newtFire {dh|ds}
Maintained by: Elisa E. Beshero-Bondar (ebb8 at pitt.edu) Last modified: Sunday, 22-Oct-2017 21:10:14 EDT.
Powered by firebellies.

XSLT Exercise 5

The input collection on our DHClass-Hub

For this assignment and the next, you will be working with a digitized XML collection of Emily Dickinson's poems and you will need to access this collection in GitHub. This assignment requires you to use our **DHClass-Hub** so you can work with a **a local directory of files** rather than just one at a time as we have been doing up to this point. Here is how to access the directory:

- Sync the DHClass-Hub to your computer, or clone the repository if it is not already on your computer. (If you have not synced or cloned a GitHub repository in a while; please see the instructions posted in our Readme file).
- When you have synced the repository, open the DHClass-Hub locally on your computer, and find in it the **Assignment-Files** directory. Inside it is a directory named **Dickinson** that contains a eleven XML files that we are working with as a collection in this assignment.
- **Copy this Dickinson directory to some other location on your computer** outside of your GitHub directories. (We do not want you to push your homework to the whole class over our DHClass-Hub, so we just need you to make your own private copy of this directory to work with in the same folder in which you do your homework for this assignment and the next.
- Do not rename the file folder or the files inside, as we need to refer to them as a coherent collection.

Please be careful to **copy** rather the move the directory out of GitHub! If you move it out of the directory, the next time you sync our DHClass-Hub, GitHub will prompt you to commit the change and push it, which will effectively eliminate the Dickinson folder. One of us instructors can easily put it back if that happens, but please alert us ASAP if something goes awry!

Working with a Collection of Files in XSLT

Emily Dickinson made little bundles of her manuscript poems with a needle and thread, and these have come to be known as *fascicles* by Dickinson scholars. We have digitally reproduced a bundle that Dickinson scholars have named Fascicle 16 by using a folder or directory, which holds a digital collection of files together. We can process a whole directory of files using the `collection()` function in XSLT, so we can represent content from a whole collection of XML files in one or more output HTML files. One useful application for working with a collection is to process several short XML files and unify them on a single HTML page designed to merge their content. In this case, we will be representing the poems encoded in eleven small XML files inside one HTML page, which we will produce with a table of contents giving poems by number and first lines, followed by the full text of the poems themselves, formatted in HTML with numbered lines. Since these poems are all encoded with the same structural elements, we can use the `collection()` function to reach into them as a group, and output their content one by one based on their XML hierarchy. Really, we are treating the collection itself as part of the hierarchy as we write our XSLT, so we move from the directory down into the document node of each file to do our XSLT processing.

Using modal XSLT

Besides working with a collection of files, the other interesting new application in this assignment is **modal XSLT**, which lets you process the same nodes in your document in two different ways. How can you output the same element contents to sit as list items in a table of contents at the top of an HTML page, *and also* as headers positioned throughout the body of your document, below the table of contents? Wouldn't it be handy to be able to have two completely different template rules that match exactly the same elements: one rule to output the data as list items in the table of contents, and the other to output the same data as headers? You can write two template rules that will match the same nodes (have the same value for their `@match` attribute), but how do you make sure that the correct template rule is handling the data in the correct place?

To permit us to write multiple template rules that process the same input nodes in different ways for different purposes, we write **modal XSLT**, and that is what you will be learning to write with this assignment. Modal XSLT allows you to output the same parts of the input XML document in multiple locations and treat them differently each time. That is, it lets you have two different template rules for processing the same elements or other nodes in different ways, and you use the `@mode` attribute to control how the elements are processed *at a particular place* in the transformation. Please read the explanation and view the examples in Obdurodon's tutorial on Modal XSLT before proceeding with the assignment, so you can see where and how to set the `@mode` attribute and how it works to control processing.

Overview of the assignment

For this assignment you want to produce in one HTML page our collection of Emily Dickinson's eleven poems in Fascicle 16, and that page needs to have a table of contents at the top. The table of contents should have one entry for each poem, which produces the information we have encoded in `<title>` element that is a descendant of the `<body>` element in our XML source code, together with the first line, and a count of the number of variants we have recorded in each poem. Below the full table of contents (one line for each poem) you should render the complete text of all eleven poems, and wrap span elements around the text we have marked as variants, ideally by using a `@class` attribute that holds the same information as the `@wit` attribute on the `<rdg>` element in our source texts. To generate the attribute value on @class, we used an Attribute Value Template, which you should read about here. You can see our output at http://newtfire.org/dh/dickinson-5.html.

Housekeeping with the stylesheet template: From TEI to XHTML

Our Emily Dickinson collection is coded in the TEI namespace, which means that your XSLT stylesheet must include an instruction at the top to specify that when it tries to match elements, it needs to match them in that TEI namespace. When you create an new XSLT document in <oXygen/> it won't contain that instruction by default, so whenever you are working with TEI you need to add it (See the text in blue below). To ensure that the output would be in the XHTML namespace, we added a default namespace declaration (in purple below). To output the required DOCTYPE declaration, we also created `<xsl:output>` element as the first child of our root `<xsl:stylesheet>` element (in green below), and we needed to include an attribute there to omit the default XML declaration because if we output it that XML line in our XHTML output, it will not produce valid HTML with the w3C and might produce quirky problems with rendering in various web browsers. So, our modified stylesheet template and xsl:output line is this, and you should copy this into your stylesheet:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xpath-default-namespace="http://www.tei-c.org/ns/1.0"
         xmlns:math="http://www.w3.org/2005/xpath-functions/math"
    exclude-result-prefixes="xs math"
    xmlns="http://www.w3.org/1999/xhtml"
         version="3.0">

         <xsl:output method="xhtml" encoding="utf-8" doctype-system="about:legacy-compat"
        omit-xml-declaration="yes"/>

</xsl:stylesheet>
```

How to begin

Begin by forgetting about the table of contents, and concentrate on just outputting the full text of the poems. Except for having to pull the poems from a collection of files, this is just like the XML-to-HTML transformations you have already written, and you'll use regular template rules (without a `@mode` attribute) to perform the transformation.

The collection() function: Here is how we write and run XSLT to process a collection of files. Just ahead of the first template match, after the `<xsl:output method>` statement, we define a **variable** in XSLT, which simply sets up a convenient shorthand for something complicated that we need to use more than once, so we don't have to keep retyping it.

<xsl:output method="xml" encoding="utf-8" indent="yes" doctype-system="about:legacy-compat"/>
`<xsl:variable name="dickinsonColl" select="collection('Dickinson')"/>`

An `xsl:variable` works by designating an `@name` which holds any name you like to refer to it later (we have used "dickinsonColl" here to refer to the Dickinson collection of files), and with `@select` it holds anything you wish: a complicated XPath expression or a function, or whatever it is that is easier to store or process in a variable rather than typing it out multiple times. We use variables to help keep our code easy to read! In this case, we are using a variable to define our collection, using the `collection()` function in the `@select` attribute. The `collection()` function is set to *designate the directory location of the collection of poems in relation to the stylesheet I am currently writing*. My XSLT is saved in the directory immediately above the Dickinson collection, so I am simply instructing the XSLT parser to take a path-step down to it by designating Dickinson inside the collection function. (You may wish to save your stylesheet in relation to the Dickinson collection just as I did, but in case you did not, you will simply need to figure out how to step up or down your file directory structure to reach the Dickinson folder, using `..` to climb up and `/` or `//` to step down.)

Within the stylesheet as we will see below, we will call this variable whenever we need it, to show how we are stepping into our collection of poems. That will happen in the first template rule that matches on the root element. Open any one of the input XML files in the Dickinson collection in <oXygen/> and you will see that the title and content of the poems are all coded within the `<body>` element, so we can write this stylesheet to look through the whole collection of files and process only the elements below `<body>`. You call or invoke the variable name for the collection by signalling it first with a dollar sign `$`, giving the variable name, and then simply step down the descendant axis straight to the `<body>` element in each file. Here is how the code looks to call or invoke the variable in our first template match:

```
<xsl:apply-templates select="$dickinsonColl//body"/>
```

Note on running the transformation: Unlike other transformations we do on single XML files, when we run the XSLT in <oXygen/> it actually doesn't matter what file we have selected in the XML input, because we have indicated in the stylesheet itself what we are processing, with the `collection()` function. We can actually set even a file that is outside of our collection as the input XML file (and we ran it successfully with the HTML file of the previous exercise selected). You do need to enter something in the input window, but when you work with the `collection()` function, your input file is just a dummy or placeholder that <oXygen/> needs to have entered so it can run your XSLT transformation.

In our HTML output (scroll down past the table of contents, to where the full text of the poems is rendered), the Poem number (and publication info in parentheses) are inside an HTML `<h2>` element and the stanzas of each poem are held and spaced apart using HTML `<p>` elements. To make each line of the poems start on a new line, we add an HTML empty `<br/>` ("[line] break") element at the end of each line within the stanza. If you don't include the `<br/>` elements, the lines will all wrap together in the browser. Numbering the lines is optional for our assignment, but we have done so in our sample output by using the `count()` function over the `<l>` elements on the `preceding::` axis (which we used instead of `preceding-sibling::`, because we wanted to number lines by counting them consecutively within each file rather than within each line group. (You can read about the `preceding::` axis in the Michael Kay book on page 612.) Here's the HTML output for one of our poems:

```
 <h2 id="p1611">Poem 11 </h2>
<p>
   1: He showed me Hights I never saw—<br/>
   2: "Would'st Climb," —He said?<br/>
   3: I said—"Not so"—<br/>
   4: "With me—" He said—"With me"?<br/>
   5: He showed me Secrets—Morning's Nest—<br/>
   6: The Rope the Nights were put across—<br/>
   7: "And now—"Would'st have me for a Guest"?<br/>
   8: I could not find my "Yes".
</p>
<p>
   9: And then, He brake His Life—And lo,<br/>
   10: A Light, for me, did solemn glow,<br/>
   11:
         <span class="var0">The steadier, as my face withdrew</span>
         <span class="var1">The larger—as my face withdrew</span>

   <br/>
   12: And could I, further, "No"?
</p>
```

The fine print: Don't worry if your HTML output isn't wrapped the same way ours is, if it puts the empty line break elements at the beginnings of lines instead of at the ends, or if it serializes (spells out) those empty line break elements as `<br></br>` instead of as `<br/>`. Those differences are not *informational* in an XML context. You can open your HTML output in <oXygen/> and pretty-print it if you'd like, which may make it easier to read, but as long as what you're producing is valid HTML and renders the text appropriately, you don't have to worry about non-informational differences between your markup and ours.

More fine print: You need a line break only between lines, which is to say that you don’t need a `<br/>` element at the end of the last line of the poem because that’s the end of the containing `<p>`, and not between lines. In our solution we used an `<xsl:if>` element to check the position of the line and output the `<br/>` only for non-final lines. If you’re feeling ambitious, you can look up `<xsl:if>` at http://www.w3schools.com/xsl/xsl_if.asp or by searching for xsl:if on Obdurodon’s XSLT Advanced Features tutorial, or looking it up in Michael Kay so you can perform this check yourself. If not, you can just output the `<br/>` element after all the concluding lines of line-groups in the poems. That’s not really considered good HTML style, and you don’t want to do it in your own projects, but it won’t interfere with the legibility in the browser and we’ll let it pass for homework purposes.

Once your poems are all being formatted correctly in HTML, you can add the functionality to create the table of contents at the top, using modal XSLT.

Adding the table of contents

The template rule for the document node in our solution, revised to output a table of contents with all the information we wish to show before the text of the poems, looks like the following:

```
            <xsl:variable name="dickinsonColl" select="collection('Dickinson')"/>

     <xsl:template match="/">
      <html>
          <head><title>Emily Dickinson’s Fascicle 16</title></head>
          <body>

          <h1>Emily Dickinson’s Fascicle 16</h1>
              <h2>Table of Contents</h2>
        <ul><xsl:apply-templates select="$dickinsonColl//body" mode="toc"/></ul>
              <hr/>
  <!--ebb: This template rule sets up my "toc" mode for the table of contents,
      so that in the top part of the document we’ll output a selection of the body elements specially formatted for my Table of
      and so that in another section of my document below, which I’ve put inside a <div> element, we can also output the full t
      Notice how I have invoked my variable multiple times here with the $ notation: $dickinsonColl -->
          <div id="main">
             <xsl:apply-templates select="$dickinsonColl//body"/>

          </div>

          </body>

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

The highlighted code is what we added to include a table of contents, and the important line is `<xsl:apply-templates select="$dickinsonColl//body" mode="toc"/>`. This is going to apply templates to each poem *with the `@mode` attribute value set to "toc"*. The value of the `@mode` attribute is up to you (we used "toc" for "table of contents"), but whatever you call it, setting the `@mode` to any value means that only template rules that also specify a `@mode` with that same value will fire in response to this `<xsl:apply-templates>` element. Now we have to go write those template rules!

What this means is that when you process the `<body>` elements to output the full text of the poems, you use `<xsl:apply-templates>` and `<xsl:template>` elements without any `@mode` attribute. To create the table of contents, though, you can have `<xsl:apply-templates>` and `<xsl:template>` elements that select or match the same elements, but that specify a mode and apply completely different rules. A template rule for `<body>` elements in table-of-contents mode will start with `<xsl:template match="$dickinsonColl//body" mode="toc">`, and you need to tell it to create an `<li>` element that contains the text of the `<title>` element and a first line, both fetched from the poem in the input XML collection of files. The rule for those same elements not in any mode will start with `<xsl:template match="$dickinsonColl//body">` (without the `@mode` attribute). That rule will create the `<h2>` header to hold the text of the `<title>` element and then output the full text of the poem in a `<p>`, with `<br/>` elements between the lines. In this way, you can have two sets of rules for the poems, one for the table of contents and one to output the full text, and we use modes to ensure that each is used only in the correct place.

Remember: both the `<xsl:apply-templates>`, which tells the system to process certain nodes, and the `<xsl:template>` that responds to that call and does the processing must agree on their mode values. For the main output of the full text of every poem, neither the `<xsl:apply-templates>` nor the `<xsl:template>` elements specifies a mode. To output the table of contents, both specify the same mode.

Completing and checking your work

- *Before submitting your homework, you must run the transformation at home* to make sure the results are what you expect them to be. Remember, there’s a guide to running XSLT transformations inside <oXygen/> in our Intro to XSLT tutorial. If you don’t get the results you expect and can’t figure out what you’re doing wrong, remember that you can post a query to our DHClass-Hub Issues board. You can’t just ask for the answer, though; you need to describe what you tried, what you expected, what you got, and what you think the problem is. We often find, just as we’re preparing to post our own queries to coding discussion boards, that having to write up a description of the problem helps us think it through and solve it ourselves. We’re also encouraging you to discuss the homework on the discussion boards because that’s also helpful for the person who responds. Answering someone else’s inquiry and troubleshooting someone else’s problem often helps us clarify matters for ourselves!
- When you complete this assignment, submit your XSLT file and your output HTML file to Courseweb, following our usual homework file-naming conventions. You need not generate CSS for this, because we will ask you to create one for a modified version of this output in the next assignment.