12

Chapter

Labeling and Renditions

This chapter describes how to set a variety of label properties and renditions using the API.

- Labeling Overview
- LabelProperties Class
- Label Code Example
- Rendition Overvie
- Per-Feature Renditions
- Rendition Properties
- Migrating Renditions from 2.x to 3.0

Labeling Overview

Labeling your map is an art form unto itself. Labels are a map element that greatly contribute to the message you want to get across to the viewer. There are many aspects of a label to consider: size, color, style, position, use of creative effects such as haloing and outlining, and its importance in relation to other labels and map features.

These label elements can be controlled through the MapXtreme Java API. Additionally, these properties can also be manipulated via the Label button in the Layer Control dialog in the Map Definition Manager or through the Layer Control Bean. This chapter focuses on the API.

LabelProperties Class

The LabelProperties class contains methods that control how labels are drawn for each layer. With the methods in this class, you can control the content, visibility, appearance, position, and relative importance of labels.

Label Column

The text of the label comes from the an attribute that is associated with the map feature. These two elements are dynamically linked. If the underlying attribute changes, the label text will change as well.

To control which attribute column will be used for the label for a layer, use the **setLabelColumn** method in the LabelProperties class. By default the first attribute in the Feature is used.

For example, to make your map more meaningful to your audience, you might label your school district regions with school-age population instead of the name of the school district.

Label Style

Label style covers a variety of font appearance elements such as font used for the label text, its size, foreground and background color, and special effects.

Any font that is supported by the Java 2D Platform, suchas Type 1or TrueType fonts, can be used for labeling. The Rendition associated with the layer's LabelProperty object (which is retrieved from its getRendition method) is used to control the font, its color and size, the effects for the label, including bold, underline, italic, and background color for box, halo or outline. These are controlled through the properties of the Rendition object.

For example, it's customary to label capital cities of countries larger than labels used for other cities. To draw out the prominence of these capital cities, they might be labeled with a halo effect that makes them stand out from the other surrounding cities.

This example indicates that the label text will be changed to bold, red, italic text:

```
//Change the Rendition

LabelProperties labelProp = layer.getLabelProperties();

Rendition labelRend = labelProp.getRendition();

labelRend.setValue(Rendition.FONT_WEIGHT, 2f);

labelRend.setValue(Rendition.SYMBOL_FOREGROUND,
   Color.red);

labelRend.setValue(Rendition.FONT_STYLE,
   Rendition.FontStyle.ITALIC);

labelProp.setRendition(labelRend);
```

The following code example illustrates how to set the font size:

```
// This example sets a font size of 18 for the layer's
   labels

labelRend.setValue(Rendition.FONT_SIZE, 18);

labelProp.setRendition(labelRend);
```

Label Visibility

The MapJ API includes several ways to control the visibility of your labels: setting a zoom range, whether to allow duplicate and overlapping text, and setting the label priority.

Setting the zoom range for a label is similar to setting the zoom for the layer. You determine at which scale (distance across the map) you want the label to display and set the minimum and maximum values in the setZoomMin and setZoomMax. Zoom settings take a double value that specifies the distance scale in map units.

If you have two features with the same name use **setDuplicationAllowed** method that allows you to label both features. For example, you may have a state boundary called New York and a city boundary called NewYork.

The method **setOverlapAllowed** permits multiple labels in a concentrated area to be visible. The default behavior for overlapped labels is False. Use caution with overlapped labels as a crowded map can be harder to read.

To control the density of labels in an area, you can set a priority level for each layer. The **setOverridePriority** method sets whether to use the default priority or an override value. The override value is set by the **setPriority** method. By default, labels for layers toward the top of the layer list have priority in drawing when setDuplicationAllowed or setOverlapAllowed are setto True.

The **setPriority** method changes the priority of labeling for the layer. The default label priority value is given by the equation: (Number of layers - layer position) * 10. So, if a Layers object contained 20 layers, the default label priority for the layer at position 5 is 150. A layer's default label priority may change when other layers are added or removed from its containing Layers object. Higher values have greater priority. Labels with greater priority will be rendered in the case of overlaps or duplicates.

The following is an example of setting the several LabelProperties methods that affect label visibility:

```
// This example sets zoom labeling from 10 to 30
   kilometers,
// allows overlap, uses the second column to label, and
   increases the label priority for this layer to 200
LabelProperties labelProp = layer.getLabelProperties();
labelProp.setZoomLabel(true);
labelProp.setZoomMin(10.0, LinearUnit.kilometer);
labelProp.setZoomMax(30.0, LinearUnit.kilometer);
labelProp.setOverlapAllowed(true);
labelProp.setOverridePriority(true);
labelProp.setPriority(200);
labelProp.setLabelColumn(1);
layer.setLabelProperties(labelProp);
```

Label Position

The LabelProperties class provides methods for controlling the position of labels, both the alignment to the anchor point and the offset from it.

A Feature's label position is calculated using the following algorithm: the initial position is at the label point returned by the Geometry's **getLabelPoint** method. If the method returns null (which may occur for some features), then a label point is computed which corresponds roughly with the centroid of the region or mid-point of the line. The initial position is then adjusted for alignment and offset.

Labels are horizontally and vertically aligned to the label point. Horizontal alignment can be specified as Left, Center, or Right aligned, or a default alignment is used. VectorGeometries are Center aligned and PointGeometries are Left aligned by default.

Vertical alignment can be set aligned to the Baseline,Top, Bottom, or Center. If the vertical alignment is not specified, a default of Baseline alignment is used.

The second element of label positioning is the offset value. The values of the Offset position are in device units and are with respect to user space. Note that in Java's standard user space the positive y-axis is beneath the x-axis (positive y goes down). If the offset is unspecified, a default value is used. The default offset for regions is (0, 0); the default for lines takes into account the line width and is (0, -w/2) where w is the width. The default for points takes into account the point's symbol size, and is (s/2 + 2, -s/2 + 2) where s is the symbol size.

A final element to control label position for a line is whether the label will follow the slope of the line, its default behavior. It can be overridden by using the **setLineLabelHorizontal** method.

This example uses alignment and offset properties.

```
// change the horizontal and vertical alignments
// and offset
LabelProperties labelProp = layer.getLabelProperties();
labelProp.setHorizontalAlignment(LabelProperties.HORIZ_AL
   IGN_RIGHT);
labelProp.setVerticalAlignment(LabelProperties.VERT_ALIGN
   _TOP);
labelProp.setOffset(new DoublePoint(10, -15));
layer.setLabelProperties(labelProp);
```

Label Code Example

This section provides a code example for changing the label style. The code can also be found in the \codesamples directory of MapXtreme Java.

```
// specify the url to the MapXtreme servlet which
   remotely connects to the map engine

String MapXtremeURL =
   "http:\\localhost:8080\mapxtreme\servlet\mapxtreme";

// set property to display labels

layer.setAutoLabel(true);

// Retrieve the LabelProperties from the layer and then
   assign the Rendition to our rend instance

LabelProperties labelProp = layer.getLabelProperties();

Rendition rend = labelProp.getRendition();

// Set the new rendition values

rend.setValue(Rendition.SYMBOL_FOREGROUND, Color.green);

rend.setValue(Rendition.SYMBOL_BACKGROUND, Color.blue);

rend.setValue(Rendition.FILTER_EFFECTS,
   Rendition.FilterEffects.HALO);

labelProp.setRendition(rend);

layer.setLabelProperties(labelProp);

// Render the map to see the labels

MapXtremeImageRenderer rr = new
   MapXtremeImageRenderer(MapXtremeURL);

map.render(rr);

rr.toFile(c:\\temp\\file.gif);
```

Rendition Overview

A Rendition object is a sparse collection of display properties, each of which controls one aspect of how a map feature or label will be displayed on the map. It is a sparse collection in that you (the user) only need to set those display properties that matter, the rest will come from combining or merging with other renditions. This is how Themes work in MapXtreme Java. A Theme contains a Rendition that changes just one or two aspects of a map feature (such as the region fill color), which leaves all of the other properties (like the region's edge color, width, etc.) alone.

As of MapXtreme Java version 3.0, many new types of display properties have been added to take better advantage of all of the rendering capabilities of the Java2D API. These include symbol paint for lines and regions, dashed and parallel lines, vector symbols, and more.

Also with version 3.0, the RDBM data sources (Oracle8*i*, Informix, DB2, etc.) have been enhanced to allow for per feature (record) renditions. This allows for finer control over the display of features coming out of RDBM data sources, thus allowing them to look more like the maps that MapInfo users have come to expect from TAB files.

This section covers per-feature renditions and rendition properties.

Per-Feature Renditions

MapXtreme Java allows you to specify a table level rendition (returned by the FeatureSet's getRendition method) in the MAPINFO_MAPCATALOG[1]. This table level rendition is used as the base set of display properties for all features from that table. As of MapXtreme Java 3.0, an additional column can be specified within your spatial table that will be used as a per-feature rendition (returned by the Feature's getRendition method). This per-feature rendition is merged (overrides) with the table (FeatureSet) rendition to determine the set of properties used to display that feature.

The rendition column is determined either as a parameter at layer creation time (either programatically or via the LayerControl Bean) or from the MAPINFO_MAPCATALOG. This rendition column within the spatial table can either be Null, a MapBasic style string, or a MapXtreme Java rendition.

1. See Appendix C for more information.

Rendition Properties

Rendition properties are used to describe how to display a map feature. The Rendition API supports three categories of properties: fill, stroke, and symbol.

The fill properties control how a region is filled. The stroke properties control how a line (either a line geometry or the edge of a region) will be drawn. The symbol properties control how symbols are drawn for either point geometries, line markers, or symbol fills.

Fill Properties

Regions can be filled with either a solid color or a symbol.To set a color, you specify a Java Color object that represents the color you want. For example:

```
Rendition rend = new Rendition();
rend.setValue(Rendition.FILL, Color.red);
```

Fills can also be defined by using a symbol for the fill paint. Symbols can be any of the supported symbol types (font, image, or vector). Symbol paint can fill a region or a wide line (think of wide lines as a polygon STROKE_WIDTH units in diameter, filled with the STROKE paint). The Symbol is used to create a 'tile' that is repeatedly drawn to fill the region.

Think of tile filling this way. Imagine a ceramic tile floor where each tile has a pattern on it that matches the symbol rendition specified for the region fill. The region is like a hole cut through a sheet of paper that is laid onto the tile floor. The tile pattern will show through the hole. If multiple features use the same symbol paint, the symbol tile pattern will match or 'line up' such that it will appear as though all of the features were drawn at the same time.

An example of symbol paint would be in specifying a swampy region. A 'swamp' symbol is created (e.g., a GIF file that represents swamp grass). This 'swamp grass' symbol will be used to tile fill the region, giving the appearance that the region is filled with swamp grass.

```
Rendition rendSymbol = new Rendition();

rendSymbol.setValue(Rendition.SYMBOL_MODE,
   Rendition.SymbolMode.IMAGE);

rendSymbol.setValue(Rendition.SYMBOL_URL, "http://
   www.myhost.com/image/swamp.gif");

Rendition rendFill = new Rendition();

rendFill.setValue(Rendition.FILL, rendSymbol);
```

With both solid and symbol paints you can also control the opacity of the paint using the Rendition.FILL_OPACITY property.Values range from 0.0 for a completely transparent fill to 1.0 for a completely opaque fill.

```
Rendition rend = new Rendition();

rend.setValue(Rendition.FILL, Color.blue);

rend.setValue(Rendition.FILL_OPACITY, 0.5f);
```

Stroke Properties

Stroke properties control how a line or region edge will be displayed. The various properties control the stroke paint, line width, line segment joins and caps, dashing patterns, and more.

The Rendition.STROKE property controls the paint used when drawing the line or edge. The property value can be either a color, which will set the RGB for the solid color line, or a Rendition used to specify the symbol paint. This is very similar to the Rendition.FILL property described above. When the value is a Rendition, like the FILL property, the symbol properties from that Rendition are used to create a symbol that will be used to tile fill the line. Generally, symbol fills on lines only make sense when the STROKE_WIDTH >> 1.

The Rendition.STROKE_WIDTH controls the width of the line (in points) and Rendition.STROKE_OPACITY, the opacity (0.0 for transparent to 1.0 for opaque).

```
Rendition rend = new Rendition();
rend.setValue(Rendition.STROKE, Color.red);
rend.setValue(Rendition.STROKE_WIDTH, 3.2f);
rend.setValue(Rendition.STROKE_OPACITY, 0.3f);
```

Parallel Lines

The Rendering engine now supports the ability to render one or more lines parallel to the base line. The Rendition.STROKE_PARALLELARRAY property can contain an array of one or more Rendition.ParallelLine objects. Each Rendition.ParallelLine object contains the offset and rendition of a line that is to be drawn parallel to the base line.

The offset tells the rendering engine how many units to draw the line from the base line. Currently all units are specified in printer's points (1/72 inch). The offset can be any number where 0 means draw the parallel line on top of the base line (no offset), +N means draw the parallel line N units to the right of the base line, and -N means draw the parallel line N units to the left. Right and left are determined based on the direction of the base line. If the first point of a line starts on the left side of the screen and the next point is to the right of that point, then an +N offset would have the parallel line below or to the right of the base line's direction of travel.

Each parallel line has a separate Rendition to specify how it should be drawn.

Parallel lines are always drawn after the base line is rendered.

The classic example of parallel lines is that of railroad tracks. Here you have a transparent base line with two parallel lines, one on each side the same distance apart.

```
Rendition rendParallel = new Rendition();

rendParallel.setValue(Rendition.STROKE, Color.black);

Rendition.ParallelLine parallel1 = new
   Rendition.ParallelLine(3, rendParallel);

Rendition.ParallelLine parallel2 = new
   Rendition.ParallelLine(-3, rendParallel);

Rendition.ParallelLine[] parallelArray = {parallel1,
   parallel2};

Rendition rendBaseLine = new Rendition();

rendBaseLine.setValue(Rendition.STROKE_OPACITY, 0f);

rendBaseLine.setValue(Rendition.STROKE_PARALLELARRAY,
   parallelArray);
```

Dashed lines

Dashed lines are defined by an array of numbers (float[]), specified in pairs. Each pair specifies the length of the dash and length of space till the next dash. For example a dashed line with values {5,3} will have 5 units of line and 3 units of space in between. The STROKE_DASHARRAY property can be used with any type of line or edge.

The STROKE_DASHOFFSET property controls how many units into the dash array to start the dashing pattern. For example, assume a STROKE_DASHARRAY property value of {5, 3}. If the STROKE_DASHOFFSET is not set or set to 0, than the dashing pattern will start with 5 units of dash followed by 3 units of space. If the STROKE_DASHOFFSET is set to 3, then the line would have 2 units of line followed by 3 units of space and then 5 units of line followed by 3 units of space, etc.

An example of dashed lines is to show roads under construction or proposed underground cables.

```
Rendition rend = new Rendition();

rend.setValue(Rendition.STROKE_DASHARRAY, new float[]{5,
   3});

rend.setValue(Rendition.STROKE_OFFSET, 3.2f);
```

Line Markers

Stroke markers are similar to Symbol paint, except that the symbol is rotated to match the angle of the line. Use line markers to mark the path of a line with repeating symbols.

For example, use Rendition.STROKE_MARKERARRAY and an image of a car to render the effect of rush hour traffic with one car following another down a road. In this case you would set the Rendition.SYMBOL_MODE to Rendition.SymbolMode.IMAGE and point the Rendition.SYMBOL_URL to the URL of a GIF file of a transparent car image. (Symbols are explained below). Then the Rendition.STROKE_MARKERARRAY of a line geometry is set to the symbol rendition. When rendered, the line geometry will be drawn based on its other STROKE properties and then the car image will be rotated and repeatedly drawn along the path of each line segment (a line segment is the path between any two points of a line geometry).

```
Rendition rendSymbol = new Rendition();

rendSymbol.setValue(Rendition.SYMBOL_MODE,
   Rendition.SymbolMode.IMAGE);

rendSymbol.setValue(Rendition.SYMBOL_URL, "http://
   www.myhost.com/image/car.gif");

Rendition.Marker marker = new
   Rendition.Marker(rendSymbol);

Rendition rendLine = new Rendition();

rendLine.setValue(Rendition.STROKE_MARKERARRAY, new
   Rendition.Marker[]{marker});
```

Line Caps, Joins

The Rendition API provides a variety of ways to finish the ends of lines and to join lines together.

Use Rendition.LineCap with a STROKE_LINECAP property to complete lines with round or square endcap decorations, or no decoration (use round, square, or butt properties, respectively).

Similarly, the Rendition.STROKE_LINEJOIN property has three ways to connect line segments: connect outer corners (bevel), extend outer edges to connect (miter), and round off the corner (round).

Symbol Properties: Font, Image and Vector

Symbols in MapXtreme Java can do a lot more than just mark a point location. As mentioned above, symbols can be used as the Rendition to fill regions, wide lines, or line markers. Symbols are divided into three types: font, image, and vector.

Font Symbols

Any font that is supported by the Java2 platform, such as Type1 or TrueType, can be used as a symbol. MapXtreme provides a number of TrueType symbol sets that ar typically used in mapping, including:

- MapInfo Cartographic
- MapInfo Transportation
- MapInfo Real Estate
- MapInfo Miscellaneous
- MapInfo Oil & Gas
- MapInfo Weather
- MapInfo Arrows
- MapInfo Shields
- MapInfo Symbols
- Map Symbols

These fonts are located in \server\fonts directory after installation. You must register these fonts with your operating system in order to use them in MapXtreme Java Edition. Note that the MapInfo Symbols and Map Symbols fonts may show the name as MapInfo S_ymbols and Map S_ymbols. To access the fonts programmatically use the common name (e.g., MapInfo Arrows, MapInfo Cartographic, MapInfo Symbols).

To view the fonts, use your operating system's font viewer tool, such as CharMap on Windows.

When the Rendition.SYMBOL_MODE property is set to Rendition.SymbolMode.FONT, the font properties (i.e. Rendition.FONT_FAMILY) ar used with the Rendition.SYMBOL_STRING property to specify a font symbol. When using a font symbol, you can choose the font family, size, background and foreground color, and creative effects such as bold, italic, underline, halo, box, and outline.

```
Rendition rend = new Rendition();

rend.setValue(Rendition.SYMBOL_MODE,
   Rendition.SymbolMode.FONT);

rend.setValue(Rendition.FONT_FAMILY, "MapInfo
   Cartographic");

rend.setValue(Rendition.FONT_SIZE, 12);

rend.setValue(Rendition.SYMBOL_STRING,
   String.valueOf((char)33));
```

Image Symbols

A symbol can also be represented by an image (GIF, JPEG, PNG, etc.). When the Rendition.SYMBOL_MODE property is set to Rendition.SymbolMode.IMAGE, the Rendition.SYMBOL_URL property is used to retrieve an image from the specified URL. The Rendition.SYMBOL_URL property contains a URL (i.e. http://myhost.com/image/truck.gif).

For example, refer again to the example above of the repeating transparent car image. This example used Rendition.STROKE_MARKERARRAY to specify that a symbol of a car be repeated along the line.

```
Rendition rend = new Rendition();

rend.setValue(Rendition.SYMBOL_MODE,
   Rendition.SymbolMode.IMAGE);

rend.setValue(Rendition.SYMBOL_URL, "http://myhost.com/
   image/car.gif");
```

MapXtreme Java provides a set of custom symbol GIF images that you can use to mark your map. See Appendix E for descriptions.

Vector Symbols

If fonts and image symbols do not provide you with the symbology you need, you can draw your own. MapXtreme Java now supports vector symbols, that is any shape that can be specified by the Java2D Shape interface.

When the Rendition.SYMBOL_MODE property is set to Rendition.SymbolMode.SHAPE, the Rendition.SYMBOL_SHAPE property will be used to create the symbol. The Rendition.SYMBOL_SHAPE property contains a Rendition.SymbolShape object. This object consists of a Rendition object and an object that implements the java.awt.Shape interface.

For example, many of the objects in the java.awt.geom package (like Rectangle2D, Polygon, etc.) implement the Shape interface. You could also use the java.awt.geom.GeneralPath object to specify more complex geometries using commands like moveTo, lineTo, etc. The Rendition.SymbolShape's Rendition object is used when displaying the Shape. That is the Rendition might specify the FILL paint to use for a region shape.

```
Rendition rendShape = new Rendition();

rendShape.setValue(Rendition.FILL, Color.red);

Rectangle2D rect = new Rectangle2D.Float(0, 0, 10, 10);

Rendition.SymbolShape symbolShape = new
   Rendition.SymbolShape(shape, rend);

Rendition rend = new Rendition();

rend.setValue(Rendition.SYMBOL_MODE,
   Rendition.SymbolMode.SHAPE);

rend.setValue(Rendition.SYMBOL_SHAPE, symbolShape);
```

Migrating Renditions from 2.x to 3.0

Due to the improvements to the Rendition engine, new style properties have been created and some existing ones were changed.

The old style mappings, for the most part, will continue to behave as before, some of which have been renamed. Those that will no longer work with MapXtreme Java 3.0 include any of the XOR and MULTI_STYLE properties (e.g. LINE_XOR, EDGE_XOR, LINE_MULTI_STYLE, etc.). Both LINE and EDGE properties are now covered by STROKE.

If you previously depended on the line and edge properties to be separate within any given instance of a Rendition object (e.g. used settings for both LINE_COLOR and EDGE_COLOR in a single Rendition) please note that now the last one set will be the one used for both line geometries and the edge of polygons (e.g., LINE_COLOR and then EDGE_COLOR; EDGE_COLOR will be used).We strongly recommend that you convert your application to use the newer, non-deprecated Rendition properties.

When switching to the new style properties you need to be careful in several areas.

FILL_COLOR has been replaced by FILL, and LINE_COLOR and EDGE_COLOR have been replaced by STROKE. FILL and STROKE can be either a Color or a Rendition. If you just replace existing code that calls <Rendition instance>.setValue(XXX_COLOR, color) with <Rendition instance>.setValue(FILL, color), everything will work as expected. However, make sure that any existing code that accesses these properties (e.g., <Rendition instance>.getValue(FILL)) does not assume that the return value will be a Color. For example:

existing code: **`Color lineColor = rend.getColor(Rendition.LINE_COLOR)`**

new code: **`Object stroke = rend.getValue(Rendition.STROKE);`**

Note the two changes: the getValue method is used instead of the getColor method and the return value type is Object versus Color. The return from getValue(STROKE) can be tested if it is either a Color or Rendition by:

```
if (stroke instanceof Color) {
Color c = (Color)stroke;
} else {
Rendition r = (Rendition)stroke;
}
```

The Rendition.SYMBOL_CHAR property is now Rendition.SYMBOL_STRING. Font symbols are now represented by a string. Be sure to change the data type as well as the property name. For example, you might have existing code like this.

```
rend.setValue(Rendition.SYMBOL_CHAR, 45);
```

You will need to change this to

```
rend.setValue(Rendition.SYMBOL_STRING,
  String.valueOf((char)45));
```

to get the same behavior. Note: the Rendition.SYMBOL_STRING property supports more than one character symbol, so it can be used to display full strings of text and / or symbols at a point location.

XXX_XOR, XXX_MULTI_STYLE and SECONDARY_RENDITION properties still exist, but will be ignored. There is currently no replacement for these classes.

The table below lists old and new style properties for Rendition. Note: This list is not the complete list of available properties, only those that have changed in this release. .

2.x Style Properties	3.0 Style Properties
LINE_COLOR (Color)	STROKE (Color | Rendition)
LINE_XOR	Ignored
LINE_XOR_COLOR	Ignored
LINE_WIDTH (Integer)	STROKE_WIDTH (Number)
LINE_MULTI_STYLE	Ignored
FILL_COLOR (Color)	FILL (Color | Rendition)
FILL_XOR	Ignored
FILL_XOR_COLOR	Ignored

FILL_TRANSPARENT (Boolean)
- TRUE
- FALSE
	FILL (Color | Rendition)
	new Color(R, G, B, 0.0);
	new Color(R, G, B, 1.0);

EDGE_COLOR (Color)	STROKE (Color | Rendition)
EDGE_XOR	Ignored
EDGE_XOR_COLOR	Ignored
EDGE_WIDTH (Integer)	STROKE_WIDTH (Number)
EDGE_MULTI_STYLE	Ignored

2.x Style Properties	3.0 Style Properties
SYMBOL_TYPE (Integer) - SYMBOL_TYPE_FONT - SYMBOL_TYPE_CUSTOM	SYMBOL_MODE (SymbolMode) SymbolMode.FONT SymbolMode.IMAGE
SYMBOL_FONT_NAME (String)	FONT_FAMILY (String)
SYMBOL_CHAR (Integer; ASCII number)	SYMBOL_STRING (String)
SYMBOL_SIZE (Integer)	FONT_SIZE (Number)
SYMBOL_COLOR (Color)	FILL (Color | Rendition)
SYMBOL_XOR	Ignored
SYMBOL_XOR_COLOR	Ignored
TEXT_FONT_HINT	Ignored (always TEXT_FONT_HINT_JDK12)
TEXT_FONT_NAME (String)	FONT_FAMILY (String)
TEXT_SIZE (Integer)	FONT_SIZE (Number)
TEXT_FORE_COLOR (Color)	SYMBOL_FOREGROUND (Color | Rendition)
TEXT_BACK_COLOR (Color)	SYMBOL_BACKGROUND (Color | Rendition)
TEXT_BOLD (Boolean) - TRUE - FALSE	FONT_WEIGHT (Number) new Float(2.0) new Float(1.0)
TEXT_ITALIC (Boolean) - TRUE - FALSE	FONT_STYLE (FontStyle) FontStyle.ITALIC FontStyle.NORMAL
TEXT_UNDERLINE (Boolean) - TRUE - FALSE	TEXT_DECORATIONS (Number) new Integer(TextDecorations.UNDERLINE) new Integer(TextDecorations.NONE)
TEXT_HALO (Boolean) - TRUE - FALSE	FILTER_EFFECTS (FilterEffects.HALO) FilterEffects.HALO FilterEffects.NONE
SECONDARY_RENDITION	Ignored
get/setOverrideColor()	FILL
get/setOverrideSize()	FONT_SIZE
get/setOverrideWidth()	STROKE_WIDTH