Creating indexes using RenderX extensions

Building page number lists for indexes is not possible within XSL 1.0. RenderX XEP provides this functionality via extension elements/properties.

indexes creation can be devided in two steps: first one presumes defining index terms and ranges in the text, second one – inserting index page and setting index entries properties, which control visual appearance. First step performed using `rx:key` property (applicable to any element that can carry an id attribute) and `rx:begin-index-range`, `rx:end-index-range` elements usefull for creating ranges in the index. Second step utilizes `rx:page-index` element together with `rx:index-item` elements. Both can have standard font and style properties, while latter can also carry four special properties: `ref-key` (required), `range-separator`, `merge-subsequent-page-numbers`, `link-back`. You can find detailed description of these elements/properties in XEP documentation included in distribution.

XEP 4.0 Reference for Java, section 3.6.3. "Indexes" (reference.pdf)

Following excerpt shows how code with two stand alone index terms and one index range will looks like:

```
<fo:block ...>Processing a Stylesheet</fo:block>
 <fo:block ...>
   <rx:begin-index-range rx:key="transformation" id="transformation-id
An <fo:inline rx:key="XSL">XSL</fo:inline>
   <fo:inline rx:key="stylesheet processor-primary">stylesheet process
   accepts a document or data in XML and an XSL stylesheet and produces
   ...
   <rx:end-index-range ref-id="transformation-id"/>
 </fo:block>
```

And this code shows how index page will be organized:

```
   <fo:block font="bold 16pt Helvetica" ...>INDEX</fo:block>
   ...
       <fo:block font="12pt Times">
      stylesheet processor
         <rx:page-index>
           <rx:index-item ref-key="stylesheet processor-primary"
                          color="blue"
                          font-style="italic"
                          font-weight="bold"
                          link-back="true"/>
         </rx:page-index>
       </fo:block>
       ...
       <fo:block font="12pt Times">
      transformation
         <rx:page-index>
           <rx:index-item ref-key="transformation-primary"
```

```
                        color="blue"
                        font-style="italic"
                        font-weight="bold"
                        link-back="true"/>
      </rx:page-index>
    </fo:block>
    ...
    <fo:block font="12pt Times">
   XSL
<rx:page-index>
        <rx:index-item ref-key="XSL-primary"
                        color="blue"
                        font-style="italic"
                        font-weight="bold"
                        link-back="true"/>
        <rx:index-item ref-key="XSL"
                        color="blue"
                        font-style="italic"
                        font-weight="normal"
                        link-back="true"/>
      </rx:page-index>
    </fo:block>
...
```

It creates two index entries, all linked back to the index terms location in the main text. By default, page numbers separated by colon symbol followed by space; subsequent page numbers not merged; en dash ("–", U+2013) used to separate page numbers in a range.

Starting from the next page you can see some formated text which contains a bunch of keywords, which should be placed in the index. Some of index terms have ranged structure and so must be represented in the index by the pages range. Text below devided into several page-sequences in order to demonstrate how index ranges works over page-sequence borders.

Introduction and Overview

This specification defines the *Extensible Stylesheet Language (XSL)*. XSL is a language for expressing stylesheets. Given a class of arbitrarily structured XML documents or data files, designers use an XSL stylesheet to express their intentions about how that structured content should be presented; that is, how the source content should be styled, laid out, and paginated onto some presentation medium, such as a window in a Web browser or a hand-held device, or a set of physical pages in a catalog, report, pamphlet, or book.

Processing a Stylesheet

-> An *XSL stylesheet processor* accepts a document or data in XML and an XSL stylesheet and produces the presentation of that XML source content that was intended by the designer of that stylesheet. There are two aspects of this presentation process: first, constructing a result tree from the XML source tree and second, interpreting the result tree to produce formatted results suitable for presentation on a display, on paper, in speech, or onto other media. The first aspect is called *tree transformation* and the second is called *formatting*. The process of formatting is performed by the *formatter*. This formatter may simply be a rendering engine inside a browser.

Tree transformation allows the structure of the result tree to be significantly different from the structure of the source tree. For example, one could add a table-of-contents as a filtered selection of an original source document, or one could rearrange source data into a sorted tabular presentation. In constructing the result tree, the *tree transformation* process also adds the information necessary to format that result tree.

Formatting is enabled by including formatting semantics in the result tree. Formatting semantics are expressed in terms of a catalog of classes of *formatting objects*. The nodes of the result tree are formatting objects. The classes of *formatting objects* denote typographic abstractions such as page, paragraph, table, and so forth. Finer control over the presentation of these abstractions is provided by a set of *formatting properties*, such as those controlling indents, word- and letter spacing, and widow, orphan, and hyphenation control. In *XSL*, the classes of *formatting objects* and *formatting properties* provide the vocabulary for expressing presentation intent.

The *XSL* processing model is intended to be conceptual only. An implementation is not mandated to provide these as separate processes. Furthermore, implementations are free to process the source document in any way that produces the same result as if it were processed using the conceptual XSL processing model. A diagram depicting the detailed conceptual model is shown below.

Tree Transformations

-> *Tree transformation* constructs the result tree. In XSL, this tree is called the *element and attribute tree*, with objects primarily in the "formatting object" namespace. In this tree, a formatting object is represented as an XML element, with the properties represented by a set of XML attribute-value pairs. The content of the formatting object is the content of the XML element. Tree transformation is defined in the *XSLT* Recommendation. A diagram depicting this conceptual process is shown below.

The XSL stylesheet is used in tree transformation. A stylesheet contains a set of tree construction rules. The tree construction rules have two parts: a pattern that is matched against elements in the source tree and a template that constructs a portion of the result tree. This allows a stylesheet to be applicable to a wide class of documents that have similar source tree structures.

In some implementations of *XSL/XSLT*, the result of tree construction can be output as an XML document. This would allow an XML document which contains *formatting objects* and *formatting properties* to be output. This capability is neither necessary for an XSL processor nor is it encouraged. There are, however, cases where this is important, such as a server preparing input for a known client; for example, the way that a *WAP* (http://www.wapforum.org/faqs/index.htm) server prepares specialized input for a *WAP* capable hand held device. To preserve accessibility, designers of Web systems should not develop architectures that require (or use) the transmission of documents containing *formatting objects* and properties unless either the transmitter knows that the client can accept formatting objects and properties or the transmitted document contains a reference to the source document(s) used in the construction of the document with the *formatting objects* and properties. <- <-

Formatting

-> Formatting interprets the result tree in its *formatting object tree* form to produce the presentation intended by the designer of the stylesheet from which the XML element and attribute tree in the "fo" namespace was constructed.

The vocabulary of *formatting objects* supported by XSL - the set of `fo:` element types - represents the set of typographic abstractions available to the designer. Semantically, each formatting object represents a specification for a part of the pagination, layout, and styling information that will be applied to the content of that formatting object as a result of formatting the whole result tree. Each formatting object class represents a particular kind of formatting behavior. For example, the block formatting object class represents the breaking of the content of a paragraph into lines. Other parts of the specification may come from other *formatting objects*; for example, the formatting of a paragraph (block formatting object) depends on both the specification of properties on the block formatting object and the specification of the layout structure into which the block is placed by the *formatter*.

The properties associated with an instance of a formatting object control the formatting of that object. Some of the properties, for example "color", directly specify the formatted result. Other properties, for example 'space-before', only constrain the set of possible formatted results without specifying any particular formatted result. The *formatter* may make choices among other possible considerations such as esthetics.

Formatting consists of the generation of a tree of geometric areas, called the *area tree*. The geometric areas are positioned on a sequence of one or more pages (a browser typically uses a single page). Each geometric area has a position on the page, a specification of what to display in that area and may have a background, padding, and borders. For example, formatting a single character generates an area sufficiently large enough to hold the glyph that is used to present the character visually and the glyph is what is displayed in this area. These areas may be nested. For example, the glyph may be positioned within a line, within a block, within a page.

Rendering takes the *area tree*, the abstract model of the presentation (in terms of pages and their collections of areas), and causes a presentation to appear on the relevant medium, such as a browser window on a computer display screen or sheets of paper. The semantics of rendering are not described in detail in this specification.

The first step in formatting is to "objectify" the element and attribute tree obtained via an *XSLT* transformation. Objectifying the tree basically consists of turning the elements in the tree into formatting object nodes and the attributes into property specifications. The result of this step is the *formatting object tree*.

As part of the step of objectifying, the characters that occur in the result tree are replaced by fo:character nodes. Characters in text nodes which consist solely of white space characters and which are children of elements whose corresponding *formatting objects* do not permit fo:character nodes as children are ignored. Other characters within elements whose corresponding *formatting objects* do not permit fo:character nodes as children are errors.

The content of the fo:instream-foreign-object is not objectified; instead the object representing the fo:instream-foreign-object element points to the appropriate node in the element and attribute tree. Similarly any non-XSL namespace child element of fo:declarations

is not objectified; instead the object representing the fo:declarations element points to the appropriate node in the element and attribute tree.

The second phase in formatting is to refine the formatting object tree to produce the *refined formatting object tree*. The *refinement* process handles the mapping from properties to traits. This consists of: (1) shorthand expansion into individual properties, (2) mapping of corresponding properties, (3) determining computed values (may include expression evaluation), (4) handling white-space-treatment and linefeed-treatment property effects, and (5) inheritance. Details on *refinement* are found in "§ 5 - Property *Refinement*/Resolution"

The third step in formatting is the construction of the area tree. The *area tree* is generated as described in the semantics of each formatting object. The traits applicable to each formatting object class control how the areas are generated. Although every formatting property may be specified on every formatting object, for each formatting object class, only a subset of the formatting properties are used to determine the traits for objects of that class.

<-

INDEX