

JOpenChart Developer Guide

Sebastian Müller
sebastian.mueller@berlin.de

20.4.2002

Contents

1	Coding Styles	1
1.1	Class Style	1
1.2	Method Style	2
2	Developer Guide	2
2.1	Package Overview	2
2.2	High-Level Class Overview	3
2.2.1	The ChartDataModel Classes	3
2.2.2	The CoordSystem Class	4
2.2.3	The Render Architecture	5
2.2.4	The Easy Ones: Titles and Legends	5
2.2.5	The Interesting Ones: ChartRenderers	5
2.3	Algorithms Explained	5
2.3.1	The AbstractRenderer image scaling	5
2.3.2	The coordinate transformation	5
2.3.3	The ChartDataModel algorithms	5
2.3.4	The Autoscaling	5
2.3.5	The CoordSystem margin adaption	5

1 Coding Styles

In the following I will shortly explain the basic coding standards used in my code. Any developer who wants to submit patches is encouraged to follow those. My main interest is in the documentation of all changed or added methods and variables etc. There's nothing worse than guessing what code changes are meant to do. The documentation standards are pretty obvious actually if you look at the code. But to make things easier I will explain them nonetheless.

1.1 Class Style

Every class contains the GPL notice at the beginning for obvious reasons. This comment also contains the class' filename and its creation date in the last lines. Every developer is strongly encouraged to add a changed date together with a short description of the committed changes there. Additionally, I always use the javadoc class documentation to create a short description of the things a class is there for. This documentation also contains an `@author` field where any other developer should just add his or her name:

```

/** This class contains the chart title. It's also a Renderer object
 * with some extra properties.
 *
 * @author mueller
 * @version 1.0
 */
public class Title extends AbstractRenderer { ... }

```

I always try to sort class methods in a way, that makes the class generally more readable. That usually means that public methods are written first, followed by the protected and the private ones in that order. Variables are only and exclusively defined in the beginning of a class and import statements are in most cases written using the fully qualified domain name instead of using package imports.

1.2 Method Style

Every method has a javadoc comment as complete as possible. This means especially to use the `@param` and `@return` fields evaluated by the javadoc tool to document the parameters and the return values. If necessary, use `@throws` to document the Exceptions which can be thrown and more importantly the conditions in which they are thrown. If a method is not an interface method with an obvious meaning also document where this method is used to make it easier to understand code patches. You will find multiple examples in my code where I also did that. With all that polymorphism and other object-oriented it's not always easy to see. If you developed some longer more complicated algorithm, which is something not always avoidable in graphics programming, also document what your algorithm does using one line comments.

```

/** This method is called by the paint method to do the actual
 * painting. The painting is supposed to start at point (0,0)
 * and the size is always the same as the preferred size.
 * The paint method performs the possible scaling.
 *
 * @param g the <code>Graphics2D</code> object to paint in
 */
public void paintDefault(Graphics2D g) { ... }

```

2 Developer Guide

2.1 Package Overview

The whole API is split up in 6 packages beginning with the main package `de.progra.charting`.

- `de.progra.charting` - This package contains the main classes and interfaces like the basic `Chart`, the `CoordSystem`, the `Title` etc.
- `de.progra.charting.event` - The event package contains the `Event` classes naturally. They would be used for editable data models but they aren't used yet.
- `de.progra.charting.model` - Quite obviously, this package contains the data model classes beginning with the `ChartDataModel` interface and the `DefaultChartDataModel` and `ObjectChartDataModel` implementations. The `FunctionPlotter` class which plots arbitrary mathematical functions is also part of this package because this class creates a `ChartDataModel`.

- `de.progra.charting.render` - This is the second important package containing all rendering classes starting with the `Renderer` interface and all `ChartRenderers`. To create your own `Renderer` just extend `AbstractChartRenderer` and look at the other renderers to see how it's done.
- `de.progra.charting.servlet` - Up to now, this package contains exactly one class, the `ChartServlet`. This is a web application wrapper which takes the image type and the `Chart` object as arguments and returns an image stream of the desired format.
- `de.progra.charting.swing` - Similarly to the servlet package, this class contains a Swing wrapper - the `ChartPanel`. It works exactly like the `DefaultChart` class and can be embedded into a Swing application.
- `de.progra.charting.test` - This package contains two important classes: `GraphFrame` and `TestChart`. The first one can be executed to display a chart in a Swing application while the latter creates several sample charts and exports them to the user's home directory. The two JSP pages show the possibilities of the `ChartServlet` class, especially the `FunctionPlot.jsp` is a nice example.

2.2 High-Level Class Overview

The whole library is centered around the `DefaultChart` class. This is the point to start when you create a new chart but also if you want to understand how things work. A chart contains several parts as depicted below.

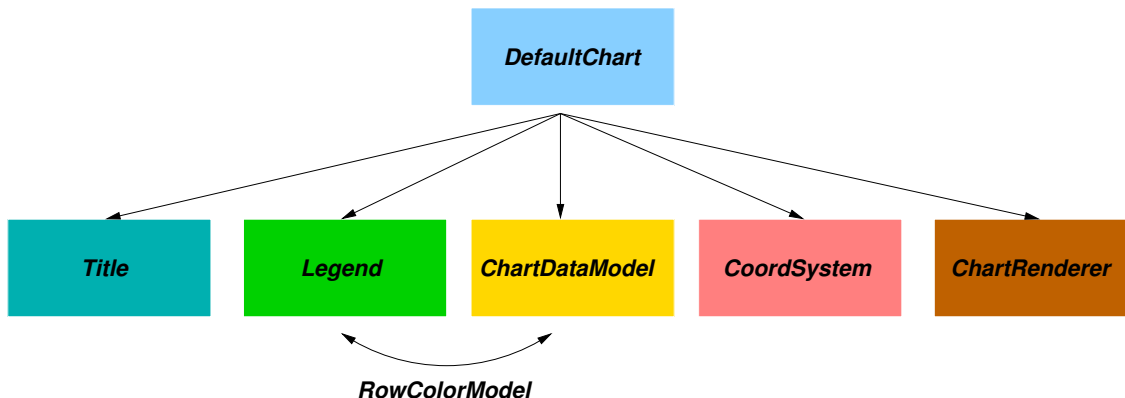


Figure 1: The basic structure of the charting classes.

The `Title` class just encapsulates the `Chart`'s title as the name implies. Like the `Legend` this class extends `AbstractRenderer` which will be explained later on. The `Legend` is the second one of the easy classes. Again it's pretty obvious what it is, the `Chart`'s legend of course. The last three ones - the `ChartDataModel`, the `CoordSystem` and the `ChartRenderer` Architecture - will be explained in more detail later on. As you can see in the picture, the `Legend` and the `ChartDataModel` have a special connection which is the `RowColorModel`. This class encapsulates the connection between data sets and colors.

2.2.1 The `ChartDataModel` Classes

You can get a first grasp of the model architecture if you take a look at the `ChartDataModel` interface. It looks pretty similar to the `TableDataModel` from the Swing package. Superficially,

this is right. For a moment, just think of the `ChartDataModel` as a simple table. The table columns define the x-axis values while every table row is a new data set:

<i>x-axis values</i>	0	1	2	3	4
<i>Data Set 1</i>	0	2	4	6	8
<i>Data Set 2</i>	0	1	4	9	16

As you can see, *DataSet 1* defines the linear function $f(x) = 2x$ and *DataSet 2* defines the function $f(x) = x^2$. You find all the usual table methods in the interface. You can read a certain value with `getValue(int set, int index)`, read a certain column (ie x-axis) value, determine the length of a certain data set and get the title of a data set (eg *Data Set 1* in the table above). There is another method called `getChartDataModelConstraints(int axis)`. This returns a `ChartDataModelConstraints` object which you will easily understand if you look at the `ChartDataModelConstraints` interface. This interface just contains four methods which provide access to the minimal and maximal x and y-axis values. This is necessary for rendering the coordinate system.

Of course, the whole thing wouldn't be very flexible if the data model was really a table. You will find an extra `DataSet` interface and a `DefaultDataSet` implementation in the model package and those two encapsulate the data sets. This way, a `ChartDataModel` can contain multiple `DataSets` with different length and they can even have their own x-axis values. At least, this is true for data models with numerical x-axis values where you can paint every arbitrary value as long as you know the `ChartDataModelConstraints`, basically because numbers are automatically ordered in a mathematical sense. This is different for data models with non-numerical x-axis values, eg usual bar charts fall in this category. In the next release it will be possible to create such charts using data sets with different lengths, as long as you provide an ordering for the different x-axis values.

There are two different `ChartDataModel` implementations which meet those two different requirements. `DefaultChartDataModel` implements the data model for the use with numerical x-axis values while `ObjectChartDataModel` has arbitrary `Objects` as x-axis values. They could be `Strings` or `Dates` for example. Both implementations are pretty straightforward, except computing the `ChartDataModelConstraints` which will be explained in detail in the Algorithms part of this guide.

2.2.2 The CoordSystem Class

The coordinate system implementation is the core class of all the later rendering stuff. That's why this class has to do a bit of computing. It needs to know the `ChartDataModelConstraints` and it's own bounds and then it can compute the placing of the axes, the point (0, 0) and the pixel / point ratio. Besides that, this class contains a lot of rendering code for painting the axes, the ticks, labels etc. pp. All this stuff has been externalized to the `CoordSystemUtilities` class to make the whole shebang more readable. The most important part is transforming the user space coordinates of the `ChartDataModel` to the device space coordinates used to paint in an image or on the screen. This is done using the `AffineTransform` class of the `java.awt.geom` package. This class defines a multiplication matrix which is used by the `ChartRenderers` to transform those coordinates. The mathematical part is as usual explained in the Algorithms part. It is possible to chain `AffineTransforms` which will be used in the Swing chart classes to move charts around on the screen.

Another thing you may wonder about is the second y-axis. It is not really implemented yet, but sometimes it is desirable to combine two data sets in one chart although their y-axis values have different units. You probably know charts which visualize the development of processor speed

and transistors per processor. Those charts need to y-axis because processor speed and transistors per processor are different units.

2.2.3 The Render Architecture

Now I finally come to the interesting part which still needs some work. The place to start is the `Renderer` interface. It defines only a few methods for setting the bounds, getting the preferred size and finally rendering what has to be rendered. This interface is not incredibly interesting of course. It gets more interesting when you look at the `AbstractRenderer` and the `AbstractChartRenderer`. The former implements the boring getter and setter method for the bounds object but it also provides an implementation for the render method. Additionally, it defines a `paintDefault(Graphics2D g)` method. So what is that? Well, the default render method takes care for the fact, that the assigned bounds to a render object might be smaller than the preferred size. In this case it creates an offscreen image in which the actual renderer will paint and which will be scaled down afterwards to the size of the bounds object. This means that every `Renderer` implementation only has to provide an implementation for the `paintDefault(Graphics2D g)` method which can always assume the preferred size and to start painting at coordinates (0, 0).

The latter class I mentioned - `AbstractChartRenderer` - provides default implementations for `ChartRenderers`. This includes mainly several interface methods like getters and setters for the `CoordSystem` etc. The only method left to implement in a real `ChartRenderer` is the `render(Graphics2D g)` method. These two abstract classes simplify creating new `Renderers` a lot. Have fun, there are a lot of `ChartRenderers` missing.

2.2.4 The Easy Ones: Titles and Legends

Maybe it's a little bit surprising, but in fact, both `Title` and `Legend` objects are themselves `Renderers`. This is pretty handy if you think of the automatic scaling capabilities in the `AbstractRenderer` class. Besides that, there's not much to say about them. Just look at the code, it's pretty straightforward.

2.2.5 The Interesting Ones: ChartRenderers

The more interesting renderers are the real `ChartRenderers`. As described above, all they do is to implement the render method. This isn't extraordinarily exciting either. In fact, if you compare the `Renderers` you'll find that they are all very similar. They all iterate over all `DataSets` in the `ChartDataModel` and somehow they all paint the data found in there in a small iteration loop.

2.3 Algorithms Explained

2.3.1 The AbstractRenderer image scaling

2.3.2 The coordinate transformation

2.3.3 The ChartDataModel algorithms

2.3.4 The Autoscaling

2.3.5 The CoordSystem margin adaption

The image below depicts the main coordinate system constraints. The innerbounds box is pretty self-explanatory. The `xunit` and `yunit` labels mark the space were the axes' units are rendered.

The ticklabels boxes are the places where the axis labels are rendered and the xlength and ylength values are the lengths of the left x-axis and the lower y-axis part.

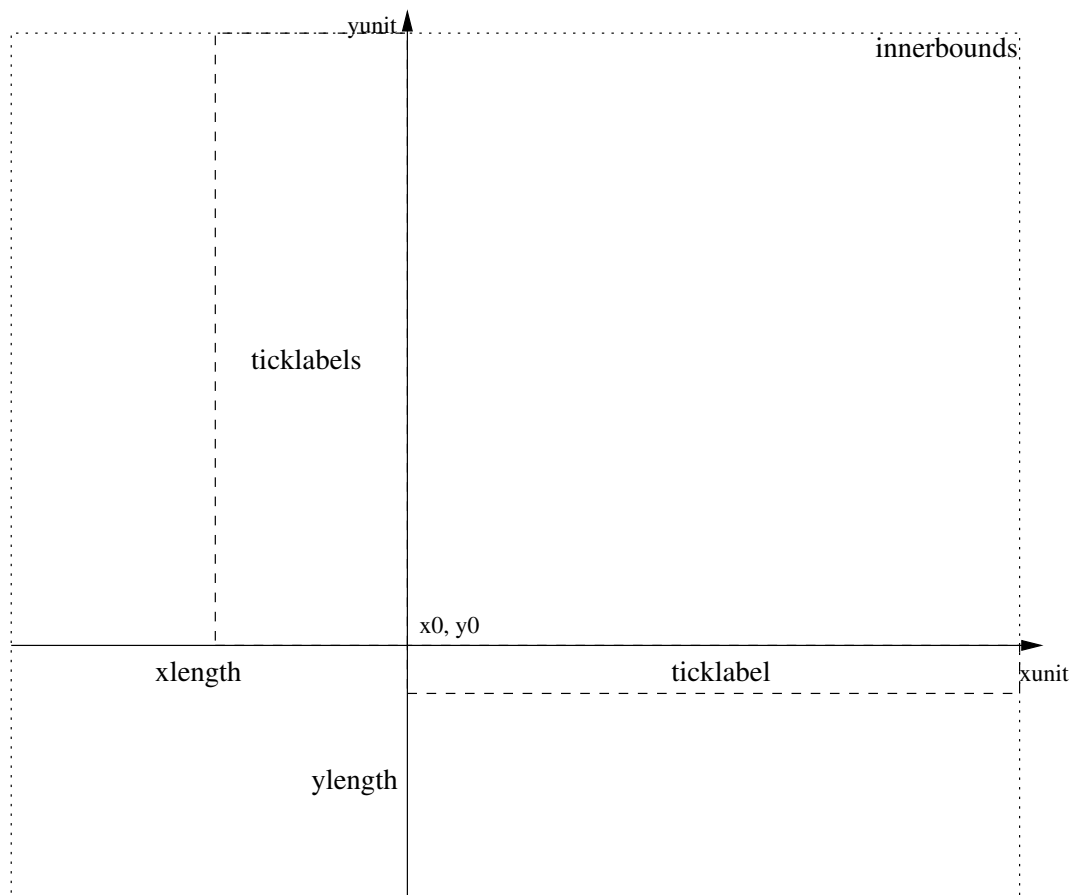


Figure 2: A sketch of the coordinate system's rendering constraints.

The margin is the space between the inner bounds and the total bounds in which all the renderings of the coordinate system have to fit. In this example the top and right margin are easily computable:

$$rightmargin = \text{Math.max}(arrowlen, xunit.width) \quad (1)$$

$$topmargin = \text{Math.max}(arrowlen, yunit.height) \quad (2)$$

Now comes the part with all the work. The difficulty is computing the lower and the left margin. The maximal left margin can be computed as

$$maxlmargin = \text{Math.max}(ticklabels.width, yunit.width) \quad (3)$$

The real margin is the part of the maxlmargin which is longer than xlength, ie the length of the negative x-axis. The equation for computing the margin is developed in the next few steps where xmin and xmax are the minimal and maximal x-axis values:

$$xlength \leq bounds.width - rightmargin \quad (4)$$

$$maxlmargin - margin = \frac{|xmin|}{|xmin| + xmax} \cdot xlength > maxlmargin \quad (5)$$

$$\Leftrightarrow xlength = (maxlmargin - margin) \cdot \frac{xmax + |xmin|}{|xmin|} \leq bounds.width - rightmargin \quad (6)$$

$$\Rightarrow margin = maxlmargin - \frac{|xmin|}{|xmin| + xmax} \cdot (bounds.width - rightmargin) \quad (7)$$

The algorithm is equivalent for the lower margin. In this case, the lower and left margins have to be computed after the right and top margin. They have to be computed the other way around, if the maximum x-axis value is negative which means that the y-axis is painted at the right side. The same applies if the maximum y-axis value is negative.