JAVAFX CHARTS: IMPLEMENTATION OF MISSING FEATURES

BE
Beams Department

G.Kruk, O.Alves, L.Molinari, CERN, Geneva, Switzerland

Introduction

JavaFX, the GUI toolkit included in the standard JDK, provides charting components with commonly used chart types, a simple API and wide customization possibilities via CSS. Nevertheless, while the offered functionality is easy to use and of high quality, it lacks a number of features that are crucial for scientific or controls GUIs. Examples are the possibility to zoom and pan the chart content, superposition of different plot types, data annotations, decorations or a logarithmic axis. The standard charts also show performance limitations when exposed to large data sets or high update rates.

Some of these features are present in the enhancements list in OpenJFX but up to Java 10 (including) there are no plans to address them by the JavaFX team.

JavaFX Chart Extensions

The central entity of the extension package is *XYChartPane* class, overlaying instances of *XYChart* (see Fig. 1) and managing nodes belonging to custom chart plugins.

The main (base) chart must be specified at the construction of *XYChartPane* but the additional charts, drawn on top of each other, can be added and removed at any moment via exposed observable list of overlay charts.

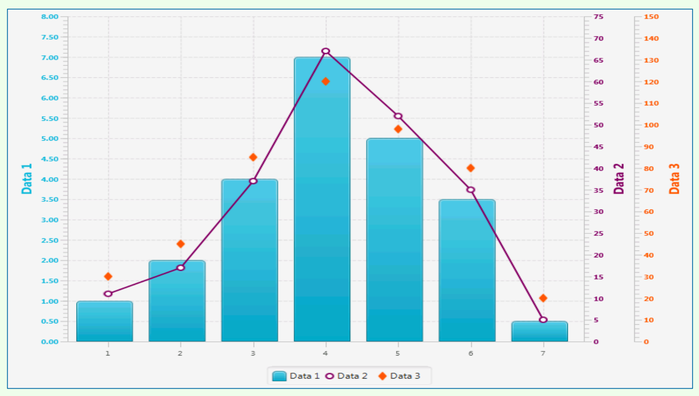

Figure 1: *XYChartPane* with three chart types

```
BarChart<String, Number> barChart =
        new BarChart<>(xAxis(), yAxis());
barChart.getYAxis().setLabel("Data 1");

LineChart<String, Number> lineChart =
        new LineChart<>(xAxis(), yAxis());
lineChart.setTitle("Data 2");
lineChart.getYAxis().setSide(Side.RIGHT);

ScatterChart<String, Number> scatterChart =
        new ScatterChart<>(xAxis(), yAxis());
scatterChart.getYAxis().setLabel("Data 3");
scatterChart.getYAxis().setSide(Side.RIGHT);

XYChartPane<String, Number> chartPane =
        new XYChartPane<>(barChart);
chartPane.setCommonYAxis(false);
chartPane.setLegendVisible(true);
chartPane.getOverlayCharts().add(lineChart);
chartPane.getOverlayCharts().add(scatterChart);
```

NumericAxis and LogarithmicAxis

The JavaFX *NumberAxis* does not allow adding zoom capabilities to *XYChart*s therefore we implemented *NumericAxis* that provides the necessary enhancements. In addition, it can be further customized via dedicated properties:

- ***autoRangePadding*** – fraction of the range to be applied as padding on both sides of the axis.
- ***autoRangeRounding*** – determines if the automatically calculated range should be extended to the next major tick value.
- ***tickUnitSupplier*** – strategy (*TickUnitSupplier*) responsible for calculation of major tick units.

We implemented also *LogarithmicAxis* with a configurable logarithm base and minor tick count (see Fig. 3).

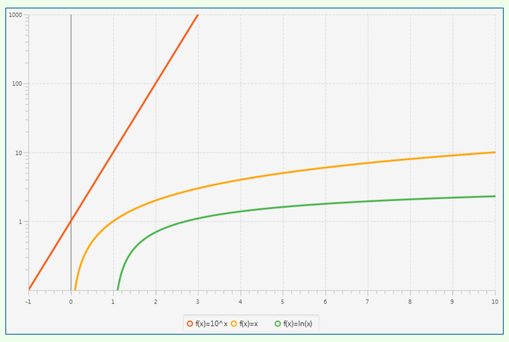

Figure 3: *NumericAxis* (X) and *LogarithmicAxis* (Y)

```
NumericAxis xAxis = new NumericAxis();
LogarithmicAxis yAxis = new LogarithmicAxis();
yAxis.setLowerBound(0.1);
yAxis.setUpperBound(1000);
yAxis.setAutoRanging(false);

LineChart<Number, Number> lineChart =
        new LineChart<>(xAxis, yAxis);
lineChart.setCreateSymbols(false);

lineChart.getData().add(generatePow10());
lineChart.getData().add(generateLine());
lineChart.getData().add(generateLn());
```

Heat Map Chart

HeatMapChart is a specialised chart that uses colours to represent data values contained in a matrix. At CERN, it is typically used by applications displaying beam images like on Figure 4.

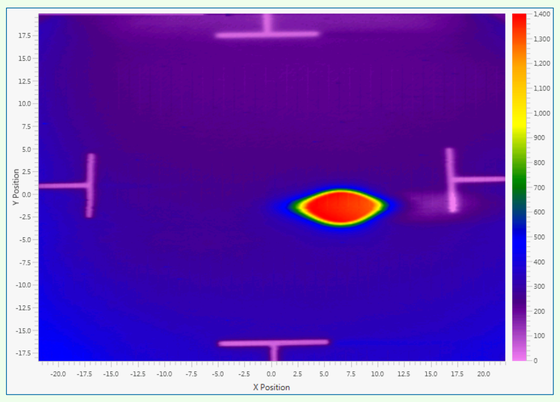

Figure 4: Heat Map Chart with LHC beam image

```
HeatMapChart<Number, Number> chart =
            new HeatMapChart<>(xAxis, yAxis);
chart.setTitle("Beam Image");

chart.getZAxis().setAutoRanging(false);
chart.getZAxis().setUpperBound(1500);
chart.getZAxis().setLowerBound(0);
chart.getZAxis().setTickUnit(100);

Number[] xValues = getXValues();
Number[] yValues = getYValues();
double[][] zValues = getZValues();
DefaultData<Number, Number> imageData =
        new DefaultData<>(xValues, yValues, zValues);
chart.setData(imageData);
chart.setLegendVisible(true);
chart.setLegendSide(Side.RIGHT);
```

Chart Plugins

The *XYChartPane* provides an extension point via *XYChartPlugin* class that allows to interact with the content of displayed charts and to add graphical elements that are drawn on top of them (see Fig. 2).

At the moment the package provides the following plugins:

- ***ChartOverlay*** - allows adding to the chart area any *Node* at an arbitrary position.
- ***CrosshairIndicator*** – cross following the mouse cursor and displaying its coordinates.
- ***DataPointTooltip*** – tooltip label with coordinates of the hovered data point
- ***Zoomer*** – zooms the visible area to the drawn rectangle drawn with mouse cursor.
- ***Panner*** – allows dragging the visible chart area.
- ***XValueIndicator, YValueIndicator*** – line indicating specified X or Y value, with an optional text label.
- ***XRangeIndicator, YRangeIndicator*** – rectangle indicating range between specified X or Y values, with an optional text label.

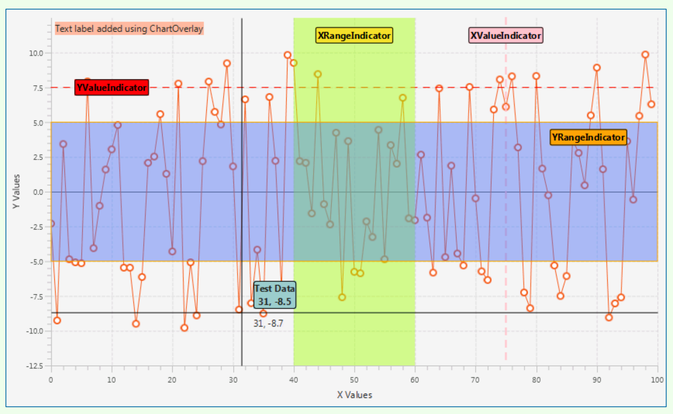

Figure 2: Decorations and annotations drawn on top of *XYChart*

```
List<XYChartPlugin<Number, Number>> plugins = new LinkedList<>();
plugins.add(new XValueIndicator<>(75, "XValueIndicator"));
plugins.add(new YValueIndicator<>(7.5, "YValueIndicator"));
plugins.add(new XRangeIndicator<>(40, 60, "XRangeIndicator"));
plugins.add(new YRangeIndicator<>(-5, 5, "YRangeIndicator"));

Label label = new Label("Label added using ChartOverlay");
AnchorPane.setLeftAnchor(label, 5.0);
AnchorPane.setTopAnchor(label, 5.0);
plugins.add(
        new ChartOverlay<>(PLOT_AREA, new AnchorPane(label)));

plugins.add(new Zoomer());
plugins.add(new Panner());
plugins.add(new CrosshairIndicator<>());
plugins.add(new DataPointTooltip<>());

XYChartPane<Number, Number> chartPane = new XYChartPane<>(chart);
chartPane.getPlugins().addAll(plugins);
```

Every plugin can listen for mouse and keyboard events generated on the *XYChartPane* and react on them accordingly. For instance *CrosshairIndicator* listens for mouse move events, changing location of horizontal and vertical line (following the cursor), and updating text label that displays current coordinates.

Plotting Large Data Sets

The JavaFX charting package performs well with series containing up to a few thousands data points. However drawing series containing tens of thousands points takes it several seconds, efficiently blocking the FX thread and making the application unresponsive.

To address this issue, we developed *DataReducingObservableList*, a specialized implementation of the *ObservableList* interface, performing data reduction to the specified number of most significant points. It is a wrapper over a source list (containing all points) that triggers execution of the reduction algorithm on every change of the source data or X-axis range, exposing to the chart *Series* reduced number of points from the current range. This allows users to see the entire signal and zoom-in to the interesting segments to see more details (see Fig. 5 and 6).

By default, *DataReducingObservableList* uses Ramer-Douglas-Peucker reduction algorithm that is fast and suitable for vast majority of cases. The reduction of 100,000 points to 500 takes around 50-60ms, preserving the original shape of the signal. Usage of other algorithms is possible by providing its implementation via dedicated property.

Figure 5: Chart with two series (100,000 points each) and zoom-in rectangle

Figure 6: Zoomed part of the chart

Conclusion

The implemented extension fulfils the substantial set of features missing in the JavaFX charting package, enabling its usage for all controls applications. All added components follow JavaFX design principles and API style, making their usage simple and intuitive.

The application of data reduction algorithm addresses the performance issues, allowing visualisation of large data sets flawlessly.

Grzegorz Kruk
e-mail: *grzegorz.kruk@cern.ch*
CERN BE/CO

ICALEPCS2017
Barcelona · Spain, October 8-13 · Palau de Congressos de Catalunya