Advanced XML for Developers

XML, DTDs, Schemas, XSLT & XPath

Presented by
XMaLpha Technologies

Dale Waldt
Senior Instructor
dale@xmalpha.com

XMLADV 31 Aug 07

Introductions

- Who am I?
  - 60 Second Introduction
    - Name
    - Title, Affiliations
    - Experience
- Who are you?
  - 60 Second Introductions
    - Name
    - Title, Department
    - Role
    - Technical Background
    - Reasons for taking this class

Advanced XML for Developers - Agenda

	Day 1	Day 2
AM	XML Documents	Schema Structures
	XML Validation	Schema Data Types
PM	DTD Structures	Formatting XML Data
	DTD Organization	Processing XML Data

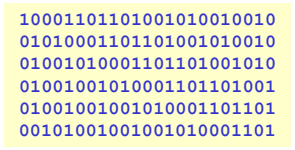
1. What is an XML Document?

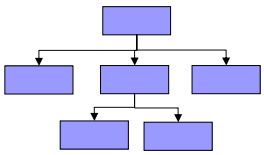
Objectives

- Define XML Markup
- Define Welformedness Rules for XML Documents
- Provide practical usage examples
- Reinforce understanding with exercises

Information May Consist of 3 Things

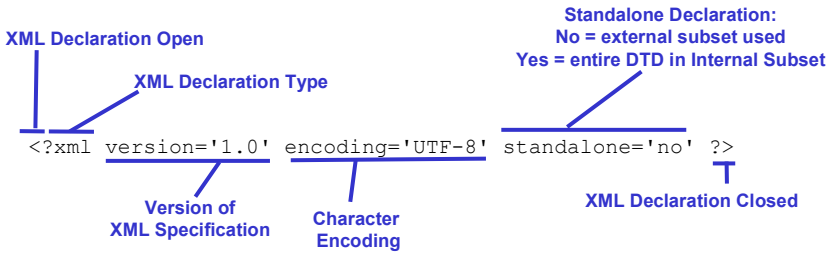
- Data
  - Information elements & values
- Structure
  - Relationship of elements
  - Location of information elements
- Presentation
  - Rendering of information to aid consumption





What is an XML Document?

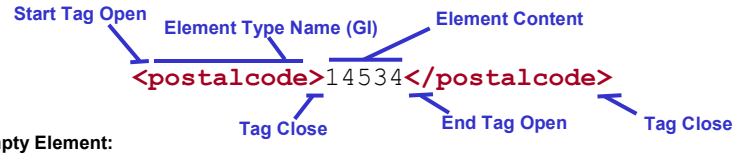
- Any data that starts with an XML Declaration
  - XML Declaration provides key information to system
  - Uses special XML Declaration syntax
  - Follows wellformed document rules



Elements in Data

- Elements are the building blocks of XML structure definitions, they can:
  - Contain other element content only
  - Contain character content only
  - Contain both character and element content
  - Contain nothing but attributes (i.e., called Empty elements)

Elements with Character Content:



Empty Element:

Attributes in Data

- Attributes appear within Element Start Tags
  - Literals delimit attribute value boundaries
    - Can be either ' or ", but must be paired consistently
  - Is case sensitive
  - Follows Name Character Rules
  - White space used to separate
  - Order not enforced
  - Attributes defined with default value or as #IMPLIED can be omitted in data instance

```
<chapter id='C01' security='secret' language='English'
                        revdate="02/12/2002">

<author name="Pat O'Mally" Reviewer='Joey "The Ox" Gambino'>
```

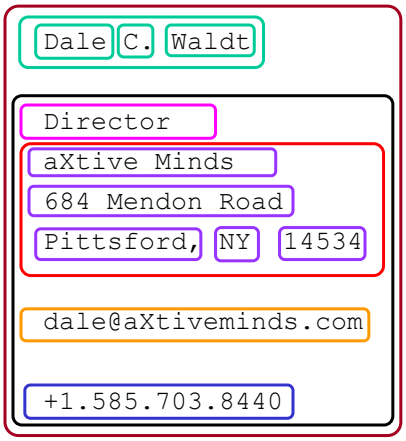
Attribute & Empty Element Example

```
<Book author="Joe Cool">
  <Chapter topic="sports">
    <title uid="X234Y">
      My Guide to the Best Sports Teams Ever
    </title>
    <para>
      Needless to say, <bold>Baltimore</bold> has the
      <italic>Orioles</italic>.
    </para>
    <para>They are a fantastic team.</para>
    <para>
      This year may be an exception. For more information
      see <ref src='S34456'/>.
    </para>
  </Chapter>
</Book>
```


Basic Well-Formedness Rules

1. Must use XML syntax for markup
   `<  >  </  &  ;  <?  ?>`
2. Case sensitive element names, start & end must match
   - `<Para>, <PARA>, <para>, & <PaRa>` all mark different elements
3. Name Characters rules in effect
   - Must begin with a letter, an underscore (_), or a colon (:)
   - May then include any combination of letters, digits, periods (.), hyphens (-), underscores (_), or colons (:)
   - Names should not begin with “XML”; all case variations are reserved for the standard
4. All literal delimiters required & must match ('value' or "value")
5. Elements require both start & end tags
6. Elements must nest consistently
7. Only one occurrence of root element

Hierarchy & Nesting of Elements




```
<contact>
  <name>
    <firstname>Dale</firstname>
    <middleinitial>C</middleinitial>
    <lastname>Waldt</lastname>
  </name>
  <affiliation>
    <title>Director</title>
    <address>
      <company>aXtive Minds</company>
      <street>684 Mendon Road</street>
      <city>Pittsford</city>
      <state>NY</state>
      <postalcode>14534</postalcode>
    </address>
    <email>dale@aXtiveminds.com</email>
    <phone>+1.585.703.8440</phone>
  </affiliation>
</contact>
```


Well-Formed Markup Example

```
<Book>
  <Chapter>
    <title>My Guide to the Best Sports Teams Ever</title>
    <para>
      The <bold>New York Yankees</bold>, have won more World
   Series titles than any other team.
    </para>
    <para>They are a fantastic team.</para>
    <para>This year may be an exception.</para>
  </Chapter>
</Book>
```

Not Well-Formed Markup Example

```
<Section>
  <title>My Guide to the Best Sports Teams Ever
  <para>4004 Boston Red Sox Beat the Curse!</PARA>
  <Section>
    <title>Bo Sox Sweep St. Louis
    <para>
      After coming back from a three game deficit, the Boston Red
  Sox won eight straight to beat the St. Louis Cardinal and win
  the 2004 World Series.
    </title></para>
  </Section>
</Section>
<Section>
  <title>My Guide to the Worst Sports Teams Ever
  <para>1986 Boston Red Sox Had a Rough Finish</para>
</section>
```


General Entities

- General Entities are named "things"
  - Only occur in data instances
  - Use "&" and ";" delimiters in markup
  - Defined in DTD & referenced in data instances
  - Defined value can contain:
    - Parsed text characters, including markup
    - Unparsed text characters or binaries
    - Unicode character references
    - External files
    - Other General Entities

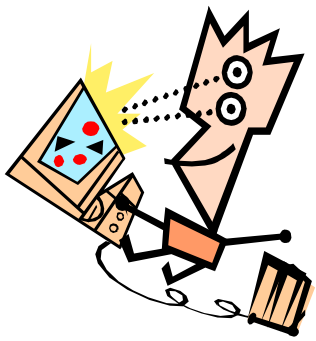
Ω Omega symbol from the Symbol font character set
&OHgr; from ISO Greek character set

Σ Summation symbol from the Symbol font character set
∑ from the ISO Added Math Symbols character set

Example with Entity References

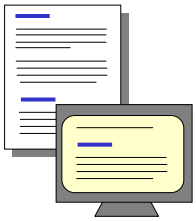
```
<Book author="CZ">
  <Chapter topic="sports">
    <title ID="X234Y">
      My Guide to the best sports teams ever.
    </title>
    <rights>&copy; 2004, Dale Waldt</rights>
    <para>
      The <bold>New York Yankees&tm;</bold>, have won more World
   Series titles than any other team.
    </para>
    <para>They are a fantastic team.</para>
    <para>This year may be an exception.</para>
  </Chapter>
</Book>
```

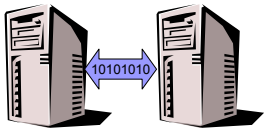

Demo



So What Can Be a Document?

- Can be a lot of different things to different applications:
  - Content Management / Publishing
    - Articles, treatises, announcements, manuals, laws, etc.
  - Data Interchange / Information Supply Chain
    - Product data, price lists, directories, purchase orders, etc.
  - Localized Software Integration / APIs
    - Records, state info, commands, etc.
  - Enterprise Application Integration (EAI)
    - Records, RPC, content, directories, etc.
  - Web Services / e-Business
    - Messages, transactions, requests, RPCs, etc.
  - Knowledge Management
    - Artifacts, actions, agent info, etc.



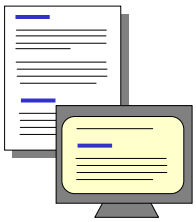


Data Fitting a Simple Model

```
<timesheet>
  <employee>Dale Waldt</employee>
  <date>May 1, 2002</date>
  <hours>40</hours>
  <overtime>0</overtime>
  <address>123 Main St., Pleasant, NY 14444</address>
</timesheet>
```

Data Fitting a More Robust Model

```
<timesheet weekstart='05/01/2002'>
  <employee>
    <fname>Dale</fname>
    <lname>Waldt</lname>
  </employee>
  <hours reg='40.00' ot='00.00'/>
  <address>
    <street>123 Main St.</street>
    <city>Pleasant</city>
    <state>NY</state>
    <zip>14444</zip>
  </address>
</timesheet>
```



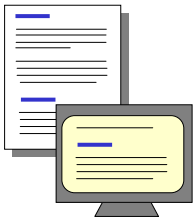
An Even More Robust Model

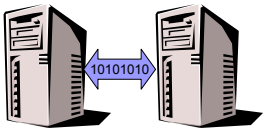
- This XML data instance:

```
<timerecord weekstart='05/01/2002' empid='335-53-3535'
            reghrs='40.00' othrs='00.00'/>
```

- Easily maps to this relational table:

key	week_start	reg_hrs	ot_hrs
335-53-3535	20020501	40.00	00.00





What About Text?

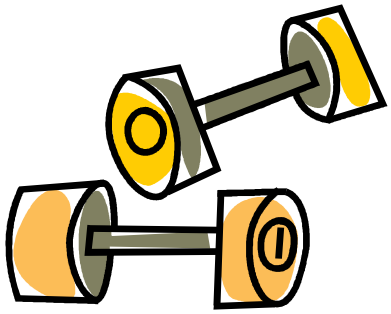
- How do you express this data in a relational table?

```
<Book>
  <Chapter>
    <title>My Guide to the Best Sports Teams Ever</title>
    <para>
      Needless to say, <bold>Baltimore</bold>, has
      the <italic>Orioles</italic>.
    </para>
    <para>They are a fantastic team.</para>
    <para>This year may be an exception.</para>
  </Chapter>
</Book>
```


XML Data Analysis Process

- Maintain an enterprise view of requirements
  - Consider entire data lifecycle
- Design team must have broad representation
  - Not created by engineers alone
  - Requires domain expertise
- Normalize Dissimilar Definitions
  - Normalize vocabulary to be robust enough to serve all uses
  - Eliminate redundancies, ambiguity & manual work where feasible
  - Use "obvious" names & structures
- Set Appropriate Scope
  - Define your target data type
  - Prioritize & work toward most valuable needs
  - Eliminate Impossible Goals
- Validate your design
  - Eliminate ambiguity
  - Test design against expectations

Exercises - Module 1



2. XML Validation

Objectives

- Define XML Validation Concepts
- Describe XML Processing & Infosets
- Provide practical usage examples
- Reinforce understanding with exercises

Well-formed XML Info May be Enough

- Well-formed documents are useful when variety & inconsistency is okay
  - Simple documents can be rendered & read
    - HTML
    - XML/CSS or XML/XSLT
    - Messages
- Not suitable when specific structure or vocabulary are required
  - Record structure verification
    - E.g., APIs, RPC, EDI, etc.
  - Minimum required information set
    - E.g., Forms & other data entry
  - Use of Rules Schema Assists processing

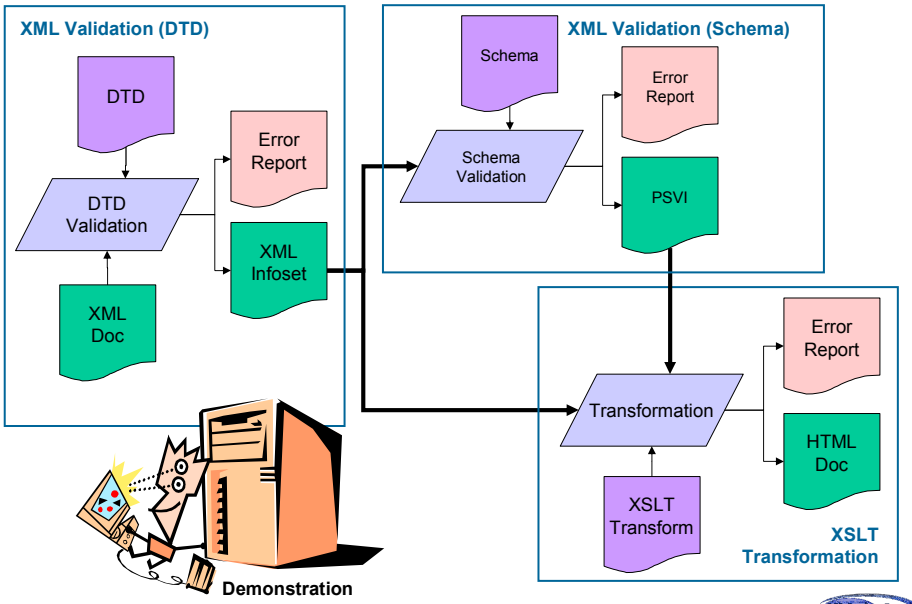
XML Documents may be Validated

```
<?xml version='1.0'?>
<!--?xml-stylesheet type='text/xsl' href='headlines2.xsl'?-->
<!DOCTYPE headlines  [
<!-- General Entity Declarations -->
<!ENTITY axm        "aXtive Minds" >
<!-- Elements & Attributes -->
<!ELEMENT headlines (article+) >
<!ELEMENT article   (head, city?, p) >
<!ATTLIST article   date CDATA #REQUIRED >
<!ELEMENT head      (#PCDATA) >
<!ELEMENT city      (#PCDATA) >
<!ELEMENT p         (#PCDATA) >
]>
<headlines>
 <article date='February 24, 2002'>
  <head>Canadian Curlers Sweep!
  </head>
  <city>Salt Lake City
  </city>
  <p>Canadian curling teams
  swept the final competitions to win gold, silver and
  bronze medals.
  </p>
 </article>
</headlines>
```

Validation Rules

Data

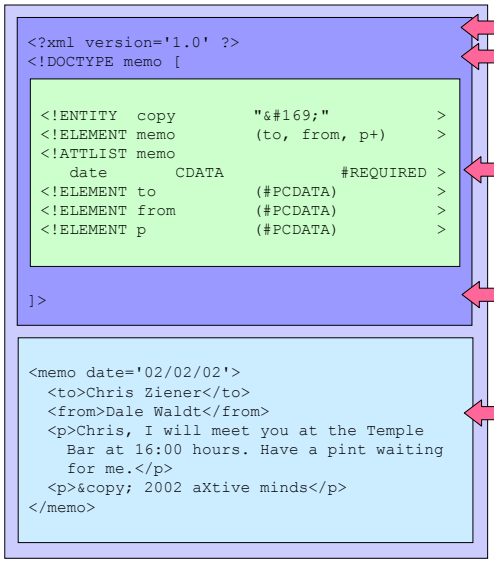
XML Validation & Transformation



Document Type Definitions (DTDs)

- Models one document type
  - (e.g., email, manual, balance sheet, command line, article, wire transfer, error message, report, memo, statement, invoice, treatise, play, etc.)
- Describes rules for documents instance of that type
  - (e.g., names, values, structure, sequence, objects, etc.)
  - Uses ELEMENT, ATTLIST, ENTITY, & NOTATION Declarations
- Is written in formal XML Declaration syntax
- Useful to communicate rules to other users & applications
- Used to test data instance compliance to these rules
- Provide limited datatyping capabilities
  - XML Schemas much more powerful for datatyping
- Can be referenced, stored inline, or dynamically assembled from fragments

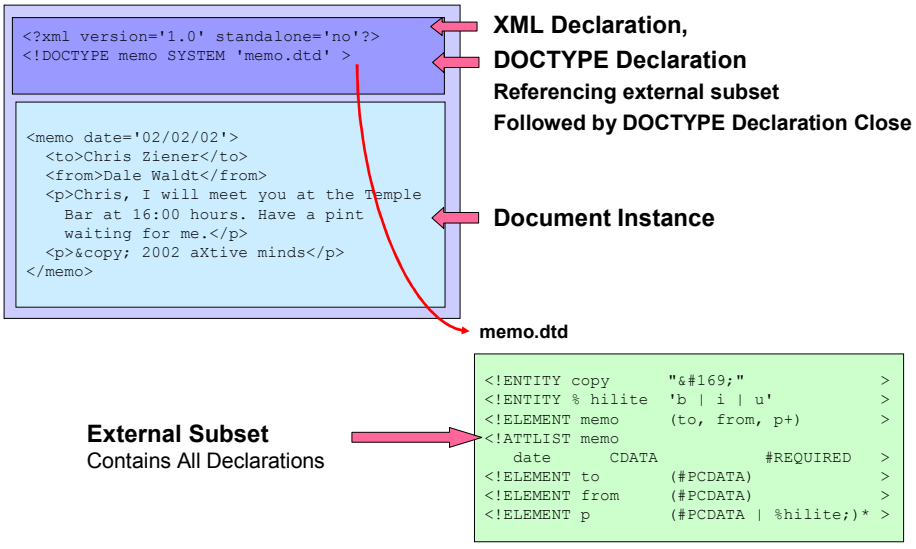
Internal Subset




```
<?xml version='1.0' ?>
<!DOCTYPE memo [

  <!ENTITY  copy        "&#169;"            >
  <!ELEMENT memo        (to, from, p+)      >
  <!ATTLIST memo
     date       CDATA            #REQUIRED >
  <!ELEMENT to          (#PCDATA)           >
  <!ELEMENT from        (#PCDATA)           >
  <!ELEMENT p           (#PCDATA)           >

]>
```

```
<memo date='02/02/02'>
  <to>Chris Ziener</to>
  <from>Dale Waldt</from>
  <p>Chris, I will meet you at the Temple
    Bar at 16:00 hours. Have a pint waiting
    for me.</p>
  <p>&copy; 2002 aXtive minds</p>
</memo>
```


External Subset




```
<?xml version='1.0' standalone='no'?>
<!DOCTYPE memo SYSTEM 'memo.dtd' >
```

```
<memo date='02/02/02'>
  <to>Chris Ziener</to>
  <from>Dale Waldt</from>
  <p>Chris, I will meet you at the Temple
    Bar at 16:00 hours. Have a pint
    waiting for me.</p>
  <p>&copy; 2002 aXtive minds</p>
</memo>
```

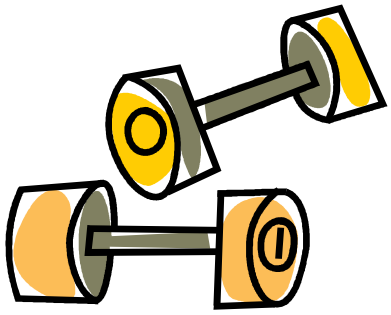
```
<!ENTITY copy      "&#169;"                 >
<!ENTITY % hilite  'b | i | u'              >
<!ELEMENT memo     (to, from, p+)           >
<!ATTLIST memo
   date      CDATA           #REQUIRED     >
<!ELEMENT to       (#PCDATA)                >
<!ELEMENT from     (#PCDATA)                >
<!ELEMENT p        (#PCDATA | %hilite;)* >
```

Comments in DTDs & Markup

- Information for humans or non-XML processes
  - should be ignored by the XML processor
- Can be used to add additional information to DTDs & documents

```
<article>
<!-- UID:0883876;REV:03012001;AU:DCW -->
    <para>Surfing is easy and fun.</para>
<!-- NOTE: Dale,
           Please check quotes!
           Thanks, Chris -->
</article>
```


Exercises - Module 2



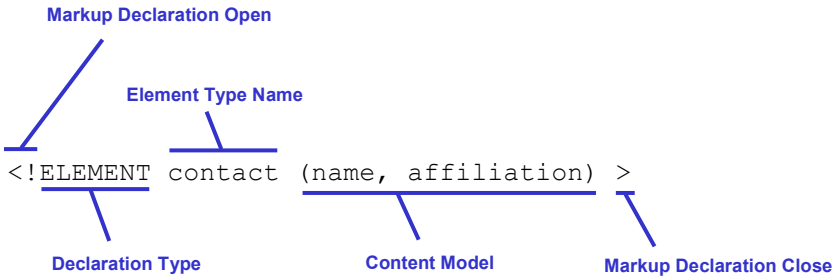
3. Defining Document Structure

Objectives

- Define Element concepts & usage
- Define Element Declarations & structure
- Define Element instance markup syntax & structure
- Reinforce understanding with exercises

Element Declaration Structure

- Declared using ELEMENT declarations:



Content Models

- Separators (Connectors)

Type	Symbol	Read As
Sequence	,	“followed by”
Choice	|	"or"

Caution: Separators cannot be mixed within a group in a content model. They must be used consistently within a group.

- Occurrence Indicators

Type	Symbol	Read As
Required	*(no symbol)*	Must have one, and only one
Optional	?	May have zero or only one
Required & Repeatable	+	Must have one or more
Optional & Repeatable	*	May have zero or more

Groups in Content Models

- Elements can be grouped in content models
  - Use "(" for group start and ")" for group end
  - Groups may be modified by Occurrence Indicators
  - Groups can be nested

```
<!ELEMENT memo (date, (to | from)+, p+) >
<!ELEMENT contact (name, (company, (address | email)+)*) >
```

  - Only one type of Separator allowed within a group
  - Complex modeling rules can be made
  - Group Nesting allows use of more than one type of Separator in Content Model

```
<!ELEMENT a  ((b | c) | ((d | e)+, (f | g)?)+) >
```


Element Content Types

- Element Content

```
<!ELEMENT person (name+) >
<person><name>Dale Waldt</name></person>
```

- Data Content

```
<!ELEMENT firstname (#PCDATA) >
<firstname>Dale</firstname>
```

- Mixed Content

```
<!ELEMENT p (#PCDATA | b | i)* >
<p>This is a paragraph with a <b>Bold</b> word</p>
```

- Empty Content

```
<!ELEMENT image EMPTY  >
<!ATTLIST image href CDATA #REQUIRED >
<image href="bigdog.jpg"/>
```


Mixed Content Rules

- Elements contain other elements AND data

```
<paragraph>This is a sample textual paragraph
with a <bold>highlighted</bold> word illustrating
the concept of Mixed Content.</paragraph>
```

- Must declare data content before elements in content model
- Must use "or" separator
- Must have Optional/Repeatable indicator (*)

```
<!ELEMENT paragraph (#PCDATA | bold)* >
```

Caution: Mixed content may require special handling in processing!

EMPTY Elements

- Elements can be declared to have no content
- Use reserved word EMPTY for content model
- Use EMPTY tag delimiters **<** and **/>** (no end tag)

```
<!ELEMENT poem      (#PCDATA | linebreak)* >
<!ELEMENT linebreak EMPTY >
```

```
<poem>
mr u will not be missed<linebreak/>
who as an anthologist<linebreak/>
sold the many on the few<linebreak/>
not excluding mr u
</poem>
```

mr u will not be missed
who as an anthologist
sold the many on the few
not excluding mr u

Content Model Examples

"Contact" contains one required "name", followed by one required "affiliation":

```
<!ELEMENT contact (name, affiliation) >
```

"Bizdoc" requires either one "purchase order" or one "invoice":

```
<!ELEMENT bizdoc (purchaseorder | invoice) >
```

"Memo" must have one "date", followed by one or more "names", followed by one or more "paragraphs", followed by one optional "enclosure":

```
<!ELEMENT memo (date, name+, p+, enclosure?) >
```

"Paragraph" can have any number of "bold" or "name" or Parsed Character Data in any order:

```
<!ELEMENT paragraph (#PCDATA | bold | name)* >
```

"Postalcode" can only contain Parsed Character Data:

```
<!ELEMENT postalcode (#PCDATA) >
```


Erroneous Content Models

Errors

```
<!ELEMENT contact (name, affiliation | title) >

<!ELEMENT p (#PCDATA | (affiliation | title))* >

<!ELEMENT bizdoc (purchaseorder | invoice) signature) >

<!ELEMENT paragraph (bold | name | #PCDATA)+ >
```

Bad Practice

```
<!ELEMENT memo (date, name, enclosure)* >

<!ELEMENT linebreak (EMPTY) >

<!ELEMENT postalcode (PCDATA) >
```


Modeling Design Choices

Simpler models are usually easier to understand & maintain

Flat: few hierarchy levels used to keep model simple

```
<!ELEMENT book (title, ((chptitle, p+)+ | (sectitle,
     p+,(chptitle, p+)+)+)) >
```

Layered: emphasizes hierarchical relationships

```
<!ELEMENT book (title, (chapter+ | section+)) >
```

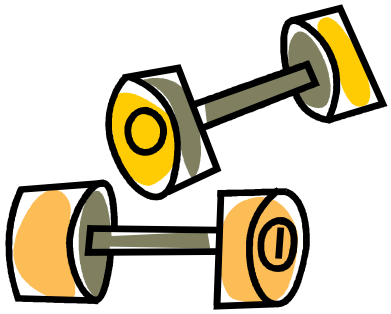
Fully Constrained: very stringent rules enforced

```
<!ELEMENT date ((d, m, y) | (m, d, y) | (y, m, d) |
     (m, y)) >
```

Allowed Children: enforces lower-level names but not order for flexibility

```
<!ELEMENT date (d | m | y)+ >
```


Exercises - Module 3



4. Adding Detail to Structure

Objectives

- Define Attribute concepts & usage
- Define Attribute List Declarations & structure
- Define Attribute values & types
- Reinforce understanding with exercises

Attributes

- Modify Elements
  - Must be tied to a defined XML Element
  - Provide additional information to the Element
- Can be a list of predefined values
- Can have default values or declared value types
- Can be optional or required

```
<!ELEMENT chapter (title, p+) >
<!ATTLIST chapter number CDATA #REQUIRED >
...
<chapter number='1.0'>
```

Note: If Elements are like Nouns, then Attributes are like adjectives!

Attributes Modify Elements

- Provide additional information about the element
  - Examples:
    - Figures (source, size, alignment)
    - Documents & Sections (IDs, authors, security levels)
    - Messages (associations, time/date, type)
    - APIs (commands, parameters, values)
- Must appear in the start tag, in the form:

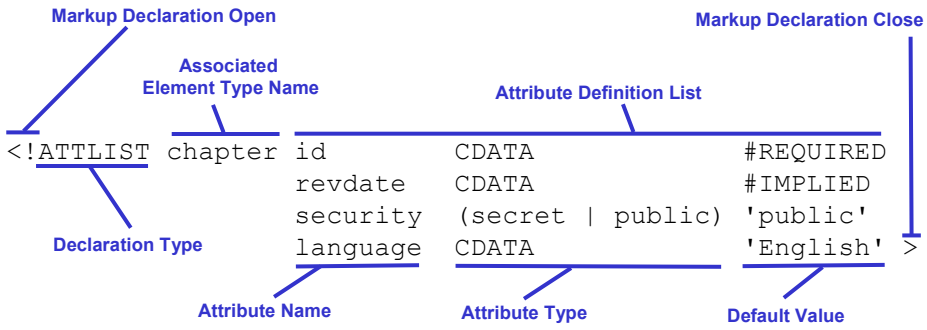
```
<Tag Attribute="value">
    or
<Tag Attribute='value'>
```


Attribute & Empty Element Example

```
<Book author="Joe Cool">
  <Chapter topic="sports">
    <title uid="X234Y">
      My Guide to the Best Sports Teams Ever
    </title>
    <para>
      Needless to say, <bold>Baltimore</bold> has the
      <italic>Orioles</italic>.
    </para>
    <para>They are a fantastic team.</para>
    <para>
      This year may be an exception. For more information
      see <ref src='S34456'/>.
    </para>
  </Chapter>
</Book>
```


Declaring Attributes

- Attribute List Declaration consists of:
  - Reference to Associated Element Type
  - Attribute List (may define multiple attributes)
    - Includes Name & values/rules for each Attribute



Attribute Types & Keywords

String Type Attributes	Description
CDATA	Zero or more characters, may include entities & spaces
Tokenized Value Types	**Description**
NMTOKEN	Name Token: Zero or more Name Characters without spaces
NMTOKENS	Name Tokens: One or more Name Token values separated by spaces
ID	Unique Identifier for this Element
IDREF	Reference to an existing Unique Identifier for an Element
IDREFS	One or more IDREF values separated by spaces
ENTITY	Name of a General Entity which has been declared
ENTITIES	One or more ENTITY values separated by spaces
Enumerated List Types	**Description**
(value1 | value2 | value3)	Enumerated list of declared values enclosed in a group with "or" separators
NOTATION (value1 | value2)	Keyword followed by enumerated list of Notations values enclosed in a group with "or" separators, NOTATION type must be declared elsewhere using NOTATION declaration

Format for Attribute Declarations:

```
<!ATTLIST element-type-name  attribute-name  attribute-type  attribute-default >
```


Default Values & Default Value Types

"*value*"	Default value applied when attribute is omitted from data
#REQUIRED	Indicates a value is required and must appear in data and match corresponding declared Attribute Type
#IMPLIED	Indicates a value is optional and may or may not appear in data but must match corresponding declared Attribute Type if entered
#FIXED "*value*"	All occurrences of this Attribute will be set to the declared value and an error will be reported if a value entered in data does not match this value

Default Value Type

```
<!ATTLIST image id       ID                      #REQUIRED
               caption   CDATA                   #IMPLIED
               rights    (private | public)      'public'
               format    NOTATION  (jpeg | gif)  'jpeg' >
<image id='G1234' format='gif'>
```

Default Value

Format for Attribute Declarations:

```
<!ATTLIST element-type-name  attribute-name  attribute-type  attribute-default >
```


Multiple Attribute Declarations

- Multiple ATTLISTs can modify an Element

```
<!ELEMENT chapter (title, p+) >
<!ATTLIST chapter number  CDATA #REQUIRED >
<!ATTLIST chapter revdate CDATA #REQUIRED >
. . .
<chapter number='1.0' revdate='02/02/02'>
```

  - If two are declared with the same Attribute name, the first Attribute declared takes precedence
  - May support dynamically assembled DTDs
  - Can cause problems in data instances when all DTD modules are not in use
  - (More on DTD Modularization in Module Module 9)

Reserved Attributes

- xml:lang
  - Refers to language of Elements and Attributes in document
  - Value is a Name Token
  - Defined by IETF RFC 1766
  - Is not case sensitive

```
<article xml:lang='EN' >. . .
```

- xml:space
  - Indicates how white space should be handled
    - Whitespace is: space, tab, blank line
    - "default" value tells application to handle it normally
    - "preserve" value tells application to retain as is

```
<programcode xml:space='preserve' >. . .
```


Attributes vs. Elements – Best Practices

- Elements are best for:
  - Hierarchy (Parent / Child relationship)
  - Containers
  - Text passages
  - Sequencing
  - Content without constraints
- Attributes are best for:
  - Modifying information
  - Metadata
  - Enumeration
  - Constrained values or types

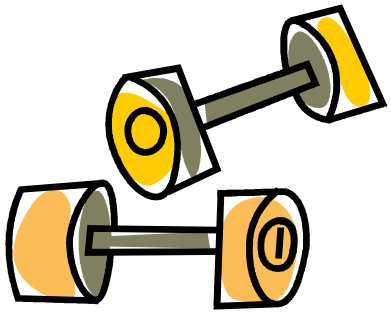
Note: If Elements are like Nouns, then Attributes are like adjectives!

Group Discussion Exercise

Read aloud the following Attribute List Declarations in plain language:

```
<!ATTLIST memo security (secret | public)      #REQUIRED
              author   CDATA                   #IMPLIED     >

<!ATTLIST date m  (01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 |
                   10 | 11 | 12 )              #REQUIRED
               d  (01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 |
                   10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
                   19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 |
                   28 | 29 | 30 | 31)          #REQUIRED
               y   (2002 | 2003 | 2004 | 2005) '2003'       >

<!ATTLIST response message  ID        #REQUIRED
                  recipient CDATA    #REQUIRED >

<!ATTLIST user      userid  NMTOKENS  #IMPLIED  >

<!ATTLIST xref      refid   IDREFS    #REQUIRED >
```


Exercises - Module 4



5. General Entities

Objectives

- Define General Entity concepts & usage
- Define General Entity Declarations & structure
- Define General Entity Content Types
- Reinforce understanding with exercises

General Entities

- General Entities are named "things"
- General Entities only occur in data instances
  - (Note: Parameter Entities only occur in DTDs)
  - Use "&" and ";" delimiters in markup
  - Global definitions defined in DTD & referenced in data instances
  - Defined value can contain:
    - Parsed text characters, including markup
    - Unparsed text characters or binaries
      - Note: Special Use Rules Apply
    - Unicode character references
    - External files
    - Other General Entities

General Entity References

- Entities are:
  - Used to include parts of a document by reference
  - Expressed as an “&” followed by the entity name and ending with a “;” for example…

Ω Omega symbol from the Symbol font character set

- &OHgr; from ISO Greek character set

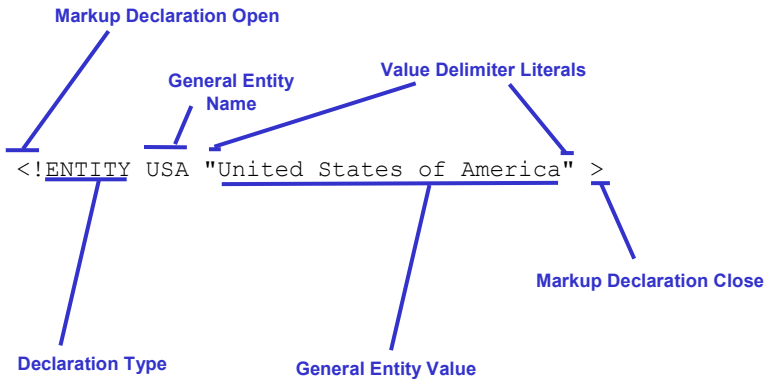
Σ Summation symbol from the Symbol font character set

- ∑ from the ISO Added Math Symbols character set

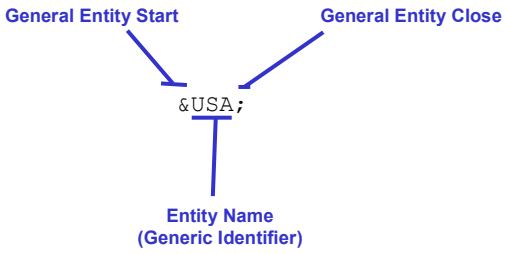
Example with General Entity References

```
<Book author="CZ">
  <Chapter topic="sports">
    <title ID="X234Y">
      My Guide to the best sports teams ever.
    </title>
    <rights>&copy; 2002, Chris Ziener</rights>
    <para>
      Needless to say, <bold>Baltimore</bold> has the
      <italic>Orioles&tm;</italic>.
    </para>
    <para>They are a fantastic team.</para>
    <para>This year may be an exception.</para>
  </Chapter>
</Book>
```

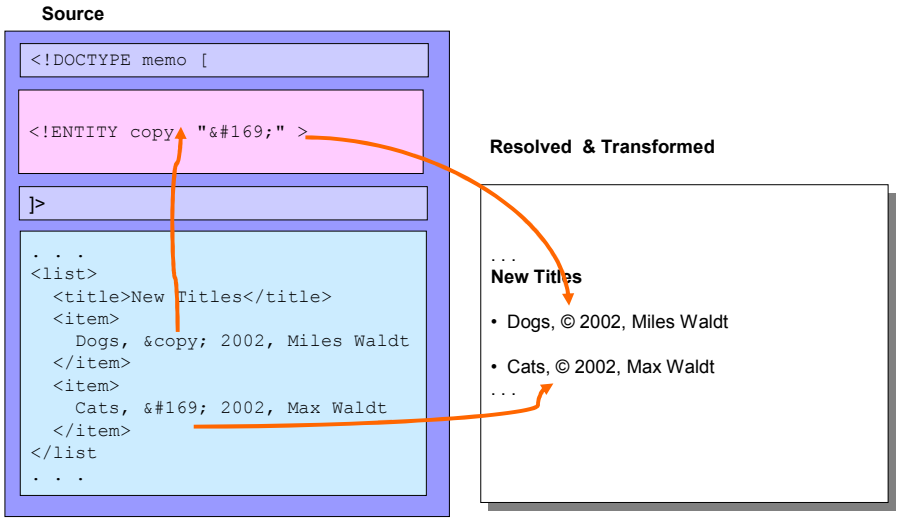

General Entities – Declarations



General Entities - Markup



General Entity Reference Resolution



General Entity Nesting

- General Entities can be nested:

Given:

```
<!ENTITY amlogo     "a<hi1>X</hi1>tive minds" >
<!ENTITY error-22   "Error 22, Call your &amlogo; hotline!" >
```

This:

```
<errormessage>&error-22;</errormessage>
```

Resolves to:

```
<errormessage>Error 22, Call your &amlogo;
hotline!</errormessage>
```

Which in turn resolves to:

```
<errormessage>Error 22, Call your a<hi1>X</hi1>tive minds
hotline!</errormessage>
```

Which is rendered as:

Error 22, Call your a**X**tive minds hotline!

General Entities

- Some rules & rules of thumb:
  - Can be defined but not referenced
  - Must be defined if referenced or validation error is reported
  - Can resolve to SYSTEM or PUBLIC external identifiers
  - Case sensitive, Name Character rules apply
  - Often used for
    - Special characters
    - Boilerplate text
    - Modular data

Built-in General Entities

- XML Defines Entities for Special Characters used as Markup Syntax:
  - **<** can be expressed as **<**
  - **>** can be expressed as **>**
  - **&** can be expressed as **&**
  - **"** can be expressed as **"**
  - **'** can be expressed as **'**
  - Can use Unicode Character References
    - **&#** start delimiter indicates a decimal value
    - **&#x** start delimiter indicates a hexadecimal value
- New Characters can be Defined:

Given:

```
<!ENTITY copy "&#169;"      >
```

This:

```
&lt;Markup&gt; Explained &amp; Illustrated, &copy; 2002 Dale Waldt
```

Becomes this:

<Markup> Explained & Illustrated, © 2002 Dale Waldt

General Entities in Practice

- Boilerplate Text

Note: See how the same General Entity Reference can be declared for different results for different uses without changing any data!

Given:

```
<!ENTITY USpres "George W. Bush" >
```

This:

```
<p>The President of the United States, &USpres;,
welcomes you!</p>
```

Becomes this:

The President of the United States, George W. Bush, welcomes you!

Or, in a different context, given:

```
<!ENTITY USpres "Alfred E. Neuman" >
```

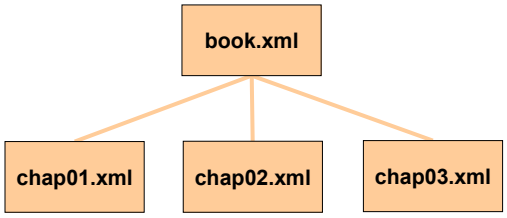
It becomes this:

The President of the United States, Alfred E. Neuman, welcomes you!

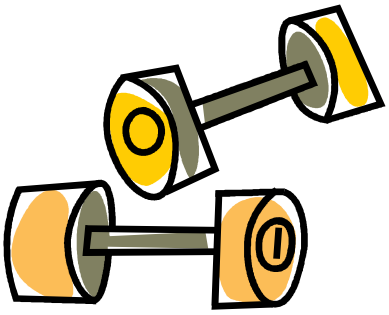
General Entities in Practice

- Modular Data
  - Documents can be dynamically assembled

```
<!ENTITY chapter01 SYSTEM "chap01.xml" >
<!ENTITY chapter02 SYSTEM "chap02.xml" >
<!ENTITY chapter03 SYSTEM "chap03.xml" >
. . .
<book><title>A book with 3 Chapters</title>
&chapter01;
&chapter02;
&chapter03;
</book>
```



Exercises - Module 5



6. DTD Organization & Best Practices

Objectives

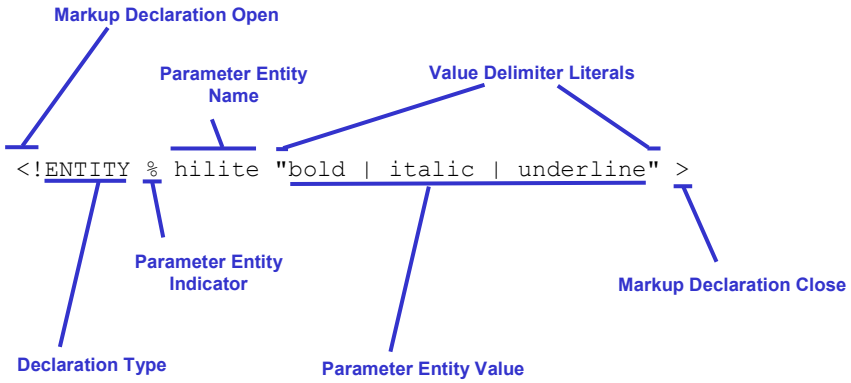
- Define Parameter Entity concepts & usage
- Define Parameter Entity Declarations & structure
- Reinforce understanding with exercises

Parameter Entities

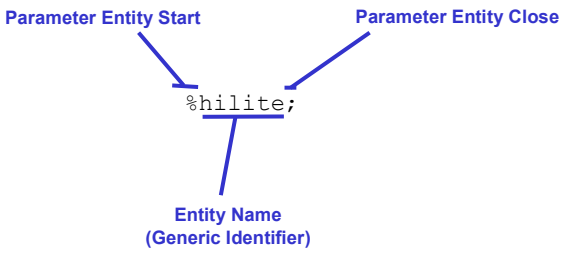
- General Entities are named objects within data instances (more on that later…)
- Parameter Entities are named objects within DTD
  - Use % and ; delimiters in markup
  - Global definitions referenced throughout DTD
  - Must be defined before (above) being used
  - Are parsed after being resolved
  - Defined value can contain:
    - Text characters
    - Declarations (or blocks of/portions of declarations)
    - Other Parameter Entities

```
<!ENTITY % a "b | c " >
<!ELEMENT p (#PCDATA | %a;)* >
is the same as…
<!ELEMENT p (#PCDATA | b | c )* >
```

Parameter Entities - Declarations



Parameter Entities - References



Parameter Entities - Resolution

Source

```
<!-- DTD in an External Subset -->
<!ENTITY % hilite 'b | i | u' >

<!ELEMENT memo (p+) >

. . .
<!ELEMENT p (#PCDATA | %hilite;)* >
. . .
```

Resolved

```
<!-- DTD as Internal Subset with PERs
     Resolved -->
. . .

<!ELEMENT memo (p+) >
. . .
<!ELEMENT p  (#PCDATA | b | i | u )* >
. . .
```


Parameter Entities

- Defined value is "replaced" where referenced:

Given:

```
<!ENTITY % hilite "bold | italic | underline" >
<!ENTITY % refs   "a | xref" >
```

This:

```
<!ELEMENT paragraph (#PCDATA | %hilite; | %refs;)* >
```

Resolves to:

```
<!ELEMENT paragraph (#PCDATA | bold | italic | underline
                          | a | xref)* >
```

Parameter Entity Nesting

- Parameter Entities can be nested:

Given:

```
<!ENTITY % hilite "bold | italic | underline" >
<!ENTITY % refs   "a | xref" >
<!ENTITY % pstuff "%hilite; | %refs;" >
```

This:

```
<!ELEMENT paragraph (#PCDATA | %pstuff;)* >
```

Resolves to:

```
<!ELEMENT paragraph (#PCDATA | bold | italic | underline
                        | a | xref)* >
```


Parameter Entities in Practice

- Grouping Like Things:

Reference is to Entity in External Subset

```
<!ENTITY % plevel "p | list | graphic | table | note" >
. . .
<!ELEMENT document (head, (%plevel;)+)   >
```


Parameter Entities in Practice

- Improving Readability & Maintainability:

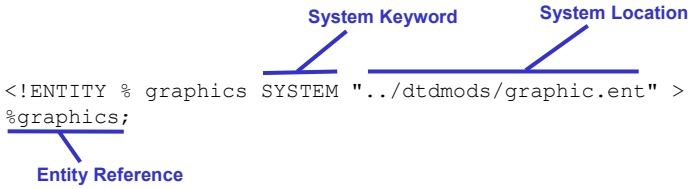
```
<!ENTITY % plevel "p | list | graphic | table | note" >
<!ENTITY % atts   "security (secret | public) 'public'
                   revdate  CDATA     #REQUIRED
                   author   NMTOKENS  #IMPLIED
                   type     (draft | final)    'draft' " >
. . .
<!ELEMENT document (head, (%plevel;)+)   >
<!ATTLIST document %atts; >

<!ELEMENT memo     (head, (%plevel;)+)   >
<!ATTLIST memo     %atts; >

<!ELEMENT letter   (head, (%plevel;)+)   >
<!ATTLIST letter   %atts; >

<!ELEMENT form     (head, (%plevel;)+)   >
<!ATTLIST form     %atts; >
```


Parameter Entities

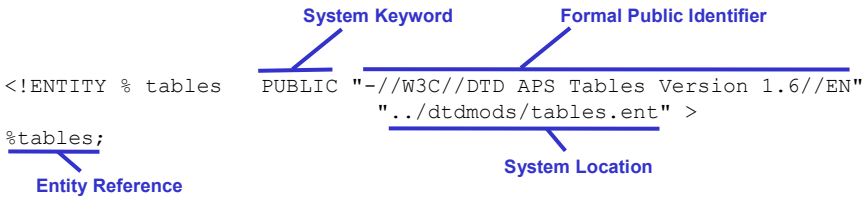
- Some rules & rules of thumb:
  - Can be defined but not referenced
  - Must be defined if referenced or validation error is reported
  - Often used for:
    - Making a DTD look cleaner & easier to read & maintain
    - Commonly repeated elements in content models
    - Commonly repeated attributes in attribute lists
    - Commonly reused DTD fragments
  - Can resolve to SYSTEM or PUBLIC external identifiers
    - Modular DTD fragments for dynamic DTD assembly
    - "Official" DTD fragments found "elsewhere"

Parameter Entities in Practice

- External References for DTD Modularity
  - System External Reference



  - Public External Reference



DTD Module Assembly

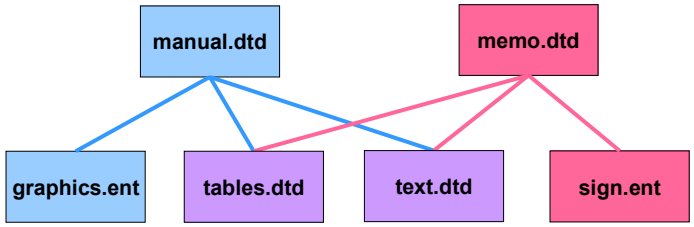
- External Parameter Entities used to create DTD Modules

```
<!DOCTYPE manual  [
<!ENTITY % text SYSTEM "../text.dtd" >
%text;
<!ENTITY % graphics SYSTEM "../graphics.ent" >
%graphics;
<!ENTITY % tables SYSTEM "../tables.dtd"
%tables; . . .
```

Training Manual DTD

```
<!DOCTYPE memo  [
<!ENTITY % text SYSTEM "../text.dtd" >
%text;
<!ENTITY % signature SYSTEM "../sign.ent" >
%signature;
<!ENTITY % tables SYSTEM "../tables.dtd"
%tables; . . .
```

Memo DTD



XML Entities Summary

Entity Type	Delimiters Start	Delimiters End	Value Locations	Location Keywords	Referenced In
Parameter	%	;	Internal	n/a	DTD
			External	SYSTEM PUBLIC	DTD
General	&	;	Internal	n/a	Data
			External Parsed or External Unparsed	SYSTEM PUBLIC	Data

Formatting DTDs for Readability

```
<?xml version='1.0'
standalone='no'?>
<!DOCTYPE note [
<!ENTITY axm "aXtive Minds">
<!ENTITY copyright "&#169;">
<!ELEMENT note (to, from, p+)>
<!ATTLIST note date CDATA
#REQUIRED sec (s | p) 's'>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT p (#PCDATA)>]>
<note date='2 Feb
2002'><to>Kurt</to><from>Dale
</from><p>Please review these
&axm; slides. Thanks!</p></note>
```

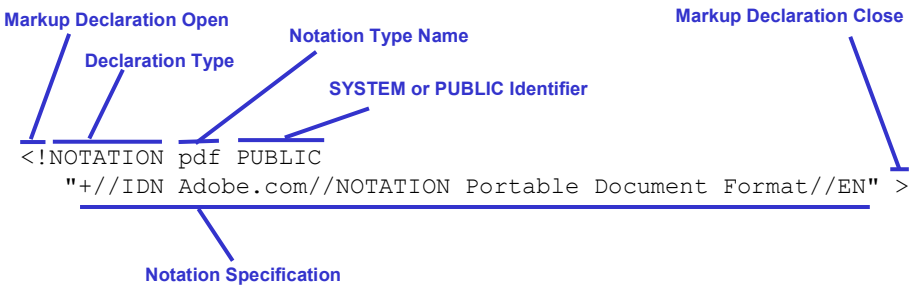
Less Legible

```
<?xml version='1.0' standalone='no'?>
<!DOCTYPE note [
<!-- Last Update: DCW 02/02/2002       -->

<!-- Entities                          -->
<!ENTITY  axm         "aXtive Minds"     >
<!ENTITY  copyright   "&#169;"           >

<!-- Structural Elements               -->
<!ELEMENT note        (to, from, p+)     >
<!ATTLIST note
   date     CDATA           #REQUIRED
   sec      (s | p)         's'          >

<!-- Terminal Node Elements            -->
<!ELEMENT to          (#PCDATA)          >
<!ELEMENT from        (#PCDATA)          >
<!ELEMENT p           (#PCDATA)          >
]>
<note date='2 Feb 2002'>
 <to>Kurt</to>
 <from>Dale</from>
 <p>Please review these &axm; slides.
 Thanks!</p>
</note>
```

More Legible

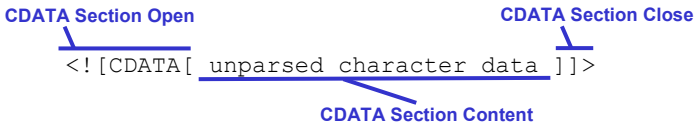
NOTATION Declarations

- Used to declare non-XML datatypes in use
  - Refers to SYSTEM or PUBLIC identifier
- Unparsed Entities declared with an associated Notation
  - (E.g., graphics, audio, video, proprietary encoding, executables, etc.)
- May be passed to processor to invoke appropriate interpreter



CDATA Sections

- Used to define a Section of a document
  - Markup Not recognized in CDATA Section
  - Only Data Content contained in CDATA Section
- Used to remove blocks of text from XML processing
  - Avoid confusion of characters that only look like XML markup
  - Avoid processing XML markup intentionally
- Delimited by <![CDATA[for start and]]> for end
  - Content cannot include]] or it will end the CDATA Section



CDATA Sections in Practice

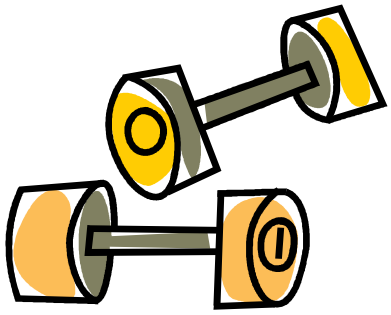
- To enter "display markup"
  - Not processed, displayed or printed in output

```
<para>An XML tag for a Chapter looks like <![CDATA[<chapter
number='1'>]]>.</para>
```

- To include inline non-XML data that may look like XML markup but avoid processing it:

```
<example id='T1234'><name>Sample Javascript in HTML.</name>
<content><![CDATA[
<html><body>
<SCRIPT LANGUAGE="JavaScript">
<!--
function prompter()
{
name=prompt('What\'s your name?',"Ezekial");
Response=alert("Welcome "+name);
}
//-->
</SCRIPT>
</body></html>
]]></content>
</example>
```


Exercises - Module 6



7. Intro to W3C Schema

Objectives

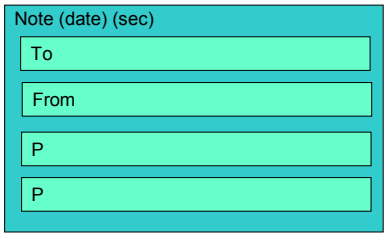
- Understand XML Schemas
  - What schemas provide
  - How are they different than XML DTDs
- Understand Schema Structures
- Understand Schema Element Definitions
- Understand Global vs. Local Definition Concepts
- Reinforce with Basic Schema Element Exercise

W3C XML Schema Language

- W3C Recommendation May 2, 2001
- Replaces & supplements DTD functionality
- Overcomes many limitations of DTDs
- Uses an XML syntax
  - Very verbose
- Two parts of XML Schema
  - XML Schema Part 1: Structures
  - XML Schema Part 2: Datatypes
- Several "competing" schema approaches out there, but convergence is happening

Simple XML DTD with Content Models

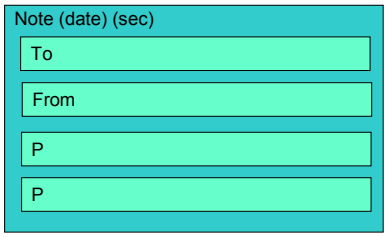
```
<?xml version='1.0' standalone='no'?>
<!DOCTYPE note [
<!- Last Update: DCW 02/02/2002       -->

<!- Entities -->
<!ENTITY  axm       "aXtive Minds"      >
<!ENTITY  copyright "#x169"             >
<!ENTITY  pointer   SYSTEM
                    "./images/hand.gif" >

<!- Structural Elements                 -->
<!ELEMENT note  (to, from, p+)          >
<!ATTLIST note  date  #PCDATA  #REQUIRED
                sec   (s | p)  's'      >

<!- Terminal Node Elements              -->
<!ELEMENT to    (#PCDATA)               >
<!ELEMENT from  (#PCDATA)               >
<!ELEMENT p     (#PCDATA)               >
]>


<note date='2002-02-02'>
 <to>Kurt</to>
 <from>Dale</from>
 <p>Please review these slides.</p>
 <p>Thanks!</p>
</note>
```



- **Specifies**
  - Names
  - Hierarchy
  - Limited value types
  - Sequence
  - Order
  - Limited occurrence
  - Limited default values
  - Special Characters & other objects

Simple W3C Schema Definitions

```
<?xml version='1.0' ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="note">
 <xs:complexType>
  <xs:sequence>
   <xs:element name="to" type='xs:string'/>
   <xs:element name="from" type='xs:string'/>
   <xs:element name="p" type="pType"
               maxOccurs='unbounded'/>
  </xs:sequence>
  <xs:attribute name='date' type='xs:date'/>
  <xs:attribute name='sec'  type='xs:string'
               default='s'/>
 </xs:complexType>
</xs:element>
<xs:complexType name='pType' mixed="true">
  <xs:choice minOccurs='0' maxOccurs='unbounded'>
    <xs:element name="b" type='xs:string'/>
    <xs:element name="i" type='xs:string'/>
  </xs:choice>
</xs:complexType>
</xs:schema>

<note date='2002-02-02'>
 <to>Kurt</to>
 <from>Dale</from>
 <p>Please review these slides.</p>
 <p>Thanks!</p>
</note>
```



- **Specifies**
  - All XML DTD Structure, Naming, Occurrence, etc.
  - Enhanced modeling constraints
  - Enhanced data typing
  - Namespaces

Other Schema Languages

- OASIS RELAX NG TC
  - "Competing" Schema Specification
  - Editors: James Clark & Murata Makoto
  - Convergence of two similar Schema Specifications
  - Simpler Expression language
  - http://oasis-open.org/relax-ng

- Schematron
  - Complimentary Extension of Either Schema Language
  - W3C Note submitted by Rick Jelliffe, Topologi.com
  - Based on Assertion Rules
  - Supports Non-Grammar Structures
  - http:topologi.com or http://w3.org/schema

- Expect Further Convergence in Schema Arena…

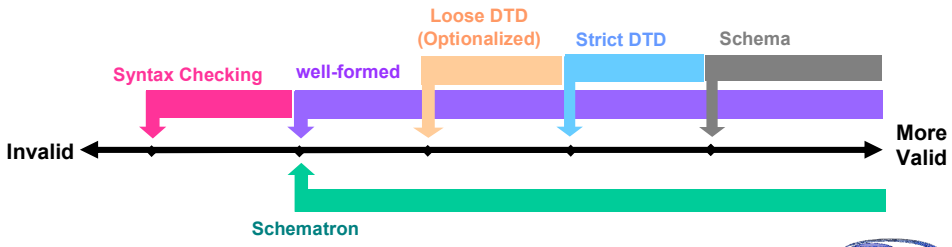
Schema Languages Compared

Requirement	DTDs	W3C Schema	RELAX NG	Schematron
Defines Structures	●	●	●	
Defines Structure Types	❍	●	●	
Defines Attributes	●	●	●	
Defines Attribute Types	❍	●	●	
Defines General Entities	●			
Defines Element Content Data Types		●	●	
Allows Derived Element Content Data Types		●	●	
Defines Attribute Value Data Types	❍	●	●	
Allows Derived Attribute Value Data Types		●	●	
Conditional Realtionships & Tests			❍	●
Concise & Easy to Read	●	❍		●
Object Oriented Approach		●		
Widespread Adoption	●	●		
Tools Available	●	●	❍	❍

❍ Partially Meets Requirement
● Fully Meets Requirement

XML Validation Spectrum

- Degree of validation may depend on process requirements or other criteria
  - Timing
  - Business Requirements / Policy
  - Feasibility
  - Roles
- Different Tools and Schema Types can be applied for varying degrees of validation



XML Schema Structure

- Schema Definitions are XML Documents called "Modules"
- Top Level Element is <xs:schema>
- Most Structure will be Defined Using:

```
<xs:element>
<xs:group>
<xs:attribute>
<xs:attributeGroup>
<xs:simpleType>
<xs:complexType>
<xs:include>
<xs:ignore>
<xs:annotation>
```

Referencing Schemas in Documents

No Namespace Association Example

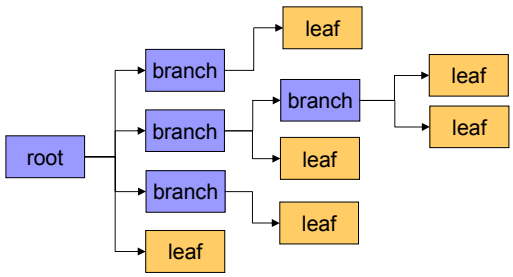
```
<?xml version='1.0' ?>
  <contact
     xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
     xsi:noNamespaceSchemaLocation='contact.xsd' >
        <name>Dale Waldt</name>
        <title>Director</title>
        <email>dale@aXtiveminds.com</email>
  </contact>
```


Simple Element Definitions

- Simple Definitions can contain only content
  - Similar to content only elements in DTDs

    ```
    <!ELEMENT foo  (#PCDATA) >
    ```

  - Represent terminal node elements
    - "Leaves" at end of "branches" of document "tree" structure
    - Can be at any level accept root



Simple Element Declaration

- Example:

```
<xs:element name='person' type='xs:string'/>
```

  - **Uses "xs:element" element to declare an element named "person" (Say that three times really fast!)**
  - **Name attribute defines name of element (Are you confused yet?)**
  - **Type attribute defines value type**

Valid:

```
<person>Alfred E. Neuman</person>
```


Complex Element Declaration

- Example of anonymous <xs:complexType>:

```
<xs:element name="memo">
 <xs:complexType>
  <xs:sequence>
   <xs:element name="to" type="xs:string"/>
   <xs:element name="from" type="xs:string"/>
   <xs:element name="p" type="xs:string"/>
  </xs:sequence>
 </xs:complexType>
</xs:element>
```

Valid:

```
<memo>
  <to>Alfred E. Neuman</to>
  <from>Mel Haney</from>
  <p>Wipe that silly grin off your face!</p>
</memo>
```


Nesting <xs:element> Declarations

- Nesting Element Definitions implies data model hierarchy

```
<xs:element name="memo">
 <xs:complexType>
  <xs:sequence>
   <xs:element name="to">
    <xs:complexType>
     <xs:sequence>
      <xs:element name="firstName" type="xs:string"/>
      <xs:element name="lastName" type="xs:string"/>
     </xs:sequence>
    </xs:complexType>
   </xs:element>
   <xs:element name="from" type="xs:string"/>
   <xs:element name="p" type="xs:string"/>
  </xs:sequence>
 </xs:complexType>
</xs:element>
```


Compositors: all, choice, sequence

- Used inside content models
  - <xs:choice>
    - Must choose one item only from the group
    - DTD syntax: "|"
  - <xs:sequence>
    - Items must appear in the order of their sequence
    - Default occurrence is one of each
    - DTD syntax: ","
  - <xs:all>
    - All must appear
    - Can appear in any order
    - Default occurrence is 1

Compositors: xs:choice & xs:sequence

```
<xs:element name='invoice'>
 <xs:complexType>
  <xs:choice>
      <xs:element name="shipAndBill"     type="shipAndBill"/>
      <xs:element name="singleUSAddress" type="USAddress"/>
    </xs:choice>
 </xs:complexType>
</xs:element>
```

```
<xs:complexType name="shipAndBill">
  <xs:sequence>
    <xs:element name="shipTo" type="USAddress"/>
    <xs:element name="billTo" type="USAddress"/>
  </xs:sequence>
</xs:complexType>
```


Compositor: xs:all

```
<xs:complexType name="date">
 <xs:all>
  <xs:element name="d" type="xs:string "/>
  <xs:element name="m" type="xs:string "/>
  <xs:element name="y" type="xs:string"/>
 </xs:all>
</xs:complexType>
```

```
<xs:complexType name="nameSet">
  <xs:all>
    <xs:element name="first" type="xs:string"/>
    <xs:element name="middle" type="xs:string"/>
    <xs:element name="last" type="xs:string"/>
  </xs:all>
</xs:complexType>
```


Specifying Occurrence

```
<xs:element name="item" type="xs:string" minOccurs="0"
  maxOccurs="unbounded">
```

- Use attributes to state how many times an element may appear
  - minOccurs

    Value is non-negative integer (0, 1, 2, 3...)
  - maxOccurs

    Value is non-negative integer (0, 1, 2, 3...) or "unbounded"
- These attributes can be used on the following elements:

```
xs:element
xs:any
xs:choice
xs:sequence
xs:group
```

```
<xs:element name="name" type="xs:string"
      minOccurs='3' maxOccurs='9'/>
```


Reusable Groups of Elements

Use <xs:group> to create a group of element declarations that can be referenced elsewhere in the schema:

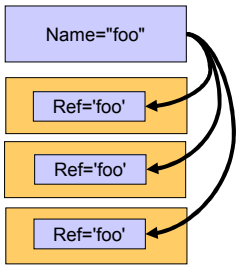
```
<xs:group name="shipAndBill" minOccurs='3' maxOccurs='9'/> >
  <xs:sequence>
    <xs:element name="shipTo" type="USAddress" />
    <xs:element name="billTo" type="USAddress" />
  </xs:sequence>
</xs:group>

<xs:complexType name="PurchaseOrderType">
  <xs:choice>
   <xs:group   ref="shipAndBill" />
   <xs:element name="singleUSAddress" type="USAddress" />
  </xs:choice>
</xs:complexType>
```


Element Scope

- Local Definitions
  - Elements are scoped when defined in context
  - Elements cannot be used outside of scope where they are defined
  - Elements defined in a <xs:sequence> exist only in that context
  - Elements of the same name that are defined in different context may have different content models
- Global Definitions
  - Elements can be declared once and used many times
  - Can simplify Schema
  - This parallels the way that element types are declared and referenced in DTDs

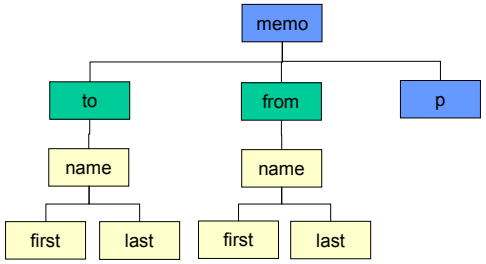
Global Definitions Can be Reused

- Global Definitions are reusable throughout the Schema Module
  - As opposed to local definitions only being usable in the context where they were defined
  - Global definitions are sort of like Parameter Entities in DTD syntax



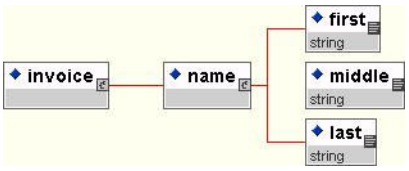
Global Elements

```
<xs:element name="name">
 <xs:complexType>
  <xs:sequence>
   <xs:element name="first" type="xs:string"/>
   <xs:element name="last" type="xs:string"/>
  </xs:sequence>
 </xs:complexType>
</xs:element>

<xs:element name="to">
 <xs:complexType>
  <xs:sequence>
   <xs:element ref="name"/>
  </xs:sequence>
 </xs:complexType>
</xs:element>

<xs:element name="from">
 <xs:complexType>
  <xs:sequence>
   <xs:element ref="name"/>
  </xs:sequence>
 </xs:complexType>
</xs:element>
```



Anonymous ComplexType Definition

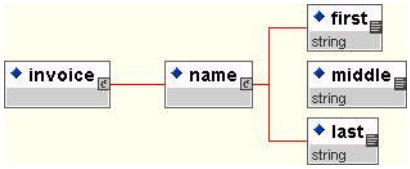
```
<xs:element name='name'>
 <xs:complexType>
  <xs:sequence>
    <xs:element name="first" type="xs:string"/>
    <xs:element name="middle" type="xs:string"/>
    <xs:element name="last" type="xs:string"/>
  </xs:sequence>
 </xs:complexType>
</xs:element>
```



Named ComplexType Definition

```
<xs:complexType name='nameSequence'>
 <xs:sequence>
   <xs:element name="first" type="xs:string"/>
   <xs:element name="middle" type="xs:string"/>
   <xs:element name="last" type="xs:string"/>
 </xs:sequence>
</xs:complexType>
```

```
<xs:element name='name' type='nameSequence'/>
```



Declaring Mixed Content in Complex Types

- Consider:

```
<salutation>Dear Mr.<name>Bob Smith</name>,</salutation>
```

- DTD syntax implicitly allowing Mixed Content:

```
<!ELEMENT salutation (#PCDATA | name)* >
```

- Schema syntax uses "mixed" attribute to explicitly allow Mixed Content:

```
<xs:element name="salutation">
  <xs:complexType mixed="true">
    <xs:choice minOccurs='0' maxOccurs='unbounded'>
      <xs:element name="name" type="xs:string"/>
    </xs:choice>
  </xs:complexType>
</xs:element>
```

Attribute Declarations

- Attributes carry additional information about the element they are associated with
- Declared using the xs:attribute element
- Declarations appear after sub-element content

```
<xs:element name='name'>
 <xs:complexType>
    <xs:sequence>
      <xs:element name="first" type="xs:string"/>
      <xs:element name="middle" type="xs:string"/>
      <xs:element name="last" type="xs:string"/>
    </xs:sequence>
    <xs:attribute name="prefix" type="xs:string"/>
    <xs:attribute name='suffix' type="xs:string"/>
  </xs:complexType>
</xs:element>

<name prefix='Mr.' suffix='Esquire'>
  <first>Dale</first>
  <middle>Charles</middle>
  <last>Waldt</last>
</name>
```


Attributes on Attribute Declarations

Name	Specifies a value to be used as the name of the attribute
Ref	Specifies previously declared attribute name, inherits the attribute properties
Use	Specifies whether the attribute is required or optional ▪ Possible values for **use=** **prohibited** **optional** **required**
Type	Specifies a primitive or derived type applied to the attribute value
Default	Specifies a value to which the attribute is defaulted
Fixed	Specifies a value to which the attribute is fixed
Form	Specifies whether attribute is namespace qualified ▪ Possible values for **form=** **qualified** **unqualified**

Empty Elements with Attributes

Elements can be empty and only have attributes associated with them

```
<xs:element name="graphic">
  <xs:complexType>
    <xs:attribute name="height" type="xs:decimal"/>
    <xs:attribute name="width" type="xs:decimal"/>
    <xs:attribute name="fileName" type="xs:string"/>
  </xs:complexType>
</xs:element>
```

```
<graphic height='120' width='80' filename='dog.jpg'/>
```


Attributes with Default Values

- Use of default= sets the attribute value to a single default value when no other value is specified
  - Processor will "insert" default value
  - Given:

```
<xs:attribute name="counter" use='optional'
              type="xs:int" default="1"/>
```

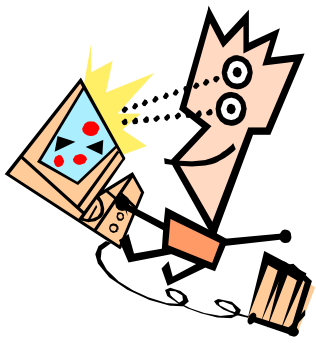
  - Validator will interpret all "counter" attributes to have a value of the integer "1" if no other value is provided
  - May or may not appear in data, but will contain the declared default value if it does not appear

Attribute Groups

- Just as elements can be declared globally in groups, so can attributes

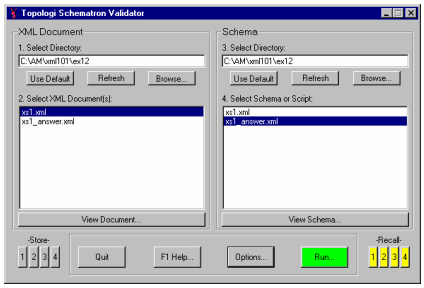
```
<xs:element name="graphic">
  <xs:complexType>
    <xs:attributeGroup ref="graphicAttrs"/>
  </xs:complexType>
</xs:element>

<xs:attributeGroup name="graphicAttrs">
  <xs:attribute name="height" type="xs:decimal"/>
  <xs:attribute name="width" type="xs:decimal"/>
  <xs:attribute name="fileName" type="xs:string"/>
</xs:attributeGroup>
```

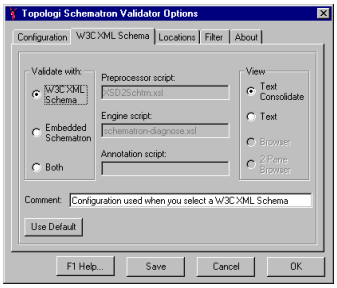

Schema Validation Demo



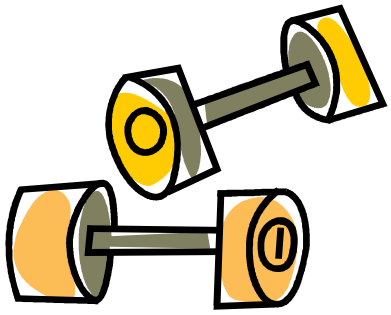
New Exercise Tool

- Topologi Schematron Validator
  - Freeware XML, Schema & RELAX-NG parser environment
  - Left window selects XML document or DTD to validate
  - Right window is for selecting associated **XML Schema Module** only
  - Run by pressing the big green **Run** button
  - Default directory should be set to c:/xml101/ex12 for both the left and right windows
  - When processing using a schema, the options need to be set to allow an associated schema module
    - Under Options select the W3C XML Schema tab and check the Validate with W3C XML Schema box

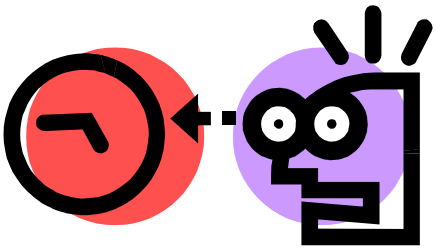




Exercises - Module 7



End of Day 1



8. Intro to Schema Data Types

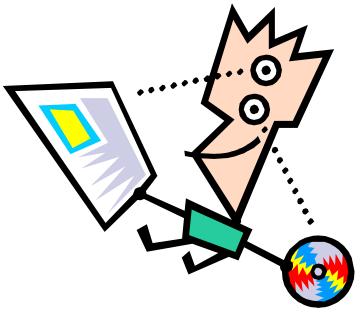
Objectives

- Understand Built-in & Derived Datatypes in Schema
- Understand Regular Expressions
- Understand Schema Pattern Definitions
- Reinforce with Exercises

Common Business Tools Use Data Types

- Spread Sheets
- Databases
- Forms software
- Software applications

- We Work with Data Types Every Day
  - Some Common Patterns



999-99-99	dale@aXtiveminds.com
1-585-703-8440	www.w3.net
02/02/2002	123
14534-9775	3.14
NY	$1,234.56
12:30	45 AmJur 395

Built-in data types

- There are 44 simple primitive data types or built-in data types that are not defined in terms of other types (they are axiomatic)

Category	Built-in Types
strings & names	`string, normalizedString, token, Name, NCName, Qname, language`
numeric	`float, double, decimal, integer, long, int, short, byte, positiveInteger, nonPositiveInteger, negativeInteger, nonNegativeInteger, unsignedLong, unsignedInt, unsignedShort, unsignedByte`
date & time	`duration, dateTime, date, time, gyear, gYearMonth, gMonth, gMonthDay, gDay`
legacy types	`ID, IDREF, IDREFS, ENTITY, ENTITIES, NMTOKEN, NMTOKENS, NOTATION`
other	`boolean, hexBinary, base64Binary, anyURI`

Primitive Simple Data Types Examples

string	"Hello World"
boolean	{true, false, 1, 0}
float	12.33E5, 12, 12560, 0, -0
decimal	7.08, 0, 1000.00
anyURI	http://aXtiveminds.com
NOTATION	notation
hexBinary	0FB7
date	2002-02-02
gYear	2002

Deriving Types with Facets

- A Simple Type can optionally have a set of "Facets" that characterize properties of the Value Space
  - All Facets are Optional
  - All Facets have a Value Attribute that specifies the value of the Facet

Category	Facets
bounds	`minInclusive, maxInclusive, minExclusive, MaxExclusive`
length	`length, minLength, maxLength`
precision	`totalDigits, fractionDigits`
enumerated values	`enumeration`
pattern matching	`pattern`
whitespace processing	`whitespace`

Range Checking Example

```
<xs:element name='month'>
 <xs:simpleType>
  <xs:restriction base="xs:positiveInteger">
   <xs:minInclusive value="1"/>
   <xs:maxInclusive value="12"/>
  </xs:restriction>
 </xs:simpleType>
</xs:element>
```

Valid:

```
<month>6</month>
```

Invalid:

```
<month>0</month>
<month>223</month>
<month>-6</month>
```

Allowed Values:

```
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
```


Enumerated Type Example

```
<xs:element name='day'>
 <xs:simpleType>
    <xs:restriction base="xs:string">
     <xs:enumeration value="Sunday"    />
     <xs:enumeration value="Monday"    />
     <xs:enumeration value="Tuesday"   />
     <xs:enumeration value="Wednesday"/>
     <xs:enumeration value="Thursday"  />
     <xs:enumeration value="Friday"    />
     <xs:enumeration value="Saturday"  />
    </xs:restriction>
   </xs:simpleType>
</xs:element>
```

Valid:

```
<day>Sunday</day>
```

Invalid:

```
<day>Domingo</day>
```

Only the names of the days of the week as defined are allowed.

Length Limit Example

```
<xs:simpleType name="fileName">
 <xs:restriction base="xs:string">
  <xs:minLength value="1"   />
  <xs:maxLength value="25" />
 </xs:restriction>
</xs:simpleType>
```

Valid:

```
<fileName>../dtdmodules/tables.dtd</fileName>
```

Invalid:

```
<fileName>/dtdmodules/aXtiveminds/tables.dtd</fileName>
<fileName></fileName>
```

Only values that have 1 to 25 characters are valid

Derived Type Examples

Simple Positive Integer Type

```
<xs:simpleType name='oneToTenType'>
  <xs:restriction base='xs:positiveInteger'>
     <xs:minInclusive value='1'/>
     <xs:maxInclusive value='10'/>
  </xs:restriction>
</xs:simpleType>
```

Allows

```
1, 2, 3, 4,
5, 6, 7, 8,
9, or 10 only
```

Pattern Type

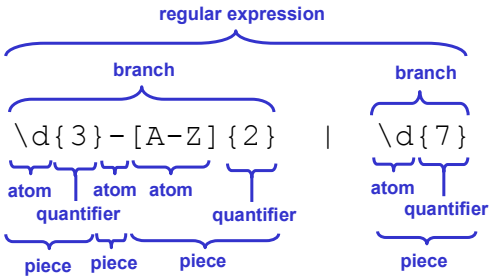
```
<xs:simpleType name='productCode'>
  <xs:restriction base='xs:string'>
    <xs:pattern value='[A-Z]{3}-\d{3}'/>
    <xs:pattern value='\d{3}-[A-Z]{3}'/>
  </xs:restriction>
</xs:simpleType>
```

Allows

```
'ABC-123' or
'987-XYZ'
```


Structure of Regular Expressions

- Regular expressions may contain
  - Branches, which may contain
    - Pieces, which may contain
      - Atoms and Quantifiers



Pattern Example

```
<xs:element name="courseNumber">
 <xs:simpleType>
  <xs:restriction base="xs:string">
   <xs:pattern value="[A-Z]{3,4}-\d{3}" />
   <xs:pattern value="[A-Z]{3,4}-\d{3}[a-z]{1}" />
  </xs:restriction>
 </xs:simpleType>
</xs:element>
```

Valid:

```
<courseNumber>XML-101</courseNumber>
<courseNumber>XML-102</courseNumber>
<courseNumber>XSLT-101e</courseNumber>
```

Invalid:

```
<courseNumber>XML-101, XML-102</courseNumber>
<courseNumber>XSchema-101</courseNumber>
<courseNumber>XML-101E</courseNumber>
```

Extension Example

```
<xs:complexType name="ParaType">
  <xs:simpleContent>
    <xs:extension base='xs:string'>
      <xs:attribute name='label' type="xs:string"
            use='required'/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>

<xs:element name='para' type='ParaType'/>

<para label='abc'>Paragraph string text.</para>
```


Another Extension Example

```
<xs:complexType name="nameType">
  <xs:sequence>
   <xs:element name='fn' type='xs:string'/>
   <xs:element name='ln' type='xs:string'/>
  </xs:sequence>
</xs:complexType>

<xs:element name='name' type='nameType'/>

<name><fn>Dale</fn><ln>Waldt</ln></name>
```

```
<xs:complexType name="extendedNameType">
  <xs:extension base='nameType'>
    <xs:sequence>
       <xs:element name='gen' type='xs:string'/>
    </xs:sequence>
  </xs:extension>
</xs:complexType>

<xs:element name='person' type='extendedNameType'/>

<person><fn>Jim</fn><ln>Waldt</ln><gen>Jr</gen></person>
```


Resolved Extension Example

```
<xs:complexType name="ExtendedNameType">
 <xs:sequence>
  <xs:sequence>
   <xs:element name='fn' type='xs:string'/>
   <xs:element name='ln' type='xs:string'/>
  </xs:sequence>
  <xs:sequence>
    <xs:element name='gen' type='xs:string'/>
  </xs:sequence>
 </xs:sequence>
</xs:complexType>

<xs:element name='person' type='extendedNameType'/>


<person><fn>James</fn><ln>Waldt</ln><gen>Jr</gen></person>
```


Example Without Structure Restriction

```
<xs:complexType name="basePerson">
  <xs:sequence>
   <xs:element name='fname' type='xs:string'/>
   <xs:element name='lname' type='xs:string'/>
   <xs:element name='age' type='xs:integer'/>
  </xs:sequence>
  <xs:attribute name='title' type='xs:string' use='optional'/>
</xs:complexType>

<xs:element name='who' type='basePerson'/>
```

```
<who title='Mr.'>
<fname>Dale</fname>
<lname>Waldt</lname>
<age>41</age>
</who>
```


Prohibiting Attribute Example

```
<xs:complexType name="simplerPerson">
 <xs:complexContent>
  <xs:restriction base='basePerson'>
    <xs:sequence>
     <xs:element name='fname' type='xs:string'/>
     <xs:element name='lname' type='xs:string'/>
     <xs:element name='age' type='xs:integer'/>
    </xs:sequence>
    <xs:attribute name='title' use='prohibited'/>
  </xs:restriction>
 </xs:complexContent>
</xs:complexType>

<xs:element name='who' type='simplerPerson'/>
```

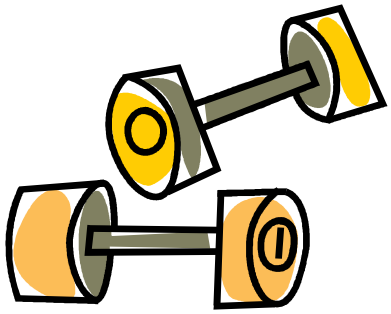
```
<who>
<fname>Dale</fname>
<lname>Waldt</lname>
<age>41</age>
</who>
```


Example With Structure Restriction

```
<xs:complexType name="simplerPerson">
 <xs:complexContent>
  <xs:restriction base='basePerson'>
    <xs:sequence>
     <xs:element name='name' type='xs:string'/>
    </xs:sequence>
  </xs:restriction>
 </xs:complexContent>
</xs:complexType>

<xs:element name='who' type='simplerPerson'/>
```

```
<who>
<name>Dale Waldt</name>
</who>
```


Exercises - Module 8



9. Schema Organization

Objectives

- Understand Schema Organization & Modularization
- Understand Schema Annotations

Design Models

- Schema elements & attributes can be defined and organized several different ways
  - Global vs. local declarations
  - New declaration vs. referring to existing declarations
  - Anonymous vs. names complex or simple types
- This allows Schemas to be organized using several different design models
  - Russian Doll Design
  - Salami Slice Design
  - Venetian Blind Model

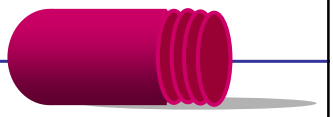
Russian Doll Design

- Mirrors the structure of the instance
  - Element & attribute declarations contained inside other element declarations
  - Child element & attributes are opaque to other schema components since they are not defined globally
  - They have localized scope
    - Different models can have the same names in different scopes
    - They are decoupled & changes will not affect others outside of their scope
    - May cause "name-in-context" confusion
  - Become very deep very quickly
  - May have redundancies in declarations
  - Difficult to manage for large complex models,
  - Fairly easy to interpret

Russian Doll Design Example

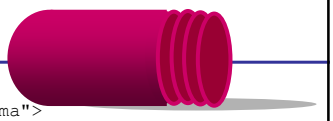
```
<xs:schema>
 <xs:element name='newsletter'>
  <xs:element name="article">
   <xs:complexType>
    <xs:sequence>
     <xs:element name="articletitle" minOccurs='1' maxOccurs="1">
      <xs:attribute name="uid" type="xs:string" use="required"/>
     </xs:element>
     <xs:element name="author" minOccurs="0" maxOccurs="3"/>
     <xs:element name="datepub" type="xs:date"  maxOccurs="1"/>
     <xs:element name="body" maxOccurs="1"/>
     <xs:element name='para'/>
    </xs:sequence>
    <xs:attribute name="issn" type="xs:string" use="required"/>
    <xs:attribute name="edition" type="xs:string" use="required"/>
   </xs:complexType>
  </xs:element>
 </xs:element>
</xs:schema>
```


Salami Slice Design



- Opposite of Russian Doll Design
  - All elements & attributes are declared globally
- Child elements and attributes are transparent since they can be referenced from other complex types
- Have global scope
  - Two global element declarations cannot be declared with the same name
  - **Any global element can serve as a root in an instance!**
- They are considered coupled
  - Change to a declaration will affect any complex type that references it

Salami Slice Design Example



```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="article">
  <xs:complexType>
    <xs:sequence>

   <xs:element ref="title"/>

   <xs:element ref="author"/>

   <xs:element ref="para"/>

    </xs:sequence>
  </xs:complexType>
 </xs:element>


<xs:element name="title" type="xs:string"/>

<xs:element name="author" type="xs:string"/>

<xs:element name="para" type="xs:string"/>


</xs:schema>
```


Venetian Blind Design

- Combination of Russian Doll and Salami Slice Designs
- Encourages use of named complex types rather than just element declarations
  - Allows reuse of named types
  - Simplifies reuse & maintenance
- Similar to defining & using classes in object-oriented programming languages

"Venetian Blind" Hybrid Design Example

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

 <xs:element name='newsletter'>
  <xs:complexType>
   <xs:choice>
    <xs:element ref='article'/>
   </xs:choice>
  </xs:complexType>
 </xs:element>

 <xs:element name="article" type="articleContent"/>

 <xs:complexType name='articleContent'>
    <xs:sequence>
     <xs:element name="articletitle" minOccurs='1' maxOccurs="1"/>
     <xs:element name="author" minOccurs="0" maxOccurs="3"/>
     <xs:element name="revdate">
      <xs:complexType>
       <xs:all>
        <xs:element name="mm" minOccurs="0"/>
        <xs:element name="yy" type="xs:integer"/>
       </xs:all>
      </xs:complexType>
     </xs:element>
     <xs:element name='para' maxOccurs="unbounded"/>
    </xs:sequence>
 </xs:complexType>
 </xs:schema>
```


Modular Schemas

- When schemas become large it is often desirable to divide their content among several schema modules to simplify overview and maintenance
- Three ways of managing schema modules
  - <xs:include>
  - <xs:redefine>
  - <xs:import>
- These elements must appear before all element declarations and type definitions in the schema

<xs:include>

- <xs:include> brings in definitions and declarations contained in external schema modules

```
<xs:include schemaLocation="includedSchema.xsd">
```

- The target namespace of the included components must be the same as the target namespace of the including schema module
- The included components can also be declared in a schema module without target namespace
  - The included components are then added to the target namespace of the including schema (Cameleon effect)

<xs:include> example

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
   targetNamespace="www.allette.com.au/Person"
   xmlns:p="www.allette.com.au/Person"
   elementFormDefault="qualified">

 <xs:include schemaLocation="PersonAddress.xsd"/>

 <xs:complexType name="Person">
  <xs:sequence>
   <xs:element name="Name" type="xs:string"/>
   <xs:element name="Address" type="p:Address"/>
  </xs:sequence>
 </xs:complexType>
</xs:schema>
```


<xs:import>

- The <xs:import> element enables schema components from different target namespaces to be used together
- The import mechanism identifies the namespace that is imported to your schema using the *namespace* attribute of the element <xs:import>
- Each namespace to be imported must be identified with a separate <xs:import> element
- The <xs:import> elements must appear as the first children of the schema element

<xs:import> (cont'd)

- Each imported namespace must be associated with a prefix using a standard namespace declaration

```
xmlns:???="imported namespace URI"
```

- This prefix is used to qualify references to any component belonging to that namespace
- Import elements optionally contain a *schemaLocation* attribute to help locate resources associated with the namespaces

```
<xs:import namespace="http://www.import.org"
     schemaLocation="importedSchema.xsd"/>
```


<xs:import> Example

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
   targetNamespace="www.allette.com.au/Person"
   xmlns:p="www.allette.com.au/Person"
   xmlns:a="www.allette.com.au/Address"
   elementFormDefault="qualified">
 <xs:import namespace="www.allette.com.au/Address"
     schemaLocation="Address.xsd"/>

 <xs:complexType name="PersonType">
  <xs:sequence>
   <xs:element name="Name" type="xs:string"/>
   <xs:element name="Address" type="a:Address"/>
  </xs:sequence>
 </xs:complexType>
 <xs:element name="Person" type="p:PersonType"/>
</xs:schema>
```


Annotations

- Schema provides a formal mechanism for adding comments & documentation to your schema
- The <annotation> element contains either
  - A <documentation> element
    - Recommended location for human readable material
  - Or an <appinfo> element
    - can be used to provide information for tools, stylesheets and other applications
  - **NOTE:** Annotation sub-elements can contain markup that would not be processed as part of the target namespace

Annotation Example

```
<xs:element name="internationalPrice">
  <xs:annotation>
    <xs:documentation>Int. Price in USD</xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:annotation>
      <xs:documentation>Empty element type</xs:documentation>
    </xs:annotation>
    <xs:complexContent>
      <xs:restriction base="xs:anyType">
        <xs:attribute name="currency" type="xs:string" />
        <xs:attribute name="value"    type="xs:decimal" />
      </xs:restriction>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```


xs:anyType

- anyType represents an abstraction of the ur-type which is the base type from which all other types are derived
  - Does not constrain its content in any way
    - Can be text only, element only, or mixed
  - The anyType is the default if no type is specified
    - <xs:element name='anything'/>
  - A complexType without <xs:simpleContent> or <xs:complexContent> is shorthand for complex content that restricts anyType

Wildcard Example Preventing HTML Validation

```
<xs:element name='HTMLExample'>
 <xs:complexType mixed='true'>
  <xs:sequence>
   <xs:any minOccurs='0' maxOccurs='unbounded' processContents='skip'/>
   </xs:sequence>
   <xs:anyAttribute processContents='skip'/>
 </xs:complexType>
</xs:element>
```

```
<HTMLExample href='http://www.w3.org'>
  <tr>
    <th align='left'>Table Head</th>
  </tr>
</HTMLExample>
```


Wildcards

- Wildcards allow flexible content models
- Elements <xs:any> and <xs:anyAttribute> are called wildcards
  - The number of Attributes cannot be constrained
- Wildcards allow any well formed XML
- Validation of wildcards content be controlled with the processContents attribute
  - Strict
    - Validate against the schema of the assigned namespace (default)
    - The Namespace attribute specifies which Namespace the elements & attributes should belong to
      - ##any Any Namespace (default)
      - ##local Not qualified (not declared in any namespace)
      - ##other Not from the Target Namespace of the schema
      - "http://www.w3.org/1999/xhtml ##targetNamepsace" Any Namespace in list
  - Lax
    - Validate the elements and attributes for which a schema can be found
  - Skip
    - No validation

Substitution Groups

- Any globally declared element can serve as a model for other elements
  - This type of element is called a Head Element
- Elements that are modeled on a Head Element are said to be members of a Substitution Group
- Wherever the head element is used in a content model, members of the substitution can also be used and appear in the instance
- The Type of a member of a substitution group must be the same as the type of the head element
- If the type is the same, the type attribute of the <xs:element can be omitted

Substitution Group Example

```
<xs:element name="para" type='xs:string'/>
<xs:element name="subpara" substitutionGroup='para'/>

<xs:element name='chapter'>
  <xs:complexType>
    <xs:sequence>
     <xs:element ref='para' maxOccurs='unbounded'/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

```
<chapter>
<subpara>This subpara is substituting for para.</subpara>
<para>This para is a para!</para>
</chapter>
```

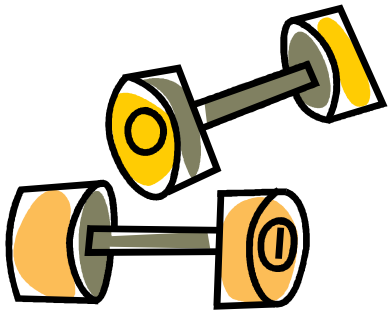

Abstract Elements

- Substitution for a class of elements can be forced
  - The head element is defined as an Abstract element instead
  - It will not appear in the text instance
  - It must only be used indirectly using a substitution group attribute
  - Members of the substitution group may appear in text

```
<xs:element name='newsletter'>
 <xs:complexType>
  <xs:sequence>
   <xs:element ref='articles' maxOccurs='unbounded'/>
  </xs:sequence>
 </xs:complexType>
</xs:element>
<xs:element name='articles' type='xs:string' abstract='true'/>
<xs:element name='news' substitutionGroup='articles'/>
<xs:element name='feature' substitutionGroup='articles'/>
```

```
<newsletter>
  <feature/>
  <news/>
</newsletter>
```


Exercises - Module 9



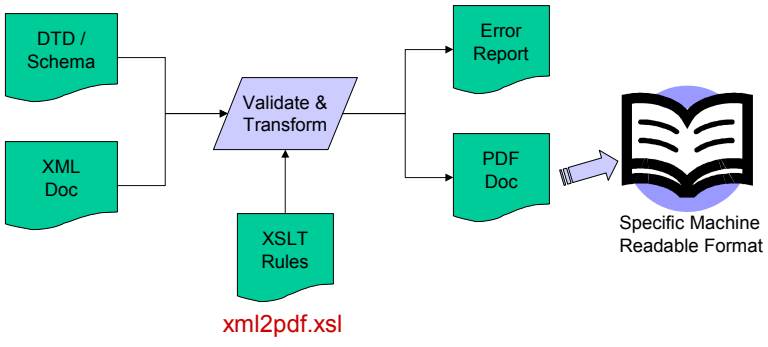
10. Document Rendering & Processing

Objectives

- Understand how XML turns into Print & electronic formats

XML Transformation

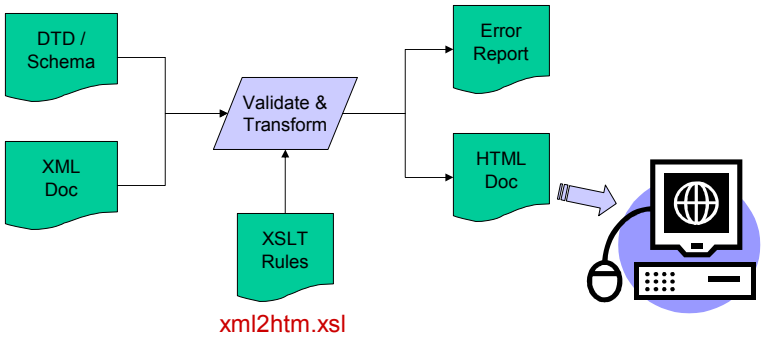
- Transformation of content using "XML to PDF" rules



xml2pdf.xsl

XML Transformation

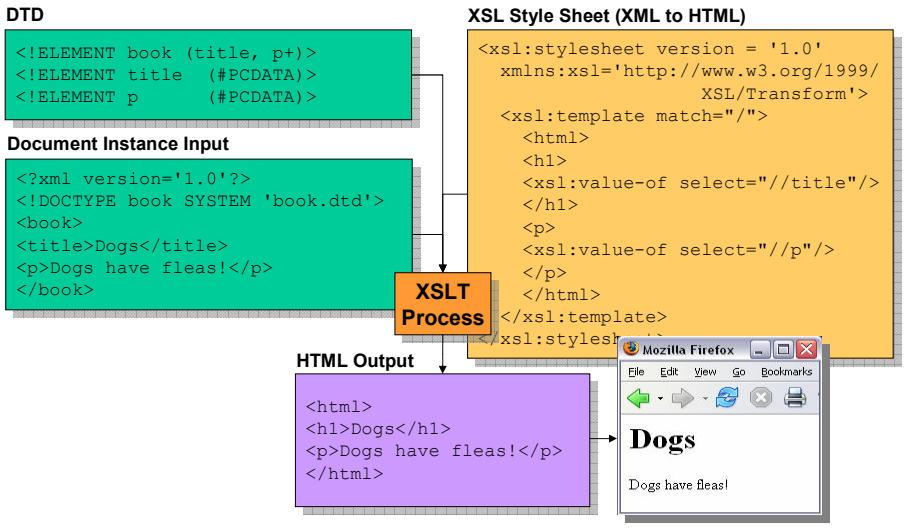
- Transformation of content using "XML to HTML" rules



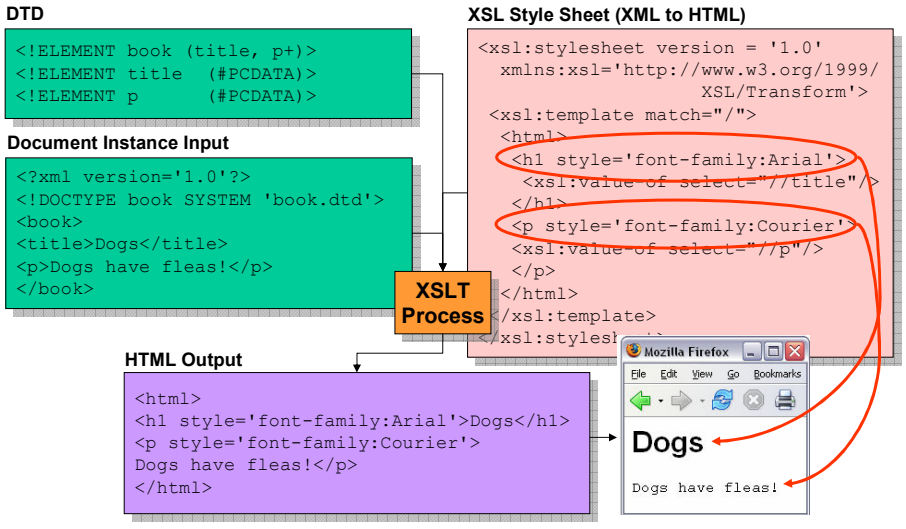
What is XSLT?

- Extensible Stylesheet Language Transformation
- W3C official recommendation
  - Version 1.0, 16 November 1999
- A programming language for transforming XML documents
- XSLT transformation is expressed in XML syntax
- Separates data from presentation

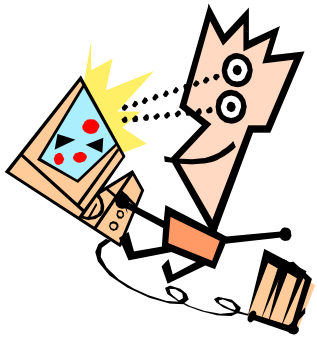
XSL Style Transformation



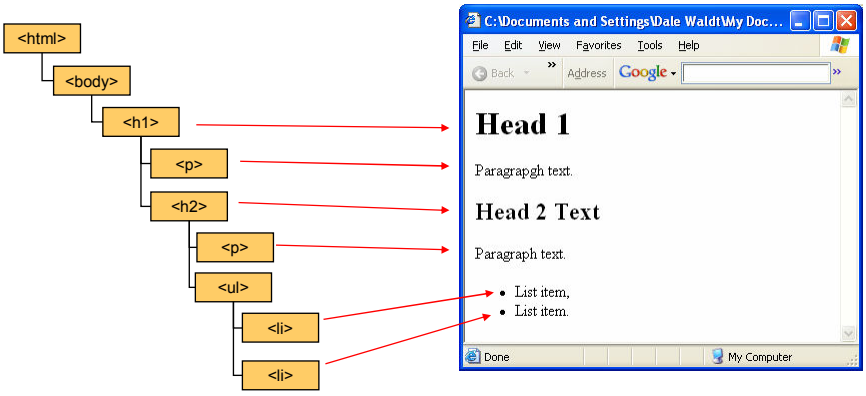
Same Data Different Style Sheet



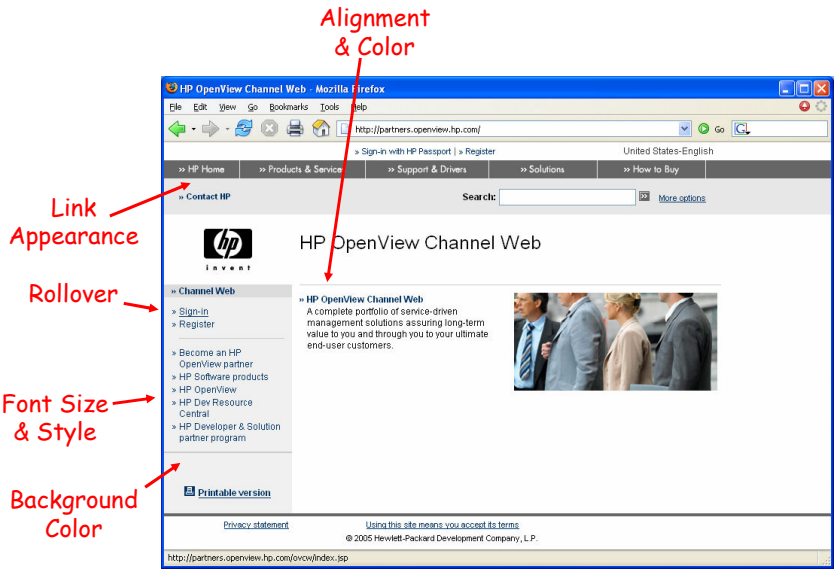
Transformation Demo



Simple HTML Document / Default Rendering



Complex Web Page Formatting

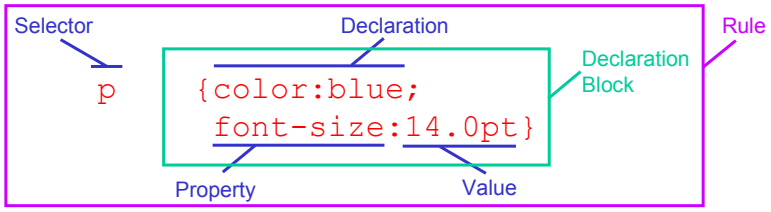


Cascading Style Sheets

- Override the browser's default settings for formatting
- Contain rules, composed of *selectors* & *declarations* that define how styles will be applied
  - Font
  - Color
  - Margins
  - Background
  - Etc.
- Provide consistency of look & feel

CSS Basics

- Understanding Style Rules
  - Style Rules are composed of two parts:
    - Selector → determines element to which rule is applied
    - Declaration → details the exact property values
    - Rule may contain one or more declarations, each which have two parts also:
      - Property → quality or characteristic (e.g., color)
      - Value → precise specification of property (e.g., blue)



Style Location Options

- Style declarations can reside:
  - Inline using a `style=` attribute
    - scope is local to the element modified
  - Embedded within HTML document using the `<style>` … `</style>` element
    - Scope is global to the whole document
  - Externally in separate Style Sheet document, typically with a `.css` extension
    - HTML document provides a *link* to the external CSS, or
    - *Imports* the CSS into the HTML

Inline Style - Example

- Using a **`style=`** attribute in an element:

```
<p style = "font-size:14pt">This paragraph will be
rendered in 14 point font size</p>
```

  - The inline **`style`** attribute allows specification of a style for the element it modifies
  - Each inline *CSS property* (e.g., **`font-size`**) is followed by a colon and the argument, or value for that property – *a property is like an attribute*

Inline Style - Example

- Using a **`style`** attribute with multiple properties in an element:

```
<p style = "font-size: 14pt; color: #FF0000">This
paragraph will be rendered in 14 point font size & the
text will be red in color</p>
```

  - When two or more properties are applied to the same element, the properties are separated by a semicolon
  - IMPORTANT: Inline styles override any other styles applied by the methods covered later

Style Element – Syntax

- Using a <style …> element:

```
<html>
 <head>
  <style type = "text/css">
   P    { font-size: 14pt;
          color: #FF0000 }
  </style>
 </head>
 <body>
    .
    .
    .
```

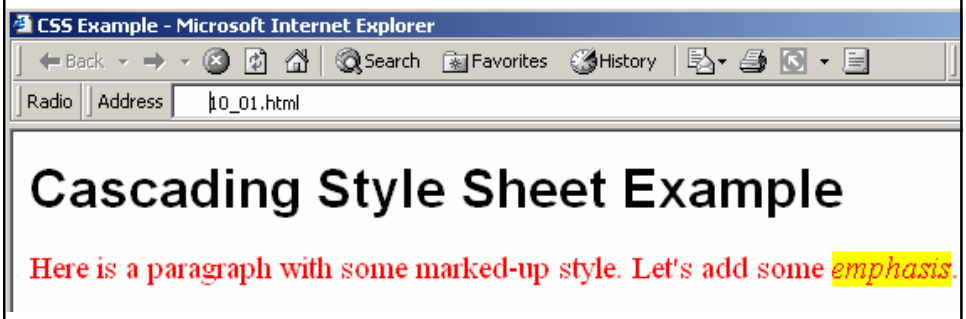
  - <style> element appears inside <head> element

Sample Embedded Style Sheet

10_01.html

```
1  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
2  <html>
3  <head>
4  <title>CSS Example</title>
5  <style type="text/css">
6
7    p    { font-size: 14pt;
8           color: #FF0000 }
9    h1   { font-family: Arial, sans-serif }
10   em   { background-color: #FFFF00 }
11
12 </style>
13 </head>
14 <body>
15 <h1>Cascading Style Sheet Example</h1>
16 <p>Here is a paragraph with some marked-up
17    style.  Let's add some <em>emphasis</em>.</p>
18 </body>
19 </html>
```


The Rendered Result



Combining CSS Rules with HTML

- External style sheet
  - Text document that contains style rules
  - Allows specification of rules for multiple HTML documents
  - Does not contain HTML code
- Linking to an external style sheet
  - <LINK> element establishes document relationships
    - Can only be used in the <HEAD> section of a document
    - Tells the browser where to find the external style sheet

Sample Linked External Style Sheet

HTML file

10_02.html

```
1  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
2           Transitional//EN">
3  <html>
4  <head>
5  <title>CSS Example</title>
6  <link rel="stylesheet" type="text/css"
7        href="style.css" />
8  </head>
9  <body>
10 <h1>Cascading Style Sheet Example</h1>
11 <p>Here is a paragraph with some marked-up
12    style.  Let's add some <em>emphasis</em>.</p>
13 </body>
14 </html>
```

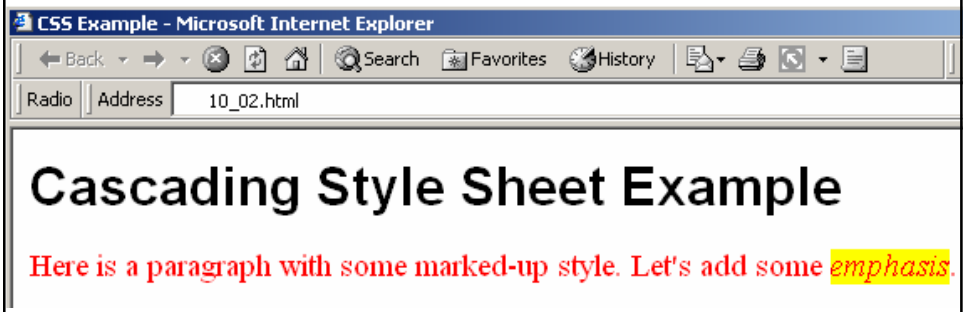
External CSS

style.css

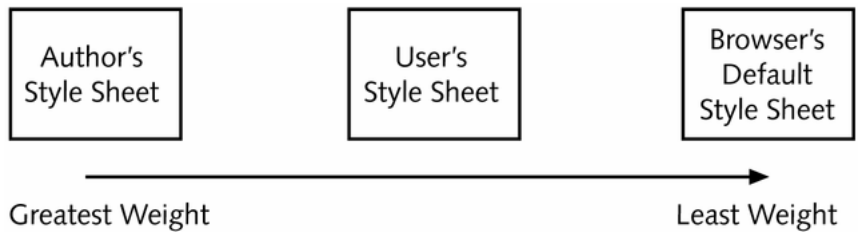
```
1  p   { font-size: 14pt;
2        color: #FF0000 }
3  h1  { font-family: Arial, sans-serif }
4  em  { background-color: #FFFF00 }
```


The Rendered Result

- The result is identical to the earlier example using an embedded CSS



The Cascading Order of Precedence



Font Properties

- Font properties are frequently formatted in CSS, some of the most frequently used properties include
  - font-family
  - font-style
  - font-size
  - font-variant
  - font-weight
  - font-stretch

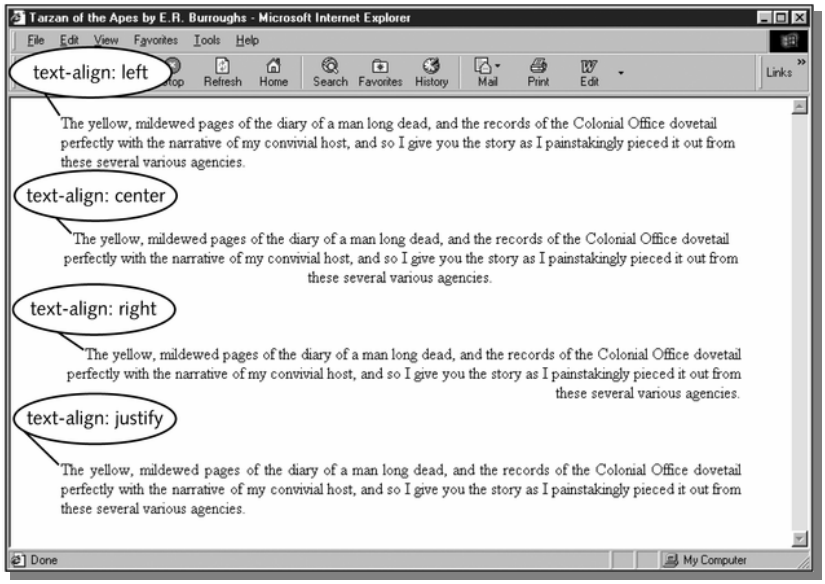
Text Properties

- Text properties cover aspects of formatting text other than what is controlled by the font, some of the most frequently used properties include
  - text-indent
    - <length> | <percentage> | inherit
  - text-align
    - left | right | center | justify | <string> | inherit
  - text-decoration
    - none | [underline || overline || line-through || blink] | inherit
  - text-transform
    - capitalize | uppercase | lowercase | none | inherit
  - white-space
    - normal | pre | nowrap | inherit

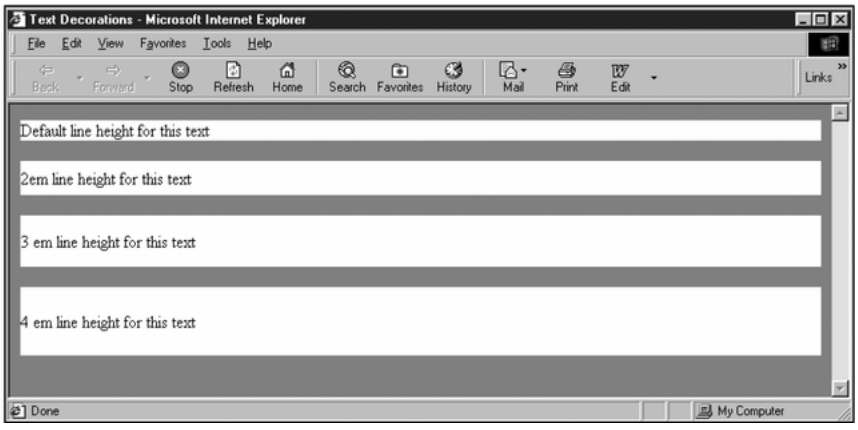
RGB Color

- hex numbers between 00 and FF
  - #FF0000 for red
- decimal numbers between 0 and 255
  - rgb(0, 255, 0) for green
- percentages
  - rgb(0%,0%,100%) for blue

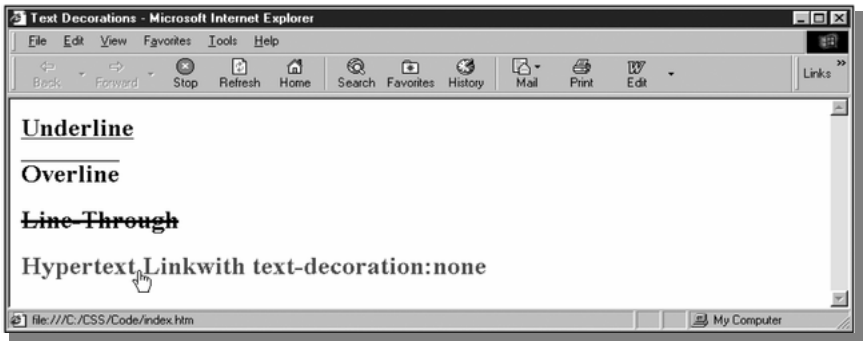
Text Alignments



Line Height



Text Decorations



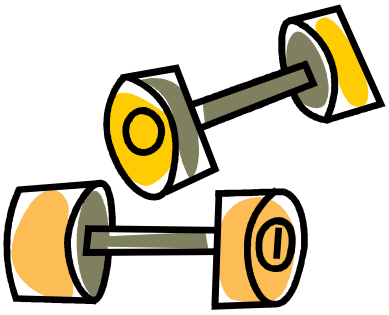
Associating CSS with XML Documents

- A stylesheet associated with an XML document using the xml-stylesheet processing instruction
- For example
  - <?xml-stylesheet type="text/css" href="style.css"?>

CSS vs. XSL for Style Processing

Feature	CSS	XSL
Can be used with HTML?	●	
Can be used with XML?	●	●
Can transform markup?		●
Can manipulate & reorganize data?		●
Extensible?	●	●
Easy to learn & code?	●	●
Skills & resources available?	●	●
Syntax used?	CSS	XML

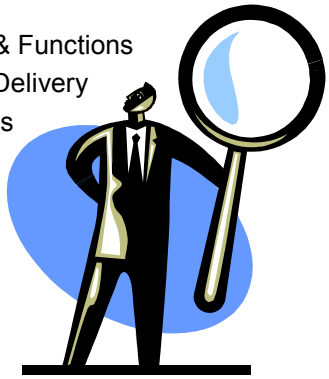
Exercises - Module 10



11. Understanding XPath & XSLT

Objectives

- Describe Role of XSLT & XPath
- Describe XSLT Style Sheets
- Describe XSLT template Styles
- Understand Nodes, Axes, Predicates & Functions
- Describe XSLT Content Rendering & Delivery
- Reinforce understanding with exercises



Basic XSL Template Structure

Stylesheet

Template

XML Content Output

Literal Output

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

  <xsl:template match="/">
    <h1>
    <xsl:value-of select="//title"/>
    </h1>
    <h2>
    <xsl:value-of select="//p"/>
    </h2>
  </xsl:template>


</xsl:stylesheet>
```

XSL Stylesheet is an XML Document

- Style Sheet must be contained within an XSL processing instruction

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


Templates

- XSL is Template-driven expression language
  - Templates associated with XML content using XPath references
  - Templates can format one or more elements, attributes or text content nodes

```
<xsl:template match="chap">
    <h1>
    <xsl:value-of select="/chap/title"/>
    </h1>
    <h2>
    <xsl:value-of select="//p"/>
    </h2>
  </xsl:template>
```

Template associated with chap element

Action tied to title within a chap

Action tied to p child element

Output Content Values

- XML Content can be output 2 ways

```
<xsl:template match="chapter">
    <h1>
    <xsl:value-of select="//title"/>
    </h1>
    <h2>
    <xsl:value-of select="//p"/>
    </h2>
  </xsl:template>
```

Or:

```
<xsl:template match="name">
  <b>
    <xsl:apply-templates select="GivenName"/>
  </b>
  <i>
    <xsl:apply-templates select="FamilyName"/>
  </i>
</xsl:template>
```


Literal result element example

```
<xsl:template match="/">
  <html>
      <head><title>Page title</title></head>
      <body><xsl:apply-templates/></body>
  </html>
</xsl:template>

<xsl:template match="item">
  <tr>
     <td><xsl:value-of select="."/></td>
  </tr>
</xsl:template>
```


XPath in XSLT Data Model

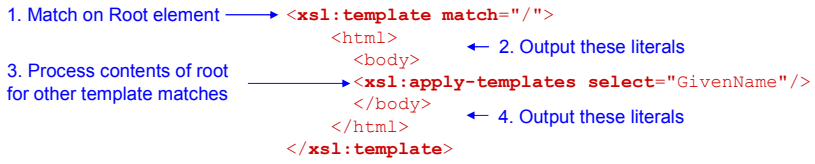
- XML Path Language
- W3C official recommendation
  - Version 1.0, 16 November 1999
- XPath is a language for addressing parts of XML document
  - Acts as a sub-language within XSLT
- XPath syntax is heavily used in XSLT

The XPath Data Model

- Every XML document in XSLT is represented as a tree of nodes
  - The XSLT tree model can represent every well-formed XML document
- Different types of nodes correspond to appropriate types of markup and data in XML documents
- Seven types of nodes:
  - Root
  - Element
  - Text
  - Attribute
  - Processing instruction
  - Comment
  - Namespace

Matching on Document Root

- A document root is the very beginning of the file
  - Expressed in XPath using the "/" node name
  - Style sheet should begin processing at the root



Stylesheet elements example

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

<xsl:output method="html" indent="yes" encoding="iso-8859-1"/>

<xsl:template match="para">
  <p><xsl:apply-templates/></p>
</xsl:template>

<xsl:template match="emph">
  <strong><xsl:value-of select="."/></strong>
</xsl:template>

</xsl:stylesheet>
```

XPath Syntax in Context of XSLT

```
<xsl:template match="chapter/title[1]">
  <h2><value-of select="@label"/></h2>
  <xsl:apply-templates/>
</xsl:template>

<xsl:template match="intro">
  <xsl:apply-templates select="//para"/>
</xsl:template>

<xsl:template match="para">
  <p><xsl:value-of select="."/><p>
</xsl:template>
```


Selecting Attributes

- Attributes are addressable nodes
  - Indicated with "@" prefix or "attribute:" prefix

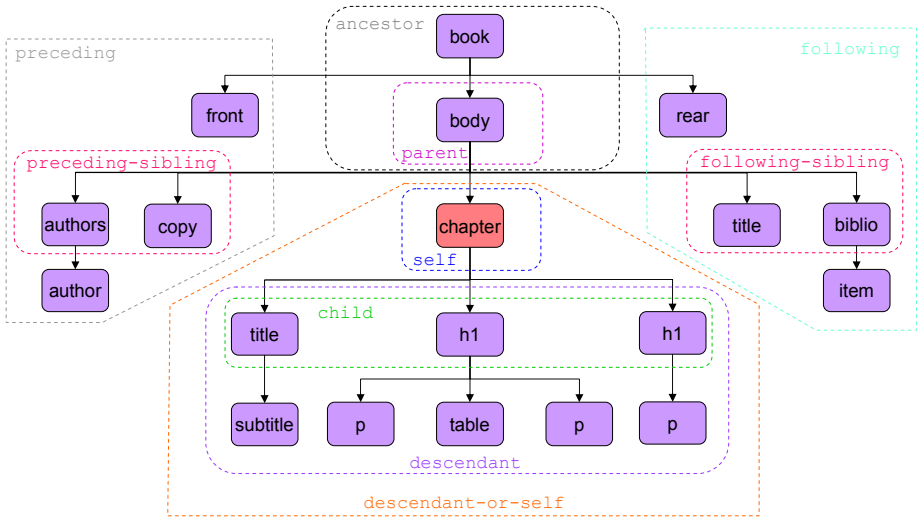
```
<xsl:template match="note">
    <p>
     <b>
         <xsl:value-of select="@security"/>:
     </b>
         <xsl:value-of select="note"/>:
   </p>
 </xsl:template>
```

Output value of security attribute

Output text value of note element

Note: if no template selects an existing attribute, its value will be surpressed on output.

Axes Refine XPath Hierarchy



Match Using an Axis

- Check to see what security attribute values exist on elements within a section
- For-each allows repeating action for all sub elements

```
<xsl:template match="section[@security=(public)]">
  <xsl:for-each select="child::*">
    <xsl:value-of select="./@security"/>
    <xsl:apply-templates/>
  </xsl:for-each>
</xsl:template>
```

Axes (cont'd)

```
ancestor::
ancestor-or-self::
attribute::
child::
descendant::
descendant-or-self::
following::
following-sibling::
namespace::
parent::
preceding::
preceding-sibling::
self::
```


Abbreviated syntax

`/`	node tree root
`.`	`self::node()`
`@`	`attribute::`
`..`	`parent::node()`
`//`	`/descendant-or-self::node()/`
omitted axis specification	`child::`
`*`	any element

Examples of Predicates

`select="para[1]"`

the first child element node named "para"

`select="para[position()=2]"`

the second child element node named "para"

`select="//para[1]"`

descendant element nodes from the root named "para" that are first among sibling "para" elements

`select="(//para)[1]"`

the first element node with the "para" in the entire document

`select="para[position()=last()]"`

the last child element node named "para"

Operators in XPath

- Comparison operators

  `<, <=, >, >=`
- Equality operators

  `=, !=`
- Unary minus operator

  `-`
- Arithmetic operators

  `*, div, mod, +, -`
- Logical operators

  `and, or, not`

Examples of Operators

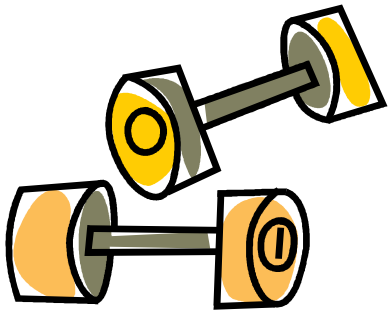
```
select="section[position()=1 or position()=3]"
```

the first or third child element node named "section"

```
select="para[@id and @type]"
```

the child element node called "para" which has attributes "id" and "type"

Exercises - Module 11



12. XSLT Processing

Objectives

- Describe XSLT Processing Capabilities
- Describe Expressions, Functions, Sorting, Variables & Operators
- Describe XSLT Data Transformation Capabilities
- Describe XML Output Types
- Describe XSLT Processing Examples
- Reinforce understanding with exercises

A Simple Count

XML Source

```
<source>
 <name>Bob</name>
 <name>Alan</name>
 <name>Tina</name>
 <name>Jane</name>
</source>
```

XSL Template

```
<xsl:template match="/">
 <P>
  <B>Number of Names:</B>
   <xsl:value-of select="count(//name)"/>
 </P>
</xsl:template>
```

HTML Output

```
<P>
 <B>Number of Names:</B> 4
</P>
```

<xsl:for-each>

- Selects a set of nodes and performs the same processing for each node in the set
- Always used in the template body
- The *select* attribute contains XPath expression which selects a set of nodes
  - The attribute is mandatory
  - XPath expression should always return a node-set
- The most common use of <xsl:for-each> is to iterate over a set of nodes
  - Can also be used to change the current node
  - May provide an alternative to <xsl:apply-templates> element
- Each iteration within <xsl:for-each> provides its own scope

<xsl:for-each> example

```
<chapter>
  <intro>Intoduction</intro>
  <para>First paragraph</para>
  <para>Second paragraph</para>
</chapter>

<xsl:template match="chapter">
  <xsl:for-each select="para">
       <p><xsl:value-of select="."/></p>
  </xsl:for-each>
</xsl:template>

<p>First paragraph</p>
<p>Second paragraph</p>
```


A Simple Sort

XML Source

```
<source>
 <name>Bob</name>
 <name>Alan</name>
 <name>Tina</name>
 <name>Jane</name>
</source>
```

HTML Output

```
<TABLE>
 <TR>
  <TD>Alan</TD>
 </TR>
 <TR>
  <TD>Bob</td>
 </TR>
 <TR>
  <TD>Jane</TD>
 </TR>
 <TR>
  <TD>Tina</TD>
 </TR>
</TABLE>
```

XSL Template

```
<xsl:template match="/">
 <TABLE>
  <xsl:for-each select="//name">
   <xsl:sort order="ascending" select="."/>
   <TR>
    <TD>
      <xsl:value-of select="."/>
    </TD>
   </TR>
  </xsl:for-each>
 </TABLE>
</xsl:template>
```


<xsl:if>

- Used when there is a need for conditional processing
- Analogous to the *if-then* statement found in many procedural programming languages
- The *test* attribute holds an XPath expression which evaluates to a boolean result
- There is no *else* alternative

<xsl:if> example

```
<chapter>
  <para id="1">First paragraph</para>
  <para id="2">Second paragraph</para>
</chapter>

<xsl:template match="para">
  <xsl:if test="@id='1'">
     <h1><xsl:value-of select="."/></h1>
  </xsl:if>
  <xsl:if test="@id='2'">
     <h2><xsl:value-of select="."/></h2>
  </xsl:if>
</xsl:template>

<h1>First paragraph</h2>
<h2>Second paragraph</h2>
```


A Simple Conditional

XML Source

```
<list>
 <entry id='10'/>
 <entry id='20'/>
 <entry id='30'/>
 <entry id='40'/>
</list>
```

XSL Template without Condition

```
<xsl:template match="list">
 <xsl:for-each select="entry">
  <xsl:value-of select="@id"/>
   <xsl:text>, </xsl:text>
 </xsl:for-each>
</xsl:template>
```

Output

```
10, 20, 30, 40,
```

XML Source

```
<list>
 <entry id='10'/>
 <entry id='20'/>
 <entry id='30'/>
 <entry id='40'/>
</list>
```

XSL Template with Condition

```
<xsl:template match="list">
 <xsl:for-each select="entry">
  <xsl:value-of select="@id"/>
  <xsl:if test="not(position()=last())">
   <xsl:text>, </xsl:text>
  </xsl:if>
 </xsl:for-each>
</xsl:template>
```

Output

```
10, 20, 30, 40
```

<xsl:choose> and <xsl:when>

- Used to define a choice between a number of alternatives
- <xsl:choose> element is always used in the template body
- <xsl:when> element is always used within <xsl:choose> element
- The *test* attribute holds an XPath expression which evaluates to a boolean result
- <xsl:otherwise> element may be used to handle cases where no <xsl:when> conditions are satisfied

<xsl:choose> and <xsl:when> example

```
<xsl:template match="graphic">
  <xsl:choose>
     <xsl:when test="@format='gif'">
     ...
     </xsl:when>
     <xsl:when test="@format='jpeg'">
     ...
     </xsl:when>
     <xsl:otherwise>
     ...
     </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```


<xsl:variable>

- Used to declare variable in a stylesheet and give a value to it
- Can appear either as a top-level element or within template body
- Can declares either local or global variables
  - The scope of a local variable is block-structured
  - The scope of a global variable is the entire stylesheet
- The value of the variable may be given either by the value of attribute *select* or by the content of the element
- Once variable is assigned its value, it cannot be changed
  - It is sometimes said that variable is bound to its value
- Variable names are referenced by prefixing name of the variable with a dollar sign

<xsl:variable> example

Declaring variable:

```
<xsl:variable name="state"/>
```

or

```
<xsl:variable name="state" select="'NY'"/>
```

or

```
<xsl:variable name="state">NY</xsl:variable>
```

Referencing variable:

```
<xsl:value-of select="$state"/>
```


A Simple Variable

XSL Template

```
<xsl:variable name="main-page-title" >XML 101</xsl:variable>

<xsl:template match = "/" >
 <HTML>
  <H1>
   <xsl:value-of select = "$main-page-title"/>
  </H1>
  <xsl:apply-templates/>
 </HTML>
</xsl:template>

<xsl:template match='p'>
 <p><xsl:apply-templates/></p>
</xsl:template>
```

XML Source

```
<book>
 <p>...</p>
</book>
```

Output

```
<HTML>
 <BODY>
  <H1>XML 101</H1>
  <p>...</p>
 </BODY>
</HTML>
```


<xsl:param>

- Used to define either global or local parameter to a template
- Can declare either local or global parameters
  - The scope of a local variable is block-structured
  - The scope of a global variable is the entire stylesheet
- May appear as a top-level element or as an immediate child of <xsl:template> element
- There is no difference between parameters and variables in XSLT except for way initial value is set
- The value of the parameter may be given either by the value of attribute *select* or by the content of the element
- Once parameter is assigned its value it cannot be changed
- Parameter names are referenced by prefixing name of the parameter with a dollar sign

<xsl:param> example

```
<xsl:template match="/">
  <xsl:for-each select="//*">
    <xsl:call-template name="depth"/>
  </xsl:for-each>
</xsl:template>

<xsl:template name="depth">
  <xsl:param name="node" select="."/>
  <node><xsl:value-of select="$node"/></node>
</xsl:template>
```


<xsl:call-template>

- Used to invoke a named template
- The effect is similar to a procedure call in other programming languages
- Always used within template body
- Can contain zero or more <xsl:with-param> elements
- The *name* attribute contains the name of template to be called
  - It is an error if there is no template with matching name for <xsl:call-template> element
- The selected template is executed without change to the context
  - Current node and current node list remain the same
- There is no direct way of returning a result from <xsl:call-template>

<xsl:with-param>

- Used to set value of parameters when calling a template
  - The value of the parameter can be used within called template
  - Does not actually declare a variable
- Always a child of either <xsl:apply-templates> element or <xsl:call-template> element
- The *name* attribute provides the name of the parameter
- The *select* attribute contains the value of the parameter to be passed to the called template
- The value assigned to <xsl:with-param> is available within template if:
  - The called template has <xsl:param> element
  - Its name matches that of <xsl:with-param> element
- The value is ignored if the called template has no parameter

<xsl:text>

- Outputs literal text to current output destination
- Can also be used for escaping special characters by using *disable-output-escaping* attribute
- Always used as a part of template body
- May contain text only or be empty
- Text appearing within a template in the stylesheet is copied to the current output in any event
  - The only direct effect of enclosing text in <xsl:text> element is that the handling of whitespace is different
  - If whitespace appears within <xsl:text> it will be preserved

<xsl:text> Example I

```
<contact>
  <fname>John</fname>
  <lname>Citizen</lname>
</contact>

<xsl:template match="contact">
  <name>
    <xsl:value-of select="fname"/>
    <xsl:text> </xsl:text>
    <xsl:value-of select="lname"/>
  </name>
</xsl:template>

<name>John Citizen</name>
```


<xsl:text> Example II

```
<contact><fname>John</fname><lname>Citizen</lname></contact>

<xsl:template match="contact">
  <xsl:apply-templates/>
</xsl:template>

<xsl:template select="fname">
  <xsl:text disable-output-escaping="yes">&lt;name&gt;</xsl:text>
  <xsl:value-of select="."/>
  <xsl:text> </xsl:text>
</xsl:template>

<xsl:template select="lname">
  <xsl:value-of select="."/>
  <xsl:text disable-output-escaping="yes">&lt;/name&gt;</xsl:text>
</xsl:template>

<name>John Citizen</name>
```


<xsl:output> Examples

```
<xsl:output method="text" encoding="us-ascii"/>

<xsl:output method="html" encoding="utf-16" indent="yes"/>

<xsl:output method="xml" encoding="iso-8859-1" indent="yes"
  cdata-section-elements="script"
  doctype-system="book.dtd"/>
```


<xsl:element>

- Used to output an element node to the current output
- Can be used as an alternative to a literal result element
- Always used within a template body
- The name of the generated element node is determined by attributes *name* and *namespace*
- Attributes may be added to the element node by using elements <xsl:attribute>, <xsl:copy>, or <xsl:copy-of>

<xsl:element> Example

```
<chapter id="001" level="secondary"/>

<xsl:template match="chapter">
  <p>
  <xsl:for-each select="@*">
     <xsl:element name="{name()}">
       <xsl:value-of select="."/>
     </xsl:element>
  </xsl:for-each>
  </p>
</xsl:template>

<p>
  <id>001</id>
  <level>secondary</level>
</p>
```


<xsl:attribute>

- Outputs an attribute name and its value to the current output destination
- Successful only if element node has been added to the output tree and it is a first added node
- Used in either within template body or within <xsl:attribute-set> element
- The name of the attribute to be generated is determined by the value of attribute *name*

<xsl:attribute> Example

```
<manual>
...
</manual>

<xsl:template match="manual">
  <section>
     <xsl:attribute name="version">draft</xsl:attribute>
  </section>
</xsl:template>

<section version="draft">
...
</section>
```


Output Value of Attribute

XSL Template

```
<xsl:template match = "picture" >
  <img src="{@href}" style = "width:250" />
</xsl:template>
```

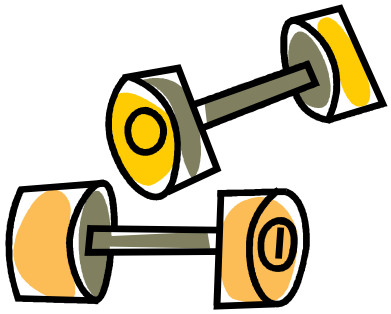
XML Source

```
<picture href='logo.gif'>
```

Output

```
<img src="logo.gif" style = "width:250" />
```


Exercises - Module 12



Questions?