XSL Concepts and Practical Use

Paul Grosso

Arbortext

Norman Walsh

Sun microsystems

Monday, 12 June 2000

XML Europe 2000, Paris, France

Version 1.5

Last Update: Monday, 06 July 2000

Slide 1: What's with stylesheets in the first place?

XML is not a fixed tag set (like HTML)

XML by itself has no (application) semantics

A generic XML processor has no idea what is "meant" by the XML

XML markup does not (usually) include formatting information

The information in an XML document may not be in the form in which it is desired to present it

Therefore there must be something in addition to the XML document that provides information on how to present or otherwise process the XML

Slide 2: Advantages to separating content from style

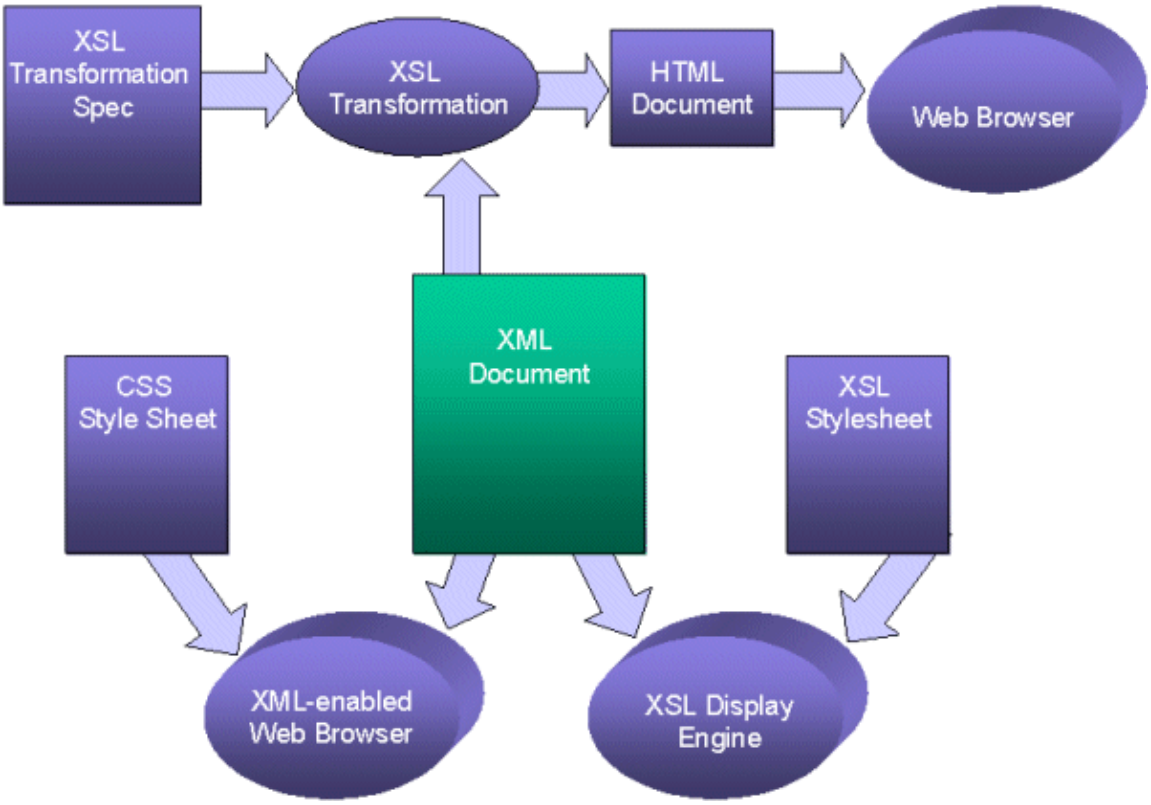
Contrary to when style information is hard−coded into the content, separation of style from content allows for

the same data to be presented in different ways. This enables:

- reuse of fragments of data: the same content should look different in different contexts
- multiple output formats: different media (paper, online), different sizes (manuals, reports), different classes of output devices (workstations, hand−held devices)
- styles tailored to the reader's preference (e.g., accessibility): print size, color, simplified layout for audio readers
- standardized styles: corporate stylesheets can be applied to the content at any time
- freedom from style issues for content authors: technical writers needn't be concerned with layout issues because the correct style can be applied later

---

Slide 3: Options for displaying XML



Slide 4: What Does a Stylesheet Do?

A stylesheet specifies the presentation of XML information using two basic categories of techniques:

-

An optional transformation of the input document into another structure

- A description of how to present the transformed information (i.e., a specification of what properties to associate to each of the various parts of the transformed information)

Slide 5: Transformation capabilities

Transformation capabilities include:

- generation of constant text
- suppression of content
- moving text (e.g., exchanging the order of the first and last name)
- duplicating text (e.g., copying titles to make a table of contents)
- sorting
- more complex transformations that "compute" new information in terms of the existing information

---

Slide 6: Description of information

Description of how to present the (possibly transformed) data includes three levels of formatting information:

- Specification of the general screen or page (or even audio) layout
- Assignment of the transformed content into basic "content container types" (e.g., lists, paragraphs, inline text)
- Specification of formatting properties (spacing, margins, alignment, fonts, etc.) for each resulting "container"

Slide 7: The components of the XSL language

The full XSL language logically consists of three component languages which are described in three W3C (World Wide Web Consortium) Recommendations:

-

XPath: XML Path Language−−a language for referencing specific parts of an XML document

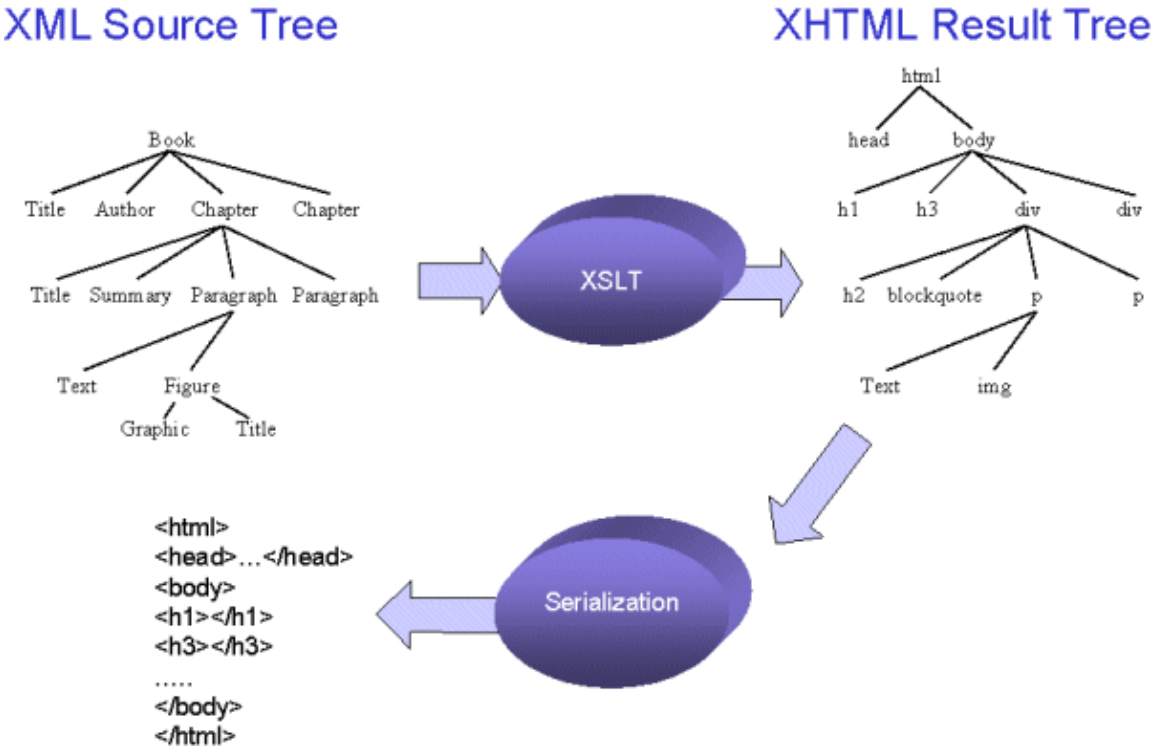
- XSLT: XSL Transformations−−a language for describing how to transform one XML document (represented as a tree) into another

- XSL: Extensible Stylesheet Language−−XSLT plus a description of a set of Formatting Objects and Formatting Properties

---

Slide 8: XML to result tree

An XSLT "stylesheet" transforms the input (source) document's tree into a structure called a *result tree* consisting of result objects



Slide 9: Result tree doctypes

The result tree's structure is that of an XML document, and its objects correspond to elements with attributes

The result tree's structure and "tag set" can match that of any XML document or doctype. In particular, the result tree could be:

HTML/XHTML
: result tree is easily written out as an HTML document

other XML doctype
: result tree is easily written out as an XML document in this other doctype (for some further application−specific processing)

FO result tree
: result tree's structure (and element and attribute names) matches the set of *formatting objects* and formatting *properties* defined by the (non−transformation) part of XSL

Serialization of the result tree is not necessary for further processing of the result tree.

Slide 10: An XSL stylesheet

An XSL stylesheet basically consists of a set of templates

Each template "matches" some set of elements in the source tree and then describes the contribution that the matched element makes to the result tree

Generally, elements in a stylesheet in the "xsl" namespace are part of the XSLT language, and non−xsl elements within a template are what get put into the result tree

---

Slide 11: HTML vs. XSL Formatting Objects

- Transformation is independent of the target result type
- Most people are more familiar with HTML so many of the examples in this tutorial use HTML
- The XSL implementation in IE5 is incomplete. The examples in this tutorial will not work in IE5
- The techniques apply equally well to XSL Formatting Objects or other tag sets
- XSLT is a tree−to−tree transformation process
- Serialization may vary depending on the selected output method
- There is a distinction between HTML element names and HTML

Slide 12: The Structure of a Stylesheet

- XSLT Stylesheets are XML documents; namespaces (http://www.w3.org/TR/REC−xml−names) are used to identify semantically significant elements.
- Most stylesheets are stand−alone documents rooted at <xsl:stylesheet> or <xsl:transform>. It is possible to have "single template" stylesheet/documents.
-

<xsl:stylesheet> and <xsl:transform> are completely synonymous.

Note that it is the mapping from namespace abbreviation to URI that is important, not the literal namespace abbreviation "xsl:" that is used most commonly.

Slide 13: Stylesheet Examples

Figure 1. A Stylesheet

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

Figure 2. A Transformation Sheet

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

Figure 3. Document as Stylesheet

```
<html xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<head>
<title>Silly Example</title>
</head>
<body>
<h1>Silly Example</h1>
<p>You'd probably use extension elements, or somthing
more interesting in real life: 3+4 is <xsl:value-of select="3+4"/>.
</p>
</body>
</html>
```

Slide 14: A Complete Example

This is a simple stylesheet that transforms source <para> and <emphasis> elements into HTML:

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

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

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

</xsl:stylesheet>
```

With this stylesheet, the following XML document:

```
<?xml version='1.0'?>
<para>This is a <emphasis>test</emphasis>.</para>
```

Would be transformed into:

```
<?xml version="1.0" encoding="utf-8"?>
<p>This is a <i>test</i>.</p>
```

Note that this has been serialized as XML.

Slide 15: Understanding A Template

Most templates have the following form:

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

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

Slide 16: Match Patterns (Locating Elements)

One critical capability of a stylesheet language is to locate source elements to be styled. CSS, for example, does this with "selectors." FOSIs do it with "e-i-c's", elements in context. XSLT does it with "match patterns" defined by the XML Path Language (XPath) (http://www.w3.org/TR/xpath).

- XPath has an extensible string-based syntax
- It describes "location paths" between parts of a document or documents
- One inspiration for XPath was the common "path/file" file system syntax

Two things to remember about XPath expressions:

- Pattern matching occurs in a context; XPath expressions and XSLT elements can change the current context and consequently the nodes which match
-

XPath is inclusive or greedy, it addresses all matching elements. You must use predicates to refine the set of nodes selected.

Slide 17: Pattern Examples

`para`
: Matches all <para> children in the current context

`para/emphasis`
: Matches all <emphasis> elements that have a parent of <para>

`/`
: Matches the root of the document

`para//emphasis`
: Matches all <emphasis> elements that have an *ancestor* of <para>

`section/para[1]`
: Matches the first <para> child of all the <section> children in the current context

`//title`
: Matches *all* <title> elements anywhere in the document

`.//title`
: Matches all <title> elements that are descendants of the current context

---

Slide 18: More Complex Patterns

`section/*/note`
: Matches <note> elements that have <section> grandparents.

`stockquote[@symbol]`
: Matches <stockquote> elements that have a "symbol" attribute

`stockquote[@symbol="XXXX"]`
: Matches <stockquote> elements that have a "symbol" attribute with the value "XXXX"

`emphasis|strong`
: Matches <emphasis> or elements

Slide 19: Pattern Syntax

The XPath pattern syntax is described formally in the XPath specification. Here we're going to look at the syntax in abstract, but less formal terms:

- An XPath pattern is a 'location path', a location path is absolute if it begins with a slash ("/") and relative otherwise
-

A relative location path consists of a series of steps, separated by slashes

- A step is an axis specifier, a node test, and a predicate

The formal syntax is made more complicated by the fact that it must describe both the abbreviated and unabbreviated syntaxes

---

Slide 20: Node Tests

Node tests are most frequently element names, but other node tests are possible:

name
: Matches <name> element nodes

***
: Matches any element node

namespace:name
: Matches <name> element nodes in the specified namespace

*namespace:**
: Matches any element node in the specified namespace

comment()
: Matches comment nodes

text()
: Matches text nodes

processing-instruction()
: Matches processing instructions

processing-instruction('target')
: Matches processing instructions with the specified target (<?target ...?>

node()
: Matches any node

Slide 21: Predicates

Predicates occur after the node test in square brackets. A wide range of expressions are possible.

nodetest[1]
: Matches the first node

Most node tests return nodes in document order, only the tests which select ancestor or preceding elements return nodes in reverse document order. The practical result of this is that the "first" node is almost always the one closest to the context node, although parenthesis can change the effective order.

nodetest[position()=last()]

Matches the last node

nodetest[position() mod 2 = 0]
Matches even nodes

element[@id='foo']
Matches the element(s) with id attributes whos value is "foo"

element[not(@id)]
Matches elements that don't have an id attribute

author[firstname="Norman"]
Match <author> elements that have <firstname> children with the content "Norman".

author[normalize−space(firstname)="Norman"]
Match "Norman" without regard to leading and trailing space.

Predicate expressions can be more−or−less arbitrarily complex.

Slide 22: Axis Specifiers

The axis of a node test determines what general category of nodes may be considered for the following node test. There are thirteen axes:

ancestor
Ancestors of the current node

ancestor−or−self
Ancestors, including the current node

attribute
Attributes of the current node (abbreviated "@")

child
Children of the current node (the default axis)

descendant
Descendants of the current node

descendant−or−self
Descendants, including the current node (abbreviated "//")

following/*following−sibling*
Elements which occur after the current node, in document order

preceding/*preceding−sibling*
Elements which occur before the current node, in document order (returned in reverse−document order)

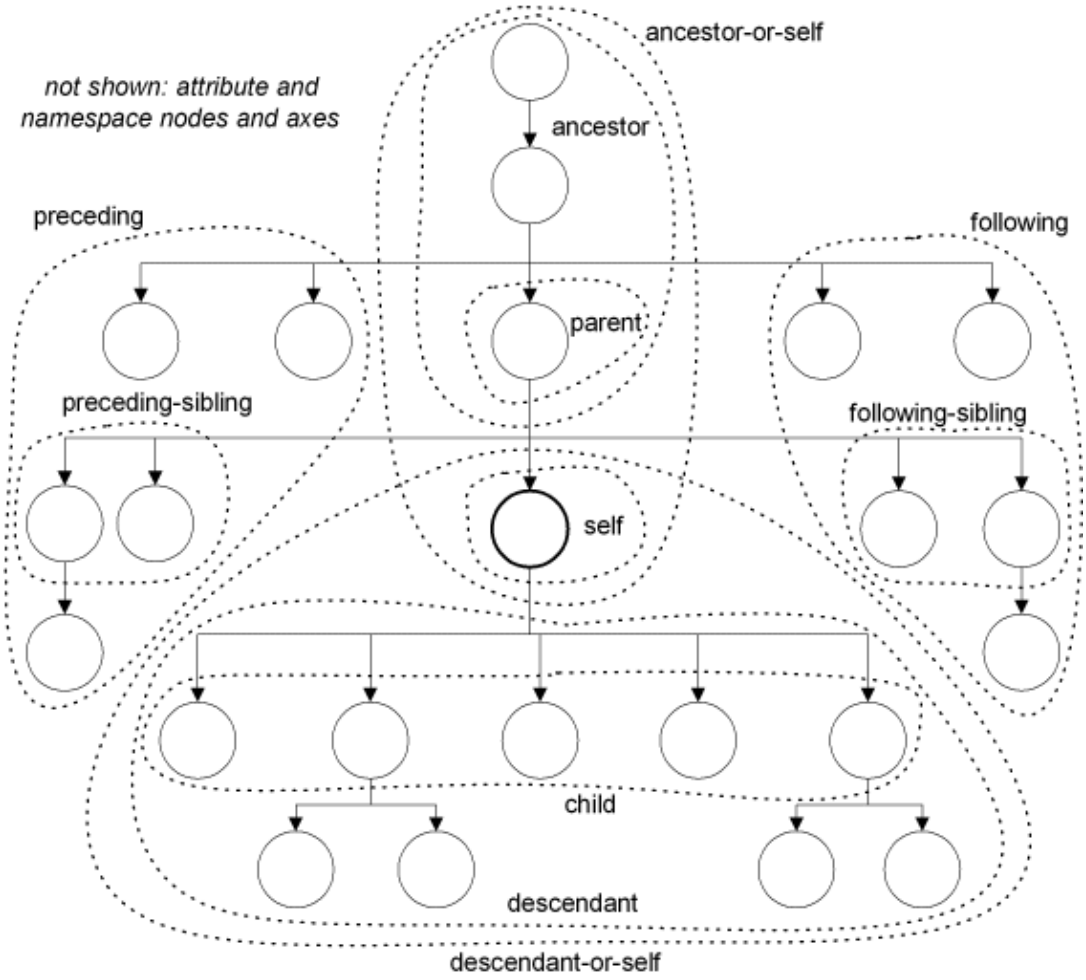
namespace
The namespace nodes of the current node

parent

The parent of the current node (abbreviated "..")

`self`
The current node (abbreviated ".")

The following graphic, courtesy of Crane Softwrights, demonstrates the axes graphically:



Slide 23: A Few More Patterns

Now that we've seen the unabbreviated syntax, let's look at a few more patterns

`ancestor::table`
Matches ancestor <table> elements

`following-sibling::paramdef`
Matches the following <paramdef> siblings

`ancestor-or-self::*/@sepchar`
Matches the sepchar attribute on the current element or any ancestor of the current element

---

Slide 24: Applying Style Recursively

One model for applying style is to allow the process to run recursively, driven primarily by the document. A series of templates is created, such that there is a template to match each context, then these templates are recursively applied starting at the root of the document.

- <xsl:template>

  ```
  <xsl:template match="section/title">
    <h2><xsl:apply-templates/></h2>
  </xsl:template>
  ```

- <xsl:apply−templates>

  ```
  <xsl:apply-templates select="th|td"/>
  ```

There are two obstacles to overcome when using the recursive model, how to arbitrate between multiple patterns that match and how to process the same nodes in different contexts.

Slide 25: Recursive Example

Figure 4. recursive.xml

```
<doc>
<para>This is a <emphasis>test</emphasis>.
<emphasis>Nested <emphasis>emphasis</emphasis></emphasis>.</para>
</doc>
```

Figure 5. recursive.xsl

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

<xsl:template match="doc">
  <html><head><title>A Document</title></head>
  <body><xsl:apply-templates/></body></html>
</xsl:template>

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

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

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

Figure 6. recursive.html

```
<html>
```

```
<head>
<title>A Document</title>
</head>
<body>
<p>This is a <i>test</i>.
<i>Nested <b>emphasis</b></i>.</p>
</body>
</html>
```

Slide 26: Conflict Resolution

The problem of multiple patterns that match is handled by conflict resolution:

- Matching templates from imported modules are not considered if there is a matching template in the current module
- Matching templates with a lower `priority` are not considered. The default priority is determined as follows:
    - Unqualified child or attribute names have a priority of 0.
    - Processing−instructions with a target have a priority of 0.
    - A namespace−qualified "*" child or attribute name has a priority of −0.25.
    - An unqualified "*" has a priority of −0.5
    - Any other template has a default priority of 0.5
    - Template priority may be specified explicitly with the `priority` attribute on <xsl:template>
- "`emphasis`", "`html:p`", and "`@foo`" have a priority of 0
- "`html:*`" has a priority of −0.25
- "`*`" has a priority of −0.5
- "`para/emphasis`" has a priority of 0.5
-

"`emphasis/emphasis`" has a priority of 0.5

- "`emphasis[@role]`" has a priority of 0.5

It is technically an error if the conflict resolution process yields more than one template, however, XSLT processors may (silently) recover from this error by selecting the template that occurs last in the stylesheet.

Effectively, this means that stylesheet template order is the final arbiter.

Slide 27: Changing the Template Set

Now we have an algorithm for selecting exactly which pattern matches, but what happens if the same elements must be processed in multiple contexts? This happens, for example, when dealing with cross references, automatic tables of contents, and multiple views of the same data.

Consider the following example:

Figure 7. modes.xml

```
<?xml version='1.0'?>
<chapter id="foo"><title>Chapter Title</title>

<para>This chapter is self-referential:
<xref linkend="foo"/>.</para>
</chapter>
```

And a stylesheet that produces:

Figure 8. modes.html

```
<h2>Chapter Title</h2>

<p>This chapter is self-referential:
<i>Chapter Title</i>.</p>
```

The chapter title must be processed twice, once to produce the title and once to produce the cross reference.

There is no dynamic mechanism for adjusting template priority, but there is a mechanism for selecting specific sets of templates. That mechanism is called "modes".

Slide 28: Modes Example (Broken)

Without modes, you always get the same template:

Figure 9. modes-broken.xsl

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

```
<xsl:template match="chapter/title">
  <h2><xsl:apply-templates/></h2>
</xsl:template>

<xsl:template match="xref">
  <xsl:variable name="linkend" select="@linkend"/>
  <xsl:apply-templates select="//*[@id=$linkend]/title"/>
</xsl:template>

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

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

</xsl:stylesheet>
```

Figure 10. modes-broken.html

```
<?xml version="1.0" encoding="utf-8"?>
<h2>Chapter Title</h2>

<p>This chapter is self-referential:
<h2>Chapter Title</h2>.</p>
```

Slide 29: Modes Example

Figure 11. modes.xsl

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

<xsl:output method="html"/>

<xsl:template match="chapter/title">
  <h2><xsl:apply-templates/></h2>
</xsl:template>

<xsl:template match="chapter/title" mode="crossref">
  <i><xsl:apply-templates/></i>
</xsl:template>

<xsl:template match="xref">
  <xsl:variable name="linkend" select="@linkend"/>
  <xsl:apply-templates select="//*[@id=$linkend]/title"
                       mode="crossref"/>
</xsl:template>

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

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

```
</xsl:stylesheet>
```

Figure 12. modes.html

```
<h2>Chapter Title</h2>

<p>This chapter is self−referential:
<i>Chapter Title</i>.</p>
```

Slide 30: Applying Style Procedurally

The other model for applying style is to select each action procedurally. A series of templates is created, such that each template explicitly selects and processes the necessary elements.

- <xsl:for−each>

  ```
  <xsl:for−each select="row">
    <tr><xsl:apply−templates/></tr>
  </xsl:for−each>
  ```

- Named Templates

  - <xsl:param>

    ```
    <xsl:param name="type">warning</xsl:param>
    ```

- <xsl:call−template>

  ```
  <xsl:call−template name="admonition"/>
  ```

  - <xsl:with−param>

    ```
    <xsl:call−template name="admonition">
      <xsl:with−param name="type">caution</xsl:with−param>
    </xsl:call−template>
    ```

Slide 31: For−each Example

Figure 13. foreach.xml

```
<?xml version='1.0'?>
<table>
  <row><entry>a1</entry><entry>a2</entry></row>
  <row><entry>b1</entry><entry>b2</entry></row>
  <row><entry>c1</entry><entry>c2</entry></row>
```

```
</table>
```

Figure 14. foreach.xsl

```
<?xml version='1.0'?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"/>

<xsl:template match="table">
  <table>
    <xsl:for-each select="row">
      <tr>
        <xsl:for-each select="entry">
          <td><xsl:apply-templates/></td>
        </xsl:for-each>
      </tr>
    </xsl:for-each>
  </table>
</xsl:template>
</xsl:stylesheet>
```

Figure 15. foreach.html

```
<table>
<tr>
<td>a1</td><td>a2</td>
</tr>
<tr>
<td>b1</td><td>b2</td>
</tr>
<tr>
<td>c1</td><td>c2</td>
</tr>
</table>
```

Challenge: rewrite this stylesheet in recursive terms

Slide 32: Named Template Example

Figure 16. namedtemplate.xml

```
<chapter>

<warning>
<para>Using a damaged extension cord may cause a fire.</para>
</warning>

<caution>
<para>Freshly brewed coffee is hot.</para>
</caution>
</chapter>
```

Figure 17. namedtemplate.xsl

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

```
<xsl:output method="html"/>

<xsl:template name="admonition">
  <xsl:param name="type">Warning</xsl:param>
  <table border="1">
    <tr><th><xsl:value-of select="$type"/>:</th></tr>
    <tr><td><xsl:apply-templates/></td></tr>
  </table>
</xsl:template>

<xsl:template match="warning">
  <xsl:call-template name="admonition"/>
</xsl:template>

<xsl:template match="caution">
  <xsl:call-template name="admonition">
    <xsl:with-param name="type">Caution</xsl:with-param>
  </xsl:call-template>
</xsl:template>

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

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

Figure 18. namedtemplate.html

```
<table border="1">
<tr>
<th>Warning:</th>
</tr>
<tr>
<td>
<p>Using a damaged extension cord may cause a fire.</p>
</td>
</tr>
</table>

<table border="1">
<tr>
<th>Caution:</th>
</tr>
<tr>
<td>
<p>Freshly brewed coffee is hot.</p>
</td>
</tr>
</table>
```

Slide 33: Creating the Result Tree

Literal Result Elements

Any element in a template rule that is not in the XSL (or other extension) namespace is copied literally to the result tree

<xsl:text>

The content of <xsl:text> elements is copied directly to the result tree; whitespace is preserved by default

```
<xsl:text>Literal result text</xsl:text>
```

<xsl:value−of>

Inserts the value of an expression into the result tree, converting it to a string first if necessary

```
<xsl:value−of select="$count + 1"/>
```

<xsl:copy> *and* *<xsl:copy−of>*

Copies the current node or, in the case of xsl:copy−of, the selected nodes, into the result tree without first converting them to a string

```
<xsl:copy−of select="title"/>
```

<xsl:element>

Instantiates the named element

```
...
<xsl:param name="header">h3</xsl:param>
...
<xsl:element name="{$header}">
  <xsl:apply−templates/>
</xsl:element>
```

<xsl:attribute>

Adds the named attribute to the nearest containing element

```
<table>
  <xsl:if test="@pgwide='1'">
    <xsl:attribute name="width">100%</xsl:attribute>
  </xsl:if>
  ...
</table>
```

Slide 34: Conditional Processing

<xsl:if>

Simple conditional (no "else")

```
<xsl:if test="$somecondition">
  <xsl:text>this text only gets used if $somecondition is true()</xsl:text>
</xsl:if>
```

<xsl:choose>

Select among alternatives with <xsl:when> and <xsl:otherwise>

```
<xsl:choose>
  <xsl:when test="$count > 2"><xsl:text>, and </xsl:text></xsl:when>
  <xsl:when test="$count > 1"><xsl:text> and </xsl:text></xsl:when>
  <xsl:otherwise><xsl:text> </xsl:text></xsl:otherwise>
</xsl:choose>
```

To report errors, use <xsl:message>.

```
<xsl:message>
  <xsl:text>Error: no ID found for linkend: </xsl:text>
  <xsl:value-of select="@linkend"/>
  <xsl:text>.</xsl:text>
</xsl:message>
```

Slide 35: Declaring Variables

- <xsl:variable> allows you to associate a variable with a string, node list, or result tree fragment.
- Variables are "single assignment" (no side effects)
- Variables are lexically scoped
- Variables assigned inside a conditional only apply inside that conditional. This is rarely correct:

  ```
  <xsl:if test="$foo">
    <xsl:variable name="bar">...</xsl:variable>
  </xsl:if>
  ```

  Usually, you want to put the conditional inside the variable

  ```
  <xsl:variable name="bar">
    <xsl:if test="$foo">...</xsl:if>
  </xsl:variable>
  ```

Slide 36: Using Variables

After variables (or parameters) have been declared, they can be used in two places:

- In XSL element attributes that expect an expression. These are summarized in the following table:

XSLT Element	Attribute
xsl:apply−templates	select
xsl:value−of	select
xsl:number	value
xsl:for−each	select
xsl:if	test
xsl:when	test
xsl:sort	select

-

In attribute value templates; attribute value templates have the form "{$variable}". They are allowed in the following places:

Element	Attribute
Literal result elements	any attribute
xsl:element	name
	namespace
xsl:attribute	name
	namespace
xsl:number	level
	count
	from
	format
	lang
	grouping−separator
	grouping−size
xsl:sort	order
	lang
	data−type
	case−order
xsl:processing−instruction	name

- To insert a literal "{" character in a context where attribute value templates are expanded, use "{{".

Slide 37: Numbering

The <xsl:number> element performs two functions:

1. It evaluates a numeric expression and converts the result into a formatted string:

```
<xsl:number value="3" format="A. "/>
<xsl:number value="count(listitem)" format="01"/>
```

2. It counts elements in the *source* tree and converst the result into a formatted string:

```
<xsl:number count="listitem" format="i. "/>
<xsl:number count="chapter" from="book" level="any" format="1. "/>
<xsl:number count="h1|h2|h3" level="multiple" from="chapter|appendix" format="1."/>
```

The details of number to string conversion is spelled out in great detail in Section 7.7.1 (http://www.w3.org/TR/xslt.html#convert) of the XSLT Recommendation (http://www.w3.org/TR/xslt.html). See also, <xsl:decimal−format> and the Java JDK 1.1 DecimalFormat (http://java.sun.com/products/jdk/1.1/docs/api/java.text.DecimalFormat.html) class.

Slide 38: Sorting

The <xsl:sort> element sorts a set of nodes according to the criteria specified:

```
<xsl:apply-templates select="row">
  <xsl:sort data-type="number" select="entry[2]"/>
</xsl:apply-templates>
```

It can appear as a child of <xsl:apply−templates> or <xsl:for−each>. It can also be nested.

Slide 39: Sorting Example

Figure 19. sort.xml

```
<doc>
<para>Here's a table of sales:</para>
<table>
<row><cell>3000</cell><cell>Widgets 'R' Us</cell></row>
<row><cell>2400</cell><cell>Widget Design and Implementation</cell></row>
<row><cell>10000</cell><cell>Widgets for Dummies</cell></row>
<row><cell>101</cell><cell>101 Uses for a Dead Widget</cell></row>
</table>
</doc>
```

Figure 20. sort.xsl

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

<xsl:import href="element.xsl"/>

<xsl:template match="table">
  <xsl:if test="@id"><a name="{@id}"/></xsl:if>
  <table>
    <xsl:apply-templates select="row">
      <xsl:sort data-type="number" select="./cell[1]"/>
    </xsl:apply-templates>
  </table>
</xsl:template>
</xsl:stylesheet>
```

Figure 21. sort.html

```
<p>Here's a table of sales:</p>
<table>
<tr>
<td>101</td><td>101 Uses for a Dead Widget</td>
</tr>
<tr>
<td>2400</td><td>Widget Design and Implementation</td>
</tr>
<tr>
<td>3000</td><td>Widgets 'R' Us</td>
</tr>
<tr>
<td>10000</td><td>Widgets for Dummies</td>
```

```
</tr>
</table>
```

Slide 40: Formatting objects and properties

```
XSL = XSLT + vocabulary of FOs and properties
```

XSL defines a powerful set of formatting objects

XSL uses (and extends) a set of Common Formatting Properties developed jointly with the CSS&FP (Cascading Style Sheet and Formatting Property) Working Group

When a result tree uses this standardized set of formatting objects and properties, then an XSL−compliant formatter can process that result tree to produce the specified output

---

Slide 41: Sample FO result tree

```
<?xml version="1.0" encoding="utf−8"?>
<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
...
<fo:page−sequence>
  <fo:flow>
    <fo:block font−size="18pt" font−weight="bold"
              text−align="centered">
      Preface
    </fo:block>
    <fo:block font−size="12pt" space−before="1pc"
              text−align="justified">
      This is a simple test document.  It shows a
      <fo:inline font−style="italic">partial</fo:inline>
      fo−result tree (page layout missing).
    </fo:block>
  </fo:flow>
</fo:page−sequence>
...
</fo:root>
```

Slide 42: Some XSL−FO templates

```
<xsl:template match="chapter">
  <fo:flow>
    <xsl:apply−templates/>
  </fo:flow>
</xsl:template>

<xsl:template match="chapter/title">
  <fo:block font−size="18pt" font−weight="bold"
            text−align="centered">
    <xsl:apply−templates/>
  </fo:block>
</xsl:template>

<xsl:template match="para">
```

```
  <fo:block font-size="12pt" space-before="1pc"
            text-align="justified">
    <xsl:apply-templates/>
  </fo:block>
</xsl:template>

<xsl:template match="emphasis">
  <fo:inline font-style="italic">
    <xsl:apply-templates/>
  </fo:inline>
</xsl:template>
```

Slide 43: Formatting Object Basics

- Inline versus block objects
- Common formatting properties--harmonized with CSS
- Inheritance of properties within the result tree--inheritable and non-inheritable properties

---

Slide 44: Common Formatting Objects

- page-sequence--a major part (such as front or body) in which the basic page layout may differ from other parts
- flow--a chapter- or section-like division within a page-sequence
- block--a paragraph (or title or block quote, etc.)
- inline--e.g., a font change within a paragraph
- wrapper--a "transparent" object usable as either a block or inline object that has no effect other than to provide a place to hang inheritable properties
- list FOs--list-block, list-item, list-item-label, list-item-body
- graphic--references an external graphic object
- table FOs--mostly analogous to the standard (CALS, OASIS, HTML) table models

---

Slide 45: Basic properties

- font properties
- margin and spacing properties
- border and padding properties
- keeps/breaks
- horizontal alignment/justification
- indentation
- more formatting object specific properties

Slide 46: Overall XSL formatting capabilities

XSL FO formatting capabilities in XSL 1.0 are approximately the union of:

- HTML + CSS capabilities
- most high quality print output capabilities including internationalization features

Not included are complex page layouts (e.g., magazine and newspaper layout), complex layout−driven formatting (e.g., copyfitting and complex floats), and looseleaf pagination (change page production)

Slide 47: Chapter, title, para example

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

<xsl:include href="fo−root.xsl"/>

<xsl:template match="article">
  <fo:page−sequence font−family="serif" font−size="12pt">
    <xsl:apply−templates/>
  </fo:page−sequence>
</xsl:template>

<xsl:template match="chapter">
```

```
  <fo:flow>
    <xsl:apply-templates/>
  </fo:flow>
</xsl:template>

<xsl:template match="para">
  <fo:block space-before="6pt">
    <xsl:apply-templates/>
  </fo:block>
</xsl:template>

<xsl:template match="chapter/title">
  <fo:block font-family="sans-serif" color="blue"
         font-weight="bold" font-size="18pt"
         space-after="0.5em">
    <xsl:number level="multiple" count="chapter"/>
    <xsl:text> </xsl:text>
    <xsl:apply-templates/>
  </fo:block>
</xsl:template>

<xsl:template match="section/title">
  <fo:block font-family="sans-serif"
         font-weight="bold" font-size="16pt"
         space-after="0.5em">
    <xsl:number level="multiple" count="chapter|section"/>
    <xsl:text> </xsl:text>
    <xsl:apply-templates/>
  </fo:block>
</xsl:template>

<xsl:template match="article/title">
  <fo:block font-family="sans-serif" font-size="24pt"
         space-after="2pc">
    <xsl:apply-templates/>
  </fo:block>
</xsl:template>

</xsl:stylesheet>
```

Slide 48: Modularizing Stylesheets

<xsl:include>
: Performs a roughly textual include of the referenced stylesheet

<xsl:import>
: The importing stylesheet "extends" the imported stylesheet

Slide 49: <xsl:import>

An <xsl:import> statement loads an auxilary stylesheet

```
<xsl:import
  href = uri-reference />
```

href
: The URI of the stylesheet to import

Templates imported from another stylesheet have a different import precedence.

Slide 50: <xsl:include>

An <xsl:include> statement performs a "textual include" of another stylesheet:

```
<xsl:include
  href = uri-reference />
```

href
: The URI of the stylesheet to import

Templates included from another stylesheet have the same import precedence and behave exactly as if they had been literally inserted where the <xsl:include> statement occurs.

Slide 51: Controlling Input Whitespace

XSLT has to handle whitespace in both the stylesheet and the source document in a reasonable and controllable way.

To deal with whitespace in the stylesheet, it employs the following rules:

- If a text node contains non−whitespace characters, it is preserved
- If a text node occurs inside <xsl:text> it is preserved
- If the nearest ancestor `xml:space` attribute has the value `preserve`, it is preserved
- Otherwise it is stripped (discarded from the tree before any transformation begins)

The rules for whitespace in the source document are the same, with the exception that you can specify the default stripping rules with <xsl:preserve−space> and <xsl:strip−space>:

- If a text node contains non−whitespace characters, it is preserved
- If a text node occurs inside an element listed in <xsl:preserve−space>, it is preserved
- If the nearest ancestor `xml:space` attribute has the value `preserve`, it is preserved
- Otherwise it is stripped (discarded from the tree before any transformation begins)

Slide 52: <xsl:preserve−space>

The <xsl:preserve−space> element lists the source tree elements in which input whitespace is to be preserved.

```
<xsl:preserve-space
  elements = tokens />
```

elements
: The list of elements in which whitespace should be preserved by default

Note that namespace:* and * can be used to incorporate all elements or all elements from a particular namespace by default.

Slide 53: <xsl:strip−space>

The <xsl:strip−space> element lists the source tree elements in which input whitespace is to be discarded.

```
<xsl:strip-space
  elements = tokens />
```

elements
: The list of elements in which whitespace should be discarded by default

Note that namespace:* and * can be used to incorporate all elements or all elements from a particular namespace by default.

Slide 54: Controlling Output Serialization

XSLT is a tree−to−tree transformation. It isn't specifically the job of the XSLT processor to write out a text representation (serialization) of the tree. However, XSLT does allow the stylesheet author to suggest the desired serialization procedure:

The selection of namespace aliases is controlled by <xsl:namespace−alias>.

Slide 55: <xsl:output>

```
<xsl:output
  method = "xml" | "html" | "text" | qname-but-not-ncname
  version = nmtoken
  encoding = string
  omit-xml-declaration = "yes" | "no"
  standalone = "yes" | "no"
  doctype-public = string
  doctype-system = string
  cdata-section-elements = qnames
  indent = "yes" | "no"
  media-type = string />
```

method
: The serialization method

version
: The version of the serializer

encoding
: The encoding to use, if possible

omit−xml−declaration
: Do not output the <?xml ...?> declaration

standalone
: Identify the result as a standalone document (in the XML sense)

doctype−public
: The public identifier to use on the output <!DOCTYPE> declaration

doctype−system
: The system identifier to use on the output <!DOCTYPE> declaration

cdata−section−elements
: A list of element names; the content of these element should be escaped with CDATA sections if possible

indent
: Add additional space to the serialized result tree for readability

media−type
: The media type of the serialized result

Slide 56: <xsl:namespace−alias>

Allows the stylesheet to alias one namespace prefix for another in the result tree

```
<xsl:namespace-alias
  stylesheet-prefix = prefix | "#default"
  result-prefix = prefix | "#default" />
```

stylesheet−prefix
: The prefix used in the stylesheet

result−prefix
: The desired prefix in the result

Slide 57: Applying Style

Suppose that you have a stylesheet which customizes another stylesheet, perhaps changing the formatting of a few elements. Suppose that one of these elements is formatted by a complex template in the base stylesheet, and all you want to change is one small, special case. How can you do this?

The answer is with <xsl:apply−imports>:

```
...
<xsl:import href="basestyle.xsl"/>
```

```
...
<xsl:template match="table">
  <xsl:choose>
    <xsl:when test="@role='something-special'">
      <xsl:call-template name="special-table-handler"/>
    </xsl:when>
    <xsl:otherwise>
      <xsl:apply-imports/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

Slide 58: <xsl:apply-imports>

Process the context node with the "next matching" pattern from an *imported* stylesheet.

```
<xsl:apply-imports/>
```

Slide 59: Creating the Result Tree

There are several more elements that can be used in creating the result tree:

<xsl:attribute-set>
: Names a collection of attributes for use as a set

<xsl:comment>
: Creates an XML comment (<!-- like this -->) in the result tree

<xsl:processing-instruction>
: Creates an XML processing instruction (<?pitarget like="this"?>) in the result tree

---

Slide 60: <xsl:attribute-set>

The <xsl:attribute-set> element gives a name to a collection of attribute/value pairs.

```
<xsl:attribute-set
  name = qname
  use-attribute-sets = qnames>
  <!-- Content: xsl:attribute* -->
</xsl:attribute-set>
```

name
: The attribute set name

use-attribute-sets
: Uses another collection of attributes as the default set for this attribute set

---

Slide 61: <xsl:comment>

A <xsl:comment> element inserts an XML comment into the result tree.

```
<xsl:comment>
  <!−− Content: template −−>
</xsl:comment>
```

Slide 62: <xsl:processing−instruction>

A <xsl:processing−instruction> element inserts an XML processing instruction into the result tree.

```
<xsl:processing−instruction
  name = { ncname }>
  <!−− Content: template −−>
</xsl:processing−instruction>
```

name
: The processing instruction target name

Slide 63: Attribute Set Example

Figure 22. attrset.xsl

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

<xsl:output method="html"/>

<xsl:attribute−set name="table−cell−attrs">
  <xsl:attribute name="align">left</xsl:attribute>
  <xsl:attribute name="valign">top</xsl:attribute>
  <xsl:attribute name="bordercolor">blue</xsl:attribute>
</xsl:attribute−set>

<xsl:template match="doc">
  <xsl:apply−templates/>
</xsl:template>

<xsl:template match="table">
  <xsl:if test="@id"><a name="{@id}"/></xsl:if>
  <xsl:comment>Attributes created with xsl:attribute−set</xsl:comment>
  <table border="1">
    <xsl:apply−templates/>
  </table>
</xsl:template>

<xsl:template match="head|cell">
  <xsl:variable name="output−element">
    <xsl:choose>
      <xsl:when test="local−name(.)='head'">th</xsl:when>
      <xsl:otherwise>td</xsl:otherwise>
    </xsl:choose>
  </xsl:variable>
  <xsl:element name="{$output−element}" use−attribute−sets="table−cell−attrs">
    <xsl:copy−of select="@*"/>
    <xsl:apply−templates/>
  </xsl:element>
</xsl:template>

<xsl:template match="row">
```

```
  <tr>
    <xsl:copy-of select="@*"/>
    <xsl:apply-templates/>
  </tr>
</xsl:template>

<xsl:template match="link">
  <a>
    <xsl:attribute name="href">
      <xsl:text>#</xsl:text>
      <xsl:value-of select="@linkend"/>
    </xsl:attribute>
    <xsl:apply-templates/>
  </a>
</xsl:template>

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

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

</xsl:stylesheet>
```

Figure 23. attrset.html

```
<p>This is some paragraph of text. See
<a href="#foo">the table</a>.</p>

<a name="foo"></a><!--Attributes created with xsl:attribute-set-->
<table border="1">
<tr>
<th align="left" valign="top" bordercolor="blue">Head 1</th><th align="left" valign="top" bordercolor=
</tr>
<tr>
<td align="center" valign="top" bordercolor="blue">Value 1</td><td align="left" valign="top" borderco
</tr>
<tr>
<td align="left" valign="top" bordercolor="blue">Value 3</td><td align="left" valign="top" bordercolo
</tr>
<tr>
<td align="left" valign="top" bordercolor="blue">Value 5</td><td align="left" valign="top" bordercolo
</tr>
</table>
```

Slide 64: Disabled Escaping Example

Figure 24. disable-escaping.xml

```
<doc>
This document is <copyright>No One In Particular</copyright>.
</doc>
```

Figure 25. disable-escaping.xsl

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

```
                version="1.0">

<xsl:output method="xml"/>

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

<xsl:template match="copyright">
  <xsl:text>Copyright </xsl:text>
  <xsl:text disable-output-escaping="yes">&amp;copy;</xsl:text>
  <xsl:text> 1999 </xsl:text>
  <xsl:apply-templates/>
</xsl:template>

</xsl:stylesheet>
```

Figure 26. disable-escaping.html

```
<?xml version="1.0" encoding="utf-8"?>
<p>
This document is Copyright &copy; 1999 No One In Particular.
</p>
```

A better technique for this particular example would be simply to use the numeric character reference for the Copyright symbol (©) and allow the serializer to "do the right thing."

Slide 65: Sorting and Numbering

The conversion of numbers from their internal, binary representation to a text representation is controlled by the <xsl:decimal-format> element.

Slide 66: Multiple Documents

A stylesheet can load additional documents with the document() function. In the following example, we recast the sorted table example using several features we have learned:

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

<xsl:import href="element.xsl"/>

<xsl:variable name="tabledoc">sorttable.xml</xsl:variable>

<xsl:template match="insert-table">
  <xsl:if test="@id"><a name="{@id}"/></xsl:if>
  <xsl:apply-templates select="document($tabledoc)"/>
</xsl:template>

<xsl:template match="table">
  <table>
    <xsl:apply-templates select="row">
      <xsl:sort data-type="number" select="./cell[1]"/>
    </xsl:apply-templates>
```

```
  </table>
</xsl:template>

</xsl:stylesheet>
```

Slide 67: Keys

The ID/IDREF mechanism for locating elements in XML documents has been generalized in XSL to the notion of keys.

- ID/IDREF is only useful when:
  - The document has declarations that identify the ID and IDREF attributes and
  - The processor is capable of processing the declarations
- Using select expressions (XPath) to locate elements may be inefficient

Declaring keys gives the stylesheet processor an indication of what elements should be cached for fast access.

Slide 68: Key Example

Figure 27. keys.xsl

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

<xsl:key name="bibkey" match="//biblio/bib" use="@abbrev"/>

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

<xsl:template match="biblio">
  <!-- suppressed -->
</xsl:template>

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

<xsl:template match="bibref">
  <xsl:variable name="bib" select="key('bibkey',string(.))"/>
  <xsl:choose>
    <xsl:when test="$bib">
      <i><xsl:apply-templates select="$bib"/></i>
    </xsl:when>
    <xsl:otherwise>
      <b>NO BIB FOR <xsl:apply-templates/></b>
```

```
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>

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

</xsl:stylesheet>
```

Figure 28. keys.xml

```
<doc>
<para>See <bibref>xyzzy</bibref>.</para>
<biblio>
<bib abbrev="xyzzy">The Great Grue</bib>
<bib abbrev="abcde">That Alphabet Song</bib>
</biblio>
</doc>
```

Figure 29. keys.html

```
<?xml version="1.0" encoding="utf-8" ?>
See <i>The Great Grue</i>.
```

Slide 69: Extending XSLT

- Extension Functions
- Extension Elements
- <xsl:fallback>

Slide 70: Extension Examples

These examples are from James' XT distribution:

Figure 30. ext-func.xsl

```
<xsl:stylesheet
  version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:date="http://www.jclark.com/xt/java/java.util.Date">

<xsl:template match="/">
  <html>
    <xsl:if test="function-available('date:to-string') and function-available('date:new')">
      <p><xsl:value-of select="date:to-string(date:new())"/></p>
```

```
    </xsl:if>
  </html>
</xsl:template>

</xsl:stylesheet>
```

Figure 31. ext−elem.xsl

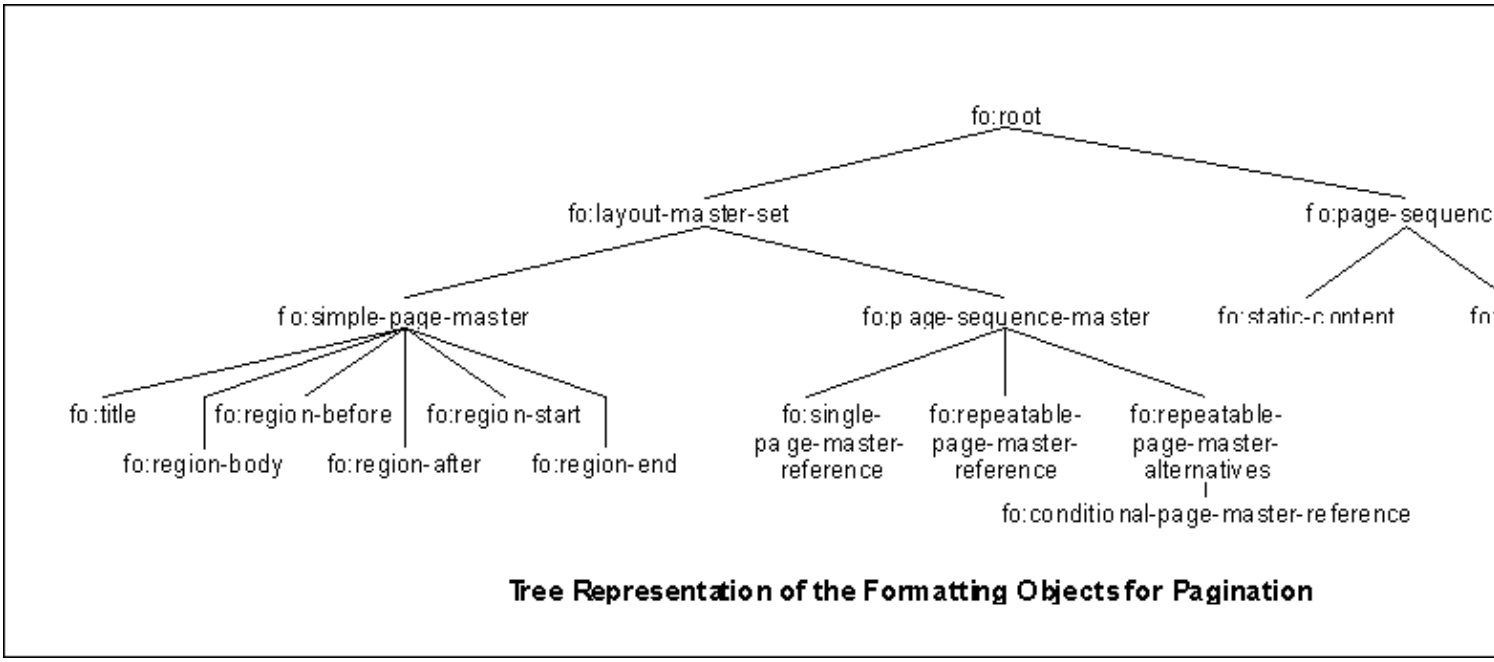
```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0"
                xmlns:xt="http://www.jclark.com/xt"
                extension−element−prefixes="xt">

<xsl:output method="text"/>

<xsl:template match="file">
  <xt:document href="{@name}" method="xml">
    <xsl:fallback>This should not happen.</xsl:fallback>
    <xsl:apply−templates mode="copy"/>
  </xt:document>
  <xsl:text>Created file </xsl:text>
  <xsl:value−of select="@name"/>
  <xsl:text>&#xA;</xsl:text>
</xsl:template>

<xsl:template mode="copy" match="*|@*">
  <xsl:copy>
    <xsl:apply−templates select="@*" mode="copy"/>
    <xsl:apply−templates select="node()" mode="copy"/>
  </xsl:copy>
</xsl:template>

</xsl:stylesheet>
```

Slide 71: Nested list example

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

<xsl:template match="doc">
  <fo:block>
    <xsl:apply−templates/>
  </fo:block>
</xsl:template>

<xsl:template match="p">
  <fo:block>
    <xsl:apply−templates/>
  </fo:block>
</xsl:template>

<xsl:template match="list">
  <fo:list−block space−before="10pt"
                 provisional−distance−between−starts="3pc"
                 provisional−label−separation="1pc">
    <xsl:apply−templates/>
  </fo:list−block>
```

```
</xsl:template>

<xsl:template match="list//list">
  <fo:list-block space-before="8pt"
                 provisional-distance-between-starts="3pc"
                 provisional-label-separation="1pc"
                 start-indent="3pc">
    <xsl:apply-templates/>
  </fo:list-block>
</xsl:template>

<xsl:template match="list/item">
  <fo:list-item space-before="6pt">
    <fo:list-item-label>
      <fo:block>
        <xsl:number count="item" format="1."/>
      </fo:block>
    </fo:list-item-label>
    <fo:list-item-body>
      <fo:block>
        <xsl:apply-templates/>
      </fo:block>
    </fo:list-item-body>
  </fo:list-item>
</xsl:template>

<xsl:template match="list//list/item" priority="1">
  <fo:list-item space-before="6pt">
    <fo:list-item-label>
      <fo:block>
        <xsl:number count="item" format="a."/>
      </fo:block>
    </fo:list-item-label>
    <fo:list-item-body>
      <fo:block>
        <xsl:apply-templates/>
      </fo:block>
    </fo:list-item-body>
  </fo:list-item>
</xsl:template>

<xsl:template match="list//list//list/item" priority="2">
  <fo:list-item space-before="6pt">
    <fo:list-item-label>
      <fo:block>
        <xsl:number count="item" format="i."/>
      </fo:block>
    </fo:list-item-label>
    <fo:list-item-body>
      <fo:block>
        <xsl:apply-templates/>
      </fo:block>
    </fo:list-item-body>
  </fo:list-item>
</xsl:template>

</xsl:stylesheet>
```

Slide 72: Pagination basics



Tree Representation of the Formatting Objects for Pagination

Slide 73: Pagination example

```
<fo:layout-master-set>
  <fo:simple-page-master page-master-name="right">
    <fo:region-body margin-bottom="50pt"/>
    <fo:region-after extent="25pt"/>
  </fo:simple-page-master>

  <fo:simple-page-master page-master-name="left">
    <fo:region-body margin-bottom="50pt"/>
    <fo:region-after extent="25pt"/>
  </fo:simple-page-master>
</fo:layout-master-set>

<fo:page-sequence>
  <fo:static-content flow-name="xsl-after">
    <fo:block><fo:page-number/></fo:block>
  </fo:static-content>

  <fo:flow>
   . . .
  </fo:flow>
</fo:page-sequence>
```

Slide 74: Defining the semantics of FOs and properties

The final result of an XSL processor is the FO result tree that would then be passed to an XSL−enabled formatter for the final composition

To allow for the precise definition of the semantics of each formatting object and property, the XSL specification describes an *area structure* that is approximately analogous to a formatter's unpaginated "galley" structure

Slide 75: The area structure

The area structure is conceptually one of nested boxes laid out with spaces around them and with formatting traits attached

There are two types of areas: block areas and inline areas

A line area is an example of a block are; a glyph area (for a "character") is an example of an inline area

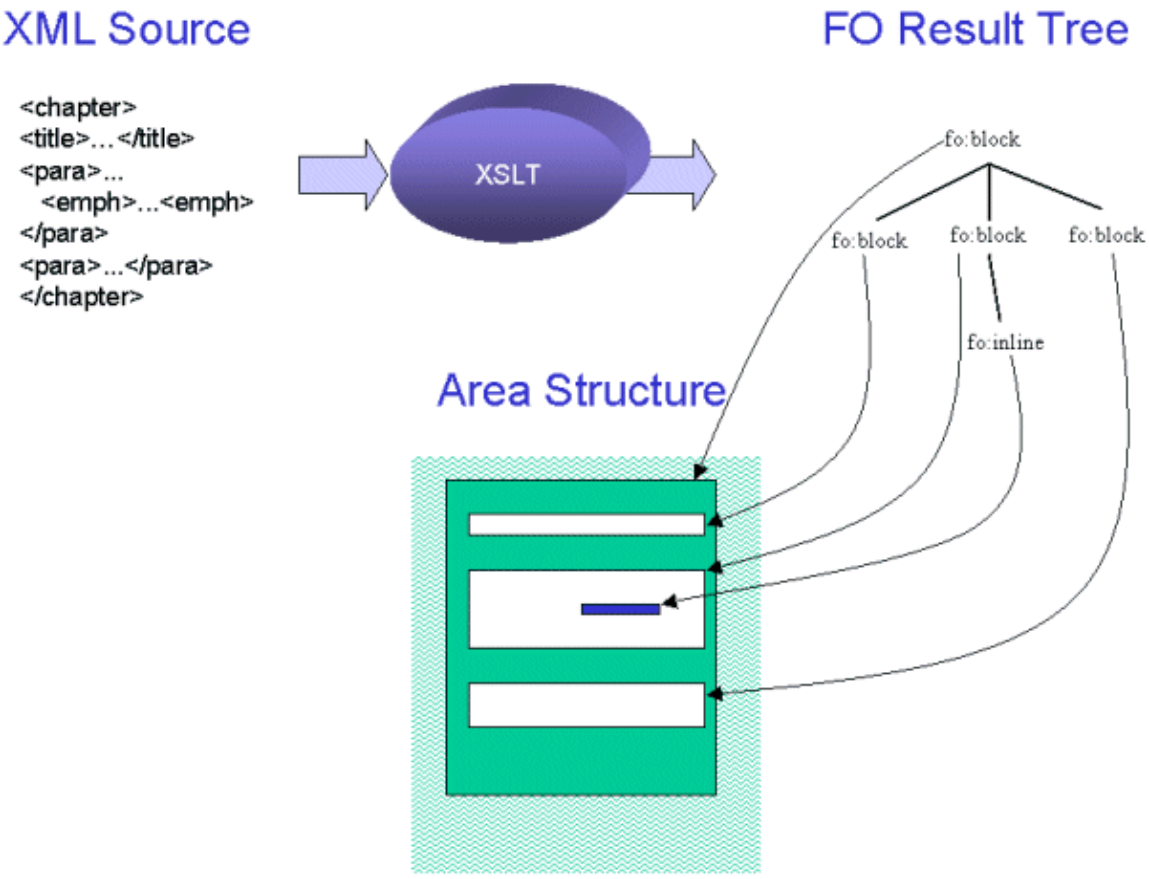
The areas are separated by display space and inline space

The areas are placed within area containers (e.g., those corresponding to pages)

Areas have traits which are derived from the FO's properties (often directly, but sometimes via some computation)

Slide 76: Basic Formatting Model

The XSL formatting model consists of converting an fo−result tree into a structure of areas and traits



Slide 77: Inheritance of properties

Properties are either inheritable or not−−inheritance occurs within the result tree

Inheritable properties may not have any effect on a particular formatting object on which they are set but their values may affect descendant objects

Properties have some initial (default) value

Therefore, there is always a value defined for every property on every formatting object (either inherited, defaulted, or assigned), though not every property will be relevant to every formatting object

The description of each FO says which properties are relevant to it and how

---

Slide 78: Refinement of properties

Some properties can have values that require *refinement* to compute their value

Refinement occurs after inheritance

Examples of refinement:

- shortcuts such as font which map to multiple properties such as font−size, font−weight, etc.
- cases where multiple properties can affect similar traits such as padding−left and padding−start or space−before and margin−top
- values (e.g., margins) that can be specified in terms of percentage of the column width need to be resolved to a distance

---

Slide 79: Space specifiers

Properties such as space−before, space−after, and others are given as a *space specifier*

A space specifier consists of five parts:

- a length giving the *optimum* spacing amount
- a length giving the *minimum* spacing permissible
- a length giving the *maximum* spacing permissible
- an integer giving the *precedence* of this spacing amount

- a boolean indicating the *conditionality* of this spacing

Consecutive space specifiers are "merged" or "collapsed" according to an algorithm that takes their precedence and lengths into account

Space at the boundary of an area container (e.g., a page) gets discarded if its conditionality indicates that it is discardable

---

Slide 80: Block spacing example

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:fo="http://www.w3.org/1999/XSL/Format">

<xsl:template match="chapter">
  <fo:block break-before="page">
    <xsl:apply-templates/>
  </fo:block>
</xsl:template>

<xsl:template match="chapter/title">
  <fo:block text-align="center" space-after="8pt"
            space-before="16pt" space-after.precedence="3">
    <xsl:apply-templates/>
  </fo:block>
</xsl:template>

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

<xsl:template match="section/title">
  <fo:block text-align="center" space-after="6pt"
            space-before="12pt" space-before.precedence="0"
            space-after.precedence="3">
    <xsl:apply-templates/>
  </fo:block>
</xsl:template>

<xsl:template match="paragraph[1]" priority="1">
  <fo:block text-indent="0pc" space-after="7pt"
            space-before.minimum="6pt" space-before.optimum="8pt"
            space-before.maximum="10pt">
    <xsl:apply-templates/>
  </fo:block>
</xsl:template>

<xsl:template match="paragraph">
  <fo:block text-indent="2pc" space-after="7pt"
            space-before.minimum="6pt" space-before.optimum="8pt"
            space-before.maximum="10pt">
    <xsl:apply-templates/>
  </fo:block>
</xsl:template>

</xsl:stylesheet>
```

When the above stylesheet is applied to the following document, spacing will resolve as described below the following document.

```
<chapter>
<title>Chapter title</title>
<section>
<title>First section title</title>
<para>Section one's first paragraph.</para>
<para>Section one's second paragraph.</para>
</section>
<section>
<title>Second section title</title>
<para>Section two's only paragraph.</para>
</section>
</chapter>
```

Chapter title appears at the top of the page (its space−before is discarded).

Space between Chapter title and First section title is (8pt,8pt,8pt): the chapter title's space−after has a higher precedence than the section title's space−before (which takes on the initial value of zero), so the latter is discarded.

Space between First section title and Section one's first paragraph is (6pt,6pt,6pt): the section title's space−after has higher precedence than the paragraph's space−before, so the latter is discarded.

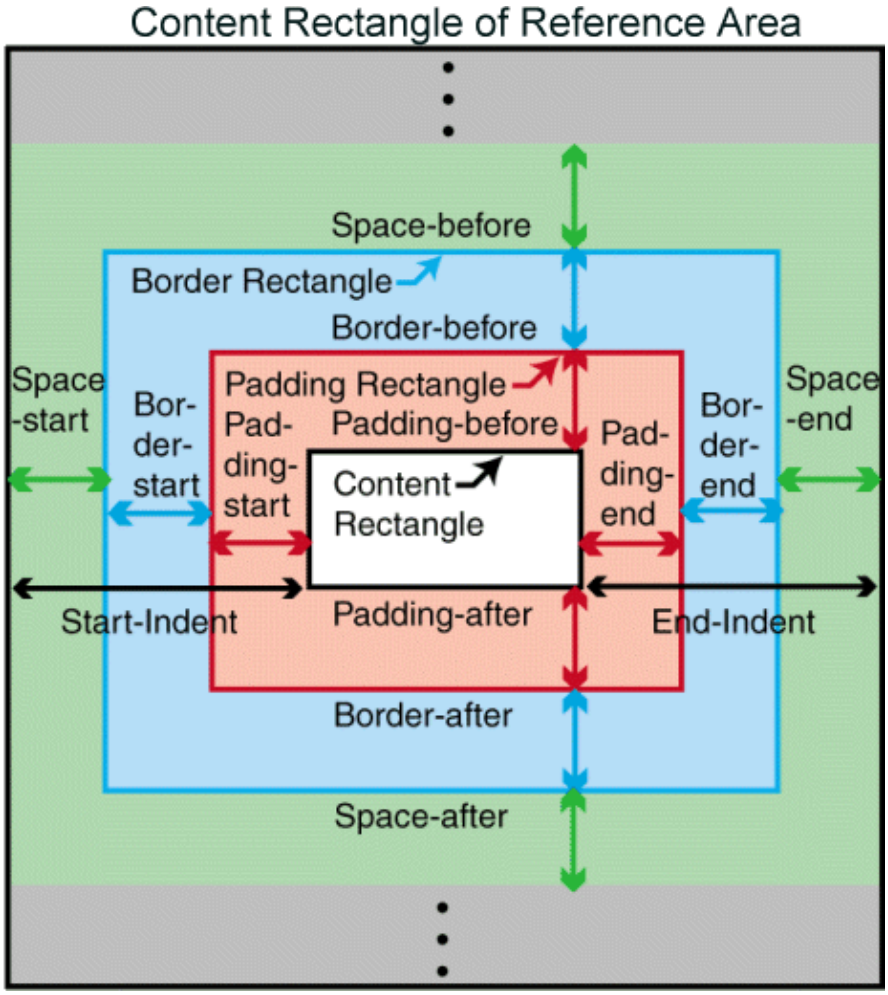
Space between the two paragraphs is (6pt,8pt,10pt): the space−after the first paragraph is discarded because its precedence is equal to that of the space−before the next paragraph, and the optimum of the space−after of the first paragraph is greater than the optimum of the space−before of the second paragraph.

Space between the second paragraph of the first section and the title of the second section is (12pt,12pt,12pt): the space−after the paragraph is discarded because its precedence is equal to that of the space−before of the section title, and the optimum of the space−after of the paragraph is less than the optimum of the space−before of the section title.

The indent on the first line of the first paragraph in section one and the only paragraph in section two is 2pc; the indent on the first line of the second paragraph in section one is zero.

Slide 81: Borders, padding, box model

XSL has a box model that includes CSS's border, padding, and margin concepts.



Slide 82: Other formatting objects

- graphic
- rule
- float, footnote
- simple−link
- multi−switch, multi−case, multi−toggle

Slide 83: Related Links

Other Information

XSL Concepts and Practical Use (http://nwalsh.com/docs/tutorials/xsl/)

An online version of this tutorial

Crane Softwrights Ltd. (http://www.CraneSoftwrights.com/links/training3.htm)
Ken Holman offers a number of commercial tutorials

Specifications

XPath (http://www.w3.org/TR/xpath)
XML Path Language 1.0, W3C Recommendation

XSLT (http://www.w3.org/TR/xslt)
XSL Transformations 1.0, W3C Recommendation

XSL (http://www.w3.org/TR/xsl)
Extensible Style Language 1.0, W3C Proposed Recommendation

XML (http://www.w3.org/TR/1998/REC−xml)
Extensible Markup Language 1.0, W3C Recommendation

Namespaces (http://www.w3.org/TR/REC−xml−names)
Namespaces in XML 1.0, W3C Recommendation

Implementations

XT (http://www.jclark.com/xml/xt.htm)
James Clark's XSLT processor

Lotus XSL (http://www.alphaworks.ibm.com/)
Scott Boag's XSLT processor

FOP (http://www.jtauber.com/fop/)
James Tauber's open source XSL formatter and renderer; see also http://xml.apache.org/ (http://xml.apache.org/)

Slide 84: XSLT Element Index

- <xsl:apply−imports>
- <xsl:apply−templates>
- <xsl:attribute>
- <xsl:attribute−set>
- <xsl:call−template>
- <xsl:choose>

- <xsl:comment>
- <xsl:copy>
- <xsl:copy−of>
- <xsl:decimal−format>
- <xsl:element>
- <xsl:fallback>
- <xsl:for−each>
- <xsl:if>
- <xsl:import>
- <xsl:include>
- <xsl:key>
- <xsl:message>
- <xsl:namespace−alias>
- <xsl:number>
- <xsl:otherwise>
- <xsl:output>
- <xsl:param>
- <xsl:preserve−space>
-

<xsl:processing−instruction>

- <xsl:sort>
- <xsl:strip−space>
- <xsl:stylesheet>
- <xsl:template>
- <xsl:text>
- <xsl:transform>
- <xsl:value−of>
- <xsl:variable>
- <xsl:when>
- <xsl:with−param>

Slide 85: XPath Functions

XPath defines the grammer of location paths and a set of functions:

number last()
: Returns the ordinal value of the last element in the current context

number position()
: Returns the ordinal value of the current element in the current context

number count(node−set)
: Returns the number of nodes in the specified node set

node−set id(object)
: Returns the set of elements with the specified ID value

string local−name(node−set)
: Returns the local name (the name without any namespace qualifiers) of the specified node (or the first node in document order)

string namespace−uri(node−set)

Returns the namespace URI of the specified node (or the first node in document order)

string name(node−set)
Returns the fully qualified name of the specified node (or the first node in document order)

string string(object)
Returns the string representation of the object

string concat(string1, string2, [string3, [string4], ...])
Returns the concatenation of its arguments

boolean starts−with(string1, string2)
Returns true if string1 starts with the characters in string2

boolean contains(string1, string2)
Returns true if string1 contains string2

string substring−before(string1, string2)
Returns the substring of string1 that occurs before the first occurance of string2, or the empty string if not(contains(string1, string2))

string substring−after(string, string)
Returns the substring of string1 that occurs after the first occurance of string2, or the empty string if not(contains(string1, string2))

string substring(string, startpos, [length])
Returns the substring of string starting at position startpos of length length, or the rest of the string if length is not specified

number string−length(string?)
Returns the length of the specified string

string normalize−space(string?)
Returns the specified string with leading and trailing whitespace removed and internal sequences of multiple whitespace characters replaced by a single space

string translate(string1, string2, string3)
Returns string1 with the characters in string2 replaced by the characaters in string3

boolean boolean(object)
Returns the boolean representation of the object

boolean not(boolean)
Returns the negation of boolean

boolean true()
Returns true

boolean false()
Returns false (i.e., not(true())

boolean lang(string)
Returns true if the language of the current context node (as specified by xml:lang) matches the specified language string

number number(object?)
Returns the numeric representation of the object

number sum(node−set)
Returns the sum of the numeric representations of the nodes in the node−set

number floor(number)
Returns the largest integer not greater than number

number ceiling(number)
Returns the smallest integer not less than number

number round(number)
Returns the integer value closest to number

Slide 86: XSLT Functions

XSLT extends the set of available functions with these:

node−set document(object, node−set?)
Allows XSLT to access the nodes in another document

node−set key(keyname, object)
Returns the nodes that match object in the specified key

string format−number(number, formatstring, [decimalformat])
Returns the string representation of number using the formatstring specified and the decimalformat (or the default decimal format)

node−set current()
Returns the current context node

string unparsed−entity−uri(string)
Returns the URI of the specified unparsed entity

string generate−id(node−set)
Returns a string which uniquely identifies the specified node, or the first node in document order

object system−property(string)
Returns the value of the specified system property

boolean element−available(string)
Returns true if the specified element is available in the implementation

boolean function−available(string)
Returns true if the specified function is available in the implementation

---

Slide 87: Top−Level Stylesheet Elements

There are nine top−level XSLT elements:

-

xsl:import and xsl:include

- xsl:strip−space and xsl:preserve−space
- xsl:output
- xsl:key
- xsl:decimal−format
- xsl:namespace−alias
- xsl:attribute−set
- xsl:variable
- xsl:param
- xsl:template

Slide 88: <xsl:stylesheet> or <xsl:transform>

The <xsl:stylesheet> or <xsl:transform> elements identify a complete stylesheet. They are completely synonymous.

```
<xsl:stylesheet
  id = id
  extension−element−prefixes = tokens
  exclude−result−prefixes = tokens
  version = number>
  <!−− Content: (xsl:import*, top−level−elements) −−>
</xsl:stylesheet>
```

id
: A unique identifier

extension−element−prefixes
: Identifies namespace (http://www.w3.org/TR/REC−xml−names) prefixes that are extension element prefixes

exclude−result−prefixes
: Lists namespace (http://www.w3.org/TR/REC−xml−names) prefixes that should not be put in the result tree

version

The XSLT version to which this stylesheet conforms (required)

Slide 89: <xsl:template>

A <xsl:template> constructs a portion of the result tree. It can be called either automatically by the XSLT processor or procedurally by other template(s) in the stylesheet.

```
<xsl:template
  match = pattern
  name = qname
  priority = number
  mode = qname>
  <!-- Content: (xsl:param*, template) -->
</xsl:template>
```

match
: The match pattern for this template

name
: The name of this template

priority
: A priority for conflict resolution

mode
: The mode in which this template occurs

Either a match pattern or a name must be specified.

Slide 90: <xsl:apply−templates>

The <xsl:apply−templates> element indicates where additional templates should be used to construct a portion of the result tree. See also <xsl:call−template>

```
<xsl:apply-templates
  select = node-set-expression
  mode = qname>
  <!-- Content: (xsl:sort | xsl:with-param)* -->
</xsl:apply-templates>
```

select
: Selects the nodes which are to be processed

mode
: Selects the mode in which processing will occur

Slide 91: <xsl:for−each>

A <xsl:for−each> is a loop which processes each of the selected nodes using the template that it contains.

```
<xsl:for-each
```

```
  select = node-set-expression>
  <!-- Content: (xsl:sort*, template) -->
</xsl:for-each>
```

select
: Selects the nodes which are to be processed

Slide 92: <xsl:call-template>

A <xsl:call-template> allows one template to explicitly invoke another template by name. See also <xsl:apply-templates>.

```
<xsl:call-template
  name = qname>
  <!-- Content: xsl:with-param* -->
</xsl:call-template>
```

name
: The name of the template to be called. The stylesheet must contain an <xsl:template> with the specifed name.

Slide 93: <xsl:param>

A <xsl:param> can occur at either the top level or as the first child of a <xsl:template>. It is similar to a <xsl:variable>, but its value may be changed by the calling template or stylesheet environment.

This is analagous to the concept of subroutine parameters in a more traditional programming language.

```
<xsl:param
  name = qname
  select = expression>
  <!-- Content: template -->
</xsl:param>
```

name
: The name of the parameter

select
: Selects the parameter content, if present, otherwise the element content is used

---

Slide 94: <xsl:with-param>

Used inside a <xsl:call-template>, a <xsl:with-param> specifies the value for parameters (<xsl:param>) within the called template.

```
<xsl:with-param
  name = qname
  select = expression>
  <!-- Content: template -->
</xsl:with-param>
```

name

The name of the parameter

select
: Selects the parameter content, if present, otherwise the element content is used

---

Slide 95: <xsl:text>

A <xsl:text> element inserts literal text into the result tree.

```
<xsl:text
  disable-output-escaping = "yes" | "no">
  <!-- Content: #PCDATA -->
</xsl:text>
```

disable-output-escaping
: If "yes", allows unescaped markup characters to be copied into the result tree.

Disabling output escaping *may create result trees that are not well-formed, and are therefore not XML.* Your gun, your bullet, your foot!

Slide 96: <xsl:copy>

```
<xsl:copy
  use-attribute-sets = qnames>
  <!-- Content: template -->
</xsl:copy>
```

use-attribute-sets
: Lists the names of attribute sets (<xsl:attribute-set>) to be copied into the result tree

The actual content of the <xsl:copy> element is copied literally into the result tree.

Slide 97: <xsl:copy-of>

```
<xsl:copy-of
  select = expression />
```

select
: Selects the elements to be copied literally into the result tree

---

Slide 98: <xsl:value-of>

The <xsl:value-of> element inserts the string value of an expression into the result tree.

```
<xsl:value-of
  select = string-expression
  disable-output-escaping = "yes" | "no" />
```

select

The expression to be evaluated; the result is converted to a string

disable−output−escaping
: If "yes", allows unescaped markup characters to be copied into the result tree.

Some examples of <xsl:value−of>:

<xsl:value−of select="3+4"/>
: Inserts 7

<xsl:value−of select="{$varname}"/>
: Inserts the value of the variable `varname`, as a string

<xsl:value−of select="../title[1]"/>
: Inserts the content of the first <title> child of the current context node's parent

---

Slide 99: <xsl:element>

The <xsl:element> allows you to dynamically calculate an element name.

```
<xsl:element
  name = { qname }
  namespace = { uri-reference }
  use-attribute-sets = qnames>
  <!-- Content: template -->
</xsl:element>
```

name
: The element name

namespace
: The namespace URI reference for the element

use−attribute−sets
: Lists the names of attribute sets (<xsl:attribute−set>) to be copied into the result tree

An example of <xsl:element> in use:

```
<xsl:template name="process.table.cell">
  <xsl:param name="cellgi">td</xsl:param>
  <xsl:element name="{$cellgi}">
    <!-- whatever processing is desired -->
  </xsl:element>
</xsl:template>
```

Slide 100: <xsl:attribute>

The <xsl:attribute> element allows you to dynamically or conditionally add attributes.

```
<xsl:attribute
  name = { qname }
  namespace = { uri-reference }>
  <!-- Content: template -->
</xsl:attribute>
```

name
: The attribute name

namespace
: The namespace URI reference for the attribute

An example of <xsl:attribute> being used conditionally:

```
<xsl:template name="graphic">
  <img src="{@fileref}">
    <xsl:if test="@align">
      <xsl:attribute name="align">
        <xsl:value-of select="@align"/>
      </xsl:attribute>
    </xsl:if>
  </img>
</xsl:template>
```

Note that the <img> tag has logical content in the stylesheet. The serializer is expected to render this appropriately for HTML or XML.

Slide 101: <xsl:if>

The <xsl:if> element is a simple "if" conditional:

```
<xsl:if
  test = boolean-expression>
  <!-- Content: template -->
</xsl:if>
```

test
: The test expression

If the test expression returns a true value, the content of the <xsl:if> element is evaluated and inserted into the result tree. Otherwise, the <xsl:if> element and all of its contents are ignored.

Slide 102: <xsl:choose>

For more complex conditionals, <xsl:choose> is a multi-way switching element.

```
<xsl:choose>
  <!-- Content: (xsl:when+, xsl:otherwise?) -->
</xsl:choose>
```

The content of the first <xsl:when> element with a test expression that evaluates to true is inserted into the result tree. If none of them evaluate to true, the content of the <xsl:otherwise> element is inserted.

Slide 103: <xsl:when>

A <xsl:when> element is part of a <xsl:choose> statement.

```
<xsl:when
```

```
  test = boolean-expression>
  <!-- Content: template -->
</xsl:when>
```

test
: The test expression

If the test expression returns a true value, the content of the <xsl:when> element is evaluated and inserted into the result tree.

Slide 104: <xsl:otherwise>

An <xsl:otherwise> element is part of a <xsl:choose> statement.

```
<xsl:otherwise>
  <!-- Content: template -->
</xsl:otherwise>
```

If none of the <xsl:when> elements evaluate to true, the content of the <xsl:otherwise> element is inserted into the result tree.

Slide 105: <xsl:message>

A <xsl:message> is a mechanism for the stylesheet author to communicate with the person running the stylesheet.

```
<xsl:message
  terminate = "yes" | "no">
  <!-- Content: template -->
</xsl:message>
```

terminate
: If "yes", the stylesheet processor is expected to stop processing after displaying the message

The content of the <xsl:message> element may be presented to the user in an application-dependent fashion. This could be a terminal message, a browser popup, nothing more than writing it to a log file or simply terminating the program without displaying the message at all (for example, in a server-side environment).

Slide 106: <xsl:variable>

A <xsl:variable> is a named storage location holding the results of a runtime evaluation of the variable content or select expression

```
<xsl:variable
  name = qname
  select = expression>
  <!-- Content: template -->
</xsl:variable>
```

name
: The variable name

select
: Selects the variable content, if present, otherwise the element content is used

Slide 107: <xsl:sort>

The <xsl:sort> element sorts a set of nodes.

```
<xsl:sort
  select = string-expression
  lang = { nmtoken }
  data-type = { "text" | "number" | qname-but-not-ncname }
  order = { "ascending" | "descending" }
  case-order = { "upper-first" | "lower-first" } />
```

select
: Selects the sort expression; this is evaluated for each node in the sort node set

lang
: The language to use for sorting purposes

data−type
: The sorting data−type

order
: The sort order

case−order
: Do upper−case letters sort before lower−case, or vice−versa?

The sorting data type determines how selected values are sorted; for example "02" sorts before "1" if "text" is used, after if "number" is used. Specifying any other value invokes a processor−dependent extension.

Slide 108: <xsl:number>

The <xsl:number> element counts elements in the source tree (or takes a numeric expression) and inserts a text representation of the resulting number into the result tree.

```
<xsl:number
  level = "single" | "multiple" | "any"
  count = pattern
  from = pattern
  value = number-expression
  format = { string }
  lang = { nmtoken }
  letter-value = { "alphabetic" | "traditional" }
  grouping-separator = { char }
  grouping-size = { number } />
```

level
: Identifies the counting level

count
: Selects the element(s) to count

from
: Selects the location where counting should begin

value
: A numeric expression to convert to a text representation

format
: The format string

lang
: The language to use for conversion

letter−value
: Selects alphabetic or traditional letter values (in English, alphabetic is the traditional sequence, but that is not true in all languages)

grouping−separator
: The grouping separator character (",")

grouping−size
: The number of digits in a group (3)

Slide 109: <xsl:decimal−format>

A <xsl:decimal−format> element defines the default decimal format for number to text conversion.

```
<xsl:decimal−format
  name = qname
  decimal−separator = char
  grouping−separator = char
  infinity = string
  minus−sign = char
  NaN = string
  percent = char
  per−mille = char
  zero−digit = char
  digit = char
  pattern−separator = char />
```

name
: An optional name that can be used to refer to this decimal format; if no name is given, this declaration sets the default decimal format

decimal−separator
: The decimal separator character ("." for example)

grouping−separator
: The grouping separator character (",")

infinity
: The string to use to represent infinity ("Infinity")

minus−sign
: The character to use to represent negative numbers ("−")

NaN
: The string to use to represent "not a number" ("NaN")

percent
: The percent character ("%")

per−mille
: The per−mille character ("")

zero−digit
: The character used in format strings to represent a digit where a missing digit will show as this character ("0")

digit
: The character used in format strings to represent a digit where a missing digit will be absent ("#")

pattern−separator
: The character used to separate format strings (";")

All of these attributes, and the operation of the format number function, are interpreted according to the Java JDK 1.1 definition of the DecimalFormat (http://java.sun.com/products/jdk/1.1/docs/api/java.text.DecimalFormat.html) class.

Slide 110: <xsl:key>

The <xsl:key> element declares a key for subsequent use with the key() function.

```
<xsl:key
  name = qname
  match = pattern
  use = expression />
```

name
: The name of this key

match
: Selects the nodes which are in this key set

use
: Selects the expression within each node that identifies the key

The <xsl:key> element is only useful in conjunction with the key function:

node−set key(name, object)
: Returns the nodes that match object in the specified key

Slide 111: <xsl:fallback>

The <xsl:fallback> element is evaluated if an extension element cannot be interpreted.

```
<xsl:fallback>
  <!−− Content: template −−>
```

```
</xsl:fallback>
```