This page is intended for assorted notes documenting the current implementation of FOP (HEAD). The notes are light-weight. That is, they just contain what the author has available. No fancy layout, maybe not complete, may be outdated.

This page is becoming too long. I will make subpages.

- FOPImplementationNotes OooHyphenationPatterns

Property lists

The property lists are held in a tree. FOTreeBuilder holds a reference to the current property list. Each property list holds a reference to its parent. This keeps the current property list and its ancestors accessible and alive. When the FOTreeBuilder changes the current property list to the parent property list, the reference to the current property list is lost, and it is GC’ed. This happens in FOTreeBuilder.endElement.

RetrieveMarker keeps a reference to its property list. In this way it keeps its property list and those of its ancestors alive, so that they can be searched when a marker’s subtree is bound to it.

The property values in a property list are created and they are bound to the FO node at the start of the node, in FObj.processNode in FOTreeBuilder.startElement.

FOTreeBuilder.startElement:

```java
foNode = fobjMaker.make(currentFObj);
propertyList = foNode.createPropertyList(currentPropertyList, foEventHandler);
foNode.processNode(localName, locator, attlist, propertyList);
foNode.startOfNode();
if (propertyList != null) {
    currentPropertyList = propertyList;
}
```

FObj.processNode:

```java
setLocator(locator);
pList.addAttributesToList(attlist);
pList.setWritingMode();
bind(pList);
```

FOTreeBuilder.endElement:

```java
currentFObj.endOfNode();
if (currentPropertyList.getFObj() == currentFObj) {
    currentPropertyList = currentPropertyList.getParentPropertyList();
}
```

StaticPropertyList: explicit property values (explicit) and cached property values (values); both are arrays indexed by the propId.

```java
propertyList = foNode.createPropertyList(currentPropertyList, foEventHandler);
delegates to foEventHandler.getPropertyListMaker().make(this, parent). This returns a StaticPropertyList with fobj and parent PropertyList set.

Markers have a special type of property list, MarkerPropertyList:
explicit property values (explicit): hashmap.

```java
marker.createPropertyList returns a MarkerPropertyList.
```

A marker keeps a reference to the property lists of its descendants, in member descLists, which is a hashmap keyed on the fobj.

Marker.startOfNode changes the property maker of foEventHandler to return a MarkerPropertyList. In Marker.endOfNode the original property maker is restored again. Thus all FO nodes in the subtree of a marker have a MarkerPropertyList.

8 January 2005, SimonPepping

XMLObj
A subtree of XMLObj is a tree of FONodes, where each node has a reference to its parent, a reference to a corresponding DOM element, and a reference to the DOM document node. Only the top FONode is referred to from its parent as a child. As a consequence, all FONodes except the top one are GC'ed as soon as the reference to the deepest child is released.

8 January 2005, SimonPepping

Top-level structure

The interface of FOP to other XML applications is a SAX DefaultHandler interface. It is implemented by its class fo.FOTreeBuilder. Therefore FOP can conveniently be invoked using SAX or TrAX:

```
Transformer transformer = factory.newTransformer();
```

or

```
Transformer transformer = factory.newTransformer(stylesheet);
Source src = new StreamSource(sourcefile); // FO or XML file
Result res = new SAXResult(foTreeBuilder);
transformer.transform(src, res);
```

An FOTreeBuilder object can be created by a call to fop.getDefaultHandler(), where fop is an object of type apps.Fop.

In its constructor the FOTreeBuilder creates an FOEventHandler object. Then it creates a PropertyListMaker object and registers it with the FOEventHandler. Then it sets the default element mappings and additional element mappings of the user. These element mappings map each type of XML element to a node maker, and thus determines the character of the FO node tree and DOM trees for foreign elements that are built.

For each SAX startElement event the FOTreeBuilder instantiates the appropriate maker for the element, using the registered element mappings. The maker creates a node. Then the attributes of the element are converted to property values of the FO node or attributes of the DOM node.

The FO nodes send start and end events to the FOEventHandler. At a suitable point the FOEventHandler starts the layout or structuring process. FOP has three {{FOEventHandler}} types:

- AreaTreeHandler: At the end of the PageSequence event it starts the layout process. It sets up a tree of LayoutManagers. These determine the possible break points of the lines and pages, and create the area tree. When a page is completed, it is handed over to the renderer for rendering.
- RTFHandler and MIFHandler. These two structural handlers set up the structure required for an rtf or mif output file.

FOP's process model is that of a SAX parser. The SAX parser or TrAX transformer drives the process. When the SAX events are received, FOP builds its FO tree. When the FO events are received, the FOEventHandler manages the layout process. When one or more pages are received, the renderer renders the pages. These actions are all done synchronously and in a single thread: The parser process waits until the tree building, layout and rendering processes return.

Plug-ins

After the invocation of the parse or transform methods the FOP process unfolds itself.

There are a few points in the process where the user can plug in an alternative implementation of a part of FOP: FOEventHandler, Renderer, LayoutManagerMaker, additional ElementMappings.

Alternative objects are registered with the user agent of type FOUserAgent, using suitable set methods. A renderer object can be obtained from the render.RendererFactory.newInstance(int renderType).

A LayoutManagerMaker and a Renderer only make sense if one uses AreaTreeHandler as the FOEventHandler.

In principle it is possible to use one's own default ElementMappings and PropertyListMaker, but there are no methods to override FOP's defaults.

A LayoutManagerMaker should satisfy an additional constraint not shown in the LayoutManagerMaker interface. The LayoutManager for PageSequence must be a subclass of PageSequenceLayoutManager, so that it implements the methods setAreaTreeHandler and activateLayout, cf. AreaTreeHandler.endPageSequence.

Markers
LM.addAreas() calls LM.addMarkers(starting, isfirst, islast).

PageViewport holds the marker maps markersFirstStart, markersFirstAny, markersLastStart, markersLastEnd, corresponding to the retrieve-positions first-starting-within-page, first-including-carryover, last-starting-within-page, last-ending-within-page. In addition PageViewport holds the marker map LastAny, from which a marker is retrieved when there is no marker with the requested position last-*within-page. Here first and last refer to the position on the page, starting and ending refer to the traits is-first and is-last of the area to which the marker is associated.

The LM obtains the values of isfirst and islast from the BP. addMarkers is called twice per addAreas call, once at the start with the value starting = true, before any child areas are returned, and once at the end with the value starting = false, after all child areas have been returned. When starting == true, PageViewport.addMarkers adds the markers to the maps markers*Start and markers*Any; when it is false, the markers are added to the maps markers*End and markers*Any.

Retrieval of markers according to the spec:

- **first-starting-within-page**: first qualifying area in the containing page which is-first (FirstStarting); if null, then the first qualifying area in the containing page (FirstAny).
- **first-including-carryover**: first qualifying area in the containing page (FirstAny).
- **last-starting-within-page**: last qualifying area in the containing page which is-first (LastStarting); if null, then the last qualifying area in the containing page (LastAny).
- **last-ending-within-page**: last qualifying area in the containing page which is-last (LastEnding); if null, then the last qualifying area in the containing page (LastAny).

If the containing page does not have a qualifying area, then the last qualifying area of the preceding page is used (LastAny), etc. until a qualifying area is found within the specified retrieve-boundary. (My own interpretation; the spec is not clear about preceding pages.)

Scheme of the values of the parameters of an LM.addMarkers call, and of the addition of markers to the marker maps by PageViewport.addMarkers, for an area and a child area. Note that the start and end of an area always fall on a single page:

```
+-----------------+-----------------+--------------+-----------------+
| !starting is-first is-last | !starting is-first is-last |
| !starting is-first is-last | !starting is-first is-last |
| +-----------------+-----------------+--------------+-----------------+ |
|                               | First?Starting?  | First?Any    |
|                               | LastStarting?   | LastAny      |
|                               | First?Starting?  | First?Any    |
|                               | LastStarting?   | LastAny      |
|                               | // LastAny       |
```

First? means: Only add if this is the first marker of its class.
Starting? and Ending? means: Only add if the area has the trait is-first or is-last.

20 February 2005, SimonPepping