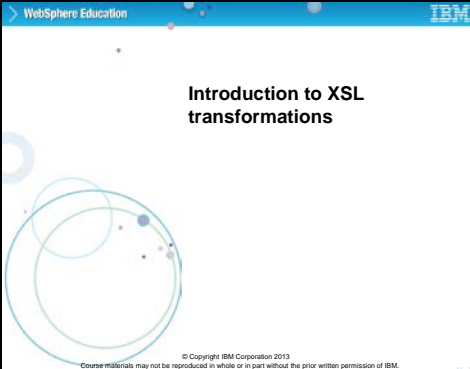


Slide 1



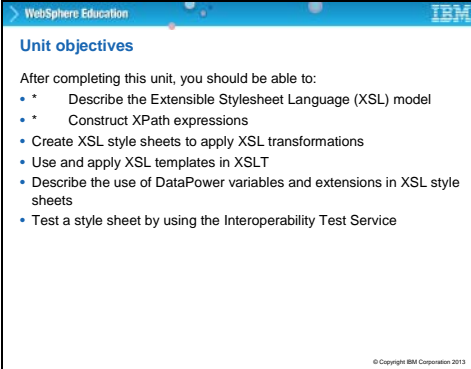
WebSphere Education

Introduction to XSL transformations

© Copyright IBM Corporation 2013
Course materials may not be reproduced in whole or in part without the prior written permission of IBM.

The slide features a blue header bar with the text 'WebSphere Education' on the left and the IBM logo on the right. The main content area is white with the title 'Introduction to XSL transformations' centered. On the left side, there is a decorative graphic consisting of several overlapping circles in shades of blue and green, with small dots scattered around them. At the bottom, there is a small copyright notice: '© Copyright IBM Corporation 2013' and 'Course materials may not be reproduced in whole or in part without the prior written permission of IBM.'

Slide 2



WebSphere Education

Unit objectives

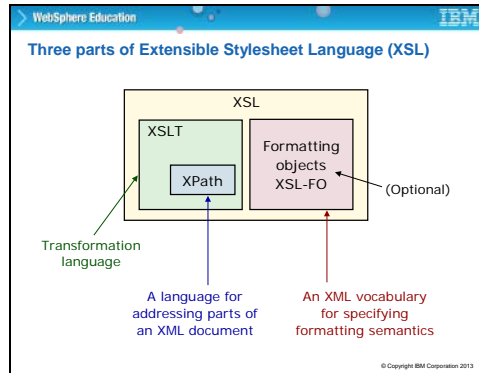
After completing this unit, you should be able to:

- * Describe the Extensible Stylesheet Language (XSL) model
- * Construct XPath expressions
- Create XSL style sheets to apply XSL transformations
- Use and apply XSL templates in XSLT
- Describe the use of DataPower variables and extensions in XSL style sheets
- Test a style sheet by using the Interoperability Test Service

© Copyright IBM Corporation 2013

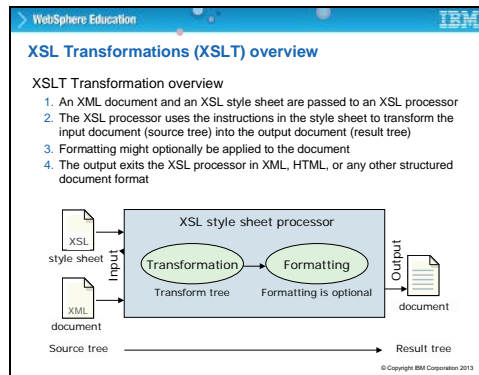
Unit overview

This unit looks at the structure of XSL transformations and how they are used in DataPower.



Three parts of Extensible Stylesheet Language (XSL)

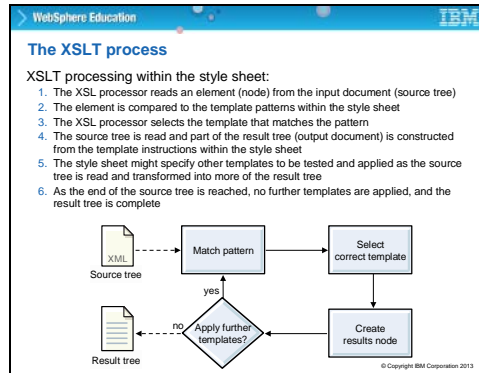
Extensible Stylesheet Language has three parts, two of which are discussed in this presentation. First, there is the transformation language, XSLT, which provides a mechanism for associating the parts of an XML file with whatever code is to provide the transformation. Next, there is XPath, which provides a way of addressing nodes or attributes in the tree of a document object. The third part is the one you are not going to be looking at: XSL-FO, or formatting objects, which is used to format information for documents, for example, to create a PDF file.



XSL Transformations (XSLT) overview

The section is the high-level overview of what is happening. All XSL transformations require two files: a style sheet that holds the content on how to run the transformation, and an XML source document. The style sheet is used to parse through the XML source file, or document, and to provide output as a function of what it finds. This output might optionally be passed to a formatter (the third part of the XSL environment), and the result is then output.

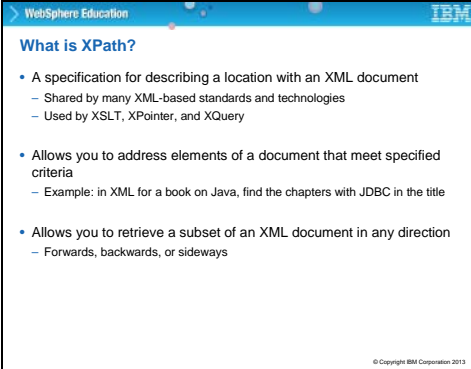
Slide 5



The XSLT process

So here you see at a slightly more fine-grained level what the transformation process is doing. At the upper left of this slide, you see the XML source file, or source tree, and the XSL style sheet. The style sheet looks for a match in the source file to a pattern listed in the XSLT file. If one is found, a corresponding template is applied and a result node created. If there are other patterns that are listed in the XSLT file, the source is searched again for a match, and so on, until there are no more matches. This process can either find discrete matches, or nested matches. When there are no more matches, the result tree is complete.

Slide 6



WebSphere Education

What is XPath?

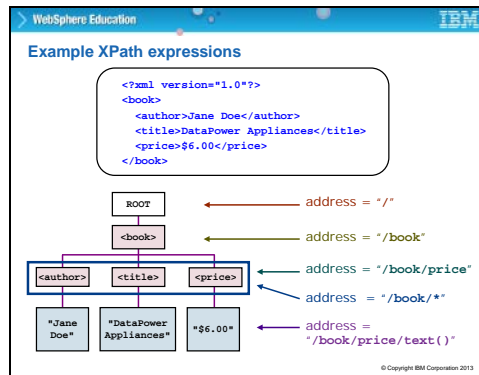
- A specification for describing a location with an XML document
 - Shared by many XML-based standards and technologies
 - Used by XSLT, XPointer, and XQuery
- Allows you to address elements of a document that meet specified criteria
 - Example: in XML for a book on Java, find the chapters with JDBC in the title
- Allows you to retrieve a subset of an XML document in any direction
 - Forwards, backwards, or sideways

© Copyright IBM Corporation 2013

What is XPath?

XPath is a powerful language for finding paths through an XML document. This might be a single node (for example, find the node with id=123). A branch of the tree (for example, the child of the node with id=123). Or a collection of nodes in different parts of the tree (the example that is given on the slide is “any chapter with JDBC in the title”).

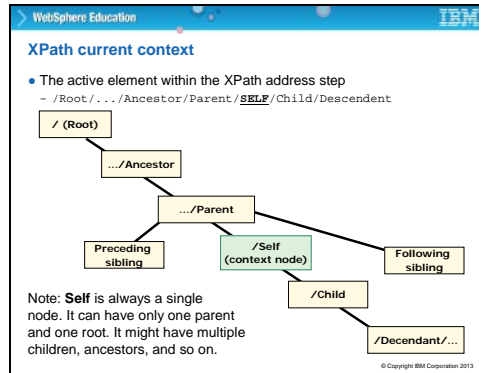
Slide 7



Example XPath expressions


An XML document has a root, a single point from which everything descends. Every other node is a child of the root node. The root node is indicated with the single slash. In the example on the slide, `/book` means "the first child under the root, which is called book". Going one step further, book has three child processes, and each one can be distinguished by specifying the path from the root. Thus, price is a child of book, but so is title and author. You can use the asterisk to encompass all of the child processes of book. The last example shows how to use a function to extract the string data from the price node.

Slide 8



XPath current context

All the examples covered up to this point used an absolute path, indicated by an initial slash, starting from the root node. Starting from the root node might be inconvenient, or even impossible to determine. It is often much better to have a relative path that starts from whatever node the processing happens to be at a particular moment. This node is called 'self'. This node can have one parent; and therefore ultimately one root, but it can have any number of peer nodes, or siblings, and any number of child processes.

WebSphere Education


XPath step syntax

- An XPath location path is made up of one or more steps that are separated by a forward slash (/)
- Each step within the path consists of a:
 - Axis:** Branch of the node tree relative to the current context node
 - Use keywords such as: ancestor, attribute, child, descendant, and so on
 - NodeTest:** Consists of the node name that is used to test node for inclusion
 - Predicate:** Optional filter of matched nodes
- Abbreviated syntax is allowed for several different axes
 - "child:." has an empty default as it is the default axis
 - "/child:catalog/child:tools/" is the same as "/catalog/tools/"
- Expression shortcuts
 - "//element]" selects element node regardless of location
 - "." selects the current node
 - ".." selects the parent of the current node
 - "@{attribute-name}" selects an attribute
- Example:
 - Locate all titles in the book that contain the string 'XPath'
 - /book/child:title[contains(text(),'XPath')]/

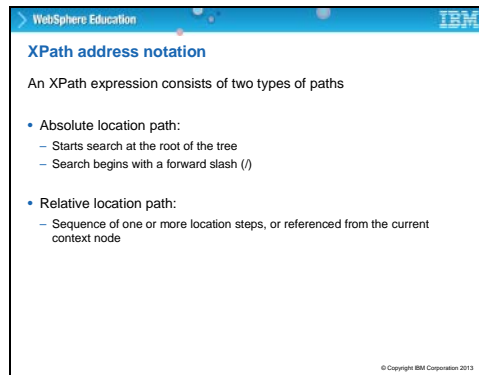
.../axis::nodetest[predicate]/...

© Copyright IBM Corporation 2013

XPath step syntax

This is a good time to sum up the syntax covered so far, and then look at several examples. The first thing to point out here is the abbreviated syntax. Nowhere is the full syntax with the word 'child' and the two colons! The example that is given is typical: rather than child-colon-colon-catalog, you see 'catalog'. There are other shortcuts that are listed on the slide, and you should become familiar with them as they are commonly used. The single dot designates the 'self' node. The parent is the double dot.

Path expressions contain functions also as the last example shows. There are two functions, the text() function, which is here used as a parameter to the contains() function.



The slide is titled "XPath address notation" and is part of a presentation from WebSphere Education, as indicated by the header. It explains that an XPath expression consists of two types of paths: absolute and relative. The absolute path starts at the root of the tree and begins with a forward slash. The relative path is a sequence of location steps referenced from the current context node.

WebSphere Education

XPath address notation

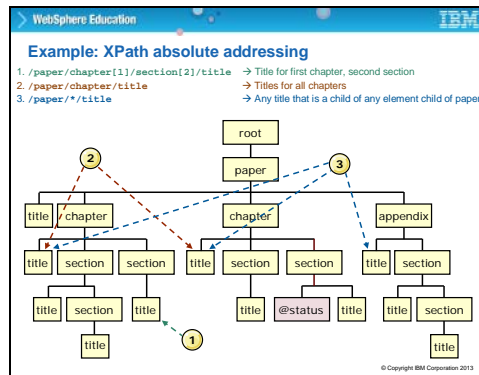
An XPath expression consists of two types of paths

- Absolute location path:
 - Starts search at the root of the tree
 - Search begins with a forward slash (/)
- Relative location path:
 - Sequence of one or more location steps, or referenced from the current context node

© Copyright IBM Corporation 2013

XPath address notation

You looked at two types of path, the absolute location and the relative location. The absolute path starts from the root node and moves down to the child nodes. You can always recognize it by the leading slash. The relative path starts from the current node and moves in any direction – up down and sideways, or parent, child, and sibling.

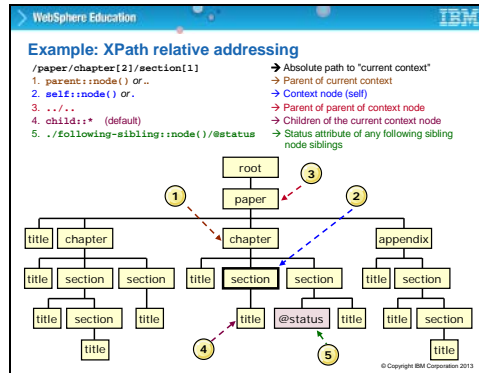


Example: XPath absolute addressing

This slide gives three examples of absolute addressing. You can determine addressing from the leading slash of each example. In the first, the child of the root is paper, which itself has four child nodes. The required node is chapter, but note that there are two chapter nodes. There is more precise information therefore; which is given by the predicate in square brackets, [1]. This node in turn has two identically named child nodes and so it too requires a predicate. Finally, the child node that is called title is given.

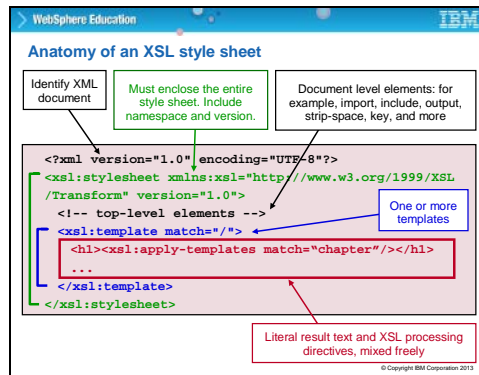
In the second example, there is no predicate for chapter; and therefore both chapter nodes are selected. The selection means that there are two final nodes returned, the title nodes of the two chapters.

In the third example, the asterisk is used to indicate every child node. There are therefore four possibilities with a child node of title (remember that the child node of paper that is called title is a sibling of the chapter nodes, not a child).



Example: XPath relative addressing

Here you see several examples of relative path addressing. The first line is in fact an absolute path that shows how the node called section might become the current context. The parent of this current context can be accessed by using the double dot. And by extension you can climb the tree again by repeating the parent double dot as often as you need, as is shown in example 3. All the child processes of the current node can be designated by using the asterisk, as in example 4. And finally the fifth example says "Starting from self, go to the next node, which is at this same level, and return its attribute, which is called status". There might be several attributes for a node, and so you name the one you want.



Anatomy of an XSL style sheet

That was a look at the XPath language; now it is time to examine the XML style sheet language and see how XPath applies to it. On this slide, you can see the fundamental format. The first line is a prolog that is not specific to XSL but prefaces all XML-type files. It provides two pieces of information, the XML level and the encoding that is used in the file. Version 1 and encoding UTF-8 are defaults, so if you do not include this prolog, those values are assumed. However, it is good practice to include this line because if your document is checked for validity (which it is in DataPower) it fails the check if the line is missing.

The second line of code states that the object is a style sheet and gives the namespace and version. The object precedes any elements that are required by the document. For example, you might be importing an auxiliary style sheet. Next, you find the template that tries to match some element in your XML file. Remember that the leading slash indicates an absolute path that starts from the root. Therefore, `match="/"` says "Go find where the XML document begins". Having positioned yourself at that top element, you start digging down looking for other matches. In this case, you are looking for a node called chapter. If found, the template for that node is applied.

You can mix literal text and template directives. In this example, there is an HTML header tag `<H1>` that precedes the template, and its corresponding closing tag after.

WebSphere Education

IBM

The <xsl:template> element

A style sheet has one or more template tags with the structure:

```
<xsl:template match="match expression">
  <!-- literal result text or XSLT elements -->
</xsl:template>
```

Specifies:

- A **match expression** defines when the template is called
 - An XPath expression
 - Test against the nodes in the XML source tree
- Literal result text is written to the output tree or XSLT elements are executed

© Copyright IBM Corporation 2013

The <xsl:template> element

The template element is the foundation of the style sheet transformation process. It contains the rules that must be applied when a matching node is found. There are four possible attributes, but you are going to concentrate on just one, 'match'. The match defines when the block of code between the template tags should be ran. The match expression is given in XPath format.

There can be a mix of literal elements and processing elements in the template.

WebSphere Education

The `<xsl:apply-templates>` element

- Looks for a matching template rule in the XSL style sheet
 - Each child of the current node in the XML source tree is evaluated for a matching template rule
- The rules that can be matched are:
 - None - it is not required to have a template rule for each child node
 - The template match rules that you define by using the `select` attribute

```
<xsl:template match="paper":  
  <h1><xsl:apply-templates select="chapter" /></h1>  
  ...  
</xsl:template>
```

Current node

Child of current node

© Copyright IBM Corporation 2013

The `<xsl:apply-templates>` element

The `apply-templates` element is used to nest template matches. The element can be seen in the example that is given on this slide. The template match is 'paper'. The location becomes the current node for any further nested template matches. The `apply-templates` tag has an attribute 'select' which has a value of 'chapter'. The processing looks for any chapter nodes that are child processes of the paper node. It is also possible to have an `apply-templates` element with no attributes. The tag would read `xsl:apply-templates`. In this case, any template match that is found is processed, in the order in which they are written.

WebSphere Education

IBM

The <xsl:value-of> element

- <xsl:value-of select="patternToMatch"/>
 - Used to extract a specific value from the source tree
 - Inserts literal values into result tree as a string, element, or attribute from *patternToMatch*
- Example:

```
<list>
  <book ID = "999">
    <author>Dan Big</author>
    <title>Large Stories</title>
    <price>$7.00</price>
  </book>
</list>
```

Result

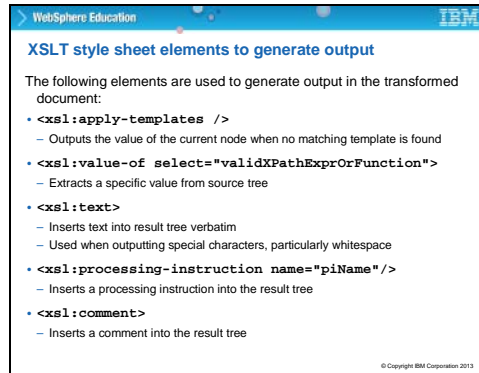
```
<td>Large Stories</td>
```

```
<xsl:template match="/list/book">
  <td><xsl:value-of select="title"/></td>
</xsl:template>
```

© Copyright IBM Corporation 2013

The <xsl:value-of> element

The value-of element can be used to pull a value out of a node and write it to the output. In the example, there is an XML file that has a root element 'list', with one child element 'book', which in turn has three child processes, 'author', 'title' and 'price'. The XSLT template looks for a match on the absolute path '/list/book'. When this node is located, the literal string '<td>' is written to the output. The 'select' attribute now looks for a match on 'title'. This is a child of the current node (it is not an absolute path). So the title node must be at /list/book/title. When located, it is the value that is extracted, or the string 'Large Stories' in this example. The content is written to the output together with the terminating literal string that closes the table data cell.



WebSphere Education

XSLT style sheet elements to generate output

The following elements are used to generate output in the transformed document:

- **<xsl:apply-templates />**
 - Outputs the value of the current node when no matching template is found
- **<xsl:value-of select="validXPathExprOrFunction">**
 - Extracts a specific value from source tree
- **<xsl:text>**
 - Inserts text into result tree verbatim
 - Used when outputting special characters, particularly whitespace
- **<xsl:processing-instruction name="piName" />**
 - Inserts a processing instruction into the result tree
- **<xsl:comment>**
 - Inserts a comment into the result tree

© Copyright IBM Corporation 2013

XSLT style sheet elements to generate output

Here is a summary of the elements covered: each of the elements generate some output string.

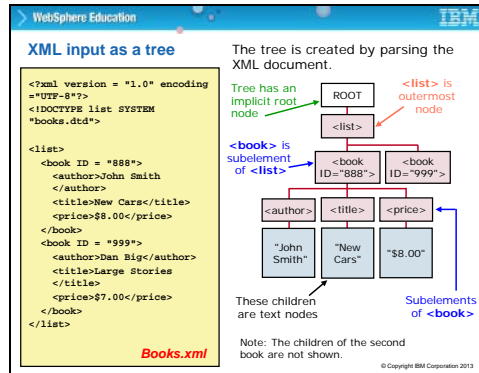
The apply-templates element can be used without attribute (as you can see here) or it can be limited by specifying a match.

value-of was covered in the previous slide. It extracts the value of the node that is defined by the XPath expression of the select attribute.

'text' element can be used when you output some specific, pre-defined string, which might be white space.

'processing-instruction' writes an instruction to the output. For example, the, you might add a reference to a cascading style sheet in the output document.

The last example inserts the string that is contained in the comment element as an XML comment in the output, which is surrounded with the opening '<!--' and the closing '-->'.



XML input as a tree

You covered tags, and trees, and nodes, and so on. Here is how these things relate to each other.

Starting with the XML document to the left, the root element is 'list'. This element is the first to be opened, and is; therefore the last to be closed. There can be one root element. 'list' has a child element 'book'. In fact, there are two book elements, which are therefore siblings. And each book element has three sibling child elements.

To the right is the tree representation of this document. The list node is first under the root, and then the other nodes are arranged to reflect the document structure.

The lowest nodes represent the text strings that are held within the innermost tags.

Attributes are not counted as node elements but are grouped together with the node to which they refer.

Wanted HTML output

```
<html>
<head><title>Book List </title></head>
<body>
<h1>Book List</h1>
<table border="1" cols="3" width="100%">
<tbody>
<tr>
<td>888</td>
<td>New Cars</td>
<td>$8.00</td>
</tr>
<tr>
<td>999</td>
<td>Large Stories</td>
<td>$7.00</td>
</tr>
</tbody>
</table>
</body>
</html>
```

The HTML that is produced must be well-formed.

Data taken from the XML document (nodes)

Book List		
888	New Cars	\$8.00
999	Large Stories	\$7.00

© Copyright IBM Corporation 2013

Wanted HTML output

The next five slides give an example of the transformation of an XML document into an HTML document. This slide shows the wanted HTML output. At the top is the HTML source code. At the bottom is the graphical output that would be seen on a browser. Most of this code is purely HTML and are defined as strings in the XSLT file. The only information from the original XML file is the contents of the data cells in the HTML table. There is much HTML code before the piece of XML data is reached. Remember this for the next slide!

The slide is titled "XML to HTML (1 of 4)" and is part of a presentation from "WebSphere Education". It illustrates the first step of an XSLT transformation process.

Books.xml (XML input):

```
<list>
  <book ID = "888">
    <author>John Smith</author>
    <title>New Cars</title>
    <price>$8.00</price>
  </book>
  ...
</list>
```

Books.xsl (XSLT stylesheet):

```
<?xml version="1.0" ?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <html>
      <head><title>Book List</title></head>
      <body>
        <table border="1" cols="3" width="100%" >
          <tbody>
            <xsl:apply-templates />
          </tbody>
        </table>
      </body>
    </html>
  </xsl:template>
  ...
  (remaining templates are omitted for clarity) </xsl:stylesheet>
```

HTML output (Result of the transformation):

```
<html>
  <head><title>
    Book List
  </title></head>
  <body>
    <table border="1"
      cols="3"
      width="100%">
      <tbody>
```

Explanatory text:

The processor looks for a `<xsl:template match = "/">` tag, which matches the root element `<list>`.

It copies non-XSLT elements to the output tree in first template so you get the first part of your HTML.

XML to HTML (1 of 4)

The first match is the root element, which is designated by the single slash. The search finds the XML tag `<list>`. The first task now is to copy the text elements to the output, starting with the HTML tag. There is much HTML code that is placed in this root match template: remember from the previous slide. The HTML tag must be opened, as must the `<body>` tag and the `<table>` tag. There is also a `<tbody>` tag, although in the present example it is not necessary. The processing that was just mentioned is all that this template match does for the moment. It now hands off to any other templates there might be with the directive 'apply-templates'. Before moving to the next slide, there is one last thing to notice. When all templates are applied, there remains one task: close all the HTML tags that were opened at the beginning of this template. They are closed in reverse order to maintain correct nesting.

The diagram illustrates the transformation of XML to HTML using XSL-FO. It is titled "XML to HTML (2 of 4)" and includes the IBM logo.

Books.xml (Source XML):

```
<list>
  <book ID = "888">
    <author>John Smith</author>
    <title>New Cars</title>
    <price>$8.00</price>
  </book>
  ...
</list>
```

Books.xsl (XSL Template):

```
<xsl:template match="/*">
  ...
  <xsl:apply-templates />
</xsl:template>

<xsl:template match="book">
  <tr>
    <td><xsl:value-of select="@ID" /></td>
    <xsl:apply-templates
      select="title|price" />
    </td>
  </tr>
</xsl:template>
```

HTML output (Result):

```
<html>
<head><title>Book List
</title></head>
<body>
  <table borders="1"
    cols="2"
    width="100%">
    <tbody>
      <tr>
        <td>888</td>
```

Explanatory Text:

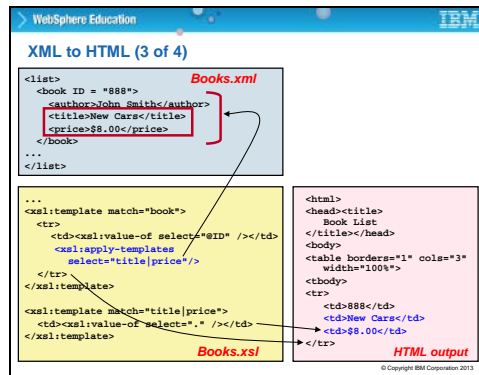
Inside the `<xsl:template match="/*">` tag, the `<xsl:apply-templates/>` tag looks for templates for the children of 'list' (that is, `<book>`). It finds `<xsl:template match="book">`, and processes that template.

`<xsl:value-of select="@ID">` writes the value of the attribute ID to the output tree.

XML to HTML (2 of 4)

The next template match is on the child tag of `<list>`, which is the `<book>` tag. There might be several books, and each is in its own row in the HTML document. Therefore, there is a table row tag and table data cell tags for each occurrence of a book node. The first thing that you can see in the template match is therefore the row and cell tags opening.

The attribute 'ID' is the first piece of data that is taken from the XML document. Since it is an attribute, the select is made on '@ID'. Next, the HTML document needs either the title or the price. The delineation is shown by the pipe symbol, the vertical line, between the words title and price. You can read the content as 'select title or price'. If the XML file is well-formed, these elements always appear in this order. If the XML document presents the two elements in either order, then they are picked up and placed in the HTML output in either order, which is not what you want!



XML to HTML (3 of 4)

The next template match is for the title and the price elements, whichever is found first. When found, the cell tag is written to the output and the value of the element that is designated by a dot selected. You saw this dot shortcut a few slides back. It is used as a reference to the current node, or the 'self' node.

XML to HTML (4 of 4)

The processor now looks for and finds another `<book>` node to process. Output for that `<book>` node is added to the output tree.

Books.xml

```
<list>
...
<book ID = "999">
  <author>Dan Sig</author>
  <title>Large Stories</title>
  <price>$7.00</price>
</book>
</list>
```

Books.xsl

```
<xsl:template match="book">
  <tr>
    <td><xsl:value-of select="@ID" /></td>
    <xsl:apply-templates
      select="title|price"/>
    </tr>
</xsl:template>

<xsl:template match="title|price">
  <td><xsl:value-of select="." /></td>
</xsl:template>
```

(Other templates have been omitted for clarity)

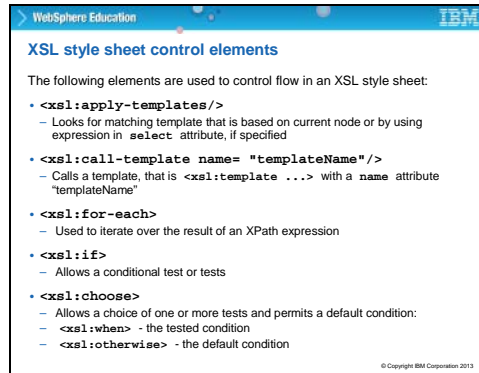
HTML output

```
<html>
..
<tr>
  <td>888</td>
  <td>New Cars</td>
  <td>$8.00</td>
</tr>
<tr>
  <td>999</td>
  <td>Large Stories</td>
  <td>$7.00</td>
</tr>
```

© Copyright IBM Corporation 2013

XML to HTML (4 of 4)

The apply-template match occurs as many times as there are nodes that match. In this case, there is two, a title and a price, and so the match is run twice. When all templates are matched, the transformation process returns to the parent match, which in this case was 'book'. There is one last piece of text to add to the output, the closing row tag. The search continues to find any sibling nodes; in other words, there might be more books that are listed in the XML file. The next book is found, with the ID 999, and the apply-template process starts again. The search then continues for another book, and if none is found control is handed back to the parent template (the one in which this apply-templates directive is nested). As you saw on the first of these example slides, the parent template, book, then writes out the final closing HTML tags, thus completing the HTML file.



WebSphere Education

XSL style sheet control elements

The following elements are used to control flow in an XSL style sheet:

- **<xsl:apply-templates/>**
 - Looks for matching template that is based on current node or by using expression in **select** attribute, if specified
- **<xsl:call-template name= "templateName"/>**
 - Calls a template, that is **<xsl:template ...>** with a **name** attribute "templateName"
- **<xsl:for-each>**
 - Used to iterate over the result of an XPath expression
- **<xsl:if>**
 - Allows a conditional test or tests
- **<xsl:choose>**
 - Allows a choice of one or more tests and permits a default condition:
 - **<xsl:when>** - the tested condition
 - **<xsl:otherwise>** - the default condition

© Copyright IBM Corporation 2013

XSL style sheet control elements

There are several elements to control processing direction in an XSL transformation. You saw the `apply-templates` directive several times already. 'call-template' does just that: it calls the template that is indicated in the name attribute. The next three (`for-each`, `if`, and `choose`) are described in detail on the following slides.

WebSphere Education

IBM

The <xsl:for-each> element

- <xsl:for-each select="nodeSetExpression">
 - Used to iterate over the result of the select expression
 - Selected node becomes the current node

```

<list>
  <book ID="666">
    <author>Jim Blue</author>
    <title>Blue Flowers</title>
  </book>
  <book ID="888">
    <author>John Smith</author>
    <title>New Cars</title>
  </book>
  <book ID="999">
    <author>Dan Big</author>
    <title>Large Stories</title>
  </book>
</list>
            
```

Books.xml

```

<xsl:template match="/*">
  <xsl:for-each select="//book">
    <p><xsl:value-of
      select="title"/></p>
  </xsl:for-each>
</xsl:template>
            
```

Books.xsl

```

<p>Blue Flowers</p>
<p>New Cars</p>
<p>Large Stories</p>
            
```

Books.html

© Copyright IBM Corporation 2013

The <xsl:for-each> element

The for-each element iterates over the selected node. Look at the example that is shown upper right on the slide. The iteration is over the book node, where ever it might be found. (The double-slash in front of book in the select attribute indicates – remember the shortcuts that you saw at the beginning of this presentation.) For each book node that is found, the requirement is to write the text string '<p>' to the output; then select the value of the title node. The book node becomes the current node, and so title is relative to that. The iteration occurs as many times as there are matching nodes that are found.

WebSphere Education IBM

The <xsl:if> element

- <xsl:if test="patternToMatch">
 - Used to conditionally process the matched expression
 - Can also be used with the logical not statement

```
<list>
  <book ID="666">
    <author>Jim Blue</author>
    <author>Mike Yellow</author>
    <author>Dan Farm</author>
    <title>Blue Flowers</title>
  </book>
</list>
```

Books.xml

Blue Flowers by Jim Blue, Mike Yellow, and Dan Farm

```
<xsl:template match="list/book">
  <xsl:value-of select="title"/> by
  <xsl:for-each select="author">
    <xsl:value-of select="."/ />
    <xsl:if test="position()=last()">, </xsl:if>
    <xsl:if test="position()=last()-1"> and </xsl:if>
  </xsl:for-each>
</xsl:template>
```

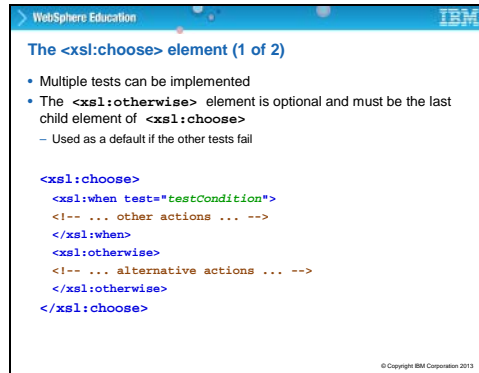
Books.xsl

© Copyright IBM Corporation 2013

The <xsl:if> element

The 'if' element checks for a condition, and if it finds a match, it runs the corresponding directive, or writes the text string to the output. Look at the XML file on the slide, books.xml. You can see that there are three authors. The requirement is to write a string that contains the title of the book followed by the authors, with the format 'a,b, and c'.

The template match is for the book node. The first piece of information is the title, and that is a simple select of the text value of the node. The processing follows a loop on the node 'author'. The value of this node is selected the dot shorthand notation), then there are two tests to see whether there should be a comma or the word 'and'. These tests are repeated for each author node that is found.



WebSphere Education

The `<xsl:choose>` element (1 of 2)

- Multiple tests can be implemented
- The `<xsl:otherwise>` element is optional and must be the last child element of `<xsl:choose>`
 - Used as a default if the other tests fail

```
<xsl:choose>
  <xsl:when test="testCondition">
    <!-- ... other actions ... -->
  </xsl:when>
  <xsl:otherwise>
    <!-- ... alternative actions ... -->
  </xsl:otherwise>
</xsl:choose>
```

© Copyright IBM Corporation 2013


The `<xsl:choose>` element (1 of 2)

The choose element is the parent to two other elements: 'when' and 'otherwise'. In the simplest case, there is a choice and a when in this format:

```
<xsl:choose><xsl:when test="condition-to-test"></xsl:when></xsl:choose>.
```

There can be as many 'when' elements as required. The 'otherwise' element is optional, but if it is present there can be one and it must be the last one.

The next slide shows an example.

WebSphere Education 

The <xsl:choose> element (2 of 2)

```
<list>
  <book ID="666">
    <chapter>First Chapter</chapter>
    <chapter>Second Chapter</chapter>
    <appendix>XSLT reference</appendix>
  </book>
</list>
```

Books.xml

```
<p>Chapter: First Chapter</p>
<p>Chapter: Second Chapter</p>
<p>Appendix: XSLT reference</p>
```

Books.html

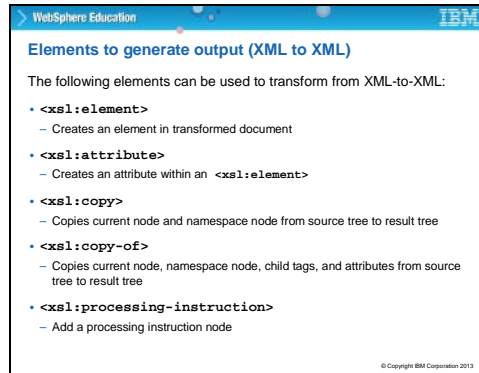
```
<xsl:stylesheet xmlns:xsl="..."
  <xsl:template match="//book">
    <xsl:for-each select="*">
      <p>
        <xsl:choose>
          <xsl:when test="name()='chapter'">Chapter: </xsl:when>
          <xsl:when test="name()='appendix'">Appendix: </xsl:when>
          <xsl:otherwise>Index: </xsl:otherwise>
        </xsl:choose>
        <xsl:value-of select="." />
      </p>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

Books.xsl

© Copyright IBM Corporation 2015

The <xsl:choose> element (2 of 2)

This slide shows an example of the xsl:choose element. In the XML file, which is shown upper left, there are a number of child elements in the book. The processing should be different according to whether there is a chapter, an appendix, or an index. The required output is shown upper right. The XSLT file starts by setting the current node to any book node (the double slash book match). From here, a for-each loop is set up that iterates over everything (the asterisk value for the select). An HTML paragraph tag is written to the output; then there is a choice that is set up. The condition for the test is read like: "the name of the selected node is 'chapter' ", or "the name of the selected node is 'appendix' ". The name() function here looks to the current node (selected in the for-each loop) and extracts the name value of that node. There is an 'otherwise' that is called if the two tests both fail. However, given the XML file that you can see on the slide, the tests never fail, and so the 'otherwise' never gets called.



WebSphere Education

Elements to generate output (XML to XML)

The following elements can be used to transform from XML-to-XML:

- **<xsl:element>**
 - Creates an element in transformed document
- **<xsl:attribute>**
 - Creates an attribute within an **<xsl:element>**
- **<xsl:copy>**
 - Copies current node and namespace node from source tree to result tree
- **<xsl:copy-of>**
 - Copies current node, namespace node, child tags, and attributes from source tree to result tree
- **<xsl:processing-instruction>**
 - Add a processing instruction node

© Copyright IBM Corporation 2013

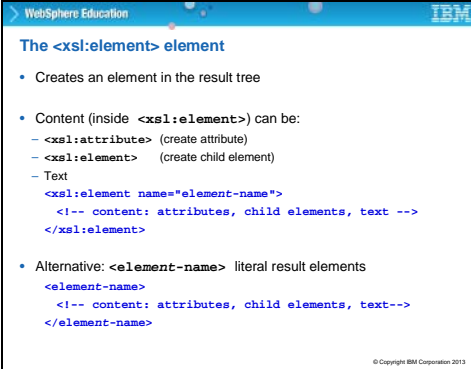
Elements to generate output (XML to XML)

Often the output from an XSL transformation is another XML file. The other file can be the case when the received XML file is not in the correct format, but XML is the required language. These five elements are used to create XML output.

The first two create elements and attributes. For example, given the tag `<book id="1">`, `book` is the element and `id` is its attribute. Elements can have zero or more attributes, but attributes cannot exist without an element. The next two slides give some examples.

The next two create copies of input elements on the output. Both might be used to copy a specific node, but 'copy-of' also copies child elements.

The processing-instruction element was covered in an earlier slide.



WebSphere Education

The `<xsl:element>` element

- Creates an element in the result tree
- Content (inside `<xsl:element>`) can be:
 - `<xsl:attribute>` (create attribute)
 - `<xsl:element>` (create child element)
 - Text
- Alternative: `<element-name>` literal result elements

```
<xsl:element name="element-name">  
  <!-- content: attributes, child elements, text -->  
</xsl:element>
```

```
<element-name>  
  <!-- content: attributes, child elements, text -->  
</element-name>
```

© Copyright IBM Corporation 2013

The `<xsl:element>` element

Here is a bit more information about creating elements. Elements can be nested within other elements; and therefore the content of the element `<xsl:element>` can be other xsl elements. There can also be attributes. Again, you already covered nesting; that attributes can be associated with elements, but cannot exist alone. The third option is to put text inside the xsl element. The example on the slide sums up the concepts.

WebSphere Education IBM

The <xsl:attribute> element

- Creates an attribute in the result tree
- All attributes must precede the first child element

```
<xsl:attribute name="attribute-name">  
  <!-- content: text value -->  
</xsl:attribute>
```

Example:

```
<xsl:attribute name="id">  
  <xsl:value-of select="@no" />  
</xsl:attribute>
```

Create an attribute named "id"

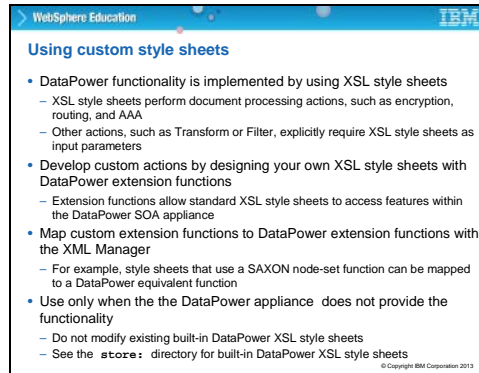
XPath: attribute "no" of current node

The value of the id attribute is the value of attribute "no" of the current node

© Copyright IBM Corporation 2013

The <xsl:attribute> element

An attribute is associated with the element immediately preceding it. There can be no child element before the attribute; otherwise the attribute becomes associated with the child element. The example shows the format of this element. The attribute tag has a name attribute, in this case 'id'. This means that an attribute called 'id' is inserted into the immediately preceding element. The value-of directive is then used to find a value for this attribute. In this case, the value is selected from the number attribute of the current node.



WebSphere Education

Using custom style sheets


- DataPower functionality is implemented by using XSL style sheets
 - XSL style sheets perform document processing actions, such as encryption, routing, and AAA
 - Other actions, such as Transform or Filter, explicitly require XSL style sheets as input parameters
- Develop custom actions by designing your own XSL style sheets with DataPower extension functions
 - Extension functions allow standard XSL style sheets to access features within the DataPower SOA appliance
- Map custom extension functions to DataPower extension functions with the XML Manager
 - For example, style sheets that use a SAXON node-set function can be mapped to a DataPower equivalent function
- Use only when the the DataPower appliance does not provide the functionality
 - Do not modify existing built-in DataPower XSL style sheets
 - See the `store: directory` for built-in DataPower XSL style sheets

© Copyright IBM Corporation 2013

Using custom style sheets


XSLT is the only way DataPower can be programmed. You do not have any programming language as such; and therefore you cannot write applications or functions for DataPower. You can give it what might be called declarative coding and leave it to DataPower to run.

DataPower already has many style sheets that it uses to implement different functions such as routing, or triple A, and for some actions you provide a style sheet as input. You can create your own style sheet for DataPower, and there are also some DataPower style sheet extensions that you can take advantage of. You can modify the style sheets that are provided with DataPower, but you should copy them over to your domain first. Look in the `store: directory` to see what is available.

WebSphere Education 

How to develop style sheets with DataPower extensions

1. Write the XSL style sheet by using an XML editor:
 - For example:
 - Eclipse V3.2 Web Tools
 - IBM Rational Application Developer
2. Compile the style sheet by using the DataPower XSLT compiler
 - Upload and compile your custom style sheet on the DataPower SOA appliance by using the supplied Eclipse plug-ins
3. Add a Transform action to apply the XSL style sheet in the document processing policy



© Copyright IBM Corporation 2013

How to develop style sheets with DataPower extensions

You might develop an XSL style sheet in a simple text editor if you wanted to, but it is usually more convenient to use an XML editor such as the one included with Eclipse. Eclipse has two plug-ins that are targeted to DataPower. One is for XML management and the other is an XSL co-processor, where you can write a style sheet, compile it and send XML to it.

Style sheets are called from within a service policy and are a subject in a further unit. Your policy can have a rule with a transform action, and the transform action can be set to call your custom style sheet.

WebSphere Education

IBM

XSLT variables

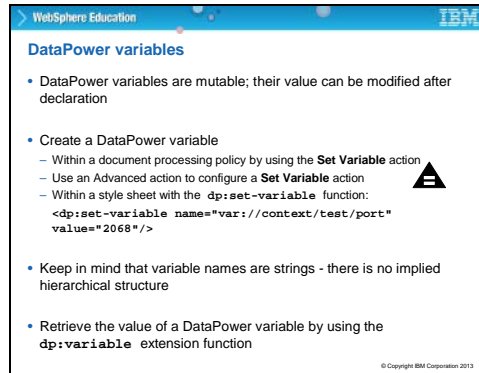
- XSLT variables are *immutable*: their value cannot change once set
- Retrieve the value of a variable by using the *\$variable* notation
 - Use *{ \$variable }* to retrieve a value within an attribute that does not support a nodeset

```
<?xml version="1.0" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
  <xsl:template match="/">
    <xsl:variable name="serviceOp">findByName</xsl:variable>
    The name of the called service is
    <xsl:value-of select="//soap-env:body/local-name(.)" />
    The value of the serviceOp variable is
    <xsl:value-of select="$serviceOp" />
  </xsl:template>
</xsl:stylesheet>
```

© Copyright IBM Corporation 2013

XSLT variables

Within XSLT, there are things that are called variables. Variables are similar to all other programming languages, but there is one significant difference here. When you set a value for an XSLT variable, you cannot change it. It is an immutable value. In a sense, the concept is a contradiction in terms to say that there is a variable that cannot be varied! They are really constants. The slide shows an example. A variable is declared called 'serviceOp', and it is given the value 'findByName'. The variable becomes its immutable value, so when it is used as the selected value in the value-of directive, it returns 'findByName'.



WebSphere Education

DataPower variables

- DataPower variables are mutable; their value can be modified after declaration
- Create a DataPower variable
 - Within a document processing policy by using the **Set Variable** action
 - Use an Advanced action to configure a **Set Variable** action
 - Within a style sheet with the **dp:set-variable** function:

```
<dp:set-variable name="var://context/test/port" value="2068"/>
```
- Keep in mind that variable names are strings - there is no implied hierarchical structure
- Retrieve the value of a DataPower variable by using the **dp:variable** extension function

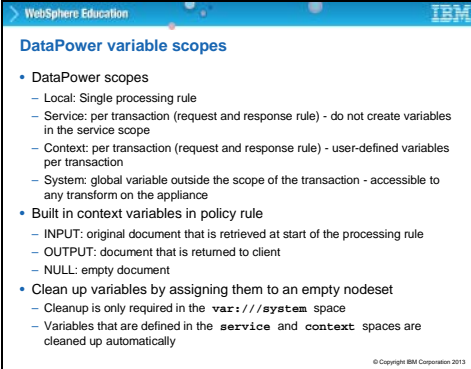
© Copyright IBM Corporation 2013

DataPower variables

DataPower variables are more flexible. Their value is mutable, in other words they really are 'variable'. One way to create a DataPower variable is in the service policy. You can add an action that is called Set Variable to a rule and give it two attributes, a name and a value. This value is associated temporarily with the name; it can be modified at any time. The element is prefixed with dp, not xsl, indicating that the variable is a DataPower variable that is being set.

The name value looks as if it were part of a hierarchical structure, but this is not the case. If you develop another variable and give it the name of, say, "var://context/test/port/new", there is no link between the two variables.

When you have set a variable, you can retrieve the value by calling the variable name by using the 'dp:variable' element.



WebSphere Education

DataPower variable scopes

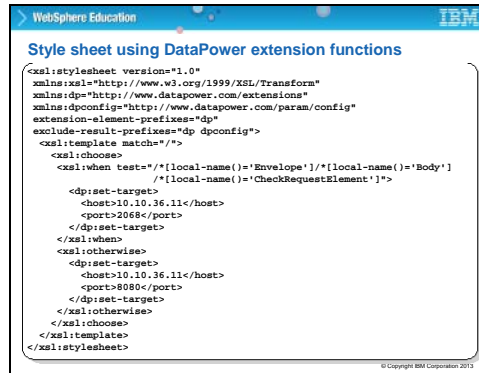
- DataPower scopes
 - Local: Single processing rule
 - Service: per transaction (request and response rule) - do not create variables in the service scope
 - Context: per transaction (request and response rule) - user-defined variables per transaction
 - System: global variable outside the scope of the transaction - accessible to any transform on the appliance
- Built in context variables in policy rule
 - INPUT: original document that is retrieved at start of the processing rule
 - OUTPUT: document that is returned to client
 - NULL: empty document
- Clean up variables by assigning them to an empty nodeset
 - Cleanup is only required in the `var:///system` space
 - Variables that are defined in the `service` and `context` spaces are cleaned up automatically

© Copyright IBM Corporation 2013

DataPower variable scopes

DataPower variables exist in different scopes. There are four all together, starting with the most restricted, local, where the variable exists during a single processing rule. Service and Context both span the life of the transaction. Service is for DataPowers use. You can read these variables, and in some cases write a value to them, but you should not write your own variables in this scope. Context is the transaction scope for user-defined variables. System scope is like global variables, which have a life span beyond any transaction. They are available to any action in any service on the appliance where they are defined.

When you start looking at service policies in a further unit, you find some built-in variables that are used in the rules in the policies. The variables are input, output, and null. The course comes back to the predefined variables later.



Style sheet by using DataPower extension functions

Here is an example of using extension functions in a style sheet. The extension function prefix is defined as dp. The choose element has one when clause and an otherwise. If the test returns true, the dp extension element set-target is called with host and port information. If it returns false, a different port is used.

Slide 38

WebSphere Education

IBM

Testing style sheets with DataPower content

- To test a style sheet that contains any DataPower content, you must run the code on an appliance
 - DataPower variables
 - DataPower extension functions and extension elements
- Typical procedure:
 - Create style sheet in editor or development tool
 - Upload style sheet
 - Create service, policy, rule, transform action to test style sheet
 - Send message/test data to service to test style sheet behavior
 - If more testing needed, upload edited style sheet and test again
- Using the Interoperability Test Service (ITS)
 - Create style sheet in editor or development tool
 - Invoke ITS, passing style sheet and test data
 - If more testing needed, invoke ITS again, passing edited style sheet

© Copyright IBM Corporation 2013

Any style sheet that contains references to DataPower specific functions can be tested only on the appliance.

Slide 39

WebSphere Education

IBM

Interoperability Test Service example

- A development-time service on the appliance for testing transformations
- Testing capabilities:
 - Send schema and XML file, validates XML file
 - Send XPath expression and XML file, returns XPath result
 - Send style sheet and XML file, returns transformation result
- Accepts HTTP, HTTPS, and basic authentication
- Documentation in Information Center tutorials
- Sample code in the Resource Kit
 - Java JAR and Secure Shell clients
 - Sample code for the Information Center examples
- Disabled by default
- For development use only, do **not** enable on production appliances

© Copyright IBM Corporation 2013

Get the DataPower Resource Kit from your DataPower administrator.

Slide 40

WebSphere Education

IBM

Interoperability Test Service example

```
./dp-interop-client.sh -x toBase64.xsl -i message.xml  
-h myDP.com -p 9990
```

- **`/dp-interop-client.sh`** is the sample Secure Shell client
- The style sheet to test is **`toBase64.xsl`**
- The file to transform is **`message.xml`**
- ITS is running on the appliance at **`myDP.com`**
- ITS is listening for HTTP requests on port **`9990`** (default)
- The **response** to the script is the result of the transformation
- For Java implementations, **`java -jar DPInteropClient.jar`** replaces **`./dp-interop-client.sh`**

© Copyright IBM Corporation 2013

This sample code is from the Information Center tutorial.

Slide 41

WebSphere Education

IBM

Unit summary

Having completed this unit, you should be able to:

- * Describe the Extensible Stylesheet Language (XSL) model
- * Construct XPath expressions
- Create XSL style sheets to apply XSL transformations
- Use and apply XSL templates in XSLT
- Describe the use of DataPower variables and extensions in XSL style sheets
- Test a style sheet by using the Interoperability Test Service

© Copyright IBM Corporation 2013

WebSphere Education

IBM

Checkpoint questions

1. What template would you use for extracting a specific value from the source tree?
 - A. `<xsl:choose ... />`
 - B. `<xsl:copy ... />`
 - C. `<xsl:value-of select="..." ... />`
 - D. `<xsl:text />`
2. List the three parts of XSL.
3. What is the difference between XSLT and DataPower variables?

© Copyright IBM Corporation 2013

WebSphere Education


IBM

Checkpoint answers

1. C. What template would you use for extracting a specific value from the source tree?
A. `<xsl:choose ... />`
B. `<xsl:copy ... />`
✓ C. `<xsl:value-of select="..." ... />`
D. `<xsl:text />`
2. The three parts of XSL are:
 - XSL-FO
 - XSLT
 - XPath
3. XSLT variables are immutable; once they are set, their value cannot change. The value of a DataPower variable can change throughout its lifespan.

© Copyright IBM Corporation 2013

Slide 44



WebSphere Education

IBM

Exercise

Creating XSL transformations

© Copyright IBM Corporation 2013
Course materials may not be reproduced in whole or in part without the prior written permission of IBM.

The slide features a blue header with 'WebSphere Education' and the IBM logo. The main content area is white with the title 'Exercise' and the subtitle 'Creating XSL transformations'. A decorative graphic of overlapping circles and dots is on the left. The footer contains copyright information for IBM Corporation 2013.

WebSphere Education

IBM

Exercise objectives

After completing this exercise, you should be able to:

- Create an XSL style sheet
- Create an XML firewall service
- Transform an XML file by using the compiled XSL style sheet
- Describe the use of DataPower variables and extensions in XSL style sheets

© Copyright IBM Corporation 2013

Slide 46

