

Analyze with XSLT, Part 3: Layering XSLT stylesheets

Layer and use chain transformations to create procedural techniques

Skill Level: Intermediate

[Chuck White](mailto:chuck@tumeric.net) (chuck@tumeric.net)
XSLT consultant and Web engineer
Freelance Developer

16 Mar 2004

In this third tutorial in a multi-part series on the benefits of using XSLT, the MindMap Team discovers the need to perform some analyses that require procedural techniques. Because XSLT is not a procedural language, this tutorial explains how to layer and use chain transformations to get around some of the limitations this structure provides. The tutorial demonstrates how to use named templates to mimic side effects, and how external documents are layered together to interoperate.

Section 1. Introduction

Should I take this tutorial?

This is the third in a multi-part series detailing the benefits of using XSL Transformations (XSLT). The first tutorial, [Analyzing non-XML data with XSLT](#), introduced you to MindMap, a fictional research team that uses XSLT as part of its analytical toolbox for interpreting data related to cognitive processes. The goal is to maximize flexibility for their clients by leveraging the ability to change algorithms and methods without having to recompile applications. The MindMap team deals with several aspects of the process, from data acquisition to visualization to tying the

data in to other resources.

You should have a basic understanding of XSLT and XML before you take this tutorial. As in Part 2 of the series, [XSLT as an analysis tool](#), if you aren't familiar with what an XSLT parameter does and how it works, you'll find some of this material a little difficult. This is because this tutorial keys in on the use of the `xsl:param` element as you learn how to pass parameters through various layers within the framework of your XSLT application. In fact, I recommend that you read the first tutorial as well, because it lays the groundwork for the entire series.

If you're looking to discover processes for making XSLT act more like a procedural programming language, this tutorial demonstrates how you can change the value of variables and parameters by calling them through different iterations of the source tree. You'll also discover the power of using multiple documents to layer templates and even XML documents to help achieve these effects.

What is this tutorial about?

Some of the more common navigation systems found on the Web today are tab-based systems. The MindMap team has found a number of different occasions to develop such a system, and so they decided they needed to come up with a flexible solution that would work across a number of different scenarios without too much adaptation. The team wanted to develop a tab-based navigation system using XSLT that would also work both on the client side and on servers. This tutorial examines their solution, which involves XML lookup tables and XSLT, and a layering system involving multiple passes of same-named parameters.

The solution that the MindMap team ultimately decided on for a tabbed navigational system took the same basic form as that presented in this tutorial. The data used might be different depending on the task, but the transformations are pretty much the same. The modular Web application they developed consists of six components, all created in this tutorial:

1. A source document named `CodeLibrary.xml` containing the data that will be displayed
2. A lookup table named `tab_images.xml` for storing information about the tabs
3. A re-usable generic XSLT stylesheet named `tabs.xsl` for generating the tabs
4. A stylesheet named `display.xsl` for displaying the data and for importing `tabs.xsl` for navigation

5. A stylesheet named `addContent.xsl` for storing the additional templates that will be called from the main stylesheet
6. A stylesheet named `stringbreak.xsl` for handling long lines of unbroken text

Tools

Make sure you install and test the following tools before beginning the tutorial:

- [Internet Explorer 5.0 or higher](http://www.microsoft.com/windows/ie/default.asp) (<http://www.microsoft.com/windows/ie/default.asp>).
- You can download the files referred throughout this tutorial in file [x-layerfiles.zip](#). You should download them before starting the tutorial so that you can follow along more easily.

Generally, when passing parameters to and from XSLT stylesheets, you pass to and from a server-side environment, such as a J2EE environment or some other Web application server. To reach the broadest audience, this tutorial uses Internet Explorer as the application server so you can focus on the concept of passing parameters. This means that if you're using a Linux or Unix box, you'll have a hard time running the examples unless you can port it to your application server. This would require some knowledge of how your Web application server manages the Document Object Model (DOM) and is beyond the scope of this tutorial, but the concepts presented here are certainly portable to any server environment.

Section 2. Layering XSLT stylesheets overview

The problem space

When the MindMap Research Team began to develop tab-based navigation systems, they quickly realized that they needed to overcome some perceived limitations in XSLT. For example, how do you get objects on a Web page to respond to events? Passing a parameter in wasn't enough; a response needed to be returned, and the Web page needed to react to that response. This proved to be a more complex task than they expected. Their solution was to create a set of images and set the properties of those images within an XML lookup table. They then needed to create multiple stylesheets that interacted with one another and the

lookup table to achieve the effect they wanted. The lookup table acted as a repository of sorts, which held information about images that, in turn, created the tabs used for the navigation interface.

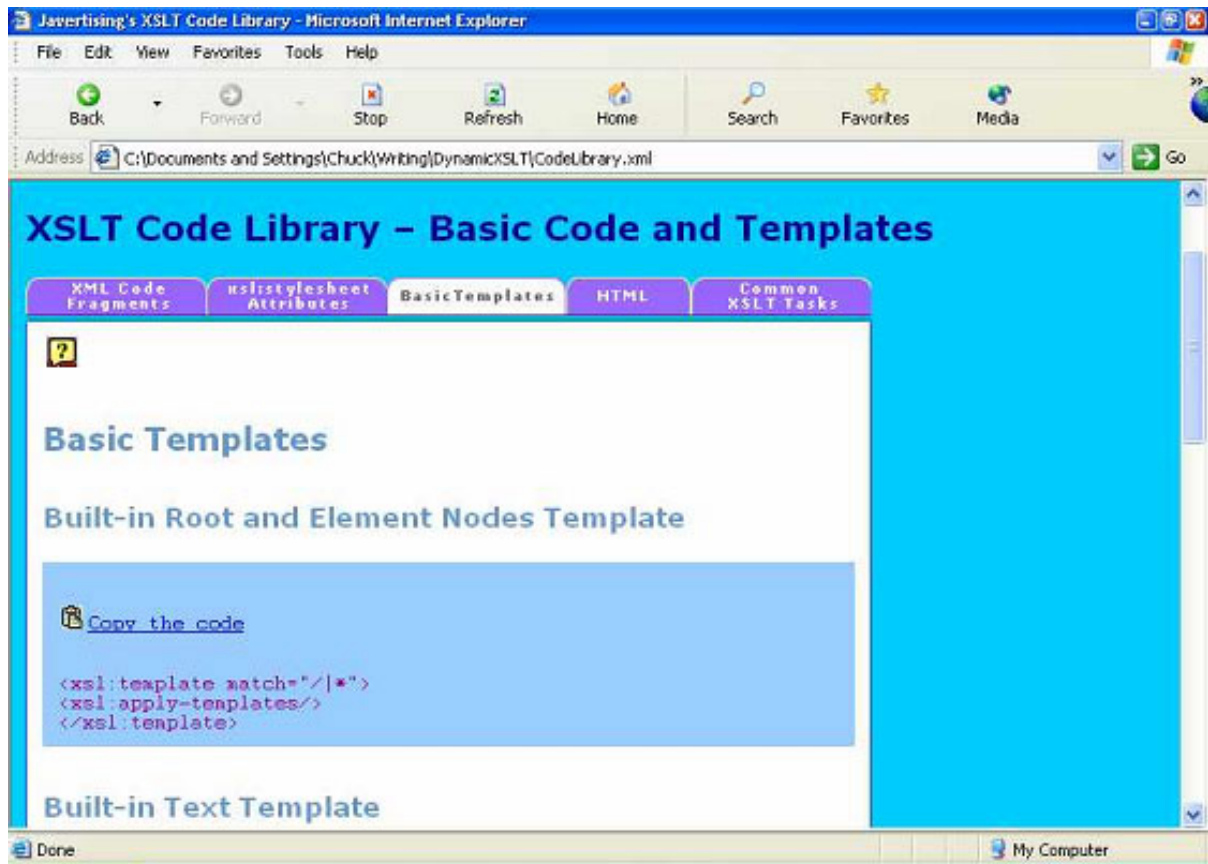
Although they could have created this application using DHTML and CSS, the MindMap team wanted the advantages that XML offers. They wanted to be able to scale the application easily and port it to other content quickly. Storing tab information in an XML-based lookup table and core content in an XML document allowed them to do that.

Exploring the tabbed navigation system

The easiest way to get a handle on how the MindMap tab navigation system works is to take it for a spin. You can do that by loading the file named `CodeLibrary.xml` into your browser. You can find this file in the file archive named `x-layerfiles.zip` (see [Tools](#)). The code only works in Internet Explorer for Windows.

When the document loads into the browser, you should see a screen like that shown below. The application is a library of XSLT code snippets that you can copy and paste into your XSLT code. The tabs help you navigate to different types of code snippets without moving to a different Web page. In this case, the application is a combination of Dynamic HTML and Dynamic XSLT, but it can be ported into a server environment to remove the client-side scripting aspect of the application.

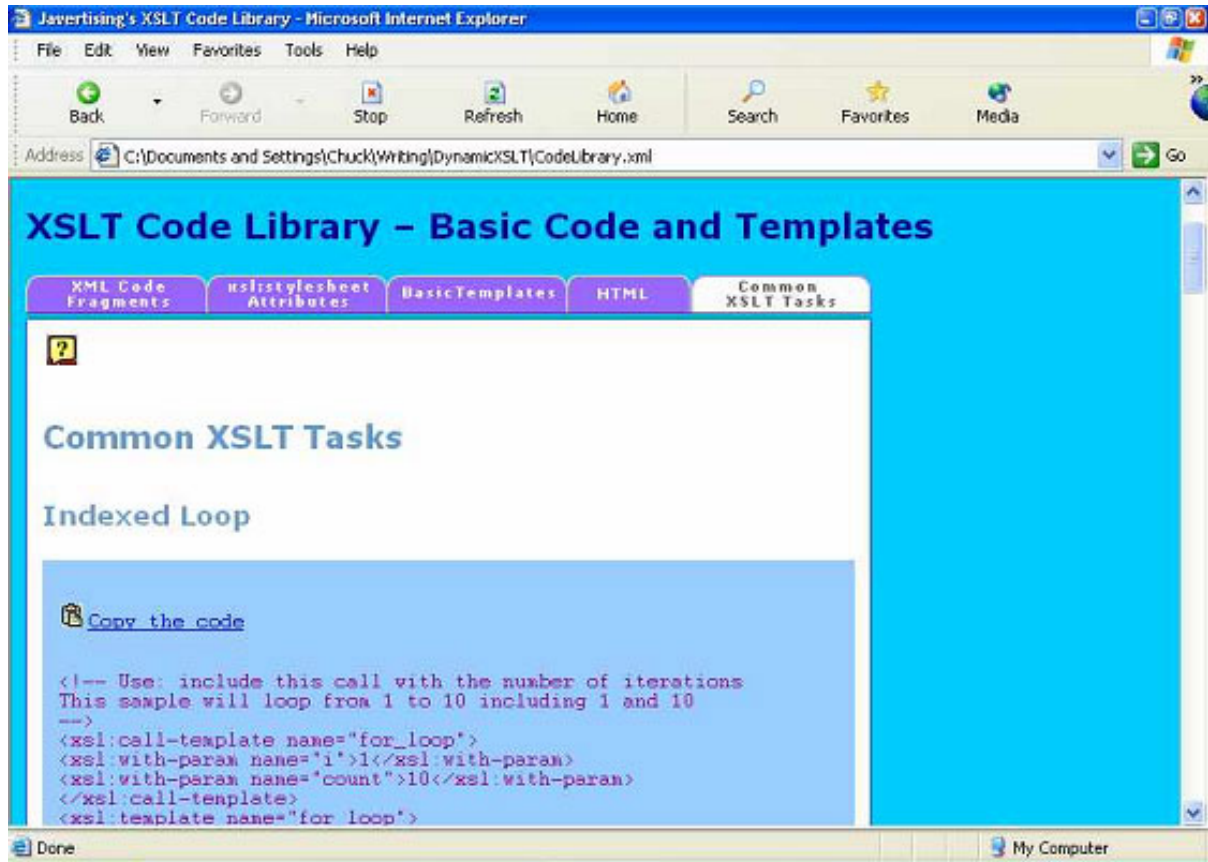
Figure 1. Example of tabbed navigation system



Clicking different tabs reveals different data

When you load the file into your browser, you can click on different tabs to reveal different panels, as shown in the image below.

Figure 2. Example of how to change tabs in the navigation system



Note how the *on* tab, which has a white background, reveals a unique set of data. You can adapt this kind of application to your own needs. The main thing to remember is that the tabs need to help users track where they are. In this case, when a user is on a topic, the relevant tab is white. The others are greyed out -- or in this case, blued out. When a user clicks one of the blue tabs, it becomes white, and new content is displayed.

The XML source file

The XML source file is an XML document named `CodeLibrary.xml`. It is derived from a word processing document. You can use WordPerfect, Microsoft Word, or OpenOffice.org's open source office suite to create XML documents from word processing documents. The structure of `CodeLibrary.xml` looks like what you see below. You can extract the file in its entirety from the archived file named `x-layerfiles.zip`.

```
<Word-Document>
  <ChapterTitle>
    <p>Appendix F </p>
    <p>XSLT Code Library -- Basic Code and Templates</p>
  </ChapterTitle>
```

```

<Para>
  <p>These templates are generic in every sense of the word. I
  certainly can't claim their invention. They're pretty standard templates
  developed over the last couple of years as part of the best practices
  coming out of the XSLT development efforts of many people.</p>
  <p>I haven't provided an explanation on how to use many of these
  code fragments, but if you understand the concepts shown throughout this
  site, you should have no problem with any of them. Some of them will
  require more adaptation than others. A few of them will require very
  little or none at all. Have fun!</p>
</Para>
<Title1>
  <p>XML Code Fragments</p>
</Title1>
<Title2>
  <p>XSLT Processing Instruction</p>
</Title2>
<CodeSnippet>
  <p>&lt;?xml-stylesheet type="text/xsl" href="" ?></p>
</CodeSnippet>
<Title2>
  <p>DOCTYPE SYSTEM</p>
</Title2>
<CodeSnippet>
  <p>&lt;!DOCTYPE name SYSTEM "" ></p>
</CodeSnippet>
<Title2>
  <p>DOCTYPE PUBLIC</p>
</Title2>
<CodeSnippet>
  <p>&lt;!DOCTYPE name PUBLIC "" "http://" ></p>
</CodeSnippet>
<Title2>
  <p>DOCTYPE Internal Subset</p>
</Title2>
<CodeSnippet>
  <p>&lt;!DOCTYPE name [ ]></p>
</CodeSnippet>
<Title2>
  <p>CDATA Section</p>
</Title2>
<CodeSnippet>
  <p>&lt;![CDATA[ 'content ' ]]></p>
</CodeSnippet>
<Title1>
  <p>xsl:stylesheet Attributes</p>
</Title1>
<Para/>
...
</Word-Document>

```

Revisiting XSLT parameters

The real guts of the tabbed navigation system reside in the XSLT stylesheets and careful maneuvering of parameters using the `xsl:param` element.

When you're thinking of Dynamic XSLT, whether client-side or server-side, parameters play an even bigger role than events, which lie at the heart of object-oriented programming on the Web.

Events are still important, of course, because they obviously trigger changes in the

display. In this case, a mouse click event triggers the calling of a template whose name is equal to the name of the tab being clicked. This is where parameters come in. The MindMap team created an XML lookup table, which can be accessed by an XSLT document using the `document()` function. By tracking the position of the nodes in relation to the user's position on the Web page, you can pass an associated parameter that displays the data you want. You'll revisit these concepts shortly. If you read the previous tutorial on managing parameter passing (see [Resources](#) for a link to Part 2 "Layering XSLT stylesheets"), keep those concepts in mind as you examine the initial pieces of this application. If you didn't read this previous tutorial, keep in mind that you'll make extensive use of parameters among the different layers of this application to glue all the pieces together.

Section 3. Transforming the XML Source

Building the content display XSLT

I won't go too deeply into how to build the basic display stylesheet, because you can use many different varieties of content-based stylesheets. The XML source file was originally derived from a word processing document that was converted to XML based on the stylesheets used in the word processing document. In other words, for every paragraph formatted with the `Para` stylesheet, a `Para` element was made, along with a child `p` element, and the content of that paragraph became the `p` element's content. The same process was used for other stylesheets in the word processing document. The hierarchy of the document was structured so that the `Title1` word processing style was the top level headline, the `Title2` style was a second level headline, and so on.

Since the MindMap team mapped every word processing stylesheet to an element with a child `p` element, it was relatively easy to create some reasonably global formatting characteristics for the `p` element.

From that point, the key was to build a series of templates that matched the major categories of word processing stylesheets, such as `Title1`, `Title2`, and so on.

The template named `content` lives in the `display.xsl` stylesheet, which can also be downloaded from the archived file named `x-layerfiles.zip`. It is the simplest part of the Web application:

```
<xsl:template name="content">
  <xsl:apply-templates select="Title1[1]" />
</xsl:template>
```

This selects the first major section of the XML source document, which includes all the nodes that follow the first `Title1` element up to the point a second `Title1` element is encountered. This is because the template matching `Title1` includes processing instructions that process its siblings. The name chosen for this template, `content`, is really an arbitrary name. It's important to note that the `addContent.xsl` stylesheet includes another template that selects exactly the same nodes. The MindMap team did this so that they could change the name of the called template easily and change it back to `content` every time the processor reloads.

The addContent.xsl file

The `addContent.xsl` file simply processes each heading level in the document. If you look at the source document, you can see that each major content node is preceded by -- but not a child of -- one of the `Titlex` elements. By creating a template for each of these `Titlex` elements, you can process the entire document. Here, then, is the `addContent.xsl` file in its entirety:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template name="xmlCodeFragments">
    <xsl:apply-templates select="Title1[1]" />
  </xsl:template>
  <xsl:template name="xslStyleSheetAtts">
    <xsl:apply-templates select="Title1[2]" />
  </xsl:template>
  <xsl:template name="BasicTemplates">
    <xsl:apply-templates select="Title1[3]" />
  </xsl:template>
  <xsl:template name="HTML">
    <xsl:apply-templates select="Title1[4]" />
  </xsl:template>
  <xsl:template name="CommonXSLT">
    <xsl:apply-templates select="Title1[5]" />
  </xsl:template>
</xsl:stylesheet>
```

Each template is based on a section name and a tab with the same name that reveals that section to the user. Each section is mapped to consecutive instances of the `Title1` element. There are five sections in total, so it's easy to simply create a named template for each one and process the appropriate node from within the named template.

Section 4. Building a chain of named parameterized templates

Building a lookup table for tabs, part 1

The first step in building a chain of parameterized templates is to build an XML lookup table that consists of information about your tabs. The XML document consists of a series of `image` elements that contain information about each: the name of each tab image, the width, the height, and a description. You could just as easily use text instead of images.

```
<?xml version="1.0"?>
<tab_images>
  <image>
    <name>xmlCodeFragments</name>
    <width>120</width>
    <height>25</height>
    <title>XML Code Fragments</title>
  </image>
  <image>
    <name>xslStyleSheetAtts</name>
    <width>120</width>
    <height>25</height>
    <title>xsl:stylesheet Attributes</title>
  </image>
  <image>
    <name>BasicTemplates</name>
    <width>120</width>
    <height>25</height>
    <title>Basic Templates</title>
  </image>
  <image>
    <name>HTML</name>
    <width>83</width>
    <height>25</height>
    <title>HTML</title>
  </image>
  <image>
    <name>CommonXSLT</name>
    <width>120</width>
    <height>25</height>
    <title>Common XSLT Tasks</title>
  </image>
</tab_images>
```

Building a lookup table for tabs, part 2

The lookup table is used by the XSLT stylesheet, technically not as a source document, but as a lookup table for building the images and choosing which template to call. XSLT's built-in `document()` function provides access to all the

nodes of an external XML document and is used in your example at the top level of the `tabs.xsl` stylesheet:

```
<xsl:variable name="tabImages"
select="document('tab_images.xml')/tab_images" />
```

When other developers want to use the generic `tabs.xsl` template, they can add image information to the lookup table. To enforce data integrity it might not be a bad idea to build a simple DTD or schema for the lookup table, but you're bypassing that step for the purposes of this article.

The lookup table, then, has two purposes:

- To build a table of images and any relevant information, such as the height and width for each image.
- To provide a name for each template that's associated with each image through the shared `name` element (remember your function call argument and the attribute value template that retrieves this element value).

Lookup tables that are based on XML source code can be extremely helpful devices in the layering process. One caveat is that on extremely high traffic sites, files using the `document()` function tend to slow things down a bit. But they can be very useful on light-to-medium traffic sites.

Building the `tabs.xsl` stylesheet

The `tabs.xsl` document includes an `img` literal result element that looks essentially like this:

```

```

If you look at `tabs.xsl`, you'll see that the image data comes directly from the elements in the lookup table. The `src` attribute in the code above uses attribute value templates (AVTs), which are XSLT expressions delimited by curly braces (`{ }`). AVTs are used as values in literal result attributes whenever you need to extract node values from your source document or, in this case, some other external XML document, and plunk them into attribute values. They can be any valid XPath expression, and you can use them more than once within the scope of a single attribute value.

In this case, the `src` attribute uses the `name` element from the lookup table to help create an image name that will ultimately look something like this:

```
tab_xmlCodeFragments_on.gif
```

It does this by concatenating string values: `tab_` + the element name + either the string "on" or the string "off" + `.gif`.

Creating an on/off switch for the tabs

Because you know that you want the tab associated with the content currently being examined by the user to look different from all the other tabs, you can create an on/off switch and name the images accordingly. That's where the next attribute value template comes in, which uses a `y` variable to store the value of the on/off switch. When a tab is clicked, it should change so that the user knows he or she is looking at the associated content. So you append "on" to the `src` attribute's value. Otherwise, you append the string "off" to the `src` attribute's value, like this:

```
tab_xmlCodeFragments_off.gif
```

Even though variables in XSLT are static, you can make them behave as if they aren't. The `y` variable is constructed with a series of conditional statements. Since those conditions may change whenever the stylesheet is reloaded, you get behavior that will remind you of good, old-fashioned, object-oriented variables. Take a look at the statements that compose the `y` variable:

```
<xsl:variable name="y">
  <xsl:choose>
    <xsl:when test="$label = $on">on</xsl:when>
    <xsl:otherwise>off</xsl:otherwise>
  </xsl:choose>
</xsl:variable>
```

`$label`, of course, comes from yet another source. It's actually not a variable, but a parameter, as is `$on`, which stores the position of the current node:

```
<xsl:param name="on" select="position()" />
```

`$label` is a parameter that gets defined earlier in the stylesheet. It stores a number based on the name of the template that is being called. For example, if the template is named `xmlCodeFragments`, the number value is 1. If the template is named `xslStyleSheetAtts`, it stores the number 2, and so on. It does this through a system of parameter passing using yet another parameter named `$discover`. It is the `$discover` parameter that actually stores the numbers associated with the called template, and you'll be revisiting it soon to see how it works.

The tabs.xsl stylesheet

Now that you've seen the basics of how `tabs.xsl` works, here it is in its entirety:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.1"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"
  indent="yes"
  doctype-public="-//W3C//DTD HTML 3.2 Final//EN"/>
<xsl:variable name="tabImages"
  select="document('tab_images.xml')/tab_images" />

<xsl:template name="makeOnTabs">
<xsl:param name="imgNumber" />
  <table border="0" cellpadding="0" cellspacing="0">
    <tr>
<xsl:apply-templates select="$tabImages/image">
  <xsl:with-param name="discover" select="$imgNumber" />
</xsl:apply-templates>
    </tr>
  </table>
</xsl:template>

<xsl:template match="image">
<xsl:param name="discover" />
  <xsl:call-template name="tab">
    <xsl:with-param name="label" select="$discover" />
  </xsl:call-template>
</xsl:template>

  <xsl:template name="tab">
    <xsl:param name="label" />
    <xsl:param name="on" select="position()" />
    <xsl:variable name="y">
      <xsl:choose>
        <xsl:when test="$label = $on">on</xsl:when>
        <xsl:otherwise>off</xsl:otherwise>
      </xsl:choose>
    </xsl:variable>

    <td align="left" height="16">
      
    </td>

  </xsl:template>

  <xsl:template name="js">
<script language="JScript">
  function chgNode(xpath, xValue) {
    xmlDoc = document.XMLDocument;
    xslDoc = document.XSLDocument;
    var matchString = xslDoc.selectSingleNode(xpath);
    matchString.text = xValue;

    output.innerHTML = xmlDoc.documentElement.transformNode(xslDoc);

    // This resets the call-template name to "content"
    // Otherwise, the preceding change
    // kills it, and the script looks for it,
    // so leaving it off results in an error
  }
</script>
</xsl:template>

```

```
    matchString.text = "content";
}
    </script>
</xsl:template>
```

Notice that `tabs.xsl` includes a variable that refers to the lookup table containing the images used by the application to represent the tabs:

```
<xsl:variable name="tabImages"
select="document('tab_images.xml')/tab_images" />
```

You can then refer to any node in that document, which essentially acts as a second source document.

XSLT may seem complex when you're first learning it, but it is a very powerful language that allows for a great deal of programming creativity. In this case, since you know that you will change the name of the template dynamically through the `selectSingleNode` method, you know you can assign different numerical values to a parameter based on the template name and compare it to the position of a current node to determine whether a label is on or off. The code within the `script` element is necessary if you're using the file in Internet Explorer. If you use server-side processing, you'll pass the parameters and template calls to the server using the server-side language of your Web application environment.

Tracking the parameters

You'll now track the parameter that manages the tabbing process. To do this, you need to look at the file named `display.xsl`. In that stylesheet, the `template makeOnTabs` (from the `tabs.xsl` document) is called with the `imgNumber` parameter that was declared in the `makeOnTabs` template.

This is where you can get kind of silly with parameters and make them act almost like the kind of variables that object-oriented programmers are used to playing with. The MindMap team wanted to be able to change the template names dynamically, so when the stylesheet loaded, the name of the template being called would initially be `content`. The team assigned a number using `xsl:number` to that test case. They knew they only wanted to display the data associated with the first `Title1` node in the source document, so they assigned the number 1 to this test. Since the second `Title1` element in the source document was the heading for the topic `xsl:stylesheet Attributes`, the team assigned the number 2 to the test case for the template named `xslStyleSheetAtts`. They kept doing this for each `Title1` node:

```
<xsl:call-template name="makeOnTabs">
```

```

<xsl:with-param name="imgNumber">
  <xsl:choose>
    <xsl:when test="$doc//*[xsl:call-template/@name = 'xmlCodeFragments']">
      <xsl:number value="1" /></xsl:when>
    <xsl:when test="$doc//*[xsl:call-template/@name = 'xslStyleSheetAtts']">
      <xsl:number value="2" /></xsl:when>
    <xsl:when test="$doc//*[xsl:call-template/@name = 'BasicTemplates']">
      <xsl:number value="3" /></xsl:when>
    <xsl:when test="$doc//*[xsl:call-template/@name = 'HTML']">
      <xsl:number value="4" /></xsl:when>
    <xsl:when test="$doc//*[xsl:call-template/@name = 'CommonXSLT']">
      <xsl:number value="5" /></xsl:when>
    <xsl:when test="$doc//*[xsl:call-template/@name = 'content']">
      <xsl:number value="1" /></xsl:when>
    <xsl:otherwise><xsl:number value="1" /></xsl:otherwise>
  </xsl:choose>
</xsl:with-param>
</xsl:call-template>

```

Discovering the images

If you next look at the `tabs.xsl` stylesheet, you'll see a template for the `image` element. Remember that since the MindMap team is accessing an external document, they can access all the nodes in that document and create templates for any of them. The `image` template contains an undefined parameter named `discovery`, whose purpose is to do what it says it will do: *discover* a value. In this case, the parameter values that I just described are:

```

<xsl:template match="image">
  <xsl:param name="discovery" />
  <xsl:call-template name="tab">
    <xsl:with-param name="label" select="$discovery" />
  </xsl:call-template>
</xsl:template>

```

This is done by applying the template in the `makeOnTabs` named template in `tabs.xsl` by giving the `discovery` parameter the value of the incoming `imgNumber` parameter; this was done in `display.xsl` using the call to the `makeOnTabs` template, and will thus be one of five numbers.

You'll see some recursive action at play here, and if you aren't familiar with recursion in XSLT you might want to look at the [Resources](#) at the end of this tutorial to learn a bit more about it.

Turning tabs on and off

Next, take a look at the `tab` template in `tabs.xsl`. This is the switch for turning the tab on or off. If a tab is on it has a white background, and if it is off it has a blue background. This is just basic navigation stuff -- people like to know where they are:

```

<xsl:template name="tab">
  <xsl:param name="label" />
  <xsl:param name="on" select="position()" />
  <xsl:variable name="y">
    <xsl:choose>
      <xsl:when test="$label = $on">on</xsl:when>
      <xsl:otherwise>off</xsl:otherwise>
    </xsl:choose>
  </xsl:variable>

  <td align="left" height="16">
    
  </td>
</xsl:template>

```

The final template in `tabs.xsl` is the template named `js`, which simply stores the JavaScript that processes everything client-side.

Section 5. Managing JavaScript

Calling JavaScript templates

One common practice among many XSLT developers, especially in high-end production environments that require JavaScript, is to build JavaScript functions in separate files and call them in templates. This application creates client-side JavaScript in the `tabs.xsl` stylesheet in a template named `js` and calls it from `display.xsl`. The `js` template should look familiar if you read the last tutorial:

```

<xsl:template name="js">
  <script language="JScript">
    function chngNode(xpath, xValue) {
      xmlDoc = document.XMLDocument;
      xslDoc = document.XSLDocument;
      var matchString = xslDoc.selectSingleNode(xpath);
      matchString.text = xValue;

      output.innerHTML =
        xmlDoc.documentElement.transformNode(xslDoc);

      // This resets the call-template name to "content"
      // Otherwise, the preceding change
      // kills it, and the script looks for it,
      // so leaving it off results in an error
      matchString.text = "content";
    }
  </script>

```

The first part of the code just initializes the XSLT processor and XML parser so that Internet Explorer can work with the application. The `matchString` variable (`var matchString`) then stores the value of a single node using a Microsoft-specific extension to the DOM named `selectSingleNode`, which does what its name implies and selects a single node using an XPath statement, highlighted in bold below:

```
<td align="left" height="16">
  
</td>
```

As you can see, to get JavaScript and XSLT to play nicely together you have to do some fancy footwork with your delimiters within your `onclick` event.

Changing the name of a template dynamically

The `chnNode()` function changes the name of the attribute in the following template, dynamically obtaining the name of the image in the `tab_images.xml` lookup table and applying that name to the `name` attribute of `xsl:call-template`. The processor finds the template in `addContent.xml`. So the name of the template changes from `content` to, for example, `xmlCodeFragments`.

When the user clicks on a tab, the XSLT processor in Internet Explorer reloads, which gives you an opportunity to dynamically change the name of the template. You have to change the name back to `content` in your JavaScript every time it reloads, however, to avoid an error because by default the main stylesheet looks for a template named `content`, which is what the application loads with (this is found in `display.xml`):

```
<xsl:template name="content">
  <xsl:apply-templates select="Title1[1]" />
</xsl:template>
```

If you look back at `addContent.xml`, you'll see where the templates associated with each image are defined. For example:

```
<xsl:template name="xmlCodeFragments">
  <xsl:apply-templates select="Title1[1]" />
</xsl:template>
```

This does the same basic thing as the `content` template, but when you choose the other image names they process different node sets. For example, the `xslStyleSheetAtts` template processes the second `Title1` element and all of

its children:

```
<xsl:template name="xslStyleSheetAtts">
  <xsl:apply-templates select="Title1[2]" />
</xsl:template>
```

Porting this application to Mozilla would require you to find a DOM method that allows you to mimic `selectSingleNode()`.

Calling the JavaScript

Calling the JavaScript itself is easy. In the `display.xml` file you insert an `xsl:call-template` element into the `head` HTML literal result element:

```
<head>
  <link rel="stylesheet" href="global.css" type="text/css" />
  <xsl:call-template name="js" />
  ...
</head>
```

It's usually a good idea to wrap JavaScript in a `CDATA` section. But I've decided to skip that here, since the JavaScript itself was short enough and contained no `>` or `<` operators or other XML character references that needed escaping.

Section 6. Putting it all together

Beginning the `display.xml` file

The Mindmap team's final task was to put everything together. The file that accomplishes this, `display.xml`, is too long to display in this tutorial, but you can download it from the `x-layerfiles.zip` archive available in [Tools](#). I'll show you the highlights of the code in this section.

The `display.xml` file creates the templates that are processed by `addContent.xml` and stores some JavaScript so that the application can work in a browser. `display.xml` also contains a call to a generic template named `stringbreak.xml` that manages the column width of text strings so that long URLs don't force the page to be unreasonably wide in a way that would make readers scroll their browser window to the right.

Importing and including documents

The first step to completing the application is to bring in any external XSLT files you may need. Note the use of the `import` and `include` statements. Generally in production environments a build sequence is structured in a way that includes are physically brought into a master file, which in this case is `display.xsl`. Import statements, however, remain references. Using includes in these kinds of build environments can be a handy way to troubleshoot problems because you can see all the relevant templates in one place after the build is done. For your purposes, since everything is being done client-side, some precedence rules are the main difference.

```
<xsl:import href="stringbreak.xsl"/>
<xsl:include href="addContent.xsl" />
<xsl:include href="tabs.xsl"/>
```

Then of course you need to bring in the lookup table that holds the images. You can do that using the `document()` function:

```
<xsl:variable name="doc" select="document('')"/>
```

Building the core stylesheet

The fact that you use a browser to pass parameters, instead of using server-side code, requires that you use the following lines of code:

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

Next, you add the template for the root element of the source document. Note the `call-template` instruction, which calls a JavaScript file that's included in the `tabs.xsl` document. This is simply a JavaScript file that reloads the Internet Explorer XSLT processor each time a tab is clicked. Its functionality is similar to that found in the previous tutorial, "XSLT as an analysis tool" (see [Resources](#)), which demonstrated how to create an XSLT-based search engine.

```
<xsl:template match="Word-Document">
<html>
  <head>
    <link rel="stylesheet" href="global.css" type="text/css" />
    <xsl:call-template name="js" />
  </head>
  <script language="jscript">
    // select and copy text functions go here
  </script>
</html>
```

```
// these are included in downloadable file
</script>
```

Turning tabs on and off through template calls

This template is quite long, but most of it is simply a series of instructions that help form the layout of the application. However, it includes one very key ingredient that allows it to manage the tabs:

```
<div id="navbar" align="left" style="border-bottom: #9966ff 1px outset;
width:{sum($tabImages//image/width)}; height:{$tabImages/image/height}">
  <xsl:call-template name="makeOnTabs">
<!-- You've already seen this part -->
  </xsl:call-template>
</div>
```

The `makeOnTabs` template is called with its parameter, `imgNumber`, which you saw a bit earlier in this tutorial in [Tracking the parameters](#). Note also that the image width and height are established using attribute value templates (values within braces like this: `{ }`). These are obtained from the lookup table that defines the images (described in [Building a lookup table for tabs, part 1](#)).

Of course, `display.xsl` offers a good deal more than this, but as I mentioned, most of it relates to the display of the content. Some additional JavaScript lets the user copy the code snippets without highlighting the text within the browser. Since the first two tutorials spent some time pushing the notion of generic templates, I'll now show you one of these generic templates in action: The `stringbreak.xsl` template is used in `display.xsl` to prevent URLs from pushing the width of the window so far to the left that users need to scroll horizontally.

Section 7. Exploring the `stringbreak.xsl` generic stylesheet

Breaking up isn't so hard to do

One problem the MindMap team noticed when developing the main content for their application was that long snippets of code in the reference forced the page to extend wide so the user must scroll horizontally to see all the text. They decided to use a template that broke long strings of letters without white space at 60 characters. This meant that if a code snippet or URL contained more than 60 characters it would

break in the page at the 61st character.

The key to using this template, like most generic templates, lies in knowing how to work with the parameters the template uses. In this case, you pass a string to the template. That string is always the code snippet that is processed from the XML source document.

You can start off with just the knowledge that the `stringbreak.xml` template will accomplish what you wish without even knowing about how it works. That's the beauty of generic templates. Even if you're an accomplished XSLT developer, sometimes it's nice to know boilerplate code is available for you to use without a lot of study. In this case, you know that the `stringbreak.xml` template breaks up strings of text, and you know (or are about to, that is) that the template breaks the text up at the 61st character.

Knowing when to break up

The MindMap team didn't want to assume the convenience of a specific element that helps choose when to look for text to break up. They also knew that they wanted the template to be reusable -- in other words, it shouldn't have any dependencies on any source XML nodes whatsoever. To make this template work, all you need to know is that if you use the template you need to call the `pageWidth` template with two parameters: `String` and `paramWidth`. The first parameter, `String`, simply passes any string you name into the template call. The MindMap team knew they didn't want to process every text node through this template, so they checked for the `http` protocol string:

```
<xsl:for-each select="p">
  <xsl:choose>
    <xsl:when test="contains(., 'http://')">
      <xsl:call-template name="pageWidth">
        <xsl:with-param name="String" select="normalize-space(.)" />
        <xsl:with-param name="pageWidth">60</xsl:with-param>
      </xsl:call-template>
      <br />
    </xsl:when>
    <xsl:otherwise>
      <xsl:value-of select="." /><br />
    </xsl:otherwise>
  </xsl:choose>
</xsl:for-each>
```

This is the basic text processing template for the `CodeSnippet` element, which the MindMap team knows contains the only elements containing strings of 60 or more unbroken characters. In this case, the offending string is the last one in the document, which is a fictional URL used as a reference. If the XML node containing the URL had been different (the `p` element, for example), they would have simply processed that node instead. To see how the URL would affect the behavior of the application, remove the parameter call to `pageWidth` within the `for-each`

statement and simply do the following:

```
<xsl:for-each select="p">
  <xsl:choose>
    <xsl:when test="contains(., 'http://')">
      <xsl:call-template name="pageWidth">
        <xsl:with-param name="String" select="normalize-space(.)" />
        <xsl:with-param name="pageWidth">60</xsl:with-param>
      </xsl:call-template>
      <br />
    </xsl:when>
    <xsl:otherwise>
      <xsl:value-of select="." /><br />
    </xsl:otherwise>
  </xsl:choose>
</xsl:for-each>
```

The last node in the document, which contains a long URL, pushes the width of the display column wider. In this case, it doesn't make it necessary to scroll horizontally, but the potential is there.

Understanding the stringbreak.xsl stylesheet

Now, take a look at the template that `display.xsl` calls and that lives in the `stringbreak.xsl` stylesheet. You can see that the template named `pageWidth` contains two parameter definitions that are used by the `call-template` statement in [Knowing when to break up](#). No values are assigned because they'll be assigned by the `with-parameter` statements used in the calling template. The variable `x` is used to store chunks of the incoming string. As you can see by the `choose` statement highlighted in bold, the variable uses XPath string functions to first test the length of the incoming string. If it's less than the count named in the `pageWidth` parameter (in this case, 60, as defined in the calling template), then only the substring consisting of characters up to the 60th character is returned:

```
<xsl:value-of select="substring($String, 1, $pageWidth)" />
```

The entire `pageWidth` template looks as follows. Pay special attention to the other `xsl:when` element, which calls another template (through the variable `x`, which stores the processed results) to manage the termination condition:

```
<xsl:template name="pageWidth">
  <xsl:param name="String" />
  <xsl:param name="pageWidth" />
  <xsl:choose>
    <xsl:when test="string-length($String) <= $pageWidth">
      <xsl:value-of select="substring($String, 1, $pageWidth)" />
    </xsl:when>
    <xsl:when test="$String">
      <xsl:variable name="x">
        <xsl:call-template name="spaceHandler">
          <xsl:with-param select="$String" name="String" />
          <xsl:with-param select="$pageWidth" name="pageWidth" />
        </xsl:call-template>
      </xsl:variable>
    </xsl:when>
  </xsl:choose>
```

```

</xsl:variable>

<xsl:value-of select="concat(substring($String, 1, $x), ' ')" />
<xsl:call-template name="pageWidth">
  <xsl:with-param select="substring($String,$x + 1)" name="String" />
  <xsl:with-param select="$pageWidth" name="pageWidth" />
</xsl:call-template>

</xsl:when>
</xsl:choose>
</xsl:template>

```

Establishing a termination condition

Remember that when you have a template call itself you need a termination statement of some kind, which in this case is managed by a call to another template called `spaceHandler`. This template also manages where in the line break it is acceptable to break lines, since you don't want to break them in the middle of a code statement:

```

<xsl:template name="spaceHandler">
  <xsl:param name="String" />
  <xsl:param name="pageWidth" />
  <xsl:param name="widthValue" />

  <xsl:choose>
    <xsl:when test="$pageWidth = 0">
      <xsl:value-of select="$widthValue" />
    </xsl:when>
    <xsl:otherwise>
      <xsl:choose>
        <xsl:when test="substring($String, $pageWidth, 1 )
          = '/' or substring($String, $pageWidth, 1 )
          = ' ' or substring($String, $pageWidth, 1 ) = '_' ">
          <xsl:value-of select="$pageWidth" />
        </xsl:when>

        <xsl:otherwise>
          <xsl:call-template name="spaceHandler">
            <xsl:with-param select="$String" name="String" />
            <xsl:with-param select="$pageWidth - 1" name="pageWidth" />
            <xsl:with-param select="$widthValue" name="widthValue" />
          </xsl:call-template>
        </xsl:otherwise>
      </xsl:choose>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>

```

The key to the template is the termination statement, which performs the kinds of cascading parameter techniques you explored in the first two tutorials in this series.

Section 8. Summary

XSLT: Limited only by your imagination

The `display.xml` stylesheet is the glue that manages the entire process of accepting and returning parameter values. However, its success is directly dependent on a series of other files that have dependencies on one another. So far, the first three tutorials in this series have focused on a key concept central to advanced XSLT processing -- something the MindMap team refers to as **cascading parameters**. You were introduced to this concept in the first two tutorials, but in this one the application is more robust and ideally demonstrates that their use is limited only by your imagination.

Overall, the concept of layering multiple documents demonstrates that you can create very dynamic applications that strongly mimic the functionality of procedural programming. This can result in everything from calculators to chess games, as well as comprehensive Web applications that involve high-level processing. This doesn't mean that XSLT is a replacement for back-end and application server logic that involves databases and the business logic used by such environments as J2EE to work with them. In fact, it's always better to do as much logic as you can in the back end before processing XSLT on the presentation layer. But often, developers make the mistake of assuming XSLT can't do something, so they turn to JavaScript, which can result in unwieldy spaghetti code that takes days to reengineer when problems surface. Sometimes in XSLT, the answers to some simple logic questions are looking right at you. You just need to know where to turn, and the place to turn is very nearly always parameters and named templates.

Layering XSLT stylesheets summary

The MindMap team's interactive journey with XSLT began, as seen in the [first tutorial](#) in this series, with the team's earliest exposure to XSLT a few years ago. Then, the team, like most developers learning about a language, gingerly stepped forward, beginning with some basic string manipulation patterns and moving on to template manipulation. This led to the world of interactive forms in the [second tutorial](#), which allowed them to create dynamic applications designed for the analysis and review of information.

Now, in this tutorial, they've taken the concept of cascading parameters seen in the second tutorial a step further, by creating an application that relies heavily on passing parameter values to such an extent that it actually seems as though the parameter values aren't static. If you look closely at `display.xml`, you'll also see how CSS styles are managed by variables that also seemingly change values. Of course, because XSLT has only static values for variables and parameters, this is all an illusion. A magic act of sorts.

In the next tutorial in the series, the MindMap team is going to take some of the data they've gathered and create a 3D representation of it using an XSLT stylesheet.

Downloads

Description	Name	Size	Download method
Sample code	x-layerfiles.zip	412KB	HTTP

[Information about download methods](#)

Resources

Learn

- Check out the rest of this *developerWorks* tutorial series on XSLT for more information on advanced uses for XSL transformations:
 - "[Analyze non-XML data with XSLT](#)" (December 2003)
 - "[XSLT as an analysis tool](#)" (February 2004)
 - "[Create 3D representations with XSLT and SVG](#)" (April 2004)
 - "[Tie in data with Web services and XSL Transformations](#)" (August 2004)
- Don't know XSLT? Better try Nicholas Chase's [tutorial on beginning XSLT](#) (*developerWorks*, March 2003).
- Read the [XPath Recommendation](#) at the W3C for a complete look at its capabilities (<http://www.w3.org/TR/xpath>).
- For a good, basic JavaScript tutorial, go to: <http://www.w3schools.com/js/default.asp>.
- Find the W3C official documentation on the Document Object Model (DOM) at <http://www.w3.org/DOM/>.
- If you're adventurous, and want to try to port this application to Mozilla, check out [the Mozilla DOM Reference](#) (<http://www.mozilla.org/docs/dom/>). The nice thing about Mozilla is that it uses the standard DOM, and doesn't cheat by adding all kinds of nasty, proprietary methods.
- To get a high-level feel for exactly how powerful XSLT can be from a functional programming perspective, check out [FXSL](#) (<http://fxsl.sourceforge.net>).
- Learn how XSL transformations can be used with WebSphere Application Server in "[XSL Transform Basics with WebSphere Application Server Version 4.0x](#)" by Joel Sundman (http://www.ibm.com/developerworks/websphere/techjournal/0204_sundman/sundman.html) (*developerWorks*, April 2002).
- Find more XML resources on the [developerWorks XML zone](#).
- [Browse for books](#) on these and other technical topics.
- Finally, find out how you can become an [IBM Certified Developer in XML and related technologies](#).

Get products and technologies

- Get a free word processing and office suite from www.openoffice.org that is as

powerful as the MS Office suite, then use that to create XML documents from your word processing files (<http://www.openoffice.org/>).

About the author

Chuck White

Chuck White, a [Studio B](#) author, has been working with XML since before its official inception in February, 1998. He was co-author of [Mastering XML Premium Edition](#) (with Linda Burman and the W3C's XML Activity Lead, Liam Quin) and author of [Mastering XSLT](#), both from Sybex Books. His latest books are [Developing Killer Web Apps with Dreamweaver MX & C#](#) (also for Sybex Books) and *HTML, XHTML, and CSS Bible, 3rd Edition* for Wiley, for which he is co-author with Steve Schafer. Chuck is currently working with the XSL Team at eBay as a project consultant and Web engineer.