newtFire {dh|ds}
Maintained by: Elisa E. Beshero-Bondar (ebb8 at pitt.edu) BY-NC-SA Last modified: Monday, 19-Mar-2018 06:42:56 EDT.
Powered by firebellies.

XSLT Exercise 2

The input text

For this assignment we'll be producing HTML from a TEI XML file developed by the Akira project team on newtFire in the spring of 2018. The XML file is available here: http://newtfire.org/dh/Akira_tei.xml. You should right-click on this link, download the file, and open it in <oXygen/> (or you can pull it in locally from the DHClass-Hub where it is in Class Examples --> XSLT).

Housekeeping: Setting Up a TEI to HTML Transformation

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 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, you should copy our modified stylesheet template and xsl:output line here 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>
```

Overview of the assignment

We're going to work with this entire XML document (on all levels of the hierarchy), concentrating on processing the XML "salad" of mixed text and in-line elements to style them for presentation on the web in HTML's limited tagset. You can use some of the basic HTML in-line elements, like `<em>` for emphasis or `<strong>` for strong emphasis, but you'll also want to use CSS to set some elements to have different colors or background colors or to alter borders or fonts or font sizes or font styles (e.g., italic) or font weights (e.g., bold) or add text decoration (e.g., underlining) or text transformation (e.g., convert to all upper case), really anything stylistically possible.

For this assignment, we aim to produce an HTML reading view of the Akira script, to help orient readers to the cast of characters and to help visualize some special markup the team has applied to help locate special scenes. The exercise will help orient you to styling and layout decisions connected with transforming XML to HTML, and it will also give you a chance to "remix" the XML creatively into something designed for display in a web browser.

Surveying the "input" TEI document, you will see it has a TEI header to hold information *about* the Akira document, its source, and the work of the project team. We will pick and choose which portions of this to process and output in the body of an HTML document (remembering that the `body` element in HTML is the part that is visible in a web browser), and we may want to change the order it appears in the new document. We will take some material from the TEI header to display in the HTML body, for example, and we will try sorting the list of characters that appears in the TEI `profileDesc` to present a title and cast list at the top of the HTML file.

The following portions of the input document are especially important to us to display in HTML:

- The `profileDesc` in the `teiHeader` contains a list of characters we will want to sort alphabetically. We will want to output make a cast list or key of names and abbreviations.
- We will want to process the script itself, with its `sp` elements and the information coded in the attributes, including the speaker id, the number, and, where available, the time segment.
- Within those `sp` elements we want to process the `l` elements to hold them on separate lines.
- When `sp` elements are inside special sections coded as `spGrp`, we want to hold these in their own HTML `div` elements to distinguish them from other parts of the script.

Some of these elements are located inside the `<teiHeader>`. Some are nested unevenly at different levels of the XML tree hierarchy, like some of the `<sp>` elements nested inside `<spGrp>` elements, and the `<l>` elements sitting inside of `<sp>` elements. You may not be sure at the outset which elements can be inside which other ones, or how deeply they can nest. Happily, with XSLT, unlike with many other programming languages, you don't need to care about those questions!

An example of possible desired output can be found here http://newtfire.org/dh/akiraSample.html, though we did not style the body paragraphs in this output file. It is important to note that the majority of the styling choices on this file are controlled with a CSS file. You will make your own CSS and relate it to your XSLT; therefore, your stylistic choices might vary greatly from ours and your output may look completely different. What should look relatively similar is the underlying raw HTML, which is generated by running the XSLT. By viewing the page source of our output you can review the underlying raw HTML (http://newtfire.org/dh/akiraSample.html).

Guide to Approaching the Problem

In XSLT, processing something normally happens in two parts. You normally have an `<xsl:apply-templates>` element that tells the system what elements (or other nodes) you want to process, and you then have an `<xsl:template>` element that tells the system exactly how you want to process those elements, that is, what you want to do with them. If you find the image helpful, you can think of this as a situation where the `<xsl:apply-templates>` elements *throw some nodes out into space and say "would someone please process these?"* and the various `<xsl:template>` elements sit around watching nodes fly by, and when they **match** something, they *grab it and process it*.

Therefore, for this assignment, your XSLT transformation (after all the housekeeping) should have several template rules:

1. Begin with **a special template rule for the document node** (`<xsl:template match="/">`), in which you set up the basic HTML structure: the `<html>` element, `<head>` and its contents, and `<body>`.
2. Inside the `<body>` element that just created, write an HTML `h1` element to hold the title you want viewers to see on the web page, and use `xsl:apply-templates` inside to select the part of the input TEI that will give you the title. When you need to set the value of the `@select` attribute on `xsl:apply-templates`, you are being choosy, pulling *just* what you need into position where you want it.
3. Beneath the main title, create a secondary header in HTML (an `h2` element), just type the words "Cast List" into it. You are basically writing some HTML code within the XSLT document!
4. Set up an HTML `table`, and give it a row of two `th` (table header) cells to help start a table of abbreviated IDs and values. After the table header, you will want to simply apply-templates to select each TEI `person` element to process in a new template rule. Anything that is going to have to be processed multiple times needs to just be called once in the special template match on the document node, to apply-templates selecting these elements.
5. Write a new template rule to match on the TEI `person` element and output (each time it finds `person`) an HTML table row (`tr`) containing two cells (`td`). Inside each cell, pull the relevant information from the person element that you wish to present. (We recommend outputting the `@xml:id` in one table cell, and just the *first* `persName` element.)
6. Then create separate template rules that match on each of the inline elements we planned above to match and style. Each rule will be "called" or "fired" as a result of the preceding `<xsl:apply-templates>` selection from our first template rule.

In this case, then, your `@select` on the `<xsl:apply-template>` elements inside the template rule for the document node will tell the system what specific elements (using their XPath location in the source XML) you want to appear and where in your output HTML you wish for them to appear. You create the order each selection appears by placing the various `<xsl:apply-template>` elements in the desired order inside of that first template rule matching on the document node. This will tell the system that you want to **select** only certain elements, at which point the template rule for the document node will call out what portions of the document need to be processed at this particular point. The processing work actually gets done by the other `<xsl:template>` rules, the ones that you write to then **match** on the elements that need styled.

The elegant simplicity of <xsl:apply-templates>

Akira's `spGrp` elements wrap around clusters of speeches unpredictably. Similarly, if you were processing prose paragraphs with markup floating around unpredictably with the text, you would have an unpredictable combination of elements that we call "mixed content", with varied and unpredictable combinations of elements. This is the problem that *declarative* XSLT was designed to solve. With a traditional *procedural* programming language, you'd have to write rules like "inside the body, if there's a `<spGrp>` do X, but if there isn't do Y, and, oh, by the way, check whether there's a `<l>` or a `<p>` inside the `<sp>` elements, etc." That is, most programming languages have to tell you what to look for at every step. The elegance of XSLT is that all you have to say inside paragraphs and other elements is "I'm not worried about what I'll find here; just process **(apply templates to)** all my children, *whatever they might be*."

The way to deal with mixed content in XSLT is to create a template rule for every element you care about and use it to output whatever HTML markup you want for that element. Then, inside that markup, you can include a general `<xsl:apply-templates/>`, not specifying a `@select` attribute. For example, if you want your `<persName>` elements to be tagged with the HTML `<strong>` tags, which means "strong emphasis" and which is usually rendered in bold, you could have a template rule like:

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

You don't know or care whether `<persName>` has any children nodes or, if it does, what they are. Whatever they are, this rule tells the system to try to process them, and as long as there's a template rule for them, they'll be taken care of properly somewhere else in the stylesheet. If there are no children nodes, the `<xsl:apply-templates/>` will apply vacuously and harmlessly. As long as every element tells you to process its children, you'll work your way down through the hierarchy of the document without having to know which elements can contain which other elements or text nodes.

Taking stock: when to use `@select`

In our XSLT tutorial we describe the use of `<xsl:apply-templates select="…"/>` which specifies exactly what you want to process and where. That makes sense when your input and output are very regular in structure. *Use the `@select` attribute when you know exactly what you're looking for and where you want to put it*. We will want to use `<xsl:apply-templates select="…"/>` in order to grab all of the `<person>` elements sitting inside of the `<particDesc>` element so we can output them up in a Cast List near the top of our HTML file, separate from the Akira script that we want to come out below, selected from the TEI `<body>` element. We will also want to use the `<xsl:apply-templates select="…"/>` in order to reach for attribute values like the `@xml:id` on `person` to output them inside HTML table cell (`td`) elements. By setting up these very specific selections of these elements and attributes, we are paring down or "trimming" the XML tree of the input document to designate exactly and only what we want. Remember. what is represented in the `<html>` element of your XSLT is the basic superstructure of your output HTML document. The content inside the HTML `<head>` element, including the `<title>` element, will not appear in the web browswerunless someone is reading your HTML source code. Hence the importance in creating visible body headings with elements (`<h1>`, `<h2>`, etc.) that contain the document title information.

After you have selected the portions of the document to process for your Cast of Characters table, and to output the script, for the rest of this assignment you don't need to write the template rules in any particular order. Those template matches will fire as elements in the document turn up to be processed, whenever it comes up. Basically, `<xsl:apply-templates/>` without the `@select` attribute says "apply templates to whatever you find." *Omit the `@select` attribute where you don't want to have to think about and cater to every alternative individually*. (You can still treat them all differently because you'll have different template rules to "catch" them, but when you assert that they should be processed, you don't have to know what they actually are.)

Sorting

An alphabetically sorted Cast of Characters may be useful for humans who want to look up more information about a speaker they see in the script. We want to make an alphabetized list sorted by the abbreviated name given in the `person/@xml:id`. Start by looking up `<xsl:sort>` in the Michael Kay book or at https://www.w3schools.com/xml/xsl_sort.asp. So far, if we want to output our cast in the order in which they occur *Akira* script, we've used a self-closing empty `<xsl:apply-templates/>` to select them with something like `<xsl:apply-templates select="descending::particDesc//person"/>`. But the self-closing empty element tag is informationally identical to writing the start and end tags separately with nothing between them, that is:

```
<xsl:apply-templates select="descendant::particDesc//person>
</xsl:apply-templates>
```

To cause the elements being processed to be sorted first, you need to use this alternative notation, with separate start and end tags, because you need to put the `<xsl:sort>` element between the start and end tags. If you use the first notation, the one with a single self-closing tag, there's no "between" in which to put the `<xsl:sort>` element. In other words, you want something like:

```
<xsl:apply-templates select="descendant::particDesc//person">
  <xsl:sort select="what specific aspect of the person element you want to sort on, such as an attribute or child element"/>
</xsl:apply-templates/>
```

Without an `@select` attribute on `<xsl:sort>` this would sort on child text content of the `<person>` elements alphabetically by their text value (and if there is no text, there won't be anything to sort, so the sort will fail). Since our `person` elements only contain other elements, we need to use the `@select` attribute on `<xsl:sort>`. Note that you can set an `@order` attribute to sort in ascending or descending order. Also you do not have to sort alphabetically. You can sort by numerical counts of something, for example, how often a particular character appears in the script. (We sorted our Cast Table in both ways.) **Challenge:** Can you figure out how to sort based on a count of the number of appearances, or the number of times the character speaks in the production? **Hint:** To do this requires searching the XML tree for the `sp` elements whose `@who` attribute values match up with the current person element. You will need a string-matching function, because the `@who` attributes have a `#` in front of the id. Typically we strip that off using the `substring-after()` function, so we look for the `substring-after(@who, '#')` to see where that substring = the `current()` xml:id. We need to use `current()` to designate the specific person element being processed (it's a little like processing $i in a for-loop).

What should the output look like

We are sure you can do better than our sample output! HTML provides a limited number of elements for styling in-line text, which you can read about at http://www.w3schools.com/html/html_formatting.asp. You can use any of these in your output, but think about your decisions. For layout purposes, *block elements* like `div` or `h1` or `p` literally take up a rectangular "block" on the page and can be styled accordingly (given padding, etc. *Inline elements*, like `span` or `em` or `strong` are meant to run within blocks (inside paragraphs, for example), and are good for highlighting within the line, for example to style speaker names or speech numbers to introduce each speech. Finally, presentational elements, the kind that describe how text looks (e.g., `<i>` for "italic"), are generally regarded as less useful than descriptive tags, which describe what text means (e.g., `<em>` for "emphasis"). Both of the preceding are normally rendered in italics in the browser, but the semantic tag is more consistent with the spirit of XML than the presentational one.

The web would be a dull world if the only styling available were the handful of presentational tags available in vanilla HTML. In addition to those options, there are also ways to assign arbitrary style to a snippet of in-line text, changing fonts or colors or other features in mid-stream. To do that:

1. Before you read any further in this page, read Obdurodon's Using `<span>` and `@class` to style your HTML page.

2. To use the strategies described on that page, create an XSLT template rule that transforms the element you want to style to an HTML `div` or `<span>` element with a `@class` attribute. For example, you might transform `<spGrp>` in the input XML to `<div class="spGrp">`...child nodes (processed in XSLT with `<xsl:apply-templates/>`) ...`</span>` in the output HTML. You can then specify CSS styling by reference to the `@class` attribute, as described in the page we link to above.

   Note that you can make your transformations very specific. For example, instead of setting all `<sp>` elements to the same HTML `@class`, you can create separate template rules to **match** on special `sp[@who="#colonel"]` and `sp[@who="#doctor"]` according to their attribute values. (You can even use the pipe (|) to unify these as two options for a template match:

   ```
   <xsl:template match="sp[@who='#doctor'] | sp[@who='#colonel']">
             <span class="commanders">
             <strong><xsl:apply-templates select="@who"></strong>
             <xsl:apply-templates/>
             </span class="commanders">
   </xsl:template>>
   ```

   Notice how we used two `<xsl:apply-templates/>` statements here, one which selected an attribute value to output, and the other just to process whatever child contents of the <sp> elements turn up. Around both of them, we set a special `<span>` element with a logical `@class` (we used the value "commanders" to help associate these two controlling figures in *Akira*). In our CSS we make reference to the `@class`, again as described in the page we link to above.

3. Setting `@class` attributes in the output HTML makes it possible to style the various `<span>` elements differently according to the value of those attributes, but you need to create a CSS stylesheet to do that. Create the stylesheet (just as you've created CSS in the past), and specify how you want to style your `<span>` elements. Link the CSS stylesheet to the XSLT by creating the appropriate `<link>` element inside of the HTML `<head>` element of your XSLT (you can remind yourself of the `<link>` element format by referencing our CSS Tutorial).
4. Besides wrapping your `<xsl:apply-templates/>` in `<span>` elements and other HTML elements, you might consider adding extra spaces or text outside some of these as well. To do this, experiment with inserting `<xsl:text>...</xsl:text>` where you would like spaces or characters (say a colon and some white space to follow a speaker name in the script).
5. You may want to style your table so you can see the outlines of the table cells, and add colors and styling. For some guidance, see the w3schools CSS tutorial on tables, which shows you some nifty tricks like how to style every other row to shade it differently.

Your Final Results

What you should produce, then, is:

- An XSLT stylesheet that transforms the contents of the source document into HTML, giving us at least one sorted Cast List and a reading view of the *Akira* script.
- The resulting HTML should also style and at least some of those styles should be set using block `<div>` and inline `<span>` elements with the `@class` attribute to group related kinds of elements visually.
- You need to create a CSS file, **linked to your output HTML**, that specifies how to style the output document. You can look up the most useful of the available CSS properties at http://www.w3schools.com/css/. We'd suggesting following the links on the left under "CSS styling" for styling backgrounds, text, and fonts, as well as the link for borders under "CSS box model".

Important

- *Before submitting your homework, you must run the transformation at home, and open the results as a new file in <oXygen/>* to make sure the results are what you expect them to be. (There's a guide to running XSLT transformations inside <oXygen/> on Obdurodon at http://dh.obdurodon.org/oxygen-xslt-configuration.html.) 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. Don'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 DHClass-Hub Issues 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 CSS file to Courseweb, following our usual homework file-naming conventions. We will run your XSLT transformation to see what output it generates, so you do not need to submit your output file. However, it is important that you include your CSS so we can locally associate it to your XSLT (keeping them in the same folder space) and see your final output. **Link the CSS in the XSLT for us, so that when we run the XSLT it generates the `<link>` element automatically.**