```
001 /*
002  * $Id: LineAndShapeGanttRenderer.java,v 1.3 2006/10/31 18:39:59 sm-santos Exp $
003  *
004  * Last changed on : $Date: 2006/10/31 18:39:59 $
005  * Last changed by : $Author: sm-santos $
006  */
007 package org.jfree.chart.renderer.category;
008
009 import java.awt.Graphics2D;
010 import java.awt.Paint;
011 import java.awt.Shape;
012 import java.awt.Stroke;
013 import java.awt.geom.Line2D;
014 import java.awt.geom.Rectangle2D;
015 import java.io.Serializable;
016
017 import org.jfree.chart.LegendItem;
018 import org.jfree.chart.axis.CategoryAxis;
019 import org.jfree.chart.axis.ValueAxis;
020 import org.jfree.chart.entity.EntityCollection;
021 import org.jfree.chart.event.RendererChangeEvent;
022 import org.jfree.chart.plot.CategoryPlot;
023 import org.jfree.chart.plot.PlotOrientation;
024 import org.jfree.data.category.CategoryDataset;
025 import org.jfree.util.BooleanList;
026 import org.jfree.util.BooleanUtilities;
027 import org.jfree.util.ObjectUtilities;
028 import org.jfree.util.PublicCloneable;
029 import org.jfree.util.ShapeUtilities;
030
031 /**
032  * A renderer that draws shapes for each data item, and lines between data
033  * items (for use with the {@link CategoryPlot} class).
034  *
035  * @author : samaxes
036  * @version : $Revision: 1.3 $
037  */
038 public class LineAndShapeGanttRenderer extends AbstractCategoryItemRenderer
039                                        implements Cloneable, PublicCloneable,
040                                                   Serializable {
041
042     /** For serialization. */
043     private static final long serialVersionUID = -1977495198692263 98L;
044
045     /** A flag that controls whether or not lines are visible for ALL series. */
046     private Boolean linesVisible;
047
048     /**
049      * A table of flags that control (per series) whether or not lines are
050      * visible.
051      */
052     private BooleanList seriesLinesVisible;
053
054     /**
055      * A flag indicating whether or not lines are drawn between non-null
056      * points.
057      */
058     private boolean baseLinesVisible;
059
060     /**
061      * A flag that controls whether or not shapes are visible for ALL series.
062      */
063     private Boolean shapesVisible;
064
065     /**
066      * A table of flags that control (per series) whether or not shapes are
067      * visible.
068      */
069     private BooleanList seriesShapesVisible;
```

```
070
071     /** The default value returned by the getShapeVisible() method. */
072     private boolean baseShapesVisible;
073
074     /** A flag that controls whether or not shapes are filled for ALL series. */
075     private Boolean shapesFilled;
076
077     /**
078      * A table of flags that control (per series) whether or not shapes are
079      * filled.
080      */
081     private BooleanList seriesShapesFilled;
082
083     /** The default value returned by the getShapeFilled() method. */
084     private boolean baseShapesFilled;
085
086     /**
087      * A flag that controls whether the fill paint is used for filling
088      * shapes.
089      */
090     private boolean useFillPaint;
091
092     /** A flag that controls whether outlines are drawn for shapes. */
093     private boolean drawOutlines;
094
095     /**
096      * A flag that controls whether the outline paint is used for drawing shape
097      * outlines - if not, the regular series paint is used.
098      */
099     private boolean useOutlinePaint;
100
101     /**
102      * Creates a renderer with both lines and shapes visible by default.
103      */
104     public LineAndShapeGanttRenderer() {
105         this(true, true);
106     }
107
108     /**
109      * Creates a new renderer with lines and/or shapes visible.
110      *
111      * @param lines  draw lines?
112      * @param shapes  draw shapes?
113      */
114     public LineAndShapeGanttRenderer(boolean lines, boolean shapes) {
115         super();
116         this.linesVisible = null;
117         this.seriesLinesVisible = new BooleanList();
118         this.baseLinesVisible = lines;
119         this.shapesVisible = null;
120         this.seriesShapesVisible = new BooleanList();
121         this.baseShapesVisible = shapes;
122         this.shapesFilled = null;
123         this.seriesShapesFilled = new BooleanList();
124         this.baseShapesFilled = true;
125         this.useFillPaint = false;
126         this.drawOutlines = true;
127         this.useOutlinePaint = false;
128     }
129
130     // LINES VISIBLE
131
132     /**
133      * Returns the flag used to control whether or not the line for an item is
134      * visible.
135      *
136      * @param series  the series index (zero-based).
137      * @param item  the item index (zero-based).
138      *
```

```
     * @return A boolean.
     */
    public boolean getItemLineVisible(int series, int item) {
        Boolean flag = this.linesVisible;
        if (flag == null) {
            flag = getSeriesLinesVisible(series);
        }
        if (flag != null) {
            return flag.booleanValue();
        }
        else {
            return this.baseLinesVisible;
        }
    }

    /**
     * Returns a flag that controls whether or not lines are drawn for ALL
     * series.  If this flag is <code>null</code>, then the "per series"
     * settings will apply.
     *
     * @return A flag (possibly <code>null</code>).
     */
    public Boolean getLinesVisible() {
        return this.linesVisible;
    }

    /**
     * Sets a flag that controls whether or not lines are drawn between the
     * items in ALL series, and sends a {@link RendererChangeEvent} to all
     * registered listeners.  You need to set this to <code>null</code> if you
     * want the "per series" settings to apply.
     *
     * @param visible  the flag (<code>null</code> permitted).
     */
    public void setLinesVisible(Boolean visible) {
        this.linesVisible = visible;
        notifyListeners(new RendererChangeEvent(this));
    }

    /**
     * Sets a flag that controls whether or not lines are drawn between the
     * items in ALL series, and sends a {@link RendererChangeEvent} to all
     * registered listeners.
     *
     * @param visible  the flag.
     */
    public void setLinesVisible(boolean visible) {
        setLinesVisible(BooleanUtilities.valueOf(visible));
    }

    /**
     * Returns the flag used to control whether or not the lines for a series
     * are visible.
     *
     * @param series  the series index (zero-based).
     *
     * @return The flag (possibly <code>null</code>).
     */
    public Boolean getSeriesLinesVisible(int series) {
        return this.seriesLinesVisible.getBoolean(series);
    }

    /**
     * Sets the 'lines visible' flag for a series.
     *
     * @param series  the series index (zero-based).
     * @param flag  the flag (<code>null</code> permitted).
     */
    public void setSeriesLinesVisible(int series, Boolean flag) {
```

```
208         this.seriesLinesVisible.setBoolean(series, flag);
209         notifyListeners(new RendererChangeEvent(this));
210     }
211
212     /**
213      * Sets the 'lines visible' flag for a series.
214      *
215      * @param series  the series index (zero-based).
216      * @param visible  the flag.
217      */
218     public void setSeriesLinesVisible(int series, boolean visible) {
219         setSeriesLinesVisible(series, BooleanUtilities.valueOf(visible));
220     }
221
222     /**
223      * Returns the base 'lines visible' attribute.
224      *
225      * @return The base flag.
226      */
227     public boolean getBaseLinesVisible() {
228         return this.baseLinesVisible;
229     }
230
231     /**
232      * Sets the base 'lines visible' flag.
233      *
234      * @param flag  the flag.
235      */
236     public void setBaseLinesVisible(boolean flag) {
237         this.baseLinesVisible = flag;
238         notifyListeners(new RendererChangeEvent(this));
239     }
240
241     // SHAPES VISIBLE
242
243     /**
244      * Returns the flag used to control whether or not the shape for an item is
245      * visible.
246      *
247      * @param series  the series index (zero-based).
248      * @param item  the item index (zero-based).
249      *
250      * @return A boolean.
251      */
252     public boolean getItemShapeVisible(int series, int item) {
253         Boolean flag = this.shapesVisible;
254         if (flag == null) {
255             flag = getSeriesShapesVisible(series);
256         }
257         if (flag != null) {
258             return flag.booleanValue();
259         }
260         else {
261             return this.baseShapesVisible;
262         }
263     }
264
265     /**
266      * Returns the flag that controls whether the shapes are visible for the
267      * items in ALL series.
268      *
269      * @return The flag (possibly <code>null</code>).
270      */
271     public Boolean getShapesVisible() {
272         return this.shapesVisible;
273     }
274
275     /**
276      * Sets the 'shapes visible' for ALL series and sends a
```

```
277     * {@link RendererChangeEvent} to all registered listeners.
278     *
279     * @param visible  the flag (<code>null</code> permitted).
280     */
281    public void setShapesVisible(Boolean visible) {
282        this.shapesVisible = visible;
283        notifyListeners(new RendererChangeEvent(this));
284    }
285
286    /**
287     * Sets the 'shapes visible' for ALL series and sends a
288     * {@link RendererChangeEvent} to all registered listeners.
289     *
290     * @param visible  the flag.
291     */
292    public void setShapesVisible(boolean visible) {
293        setShapesVisible(BooleanUtilities.valueOf(visible));
294    }
295
296    /**
297     * Returns the flag used to control whether or not the shapes for a series
298     * are visible.
299     *
300     * @param series  the series index (zero-based).
301     *
302     * @return A boolean.
303     */
304    public Boolean getSeriesShapesVisible(int series) {
305        return this.seriesShapesVisible.getBoolean(series);
306    }
307
308    /**
309     * Sets the 'shapes visible' flag for a series and sends a
310     * {@link RendererChangeEvent} to all registered listeners.
311     *
312     * @param series  the series index (zero-based).
313     * @param visible  the flag.
314     */
315    public void setSeriesShapesVisible(int series, boolean visible) {
316        setSeriesShapesVisible(series, BooleanUtilities.valueOf(visible));
317    }
318
319    /**
320     * Sets the 'shapes visible' flag for a series and sends a
321     * {@link RendererChangeEvent} to all registered listeners.
322     *
323     * @param series  the series index (zero-based).
324     * @param flag  the flag.
325     */
326    public void setSeriesShapesVisible(int series, Boolean flag) {
327        this.seriesShapesVisible.setBoolean(series, flag);
328        notifyListeners(new RendererChangeEvent(this));
329    }
330
331    /**
332     * Returns the base 'shape visible' attribute.
333     *
334     * @return The base flag.
335     */
336    public boolean getBaseShapesVisible() {
337        return this.baseShapesVisible;
338    }
339
340    /**
341     * Sets the base 'shapes visible' flag.
342     *
343     * @param flag  the flag.
344     */
345    public void setBaseShapesVisible(boolean flag) {
```

```
346         this.baseShapesVisible = flag;
347         notifyListeners(new RendererChangeEvent(this));
348     }
349
350     /**
351      * Returns <code>true</code> if outlines should be drawn for shapes, and
352      * <code>false</code> otherwise.
353      *
354      * @return A boolean.
355      */
356     public boolean getDrawOutlines() {
357         return this.drawOutlines;
358     }
359
360     /**
361      * Sets the flag that controls whether outlines are drawn for
362      * shapes, and sends a {@link RendererChangeEvent} to all registered
363      * listeners.
364      * <P>
365      * In some cases, shapes look better if they do NOT have an outline, but
366      * this flag allows you to set your own preference.
367      *
368      * @param flag  the flag.
369      */
370     public void setDrawOutlines(boolean flag) {
371         this.drawOutlines = flag;
372         notifyListeners(new RendererChangeEvent(this));
373     }
374
375     /**
376      * Returns the flag that controls whether the outline paint is used for
377      * shape outlines.  If not, the regular series paint is used.
378      *
379      * @return A boolean.
380      */
381     public boolean getUseOutlinePaint() {
382         return this.useOutlinePaint;
383     }
384
385     /**
386      * Sets the flag that controls whether the outline paint is used for shape
387      * outlines.
388      *
389      * @param use  the flag.
390      */
391     public void setUseOutlinePaint(boolean use) {
392         this.useOutlinePaint = use;
393     }
394
395     // SHAPES FILLED
396
397     /**
398      * Returns the flag used to control whether or not the shape for an item
399      * is filled. The default implementation passes control to the
400      * <code>getSeriesShapesFilled</code> method. You can override this method
401      * if you require different behaviour.
402      *
403      * @param series  the series index (zero-based).
404      * @param item  the item index (zero-based).
405      *
406      * @return A boolean.
407      */
408     public boolean getItemShapeFilled(int series, int item) {
409         return getSeriesShapesFilled(series);
410     }
411
412     /**
413      * Returns the flag used to control whether or not the shapes for a series
414      * are filled.
```

```
415         *
416         * @param series  the series index (zero-based).
417         *
418         * @return A boolean.
419         */
420        public boolean getSeriesShapesFilled(int series) {
421
422            // return the overall setting, if there is one...
423            if (this.shapesFilled != null) {
424                return this.shapesFilled.booleanValue();
425            }
426
427            // otherwise look up the paint table
428            Boolean flag = this.seriesShapesFilled.getBoolean(series);
429            if (flag != null) {
430                return flag.booleanValue();
431            }
432            else {
433                return this.baseShapesFilled;
434            }
435
436        }
437
438        /**
439         * Returns the flag that controls whether or not shapes are filled for
440         * ALL series.
441         *
442         * @return A Boolean.
443         */
444        public Boolean getShapesFilled() {
445            return this.shapesFilled;
446        }
447
448        /**
449         * Sets the 'shapes filled' for ALL series.
450         *
451         * @param filled  the flag.
452         */
453        public void setShapesFilled(boolean filled) {
454            if (filled) {
455                setShapesFilled(Boolean.TRUE);
456            }
457            else {
458                setShapesFilled(Boolean.FALSE);
459            }
460        }
461
462        /**
463         * Sets the 'shapes filled' for ALL series.
464         *
465         * @param filled  the flag (<code>null</code> permitted).
466         */
467        public void setShapesFilled(Boolean filled) {
468            this.shapesFilled = filled;
469        }
470
471        /**
472         * Sets the 'shapes filled' flag for a series.
473         *
474         * @param series  the series index (zero-based).
475         * @param filled  the flag.
476         */
477        public void setSeriesShapesFilled(int series, Boolean filled) {
478            this.seriesShapesFilled.setBoolean(series, filled);
479        }
480
481        /**
482         * Sets the 'shapes filled' flag for a series.
483         *
```

```
484      * @param series  the series index (zero-based).
485      * @param filled  the flag.
486      */
487     public void setSeriesShapesFilled(int series, boolean filled) {
488         this.seriesShapesFilled.setBoolean(
489             series, BooleanUtilities.valueOf(filled)
490         );
491     }
492
493     /**
494      * Returns the base 'shape filled' attribute.
495      *
496      * @return The base flag.
497      */
498     public boolean getBaseShapesFilled() {
499         return this.baseShapesFilled;
500     }
501
502     /**
503      * Sets the base 'shapes filled' flag.
504      *
505      * @param flag  the flag.
506      */
507     public void setBaseShapesFilled(boolean flag) {
508         this.baseShapesFilled = flag;
509     }
510
511     /**
512      * Returns <code>true</code> if the renderer should use the fill paint
513      * setting to fill shapes, and <code>false</code> if it should just
514      * use the regular paint.
515      *
516      * @return A boolean.
517      */
518     public boolean getUseFillPaint() {
519         return this.useFillPaint;
520     }
521
522     /**
523      * Sets the flag that controls whether the fill paint is used to fill
524      * shapes, and sends a {@link RendererChangeEvent} to all
525      * registered listeners.
526      *
527      * @param flag  the flag.
528      */
529     public void setUseFillPaint(boolean flag) {
530         this.useFillPaint = flag;
531         notifyListeners(new RendererChangeEvent(this));
532     }
533
534     /**
535      * Returns a legend item for a series.
536      *
537      * @param datasetIndex  the dataset index (zero-based).
538      * @param series  the series index (zero-based).
539      *
540      * @return The legend item.
541      */
542     public LegendItem getLegendItem(int datasetIndex, int series) {
543
544         CategoryPlot cp = getPlot();
545         if (cp == null) {
546             return null;
547         }
548
549         if (isSeriesVisible(series) && isSeriesVisibleInLegend(series)) {
550             CategoryDataset dataset;
551             dataset = cp.getDataset(datasetIndex);
552             String label = getLegendItemLabelGenerator().generateLabel(
```


```
553                 dataset, series
554             );
555             String toolTipText = null;
556             if (getLegendItemToolTipGenerator() != null) {
557                 toolTipText = getLegendItemToolTipGenerator().generateLabel(
558                     dataset, series
559                 );
560             }
561             String urlText = null;
562             if (getLegendItemURLGenerator() != null) {
563                 urlText = getLegendItemURLGenerator().generateLabel(
564                     dataset, series
565                 );
566             }
567             Shape shape = getSeriesShape(series);
568             Paint paint = getSeriesPaint(series);
569             Paint fillPaint = (this.useFillPaint
570                 ? getItemFillPaint(series, 0) : paint);
571             boolean shapeOutlineVisible = this.drawOutlines;
572             Paint outlinePaint = (this.useOutlinePaint
573                 ? getItemOutlinePaint(series, 0) : paint);
574             Stroke outlineStroke = getSeriesOutlineStroke(series);
575             boolean lineVisible = getItemLineVisible(series, 0);
576             boolean shapeVisible = getItemShapeVisible(series, 0);
577             return new LegendItem(label, label, toolTipText,
578                     urlText, shapeVisible, shape, getItemShapeFilled(series, 0),
579                     fillPaint, shapeOutlineVisible, outlinePaint, outlineStroke,
580                     lineVisible, new Line2D.Double(-7.0, 0.0, 7.0, 0.0),
581                     getItemStroke(series, 0), getItemPaint(series, 0));
582         }
583         return null;
584
585     }
586
587     /**
588      * This renderer uses two passes to draw the data.
589      *
590      * @return The pass count (<code>2</code> for this renderer).
591      */
592     public int getPassCount() {
593         return 2;
594     }
595
596     /**
597      * Draw a single data item.
598      *
599      * @param g2  the graphics device.
600      * @param state  the renderer state.
601      * @param dataArea  the area in which the data is drawn.
602      * @param plot  the plot.
603      * @param domainAxis  the domain axis.
604      * @param rangeAxis  the range axis.
605      * @param dataset  the dataset.
606      * @param row  the row index (zero-based).
607      * @param column  the column index (zero-based).
608      * @param pass  the pass index.
609      */
610     public void drawItem(Graphics2D g2, CategoryItemRendererState state,
611             Rectangle2D dataArea, CategoryPlot plot, CategoryAxis domainAxis,
612             ValueAxis rangeAxis, CategoryDataset dataset, int row, int column,
613             int pass) {
614
615         // do nothing if item is not visible
616         if (!getItemVisible(row, column)) {
617             return;
618         }
619
620         // do nothing if both the line and shape are not visible
621         if (!getItemLineVisible(row, column)
```

```
                && !getItemShapeVisible(row, column)) {
            return;
        }

        // nothing is drawn for null...
        Number v = dataset.getValue(row, column);
        if (v == null) {
            return;
        }

        PlotOrientation orientation = plot.getOrientation();

        // current data point...
        double xMargin = 7.0;
        double yMargin = 3.0;
        double x1 = domainAxis.getCategoryMiddle(column, getColumnCount(),
                dataArea, plot.getDomainAxisEdge()) - xMargin;
        double value = v.doubleValue();
        double y1 = rangeAxis.valueToJava2D(value, dataArea,
                plot.getRangeAxisEdge());

        Number previousValue;
        int previousColumn;
        for (int i = 1; ; i++) {
            previousColumn = column - i;
            previousValue = dataset.getValue(row, previousColumn);
            if (previousColumn == 0 || previousValue != null) {
                break;
            }
        }

        if (pass == 0 && getItemLineVisible(row, column)) {
            if (column != 0) {
                if (previousValue != null) {
                    // previous data point...
                    double previous = previousValue.doubleValue();
                    double x0 = domainAxis.getCategoryMiddle(previousColumn,
                            getColumnCount(), dataArea,
                            plot.getDomainAxisEdge());
                    double y0 = rangeAxis.valueToJava2D(previous, dataArea,
                            plot.getRangeAxisEdge());

                    Line2D line1 = null;
                    Line2D line2 = null;
                    if (orientation == PlotOrientation.HORIZONTAL) {
                        if (y1 > y0) {
                            line1 = new Line2D.Double(y0 + yMargin, x0,
                                    y0 + yMargin, x1);
                            line2 = new Line2D.Double(y0 + yMargin, x1,
                                    y1 - yMargin, x1);
                        } else if (y1 < y0) {
                            line1 = new Line2D.Double(y0 + yMargin, x0,
                                    y0 + yMargin, x1);
                            line2 = new Line2D.Double(y0 + yMargin, x1,
                                    y1 + yMargin, x1);
                        } else {
                            line1 = new Line2D.Double(y0 + yMargin, x0,
                                    y1 + yMargin, x1);
                        }
                    } else if (orientation == PlotOrientation.VERTICAL) {
                        if (y1 > y0) {
                            line1 = new Line2D.Double(x0, y0 + yMargin,
                                    x1, y0 + yMargin);
                            line2 = new Line2D.Double(x1, y0 + yMargin,
                                    x1, y1 - yMargin);
                        } else if (y1 < y0) {
                            line1 = new Line2D.Double(x0, y0 + yMargin,
                                    x1, y0 + yMargin);
                            line2 = new Line2D.Double(x1, y0 + yMargin,
```

```
                                    x1, y1 + yMargin);
                        } else {
                            line1 = new Line2D.Double(x0, y0 + yMargin, x1,
                                y1 + yMargin);
                        }
                    }
                    g2.setPaint(getItemPaint(row, column));
                    g2.setStroke(getItemStroke(row, column));
                    g2.draw(line1);
                    if (line2 != null) {
                        g2.draw(line2);
                    }
                }
            }
        }

        if (pass == 1) {
            if (previousValue != null) {
                // previous data point...
                double previous = previousValue.doubleValue();
                double y0 = rangeAxis.valueToJava2D(previous, dataArea,
                        plot.getRangeAxisEdge());

                Shape shape = getItemShape(row, column);
                if (orientation == PlotOrientation.HORIZONTAL) {
                    if (y1 > y0) {
                        shape = ShapeUtilities.createTranslatedShape(shape,
                                y1 - yMargin, x1);
                        shape = ShapeUtilities.rotateShape(shape,
                                Math.toRadians(-90),
                                (float) y1 - (float) yMargin, (float) x1);
                    } else {
                        shape = ShapeUtilities.createTranslatedShape(shape,
                                y1 + yMargin, x1);
                        if (y1 < y0) {
                            shape = ShapeUtilities.rotateShape(shape,
                                    Math.toRadians(90),
                                    (float) y1 + (float) yMargin, (float) x1);
                        }
                    }
                } else if (orientation == PlotOrientation.VERTICAL) {
                    if (y1 > y0) {
                        shape = ShapeUtilities.createTranslatedShape(shape, x1,
                                y1 - yMargin);
                        shape = ShapeUtilities.rotateShape(shape,
                                Math.toRadians(-90),
                                (float) x1, (float) y1 - (float) yMargin);
                    } else {
                        shape = ShapeUtilities.createTranslatedShape(shape, x1,
                                y1 + yMargin);
                        if (y1 < y0) {
                            shape = ShapeUtilities.rotateShape(shape,
                                    Math.toRadians(90),
                                    (float) x1, (float) y1 + (float) yMargin);
                        }
                    }
                }

                if (getItemShapeVisible(row, column)) {
                    if (getItemShapeFilled(row, column)) {
                        if (this.useFillPaint) {
                            g2.setPaint(getItemFillPaint(row, column));
                        } else {
                            g2.setPaint(getItemPaint(row, column));
                        }
                        g2.fill(shape);
                    }
                    if (this.drawOutlines) {
                        if (this.useOutlinePaint) {
```

```
                        g2.setPaint(getItemOutlinePaint(row, column));
                    } else {
                        g2.setPaint(getItemPaint(row, column));
                    }
                    g2.setStroke(getItemOutlineStroke(row, column));
                    g2.draw(shape);
                }
            }

            // draw the item label if there is one...
            if (isItemLabelVisible(row, column)) {
                if (orientation == PlotOrientation.HORIZONTAL) {
                    drawItemLabel(g2, orientation, dataset, row, column, y1,
                            x1, (value < 0.0));
                } else if (orientation == PlotOrientation.VERTICAL) {
                    drawItemLabel(g2, orientation, dataset, row, column, x1,
                            y1, (value < 0.0));
                }
            }

            // add an item entity, if this information is being collected
            EntityCollection entities = state.getEntityCollection();
            if (entities != null) {
                addItemEntity(entities, dataset, row, column, shape);
            }
        } else {
            Line2D line1 = null;
            Line2D line2 = null;
            if (orientation == PlotOrientation.HORIZONTAL) {
                line1 = new Line2D.Double(y1, x1, y1 + yMargin, x1);
                line2 = new Line2D.Double(y1 + yMargin, x1, y1 + yMargin,
                        x1 + xMargin);
            } else if (orientation == PlotOrientation.VERTICAL) {
                line1 = new Line2D.Double(x1, y1, x1, y1 + yMargin);
                line2 = new Line2D.Double(x1, y1 + yMargin, x1 + xMargin,
                        y1 + yMargin);
            }
            g2.setPaint(getItemPaint(row, column));
            g2.setStroke(getItemStroke(row, column));
            g2.draw(line1);
            g2.draw(line2);
        }
    }

}

/**
 * Tests this renderer for equality with an arbitrary object.
 *
 * @param obj  the object (<code>null</code> permitted).
 *
 * @return A boolean.
 */
public boolean equals(Object obj) {

    if (obj == this) {
        return true;
    }
    if (!(obj instanceof LineAndShapeGanttRenderer)) {
        return false;
    }

    LineAndShapeGanttRenderer that = (LineAndShapeGanttRenderer) obj;
    if (this.baseLinesVisible != that.baseLinesVisible) {
        return false;
    }
    if (!ObjectUtilities.equal(this.seriesLinesVisible,
            that.seriesLinesVisible)) {
        return false;
```

```
829             }
830             if (!ObjectUtilities.equal(this.linesVisible, that.linesVisible)) {
831                 return false;
832             }
833             if (this.baseShapesVisible != that.baseShapesVisible) {
834                 return false;
835             }
836             if (!ObjectUtilities.equal(this.seriesShapesVisible,
837                     that.seriesShapesVisible)) {
838                 return false;
839             }
840             if (!ObjectUtilities.equal(this.shapesVisible, that.shapesVisible)) {
841                 return false;
842             }
843             if (!ObjectUtilities.equal(this.shapesFilled, that.shapesFilled)) {
844                 return false;
845             }
846             if (!ObjectUtilities.equal(
847                 this.seriesShapesFilled, that.seriesShapesFilled)
848             ) {
849                 return false;
850             }
851             if (this.baseShapesFilled != that.baseShapesFilled) {
852                 return false;
853             }
854             if (this.useOutlinePaint != that.useOutlinePaint) {
855                 return false;
856             }
857             if (!super.equals(obj)) {
858                 return false;
859             }
860             return true;
861         }
862
863         /**
864          * Returns an independent copy of the renderer.
865          *
866          * @return A clone.
867          *
868          * @throws CloneNotSupportedException  should not happen.
869          */
870         public Object clone() throws CloneNotSupportedException {
871             LineAndShapeGanttRenderer clone =
872                     (LineAndShapeGanttRenderer) super.clone();
873             clone.seriesLinesVisible
874                 = (BooleanList) this.seriesLinesVisible.clone();
875             clone.seriesShapesVisible
876                 = (BooleanList) this.seriesLinesVisible.clone();
877             clone.seriesShapesFilled
878                 = (BooleanList) this.seriesShapesFilled.clone();
879             return clone;
880         }
881
882 }
883
```