes count?
    from="pattern"  -- starting point
    value="number-expr" -- force value
    format="s"      -- (not covering)
    lang="lg"       -- lang to use
    letter-value="alphabetic|traditional"
    grouping-separator="char" -- 1,000
    grouping-size="number"    -- 3 in EN
 />
```

Numbering Example

```
<xsl:template select="list">
   <xsl:element name="toplist">
     <xsl:attribute name="marker">
       <xsl:number level="single"/>
       <!--count defaults to siblings-->
     </xsl:attribute>
   </xsl:element>
 </xsl:template>
```

- 'multiple' -- gathers up sibling numbers of ancestors
- <xsl:number level="multiple" format="1.1.1" count="chap|sec|ssec"/>

Building XML from Parts

- Why?
  - Generate element type name, etc. by expression
  - Content is any template
- <xsl:**element** name="qname" namespace="uri" use-attribute-sets="qnames">
- <xsl:**attribute** name="qname" namespace="uri">
- <xsl:**processing-instruction** name="ncname">
- <xsl:**comment**>
- <xsl:**text** disable-output-escaping="yes">

Oddities of XPath and XSL(T)

- Navigational language for specifying pattern matches
- You specify the tree pattern implicitly by specifying a query for a node where a pattern will be replaced
- This sometimes makes the structure less explicit
- You can invoke further processing on children
- You use template-style access functions rather than pattern variables

Surface Oddities

- The language is a mixture of predicate / query and structural pattern
- Unix path syntax and query syntax syntax make a peculiar mix
- Matching within XSL(T) is always relative to a particular node, so the first few times results can be very puzzling

Strategies for XSL(T)

- Try to pick a single style as much as possible
  - May vary by project
  - Mixing may be necessary but can get confusing
- Be sure you understand (and probably override the default rules)
- Shorter patterns are better
  - <xsl:value-of> and <xsl:if> may be easier to deal with than a complex path

Strategies…

- Use several filters in row
  - It's often easier to manage a series of global changes, than interactions between several complex conditions.
  - Intermediate results make debugging easier
  - Intermediate results may be cacheable
    - Critical for online applications
- Where possible code things one element at a time

More on XSL(T)

- XSL(T):
  - Conflict resolution for multiple applicable rules
  - Modularization <**xsl:include**> <**xsl:import**>
  - …
- XSL Formatting Objects
  - a la CSS
- XPath (navigation syntax + functions)
- = XSL(T) ∩ XPointer
- xslt.com, xml.com

Example

Example 3

```
<xsd:schema id="person" targetNamespace="urn:person“
        xmlns="urn:person“
        xmlns:xsd="http://www.w3.org/2001/XMLSchema“
        attributeFormDefault="qualified“
        elementFormDefault="qualified">
    <xsd:element name="person">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:any namespace=“urn:employee“ />
                <xsd:any namespace="urn:customer" />
                <xsd:element name="first" type="xsd:string“ minOccurs="0" />
                <xsd:element name="last" type="xsd:string" minOccurs="0" />
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

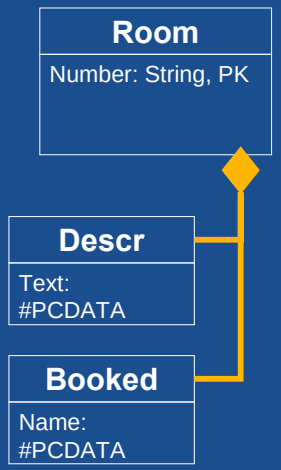
Example 4

```
<xs:schema id="customer"
   targetNamespace="urn:customer“
   xmlns="urn:customer“
   xmlns:xs="http://www.w3.org/2001/XMLSchema“
   attributeFormDefault="qualified"
   elementFormDefault="qualified">
      <xs:element name="customer">
         <xs:complexType>
            <xs:sequence>
               <xs:element name="code" type="xs:int"
   minOccurs="0" />
            </xs:sequence>
         </xs:complexType>
      </xs:element>
   </xs:schema>
```

Example 5: Room Entity (1/3)

- Pay attention to database design
- E.g. containment design does not scale…

```
<room number="R10">
    <descr>A very nice room</descr>
    <booked>Reseller Wonder Travel Corp. </booked>
</room>
<room number="R11">
    <descr>A very nice room</descr>
    <booked>Reseller Wonder  Travel Corp. </booked>
</room>
```



Example 5: Room Entity (2/3)

- Structural Linking

```
<room number="R10">
    <descr href="Nice"/></room>
<room number="R11">
    <descr href="Nice"/></room>

<descr type="Nice">A very nice room</descr>

<booked roomref="R10" by="MrG"/>
<booked roomref="R11" by="MrG"/>
<Guest gid="MrG">
    Reseller Wonder Travel Corp. </Guest>
…
```

Example 5: Room Entity (3/3)

- Logical design
  - DTD or XML-Schema, e.g. **ID** and ***IDREF***

```
<booked roomref="R10" by="MrG"/>
<Guest gid="MrG">…
```

- Shift to Physical Design
  - E.g. XML or Database/SQL
  - If XML other opportunities…

Re*think*ing: The Room Entity

- Physical Design using XML
  - → Adding a semantic support
  - E.g. enhance using Resource Description Framework (RDF)

```
<rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#" xmlns="http://hotel/rdf/syntax#">
   <room rdf:about="http://hotel/room/R10">
       <descr rdf:resource="http://hotel/descr/Nice"/>
   </room>
…
```

XHTML

What is XHTML?

“A reformulation of HTML in XML.”

—W3C **http://www.w3.org/TR/xhtml1/**

The power of XML (kind of)

The simplicity of HTML (mostly)

XHTML is **XML that acts like HTML** in browsers.

From happycog.com

XHTML Introduction

- The Extensible HyperText Markup Language (XHTML™)
  - W3C Recommendation 26 January 2000
  - http://www.w3.org/TR/2000/REC-xhtml1-20000126
- Specification defines XHTML 1.0, a reformulation of HTML 4 as an XML 1.0 application
- Three DTDs corresponding to the ones defined by HTML 4
- Semantics of the elements and their attributes are defined in the W3C Recommendation for HTML 4

Example XHTML Document

- Document Root element **html**
- Referencing xhtml namespace
- Elements and attributes must conform to XML notation rules

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "DTD/xhtml1-
    strict.dtd">
<html
    xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
    <head><title>XHTML Example</title></head>
    <body>
        <p>XHTML is great. </p> <hr/>
        <p>A     <a href="http://webengineering.org/">
                  WebE-Link</a>.</p>
    </body>
</html>
```

Differences To HTML (1/4)

- Documents must be well-formed
  - Incorrect: Overlapping Elements <a>
  - Correct: <a> or <a>
- Element and attribute names must be in lower case
- For non-empty elements, end tags are required
  - Incorrect: A new paragraph starts here
  - Correct: A new paragraph starts here
- Attribute values must always be quoted

Differences To HTML (2/4)

- Attribute minimization
  - Attribute names like compact or checked must be written in full
  - Incorrect: <dl compact>
  - Correct: <dl compact="compact">
- Using ampersands in attribute values
  - '&' must be expressed as a character entity reference
  - Incorrect: http://server/cgi/script?a=guest**&**name=bebo
  - Correct: http://server/cgi/script?a=guest**&**name=bebo

Differences To HTML (3/4)

- Empty Elements
  - Must be XML conform:
 →
  <hr>→<hr/>
- Whitespace handling in attribute values
  - User Agents will strip leading and trailing Whitespace from Attribute Values
- Script and Style elements
  - <script> <![CDATA[... unescaped script content ...]]> </script>
- SGML exclusions
  - SGML gives the Writer of a DTD the Ability to exclude specific Elements from being contained within an Element. Such Prohibitions (called "exclusions") are not possible in XML.
  - For example, the HTML 4 Strict DTD forbids the nesting of an '**a**' element within another '**a**' element to any descendant depth

Differences To HTML (4/4)

- The Elements with 'id' and 'name' Attributes
  - HTML 4 defined the **name** attribute for the elements **a**, **applet**, **form**, **frame**, **iframe**, **img**, and **map**. HTML 4 also introduced the **id** attribute.
  - name and id are attributes designed to be used as fragment identifiers (are of type ID therefore unique).
  - XHTML 1.0 Documents MUST use the **id** Attribute when defining fragment identifiers, even on elements that had a **name** attribute
  - Check compatibility – if necessary provide both: id="foo" name="foo"

Validate XHTML

XHTML validation @ W3C

http://validator.w3.org/

Type in a URL or upload a file to it and their parser will validate your document, looking for errors in your XHTML

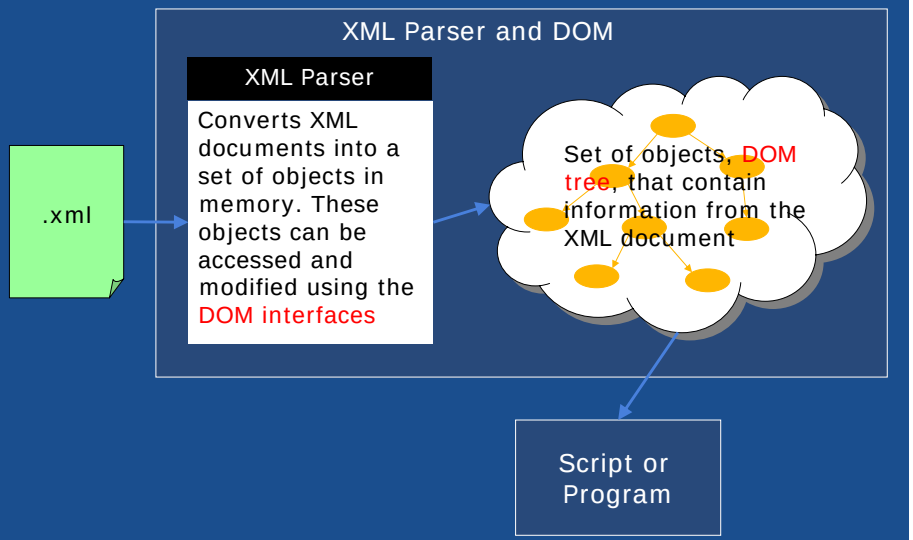
Part 3: XML and Supplementary Technologies

- The W3C Document Object Model (DOM)
  - an API that allows developers to programmatically manage and access XML nodes
  - allows programmers to update and change XML documents within an application
  - reads the whole XML file and then stores a hierarchical tree structure containing all elements within the document
  - This tree has a single root node, which is the root element, and may contain many children, each of which represents an XML element

What is the XML DOM? (1/3)

- XML Document Object Model is a standard way to manipulate (read, modify and make sense of) XML documents
- Formally, the XML DOM is a programming interface (i.e. an API) that can be used in programs for creating an XML document, and/or manipulating an existing XML document (navigating its structure, and adding, modifying, or deleting its elements)
- XML DOM is defined by the W3C (http://www.w3.org/DOM/) to be used with any programming language and any operating system

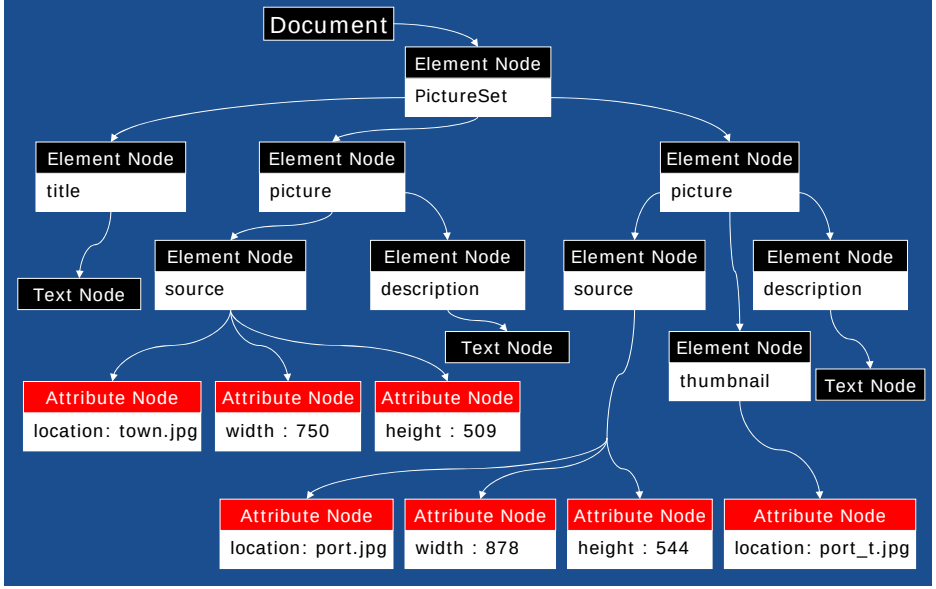
What is the XML DOM? (2/3)



Sample XML Document

```
<?xml version="1.0" encoding="UTF-8"?>
<PictureSet>
  <title>Pictures from My Holiday</title>
  <picture>
     <source location="town.jpg" width="750" height="509"/>
     <description>Seaside town</description>
  </picture>
  <picture>
     <source location="port.jpg" width="878" height="544"/>
     <thumbnail location="port_t.jpg"/>
     <description>Sea port</description>
  </picture>
</PictureSet>
```

The Corresponding DOM Tree



XML DOM Interfaces/Classes

- Document – the root in the DOM tree; provides access to all tree nodes.
- Node – represents a node in the DOM tree
- Nodelist – read-only list of Node objects.
- Element – represents an element node; derives from Node.
- Attr – represents an attribute node; derives from Node.
- CharacterData – represents character data; derives from Node.
- Text – represents a text node; derives from CharacterData.
- Comment – represents a comment node; derives from Character Data.
- Processing Instruction – represents a processing instruction, i.e. <?… ?>; derives from Node.
- CDATASection – represents a CDATA section; derives from Text.

Parsing the DOM

- To read and update - create and manipulate - an XML document, you need an XML parser.
- The Microsoft XMLDOM parser features a programming model that:
  - Supports JavaScript, VBScript, Perl, VB, Java, C++ and more
  - An ActiveX object that comes with Microsoft Internet Explorer 5.0
  - Supports W3C XML 1.0 and XML DOM
  - Supports DTD and validation

Creating XML DOM Tree with JavaScript

```
function CreateDOM(xmlfilename) {
  if (document.implementation &&
      document.implementation.createDocument) {
    xmlDoc= document.implementation.createDocument("","", null);
    xmlDoc.onload = SomeFunction;
  }
  else if (window.ActiveXObject) {
    xmlDoc = new ActiveXObject("Microsoft.XMLDOM");
    xmlDoc.onreadystatechange = checkLoad;
  }
  else {
    alert('Your browser can\'t handle this script'); return;
  }
  xmlDoc.load(xmlfilename);
}

function checkLoad() {
  if (xmlDoc.readyState == 4) SomeFunction();
}
```

Data Access

- Parsing vs. Access
  - Once an XML document is parsed, there are multiple ways to access the data.
- Random vs. sequential access
  - **D**ocument **O**bject **M**odel (random)
  - **S**imple **A**PI for **X**ML (sequential)
- Xpath for node selection

SAX and DOM

- SAX and DOM are standards for XML parsers - program APIs to read and interpret XML files
  - DOM is a W3C standard
  - SAX is an ad-hoc (but very popular) standard
- There are various implementations available
- Java implementations are provided in JAXP (Java API for XML Processing)
- JAXP is included as a package in Java 1.4
  - JAXP is available separately for Java 1.3
- Unlike many XML technologies, SAX and DOM are relatively easy

Difference between SAX and DOM (1/2)

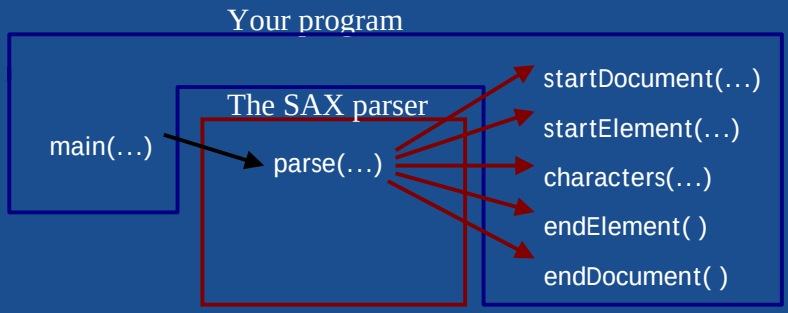
- DOM reads the entire XML document into memory and stores it as a tree data structure
- SAX reads the XML document and sends an event for each element that it encounters

Difference between SAX and DOM (2/2)

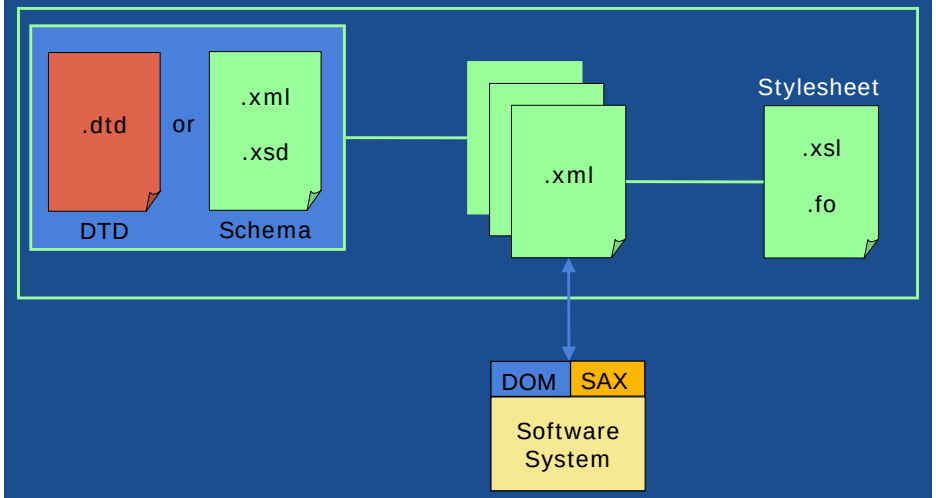
- Consequences:
  - DOM provides “random access” into the XML document
  - SAX provides only sequential access to the XML document
  - DOM is slow and requires huge amounts of memory, so it cannot be used for large XML documents
  - SAX is fast and requires very little memory, so it can be used for huge documents (or large numbers of documents)
    - This makes SAX much more popular for Web sites
  - Some DOM implementations have methods for changing the XML document in memory; SAX implementations do not

SAX Callbacks

- SAX works through callbacks: you call the parser, it calls methods that you supply



XML DOM in the Context of a Web Project



W3C DOM with JavaScript (1/3)

- Example 1: Loading the XML document: DOMDocument
  - The programmer can use a Microsoft Active X object to parse an XML file

```
//Instantiate DOMDocument object
var XMLfile = new ActiveXObject("Msxml2.DOMDocument");
XMLfile.load("newspaper.xml");
var rootElement = XMLfile.documentElement;
document.write("The root node of the XML file is: ");
document.writeln("<b>" + rootElement.nodeName +"</b>");
```

W3C DOM with JavaScript (2/3)

- Example 2: Accessing the Children Elements
  - The *childNodes* member of any element node gives the programmer access to all of the sibling nodes of that element

```
//traverse through each child of the root element
//and print out its name
for (i=0; i<rootElement.childNodes.length; i++) {
    var node = rootElement.childNodes.item(i);
    document.write("The name of the node is ");
    document.write("<b>" + node.nodeName + "</b>");
}
```

W3C DOM with JavaScript (3/3)

- Example 3: Getting Element Attributes

```
//traverse through each child of the root element
//and print out its name
    for (i=0; i<rootElement.childNodes.length; i++) {
        //get the current element
        var elementNode = rootElement.childNodes.item(i);
        document.writeln("Processing Node: " +
                elementNode.nodeName + "<BR>");

        var attributeValue;
        //get an attribute value by specific name
        attributeValue = elementNode.getAttribute("articleID");
        //print it out
        document.writeln("Attribute value: <b>" + attributeValue +
                " </b><br>");
    }
```

Cautions with DOM

- Make sure that the XML file resides in the same directory as the html file with the JavaScript code
- The Attribute node does not appear as the child node of any other node type; it is not considered a child node of an element
- Use caution when outputting raw XML to Internet Explorer.If the programmer uses the *document.writeln* method, IE attempts to interpret the XML tags and jumbles the text. Instead, use an alert box when debugging.

W3C DOM with Cascading Style Sheets (CSS)

Cascading Style Sheets (CSS)

- Style sheets describe how a document is displayed or printed
- Sets properties or rules for an XML element or set of elements
- Similar to setting attributes in HTML
  - <font color="red" size="3">

6 Popular CSS Properties

Foreground colors; background colors and images

- Fonts
  - Font-size property – absolute size (48pt), a relative percentage (200%) or a relative size (xx-small, x-small, small, medium, large, or x-large)
  - Font-family property = typeface. Set to explicit value (Times or Helvetica) or general value (sans-serif)
- Text
  - Word-spacing – control spacing between words
  - Letter-spacing – control spacing between letters
  - Text-decoration – render text underlined, overlined, with a line through it, or even blinking

CSS Properties

- **Boxes**
  - Control borders, padding, and margins around HTML elements
- **Positioning**
  - Fine-grained control of the layout of a Web page
  - X, Y, and Z coordinates can be set absolutely or relative to their default position
- **Classification**
  - How to display: inline, separate block (blockquote or table), list item, or not at all
  - Whitespace and line break display
  - List element display

CSS Syntax

Selector {property: value}

Selector: element/tag you wish to define

Property: attribute you wish to change

Value: value for the property

Example:

body {color: black; background: white}

This style means that all body text will be black with a white body background color

CSS Style Sheets

Three ways to insert a style sheet:

1. **External style sheet** (a separate file)
2. **Internal style sheet** (inside the <head> tag)
3. **Inline style** (inside an HTML element)

External Style Sheet

Within any Web page, reference a separate CSS file using the <link> tag

Example:

<link rel="stylesheet" href="style.css" type="text/css" />

rel="stylesheet" indicates that the link is to a style sheet.

href refers to the file name of the style sheet you want to use.

type specifies the style sheet language (always “text/css” for css)

Benefit: centrally located style rule – one file to update/change

CSS Applied to XHTML Example

```
<div align="center">
<a href="../"><img src="../bighistory.jpg" border="0" alt = “SLAC Web Wizards" width="400" height="150" /></a>
</div>
<hr />
<font size = "+3" color = "#990000"><b>1991</b></font>
<p>Paul Kunz installs a WWW line-mode browser on SCS VM/CMS system. </p>
<br />
<br />
<font size = "+3" color = "#990000"><b>1991</b></font>
<p>Web/SPIRES interface is created.</p>
<br />
<br />
```

XHTML for CSS

```
<div id="header">
<a href="../"><img src="../bighistory.jpg" border="0" alt = “SLAC Web Wizards" width="400" height="150" /></a>
</div>
<h2 class="year">1991</h2>
<p>Paul Kunz installs a WWW line-mode browser on SCS VM/CMS system. </p>
<h2 class="year">1991</h2>
<p>Web/SPIRES interface is created.</p>
```

Style Sheet

```
p {
    margin-bottom: 20px;
}

h2.year {
    font-weight: bold;
    color: #990000;
    font-size: 150%;
}

#header {
    text-align: center;
    border-bottom-style: ridge;
    padding-bottom: 10px;

}
```

RELAX NG

What is RELAX NG? (1/2)

- RELAX NG is a schema language for XML
  - It is an alternative to DTDs and XML Schemas
  - It is based on earlier schema languages, RELAX and TREX
  - It is not a W3C standard, but is an OASIS standard

What is RELAX NG? (2/2)

- OASIS is the Organization for the Advancement of Structured Information Standards
  - ebXML (Enterprise Business XML) is a joint effort of OASIS and UN/CEFACT (United Nations Centre for Trade Facilitation and Electronic Business)
  - OASIS developed the highly popular DocBook DTD for describing books, articles, and technical documents
- RELAX NG has recently been adopted as an ISO/IEC standard

Design Goals

- Simple and easy to learn
- Uses XML syntax
  - But there is also a “concise” (non-XML) syntax
- Supports XML namespaces
- Treats attributes uniformly with elements so far as possible
- Has unrestricted support for unordered content
- Has unrestricted support for mixed content
- Has a solid theoretical basis
- Can make use of a separate datatyping language (such W3C XML Schema Datatypes)

Basics

- XML is a tree
- RELAX NG validates at the level of the basic tree abstraction
- RELAX NG is all about patterns

Syntax

- Has an XML syntax and an equivalent non-XML compact syntax
- The compact syntax is good for humans
- The XML syntax is good for machines
- Translation is simple using available tools

Basic Patterns

- Text
  - text
- Element
  - element *name* { text }
- Attributes
  - attribute *name* { text }

Cardinality

- "?" – Zero or one
  - element *name* { text }?
- "+" – One or more
  - element *name* { text }+
- "*" – Zero or more
  - element *name* { text }*

Composition

- "," – Group
  - Patterns must occur in given order.
  - element *foo* { text } **,** element *bar* { text }
- "&" – Interleave
  - Patterns can occur in any order.
  - element *foo* { text } **&** element *bar* { text }
- "|" – Choice
  - Exactly one of the patterns can occur.
  - element *foo* { text } **|** element *bar* { text }

Named Patterns

- Enable modularization
- Allow a "flattened" schema design
- Pattern recursion is OK

patternName = element *elementName* { text }

Constraining Text Values

- Constants
  - attribute version { "1.2" }
- Enumerations
  - attribute version { "1.2" | "1.3" | "1.4" }
- Exclusion
  - attribute version { token - "1.0" }
- List
  - attribute versions { list { token* } }

Basic Structure (1/2)

- A RELAX NG specification is written in XML, so it obeys all XML rules
  - The RELAX NG specification has one root element
  - The document it describes also has one root element
  - The root element of the specification is element

Basic Structure (2/2)

- If the root element of your document is book, then the RELAX NG specifications begins:
  - <element name="book"

    xmlns="http://relaxng.org/ns/structure/1.0">
- and ends:
  - </element>

Data Elements (1/2)

- RELAX NG makes a clear separation between:
  - the *structure* of a document (which it describes)
  - the *datatypes* used in the document (which it gets from somewhere else, such as from XML Schemas)
- For starters, we will use the two (XML-defined) elements:
  - <text> … </text> (usually written <text/>)
    - Plain character data, not containing other elements
  - <empty></empty> (usually written <empty/>)
    - Does not contain anything

Data Elements (1/2)

- Other datatypes, such as <double>…</double> are not defined in RELAX NG
  - To inherit datatypes from XML Schemas, use: datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes"
    as an attribute of the root element

Defining Tags

- To define a tag (and specify its content), use
  ```
  <element name="myElement">
      <!-- Content goes here -->
  </element>
  ```
- Example: The DTD
  ```
  <!ELEMENT name (firstName, lastName)>
  <!ELEMENT firstName (#PCDATA)>
  <!ELEMENT lastName (#PCDATA)>
  ```
- Translates to:
  ```
  <element name="name">
      <element name="firstName"> <text/>
  </element>          <element name="lastName">
  <text/> </element>
  </element>
  ```
- Note: As in the DTD, the components must occur *in order*

RELAX NG Describes Patterns

- Your RELAX NG document specifies a *pattern* that matches your valid XML documents
- For example, the pattern:
  ```
  <element name="name">
      <element name="firstName"> <text/>
    </element>
      <element name="lastName"> <text/> </element>
    </element>
  ```
- Will match the XML:
  ```
  <name>
      <firstName>Bebo</firstName>
      <lastName>White</lastName>
    </name>
  ```

Easy Tags

<zeroOrMore> … </zeroOrMore>

The enclosed content occurs zero or more times

<oneOrMore> … </oneOrMore>

The enclosed content occurs one or more times

<optional> … </optional>

The enclosed content occurs once or not at all

<choice> … </choice>

Any one of the enclosed elements may occur

<!-- An XML comment - not a container, and may not contain two consecutive hyphens -->

Example

```
<element name="addressList">
  <zeroOrMore>
    <element name="name">
      <element name="firstName"> <text/> </element>
      <element name="lastName"> <text/> </element>
    </element>
    <element name="address">
      <choice>
        <element name="email> <text/> </element>
        <element name="USPost"> <text/> </element>
      </choice>
    </element>
  </zeroOrMore>
</element>
```

Enumerations

- The <value>...</value> pattern matches a specified value
  - Example:
    ```
    <element name="gender">
       <choice>
          <value>male</value>
          <value>female</value>
       </choice>
    </element>
    ```
- The contents of <value> are subject to whitespace normalization:
  - Leading and trailing whitespace is removed
  - Internal sequences of whitespace characters are collapsed to a single blank

More About Data (1/2)

- Remember: To inherit datatypes from XML Schemas, add this attribute to the root element:
  datatypeLibrary =
  "http://www.w3.org/2001/XMLSchema-datatypes"
- You can access the inherited types with the <data> tag, for instance, <data type="double>
  - The <data> pattern must match the *entire* content of the enclosing tag, not just part of it
  - ```
    <element name="illegalUse">  <!-- Don't do this! -->
       <data type="double"/>
       <element name="moreStuff"> <text/> </element>
    </element>
    ```

More About Data (2/2)

- If you don't specify a datatype library, RELAX NG defines the following for you (along with <text/> and <empty/>):
  - <string/> : *No* whitespace normalization is done
  - <token/> : A sequence of characters containing *no* whitespace

<group>

- <group>…</group> is used as “fat parentheses”
- Example:

```
<choice>
      <element name="name"> <text/> <element>
      <group>
         <element name="firstName">
            <text/>
         </element>
         <element name="lastName">
            <text/>
         </element>
      </group>
   </choice>
```
```

Attributes

- Attributes are defined practically the same way as elements:
  - <attribute name="***attributeName***">...</attribute>
- Example:
  - <element name="name">
    <attribute name="title"> <text/> </attribute>
    <element name="firstName"> <text/> </element>
    <element name="lastName"> <text/> </element>
    </element>
- Matches:
  - <name title="Prof.">
    <firstName>Bebo</firstName>
    <lastName>White</lastName>
    </name>

More About Attributes

- With attributes, as with elements, you can use <optional>, <choice>, and <group>
- It doesn't make sense to use <oneOrMore> or <zeroOrMore> with attributes
- In keeping with the usual XML rules,
  - The order in which you list elements is significant
  - The order in which you list attributes is *not* significant

Still More About Attributes

- <attribute name="***attributeName***"> <text/> </attribute>
  can be (and usually is) abbreviated as
  <attribute name="***attributeName***"/>
- However,
  <element name="***elementName***"> <text/> </element>
  can *not* be abbreviated as
  <element name="***elementName***"/>
  - If an element has no attributes *and* no content, you must use <empty/> explicitly

<list>

- <list> ***pattern*** </list> matches a whitespace-separated list of tokens, and applies the ***pattern*** to those tokens
  - Example:

```
<!-- A floating-point number and some
integers -->
<element name="vector">
  <list>
    <data type="float"/>
    <oneOrMore>
      <data type="int"/>
    </oneOrMore>
  </list>
</element>
```

<interleave>

- <interleave> ... </interleave> allows the contained elements to occur in any order
- <interleave> is more sophisticated than you might expect
  - If a contained element can occur more than once, the various instances do not need to occur together

Interleave Example

```
<element name="contactInformation">
   <interleave>
      <zeroOrMore>
         <element name="phone"> <text/> </element>
      </zeroOrMore>
      <oneOrMore>
         <element name="email"> <text/> </element>
      </oneOrMore>
   </interleave>
</element>

<contactInformation>
    <email>bebo@slac.stanford.edu</email>
    <phone>650-926-2907</phone>
    <email>bebo.white@gmail.com</email>
</contactInformation>
```

<mixed>

- <mixed> allows mixed content, that is, both text and patterns
- If ***pattern*** is a RELAX NG pattern, then
  <mixed> ***pattern*** </mixed>
  is shorthand for
  <interleave> <text/> ***pattern*** </interleave>

Example of <mixed>

- Pattern:

```
<element name="words">
      <mixed>
         <zeroOrMore>
            <choice>
               <element name="bold"> <text/> </element>
               <element name="italic"> <text/> </element>
            </choice>
         </zeroOrMore>
      </mixed>
   </element>
```

Without this we get *one* bold or *one* italic

- Matches:

```
<words>This is <italic>not</italic> a <bold>great</bold>
    example, <italic>but</italic> it should suffice.</words>
```

The Need for Named Patterns

- So far, we have defined elements exactly at the point that they can be used
  - There is no equivalent of:
    - <!ELEMENT person (name)>
      <!ELEMENT name (firstName, lastName)>
      ...use* person *several places in the DTD...
  - With the RELAX NG we have discussed so far, each time we want to include a person, we would need to explicitly define both person and name at that point:
    - <element name="person">
      <element name="firstName"> <text/> </element>
      <element name="lastName"> <text/> </element>
      </element>
- The <grammar> element solves this problem

Syntax of <grammar>

<grammar xmlns="http://relaxng.org/ns/structure/1.0">
  <start>
    ...usual RELAX NG elements, which may include:
    <ref name="***DefinedName***"/>
  </start>

  <!-- One or more of the following: -->
  <define name="***DefinedName***">
    ...usual RELAX NG elements, attributes, groups, etc.
  </define>
</grammar>

Use of <grammar>

- To write a <grammar>,
  - Make <grammar> the root element of your specification
    - Hence it should say xmlns="http://relaxng.org/ns/structure/1.0"
  - Use, as the <start> element, a pattern that matches the entire (valid) XML document
  - In each <define> element, write a pattern that you want to use other places in the specification
  - Wherever you want to use a defined element, put <ref name="***NameOfDefinedElement***">
  - Note that defined elements may be used in definitions, not just in the <start> element
    - Definitions may even be recursive, but
    - Recursive references must be in an element, not an attribute

Long Example of <grammar>

- <!ELEMENT name (firstName, lastName)>

- <grammar xmlns="http://relaxng.org/ns/structure/1.0">
  <start>
  <ref name="Name"/>
  </start>

  <define name="Name">
  <element name="name">
  <element name="firstName"> <text/> </element>
  <element name="lastName">
  <ref name="LastName">
  </element>
  </element>
  </define>

  <define name="LastName">
  <element name="lastName"> <text/> </element>
  </define>
  </grammar>

XML is case sensitive--
Note that defined terms are capitalized differently

Common Usage 1

- A typical way to use RELAX NG is to use a <grammar> with just the root element in <start> and every element described by a <define>
- <grammar xmlns="http://relaxng.org/ns/structure/1.0">
  <start>
  <ref name="NOVEL">
  </start>

  <define name="NOVEL">
  <element name="novel">
  <ref name="TITLE"/>
  <ref name="AUTHOR"/>
  <oneOrMore>
  <ref name="CHAPTER"/>
  </oneOrMore>
  </element>
  </define>

  ...more...

Common Usage 2

```
<define
name="TITLE">
   <element
name="title">
      <text/>
   </element>
</define>

<define
name="AUTHOR">
   <element
name="author">
      <text/>
   </element>
</define>
```

```
<define name="CHAPTER">
   <element name="chapter">
      <oneOrMore>
         <ref
name="PARAGRAPH"/>
      </oneOrMore>
   </element>
</define>

<define name="PARAGRAPH">
      <element name="paragraph">
         <text/>
      </element>
   </define>

</grammar>
```

Replacing DTDs

- With <grammar> and multiple <define>s, we can do essentially the same things as a DTD
  - Advantages:
    - RELAX NG is more expressive than a DTD; we can interleave elements, specify data types, allow specific data values, use namespaces, and control the mixing of data and patterns
    - RELAX NG is written in XML
    - RELAX NG is relatively easy to understand
  - Disadvantages
    - RELAX NG is *extremely* verbose
      - But there is a “compact syntax” that is much shorter
    - RELAX NG is not (yet) nearly as well known
      - Hence there are fewer tools to work with it
      - This situation seems to be changing

Datatype Libraries

- Complicated text constraints are expressed using external type libraries
- External datatypes can be specialized using parameters
- Two widely used general type libraries are DTD and W3C XML Schema

W3C XML Schema Type Library

- Strings
- Numerics
- Dates and times
- URIs
- XML qualified names
- Binary encodings

Modularity Support

- Inclusion of external schemas
  - Reusable pattern libraries
- Reuse of external patterns
- Overriding external patterns
- Combining patterns

W3C XML Schema Comparison

- RELAX NG patterns provide the functionality of several XML Schema features
- RELAX NG doesn't have the XML Schema determinism constraints
- RELAX NG can define several namespaces with a single schema
- XML Schema has wider vendor support
- RELAX NG doesn't enforce identity constraints

W3C XML Schema Benefits

- Can do other things than validation
  - Data type assignment
  - Type hierarchy modeling
  - Automatic object mapping
- Good third party tool support
- Built in identity constraint enforcement

RELAX NG Benefits

- Simple and elegant schema language
- Good at handling complex unordered and mixed content vocabularies
- Can handle ambiguous/nondeterministic vocabularies
- Strong basis in theory

Use in the Real World

- XHTML
- OpenOffice
- DocBook
- RDF
- WSDL
- Possibly SVG

RELAX NG Tools (1/2)

- Jing
  - An open source validator written in Java
- Sun's MSV
  - Another validator
- DTDinst
  - Translates from DTDs into RNG syntax or RNG "compact" syntax

RELAX NG Tools (2/2)

- Trang
  - Translates RNG compact syntax into RNG syntax
  - Translates RNG or RNG compact syntax into DTDs
- Sun's RELAX NG Converter
  - Translates DTDs into RNG syntax (but not well)
  - Translates an XML Schema subset into RNG syntax (imperfectly)

Schematron

What is Schematron?

- A small schema language
- Helps both RNG and XSD
- Uses Xpath based validation as opposed to grammars

Schematron vs. XML Schema

- Context dependent validation
- Algorithmic validation.

What Does It Look Like?

- Two main constructs:
  - assert
  - report

```
<sch:schema
xmlns:sch="http://www.ascc.net/xml/schematron">
<sch:pattern name="genericID">
  <sch:rule context="*[@ID]">
      <sch:report test=”xpath that matches things which
should not be there"> The error message if something that
should not be there is there</sch:report>
<sch:assert test=”xpath that says what should be
there">The error message if something which should be
there, isn’t there</sch:assert>
</sch:rule>
</sch:pattern>
</sch:schema>
```

```
<sch:schema
xmlns:sch="http://www.ascc.net/xml/schematron">
<sch:pattern name="genericID">
  <sch:rule context="*[@ID]">
      <sch:report test=”number(@ID) != @ID"> Our ID’s
should be numbers</sch:report>
<sch:assert test=”@ID = (count(preceding::*[@ID]/@ID) +
1)">The ID should be equal to the count of the number of
elements with ID attributes.</sch:assert>
</sch:rule>
</sch:pattern>
</sch:schema>
```

With Namespaces

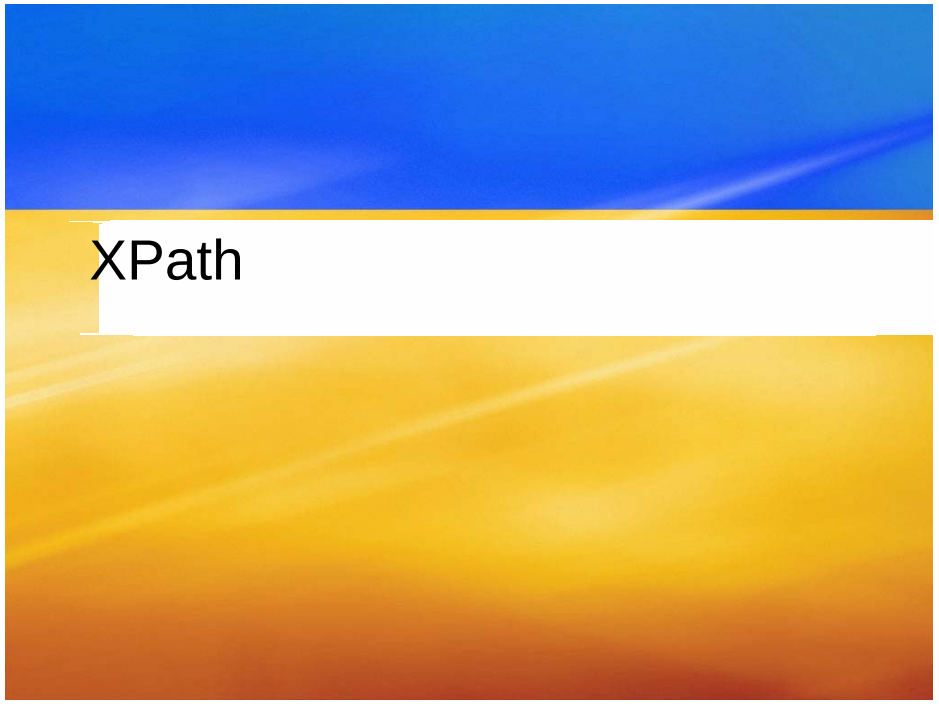
```
<sch:schema xmlns:sch=http://www.ascc.net/xml/schematron
xmlns:n=”namespace”>
<sch:ns uri=”namespace" prefix=”n"/>
<sch:pattern name="genericID">
  <sch:rule context=”n:p[@ID]">
<sch:assert test=”@ID = (count(preceding::n:p[@ID]/@ID) +
1)">The ID should be equal to the count of the number of
elements with ID attributes.</sch:assert>
</sch:rule>
</sch:pattern>
</sch:schema>
```

Equivalent to XML Schema, RNG

```
<sch:pattern name=”p-rules">
  <sch:rule context=”n:p">
<sch:assert test=”@ID">An id is needed on a p
    element.</sch:assert>
</sch:rule>
</sch:pattern>
```

Algorithmic Checks

```
<sch:pattern name=”simplealgorithm">
  <sch:rule context=”n:p[@ID]">
<sch:assert test=”(number(substring(@ID,0,5)) -
    number(substring(@ID,6,5)) ) =
    number(substring(@ID,12,1)) ">The first 5
    numbers of the id minus the next 5 numbers
    must equal the 12th number.</sch:assert>
</sch:rule>
</sch:pattern>
```



What is XPath?

- XPath is a syntax used for selecting parts of an XML document
- The way XPath describes paths to elements is similar to the way an operating system describes paths to files
- XPath is almost a small programming language; it has functions, tests, and expressions
- XPath is a W3C standard
- XPath is not itself written as XML, but is used heavily in XSL(T)

Terminology

```
<library>
  <book>

    <chapter>
    </chapter>

    <chapter>
      <section>

<paragraph/>

<paragraph/>
      </section>
    </chapter>

  </book>
</library>
```

- library is the parent of book; book is the parent of the two chapters
- The two chapters are the children of book, and the section is the child of the second chapter
- The two chapters of the book are siblings (they have the same parent)
- library, book, and the second chapter are the ancestors of the section
- The two chapters, the section, and the two paragraphs are the descendents of the book

Paths

Operating system:	XPath:
/ = the root directory	/library = the root element (if named library)
/users/bebo/foo = the (one) file named foo in bebo in users	/library/book/chapter/section = *every* section element in a chapter in every book in the library
foo = the (one) file named foo in the current directory	section = *every* section element that is a child of the current element
. = the current directory	. = the current element
.. = the parent directory	.. = parent of the current element
/users/bebo/* = all the files in /users/bebo	/library/book/chapter/* = all the elements in /library/book/chapter

Slashes (1/2)

- A path that begins with a / represents an absolute path, starting from the top of the document
  - Example: /email/message/header/from
  - Note that even an absolute path can select *more than one* element
  - A slash by itself means "the whole document"
- A path that does *not* begin with a / represents a path starting from the current element
  - Example: header/from

Slashes (2/2)

- A path that begins with // can start from *anywhere* in the document
  - Example: //header/from selects every element from that is a child of an element header
  - This can be expensive, since it involves searching the entire document

Brackets and last() (1/2)

- A number in brackets selects a particular matching child (counting starts from 1, except in Internet Explorer)
  - Example: /library/book[1] selects the first book of the library
  - Example: //chapter/section[2] selects the second section of every chapter in the XML document
  - Example: //book/chapter[1]/section[2]
  - Only *matching* elements are counted; for example, if a book has both sections and exercises, the latter are ignored when counting sections

Brackets and last() (1/2)

- The function last() in brackets selects the last matching child
  - Example: /library/book/chapter[last()]
- You can even do simple arithmetic
  - Example: /library/book/chapter[last()-1]

Stars

- A star, or asterisk, is a “wild card”—it means “all the elements at this level”
  - Example: /library/book/chapter/* selects every child of every chapter of every book in the library
  - Example: //book/* selects every child of every book (chapters, tableOfContents, index, etc.)
  - Example: /*/*/*/paragraph selects every paragraph that has exactly three ancestors
  - Example: //* selects every element in the entire document

Attributes (1/2)

- You can select attributes by themselves, or elements that have certain attributes
  - Remember: an attribute consists of a name-value pair, for example in <chapter num="5">, the attribute is named num
  - To choose the attribute itself, prefix the name with @
  - Example: @num will choose every *attribute* named num
  - Example: //@* will choose *every attribute, everywhere* in the document
- To choose *elements* that have a given attribute, put the attribute name in square brackets
  - Example: //chapter[@num] will select every chapter element (anywhere in the document) that has an attribute named num

Attributes (2/2)

- //chapter[@num] selects every chapter element with an attribute num
- //chapter[not(@num)] selects every chapter element that does *not* have a num attribute
- //chapter[@*] selects every chapter element that has *any* attribute
- //chapter[not(@*)] selects every chapter element with *no* attributes

Values of Attributes

- //chapter[@num='3'] selects every chapter element with an attribute num with value 3
- The normalize-space() function can be used to remove leading and trailing spaces from a value before comparison
  - Example: //chapter[normalize-space(@num)="3"]

Axes (1/2)

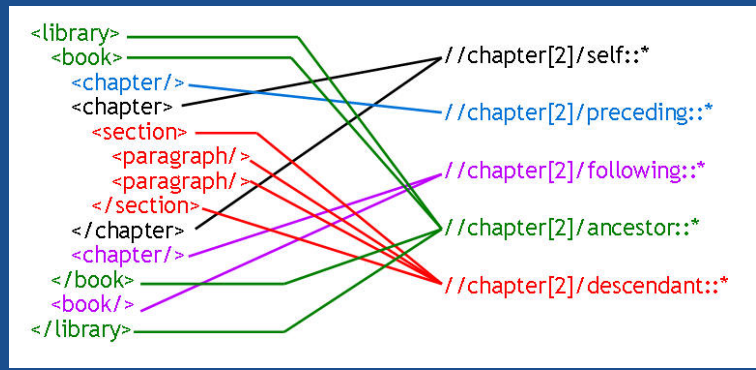
- An axis (plural axes) is a set of nodes relative to a given node; X::Y means "choose Y from the X axis"
  - self:: is the set of current nodes (not too useful)
    - self::node() is the current node
  - child:: is the default, so /child::X is the same as /X
  - parent:: is the parent of the current node
  - ancestor:: is all ancestors of the current node, up to and including the root

Axes (2/2)

- descendant:: is all descendants of the current node
  (Note: never contains attribute or namespace nodes)
- preceding:: is everything before the current node in the entire XML document
- following:: is everything after the current node in the entire XML document

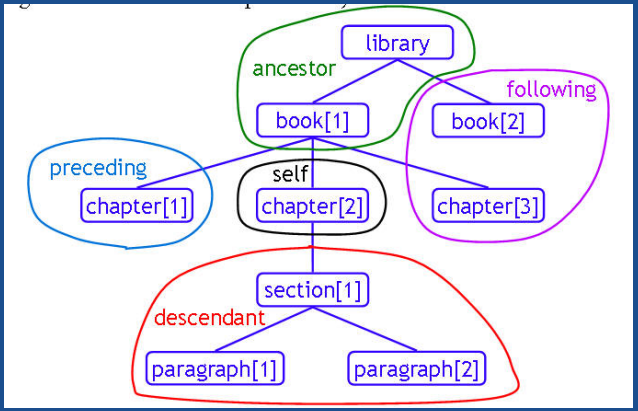
Axes (outline view)

Starting from a given node, the self, preceding, following, ancestor, and descendant axes form a partition of all the nodes (if we ignore attribute and namespace nodes)



Axes (tree view)

- Starting from a given node, the self, ancestor, descendant , preceding, and following axes form a *partition* of all the nodes (if we ignore attribute and namespace nodes)



Axis Examples

- //book/descendant::* is all descendants of every book
- //book/descendant::section is all section descendants of every book
- //parent::* is every element that is a parent, i.e., is not a leaf
- //section/parent::* is every parent of a section element
- //parent::chapter is every chapter that is a parent, i.e., has children
- /library/book[3]/following::* is everything after the third book in the library

More Axes

- ancestor-or-self:: ancestors plus the current node
- descendant-or-self:: descendants plus the current node
- attribute:: is all attributes of the current node
- namespace:: is all namespace nodes of the current node
- preceding:: is everything before the current node in the entire XML document
- following-sibling:: is all siblings after the current node

- Note: preceding-sibling:: and following-sibling:: do not apply to attribute nodes or namespace nodes

Abbreviations for Axes

(none) is the same as child::

@ is the same as attribute::

. is the same as self::node()

.//X is the same as self::node()/descendant-or-self::node()/child::X

.. is the same as parent::node()

../X is the same as parent::node()/child::X

// is the same as /descendant-or-self::node()/

//X is the same as /descendant-or-self::node()/child::X

Arithmetic Expressions

+	add
-	subtract
*	multiply
div	(not /) divide
mod	modulo (remainder)

Equality Tests

- = means “equal to” (Notice it’s *not* ==)
- != means “not equal to”
- But it’s not that simple!
  - ***value*** = ***node-set*** will be true if the ***node-set*** contains ***any node*** with a value that matches ***value***
  - ***value*** != ***node-set*** will be true if the ***node-set*** contains ***any*** node with a value that does ***not*** match *value*
- Hence,
  - ***value*** = ***node-set*** and ***value*** != ***node-set*** may both be true at the same time!

For XML Information

- Id(object)
  - If arg is a node-set, each node is cast to string
    - E.g. context of //footnote/attr(‘ref’) gets ref attributes
  - Else arg is cast to a string
  - Filters the context by picking node w/ ids in list
    - Many space-separated Ids may be included
- Lang

For Looking Around the Context

- Count(node-set)
  - Returns number of nodes in the argument
- Last()
  - Returns number of nodes in the context
- Position()
  - Returns the position of the current node in the context

For Names and Namespaces

- Local-name(node-set?)
  - Returns local part of the name of the first node
- Name(node-set?)
  - Returns entire qualified name of the first node
- Namespace-uri(node-set?)
  - Returns the uri identifying the namespace of the first node

Other Boolean Operators

- and (infix operator)
- or (infix operator)
  - Example: count = 0 or count = 1
- not() (function)

- The following are used for *numerical* comparisons only:
- < "less than" Some places may require <
- <= "less than or equal to" Some places may require <=
- > "greater than" Some places may require >
- >= "greater than or equal to" Some places may require >=

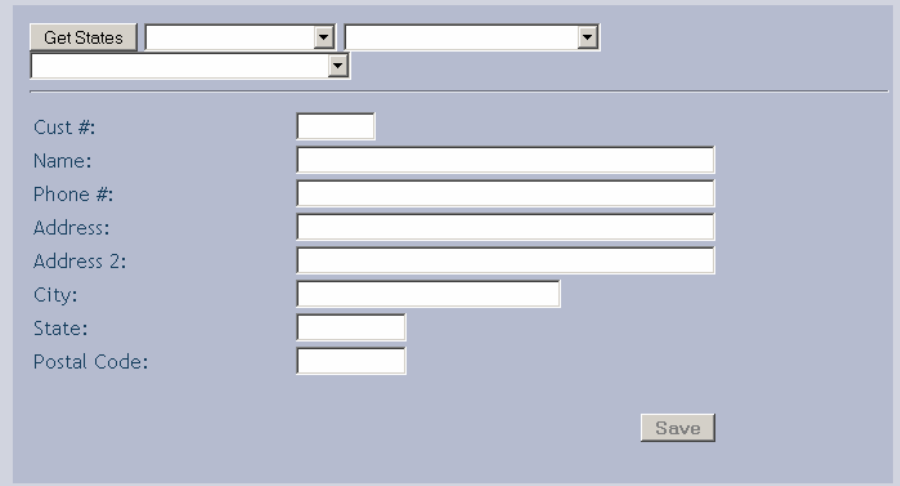
Some XPath Functions

- XPath contains a number of functions on node sets, numbers, and strings; here are a few of them:
  - count(***elem***) counts the number of selected elements
    - Example: //chapter[count(section)=1] selects chapters with exactly two section children
  - name() returns the name of the element
    - Example: //*[name()='section'] is the same as //section
  - starts-with(***arg1***, ***arg2***) tests if ***arg1*** starts with ***arg2***
    - Example: //*[starts-with(name(), 'sec']
  - contains(***arg1***, ***arg2***) tests if ***arg1*** contains ***arg2***
    - Example: //*[contains(name(), 'ect']

What is AJAX?

- **A**synchronous **J**avaScript **a**nd **X**ML (or, **A**synchronous **Ja**vaScript and **X**ML)
- Allows for the creation of fat-client web applications
- Also known as: XMLHTTP, Remote Scripting, XMLHttpRequest, etc.)

AJAX Example (1/2)



AJAX Example (2/2)

- An AJAX call is made to the server to get the states and populate the select box (drop down). The server dynamically creates JavaScript that is executed through an *eval* statement. The final line of the JavaScript calls the *onchange* event of the state drop down which then retrieves all the cities for that state.
- The city drop down is populated and its *onchange* event retrieves all the customers for a given city.
- When the customer changes, the server returns JavaScript that replaces all customer fields with the appropriate values.

AJAX is Not New

- Active use for at least six years
- Was only available in IE (since IE5 public preview in 1999) until about three years ago, in Mozilla (versions just before 1.0)
- Primarily referred to as ‘XMLHTTP’

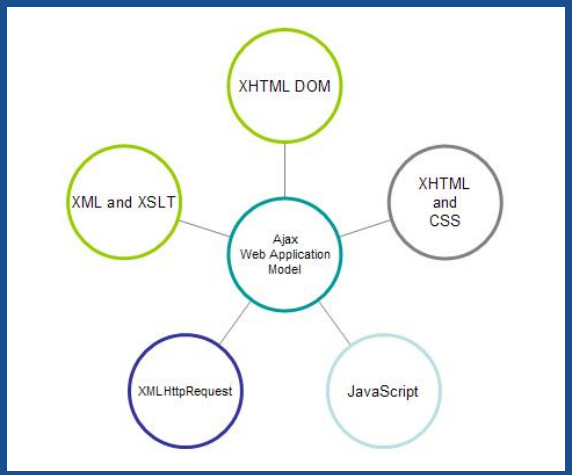
Traditional vs. AJAX

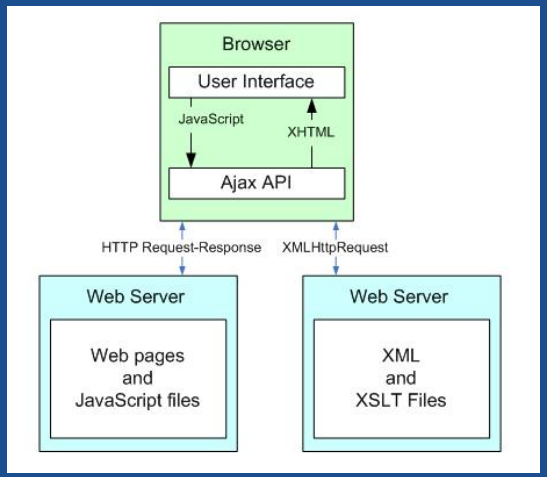
- Interface construction is mainly the responsibility of the server
- User interaction via form submissions
- An entire page is required for each interaction (bandwidth)
- Application is unavailable while an interaction is processing (application speed)

- Interface is manipulated by client-side JavaScript manipulations of the Document Object Model (DOM)
- User interaction via HTTP requests 'behind the scenes'
- Communication can be restricted to data only
- Application is always responsive

XMLHTTP

- An interface that allows for the HTTP communication without a page refresh
- In IE, it is named XMLHTTP and available as an ActiveX Object
- Mozilla and others then modeled a native object called XMLHttpRequest after IE's ActiveX Object
- (The 'others' are Safari 1.2+, Opera 8+, and Konqueror)





XMLHTTP Methods

abort()	Aborts a request.
open(method, uri, [async, [username, [password]]])	Sets properties of request (does not send). (Note about async.)
send(content)	Sends the request with content in the body. content should be null unless method is 'post'.

XMLHTTP Properties

onreadystatechange	Function object to handle request progress.
readyState	Read only. Current request progress.
responseText	Read only. response body as string.
responseXML	Read only. Response body parsed as text/xml and in DOM Document object
status	Read only. HTTP response status code.

Instantiating XMLHTTP

```
function getXMLHTTP() {
   var req = false;
   if(window.XMLHttpRequest) {
      try {
         req = new XMLHttpRequest();
      }
      catch(e) {}
   }
   else if(window.ActiveXObject) {
      try {
         req = new ActiveXObject("Msxml2.XMLHTTP");
      }
      catch(e) {
         try {
            req = new ActiveXObject("Microsoft.XMLHTTP");
         }
         catch(e) {}
      }
   } // end if
   return req;
} // end function

var request = getXMLHTTP();

request.onreadystatechange = handleReadyStateChange;

request.open('get','/bar/checkuname.php?u=bebo');

request.send(null);
```

Handling Responses

```
function handleReadyStateChange() {
   if (!request) return;
   // ignore unless complete readyState
   if (request.readyState == 4) {
      // Ignore unless successful.
      if (request.status == 200) {
         // act on the response
         var xmlbody = request.responseXML;
         // the line below is important!
         request = false;
         processResponse(xmlbody);
      }
   }
} // end function
```

Example Response XML

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<usernameresult>
   <username value="bebo" available="false" />
   <usernamealts>
      <username value="bibo" available="true" />
      <username value="bebow" available="true" />
      <username value="beabeaux" available="true" />
   </usernamealts>
</usernameresult>
```

- Requested username is unavailable, but the script has determined some alternatives

Acting on a Response

- At this point the JavaScript *alert()* function could be used but that would defeat one of the reasons to use AJAX - allowing the application to always remain responsive.
- A better choice is to manipulate the DOM with JavaScript

Altered DOM After Manipulation

- …
- parentNode
  - ul
    - li
      - Username bebo is not available
      - Username bibo is available
      - Username bebow is available
      - Username beabeaux is available
  - maindiv
- …

AJAX Libraries

- SAJAX
  - Makes things marginally easier
  - Doesn't support XML responses
  - uses innerHTML, so can't even use DOM
- CPAINT
  - Supports XML and text responses
  - Actively developed and mature
  - Documentation a little immature
- JPSPAN
  - Excellent abstraction
  - Seamlessly "imports" server-side objects into JavaScript
  - Clear documentation
  - Doesn't support Opera
  - Limited usefulness

Determining Whether to Use AJAX

- AJAX is good for making Web-based versions of traditionally desktop applications
- AJAX opens up ***new*** possibilities for web apps, but does not necessarily benefit traditional possibilities
- If you need serious workarounds to bring usability up to expected levels, you're probably misusing AJAX

Thank You for Your Patience and Understanding!

Comments and questions are welcome

bebo@slac.stanford.edu

Interested in classes or tutorials at your site?

Let's talk

bebo@slac.stanford.edu