IBM ILOG JViews Charts V8.6

Developing with the JViews Charts SDK

Copyright

Table of contents

Introducing the Main Classes

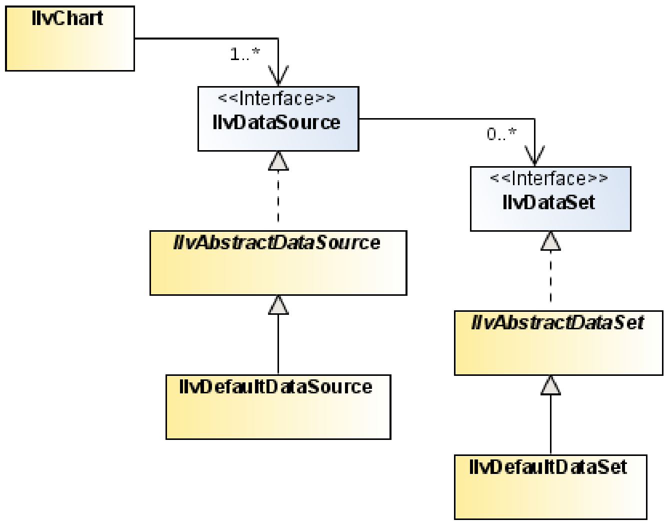
Gives a brief introduction to the main classes of JViews Charts.

In this section

Data Model Classes

The data model is defined by two distinct interfaces: `IlvDataSet` and `IlvDataSource`. The `IlvDataSet` interface acts as a data holder. The objects that implement this interface manage data as a set of data points, and provide the required Application Programming Interface (API) to fetch and modify data. An abstract implementation of this interface called `IlvAbstractDataSet` is provided in the library as well as several concrete implementations.

The `IlvDataSource` interface acts as a data set source. It handles a collection of data sets and provides the required API to access them.The implementation of this interface allows you to import data from an external package (for example a database, an XML file, and so on) into the JViews Charts library. An abstract implementation called `IlvAbstractDataSource` is provided in the library as a starting point for your own custom implementation. A memory-based implementation is provided by the `IlvDefaultDataSource` class.



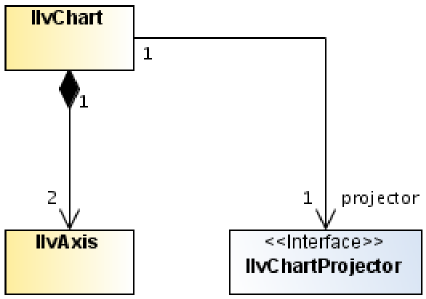
Data Model Classes

Data Projection

Data Projection is the process of mapping data points provided by the data model into display points used by chart renderers. This mapping corresponds to a conversion between two coordinate systems:

The data space, which defines the coordinate system where data is expressed. This coordinate system is defined by chart coordinate axes.

The display space, which defines the coordinate system where data points are projected. This coordinate system is equivalent to the user space in Java2D terminology. In other words, it corresponds to the coordinate system used by rendering routines.



Data Projection Classes Relationships

Chart Axis and Chart Projector

Each chart uses several coordinate axes, which are represented by the `IlvAxis` class.

The conversion between data space and display space is performed by a projector owned by the chart. Depending on its type, a chart uses one of the two predefined projectors available in the Charts package:

- ♦ Cartesian projector
- ♦ Polar projector

The projector used by a chart can be retrieved with the `IlvChart.getProjector` method.

Data Display Classes

The graphical representation of a data source is defined according to the following criteria:

- ♦ The global characteristics of a chart (for example, whether it uses Cartesian or polar projection)
- ♦ The rendering type used to display data on the screen (as a polyline, bar, stair, and so on).

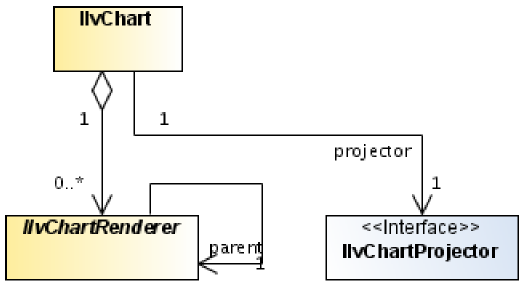
The conversion from data space to screen coordinates is handled by a projector object, which is an instance of `IlvChartProjector`. The type of a projector object is defined by the type of the chart and is a global parameter of a chart:

- ♦ A Cartesian projector when using a Cartesian chart.
- ♦ A polar projector when using a radar, polar, or pie chart.

The way data is rendered on the screen (as a polyline, a bar, or a bubble) is handled by chart renderer objects, which are instances of subclasses of `IlvChartRenderer`.

There are three types of chart renderers:

- ♦ Composite renderers (instances of `IlvCompositeChartRenderer`).
- ♦ Single renderers (instances of `IlvSingleChartRenderer`).
- ♦ Simple renderers (instances of `IlvSimpleChartRenderer`).



Data Display Classes Relationship

Composite Renderers

The composite renderers are used to render the contents of a data source using a collection of child chart renderers so that each data set in the associated data source is rendered by one of these child renderers.

Depending on the rendering type, the relation between a data set and a child renderer can be one of the following:

- ♦ unary relation: one child renderer per data set.

The unary relations are handled by instances of subclasses of `IlvSimpleCompositeChartRenderer`. An example of a simple composite chart renderer is the polyline renderer, which renders each data set of the data source by a polyline.

- n-ary relation: one child renderer using several data sets.

  The n-ary relations are handled by direct subclasses of `IlvCompositeChartRenderer`. An example of a composite chart renderer that uses several data sets for one graphical representation is the high-low renderer (hilo), which needs two data sets (high and low values) for one graphical representation.

Single Renderers

Single renderers are used as elementary chart renderers by composite renderers to draw the graphical representation of data sets. While a composite renderer handles a data source, a single renderer handles a data set. Single renderers can also be used directly when you handle a specific data set.

Simple Renderers

Simple renderers are used to render the contents of a data source. They do not make use of other renderers for particular data sets. Instead, they do the rendering all on their own.

Binding a data model and graphical representation

A data source is rendered by a composite chart renderer.

To associate a data source with a chart renderer:

- Use the following method defined in the `IlvChartRenderer` class:

```
void setDataSource(IlvDataSource dataSource)
```

To obtain the current data source of a chart renderer:

- Use the following method:

```
IlvDataSource getDataSource()
```

Interactor Classes

Chart interactors let you associate one or several behaviors with a chart object. Chart interactors define atomic interactions that can be combined together and extended to achieve complex interactive functionalities.

Each chart interactor (subclasses of `IlvChartInteractor`) implements a given type of interactive operation: scrolling, zooming, editing, or highlighting data points.

As a result of this clean separation, chart interactors are lightweight and well-defined event-handling entities that can be easily customized.

The base class used to define the behavior of a chart in response to a given action by the user is the `IlvChartInteractor` class.

Creating a Chart

Explains how to create different types of chart and how to customize them.

In this section

Creating a basic chart

The `IlvChart` class provides a set of convenience methods that may reduce the number of steps required to create a chart.

These methods are:

- ♦ `setRenderingType(int)`

  Sets the type of the chart renderers to use to represent a stand-alone data source.

- ♦ `setDataSource(ilog.views.chart.data.IlvDataSource)`

  Sets the specified data source as the new data source and represents it with the default chart renderer type.

- ♦ `addData(ilog.views.chart.data.IlvDataSource, int)`

  Connects the given data source to the chart and represents it with the specified chart renderer type.

- ♦ `addRenderer(ilog.views.chart.IlvChartRenderer, ilog.views.chart.data.IlvDataSet)`

  Adds the specified renderer to the chart and initializes its data source with the specified data set.

1. **Create the data model.**
   - ♦ Create the object representing the data source you want to display. This object is an instance of a concrete implementation of the `IlvDataSource` interface.
   - ♦ Put the data to be displayed into the created data source.
2. Create the renderer that will display the graphical representation of the data. The renderer is an instance of one of the `IlvChartRenderer` subclasses.
3. Set the data source as the chart renderer data source.
4. Create the chart.

   The created chart object is an instance of the `IlvChart` class, and its type depends on the type of chart you want to display.
5. Add the chart renderer to the chart object.

Creating a basic Cartesian chart

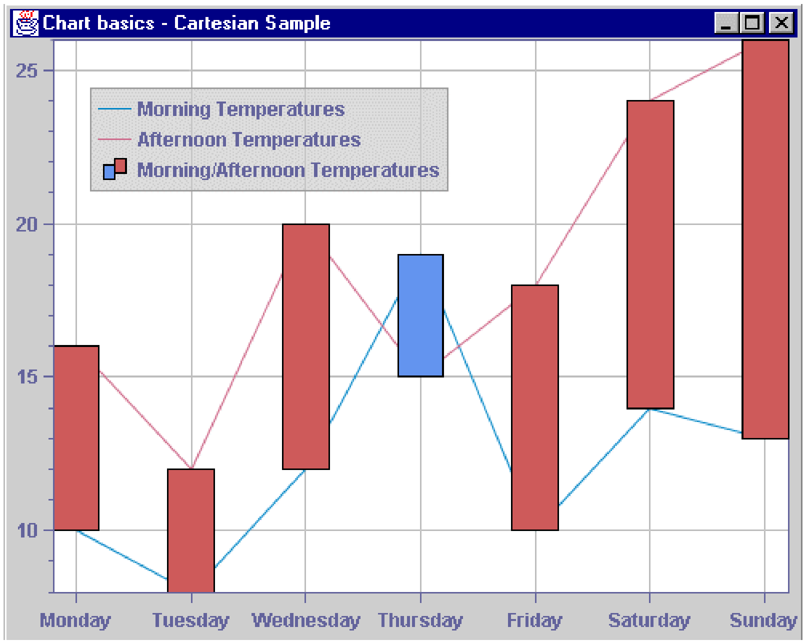
The data to be represented is the morning and afternoon mean temperatures (expressed in degrees Celsius) recorded for each day of a week. The days of the week are referenced by values ranging from 0 to 6. The data series is composed of categories. The data for the chart is listed in *Data for the Example Chart*. You will see the steps required to display this data in a Cartesian chart.

Data for the Example Chart

Day	Morning Mean Temperature (C)	Afternoon Mean Temperature (C)
0	10	16
1	8	12
2	12	20
3	19	15
4	10	18
5	14	24
6	13	26

Example Cartesian Chart shows the Temperatures Chart that will be created to display our data.

The morning mean temperatures will be displayed with a blue polyline (the bottom line of the chart) and the afternoon mean temperatures with a red polyline (the top line of the chart). The chart will also display the two sets of temperatures with high-low bars in order to illustrate the variation between the morning and afternoon temperatures for each day of the week.



Example Cartesian Chart

The complete source code of this example can be found in **`<installdir>/jviews-charts86/codefragments/chart/basic-cartesian/src/Cartesian.java`**.

Creating the data model

You are going to display two data sets on the same chart: the morning mean temperatures and the afternoon mean temperatures for each day of a week. The days will be plotted along the abscissa scale and the temperatures along the ordinate scale. Since the day values match the index of the temperature value in the data model, there is no need to define a specific *x*-series in the data model but instead only the *y*-values.

In the JViews Charts library, the data model is defined by the `IlvDataSource` interface. In this example, the data source holding both the morning and afternoon temperature values is an instance of the `IlvDefaultDataSource` class. This class provides a default memory-storage implementation of the `IlvDataSource` interface.

1. Create the values to be put in the data source.

```
// Create the initial values array.
double[][] temps = {
    {10, 8, 12, 19, 10, 14, 13},
    {16, 12, 20, 15, 18, 24, 26}
};
```

2. Create the data source.

```
String[] names = {"Morning Temperatures",
                  "Afternoon Temperatures"};
String[] labels = {"Monday","Tuesday", "Wednesday", "Thursday",
                  "Friday", "Saturday", "Sunday"};

// Create one data source to store the temperatures.
// No x series since the x-values are actually the index
// of the y-value in the data set.
IlvDataSource tempDataSource =
    new IlvDefaultDataSource(temps, -1, names, labels);
```

The data source is initialized with an array containing the temperature values for each series. These series are internally stored in the data sets. Since you do not have any *x*-series, you specify -1 as the *x*-series index, and you name the data sets holding the temperature values.

By default, the labels displayed at the major tick marks of the scales are floating values corresponding to the data values represented by the scales. In this case, the labels displayed along the abscissa scale would be the indexes from 0 to 6 referencing the days of the week. To make these values more understandable, display the names of the days instead of the indexes. To do this, you will enable the category mode of the xscale (in this mode, the scale is configured to display categories), but the first thing to do is to specify the labels you want to use. This is done during the creation of the data source, passing an array of strings as the last parameter of the `IlvDefaultDataSource` constructor. This array of labels is referenced by each data set held by the data source and will be used by the scale steps definition to label the steps.

Creating a Cartesian chart

♦ To create a Cartesian chart, use the following code:

IlvChart chart = new IlvChart();

The chart type is not explicitly specified: it is Cartesian by default.

Creating and adding the renderers

You are going to display the two sets of temperatures with two polylines and with a high-low bar representation. The first polyline represents the data set of the morning mean temperatures, and corresponds to the first series of values in the data source. The second polyline represents the data set of the afternoon mean temperatures, which corresponds to the second series of values in the data source. The high-low bar representation represents both the data sets of the morning and afternoon mean temperatures.

1. Create and add the *hilo* renderer.

```
IlvHiLoChartRenderer hiloRenderer =
            new IlvHiLoChartRenderer();
hiloRenderer.setWidthPercent(40);
hiloRenderer.setDataSource(tempDataSource);
IlvStyle[] styles = {
    new IlvStyle(Color.black, IlvColor.indianRed),
    new IlvStyle(Color.black, IlvColor.cornflowerBlue)
};
hiloRenderer.setStyles(styles);
hiloRenderer.getChild(0).setName("Morning/Afternoon temperatures");
chart.addRenderer(hiloRenderer);
```

Since the data contained in the temperature data source is rendered as *hilo* bars, the temperature data source is set as the data source of an `IlvHiLoChartRenderer` instance, which represents the variation of two series of values.

The default graphical representation of the *hilo* renderer is a bar. The width of a bar is expressed as a percentage of the space available between two categories. In this example, the width is set to 40%.

Set the rendering styles used by the *hilo* renderer so that:

- ♦ the high-low items for which the corresponding first value (that is, the morning mean temperature) is smaller than the second value (that is, the afternoon mean temperature) are drawn as a red bar.
- ♦ the high-low items for which the corresponding first value is greater than the second value are drawn as a blue bar.

2. Create and add the polyline renderers.

```
IlvChartRenderer tempRenderer = new IlvPolylineChartRenderer();
tempRenderer.setDataSource(tempDataSource);
chart.addRenderer(tempRenderer);
```

The temperature data source is set as the data source of an `IlvPolylineChartRenderer` instance, that draws one polyline for each data set held by its data source. Note that you did not specify any rendering style for this renderer. In this case, the renderer uses a default rendering style with a unique color chosen in a list of predefined default colors. This default colors list is retrieved using the `getDefaultColors()` method. If this method returns `null` (which is the default implementation), the color is then chosen in the list returned by `IlvColor.getDefaultColors` list. To change the default colors, you can either override the `IlvChartRenderer.getDefaultColors` method to return your own list for a given chart renderer class, or edit the default colors list of the `IlvColor` class to change the default colors for all chart renderer classes.

Customizing a basic Cartesian chart

The following steps illustrate how to customize the abscissa and ordinate scales, and how to add a legend to a chart.

To customize the abscissa scale:

♦ Configure the scale to display the labels associated with the category of the data sets. Enable the category display mode on the abscissa scale, so that numeric labels are replaced by the labels of the days:

```
chart.getXScale.setCategory(tempDataSource.getDataSet(0),false);
```

The computation of the steps and substeps for a given scale is performed by a dedicated object called steps definition, which is set on the scale.

Two types of steps definition are available:

♦ numerical steps definition, which handles numeric steps values,

♦ time steps definition, which handles time values.

The steps definition type depends on the scale type: numeric steps definition for `IlvScale.DEFAULT_SCALE` and time steps definition for `IlvScale.TIME_SCALE`.

Note: By default, a steps definition object is automatically set on a scale when it is created.

In this mode, the steps are determined according to the number of categories. The data set parameter is used to provide steps labels to the scale steps definition, and the `false` parameter is used to make steps appear at each category. You could have set the parameter to `true` to make steps appear between categories.

To customize the ordinate scale:

♦ Set the step and the substep units for the ordinate scale. Specify 5 as the step unit and 1 as the substep unit to have a major tick mark appear every five degrees and a minor tick mark every degree.

```
chart.getYScale(0).setStepUnit(5.,1.);
```

The steps will be marked with a major tick mark and the substeps with a minor tick mark.

When a numeric scale is created, a numeric steps definition is set and the steps values are automatically computed. You do not want the values to be automatically computed, but rather manually set. The ordinate scale represents the temperature in degrees Celsius.

To add a legend:

♦ Add the legend with the ABSOLUTE constraint:

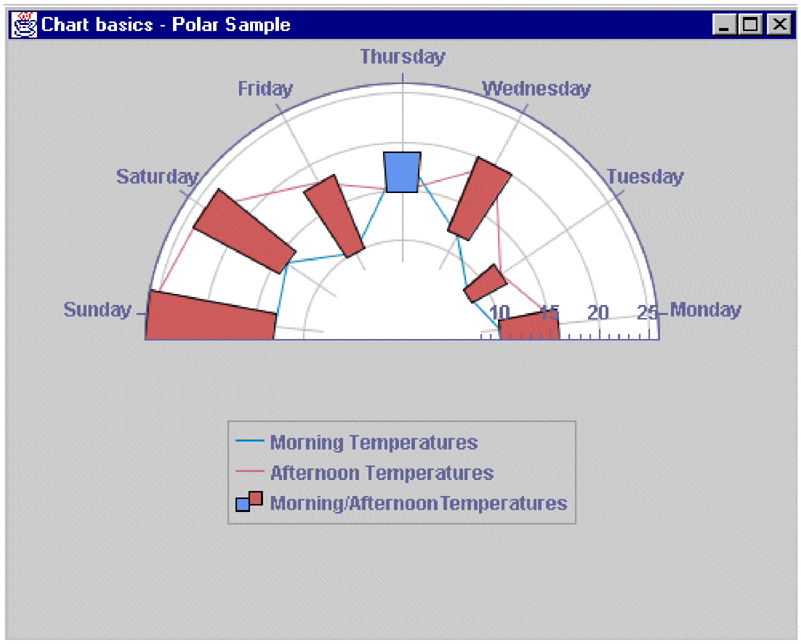
```
IlvLegend legend = new IlvLegend();
legend.setLocation(250,270);
chart.addLegend(legend, IlvChartLayout.ABSOLUTE);
```

to have the legend displayed within the chart area.

A legend is an instance of the `IlvLegend` class (a `JComponent` subclass) that is associated with an `IlvChart`. A legend can be added to an `IlvChart`, but it is not mandatory. In this case, a legend can be added at several predefined positions (see the `IlvChartLayout` class) or at an absolute position, using its current location.

Creating a basic polar chart

You are going to create a polar chart to represent the morning and afternoon mean temperatures. The days of the week will be represented along the abscissa and the temperatures along the ordinate. The abscissa values will be mapped along a circular scale and the ordinate values will be displayed radially. You will customize the scales and finally, you will add a legend to the chart.



Example Polar Chart

The complete source code of this example can be found in **`<installdir>/jviews-charts86/codefragments/chart/basic-polar/src/Polar.java`**.

Creating the Data Model

- ♦ Create the data model and put the data to be displayed into the data source. Use the same data and procedures as for the Cartesian chart example. See *Creating the data model*.

Creating a Polar Chart

1. To create a polar chart, use the following code:

```
IlvChart chart = new IlvChart(IlvChart.POLAR);
chart.setAngleRange(180);
```

2. Create an instance of `IlvChart`, and pass the `IlvChart.POLAR` type constant as parameter to the constructor.
3. Set the angle range to 180 degrees, specifying the angle range within which the data will be projected on the screen.

Creating and Adding the Renderers

♦ Because you use the same graphical representation for your data, you will use the same renderers as for the Cartesian chart. See *Creating and adding the renderers*.

Customizing a basic polar chart

The following steps illustrate how to customize the abscissa and ordinate scales, and how to add a legend to a chart.

To customize the abscissa scale:

- ♦ Customize the abscissa scale as you did for the Cartesian chart. See *Customizing a basic Cartesian chart*.

To customize the ordinate scale:

- ♦ Customize the ordinate scale as you did for the Cartesian chart. See *Customizing a basic Cartesian chart*.

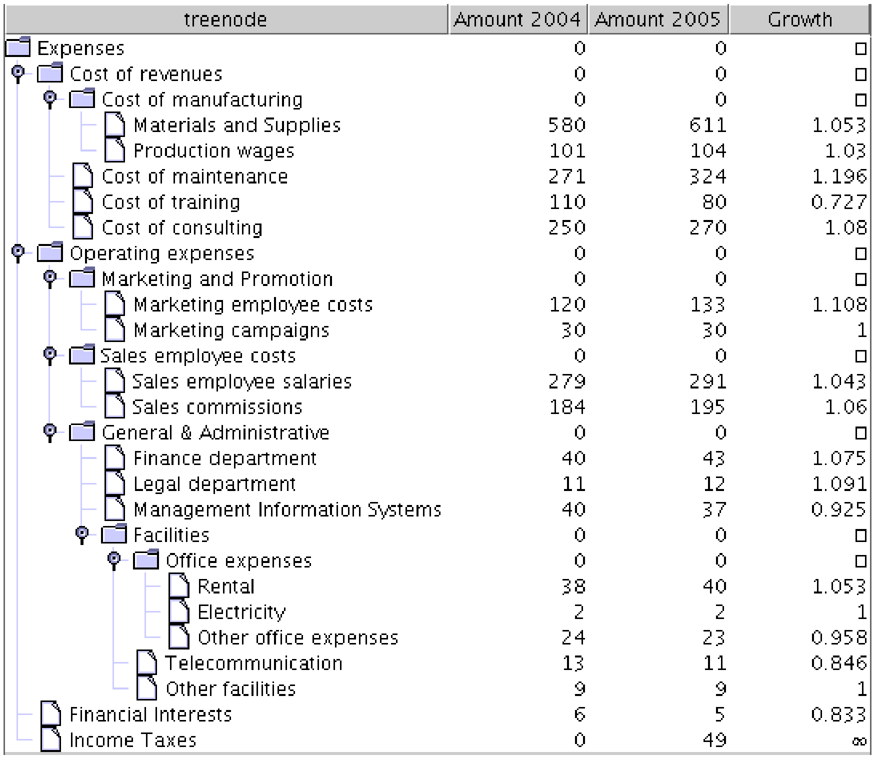
To add a legend:

- ♦ Add a legend to your data. Use the same procedures as for the Cartesian chart example. See *Customizing a basic Cartesian chart*.

Creating a basic treemap chart

A treemap displays entities in such a way that important entities stand out visually and the less important ones are visually insignificant. A treemap is optimal for getting an overview of all entities and for spotting particular ones with extraordinary characteristics.

The data to be represented is the expenses of a fictive company. Data has the shape of a tree table, where objects are organized in a tree structure with several values per object, as illustrated in *Example of Expenses Organized in a Tree Structure*.



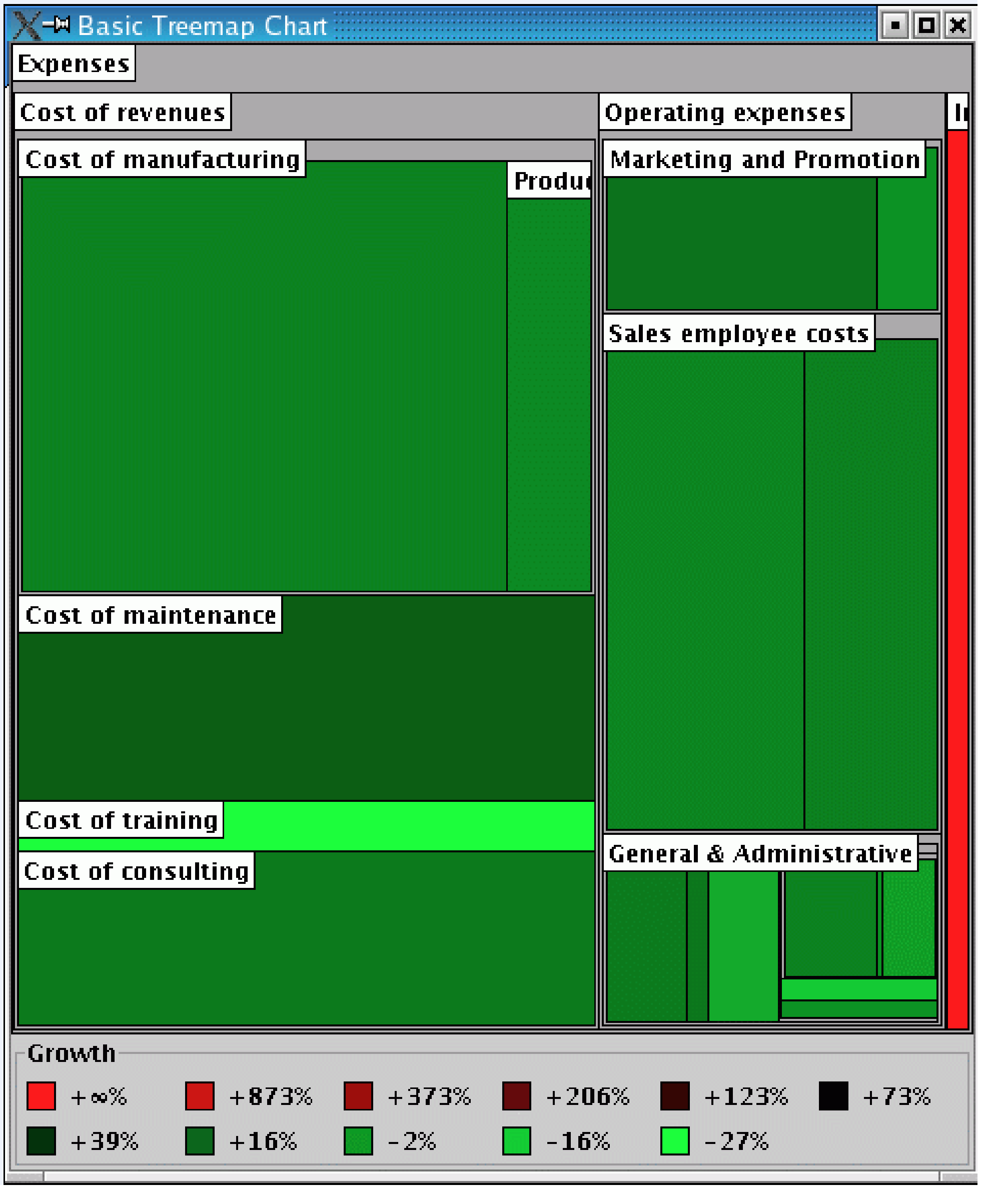
treenode	Amount 2004	Amount 2005	Growth
Expenses	0	0	□
Cost of revenues	0	0	□
Cost of manufacturing	0	0	□
Materials and Supplies	580	611	1.053
Production wages	101	104	1.03
Cost of maintenance	271	324	1.196
Cost of training	110	80	0.727
Cost of consulting	250	270	1.08
Operating expenses	0	0	□
Marketing and Promotion	0	0	□
Marketing employee costs	120	133	1.108
Marketing campaigns	30	30	1
Sales employee costs	0	0	□
Sales employee salaries	279	291	1.043
Sales commissions	184	195	1.06
General & Administrative	0	0	□
Finance department	40	43	1.075
Legal department	11	12	1.091
Management Information Systems	40	37	0.925
Facilities	0	0	□
Office expenses	0	0	□
Rental	38	40	1.053
Electricity	2	2	1
Other office expenses	24	23	0.958
Telecommunication	13	11	0.846
Other facilities	9	9	1
Financial Interests	6	5	0.833
Income Taxes	0	49	∞

Example of Expenses Organized in a Tree Structure

A treemap chart displays objects as rectangles. The important objects are represented by large rectangles while the less important objects are represented by smaller rectangles.

Rectangles relate each other in a containment structure, where each rectangle is part of another rectangle.

Example of Treemap Chart shows the treemap chart that will be created to display your data.



Example of Treemap Chart

The complete source code of this example can be found in **`<installdir>/jviews-charts86/codefragments/chart/basic-treemap/src/Treemap.java`**.

Creating the data model

The data model, among those offered in the packages `ilog.views.chart.data` and `ilog.views.chart.datax`, that best matches the tree structure is the `IlvTreeListModel`. The `IlvTreeListModel` can be found in the `ilog.views.chart.datax` package.

To create the data model, you first need create the objects that represent the data and then create an instance of the `IlvDefaultTreeListModel` and add objects to it.

1. Create the objects.

The objects in the model have a name and two numerical attributes: the amount of expenses in the years 2004 and 2005. You will also use a computed attribute: the growth from 2004 to 2005.

```
class ExpenseItem {

    private String name;
    private double amount2004;
    private double amount2005;
    /**
     * Creates an expense item without associated amounts.
     */
    public ExpenseItem(String name) {
        this.name = name;
    }
    /**
     * Creates an expense item with associated amounts.
     */
   public ExpenseItem(String name, double amount2004, double amount2005)
 {
        this.name = name;
        this.amount2004 = amount2004;
        this.amount2005 = amount2005;
    }
    /**
     * Returns the amount for the year 2004, or 0 if none.
     */
    public double getAmount2004() {
        return this.amount2004;
    }
    /**
     * Returns the amount for the year 2005, or 0 if none.
     */
    public double getAmount2005() {
        return this.amount2005;
    }
    /**
     * Returns the growth factor from 2004 to 2005.
     */
    public double getGrowth() {
        if (this.amount2004 != 0)
            return this.amount2005 / this.amount2004;
        else if (this.amount2005 > 0)
            return Double.POSITIVE_INFINITY;
        else if (this.amount2005 < 0)
            return Double.NEGATIVE_INFINITY;
        else
            return Double.NaN;
    }
    /**
     * Returns the string representation of this object.
     * For simplicity, we use the name here.
     */
    public String toString() {
        return this.name;
    }
}
```

2. Enumerate the object properties.

Every attribute of the `ExpenseItem` class corresponds to a column in the tree-table, illustrated in *Example of Expenses Organized in a Tree Structure*. Each attribute of a particular object in the tree corresponds to a table cell in the right part of the tree table.

```
IlvDataColumnInfo amount2004Column =
    new IlvDefaultDataColumnInfo("Amount 2004", Double.class);
IlvDataColumnInfo amount2005Column =
    new IlvDefaultDataColumnInfo("Amount 2005", Double.class);
IlvDataColumnInfo growthColumn =
    new IlvDefaultDataColumnInfo("Growth", Double.class);
IlvDataColumnInfo[] columns =
    new IlvDataColumnInfo[] {
        amount2004Column, amount2005Column, growthColumn
    };
```

You will also need the indices of the columns:

```
static final int amount2004Index = 0;
static final int amount2005Index = 1;
static final int growthIndex = 2;
```

3. Create the model class.

Now you are ready to create the model. It will access the methods of the `ExpenseItem` class. If you instantiate `IlvDefaultTreeListModel` without any overrides, the storage of the attribute values would be in the `IlvDefaultTreeListModel`. However, since you want the attribute values is derived from the the values of the methods `getAmount2004()`, `getAmount2005()`, and so on, you override the corresponding methods from `IlvDefaultTreeList`.

```
class ExpensesModel extends IlvDefaultTreeListModel {

    public ExpensesModel(IlvDataColumnInfo[] columns) {
        super(columns);
    }
    public double getDoubleAt(Object object, int columnIndex) {
        switch (columnIndex) {
            case amount2004Index:
                return ((ExpenseItem)object).getAmount2004();
            case amount2005Index:
                return ((ExpenseItem)object).getAmount2005();
            case growthIndex:
                return ((ExpenseItem)object).getGrowth();
            default:
                throw new IllegalArgumentException("invalid column");
        }
    }

    public Object getValueAt(Object object, int columnIndex) {
        switch (columnIndex) {
            case amount2004Index:
            case amount2005Index:
            case growthIndex:
              return new Double(getDoubleAt(object, columnIndex));
            default:
                throw new IllegalArgumentException("invalid column");
        }
    }

   public void setDoubleAt(double value, Object object, int columnIndex)
 {
       throw new UnsupportedOperationException("the values are read-only")
;
    }

    public void setValueAt(Object value, Object object, int columnIndex)
 {
       throw new UnsupportedOperationException("the values are read-only")
;
    }
}
```

4. Instantiate the model.

Instantiate the model and fill it with your data.

```
    ExpensesModel model = new ExpensesModel(columns);
    ExpenseItem root = new ExpenseItem("Expenses");
    model.setRoot(root);
    {
        ExpenseItem kind;
        kind = new ExpenseItem("Cost of revenues");
        model.addChild(kind, root);
        {
            ExpenseItem purpose;
            purpose = new ExpenseItem("Cost of manufacturing");
            model.addChild(purpose, kind);
           model.addChild(new ExpenseItem("Materials and Supplies", 580,
 611), purpose);
            model.addChild(new ExpenseItem("Production wages", 101, 104)
, purpose);
            purpose = new ExpenseItem("Cost of maintenance", 271, 324);
            model.addChild(purpose, kind);
            purpose = new ExpenseItem("Cost of training", 110, 80);
            model.addChild(purpose, kind);
            purpose = new ExpenseItem("Cost of consulting", 250, 270);
            model.addChild(purpose, kind);
        }
        kind = new ExpenseItem("Operating expenses");
        model.addChild(kind, root);
        {
            ExpenseItem kind1;
            kind1 = new ExpenseItem("Marketing and Promotion");
            model.addChild(kind1, kind);
            model.addChild(new ExpenseItem("Marketing employee costs",
120, 133), kind1);
           model.addChild(new ExpenseItem("Marketing campaigns", 30, 30)
, kind1);
            kind1 = new ExpenseItem("Sales employee costs");
            model.addChild(kind1, kind);
            model.addChild(new ExpenseItem("Sales employee salaries",
279, 291), kind1);
           model.addChild(new ExpenseItem("Sales commissions", 184, 195)
, kind1);
            kind1 = new ExpenseItem("General & Administrative");
            model.addChild(kind1, kind);
            model.addChild(new ExpenseItem("Finance department", 40, 43)
, kind1);
            model.addChild(new ExpenseItem("Legal department", 11, 12),
kind1);
            model.addChild(new ExpenseItem("Management Information
Systems", 40, 37), kind1);
            {
                ExpenseItem facilities = new ExpenseItem("Facilities");
                model.addChild(facilities, kind1);
                {
                 ExpenseItem office = new ExpenseItem("Office expenses")
;
                    model.addChild(office, facilities);
                    model.addChild(new ExpenseItem("Rental", 38, 40),
office);
                    model.addChild(new ExpenseItem("Electricity", 2, 2),
 office);
                 model.addChild(new ExpenseItem("Other office expenses",
 24, 23), office);
                }
                model.addChild(new ExpenseItem("Telecommunication", 13,
11), facilities);
                model.addChild(new ExpenseItem("Other facilities", 9, 9)
, facilities);
            }
        }
        kind = new ExpenseItem("Financial Interests", 6, 5);
        model.addChild(kind, root);
        kind = new ExpenseItem("Income Taxes", 0, 49);
        model.addChild(kind, root);
    }
```

5. Create the data source.

You need to create a data source view of the model, so that you can connect it to the chart. To do this, use the following code:

```
IlvTreeTableDataSource dataSource = new IlvTreeTableDataSource(model);
```

Creating a treemap chart with renderers

To display a treemap chart, you need an instance of `IlvChart` and an instance of `IlvTreemapChartRenderer` set on the `IlvChart` and connected to your data source.

To do this, you can proceed in two different ways:

1. Allocate the chart and the renderer, and connect the renderer to the data source.

```
IlvChart chart = new IlvChart();
IlvTreemapChartRenderer renderer = new IlvTreemapChartRenderer();
renderer.setDataSource(dataSource);
// Here you can customize the renderer.
chart.addRenderer(renderer);
```

2. Allocate the chart only, and connect the chart to the data source.

```
// Allocate a chart that will show treempas by default.
IlvChart chart = new IlvChart(IlvChart.TREEMAP);
// Connect the chart to the data source. The chart automatically
// creates the appropriate renderer.
chart.setDataSource(dataSource);
// You can fetch the implicitly created renderer and customize it.
IlvTreemapChartRenderer renderer =
                          (IlvTreemapChartRenderer)chart.getRenderer(0);
```

Customizing a treemap chart

Additional customizations of a treemap chart can be found in *Treemap Charts*.

To set the column to use for the area of each item:

♦ Use the following code:

```
renderer.setAreaColumn(amount2005Column);
```

To differentiate the growth of each item through a color:

♦ Use the following code:

```
renderer.setColorColumn(growthColumn);
renderer.setColorScheme(IlvTreemapChartRenderer.
COLORSCHEME_DIVERGING_GREEN_
RED);
```

For example, with regard to expenses, a high growth is alarming and a decrease is agreeable. Therefore, you can choose the color scheme that maps high values to red and low values to green:

To add a legend:

♦ To display the legend at the bottom of the treemap chart area, use the following code:

```
chart.addLegend(new IlvLegend(), IlvChartLayout.SOUTH_BOTTOM);
```

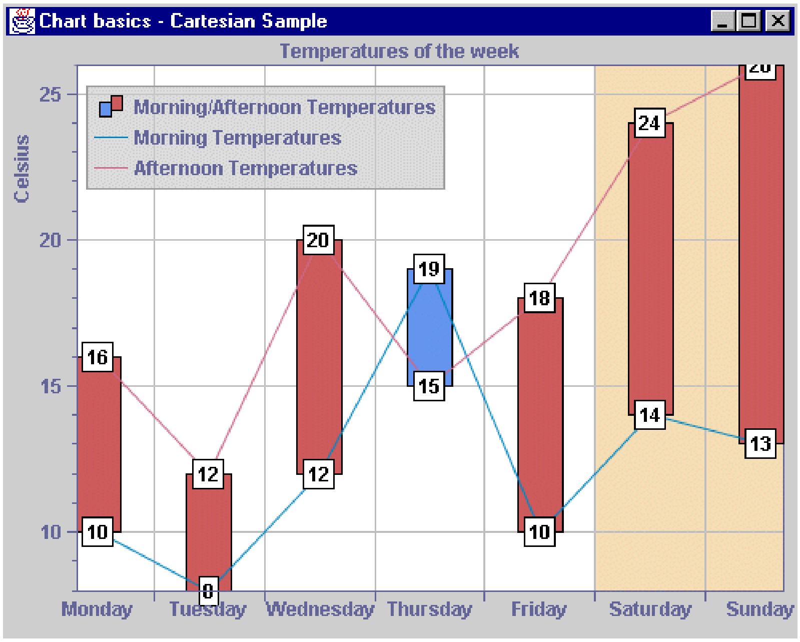
The contents of the legend is automatically determined by the treemap chart renderer, based on the setting of the color column.

In this example, the legend labels represent the growth. The growth values are around 1, but you want to present them as percentage values, so you need to customize the formatting of the legend labels.

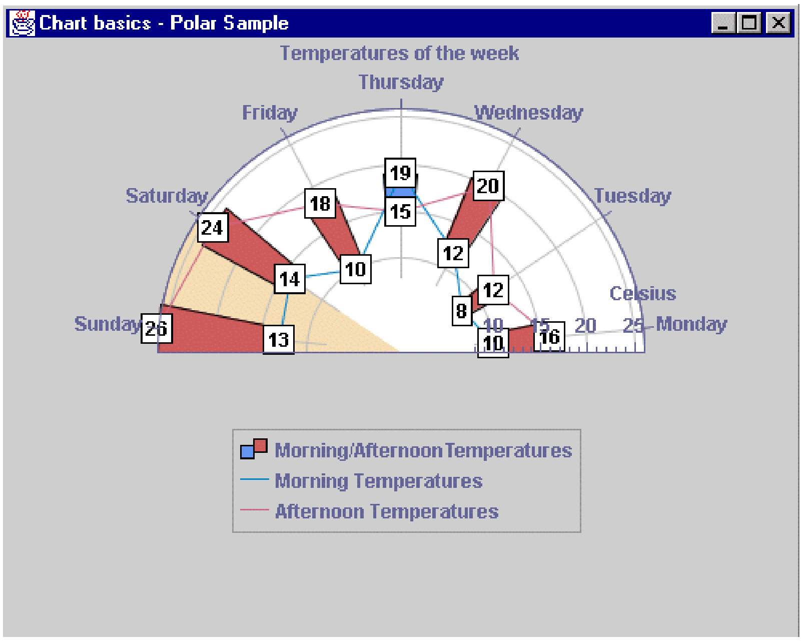
```
renderer.setLegendLabelFormat(
    new DecimalFormat("+##0%;-##0%") {
       public StringBuffer format(double number, StringBuffer result,
                           FieldPosition fieldPosition) {
               return super.format(number-1, result, fieldPosition);
      }
    });
```

Customizing a Chart

The complete source code of the examples can be found in the **`CustomCartesian.java`** file for the Cartesian chart and in the **`CustomPolar.java`** file for the Polar charts.



The Temperatures Cartesian Chart with Additional Customizations



3-D Pie Charts

To add a title at the end of the ordinate scale:

1. Set the scale title to `"Celsius"`, with a rotation angle of 90.

   The ordinate scale represents the temperatures expressed in degrees Celsius. To indicate the unit of the values represented by this scale, you can add the label `"Celsius"` at the end of the axis, rotated in the vertical direction.

2. Set its placement value to 80, meaning that the title location corresponds to 80% of the visible range of the scale (approximately 23):

```
chart.getYScale(0).setTitle("Celsius", -90);
chart.getYScale(0).getTitle().setPlacement(80);
```

To add a label displaying the temperature of each data point:

♦ To add a label annotation to the polylines data points, use the following code:

```
tempRenderer.setDataLabelling(IlvChartRenderer.Y_VALUE_LABEL);
tempRenderer.setAnnotation(new IlvDataLabelAnnotation());
```

You can add label annotations to every data point of the polylines, so that the corresponding temperature values are displayed over them. In the JViews Charts library, a renderer is annotated by means of classes implementing the `IlvDataAnnotation` interface. A data annotation is a graphical annotation that can be either a local annotation, if associated with a data point, or a global annotation if associated with a series or a renderer. The JViews Charts library provides a default implementation to handle label annotations through the `IlvDataLabelAnnotation` class. The text displayed by an instance of this class is computed according to the chart renderer labelling mode. This mode specifies the contents of the label associated with a data point: the text can be either a label, the *y*-value, the *x*-value, or both the *x*- and *y*-values.

To decorate the chart:

♦ Highlight the week-end period by using a data indicator representing a data interval equal to the week-end period:

```
IlvDataInterval inter =
  new IlvDataInterval(chart.getXScale().getStepsDefinition().previousStep
(5),
                              chart.getXScale().getStepsDefinition().
incrementStep(6));
IlvDataIndicator weInd = new IlvDataIndicator(-1, inter,null);
weInd.setStyle(new IlvStyle(Color.black, IlvColor.wheat));
chart.addDecoration(weInd);
```

The JViews Charts package introduces the notion of decoration objects that can be used to add a graphical decoration to a chart. These objects are defined by the `IlvChartDecoration` abstract class and are directly handled by the chart itself. A default concrete implementation of this class is provided in the library via the `IlvDataIndicator` class to graphically represent different kinds of data values: the *x*- or *y*-value, an interval along the *x*- or *y*-axis, or a data window.

To add a title to the chart:

♦ Use a `JLabel` instance as the header, initialized with the title text:

```
JLabel label = new JLabel("Temperatures of the week", JLabel.CENTER);
chart.setHeader(label);
```

The JViews Charts package provides the `setHeaderText(java.lang.String)` and `setFooterText(java.lang.String)` methods as convenience methods to set header (or footer) labels. These methods internally create a `JLabel` instance initialized with the given text and add it to the chart at the corresponding location.

The `IlvChart` component can hold two optional `JComponent` objects at predefined locations: one at the top of the chart area (the header) and the other at the bottom (the footer). These components can be instances of any `JComponent` subclass.

Using the Data Model

Explains the structure of this model as well as the data classes that offer predefined data connectivity.

In this section

Structure of the Extended Data Model
Presents a set of extended data models that are particularly useful to display charts of structured objects.

Predefined Extended Data Model Classes
Describes the five extended data models.

Transforming Data Models
Describes how data models can be transformed into different models referring to the same data.

Structure of the Data Model

The IlvDataSet Interface

The `IlvDataSet` interface defines the methods to access a single set of data points. A data point is defined by an (X, Y) coordinate pair expressed with double primitives, and an optional label. The `IlvDataSet` interface allows you to retrieve information about data points in two different ways:

- *Indexed access*: these requests are based on indexes of data points within the data set. They allow you to retrieve the values or the labels associated with specific data points. The following methods are available:

  `getXData(int)`, `getYData(int)`, `getDataBetween(int, int)`, `getDataLabel(int)`.

- *Spatial requests*: the `getDataInside(ilog.views.chart.IlvDataWindow, int, boolean)` method enables the caller to fetch the data points that are contained within a specified window in the data space.

When you request a set of data points, an instance of `IlvDataPoints` is returned. This object allows you to access both the values of the data points as well as their indices within the data set.

Note: The `IlvDataPoints` instances can directly be projected with the methods defined in the `IlvChartProjector` interface. More information on data projection can be found in *Configuring the Data Projection*.

In addition to the reading operations, a data set defines the following methods to edit its contents:

- `isEditable()` predicate

  Indicates whether the data set supports editing operations.

- `setData(int, double, double)` method

  Allows you to change the values of an existing data point.

- `addData(double, double)` method

  Allows you to append a new data point.

The `IlvDataSet` interface also provides the `getXRange(ilog.views.chart.IlvDataInterval)` and `getYRange(ilog.views.chart.IlvDataInterval)` methods to retrieve the limits of the data values it handles. These methods can return empty ranges if the data set does not contain any data point.

Properties

Each data set handles an open-ended list of properties expressed as key/value pairs. Each property can be added or removed with the `putProperty(java.lang.Object, java.lang.`

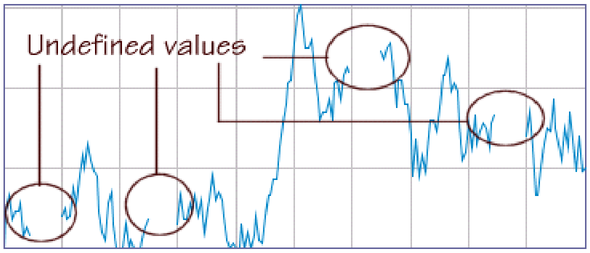
`Object, boolean)` method (some predefined properties can be added or removed through the `IlvDataSetProperty` class). Although this feature should not be seen as an alternative to subclassing, it lets you easily add metadata information to existing data sets.

Undefined Values

In the JViews Charts library, you can specify that a data set holds data points whose values are undefined.

A data point is considered undefined when its y-value is NaN (not-a-number) or equals the value returned by the `getUndefValue()` method. This method actually returns a `Double` instance, which can be `null`. In this case, it is assumed that all undefined data in the data set is indicated by a NaN y-value.

Undefined data points are discarded during the rendering process, as shown in *Undefined Values*:



Undefined Values

The IlvDataSource Interface

The `IlvDataSource` interface behaves as a data set provider, and acts as the bridge between the chart renderers, which define the graphical representation, and the data itself.

The contents of a data source is dynamic and its type depends on the origin of the data (extracted from a Swing TableModel or a JDBC ResultSet, read from an input source, and so on.)

For example, the contents of a data source connected to a table model changes when the structure of the underlying table changes. Likewise, the contents of a data source connected to an XML file changes upon reloading.

The `IlvDataSource` interface defines the methods to access the ordered collection of the provided data sets:

♦ for reading operations:

  `getDataSetCount()`, `getDataSet(int)`, `getDataSets()`

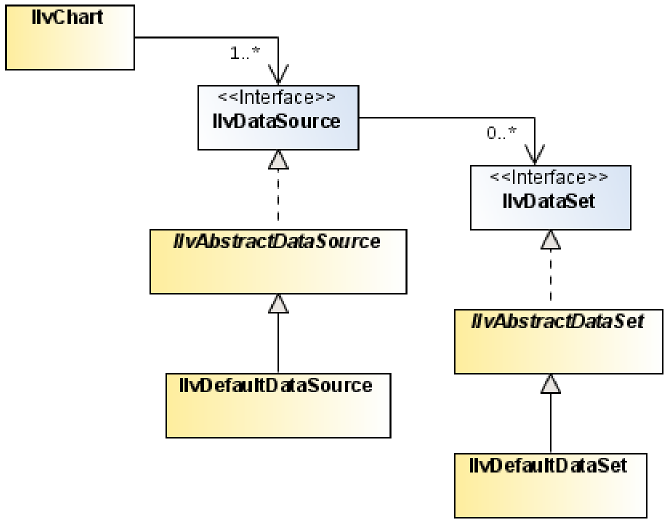
♦ for writing operations:

  `addDataSet(ilog.views.chart.data.IlvDataSet)`, `setDataSet(int, ilog.views.chart.data.IlvDataSet)`,

  `setDataSets(ilog.views.chart.data.IlvDataSet[])`

About data model, data sets and data sources

The JViews Charts library follows the popular Model-View-Controller design pattern by cleanly separating the data model from its graphical representation.



Data Model Structure

The data model distinguishes data series from sources of data. Having distinct entities to represent an elementary set of data and the whole data provides the following benefits:

♦ Data sets exist as objects rather than internal references within the data model, which makes it easier to reference and use them in an application. This also makes the API more understandable and easier to use.

♦ You can create new data set types and use them with existing data sources. Likewise, you can create new data source types that handle existing data sets.

♦ You can easily mix data sets that come from different sources (for example, data sets extracted from a database query with data sets whose values are updated by a thread).

You can create data sets as combinations or wrappers of other data sets (for more information on this feature, see *Data set combination*).

Predefined Data Classes

Describes the various classes available in the `ilog.views.chart.data` package and its subpackages, and how to use them to display your application data.

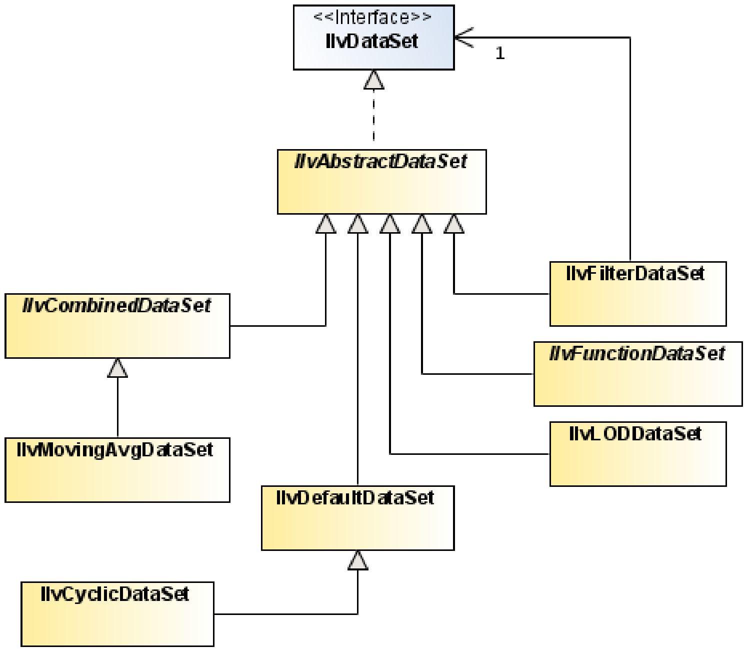
In this section

Data set classes

This section describes the various data set classes and explains how to use them.

Data set classes hierarchy

Illustrates the data set class hierarchy.



Data Set Classes Hierarchy

Abstract implementation

The `IlvAbstractDataSet` class provides default implementations for most of the methods in the `IlvDataSet` interface. For example, it provides support for listeners, properties handling, access to data limits, and spatial requests.

To create a concrete data set as a subclass of `IlvAbstractDataSet`, you need to provide an implementation for the following methods:

```
public abstract int getDataCount();
public abstract double getXData(int idx);
public abstract double getYData(int idx);
```

In other words, you have to provide the indexed access to data points. As `IlvAbstractDataSet` defines no-op methods for writing operations, the `setData(int, double, double)` and `addData(double, double)` methods need to be overridden as if you want your data set to be editable.

The `IlvAbstractDataSet` class is usually the preferred choice to write a custom data set, as it greatly simplifies the work that needs to be done. However, there may be cases with more suitable alternatives, for example:

♦ You need to control the implementation of all methods defined in the `IlvDataSet` interface so that the result matches more closely the structure of your original data.

♦ The data set that you want to implement can be expressed as a combination of values stored in existing data sets. In that case the `IlvCombinedDataSet` class can be the best choice. More information on the `IlvCombinedDataSet` class can be found in section *Data set combination*.

♦ Your data set is just an extension of one of the provided concrete implementations (for example `IlvDefaultDataSet`).

In-memory implementation

The `IlvDefaultDataSet` class provides a concrete data set implementation, where data points are stored in memory with arrays of double primitives. This class supports writing operations such as appending a new data point or changing values of an existing data point.

By using the `IlvDefaultDataSet` class, you can specify whether *x*-values should be stored in memory or computed according to the indices of data points (such data sets are also called category data sets, because *x*-values correspond to a category number). You can also specify whether the array provided to initialize the data set contents should be copied.

Here are several examples of in-memory data sets creation.

```
// Create an empty data set that stores x-values.
IlvDataSet ds1 = new IlvDefaultDataset("DS1");

// Create a category data set and initialize it with a copy of
// the specified y-values.
double[] yValues = new double[] {3., 1., 4.5, 2., 7., 6.3};
IlvDataSet ds2 = new IlvDefaultDataset("DS2", yValues);

// Create a set and initialize the specified x-values and y-values
// arrays.
double[] xValues = new double[] {1., 2., 4., 6., 7., 8.};
double[] yValues = new double[] {3., 1., 4.5, 2., 7., 6.3};
IlvDataSet ds3 = new IlvDefaultDataset("DS3", xValues, yValues, false);
```

Alternatively, you can create data sets from an *N*-dimensional array of doubles with the `create(double[][], int, java.lang.String[], java.lang.String[])` method:

```
double[][] data = new double[][]
{
{1.8, 2., 2.7, 4.5, 4.8, 2.8, 2., 2.2, 3.3, 3.5, 2.2, 1.8},
{2.1, 1., 6.6, 6.8, 8.0, 2.4, 3., 1.5, 1.5, 0.7, 0.7, 2.2},
{0.9, 0.9, 2.3, 3., 2.1, 3.4, 3.8, 5.1, 1.5, 6., 5.5, 0.4},
{1.4, 0.4, 3.8, 2.7, 6.9, 1., 0.7, 1., 2.3, 2.2, 2.4, 2.5}
};

String[] names        = { "Norm", "1998", "1999", "2000"};String[] dataLabels
   = { "Jan", "Feb", "Mar", "Apr", "May", "Jun",
                        "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"};

// Create data sets from the specified data values, data labels,
// and data set names.
IlvDataSet[] dataSets = IlvDefaultDataSet.create(data, -1, names, dataLabels)
;
```

Fixed-size storage

The `IlvCyclicDataSet` class is a subclass of `IlvDefaultDataSet` that allows only a limited number of data points to be stored in memory. When new data is appended beyond this limit, the oldest values are removed so that the cardinality remains the same.

This class is particularly useful for real-time charting where new data comes in a constant stream, and only a restricted history of past values should be kept.

The following restrictions apply when using the `IlvCyclicDataSet` class:

♦ The visible range of the *x*-axis of any chart displaying a cyclic data set must be contained within the *x*-limits of this data set.

♦ Data indices are mutable in a cyclic data set. This means that objects like rendering hints or annotations cannot refer to a data point by its index (more information on rendering hints and annotations can be found in *Handling Chart Renderers*).

Example: Using Cyclic Data Sets

The complete source code of this example can be found in **`<installdir>/jviews-charts86/codefragments/chart/fixed-size-storage/src/FixedSizeStorage.java`**.

The following code extract initializes an `IlvCyclicDataSet` with a buffer size equals to the value of the VISI_COUNT constant:

```
// Create the data sets. The data sets containing the random values are
// instances of IlvCyclicDataSet with a buffer size equal to VISI_COUNT and
// no x values storage.
inputData = new IlvCyclicDataSet("Input", VISI_COUNT, false);
chart = createChart();
IlvSingleChartRenderer r = createRenderer(chart);
chart.addRenderer(r, inputData);
```

The code below shows how new data points are added to the `CyclicDataSet`. In order to minimize the number of notifications, the `addData()` calls are wrapped between a `startBatch()/endBatch()` sequence:

```
void addData()
{
    inputData.startBatch();
    for (int j=0; j<UPDATE_COUNT; ++j) {
        inputData.addData(0, RandomGenerator.rand(counter));
        ++counter;
    }
    inputData.endBatch();
}
```

Data set combination

The `IlvCombinedDataSet` abstract class allows you to define a data set that can be expressed as a combination of one or several values from one or several data sets. Here are two examples.

Example: Creating a Data Set as the Moving Average of Another Data Set

The moving average of a data set is implemented by the `IlvMovingAvgDataSet` class. The following code shows how to create a data set as the moving average of another data set using this predefined class:

```
IlvDataSet ds = ...; // original data set
IlvDataSet movingAvg = new IlvMovingAvgDataSet(ds, 10);
```

Example: Displaying the Average of Two Data Sets

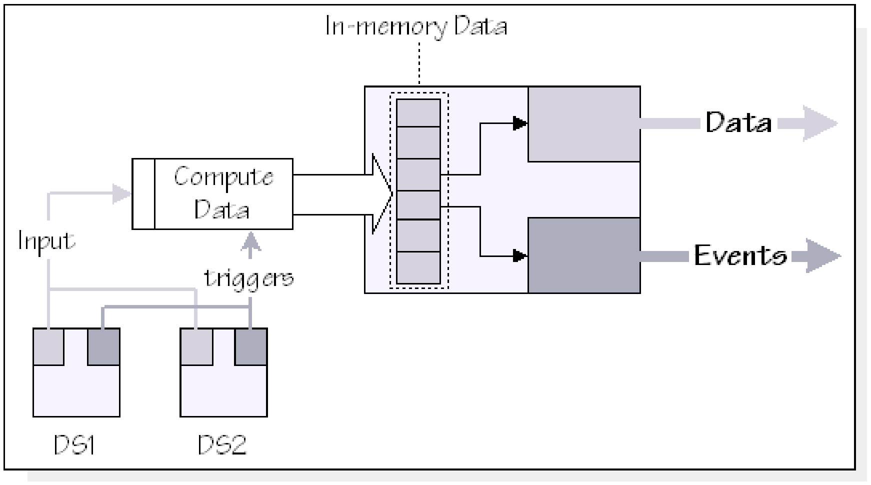
For this example, two solutions are proposed:

♦ First solution

  The complete source code of this solution can be found in **`<installdir>/jviews-charts86/samples/listener/index.html`**.

  Compute the corresponding average data points and store them in an in-memory data set (for example `IlvDefaultDataSet`). Then, we need to explicitly register listeners to the original data sets so that we can update our in-memory data set (modify data points or append new ones).

  Combine Data Set (1) illustrates this first solution:

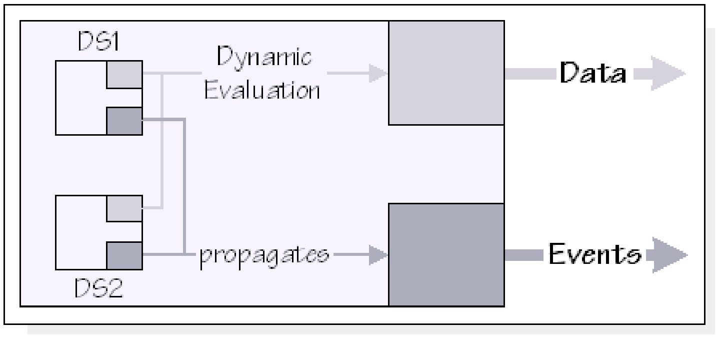


Combine Data Set (1)

♦ Second solution

The complete source code of this solution can be found in **`<installdir>/jviews-charts86/samples/minmax/index.html`**.

Create a subclass of `IlvCombinedDataSet` that references the original data sets and dynamically computes the average whenever the values of data points are queried. *Combine Data Set (2)* illustrates the second solution:



Combine Data Set (2)

The following code extract shows an implementation of such a solution. The `IlvCombinedDataSet` subclass overrides the `getYData(int idx)` method to compute the value taking into account the points of the specified index of all datasets handled by this combined data set.

```
/**
 * Returns the data resulting from an operation on the specified series.
 */
```

```
    abstract protected double getData(double[] values);

    /**
     * Returns the y value of the data point at the specified index.
     */
    public double getYData(int idx)
    {
        int count = getDataSetCount();
        double[] values = new double[count];
        for (int i=0; i<count; ++i)
            values[i] = getDataSet(i).getYData(idx);
        return getData(values);
    }
```

This implementation actually delegates the value calculation to the abstract `getData(double [] values)` method that subclass should implement.

For example, to compute the average of the values, the implementation would be;

```
        protected double getData(double[] values)
        {
            double total = 0.;
            for (int i=0; i<values.length; ++i) {
                total += values[i];
            }
            return total/values.length;
        }
```

Compared to the first solution, the advantage of using a combined data set is twofold:

- ♦ The data set combinations implicitly listen to the changes on the original data sets, so that they can send appropriate change events.
- ♦ The generated data is dynamically evaluated, which saves memory.

The use of `IlvCombinedDataSet` is only suitable when the data can be calculated from the original data sets. For example, this is not the case if we want to represent the maximum value taken by data points across time.

In this situation, we have to keep track of former values and cannot rely only on the values available at a given time.

When the choice does exist between the two solutions, the following issues must be considered:

- ♦ The tradeoff between saving memory and the overhead of computing the values dynamically. Sometimes, the expense of computing the values for complex operations outweighs the savings in memory.
- ♦ Is the context dynamic or static? If the contents of the original data sets changes, you benefit from the implicit subscription made by `IlvCombinedDataSet`. In this case, the use of dynamic evaluation is also justified, rather than storing pre-computed values. Likewise, using `IlvCombinedDataSet` has an advantage if the behavior of the created data set should be dynamic (for example, it is implemented as a mutable function of the values in the original data sets).

Function Implementation

The `IlvFunctionDataSet` abstract class is designed to represent a data set whose *y*-values are computed by a function call.

To create a concrete data set as a subclass of `IlvFunctionDataSet`, the `callFunction (double)` method must be implemented to perform the desired calculation. Function data sets are then instantiated by providing a definition domain and a number of examples.

Example: Data Set Representing the Cosine Function With a Point Every Degree

The complete source code of this example can be found in **`<installdir>/jviews-charts86/samples/logarithm/index.html`**.

Data Set Representing the Cosine Function With a Point Every Degree

```
IlvDataSet ds =
   new IlvFunctionDataSet(0, 2*Math.PI, 361){
     public double callFunction(double val){
       return Math.cos(val);
      }
};
```

The `IlvFunctionDataSet` class is primarily intended to represent mathematical functions, and has the advantage of not storing values into memory.

Load-On-Demand Data Set

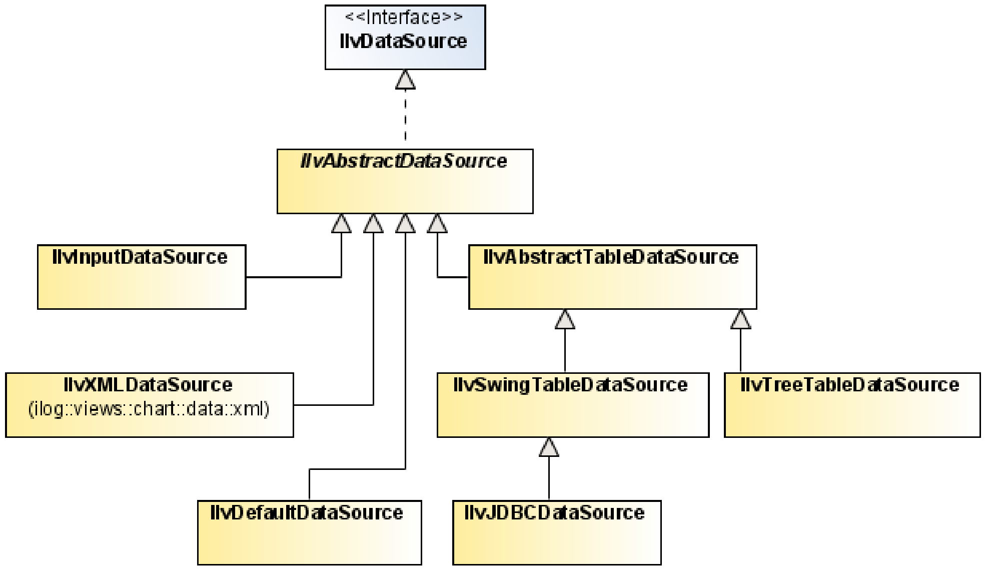
The `IlvLODDataSet` class implements a data set whose contents is loaded on demand. For more information on this feature, please refer to *Using Load-On-Demand*.

Data Source Classes

This section describes the various data source classes and explains how to use them.

Data source classes hierarchy

Illustrates the data source class hierarchy.



Data Source Classes Hierarchy

Abstract implementation

The `IlvAbstractDataSource` class stores the list of accessible data sets, and provides the reading methods to access this collection. It also manages a list of listeners that are notified of changes in the data source contents.

Note: No method of this class is abstract. Concrete subclasses will usually implement the data set creation, and use the `initDataSets` protected method to initialize the data source contents. Another alternative is to modify directly the list returned by the `getDataSetList()` protected method. All the writing methods have an empty implementation that needs to be overridden if the contents is editable from the outside.

Editable data source

The `IlvDefaultDataSource` class is a direct subclass of `IlvAbstractDataSource` that provides implementations for editing operations. It behaves as an editable data source where data sets can be added or removed explicitly, and can be used to access data that comes from different sources. This class is used by default by the chart renderers.

For more information, refer to *Creating the data model*.

Reading data from an input source

The JViews Charts library defines `IlvInputDataSource` as a general-purpose class for reading data from an input source. An input source can be of two forms:

- generic `java.io.InputStream` object
- URL

A data reader (defined by the `IlvDataReader` interface) is used to decode the data stored in the input source. This data reader is provided as parameter to the two `load` methods of the `IlvInputDataSource` class:

```
public void load(InputStream in, IlvDataReader reader) throws Exception
public void load(String url, IlvDataReader reader) throws Exception
```

By calling one of these `load` methods, you reinitialize the contents of the data source with the data sets extracted by the reader. The `IlvDataReader` interface defines the following methods to load data from the input source:

```
public IlvDataSet[] read(InputStream in) throws Exception
```

```
public IlvDataSet[] read(String url) throws Exception
```

You use `IlvDataInputSource` by writing a reader for each specific data format.

The JViews Charts library provides the following examples of custom data readers:

Example: stock

This example shows how to define a reader that reads quote values in CSV (Comma Separated Value) format.

The complete source code of this example can be found in **`<installdir>/jviews-charts86/samples/stock/src/shared/CSVDataReader.java`**.

Example: load-on-demand

This example shows how to read data stored as 32-bit integers from a binary stream.

The complete source code of this example can be found in **`<installdir>/jviews-charts86/samples/lod/src/lod/BinaryInt32Reader.java`**.

`I`n this example, you create an instance of your reader and use it with either an`InputStream` or a URL description. Both parameters are provided to the `IlvInputDataSource` constructor, which loads the data automatically.

The following code is extracted from the Load-On-Demand example and shows how to use a custom reader with an `IlvInputDataSource`.

Using a Custom Reader with an IlvInputDataSource

```
InputStream in = null;
try {
  // Create the reader.
  IlvDataReader reader = new BinaryInt32Reader();
  IlvInputDataSource sampleDataSource = null;
  if (isApplet()) {
    // Load from a URL.
    String url = getDocumentBase() + "sampleData.dat";
    sampleDataSource = new IlvInputDataSource(url, reader);
  } else {
    // Create an InputStream from a file.
    in = new BufferedInputStream(
           new FileInputStream(
             new File(System.getProperty("user.dir"),
                                        "sampleData.dat")));
    sampleDataSource = new IlvInputDataSource(in, reader);
  }
  overviewChart.setDataSource(sampleDataSource);
} catch (Exception x) {
  x.printStackTrace();
  System.err.println(x.getMessage());
} finally {
  if (in != null) try {in.close();} catch (Exception x) {}
}
```

Note: The `InputStream` object must be explicitly closed after the loading is performed, whereas connection and disconnection are automatically handled when using a URL.

Reading and writing data from an XML source

The JViews Charts library lets you export or import data in a predefined XML-based format. The `ilog.views.chart.data.xml` package contains all the classes that are related to XML serialization:

♦ `IlvXMLDataReader`, creates the data sets by interpreting the contents of an XML file.

♦ `IlvXMLDataWriter`, serializes the data sets in an XML file.

♦ `IlvXMLDataSource`, implements a data source dedicated to XML input.

In order to use the `ilog.views.chart.data.xml` package with Java™ SE 5, you need Apache™ Xerces 2.4.0 or higher in your `CLASSPATH`. A copy of Apache Xerces can be found in the `<installdir>/jviews-framework86/lib/` directory.

Reading data from an XML document

The `IlvXMLDataReader` class implements the `IlvDataReader` interface and can be used with the general-purpose `IlvInputDataSource` class.

The following code shows how to read the contents of an XML file, and make it available for charting through an `IlvXMLDataSource`:

```
// Create the data source.
IlvXMLDataSource ds = new IlvXMLDataSource();
// Create the XML reader.
IlvXMLDataReader reader = new IlvXMLDataReader();
// Optional: Specify that the parser should validate the contents of the file
reader.setValidating(true);
// Load the data.
ds.load(new org.xml.sax.InputSource("data.xml"), reader);
```

Writing data to an XML document

The `IlvXMLDataWriter` class allows you to write data sets into an XML document, as shown in the following code:

```
import javax.xml.parsers.*;
import org.w3c.dom.Document;
...
IlvDataSet[] dataSets = ...;
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
Document document       = builder.newDocument();
IlvXMLDataWriter writer = new IlvXMLDataWriter();
writer.write(document, dataSets);
```

You can also directly serialize the data sets into an output stream, as shown in the next example:

```
IlvDataSet[] dataSets = ...;
IlvXMLDataWriter writer = new IlvXMLDataWriter();
writer.write(new FileOutputStream("chartmodel.xml"), dataSets);
```

Example: serializing data

A complete example that shows how to import and export data using XML can be found in **`<installdir>/jviews-charts86/codefragments/chart/xml-serialization/src/XMLSerialization.java`**.

Description of the format

The expected format is an application of the W3C XML language. You can find the full Document Type Definition of this format in Document Type Definition for XML data file in *Using the Designer*. For more information see XML File Format.

The data is described with the following elements:

The `chartData` element, which belongs to the `ilvchart` namespace:

```
<!ELEMENT chartData (data+)>
<!ATTLIST chartData xmlns:ilvchart CDATA #FIXED
                      "http://www.ilog.com/products/jviews/chart"
                    version CDATA #REQUIRED>
```

This root element contains a set of child `data` elements:

```
<!ELEMENT data (labels?,series+)>
<!ATTLIST data  xSeries IDREF #IMPLIED>
```

Each `data` element can be seen as a table where rows are represented by one or more `series` elements. One of these series can be identified as the one holding the *x*-values. In this case, the other series are assumed to hold the *y*-values of the considered data.

Series data points might be associated with labels by means of the `labels` element. Labels can be either defined common to all the series of a `data` element, or individually for each series. In this case, the labels are specified at the `series` element level, and override `labels` that may have been set on the `data` element.

The `series` element has the following description:

```
<!ELEMENT series ((valueOperator | (value | valuesList)*),labels?,property*)>
<!ATTLIST series dateFormat CDATA           #IMPLIED
                 type (double | date)       #REQUIRED
                 id         ID              #REQUIRED>
```

Each series is identified by a unique ID and the type of data it contains (either double values or dates). The `dateFormat` attribute can be any pattern that conforms to the syntax used by the `java.text.SimpleDateFromat` class. Values can be expressed with elementary `value` elements, or as a list with the `valuesList` element:

```
<!ELEMENT value (#PCDATA)>
<!ATTLIST valuesList delimiter CDATA #IMPLIED>
```

The `delimiter` attribute specifies the separating character in the values list. The default character is a comma. Here is an example of a series element:

```
<series id="Series_1" type="double">
  <value>0.5</value>
  <valuesList>2.0,8.0,6.0,13.0,22.0,21.0,19.0,28.0,27.0
  </valuesList>
  <value>32.5</value>
</series>
```

You can also express the values of a series as an operation on the values of other series. This is performed by means of the `valueOperator` element:

```
<!ELEMENT valueOperator (seriesRef|property)*>
<!ATTLIST valueOperator    class NMTOKEN #REQUIRED>
```

For example, the following series is defined as the moving average of the "Series_1" series:

```
<series id="Series_1_Mov._Avg." type="double">
  <valueOperator class="ilog.views.chart.data.IlvMovingAvgDataSet">
    <seriesRef ref="Series_1"/>
    <property name="period">5</property>
  </valueOperator>
</series>
```

The `class` attribute refers to the class name of the data set that implements the operation. The `seriesRef` element refers to a valid series identifier within the same `data` element.

A `property` element is defined as follows:

```
<!ENTITY % propertyExt "">
<!ELEMENT property (#PCDATA %propertyExt;)*>
<!ATTLIST property name  CDATA #REQUIRED
                   javaClass CDATA #IMPLIED>
```

By default, a `property` element consists of text data. A property is defined by its name (the name attribute) and optionally by the Java class name of the Java object it refers (the `javaClass` attribute). The use of the `javaClass` attribute is explained in the next section.

The entity `propertyExt` can be defined in the internal DTD subset to add custom subelement, or custom attributes to the `property` element within a given document.

For example, the following lines extend the `property` element with additional child elements `myelement`:

```
<!DOCTYPE chartData SYSTEM 'chartxml.dtd'
[
<!ENTITY % propertyExt "|myproperty">
<!ELEMENT myproperty (myelement)*>
<!ELEMENT myelement .... myelement definition goes there ".>
]>
```

The properties can be used with `valueOperator` and `series` elements. A default mechanism allows you to specify simple properties for `valueOperator` elements. This mechanism uses reflection to determine which method should be called on the data set instance, as well as the expected parameter type. The method is found according to the Java Bean naming convention. For example, in the definition of your moving average operator, the `setPeriod (int)` method is called:

```
<valueOperator class="ilog.views.chart.data.IlvMovingAvgDataSet">
  <seriesRef ref="Series_1"/>
  <property name="period">5</property>
</valueOperator>
```

Extending the XML Reader

In this section you will see how to extend the reader to deal with series properties, as well as with complex or extended properties. The `IlvXMLDataReader` class allows you to register readers that will be used to interpret custom properties within the XML file.

A property reader must implement the `IlvXMLPropertyReader` interface, which defines the following methods:

- `readProperty(org.w3c.dom.Element)`, used to read the property from the specified DOM element.
- `setProperty(ilog.views.chart.data.IlvDataSet, java.lang.String, java.lang.Object)`, used to associate the property with the corresponding data set.

By default, a predefined reader is used to decode the property value. This predefined reader decodes the property value according to the following rules:

- if the `javaClass` attribute is set, then the XML property value is converted into the corresponding Java class using a `java.beans.PropertyEditor`.
- if the `javaClass` attribute is missing, or if the above conversation has failed, then the Java property value is the string representation of the XML property value.

You can override this default mechanism by registering your own property reader. A reader is registered by means of the `IlvXMLDataReader.registerPropertyReader` (for readers to be shared between all instances) and `IlvXMLDataReader.setPropertyReader` (for readers specific to an instance) methods.

For example:

- to register a specific property, use the following code:

  ```
  registerPropertyReader(java.lang.String, ilog.views.chart.data.xml.
  IlvXMLPropertyReader)
  ```

  and

  ```
  aReader.setPropertyReader(aPropertyName, aReader);
  ```

- to register a default reader that is used to interpret properties with no associated reader, use the following code:

  ```
  registerPropertyReader(java.lang.String, ilog.views.chart.data.xml.
  IlvXMLPropertyReader)
  ```

  ```
  aReader.registerPropertyReader(null, aReader);
  ```

When a property is read, the `IlvXMLDataReader` searches for the corresponding property reader in the following order:

- in the reader repository of the reader instance, then
- in the reader repository of the `IlvXMLDataReader` class.

If no reader has been registered for a property, then the `IlvXMLDataReader` searches for a default property reader in the following order:

♦ the default property reader of the reader instance, then

♦ the default property reader of the `IlvXMLDataReader` class.

If no property reader has been found, then the predefined reader is used as a fallback.

The following are examples of properties element read by the predefined readers:

```
<property name="product">JViews Charts</property>
property name="color" javaClass="java.awt.Color">red</property>
<property name="dataLabels">
Main classes,renderer,data,interactor,swing,servlet,java2d,util,graphic,other

</property>
```

They are respectively interpreted as the “JViews Charts” string, the `java.awt.Color.red`, and the last one is as the data labels of the data set.

The following example shows how to use this mechanism. The complete source code can be found in **`<installdir>/jviews-charts86/codefragments/chart/xml-extension/src/XMLExtension.java`**.

You want to associate a URL for each point of a data set. These URLs will be specified in the XML file as a property of the series. To support this new custom property, you first have to extend the JViews Charts DTD:

```
<!DOCTYPE chartData PUBLIC '-//ILOG//JVIEWS/Chart 1.0' 'chartxml.dtd'
[
<!ENTITY % propertyExt "| hrefs">
<!ELEMENT hrefs (#PCDATA)>
]>
```

This extension means that the property element now contains a `hrefs` element that consists of PCDATA.

Then, you add the property to the series:

```
<chartData version="1.0">
  <data>
    <series id="Series1" type="double">
      <valuesList>353.2,191.6,160.7,54.5,36.6,34.3,31.3,28.1,25.5,45.2
      </valuesList>
      ...
      <property name="hrefs">
<hrefs>
../../../../doc/refman/ilog/views/chart/package-summary.html;
../../../../doc/refman/ilog/views/chart/renderer/package-summary.html;
../../../../doc/refman/ilog/views/chart/data/package-summary.html;
../../../../doc/refman/ilog/views/chart/interactor/package-summary.html;
../../../../doc/refman/ilog/views/chart/swing/package-summary.html;
../../../../doc/refman/ilog/views/chart/servlet/package-summary.html;
```

```
../../../../doc/refman/ilog/views/chart/java2d/package-summary.html;
../../../../doc/refman/ilog/views/chart/util/package-summary.html;
../../../../doc/refman/ilog/views/chart/graphic/package-summary.html;
../../../../doc/refman/ilog/views/chart/package-summary.html
</hrefs>
</property>
    </series>
  </data>
</chartData>
```

The next step is to implement an `IlvXMLPropertyReader` to read your custom property: class `HREFPropertyReader` implements `IlvXMLPropertyReader`.

```
{
    /** The <code>hrefs</code> property element tag. */
    public static final String HREFS_TAG = "hrefs";

    /**
     * Reads the specified property element.
     * This method reads an <code>hrefs</code>
      *  element associated with a series and stores its contents in a
     * <code>List</code>.
     */
    public Object readProperty(org.w3c.dom.Element propertyElt)
    {
        Node child = propertyElt.getFirstChild();
        while (child != null) {
            if (child.getNodeType() == Node.ELEMENT_NODE &&
                child.getNodeName().equals(HREFS_TAG)) {
                Element hrefElt = (Element)child;
                StringTokenizer tokenizer =
                    new StringTokenizer(hrefElt.getFirstChild().getNodeValue
(),
                                ";\n\t ");
                List hrefs = new LinkedList();
                while (tokenizer.hasMoreTokens())
                    hrefs.add(tokenizer.nextToken());
                return hrefs;
            }
            child = child.getNextSibling();
        }
        return null;
    }

    /**
     * Sets the property on the specified data set. This method sets the
    * <code>href</code>s <code>List</code> as a property
    *  of the specified data set.
    */
    public void setProperty(IlvDataSet dataSet,
                            String propertyName,
                            Object value)
    {
        dataSet.putProperty(HREFPropertyReader.HREFS_TAG, value, false);
    }

}
```

Finally, you register this property reader on your `IlvXMLDataReader`:

```

// Register our own XMLPropertyReader
reader.setPropertyReader(HREFPropertyReader.HREFS_TAG,
                              new HREFPropertyReader());
```

The XML reader can also be extended to create instances of custom data sets. This is performed by overriding the `createDataSet(java.lang.String, double[], double[])` method. By default, this method returns an instance of the `IlvDefaultDataSet` class.

Database access through JDBC

JDBC technology offers a platform and server independent way to retrieve data stored in a database. The requests are expressed through the JDBC API, and are usually performed in three steps:

♦ Establish the connection to the server.

♦ Execute a database statement (SQL query).

♦ Process the result of the request. This result is available through a `java.sql.ResultSet` object, which presents a tabular structure.

Database processing is actually performed by a driver, which depends on the type of the database server. You can find more information about JDBC on the JavaSoft site at: *http://java.sun.com/products/jdbc*. This site also contains information about driver availability for the most popular database vendors.

The JViews Charts library supports the use of the JDBC interface to retrieve data values from database servers. This support is provided by the `IlvJDBCDataSource` class, which extracts data sets from the result of a database query.

You can specify the information related to the database request as follows:

♦ Provide directly the result of the query in the form of a JDBC `ResultSet`, either in the constructor of the data source, or with the `setResultSet(java.sql.ResultSet)` method.

♦ Provide the connection parameters as well as the SQL query statement. This can be done either in the constructor of the data source, or with the corresponding *getter* and *setter* methods. If you use this methodology, you must call the `executeQuery()` method to produce the resulting `ResultSet`.

You can control how data sets are extracted from the `ResultSet` by means of two parameters:

♦ The index of the column that holds the *x*-series.

  If this index is set to -1, the category data sets will be created.

♦ The index of the column that holds the data labels.

  If this index is set to -1, no data label is defined.

By default, the `IlvJDBCDataSource` class creates data sets that store the result of the query into memory without any binding to the database. In other words, this means that the data accessible from these data sets is not bound to the data stored in the database. For example, modifying the value of a data point with the `IlvDataSet.setData` method does not send an update statement to the database.

Since JViews 6.5, in addition to this read-only mode, the `IlvJDBCDataSource` also support a read-write mode in which any changes on the data model are committed to the data base. This read-write mode must be enabled at construction time by means of the following constructors:

```
IlvJDBCDataSource(boolean readOnly)
```

and

```
IlvJDBCDataSource(String databaseURL,
                  String user,
                  String passwd,
                  String driverName,
                  String query,
                  int xColumnIndex,
                  int dataLabelsColumnIndex,
                  boolean readOnly)
```

Example: Importing the Contents of a Microsoft Excel Worksheet through the JDBC ODBC Driver

The complete source code of the example can be found in **`<installdir>/jviews-charts86/codefragments/chart/jdbc/src/JDBCConnection.java`**.

```
        // Initialize the data source. The syntax of the database url
        // is database specific. The user and password are left blank, the
        // driver to use is the JDBC-ODBC driver
("sun.jdbc.odbc.JdbcOdbcDriver"),
        // the query is to be initialized later, and there are no x series
        // or data labels to read (-1 = no x series column).
        String dburl =  "jdbc:odbc:Driver={Microsoft Excel Driver
(*.xls)};DBQ=xlsdemo.xls";
        IlvJDBCDataSource jdbcDs =
            new IlvJDBCDataSource(dburl, // database url
                                  "",    // user name
                                  "",    // user password
                                  "sun.jdbc.odbc.JdbcOdbcDriver", // driver
                                  null,  // query
                                  -1,    // x-values column index
                                  3);    // datalabels column index

        // Set the query now.
        String query = "select * from [Sheet1$]";
        // Set the query.
        jdbcDs.setQuery(query);
        // Execute the query
        try {
            jdbcDs.executeQuery();
        } catch (SQLException e) {
            System.err.println("A database access error occurs");
        }

        // Create the area renderer.
        IlvAreaChartRenderer renderer = new IlvAreaChartRenderer();
        renderer.setDataSource(jdbcDs);
        chart.addRenderer(renderer);
```

Connecting to a Swing TableModel

The Swing `TableModel` interface defines a tabular data model that can be represented with a `javax.swing.JTable` in the GUI of your application. The `IlvSwingTableDataSource` class allows you to create a data source from an existing `TableModel`.

Note: 1. The `IlvSwingTableDataSource` acts as an adapter between both data models, which means that the original data accessible from the table model is not copied.

2. Both data models are bound, which means that modifications done through a data set are forwarded to the table model. Likewise, modifying the table model fires a change event for the corresponding data set.

There are several ways to indicate how data sets should be extracted from the table model:

♦ By specifying the series type (`ROW_SERIES`, `COLUMN_SERIES`), which indicates whether data sets are extracted in rows or columns.

♦ By specifying optional rows or columns that hold the *x*-series and the data point labels.

♦ By providing data converters to map objects stored in the table into double values. This feature is only used when series are extracted by column as it is based on the information provided by the `TableModel.getColumnClass` method. To register a data converter for a given Object class, use the `setDefaultConverter(java.lang.Class, ilog.views.chart.data.IlvDataConverter)` static method. By default, converters exist for `Date` and `Double`.

Example: Connecting to a TableModel

The complete source code of the example can be found in **`<installdir>/jviews-charts86/codefragments/chart/swingtable/src/TableModelConnection.java`**.

```
// Create the chart.
IlvChart chart = createChart();

// Create the swing TableModel containing the data.
AbstractTableModel swingModel = null;
try {
    swingModel = createSwingTableModel();
} catch (ParseException e) {
    swingModel = new DefaultTableModel();
}
// Bind an IlvSwingTableDataSource to this swing table model. The series
// being arranged by column, the data source is of type COLUMN_SERIES.
// Since a specific column is used for the abscissa (the Year column) for
// all the series, the index in the table model of the year column
// is also specified (0).
IlvSwingTableDataSource tableDs =
    new IlvSwingTableDataSource(swingModel,
                                   IlvSwingTableDataSource.COLUMN_SERIES,

                                0,
// the column index for the x-values
                                -1);
// no datalabels
// At this time, the data sets corresponding to the table model series
// have been created.

// Connect the data source to a polyline chart renderer.
// The IlvPolylineChartRenderer will create a renderer for each data set
// of its data source and hold them in an internal list. These sub-
// renderers are called child renderers and can be parsed using an
// Iterator (see below).
IlvPolylineChartRenderer r = new IlvPolylineChartRenderer();
        r.setDataSource(tableDs);
        chart.addRenderer(r);
```

Converting Data

To be properly merged into a chart data model, data imported from a database or a table model needs to be mapped into double values. This conversion is handled by means of data converters, instances of the `IlvDataConverter` interface.

A data converter handles the conversion between a particular `Object` class and its double representation, by means of the following methods:

```
public Object toObject(double value)
public double toValue(Object object)
```

For example, a data converter implementation that handles conversions between a `String` object and a double representation would be:

```
public Object toObject(double value)
{
    return Double.toString(value);
}
public double toValue(Object object)
{
    if (!(object instanceof String))
        throw new IllegalArgumentException("IlvStringConverter: object not a
java.lang.String instance.");
```

```
    try {
        return Double.parseDouble((String)object);
    } catch (NumberFormatException e) {
        throw new IllegalArgumentException("IlvStringConverter: Cannot parse 
object.");
    }
}
```

The `IlvJDBCDataSource` and `IlvSwingTableDataSource` classes handle a set of data converters common to all instances of these classes. These common converters are called default converters, and are registered using the `setDefaultConverter(java.lang.Class, ilog.views.chart.data.IlvDataConverter)` and `setDefaultConverter(java.lang.Class, ilog.views.chart.data.IlvDataConverter)` static methods.

By default, the `IlvJDBCDataSource` and `IlvSwingTableDataSource` classes register converters for `Date`, `String`, `Short`, `Integer`, `Long`, `Float` and `Double` objects.

Note: The `IlvJDBCDataSource` also registers default converters for `Time` and `Timestamp` SQL data types.

The data converters are retrieved using the `getDataConverter(int)` and `getDataConverter (int)` methods. The default implementation of these methods returns the default converter for the object type of the specified column. If none exists, it returns the first default converter able to convert it:

♦ For the `IlvSwingTableDataSource`, it returns the default converter associated with the column class (see `TableModel.getColumnClass`).

♦ For the `IlvJDBCDataSource`, it returns the default converter associated with the Java-equivalent type of the database column type as defined by the JDBC specification.

You can override this method to change the default implementation, for example if you need to define a per-instance registering mechanism.

The JViews Charts package provides three default implementations of the `IlvDataConverter` interface:

♦ string converter (`IlvStringConverter`),

♦ date converter (`IlvDateConverter`),

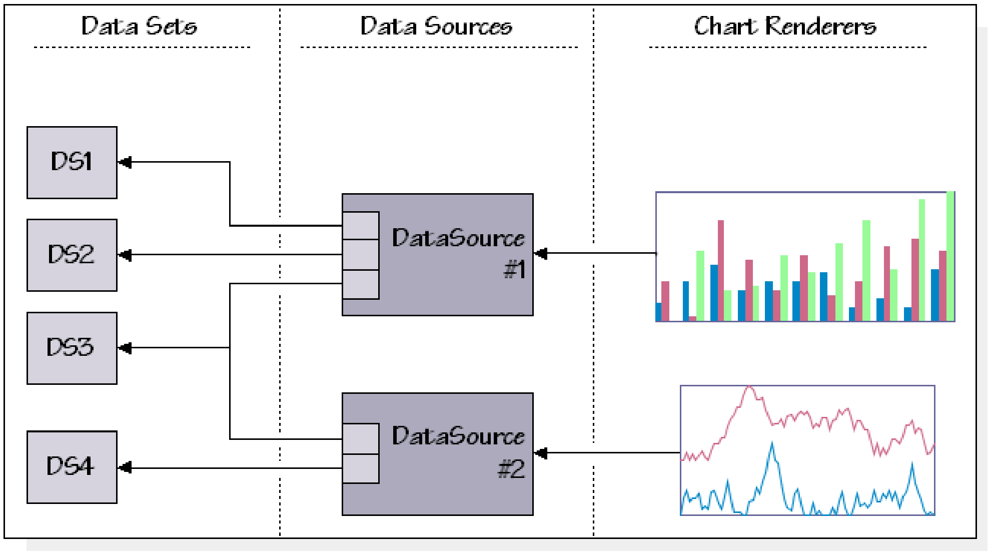
♦ number converter (`IlvNumberConverter`).

Connecting to the Data Model

Connecting to a Chart

The connection between a chart and the data model is performed through a unary relationship between a chart renderer and a data source. For more information on chart renderers, how to connect them to a data source, and how to add them to a chart, refer to *Handling Chart Renderers*.

Data Connection illustrates a simple example of data connection:



Data Connection

Events and Listeners

Both `IlvDataSet` and `IlvDataSource` send events when the underlying data changes. The chart uses implicitly this notification mechanism to update itself whenever the data is modified in a way that affects the display.

You can explicitly add or remove listeners to be notified of changes in the data model with the following methods:

♦ For data source events:

```
addDataSourceListener(ilog.views.chart.event.DataSourceListener)
```

```
removeDataSourceListener(ilog.views.chart.event.DataSourceListener)
```

♦ For data set events:

```
addDataSetListener(ilog.views.chart.event.DataSetListener)
```

```
removeDataSetListener(ilog.views.chart.event.DataSetListener)
```

Two kinds of events are sent by data sets:

- `DataSetContentsEvent` events, describe the changes in the contents of a data set (data added or modified).
- `DataSetPropertyEvent` events, describe the changes in a property of a data set (name, client properties, and so on).

Synchronizing the Contents of Several Data Sets

Your goal is to provide a display of the following measurements for a given day:

- ♦ The temperature at the beginning of the day.
- ♦ The temperature at the end of the day.
- ♦ The highest temperature during the day.
- ♦ The lowest temperature during the day.

1. Create a data set for each of these measures.
2. Store them in a data source. The update of these data sets occurs through a listener on the original temperature data set. This listener either appends new points or modifies the value of existing points in the corresponding data set.

   The complete source code can be found in **`<installdir>/jviews-charts86/samples/listener/index.html`**.

```
// Contains the (High, Low, Start, End) data sets.
protected IlvDataSource hiloDS;

DataSetListener tempListener = new DataSetListener() {
  public void dataSetContentsChanged(DataSetContentsEvent evt)
  {
    if (evt.getType() == DataSetContentsEvent.DATA_ADDED) {
      IlvDataSet ds = evt.getDataSet();
      double x = ds.getXData(evt.getFirstIdx());
      double y = ds.getYData(evt.getFirstIdx());
      int count = ds.getDataCount()-1;
      if (count < 0) count = 0;
      if ((count%24) == 0) {
  // If the point begins a new day, add it to all the hilo data
        // sets (High, Low, Start, End).
        for (int i=0; i<hiloDS.getDataSetCount(); ++i) {
          hiloDS.getDataSet(i).addData(x,y);
        }
      } else {
   //  Else, update the high/low/end values accordingly.

   ...
      }
  }

   public void dataSetPropertyChanged(DataSetPropertyEvent evt) {}
};
```

Extending the Data Model

You can extend the data model by creating new data sets and new data sources. You are going to give an example of a custom model connected directly to existing application data. The complete source code of this example can be found in **`<installdir>/jviews-charts86/codefragments/chart/datamodel-extension/src/DataModelExtension.java`**.

Consider a very simple application context: a university that maintains information about students. Consider the names of the students, and the grades they have acquired. Your goal is to display the students grade for each course. Let us suppose the application provides the following classes:

```
/** Represents a University */
class University
{
  /** Returns the list of undergraduate students */
  java.util.List getUnderGraduateList();
}

/** Represents a Student */
class Student
{
  /** Returns the name of the student */
  String getName() {...}

  /** Returns the grade for the specified course */
  int getGrade(String course){ ... }
}
```

1. Design a data source that extracts grade information from a student list and a set of courses. Each provided data set represents a given course, and contains the grade for every student. Here is the declaration of our `GradeDataSource` class:

```
class GradeDataSource extends IlvAbstractDataSource
{
  private List students;
  public GradeDataSource(List students, String[] courses)
  {
    this.students = students;
    getDataSetList().setDataSets(createDataSets(courses));
  }

  private final Student getStudent(int idx)
  {
    return (Student) students.get(idx);
  }
}
```

The data source stores the student list and initializes its contents with the data sets created from the specified courses. The `IlvAbstractDataSet` class is the easiest starting point when designing a new data set.

2. Use the `IlvAbstractDataSet` as a base class for your custom data set, which you define as a private inner class of `GradeDataSource`:

```
class GradeDataSource extends IlvAbstractDataSource
{
  ...
  private class DataSet extends IlvAbstractDataSet
  {
    private String course;
    DataSet(String course)
    {
      this.course = course;
      IlvDataSetProperty.setCategory(this, new Double(1));
    }

    /** Returns the name of the course */
    public String getName()
    {
      return course;
    }

    /** Returns the number of students */
    public int getDataCount()
    {
      return GradeDataSource.this.students.size();
    }

    /** Simply returns the student index */
    public double getXData(int idx)
    {
      return idx;
    }

    /** Returns the grade of the specified student */
    public double getYData(int idx)
    {
      return GradeDataSource.this.getStudent(idx).getGrade(course);
    }

    /** Returns the name of the specified student */
    public String getDataLabel(int idx)
    {
      return GradeDataSource.this.getStudent(idx).getName();
    }
  }
}
```

The implementation of the data set class is straightforward, as you only need to provide indexed access to the data points:

The *x*-value is equal to the student's index (we have in fact created a *category data set*, which is specified by using the `setCategory(ilog.views.chart.data.IlvDataSet, java.lang.Double)` method in the constructor). The *y*-value is equal to the grade of the student in the course referenced by the data set. The label of a point is equal to the name of the student. The `createDataSets` method of the `GradeDataSource` simply creates a `DataSet` instance for each course:

```
private IlvDataSet[] createDataSets(String[] courses)
{
  IlvDataSet[] dataSets = new IlvDataSet[courses.length];
  for (int i=0; i<courses.length; ++i) {
    dataSets[i] = new DataSet(courses[i]);
  }
  return dataSets;
}
```

3. Use your new data model to display the grades of undergraduate students:

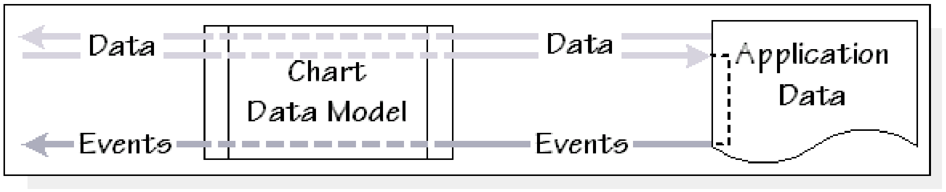
```
IlvChart chart = new IlvChart(IlvChart.RADAR);
University university = ...;
String[] courses = new String[] {"Math", "Physics", "Chemistry"};
IlvDataSource ds =
  new GradeDataSource(university.getUnderGraduateList(), courses);
chart.setDataSource(ds);
```

The `GradeDataSource` class provides a simple yet typical example of making application data compliant with the chart data model. In this example, you have assumed that the data (the grades) was immutable. In the opposite case, additional work has to be done:

♦ Have the chart data model listen to data change in the application, and send the appropriate events.

♦ Optionally, provide a way to perform modifications through the chart data model.

Extend Data Model illustrates a typical connection between application data, and the chart data model:



Extend Data Model

This example highlights one of the main motivations behind writing a custom data model: accessing data directly from the application instead of duplicating it.

There are other cases where you may wish to write your own data sets or data sources. For example, the default in-memory data set implementation available in the library (`IlvDefaultDataSet`) stores the values into arrays of double primitives. If your application is dealing with other primitive types (float, int, byte), writing a new data set can save storage space.

Structure of the Extended Data Model

IBM® ILOG® JViews Charts offers a set of extended data models that are particularly useful to display charts of structured objects.

The extended data models have the following characteristics:

♦ They are particularly appropriate to connect data to a data source when you want to create a treemap chart.

  In theory, you could connect your data directly to the `IlvTreeTableDataSource`, but it is easier to connect it to one of the extended data model classes.

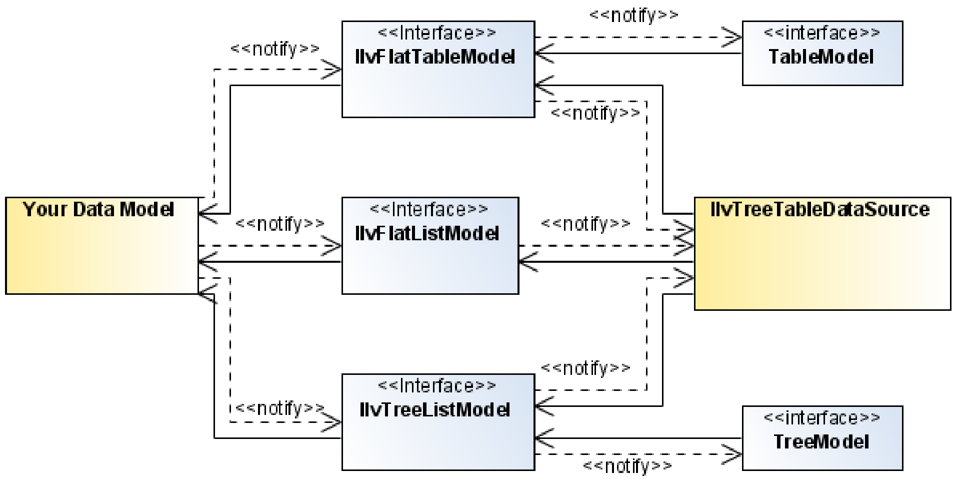
♦ They can hold objects of any type (string or object).

  For example, the `IlvDataSet` interface holds mostly numeric values.

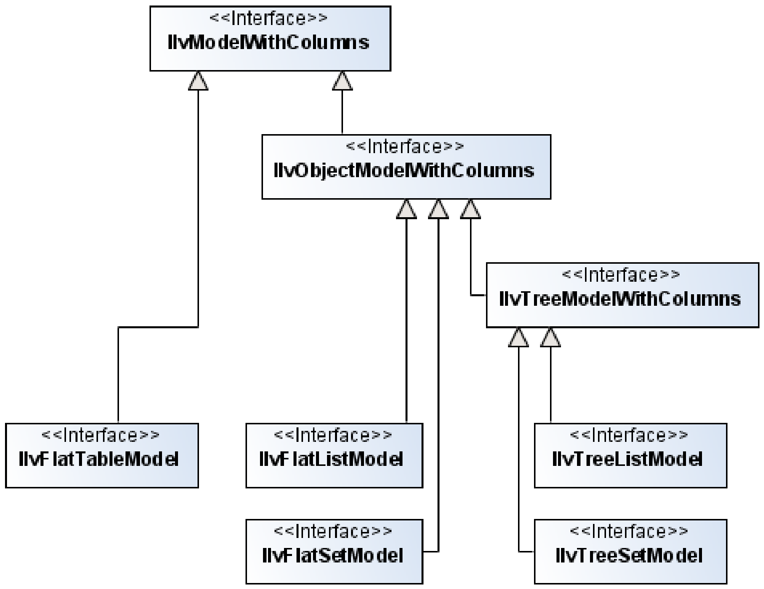
♦ They can be easily connected to Swing models.

  For example, they can be wrapped into Swing `TableModel`, `TreeModel` or `TreeTableModel` instances. This allows you to display the data in your application not only with a Charts view, but also with a Swing view.

  Also, existing instances of Swing `TableModel`, `TreeModel`, `TreeTableModel`, or `ListModel` can be viewed through a facade of extended data models. This allows you to display in a chart any data that is already displayed in a Swing view.



Connecting to Extended Data Models



Extended Data Models

The extended data models are defined in subpackages of the package `ilog.views.chart.datax`.

Extended Data Models lists the available data models and illustrates their main characteristics:

Extended Data Models

Data Model	Structure	Row Entity	Columns or Attributes
`IlvTreeListModel`	tree, ordered	object with attributes	yes
`IlvTreeSetModel`	tree, unordered	object with attributes	yes
`IlvFlatListModel`	flat, ordered	object with attributes	yes
`IlvFlatSetModel`	flat, unordered	object with attributes	yes
`IlvFlatTableModel`	flat, ordered	index or list of cells	yes
Swing `TreeTableModel`	tree, ordered	object with attributes	yes
Swing `TreeModel`	tree, ordered	object	no
Swing `TableModel`	flat, ordered	index or list of cells	yes
Swing `ListModel`	flat, ordered	object	no

Tree Data Model

In a tree data model, each object has a set of children objects. If the set of children is empty, the node is called *leaf node*.

If the set of children is not empty and there is only one node which is not the child of another node, the node is called *root node*. If the tree data model is empty there is no root node.

Ordered Data Model

In an ordered data model, the order of objects is relevant. In a tree data model, the order of the children of each object is also important. This does not mean that they are sorted by a particular criterion. This means that when an object X is inserted between A and B, the iterator will return the objects in the order A - X - B.

Unordered Data Model

In an unordered data model, the iterator order is unpredictable and objects cannot be addressed by indices.

Flat Data Model

In a flat data model, there is no parent/child relationship between objects. All objects are at the same level.

An object with attributes is an object which holds values for some given keys. When presented in tabular form, the attribute names become column names, and the attribute values become table cell values.

For example, if you have two objects Greg and Fred, with attributes Date of Birth, State of Birth, Income, defined as follows:

- `Greg.getValue("Date of birth") = 1947`
- `Greg.getValue("State of birth") = MA`
- `Greg.getValue("Income") = 81000`
- `Fred.getValue("Date of birth") = 1953`
- `Fred.getValue("State of birth") = CA`
- `Fred.getValue("Income") = 72000`

the tabular form would look like this:

Object	Date of Birth	State of Birth	Income
Greg	1947	MA	81000
Fred	1953	CA	72000

Choosing the data model to implement

When you instantiate an `IlvTreeTableDataSource`, you have to choose the type of data model through which you connect to it. The data model has to be an instance of `IlvModelWithColumns`, and you can choose among the following ones:

- `IlvTreeListModel`

Data is structured in some obvious and inherent way and the order is important.

- `IlvTreeSetModel`

  Data is structured in some obvious and inherent way and the order is not important.

- `IlvFlatListModel`

  Data is represented in a way that a row corresponds to a single object and the order is important.

- `IlvFlatSetModel`

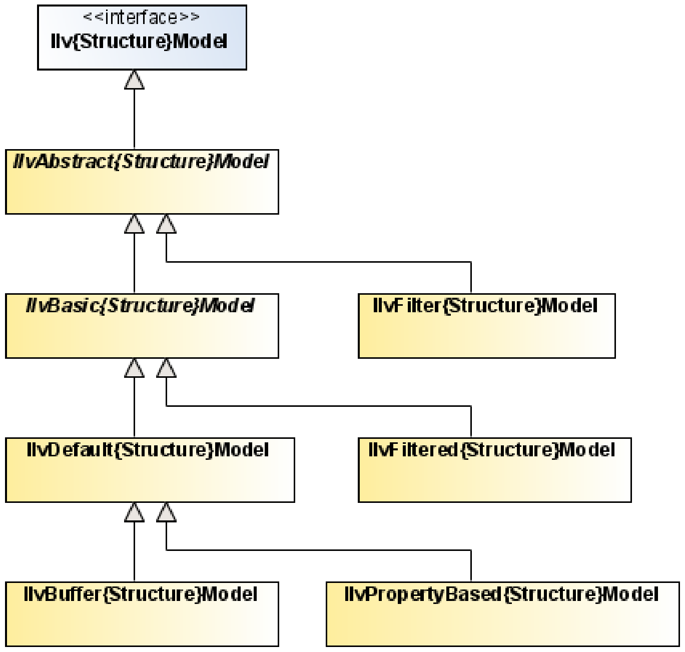
  Data is represented in a way that a row corresponds to a single object and the order is not important.

- `IlvFlatTableModel`

  Data is structured in a tabular form, where each cell represents a single object.

Predefined Extended Data Model Classes

The five extended data models share a common structure, as illustrated in *Predefined Extended Data Models*.



<<interface>>
Ilv{Structure}Model
IlvAbstract{Structure}Model
IlvBasic{Structure}Model
IlvFilter{Structure}Model
IlvDefault{Structure}Model
IlvFiltered{Structure}Model
IlvBuffer{Structure}Model
IlvPropertyBased{Structure}Model

Predefined Extended Data Models

IlvStructureModel

`Ilv`*Structure*`Model` is the model interface. It notifies the modifications in the form of *Structure*`ModelEvents` to all the attached *Structure*`ModelListeners`.

IlvAbstractStructureModel

`IlvAbstract`*Structure*`Model` is an incomplete implementation of the model. It only handles notification to the listeners, offering a `fireModelEvent` (or similar) method.

IlvBasicStructureModel

`IlvBasic`*Structure*`Model` is an incomplete implementation of the model. It handles the notification to the listeners and the management of columns.

IlvDefaultStructureModel

`IlvDefault`*Structure*`Model` is an implementation of the model that stores all the data values and columns. It can be used independently of any other model.

IlvBufferStructureModel

`IlvBuffer`*Structure*`Model` is an implementation of the model that stores data values coming from another `Ilv`*Structure*`Model` instance. It can be used when the access to the underlying `Ilv`*Structure*`Model` is slow, and the memory usage is not a problem.

IlvPropertyBasedStructureModel

`IlvPropertyBased`*Structure*`Model` is an implementation of the model that stores the object and columns. However, the data values are not duplicated in memory; instead, the objects are supposed to contain the data values, and a property-like API is used to access the data values in the objects.

IlvFilterStructureModel

`IlvFilter`*Structure*`Model` is a base class for implementations of the model that want to delegate most methods to an underlying model.

IlvFilteredStructureModel

`IlvFiltered`*Structure*`Model` is an implementation of the model that shows a subset of the objects from another `Ilv`*Structure*`Model` instance. The subset is determined by an `IlvFilter` instance.

When implementing the `Ilv`*Structure*`Model` interface, you can choose as superclass of your implementation the predefined implementation that comes closest to your needs.

Transforming Data Models

Describes how data models can be transformed into different models referring to the same data.

In this section

The IlvTreeTableDataSource data source

The `IlvTreeTableDataSource` is the basic data source that can be displayed by a treemap chart. It groups together three types of transforms in an easy-to-use API: filtering, sorting and partitioning. These transforms are optional.

The `IlvTreeTableDataSource` can be connected to one of the instances of `IlvModelWithColumns`:

- `IlvTreeListModel`
- `IlvTreeSetModel`
- `IlvFlatListModel`
- `IlvFlatSetModel`
- `IlvFlatTableModel`

The `IlvTreeTableDataSource` can filter, sort and partition the data. In all cases, the resulting `IlvTreeListModel`, accessible through `getTreeModel()`, is the result of these model operations. It is the `IlvTreeListModel` that is displayed by the treemap chart renderer.

Filtering

The filtering transform hides some model objects from the resulting model. You can set a filter object by means of the method `setFilterCriterion(ilog.views.util.filter.IlvFilter)`. This filter object defines the objects that are available in the resulting model. In a tree model, when a tree node is hidden, the entire tree branch below is hidden as well.

Note: Filtering is not available when the input model is an `IlvFlatTableModel`.

Sorting

The sorting transform sorts the objects according to a specific criterion. In the case of a tree model, it sorts also the children set of each tree node. The sort criterion is set by means of `setSortCriterion(java.util.Comparator)` and enabled through `setSorting(boolean)`.

Note: Sorting is not available when the input model is an `IlvFlatTableModel`.

The package `ilog.views.chart.datax.adapter.sort` contains several *`Comparator`* implementations that are useful in this context, like the following ones.

- `IlvColumnValueComparator` compares two objects by looking at the value in a specified column.

- `IlvLexicographicComparator` compares two objects through multiple criteria, in a "sort by ... then by ..." way.
- `IlvUniversalComparator` is able to compare any type of objects. This is useful if the given objects do not implement the *`Comparable`* interface.

Partitioning

The partitioning transform turns a flat model into a tree model, according to one or several rules that describe which objects should be grouped together and at which level. Such a rule is known as *partitioner*. Partitioners can be set by means of `setPartitionerFactories(ilog.views.chart.datax.adapter.partition.IlvPartitionerFactory[])`.

Note: Partitioning is currently not available when the input model is already a tree model. This limitation may be lifted in a future release.

The package `ilog.views.chart.datax.adapter.partition` contains several `IlvPartitionerFactory` implementations that are useful in this context. They all partition according to the value of the object in a given column, but act differently, depending on the value type and meaning.

- `IlvUniformScalePartitionerFactory` partitions according to a numerical value, dividing the range into intervals of equal size.
- `IlvCustomScalePartitionerFactory` partitions according to a numerical value, dividing the range into intervals at given threshold points.
- `IlvDatePartitionerFactory` partitions according to a date value.
- `IlvStringPartitionerFactory` partitions according to a string value.
- `IlvPathPartitionerFactory` partitions according to a string value, interpreting the string as a path, composed of path components separated through a given set of separators.
- `IlvURLPartitionerFactory` partitions according to a string value, interpreting the string as a URL.
- `IlvFilenamePartitionerFactory` partitions according to a string value, interpreting the string as a file name.
- `IlvHostnamePartitionerFactory` partitions according to a string value, interpreting the string as an Internet host name.

Here is an example showing the connection of an `IlvFlatListModel` to an `IlvTreeTableDataSource`, that groups the objects by country, sorts them alphabetically, and filters them to keep only those with positive performance.

```
final IlvFlatListModel model = ...;
IlvDataColumnInfo nameColumn = model.getColumn(0);
IlvDataColumnInfo countryColumn =
  IlvColumnUtilities.getColumnByName(model, "Country");
final int performanceColumnIndex =
  IlvColumnUtilities.getColumnIndexByName(model, "Performance");

// Create the data source.
IlvTreeTableDataSource dataSource = new IlvTreeTableDataSource();
dataSource.setUnderlyingModel(model);

// Activate filtering.
dataSource.setFilterCriterion(
  new IlvAbstractFilter() {
   public boolean evaluate(Object object) {
    return model.getDoubleAt(object, performanceColumnIndex) >= 0;
   }
  });

// Activate sorting.
dataSource.setSortCriterion(
  new IlvColumnValueComparator(model, nameColumn, null, false));
dataSource.setSorting(true);

// Activate partitioning.
dataSource.setPartitionerFactories(
  new IlvPartitionerFactory[] {
   new IlvStringPartitionerFactory(countryColumn)
  });
```

Model adapters

The `IlvTreeTableDataSource` uses some of the filtering, partioning, and sorting adapters to perform the filtering, partioning and sorting, as needed. However, sometimes it can be useful to perform these operations separately.

This can be the case when:

♦ You need two filtering passes: one before partitioning and one after partitioning.

♦ Your input model is not one of the five extended data models, but a Swing model.

To transform your model, you can use one of the following adapters:

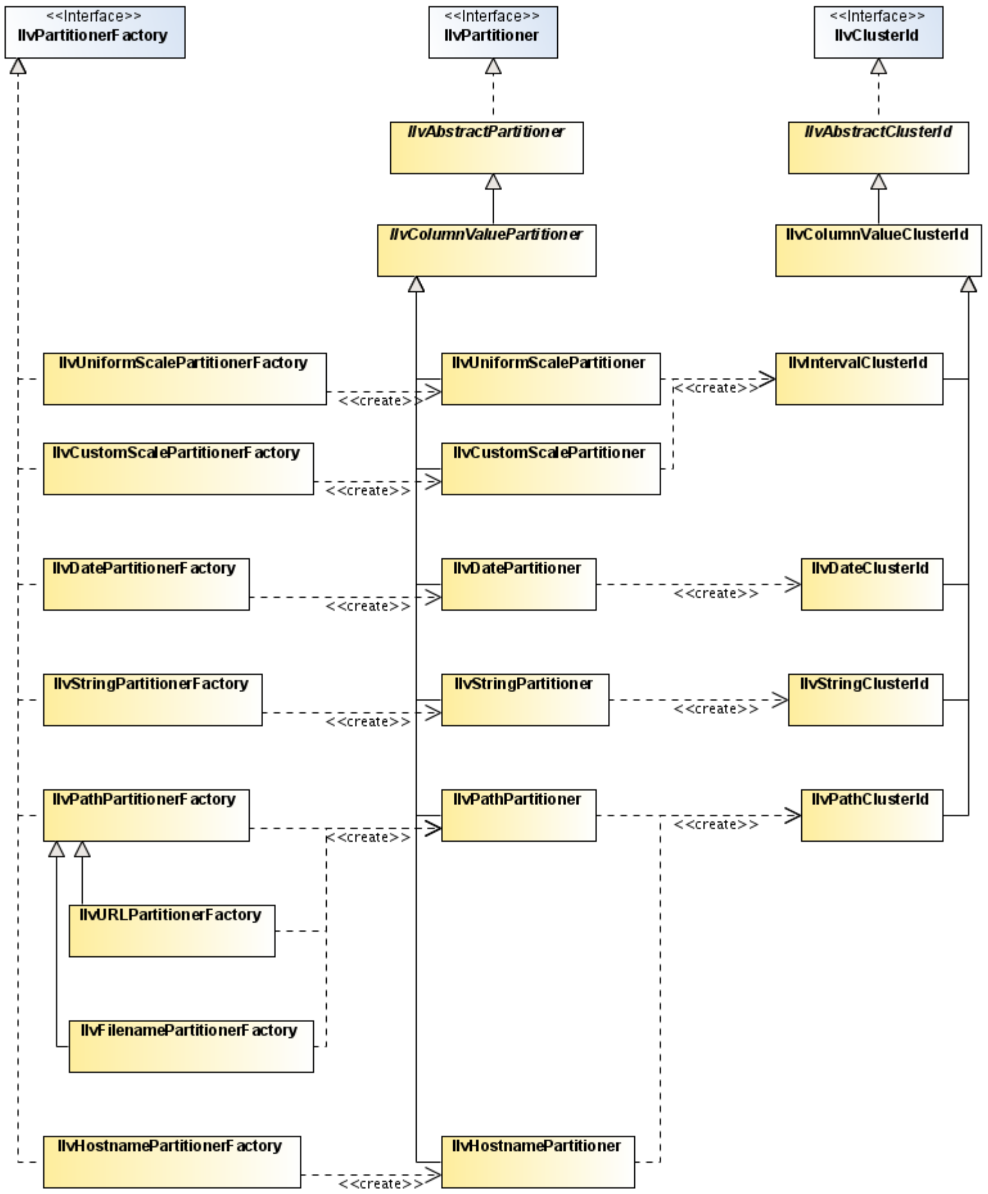
♦ *Adapters that Convert Models*

♦ *Adapters for Filtering*

♦ *Adapters for Sorting*

♦ *Adapters for Partitioning*

All these adapters are located in the package `ilog.views.chart.datax.adapter`.



Model Adapters Relationnships

Adapters that Convert Models

The adapter listed in the table below convert one type of model into models of another type, as faithfully as possible.

Adapters from `IlvTreeListModel`	`IlvTreeListToTreeSetModel`
	`IlvTreeListToFlatListModel`
	`IlvTreeListToFlatTableModel`
	`IlvTreeListToTreeTableModelFactory`
	`IlvTreeListToTreeModel`

Adapters from `IlvTreeSetModel`	`None`
Adapters from `IlvFlatListModel`	`IlvFlatListToFlatSetModel` `IlvFlatListToFlatTableModel`
Adapters from `IlvFlatSetModel`	`IlvFlatSetToFlatTableModel`

Adapters from `IlvFlatTableModel`	`IlvFlatTableToFlatListModel`
	`IlvFlatTableToTableModel`
	`IlvFlatTableToListModel`

Adapters from `IlvDataSource`	`IlvDataSourceToFlatTableModel`
Adapters from Swing `TreeTableModel`	`IlvTreeTableToTreeListModel`
Adapters from Swing `TreeModel`	`IlvTreeToTreeListModel` `IlvTreeToFlatTableModel`
Adapters from Swing `TableModel`	`IlvTableToFlatTableModel`
Adapters from Swing `ListModel`	`IlvListToFlatTableModel` `IlvListToFlatListModel`

Adapters for Filtering

These adapters perform the filtering transform of models, as discussed in the section *Filtering*.

♦ `IlvFilteredTreeListModel`

♦ `IlvFilteredTreeSetModel`

♦ `IlvFilteredFlatListModel`

♦ `IlvFilteredFlatSetModel`

These adapters select a branch of a tree model.

♦ `IlvSubTreeListModel`

♦ `IlvSubTreeSetModel`

Adapters for Sorting

These adapters perform the sorting transform of models, as discussed in the section *Sorting*.

- IlvSortedTreeListModel
- IlvSortedFlatListModel
- IlvTreeSetToTreeListModel
- IlvFlatSetToFlatListModel

Adapters for Partitioning

These adapters perform the partitioning transform of models, as discussed in the section *Partitioning*.

- IlvFlatListToTreeListModel
- IlvFlatSetToTreeSetModel

The partitioning transform introduces extra nodes in the resulting tree model. These extra nodes represent a group (cluster) of nodes and that were not present in the original model. These nodes are of the type `IlvClusterNode`. By means of the method `getId()` you can retrieve information about the common properties of the cluster. This information is of the type `IlvClusterId`. Each partitioner has a particular flavor of `IlvClusterId`, as shown in the following table:

Partitioner	The IlvClusterId Class
IlvUniformScalePartitionerFactory	IlvIntervalClusterId
IlvCustomScalePartitionerFactory	IlvIntervalClusterId
IlvDatePartitionerFactory	IlvDateClusterId
IlvStringPartitionerFactory	IlvStringClusterId
IlvPathPartitionerFactory	IlvPathClusterId
IlvURLPartitionerFactory	IlvPathClusterId
IlvFilenamePartitionerFactory	IlvPathClusterId
IlvHostnamePartitionerFactory	IlvPathClusterId

Here is an example showing how to convert a Swing `TableModel` to an `IlvTreeListModel`, that groups the rows by country, sorts them alphabetically, and filters them to keep only those with positive performance.

```
TableModel model = ...;
final int nameColumnIndex = 0;
final int countryColumnIndex = 2;
final int performanceColumnIndex = 3;

// Convert the model to a list of objects.
IlvFlatTableModel tableModel =
  new IlvTableToFlatTableModel(model,
   new IlvDataColumnInfo[] {
     new IlvDefaultDataColumnInfo("Name", String.class),
     new IlvDefaultDataColumnInfo("Founded", Date.class),
     new IlvDefaultDataColumnInfo("Country", String.class),
     new IlvDefaultDataColumnInfo("Performance", Double.class)
   });
IlvDataColumnInfo nameColumn =
  tableModel.getColumn(nameColumnIndex);
IlvDataColumnInfo countryColumn =
  tableModel.getColumn(countryColumnIndex);
IlvFlatListModel listModel =
  new IlvFlatTableToFlatListModel(tableModel);

// Add a filter.
IlvFlatListModel filteredModel =
  new IlvFilteredFlatListModel(listModel,
   new IlvAbstractFilter() {
     public boolean evaluate(Object object) {
       return model.getDoubleAt(object, performanceColumnIndex) >= 0;
     }
   });

// Add sorting.
IlvFlatListModel sortedModel =
  new IlvSortedFlatListModel(filteredModel,
    new IlvColumnValueComparator(sortedModel, nameColumn,
                                 null, false));

// Add partitioning.
IlvTreeListModel partitionedModel =
  new IlvFlatListToTreeListModel(sortedModel,
     new IlvStringPartitionerFactory(countryColumn),
     null, 1);
```

Configuring the Data Projection

Explains how to configure the axis and the projector.

In this section

Configuring the Axis

Describes the IlvAxis class and how it is used by charts.

In this section

The chart

Each chart uses several coordinate axes, which are represented by the `IlvAxis` class.

A chart is composed of:

♦ Exactly one abscissa axis, which can be retrieved with the `getXAxis()` method.

♦ One or several ordinate axes, which can be retrieved with the `getYAxis(int)` method.

The axes are automatically created by a chart, which uses by default only one *y-axis*. The first *y-axis* is also referred to as the *main ordinate axis*. Other *y*-axis can be added to a chart with the `addYAxis(boolean, boolean)` method. You can determine the type of an axis with the `getType()` method.

Within a chart, each *y*-axis forms a coordinate system with the *x*-axis. A coordinate system is an instance of the class, and can be retrieved with the `getCoordinateSystem(int)` method. Throughout the API of the library, both *y*-axis and coordinate systems are usually referenced by their index in the chart (starting at 0). For example, the third parameter to the `scroll(double, double, int)` method specifies which *y*-axis should be modified:

```
// Translates by 20. the visible range of the first y-axis.
chart.scroll(0., 20., 0)
```

Example: Synchronizing Axes

You can synchronize two chart axes with the `synchronizeAxis(ilog.views.chart.IlvChart, int, boolean)` method. After this method is invoked, both charts share the same axis instance.

The complete source code of this example can be found in **`<installdir>/jviews-charts86/codefragments/chart/axis-sync/src/AxisSync.java`**.

```
IlvChart topChart = new IlvChart();
// the topchart data source
IlvDefaultDataSource ds =
     new IlvDefaultDataSource(new double[][]
                                     {IlvArrays.randomValues(COUNT, 0, 50)},
                              -1,
                              new String[]{"Data Set 1"},
                              null);
topChart.setDataSource(ds);
// add some interactors to be able to play with the axis range
topChart.addInteractor(new IlvChartZoomInteractor());
topChart.addInteractor(new IlvChartPanInteractor());

// the bottom chart. This chart shares the same x-axis with topChart.
IlvChart bottomChart = new IlvChart();
ds = new IlvDefaultDataSource(new double[][]
                                     {IlvArrays.randomValues(COUNT, 0, 50)},
                              -1,
                              new String[]{"Data Set 2"},
                              null);
  bottomChart.setDataSource(ds);
   // synchronixe the x-axis with the one from topchart. We also want to
   // synchronize the plotarea of both charts so that grids are aligned.
  bottomChart.synchronizeAxis(topChart, IlvAxis.X_AXIS, true);
```

Axis properties

The following table lists the properties of an axis:

Property	Methods	Default Value
`reversed`	`setReversed` `isReversed`	false
`adjusting`	`setAdjusting` `isAdjusting`	false
`autoDataMin`	`setAutoDataMin` `isAutoDataMin`	true
`autoDataMax`	`setAutoDataMax` `isAutoDataMax`	true
`autoVisibleRange`	`setAutoVisibleRange` `isAutoVisibleRange`	true

Reversing an Axis

The `setReversed(boolean)` method allows you to toggle the reversed property of an axis. The values along a reversed axis are considered in backward order. For example, reversing the *x*-axis affects the orientation of a polar projector:

- The projector is oriented counter-clockwise if the *x*-axis is not reversed.
- The projector is oriented clockwise if the *x*-axis is reversed.

Specifying the Automatic Modes of Axis Ranges

Three properties define the automatic range modes of an axis, as described in section *Changing the axis ranges*:

- `isAutoDataMin()` indicates whether the minimum data value is automatically computed. You can disable the automatic mode with the `setAutoDataMin(boolean)` method, or by explicitly specifying the minimum data value.
- `isAutoDataMax()` indicates whether the minimum data value is automatically computed. You can disable the automatic mode with the `setAutoDataMax(boolean)` method, or by explicitly specifying the maximum data value.
- `isAutoVisibleRange()` indicates whether the visible range is synchronized with the data range. You can disable the synchronization of the `setAutoVisibleRange(boolean)` method, or by explicitly specifying the visible range.

Setting the Axis in an Adjusting State

The `setAdjusting(boolean)` method lets you toggle the adjusting state of an axis. This state is used when firing `AxisEvent` events, so that registered listeners know that the received notifications are part of a set of changes.

Changing the axis ranges

An `IlvAxis` object defines two ranges:

- The data range, which specifies the limits of the data values along this axis. This range can be unbounded, which means that the minimum and maximum data values are undefined.
- The visible range, which specifies the visible data interval along this axis.

The axis ensures that the visible range is always contained within the data range:

```
dataMin <= visibleMin <= visibleMax <= dataMax
```

The data range of an axis is specified according to two modes:

- Automatic mode

  In this mode, the data range is computed by the chart. The chart delegates this calculation to an `IlvDataRangePolicy` object. The default policy computes the data range so that it fits the data actually displayed by the chart.

- Manual mode

  The `setDataMin(double)`, `setDataMax(double)`, or `setDataRange(ilog.views.chart.IlvDataInterval)` methods allow you to indicate the minimum and maximum data values.

When you call a method to explicitly set a value for the minimum data and maximum data value, you toggle off the automatic mode for the corresponding value. For example, calling the `setDataMin` method disables the automatic calculation of the minimum data.

The visible range of an axis also follows two modes:

- The visible range can be synchronized with the data range. In that case, the visible range is updated each time the data range is modified.
- The visible range can be explicitly specified with the `setVisibleMin(double)`, `setVisibleMax(double)setVisibleMax`, or `setVisibleRange(double, double)` methods. By calling one of these methods, you toggle off the synchronization of the visible range with the data range.

You can find a list of the properties related to the axis data range and the axis visible range in section *Axis properties*.

Setting the axis transformer

An optional transformation can be associated with an `IlvAxis` instance. This transformation is applied to every data value along this axis before it is converted to display coordinates.

Axis transformers are implemented by subclasses of the `IlvAxisTransformer` abstract class. Each concrete subclass must implement two abstract methods:

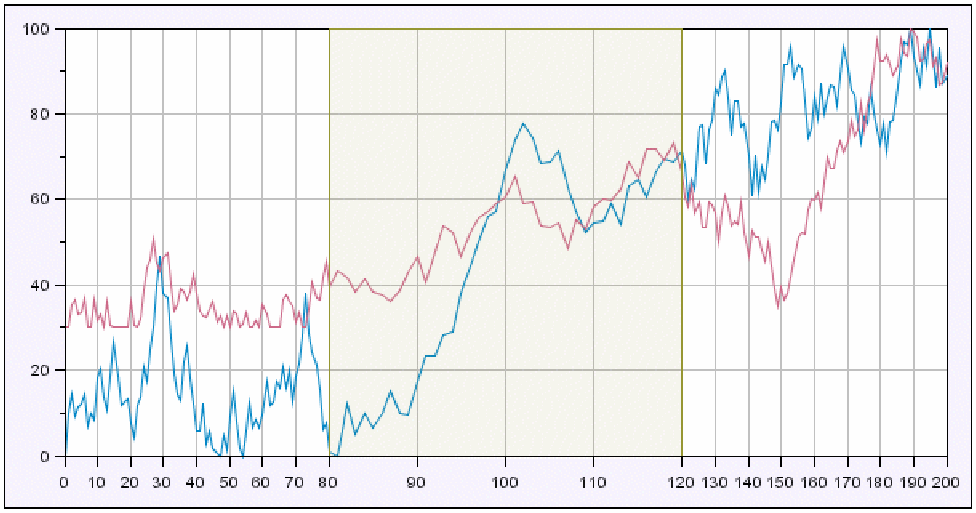
- `apply(double)`, which performs the forward transformation.
- `inverse(double)`, which performs the inverse transformation.

You can override the default implementation of the other methods of the `IlvAxisTransformer` class. For example, the `apply(double[], int)` method uses the elementary transformation on all the values of the specified array. A faster implementation can sometimes be found by making intermediate calculations only once.

The following predefined transformations are available in the JViews Charts library:

- `IlvAffineAxisTransformer` applies an affine transformation.
- `IlvLogarithmicAxisTransformer` applies a logarithmic transformation.
- `IlvLocalZoomAxisTransformer` applies a scaling factor to data values within a given range.

Local Zoom shows the effect of an `IlvLocalZoomAxisTransformer` set on the *x*-axis of a chart:



Local Zoom

Listening to axis events

The `AxisEvent` class represents the base class for axis events. Listeners can be registered to receive these events with the `addAxisListener(ilog.views.chart.event.AxisListener)` and `removeAxisListener(ilog.views.chart.event.AxisListener)` methods.

Range Events

These events are sent when the visible range or the data range of an axis changes. When this happens, two `AxisRangeEvent` events are fired:

- ♦ A first event is sent *before* the change actually occurs. This event is also called an *about-to-change* event. It gives to the listeners an opportunity to constrain the proposed new value of the range with the `setNewMin(double)` and `setNewMax(double)` methods.
- ♦ A second event is sent *after* the change. The previous values of the range can be retrieved with the `getOldMin()` and `getOldMax()` methods.

You can check whether an event is an about-to-change event with the `isAboutToChangeEvent()` method. The following code shows how a listener can be used to coerce the visible range of an axis:

```
axis.addAxisListener(new AxisListener() {

  /**
   * Constrain visible min and visible max to integer values.
   */
   public void axisRangeChanged(AxisRangeEvent ev) {
     if (ev.isChangedEvent() || !ev.isVisibleRangeEvent()) return;
     ev.setNewMin(Math.floor(ev.getNewMin()));
     ev.setNewMax(Math.ceil(ev.getNewMax()));
   }
   public void axisChanged(AxisChangeEvent evt) {}
});
```

Change Events

The change events are implemented by the `AxisChangeEvent` class. These events are sent when one of the following changes occur:

- ♦ The reversed property of the axis has been modified. The type of the event is `AxisChangeEvent.ORIENTATION_CHANGE`.
- ♦ The adjusting property of the axis has been modified. The type of the event is `AxisChangeEvent.ADJUSTMENT_CHANGE`.

The transformer of the axis has changed. The type of the event is `AxisChangeEvent.TRANSFORMER_CHANGE`. This type of event covers all the changes that can affect the transformer. It also includes setting the transformer or removing it from the axis.

Handling chart resizing

The way a change of the chart size affects the visible range of the axis is handled through a resizing policy. A resizing policy determines whether the visible range of the axis of a Cartesian chart is modified when the chart is resized.

Resizing policies are implementations of the `IlvChartResizingPolicy` interface and are set on a chart by means of the `setResizingPolicy(ilog.views.chart.IlvChartResizingPolicy)` method.

The JViews Charts package provides a default implementation of this interface by means of the `IlvChartResizingPolicy.DEFAULT_POLICY` class. This class expands the visible range of the coordinate axis when a chart area is resized so that the scaling factor of the Cartesian projection keeps the same value.

By default, a chart has no resizing policy, that is, the visible range of the axis is not changed when the chart size changes.

Note: A resizing policy applies only to Cartesian charts.

Configuring the Projector

Describes the projector and how it can be used with points and rectangular areas.

In this section

Projector Properties

The conversion between data space and display space is performed by a projector owned by the chart. Depending on its type, a chart uses one of the two predefined projectors available in the JViews Charts package:

♦ Cartesian projector

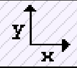
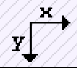
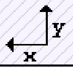
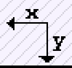
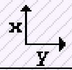
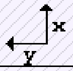
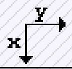
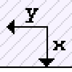
♦ Polar projector

The projector used by a chart can be retrieved with the `getProjector()` method.

Projector properties are accessible through the API of the `IlvChart` class.

The `setProjectorReversed(boolean)` method allows you to reverse a projector. A reversed projector swaps the meaning of the abscissa and ordinate coordinates of a point. For example, a reversed Cartesian projector projects *x*-data values along the *y*-axis of the screen.

Cartesian Orientation illustrates the different Cartesian orientations that can be specified by using this property in conjunction with the reversed property of an axis:

Projector Reversed	X-Axis Reversed	Y-Axis Reversed	Cartesian Orientation
false	false	false	
false	false	true	
false	true	false	
false	true	true	
true	false	false	
true	false	true	
true	true	false	
true	true	true	

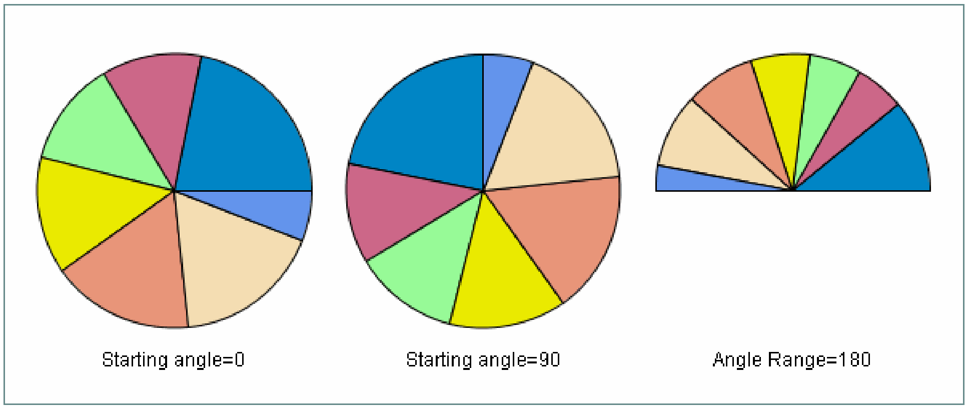
Cartesian Orientation

The `IlvChart` class also provides methods to change the specific properties of a polar projector:

♦ `setStartingAngle(double)` changes the starting angle of the projector.

♦ `setAngleRange(double)` changes the range of the projector.

Polar Properties shows how these properties modify the appearance of a pie chart:



Polar Properties

Projecting points

The `IlvChartProjector` interface defines two methods to perform the conversion between data points and display points:

- `toDisplay(ilog.views.chart.IlvDoublePoints, java.awt.Rectangle, ilog.views.chart.IlvCoordinateSystem)` achieves the *forward* projection (from data space to display space).
- `toData(ilog.views.chart.IlvDoublePoints, java.awt.Rectangle, ilog.views.chart.IlvCoordinateSystem)` achieves the *inverse* projection (from display space to data space).

Both methods use the following parameters:

- An `IlvDoublePoints` object that holds the points to project. The contents of the `IlvDoublePoints` is directly modified by the method.
- An `IlvCoordinateSystem` object that represents the data coordinate system.
- A `Rectangle` object that represents the display coordinate system.

The projection pipeline is illustrated in *Chart Projector*:

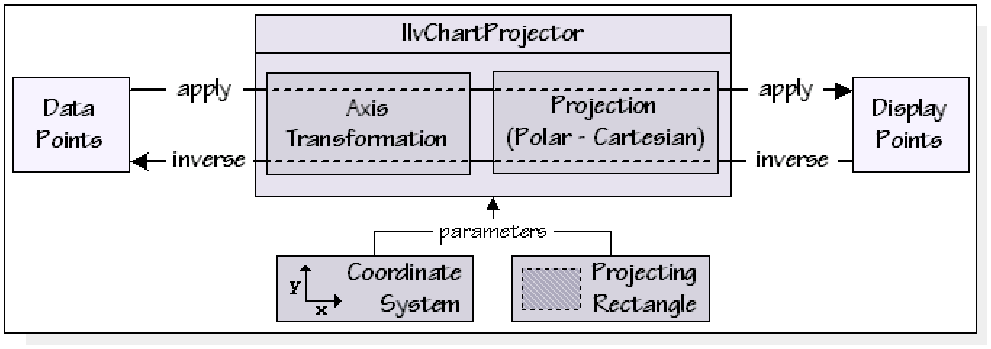


Chart Projector

The projection is performed in two stages:

- Apply the transformations that are set on the axis of the provided coordinate system.
- Transform the data values into display coordinates according to the provided projecting rectangle, and the visible range of the axis.

The following code extract shows how to project a data point:

```
IlvChart chart = ...;
// x and y are the x- and y-values of the data point.
IlvDoublePoints pts      = new IlvDoublePoints(x, y);

// The following lines are equivalent to: chart.toDisplay(pts).
```

```
IlvCoordinateSystem coordSys  = chart.getCoordinateSystem(0);
Rectangle projRect  = chart.getProjectorRect();
chart.getProjector().toDisplay(pts, projRect, coordSys);
System.out.println("Projected coords: " + pts);
```

Note: The projecting rectangle returned by the `getProjectorRect()` method is expressed in the coordinate system of the chart area component. The projected coordinates are always relative to the upper-left corner of this component. For more information on the components of a chart, please refer to *Creating a Chart*.

Projecting rectangular areas

Rectangular areas are represented by:

- `java.awt.Rectangle` objects in display space.
- `IlvDataWindow` objects in data space.

The `IlvChartProjector` interface defines the following method to convert rectangular areas:

- `toRectangle(ilog.views.chart.IlvDataWindow, java.awt.Rectangle, ilog.views.chart.IlvCoordinateSystem)` converts a data window to a display rectangle.
- `toDataWindow(java.awt.Rectangle, java.awt.Rectangle, ilog.views.chart.IlvCoordinateSystem)` converts a display rectangle to a data window.

The following code extract shows how you can use the `toDataWindow` method to retrieve all the points of a data set that are projected within a given rectangle:

```
Rectangle           selectRect = ...; // The selection rectangle
IlvDataSet          dataSet    = ...;
IlvCoordinateSystem coordSys   = chart.getCoordinateSystem(0);
Rectangle           projRect   = chart.getProjectorRect();
IlvChartProjector   prj        = chart.getProjector();

// Convert the selection rectangle into a data window.
IlvDataWindow w = prj.toDataWindow(selectRect, projRect, coordSys);

// Fetch data points that lies within the computed window.
IlvDataPoints pts = dataSet.getDataInside(w, 0, false);
```

Projecting a set of data points

Several methods in the `IlvChartProjector` interface let you project a set of data points into `java.awt.Shape` objects:

- `getShape(ilog.views.chart.IlvDataWindow, java.awt.Rectangle, ilog.views.chart.IlvCoordinateSystem)`. The considered set of data points is formed by all the points contained in the specified window.
- `getShape(double, int, java.awt.Rectangle, ilog.views.chart.IlvCoordinateSystem)`. The considered set of data points is formed by all the points that have a fixed *x*- or *y*-coordinate, and that lie within the visible window.
- `getShape(double, ilog.views.chart.IlvDataInterval, int, java.awt.Rectangle, ilog.views.chart.IlvCoordinateSystem)`. The considered set of data points is formed by all the points that have a fixed *x*-coordinate, and a *y*-coordinate that lies within a specified interval, or vice versa.

Handling Chart Renderers

Provides detailed information on data display for the charts.

In this section

Chart Renderers

The way data is rendered on the screen (polyline, bar, area, and so on) is handled by dedicated objects called *renderers*.

Renderers are divided into three categories:

♦ Single renderers, which display data from one (or several) specific data set(s).

♦ Composite renderers, which display data from a specific data source, using one or more single renderers.

♦ Simple renderers, which display data from a specific data source.

The composite renderers are defined as a combination of other renderers, which can be either single renderers or other composite renderers, so that each data set in the associated data source is rendered by one of these child renderers.

The simple renderers do the rendering of an entire data source by themselves, without dispatching the drawings of different data sets to different renderers.

Chart renderers are instances of `IlvChartRenderer` subclasses. The composite renderer classes inherit from the `IlvCompositeChartRenderer` abstract class, and the single renderer classes inherit from the `IlvSingleChartRenderer` abstract class. The simple renderer classes inherit from the `IlvSimpleChartRenderer` abstract class.

In addition to these three main categories, we can distinguish a special type of composite renderers that handles a one-to-one relation between its child renderers and its data sets (that is, one child renderer per data set). These composite renderers are instances of the `IlvSimpleCompositeChartRenderer` class, and handle `IlvSingleChartRenderer` instances as child renderers.

Most of the composite chart renderer classes available in the JViews Charts library are subclasses of `IlvSimpleCompositeChartRenderer`.

Chart Renderer Hierarchy shows the chart renderer hierarchy:

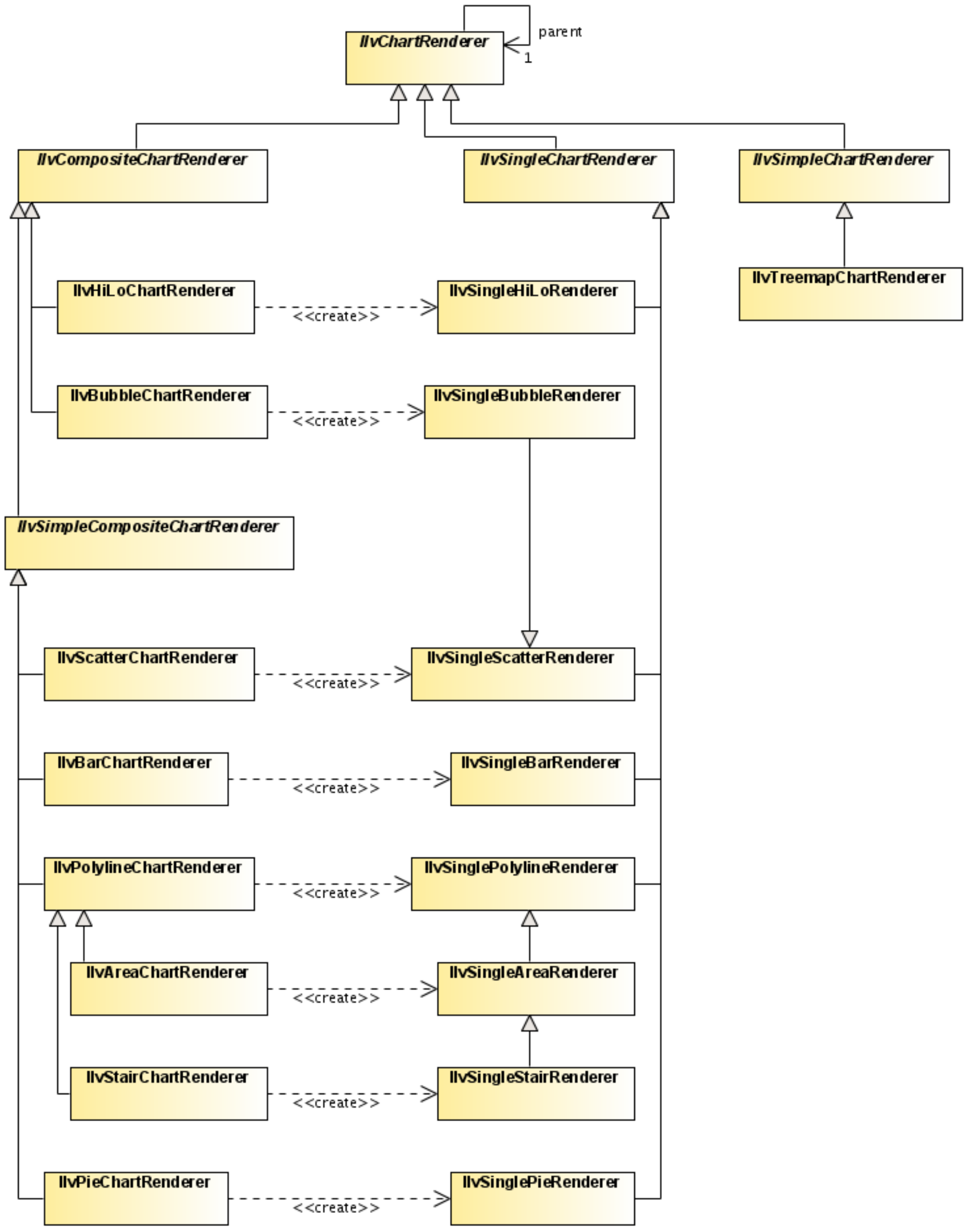


Chart Renderer Hierarchy

A data model is drawn in a chart when it is bound to a renderer added to a chart. Depending on the data model, this association is performed between:

♦ A data source and a composite chart renderer if the data model is a data source.

Set the data source as the current renderer data source by means of the `setDataSource(ilog.views.chart.data.IlvDataSource)` method.

- ♦ A data set and a single chart renderer if the data model is a data set.

  Add the data set to the renderer data source by means of the `addDataSet(ilog.views.chart.data.IlvDataSet)` method.

Using Chart Renderers

Describes all the possible graphical representations available in the JViews Charts library and a presentation of their implementation.

In this section

Treemap Charts

Describes the Treemap charts.

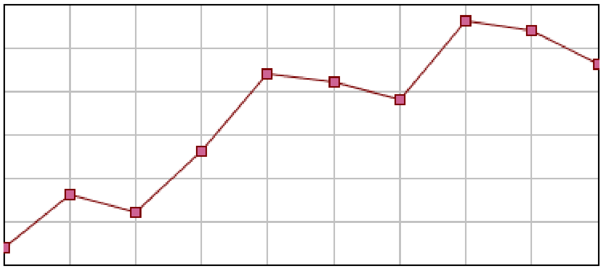
Polyline Charts

The Polyline charts have the following characteristics:

Single class	**`IlvSinglePolylineRenderer`**
Inherits from	`IlvSingleChartRenderer`
Composite class	`IlvPolylineChartRenderer`
Inherits from	`IlvSimpleCompositeChartRenderer`

IlvSinglePolylineRenderer Properties

A single polyline renderer can draw an additional marker for each data point. This marker is an instance of an implementation of the `IlvMarker` interface, and is set by means of the `setMarker(ilog.views.chart.graphic.IlvMarker)` method. You can also specify the rendering style of the marker symbol with the `setMarkerStyle(ilog.views.chart.IlvStyle)` By default, the style of the marker is computed according to the style of the renderer.



A polyline with an associated square marker

Polyline with a Square Marker Drawn on Each Data Point

IlvPolylineChartRenderer Properties

This renderer supports three representation modes: Superimposed, Stacked, and Stacked100.

Superimposed Mode

Polylines are drawn on top of each other.

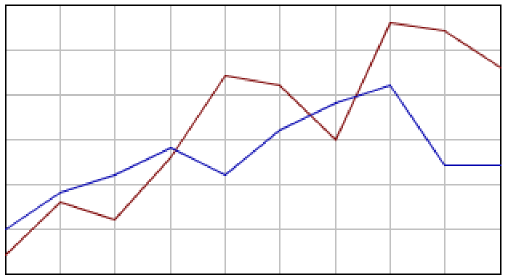
Stacked Mode

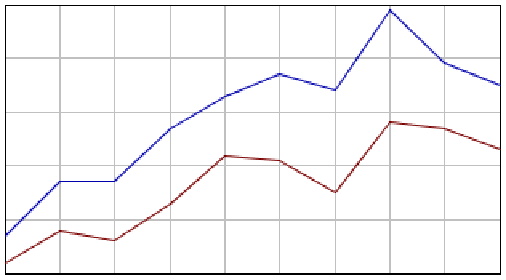
Polylines are stacked, so that each one displays the contribution of a *y*-value in a set of several *y*-values.

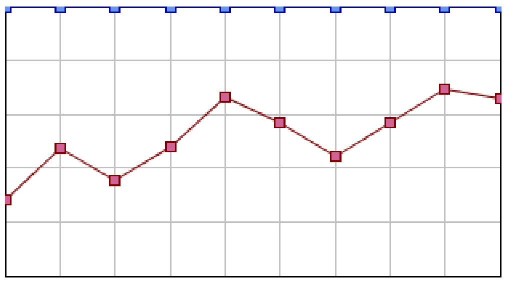
Stacked100 Mode

The contribution of a *y*-value is computed as a percentage of all the y-values for a given x-value.

The representation mode can be set either at construction time specifying a corresponding `IlvPolylineChartRenderer` mode constant as parameter of the constructor, or by means of the `setMode(int)` method.


Superimposed Polylines


Stacked polylines


Stacked Polylines (100% mode) with square marker

Representation Modes: Superimposed, Stacked, and Stacked100

Area Charts

The Area charts have the following characteristics:

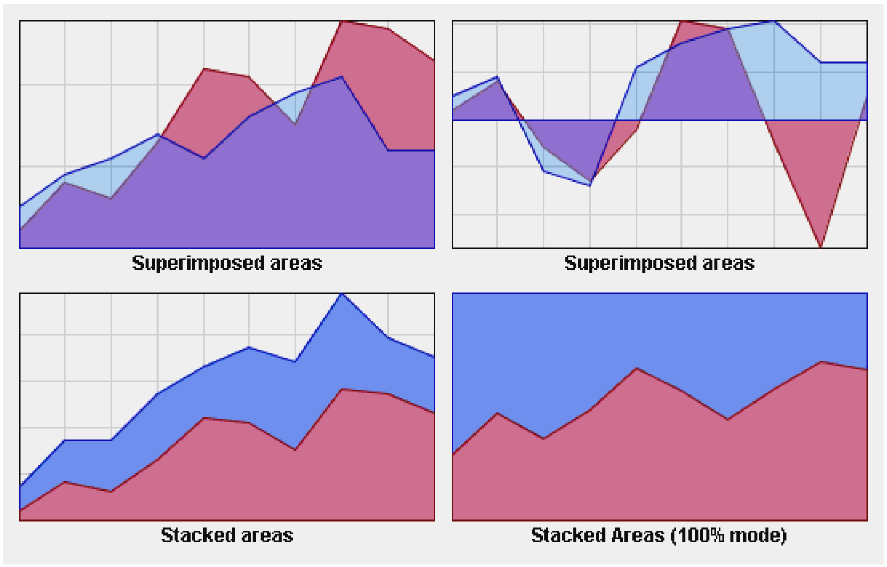
Single class	**`IlvSingleAreaRenderer`**
Inherits from	`IlvSinglePolylineRenderer`
Composite class	`IlvAreaChartRenderer`
Inherits from	`IlvPolylineChartRenderer`

IlvSingleAreaRenderer Properties

This class inherits from the `IlvSinglePolylineRenderer` properties previously listed, except for the graphical representation that represents a data set as an area instead of polylines.

IlvAreaChartRenderer Properties

This class inherits from the `IlvPolylineChartRenderer` representation modes listed in section *IlvPolylineChartRenderer Properties*.



Representation Modes: Superimposed, Stacked, and Stacked100 Mode

Bar Charts

The Bar charts have the following characteristics:

Single class	**`IlvSingleBarRenderer`**
Inherits from	`IlvSingleChartRenderer`
Composite class	`IlvBarChartRenderer`
Inherits from	`IlvSimpleCompositeChartRenderer`

IlvBarChartRenderer Properties

A bar chart renderer supports four representation modes: Superimposed, Clustered, Stacked, and Stacked100.

Superimposed Mode

Bars are drawn on top of each other.

Clustered Mode

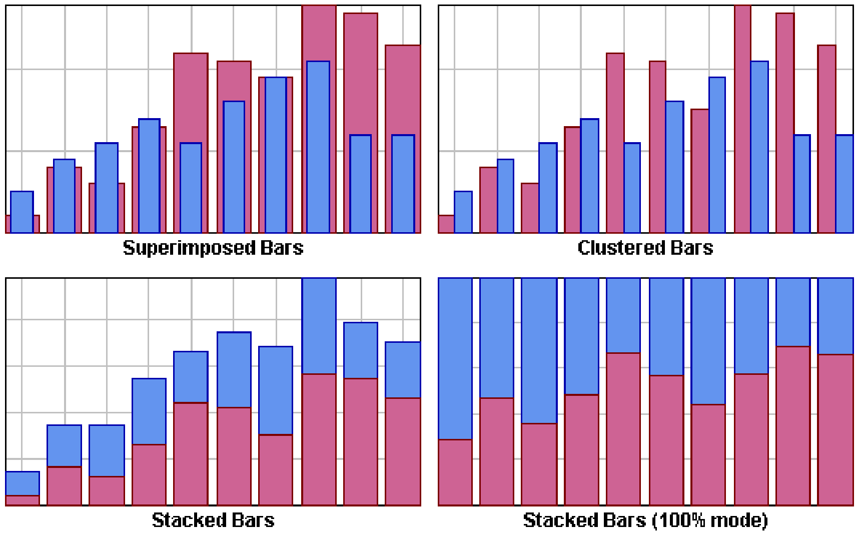
Bars are laid out in clusters, each cluster representing the set of *y*-values corresponding to a given *x*-value. The default cluster width can be modified by means of the `setClusterWidth (double)` method.

Stacked Mode

Bars are stacked, so that each one displays the contribution of a *y*-value in a set of several *y*-values.

Stacked100 Mode

The contribution of a *y*-value is computed as a percentage of all the *y*-values for a given *x*-value.



Representation Modes: Superimposed, Clustered, Stacked, and Stacked100 Mode

Note: Horizontal Bar charts can be obtained by reversing the chart projector. For more information, see `setProjectorReversed(boolean)` documentation, and the section *Projector Properties*.

Stacked Diverging Mode

Bar are stacked, in a way that accomodate negative values. In this mode, negative values are stacked separately from positive values: positive y values will be stacked together in a bar towards positive values, and negative y values will be stacked together in a bar in the opposite direction.

This rendering mode makes it easy to visually understand negative values.

Note:

1. This mode assumes that there is no particular order among the data sets; the data points belonging to negative y values are reordered, as if they all came before the positive y values.
2. Also, in this mode, the largest displayed y value is no longer the sum of all y values; rather, it is the sum of all positive y values. This can be confusing for the user.

Bubble Charts

The Bubble charts have the following characteristics:

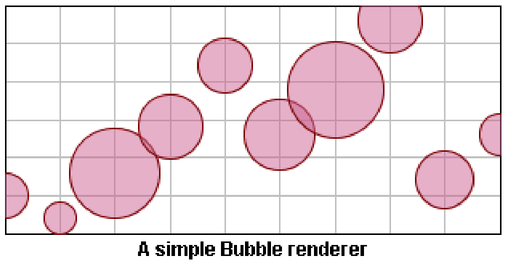
Single class	**`IlvSingleBubbleRenderer`**
Inherits from	`IlvSingleScatterRenderer`
Composite class	`IlvBubbleChartRenderer`
Inherits from	`IlvCompositeChartRenderer`

IlvSingleBubbleRenderer Properties

A bubble chart represents a two-dimensional data model as bubbles of variable size. The data model should be described by two data sets, the first data set determining the location of the bubbles, and the second data set determining the size of the bubbles.

IlvBubbleChartRenderer Properties

An `IlvBubbleChartRenderer` creates one child renderer for every pair of data sets contained in its data source.



A Bubble Renderer

High/Low Charts

The High/Low charts have the following characteristics:

Single class	**`IlvSingleHiLoRenderer`**
Inherits from	`IlvSingleChartRenderer`
Composite class	`IlvHiLoChartRenderer`
Inherits from	`IlvCompositeChartRenderer`

IlvSingleHiLoRenderer Properties

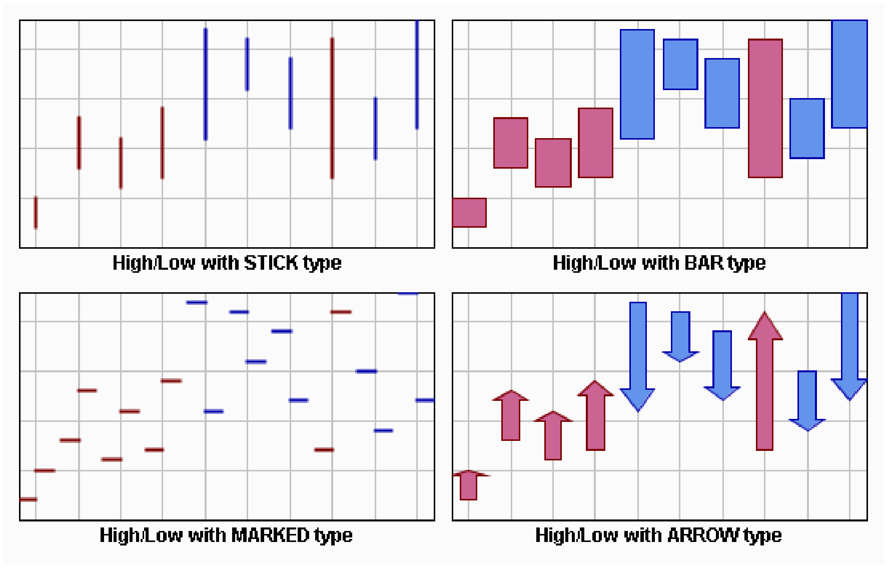
An `IlvSingleHiLoRenderer` instance renders two data sets with low and high items.

This class defines two rendering styles: a *rise style* and a *fall style*.

The *rise style* is used to draw the high-low items for which the corresponding low value is less than the high value.

The *fall style* is used to draw the high-low items for which the corresponding low value is greater than the high value.

The possible graphical representations supported by this class are Bar, Arrow, Marked, and Stick, and are illustrated in *Graphical Representations: Bar, Arrow, Marked, and Stick*:



Graphical Representations: Bar, Arrow, Marked, and Stick

IlvHiLoChartRenderer Properties

This renderer supports three representation modes: Clustered, OpenClose, and Candle.

Clustered Mode

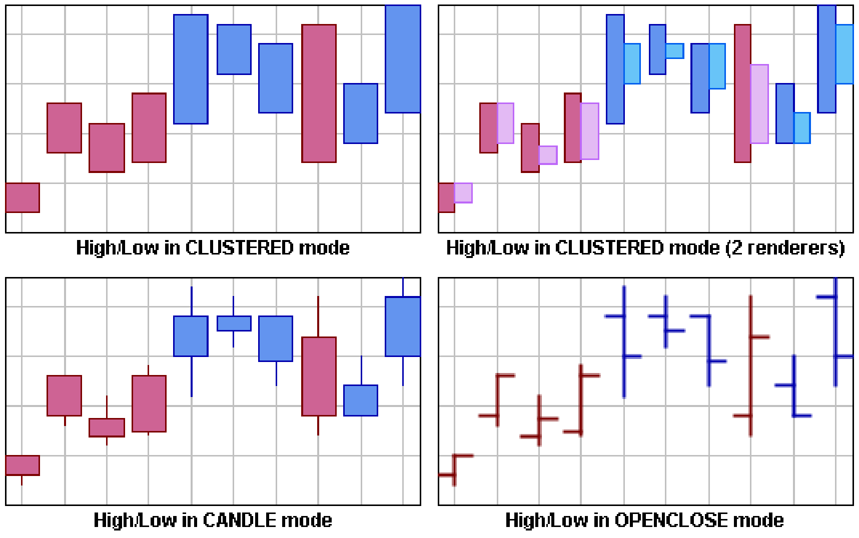
This class creates an instance of `IlvSingleHiLoRenderer` for every pair of data sets contained in its data source, and the creation of the associated legend item is delegated to the child renderer `createLegendItems()` method.

OpenClose Mode

This class handles two pairs of data sets: the first pair for the low/high values, the second pair for the open/close values, each pair being rendered by a child renderer, respectively of type Stick and Marked. In this mode, the legend item creation is directly handled by the composite renderer.

Candle Mode

This class handles two pairs of data sets as for the OpenClose mode, the low/high values being rendered with a Marked child renderer, the Open/Close values with a Bar child renderer. In this mode, the legend item creation is directly handled by the composite renderer.



Representation Modes: Clustered, Candle, and OpenClose Mode

Pie Charts

The Pie charts have the following characteristics:

Single class	**`IlvSinglePieRenderer`**
Inherits from	`IlvSingleChartRenderer`
Composite class	`IlvPieChartRenderer`
Inherits from	`IlvSimpleCompositeChartRenderer`

IlvSinglePieRenderer Properties

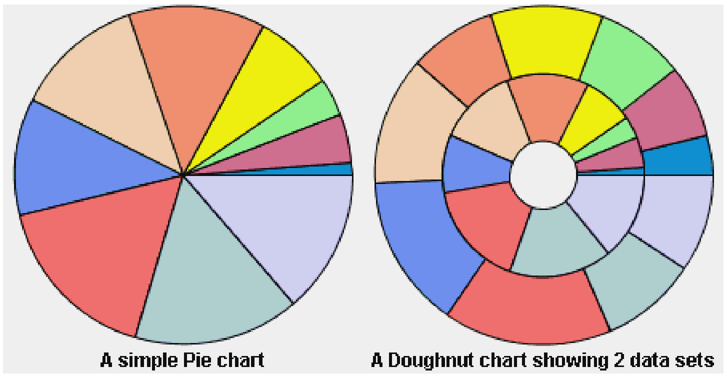
This class renders a data set as a pie chart. Each data point of a data set is rendered as a slice.

Specific slices of an `IlvSinglePieRenderer` can be exploded from the pie chart using the `setExploded(int, boolean)` method. The explode ratio of a slice can be changed by means of the `setExplodeRatio(int, int)` method.

This renderer defines one rendering style for each data point, so that each slice of the pie is drawn using its own rendering style.

IlvPieChartRenderer Properties

A Pie chart can be drawn with a hole in its center. This type of chart is called a *doughnut* chart. The hole size is expressed as a percentage of the available space that the hole will occupy. Pie renderers are usually added to pie charts but can also be used with other chart type (Cartesian for example).



Pie Chart and Doughnut Chart

Scatter Charts

The Scatter charts have the following characteristics:

Single class	**`IlvSingleScatterRenderer`**
Inherits from	`IlvSingleChartRenderer`
Composite class	`IlvScatterChartRenderer`
Inherits from	`IlvSimpleCompositeChartRenderer`

IlvSingleScatterRenderer Properties

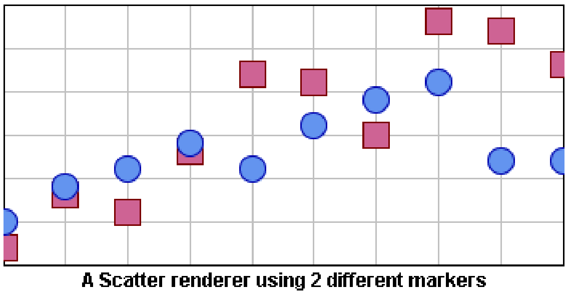
This class renders a data set with scattered graphical markers. Markers are simple graphical objects implementing the `IlvMarker` interface, and markers of predefined type can be retrieved through the `IlvMarkerFactory` class.

A scatter renderer draws square markers by default. You can change this marker by means of the `setMarker(ilog.views.chart.graphic.IlvMarker)` method, passing a new `IlvMarker` instance as parameter.

The marker size can be changed at any time by means of the `setMarkerSize(int)` method, passing the half-size of the marker as parameter.

IlvScatterChartRenderer Properties

This renderer class displays data sets as several scatter charts. Each data set is drawn by its own child renderer using a specified marker type. The marker used to draw the graphical representations can either be common to all child renderers, specified at construction time, or specific to a child renderer, by changing its own marker. By default, each data set is drawn using square markers.



Scatter Renderer Using Two Different Markers

Stair Charts

The Stair charts have the following characteristics:

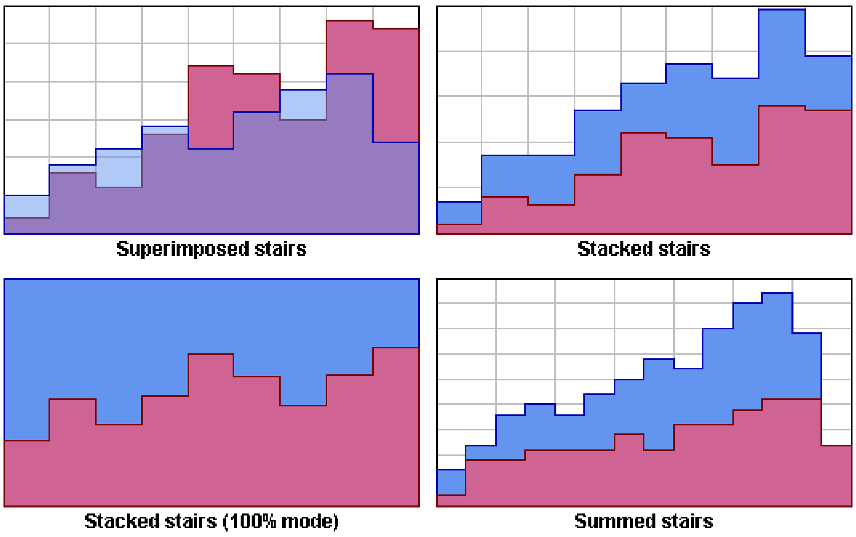
Single class	**`IlvSingleStairRenderer`**
Inherits from	`IlvSingleAreaRenderer`
Composite class	`IlvStairChartRenderer`
Inherits from	`IlvPolylineChartRenderer`

IlvSingleStairRenderer Properties

This class inherits from the `IlvSingleAreaRenderer` properties, except that the graphical representation represents a transition between two values as a stair instead of straight lines.

IlvStairChartRenderer Properties

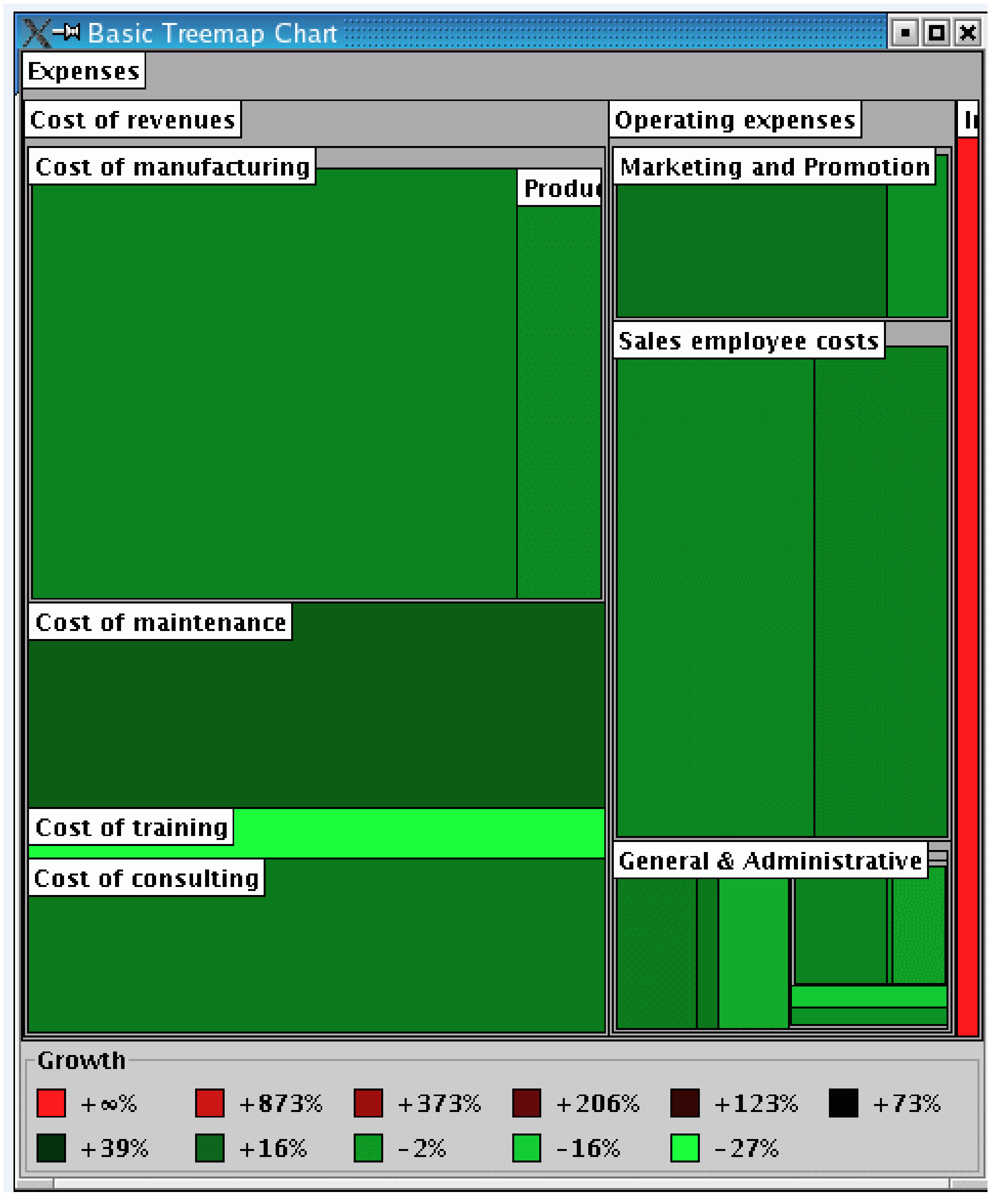
This class inherits from the `IlvPolylineChartRenderer` properties.



Representation Modes: Superimposed and Stacked Stairs Mode

Treemap Charts

A treemap chart displays objects as rectangles. The important objects are represented by large areas while the less important objects are represented by smaller areas, as illustrated in *Treemap Chart*.



Treemap Chart

In a treemap chart, rectangles relate each other in a containment structure, where each rectangle is part of another rectangle. As illustrated in *Treemap Chart*, Cost of manufacturing is part of the Cost of revenues, which itself is part of Expenses. Graphically, this relationship is displayed as follows: the rectangle associated with Cost of manufacturing occupies part of the area of the rectangle associated with Cost of revenues, and the latter is part of the area for Expenses.

Expenses is the root of the model: it is not a part of a bigger budget. This is why the Expenses rectangle occupies the entire treemap chart area.

The treemap chart can also be used to display structured data sets with 4 to 10 dimensions of data. (Other types of 2D charts can show data sets with usually up to 3 dimensions of data.) In other words, if you have a table with many rows and 4 to 10 columns, the treemap is the most appropriate chart to display your data.

The treemap chart is a generalization of the pie chart. They share the feature that important objects are represented by large areas and less important objects by smaller areas. But where a pie chart displays a linear list of objects, treemaps display a tree of objects.

The input data of a treemap chart is represented by an `IlvTreeTableDataSource` instance. See *Structure of the Extended Data Model* for a description on how to connect to an `IlvTreeTableDataSource`.

IlvTreemapChartRenderer Properties

The Treemap charts have the following characteristics:

Renderer class	**`IlvTreemapChartRenderer`**
Inherits from	`IlvSimpleChartRenderer`

Setting the Area Column

The treemap chart can be applied to any entity that can be partitioned into disjointed parts.

In a treemap chart display, the most important setting is the object attribute (or model column) which is translated into the area representing the object.

The area value must meet the following requirements:

1. Must imply the *notion of importance*: the larger, the more important.
2. Must be >= 0: it is not possible to display a negative area.
3. If several objects are combined into a single object, the area value of this single object is given by the sum of the areas of each object.

   Or similarly: If an object is split into two parts, the sum of the areas of the two parts must correspond to the area of the original object.

The area, and its *notion of importance*, depends on the application domain. Here are a few examples:

- ♦ In finance, the area is the amount of money.
- ♦ For physical solids or liquids, the area is the mass (or volume) of the substance.
- ♦ In network supervision, the area can typically be the number of packets or the amount of transferred data.
- ♦ In a software profiler, the area would correspond to the CPU time spent in a particular line of code or function or source code file.
- ♦ In a filesystem space analyzer, the area would correspond to the file size.

When the objects are indivisible and have the same importance, you can assign the same area to all objects.

The area values are normalized by the treemap renderer; therefore the scaling of the area values does not matter: the rendering will not change if all area values are multiplied by a fixed constant factor.

The area column is set by means of the methods `setAreaColumn(ilog.views.chart.datax.IlvDataColumnInfo)` or `setAreaColumnName(java.lang.String)`.

Setting the Color Scheme

Besides the area, the color is the most visible attribute of an object in a treemap chart.

The expressiveness of the treemap depends on the color column and the color scheme. The color column depends on the type of application. The color scheme is used to distinguish the important properties of the objects.

The color is chosen according to what the user wants to see in the view. It is determined by a model column, called the *color column*, and a color scheme. The color column yields the numerical values that are represented through the color. The color scheme converts the numerical values into color; it has the ability to emphasize some types of value.

The color scheme is set by means of the method `setColorScheme(ilog.views.chart.renderer.IlvColorScheme)`. The color column is set by means of the methods `setColorColumn(ilog.views.chart.datax.IlvDataColumnInfo)` or `setColorColumnName(java.lang.String)`.

The color scheme is selected together with the color column. The values of this column are transformed into a color.

♦ For a real-valued color column: The color scheme SEQUENTIAL highlights high values. The color schemes DIVERGING_RED_GREEN and similar highlight the extreme values (both high and low). Whereas the color schemes AVERAGE_RED_GREEN and similar do the contrary: They emphasize the average values and don't draw the user's attention to the extreme values.

♦ For a color column whose value range wraps around (like an angle or a time-of-day), the color scheme CYCLIC_SEQUENTIAL_HUE is most appropriate.

♦ For a discrete-valued color column, that is, when the values are taken from an enumeration, the color scheme QUALITATIVE is most appropriate.

There are also color schemes that do not use a color column at all:

♦ CONSTANT which uses a single color,

♦ DEPTH which uses the nesting depth within the tree.

Color Scheme	Situation
COLORSCHEME_DEPTH	To display only the tree structure (through the containment of rectangles). This color scheme does not depend on a table column.
COLORSCHEME_SEQUENTIAL	When the high values of the column are considered more important than the low values.
COLORSCHEME_SEQUENTIAL_HUE	When all values of the column are equally important.
COLORSCHEME_CYCLIC_SEQUENTIAL_HUE	When all values of the column are equally important, and the minimum and maximum values are semantically the same. For example, if the column represents an intraday time: 00:00 h is the same as 24:00 h.
COLORSCHEME_QUALITATIVE	When the values are just enumerated values and there is no notion of low, high or approximately equal among these values.
COLORSCHEME_DIVERGING_RED_GREEN or COLORSCHEME_DIVERGING_BLUE_YELLOW	When high and low values are considered more important than the average values.
COLORSCHEME_AVERAGE_RED_GREEN or COLORSCHEME_AVERAGE_BLUE_YELLOW	When average values are considered more important than the extreme values.

Customizing Chart Renderers

Explains how to customize the graphical representation of a data model at the data point level to add additional information (like annotations) to a data point or to modify the rendering style used to draw a specific data point.

In this section

Annotations
Describes what the annotations are and how they can be set.

The rendering style
Explains how to change temporarily and/or locally the rendering style used by a renderer during the drawing process.

Annotations

An annotation is a graphical object drawn by the renderer to add additional information about a given data point. Annotation objects are instances of implementations of the `IlvDataAnnotation` interface and are handled by a chart renderer.

An annotation can be set:

- ♦ On a specific data point of a data set, by means of the `setAnnotation(ilog.views.chart.data.IlvDataSet, int, ilog.views.chart.graphic.IlvDataAnnotation)` method. This type of annotation is said to be local to the data point.
- ♦ On all the data points of a specific data set, by means of the `setAnnotation(ilog.views.chart.data.IlvDataSet, ilog.views.chart.graphic.IlvDataAnnotation)` method. This type of annotation is said to be global to the data set.
- ♦ On all the data points of all the data sets represented by a renderer, by means of the `setAnnotation(ilog.views.chart.graphic.IlvDataAnnotation)` method. This type of annotation is said to be global to the renderer.

The JViews Charts library provides several default `IlvDataAnnotation` implementations to support data labelling and icon annotation.

Setting label annotations

Data labelling is the ability to add a label annotation to a given data point. This type of annotation can be useful when you need to display the data value of a data point next to its graphical representation.

Label annotations are instances of the `IlvDataLabelAnnotation` class, an implementation of the `IlvDataAnnotation` interface that lets you display the data value or data label associated with a data point.

By default, label annotation objects use the `IlvChartRenderer` built-in data labelling mechanism. This mechanism provides the necessary API to compute a label and its location for a given data point according to predefined modes.

This API is based on the following `IlvChartRenderer` methods:

- ♦ `setDataLabeling(int)Set`

  Lets you specify what the label should display. The possible values are:

  - ● `X_VALUE_LABEL` to display the data point x-value.
  - ● `Y_VALUE_LABEL` to display the data point y-value.
  - ● `XY_VALUE_LABEL` to display both the x- and y-data point values.
  - ● `DATA_LABEL` to display the data label associated to a data point. For more information about the data set data labels, see *Using the Data Model*.
  - ● `PERCENT_LABEL` to display the contribution as a percentage of the data point. (This mode is only meaningful with a pie chart renderer.)

- ♦ `computeDataLabel(ilog.views.chart.data.IlvDataSetPoint)`

Computes the data label for a given data point according to the current data labelling mode.

- `setDataLabelLayout(int)`

  Lets you specify the position of the data label relative to the data point. The possible values are:

  - `CENTERED_LABEL_LAYOUT` to draw the label centered on the graphical representation of the data point.
  - `OUTSIDE_LABEL_LAYOUT` to draw the label outside of the graphical representation of the data point.

- `computeDataLabelLocation(ilog.views.chart.IlvDisplayPoint, java.awt.Dimension)`

  Computes the data label location for a given data point according to the current data label layout.

The `IlvDataLabelAnnotation` class provides the following services:

- customizable label

  This label is computed in the `computeText(ilog.views.chart.IlvDisplayPoint)` method. By default, the text to display is the one returned by the `computeDataLabel(ilog.views.chart.data.IlvDataSetPoint)` method.

- customizable location

  The location of the label is computed in the `computeLabelLocation(java.lang.String, ilog.views.chart.IlvDisplayPoint)` method. By default, the label location is the one computed by the `computeDataLabelLocation(ilog.views.chart.IlvDisplayPoint, java.awt.Dimension)` method.

- customizable rendering style

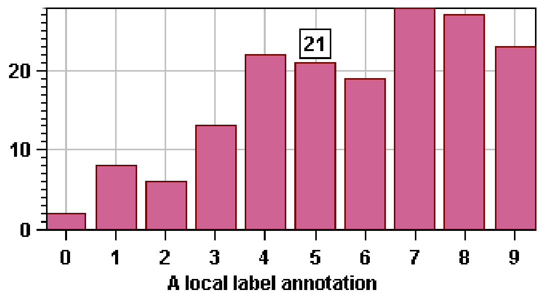
  The annotation label is drawn using an `IlvLabelRenderer` instance that defines the rendering style used to draw the label. The label renderer can be retrieved by means of the `getLabelRenderer()` method to change its default rendering attributes.

Property	Type	Description
`font`	Font	The label font.
`scalingFont`	Boolean	Whether to scale the given font.
`color`	Color	The color of the label glyphs. It may be a normal `Color` or an `IlvContrastingColor`.
`outline`	Boolean	Whether to draw in outline mode. This mode is useful to get good contrast with the background.
`opaque`	Boolean	Whether to draw a background behind the label.
`background`, `backgroundPaint`	Color/Paint	The background color or, more generally, background paint object.
`border`	javax.swing.Border	The kind of border rectangle to draw around the background rectangle.
`rotation`	double	The rotation angle applied to the entire label.
`alignment`	int	The way lines are aligned, in the case of a multiline label.
`autoWrapping`	Boolean	Whether the text is broken into lines automatically.
`wrappingWidth`	float	The maximum line width, for automatic wrapping.

Example: Setting a Label Annotation on a Given Data Point

The renderer is configured so that the label annotation displays the y-data value of the data point of index 5 on top of the representation using the default rendering style:

```
aRenderer.setDataLabeling(IlvChartRenderer.Y_VALUE_LABEL);
aRenderer.setDataLabelLayout(IlvChartRenderer.OUTSIDE_LABEL_LAYOUT);
IlvDataAnnotation annotation = new IlvDataLabelAnnotation();
aRenderer.setAnnotation(theDataSet, 5, annotation);
```



Label Annotation

Setting icon annotations

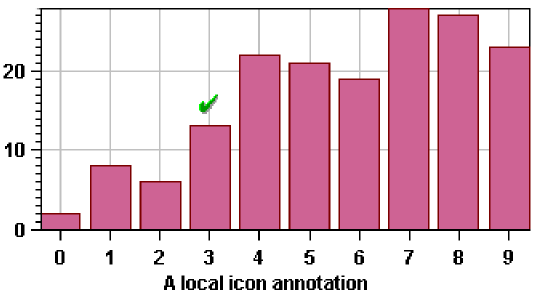
In addition to data labelling, the annotation mechanism provides a way to add icons to a renderer as an annotation. This is done by means of the `IlvDefaultDataAnnotation` class, a class that draws an object that implements the `javax.swing.Icon` interface.

The referenced object location is computed according to an anchor position and an optional offset added to this position. The anchor position is defined relative to the display point, and takes one of the `SwingConstants` direction values.

Example: Setting an Icon as a Data Point Annotation

The following example shows you how to set an icon as a data point annotation so that the bottom of the icon is drawn 5 pixels above the data point of index 3 in the data set:

```
ImageIcon icon = new ImageIcon("apply.gif");
IlvDataAnnotation annotation =
      new IlvDefaultDataAnnotation(icon, SwingConstants.NORTH, 5);
aRenderer.setAnnotation(theDataSet, 3, annotation);
```



Icon Annotation

The rendering style

The JViews Charts library provides a built-in mechanism that allows such local rendering style modification by means of the `IlvDataRenderingHint` interface. The rendering hint mechanism also allows you to dismiss the drawing of the graphical representation of a data point by returning a null rendering style.

A rendering object defines a style that should be used when a specific point of a data set is being drawn instead of the current renderer rendering style. Similarly to annotations, a rendering hint can be set:

- ♦ On a specific data point of a data set, by means of the `setRenderingHint(ilog.views.chart.data.IlvDataSet, int, ilog.views.chart.graphic.IlvDataRenderingHint)` method.
- ♦ On all the data points of a data set, by means of the `setRenderingHint(ilog.views.chart.data.IlvDataSet, ilog.views.chart.graphic.IlvDataRenderingHint)` method.
- ♦ On all the data points of all the data sets, by means of the `setRenderingHint(ilog.views.chart.graphic.IlvDataRenderingHint)` method.

The JViews Charts library provides two default implementations of the `IlvDataRenderingHint` interface:

- ♦ `IlvDefaultDataRenderingHint`

  References a rendering style applied to all the data points associated with this rendering hint.
- ♦ `IlvGradientRenderingHint`

  Draws a data point with a color computed from its y-value among a predefined range of colors.

Example: Using a Gradient Rendering Hint Object

The complete source code of this example can be found in **`<installdir>/jviews-charts86/samples/listener/src/listener/ListenerDemo.java`**.

```
double[] values = {ValueGenerator.TMIN, 0, ValueGenerator.TMAX };
Color[] colors  = {Color.blue, Color.white, Color.red};
IlvGradientRenderingHint hint =
    new IlvGradientRenderingHint(values, colors);
barR.setRenderingHint(tempDs, hint);
```

The two arrays define a binding between a value and a color that the rendering hint uses to compute the corresponding gradient. In this example, the color gradient is defined based on three colors, from blue for the minimum value to red for the maximum value, with an intermediate white color for zero value. Based on this binding, the color used to draw a data point is determined according to the data point y-value. The data point with the lowest value will be rendered as a blue bar (the first color in the gradient color array), while the data point with the highest y-value will be rendered as a red bar.

Example: Writing a new Rendering Hint

The code below shows how to write a new rendering hint so that data points with a y-value greater than a threshold are drawn with a circle marker. It implements both the `IlvDataRenderingHint` and `IlvMarker` Hint interface.

The complete source code of this example can be found in **`<installdir>/jviews-charts86/codefragments/chart/rendering-hint/src/CustomRenderingHint.java`**.

```
static final double THRESHOLD = 70.;

static class MyHint implements IlvDataRenderingHint, IlvMarkerHint
{
    public IlvStyle getStyle(IlvDisplayPoint dp, IlvStyle defaultStyle)
    {
        if (dp.getYData() > THRESHOLD)
            defaultStyle = defaultStyle.setFillPaint(IlvColor.coral);
        return defaultStyle;
    }

   public IlvMarker getMarker(IlvDisplayPoint dp, IlvMarker defaultMarker)

    {
        if (dp.getYData() > THRESHOLD)
            return IlvMarkerFactory.getCircleMarker();
        return null;
    }

}
```

Notifications from the data model

A chart renderer is notified of any modifications on its data model so that the renderer always reflects the current state of the data model.

All renderers (both composite and single) receive notifications from their data source each time a modification occurs. The following list presents the possible notifications and the associated `IlvChartRenderer` methods:

♦ a data sets are added to the data source

```
dataSetsAdded(int, int, ilog.views.chart.data.IlvDataSet[])
```

♦ a data sets are removed from the data source

```
dataSetsRemoved(int, int, ilog.views.chart.data.IlvDataSet[])
```

In addition to these notifications coming from the data source, an `IlvSingleChartRenderer` also receives notifications from its data set(s). These notifications are:

♦ the contents of an associated data set is modified

```
dataSetContentsChanged(ilog.views.chart.event.DataSetContentsEvent)
```

♦ a property of an associated data set has changed

```
dataSetPropertyChanged(ilog.views.chart.event.DataSetPropertyEvent)
```

Note: By default, a chart renderer is bound to an instance of `IlvDefaultDataSource`.

Legend items

By default, all the visible renderers with the `visibleInLegend` property set to `true` drawn in a chart appear in the chart legend (if a legend exists) as labelled symbols. The legend item that represents the renderer is an instance of the `IlvRendererLegendItem` class, an `IlvLegendItem` subclass that works with the chart renderer. The purpose of this class is to delegate the legend item drawing operation to the associated renderer, so that the legend item symbol is drawn according to the renderer type and its current rendering style.

This delegation affects both the item symbol and the label, and is handled at the chart renderer level by the following methods:

- `getLegendText(ilog.views.chart.IlvLegendItem)`

  Returns the text to display next to the symbol. The default implementation returns the name of the renderer or, in the case of a single renderer, the name of the data set if the renderer name is `null`.

- `getLegendStyle()`

  Returns the rendering style used to draw the symbol. The default implementation returns the first style contained in the styles property.

- `drawLegendSymbol(ilog.views.chart.IlvLegendItem, java.awt.Graphics, int, int, int, int)`

  Draws the legend item symbol. This method is automatically called by the renderer legend item when it is drawn.

The renderer legend items are automatically created by the chart renderer when it is added to a chart, provided that an `IlvLegend` object has been set. The items creation is performed by the `createLegendItems()` abstract method, which returns an array of legend items associated with a renderer. Being abstract, the implementation highly depends on the renderer type, but the general rule is:

- A single renderer creates one `IlvRendererLegendItem` instance.

A simple composite renderer creates one `IlvRendererLegendItem` instance for each single child renderer it references.

Scales

Explains what a scale is and how to use it.

In this section

What is a scale

A scale is displayed within a chart by a dedicated optional scale object. The base class used to represent a scale is the `IlvScale` class.

By default, scales are automatically created when a chart is constructed. You can control if scales are automatically created by means of the following constructor:

```
IlvChart(int type, boolean withScales)
```

Scales might be defined as a graphical representation of a chart axis. As such, a scale is always associated with an `IlvAxis` instance. To determine the axis that has to be associated with a scale when the scale is added to a chart, use the following methods:

♦ `setXScale(ilog.views.chart.IlvScale)` associates the specified scale with the *x*-axis.

♦ `setYScale(int, ilog.views.chart.IlvScale)` associates the specified scale with the *y*-axis of the given index.

You can retrieve the scale associated with an axis from the axis index using the following methods:

♦ `getXScale()`

♦ `getYScale(int)`

Note: Since scales are optional components, these methods may return `null` if no scale has been previously set on the specified axis.

A scale is composed of the following elements:

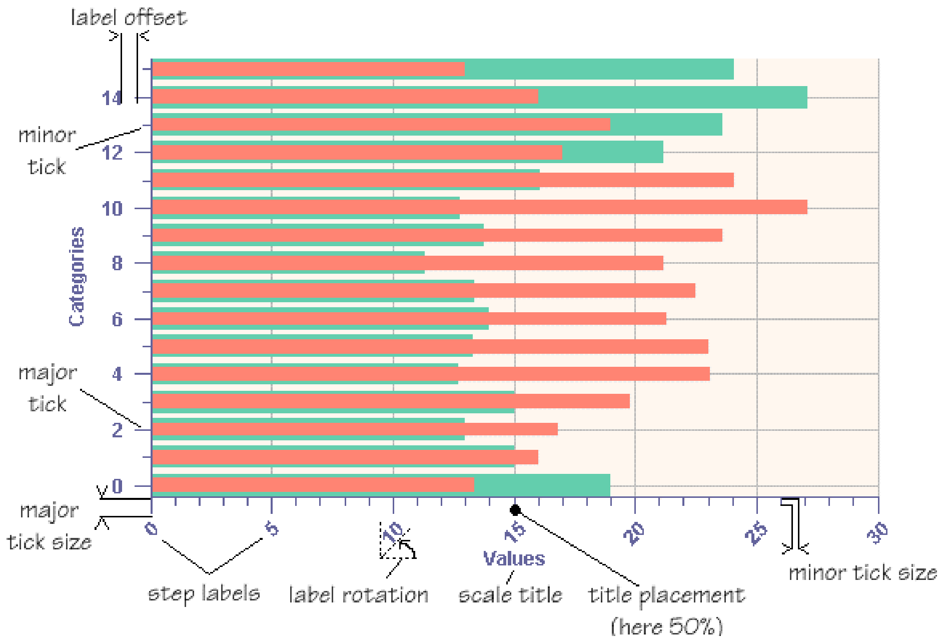
♦ An *axis* representation, which depends on the chart projection (could be a line or an arc).

♦ *Major ticks*, the marks drawn on the axis at each step of the scale.

♦ *Steps labels*, drawn next to the major ticks. These labels indicate the values of the coordinate represented by the scale.

♦ *Minor ticks*, the marks drawn on the axis at each substep of the scale.

♦ A *title*, which can be placed anywhere along the axis representation.



Scales Structure

General Properties

The following properties are defined in the `IlvScale` class to control the global visual aspect of a scale.

Property	Methods	Default value
Axis representation stroke	`getAxisStroke` `setAxisStroke`	`BasicStroke` of 1 pixel wide
Axis representation visibility	`isAxisVisible` `setAxisVisible`	true

Crossing value	`getCrossingValue`
	`setCrossingValue`
	`setCrossing`
		`IlvAxis.MIN_VALUE` or `IlvAxis.MAX_VALUE`.

Title	`getTitle` `setTitle`	`null`
Scale Visibility	`isVisible` `setVisible`	`true`
Major tick size	`getMajorTickSize` `setMajortTickSize`	`6`
Minor tick size	`getMinorTickSize` `setMinorTickSize`	`3`
Major tick visibility	`isMajorTickVisible` `setMajorTickVisible`	`true`
Minor tick visibility	`isMinorTickVisible` `setMinorTickVisible`	`true`
Position of the ticks	`getTickLayout` `setTickLayout`	`IlvScale.TICK_OUTSIDE`

Axis representation stroke

The stroke used to draw the graphical representation of the axis is defined by the `axisStroke` property. The graphical representation of the axis can be either a line or an arc, depending on the current projection and the axis attached to the scale. A scale attached to the *x*-axis of a Cartesian chart is drawn as a horizontal line, while a scale attached to the *x*-axis of a polar chart is drawn as an arc.

Position of a scale

The position of a scale is defined with respect to the scale dual axis according to a given data value. This data value is the value on the dual axis where the scale axis crosses it, and is called the scale crossing value.

For example, in a default Cartesian chart, the *x*-scale crossing value is set to `IlvAxis.MIN_VALUE`, meaning that the value where the *x*-scale crosses the *y*-axis is equal to the minimum value of the *y*-axis.

Title of a scale

A scale can have an optional title. A scale title is specified as a string by means of the `setTitle(java.lang.String, double)` method.

A scale title supports the following features:

♦ It can be rotated by a given angle.

  This rotation angle can be specified either when the text title is initialized or later by means of the `setRotation(double)` method.

♦ It is placed anywhere along the scale axis at a given position.

  This position is expressed as a percentage of the axis length, and can be set using the `setTitlePlacement(int)` method.

Visibility of a scale

The visibility of the scale is defined by the `axisVisible` property. Depending on the value of this property, the axis representation can be visible or hidden. Showing or hiding a scale affects the drawing area bounds, therefore a layout is automatically performed on the chart area when the value changes.

Note: Setting the property to `false` hides only the axis, it does not affect the scale ticks visibility.

Size of the major and minor ticks

The size of the major and minor ticks is expressed in pixels and can be changed dynamically using the `setMajorTickSize(int)` and `setMinorTickSize(int)` methods.

The major tick visibility and minor tick visibility are defined by the `majorTickVisible` and `minorTickVisible` properties. Showing or hiding scale ticks affects the scale bounds and the drawing area bounds, therefore a layout is automatically performed on the chart area when these properties change.

Position of the ticks relative to the axis

The position of the ticks relative to the axis is defined by the `tickLayout` property. The following positions are possible:

- ♦ `TICK_INSIDE`

  The ticks extend inside the data plotting area.

- ♦ `TICK_OUTSIDE`

  The ticks extend outside the data plotting area.

- ♦ `TICK_CROSS`

  The ticks cross the axis and extend both inside and outside the data plotting rectangle.

Note: By default, the position of the ticks of a scale is set to `TICK_OUTSIDE` and can be changed using the `setTickLayout(int)` method.

Computing Scale Graduation

The IlvStepsDefinition abstract class

The base class used to represent a scale steps definition is the `IlvStepsDefinition` abstract class. This class defines how scale steps are computed and how steps values are translated into a label. For more information, see *Computing scale steps*.

By default, a scale automatically creates a scale definition when it is constructed. You can also specify by hand the steps definition a scale should use, by means of the IlvScale.setStepsDefinition method.

Standard numerical values

The `IlvDefaultStepsDefinition` class provides a default numbering of numeric steps. The steps and substeps values are defined by:

- ♦ a step unit, which corresponds to the value between two consecutive steps,
- ♦ a substep unit, which corresponds to the value between two consecutive substeps.

The step and substep unit values are either automatically computed at run time or explicitly set by means of the `setStepUnit` and `setSubStepUnit` methods. Calling one of these methods disables the automatic steps calculation mode for the corresponding unit.

The steps labels are computed according to a number format, instance of `java.text.NumberFormat`. By default, the format to use is automatically computed by the steps definition. You can disable the automatic format calculation by manually specify the number format to use by means of the following method:

```
IlvDefaultStepsDefinition.setNumberFormat()
```

The `IlvScale` natively supports this default steps definition class. The method `setStepUnit (java.lang.Double, java.lang.Double)` provides a shortcut to the methods `setStepUnit` and `setSubStepUnit` on the `IlvStepsDefinition` object

Time values

The `IlvTimeStepsDefinition` class provides a default numbering for time values. The steps values are defined by a step unit, expressed as a time value. This time unit is an instance of the `IlvTimeUnit` class, and default implementations are provided to handle one of the following predefined units:

```
IlvTimeUnit.SECOND
IlvTimeUnit.MINUTE
IlvTimeUnit.HOUR
IlvTimeUnit.DAY
IlvTimeUnit.WEEK
IlvTimeUnit.MONTH
IlvTimeUnit.QUARTER
IlvTimeUnit.YEAR
IlvTimeUnit.DECADE
IlvTimeUnit.CENTURY
```

Besides these predefined time units, the JViews Charts package provides a specialized `IlvTimeUnit` subclass, the `IlvMultipleTimeUnit` class, that allows you to define new time units as a multiple of predefined time units.

For example, the following code creates a time unit equal to 15 minutes:

```
IlvTimeUnit unit = new IlvMultipleTimeUnit(IlvTimeUnit.MINUTE, 15);
```

The `IlvMultipleTimeUnit` class can also be used to easily change the default implementation of a predefined `IlvTimeUnit` subclass. This new implementation is not obtained by subclassing but by defining a new `IlvMultipleTimeUnit`. This new multiple time unit must be based on the predefined unit to modify, with a multiplier factor of 1, and must override the proper methods.

For example, the following code shows how to modify the default format string of the `IlvTimeUnit.MONTH` class so that it returns the full month name instead of the abbreviated form:

```
IlvTimeUnit monthUnit =
    new IlvMultipleTimeUnit(IlvTimeUnit.MONTH, 1) {
        public String getFormatString() {
            return “MMMMM”;
        }
    };
```

The step unit is either automatically computed at run time or explicitly set by means of the `setUnit(ilog.views.chart.IlvTimeUnit)` method. In the latter case, the automatic time unit calculation mode is disabled.

Time units chosen during the automatic unit calculation process must be specified to the time steps definition by means of the `IlvTimeStepsDefinition.setAutoUnits` method. This method takes an `IlvTimeUnit` array as a parameter that contains all the units to consider. By default, all predefined units are taken into account.

The following code shows how to replace the predefined month unit by a new month unit in the automatic unit calculation process constraining the units from hour to month:

```
IlvTimeUnit[] autoUnits = {
    IlvTimeUnit.HOUR,
    IlvTimeUnit.DAY,
    IlvTimeUnit.WEEK,
    monthUnit
};
timeStepsDefinition.setAutoUnits(autoUnits);
```

The step labels are computed using a `java.text.DateFormat` instance. This format is dependent on the current time unit and is performed in the `computeLabel(double)` method.

The `IlvScale` natively supports the `IlvTimeStepsDefinition` class. The method `setTimeUnit(ilog.views.chart.IlvTimeUnit)` installs an `IlvTimeStepsDefinition` instance, if needed, and provides a shortcut to the `setUnit` method on the `IlvTimeStepsDefinition` object.

Displaying categories

The `IlvCategoryStepsDefinition` class provides a default numbering for scales displaying categories.

The steps are computed so that there is one step for each category and one substep between two consecutive categories. The steps labels are computed to display either the categories number or the data labels of a data set.

The `IlvScale` natively supports the `IlvCategoryStepsDefinition` class. The method `setCategory(ilog.views.chart.data.IlvDataSet, boolean)` installs an `IlvCategoryStepsDefinition` instance.

Displaying logarithmic scales

The `IlvLogarithmicStepsDefinition` class provides a default numbering for scales with a logarithmic axis transformer.

The `IlvScale` natively supports the `IlvLogarithmicStepsDefinition` class. The method `setLogarithmic(double)` installs an `IlvLogarithmicStepsDefinition` instance and also sets the axis transformer that is applied to data points to a logarithmic one.

The shorthand methods

To specify which data type the scale is handling, and which steps definition class to use, the `IlvScale` class provides the following methods:

- `setStepUnit(java.lang.Double, java.lang.Double)`

  Handles numerical values and uses an `IlvDefaultStepsDefinition` instance.

- `setTimeUnit(ilog.views.chart.IlvTimeUnit)`

  Handles time values and uses an `IlvTimeStepsDefinition` instance.

- `setCategory(ilog.views.chart.data.IlvDataSet, boolean)`

  Handles categories and uses an `IlvCategoryStepsDefinition` instance.

- `setLogarithmic(double)`

Handles logarithmic values and uses an `IlvLogarithmicStepsDefinition` instance. This method also sets the axis transformer that is applied to data points to a logarithmic one.

Computing scale steps

The scale steps are computed according to a step unit. The step unit is defined as the increment between two consecutive steps, and its value depends on the data type the scale is handling: numerical value or time unit (as a day, a week, and so on).

The `IlvStepsDefinition` class provides an iterator-like API to iterate through the graduations in a step-by-step way by means of the following methods:

- ♦ `nextStep(double)`

  Returns the step value that immediately follows the specified value.

- ♦ `previousStep(double)`

  Returns the previous step value immediately before the specified value.

- ♦ `incrementStep(double)`

  Increments the specified step.

Note: The two last methods are abstract and are implemented by subclasses, depending on their data type.

Translating steps values into a label

Scale steps values are translated into string by the scale steps definition. The translation depends on the concrete implementations of the `IlvStepsDefinition` class and is performed by means of the `computeLabel(double)` method.

Scale Labels

General properties

The following table shows the properties defined by the `IlvScale` class related to the scale labels.

Property	Methods	Default Value
Label color	`getLabelColor` `setLabelColor`	`foreground`
Label offset	`getLabelOffset` `setLabelOffset`	`3`
Label rotation	`getLabelRotation` `setLabelRotation`	`0`
Label visible	`isLabelVisible` `setLabelVisible`	`true`
Skip labels when overlapping	`isSkippingLabel` `setSkippingLabel`	`false`
Label format	`getLabelFormat` `setLabelFormat`	`null`

Label color

The color used to draw the scale labels is defined by the `labelColor` property and by default equals the scale foreground color.

Label offset

Labels are drawn next to the tick marks spaced from a given offset expressed in pixels. This offset is defined by the `labelOffset` property and is set to 3 by default.

Label rotation

Steps labels can be rotated from a given rotation angle whose value is expressed in degrees and clockwise oriented. This angle is defined by the `labelRotation` property and is set to `0` by default. You can change it by means of the `setLabelRotation(double)` method.

Label visibility

The labels can be visible or hidden depending on the value of the `labelVisible` property. By default, the property is set to `true`.

Skipping labels when overlapping

Depending on the values and on the way labels are formatted, it may occur that steps labels overlap each other because of the text length. To prevent this, the `IlvScale` class provides an automatic mechanism that computes the number of steps to skip between each label so that they do not overlap. This mechanism is defined by the `skippingLabel` property and is disabled by default.

Defining the label format

Steps values are displayed in a scale as a formatted text computed from a value format. This format is defined by the `labelFormat` property. By default, the `labelFormat` property is set to `null` and the steps labels are computed from the steps values by the scale steps definition object invoking its `computeLabel(double)` method.

To set another value format, use the method:

```
IlvScale.setLabelFormat(IlvValueFormat)
```

To change the way labels are computed, override the method:

```
IlvScale.computeLabel(double)
```

Example: specifing a customized value format using the API

The complete source code of this example can be found in **`<installdir>/jviews-charts86/codefragments/chart/value-format/src/ScaleLabelsExample.java`**.

This example illustrates how an `IlvValueFormat` can be used to handle discrete time series as categories. Indeed, while an `IlvTimeScaleDefinition` and the `IlvTimeUnit` are particularly well-suited for continuous time series, it is better to use the default steps definition of a scale when dealing with discrete time series to have a category-like step numbering and labeling.

For this purpose, you will write the `CategoryTimeFormat` class, your own `IlvValueFormat` implementation. This class will convert a data point index into a corresponding date and format it accordingly.

To do so, the class needs to know:

♦ For the date conversion:

- The time origin from which an index is converted into a date.
- The time step of the categories
- For the label formatting:

♦ The format (a pattern as defined by the `java.text.DateFormat` class) used to convert the date into a string.

For example, with a time origin equals to 01/01/2000 and a time step corresponding to a year, the index 0 will be converted into 01/01/2000, the index 1 into 01/01/2001, the index 2 into 01/01/2002 and so on, that is:

```
new date = origin + 'index' time step.
```

Base on these requirements, the corresponding Java™ class is:

```
private static class CategoryTimeFormat implements IlvValueFormat
{
    private int step;
    private Calendar cal = (Calendar)Calendar.getInstance().clone();
    private SimpleDateFormat fmt = new SimpleDateFormat();
    private Date origin;

    /**
     * Initialize a new <code>CategoryTimeFormat</code>.
     * @param origin The origin of the categories.
     * @param step The step of the categories. Should be a valid
     * Calendar field value.
       * @param unit An optional time unit used to format the label. If
     * null, the default format is used.
     */
    public CategoryTimeFormat(Date origin, int step, IlvTimeUnit unit)
    {
        this.origin = origin;
        this.step = step;
        // set the format pattern, if specified.
        if (unit != null)
            fmt.applyPattern(unit.getFormatString());
    }

    /**
     * Formats the specified value into a string.
     */
    public String formatValue(double value)
    {
        // compute the date corresponding to the given index.
        cal.setTime(origin);
        cal.add(step, (int)value);
        return fmt.format(cal.getTime());
    }
}
```

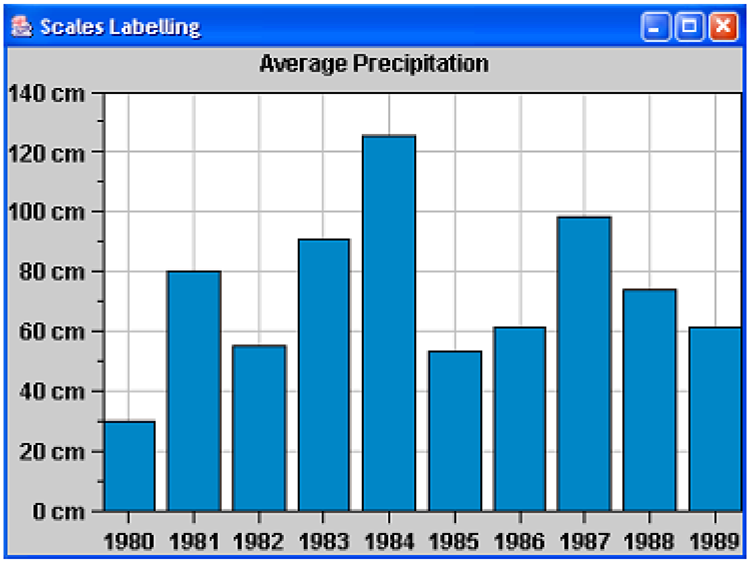
A sample application using this class is shown below. The application displays a bar chart of the average precipitation from 1980 to 1990. The data model represents a discrete time series. In other words, data is arranged along the x-axis by time categories. That means that the data series contains y-values only, the x-values of the data set will be computed according to the data point indices. In order to compute the date corresponding to a category index, we will use a `CategoryTimeFormat` instance to compute x-scale labels. Since the series is distributed year by year, from 1980 to 1990, we configure the `CategoryTimeFormat` with a time origin equals to 1980 and a time step of a year.

Furthermore, we also want to display the y-scale labels as "<step value> cm". To do so, we set a an `IlvValueFormat` that append the " cm" string to the string representation of the step values.

Here is the source code of the application (note that we have removed the comments from the original source code to ease reading):

```
import ilog.views.chart.*;
import ilog.views.chart.data.*;
import ilog.views.chart.renderer.IlvSingleBarRenderer;
import javax.swing.JFrame;
import java.text.*;
import java.util.*;

public class ScaleLabelsExample
{
  public static void main(String[] args)
  {
    final double[] yvalues = {30,80,55,91,125,53,61,98,74, 61};
    IlvDataSet dataSet = new IlvDefaultDataSet("Series A", yvalues);
    IlvChart chart = new IlvChart();
    chart.setHeaderText("Average Precipitation");
    chart.addRenderer(new IlvSingleBarRenderer(), dataSet);
    chart.getYAxis(0).setDataMin(0);

    //-- Handle scales labels.
    IlvScale xscale = chart.getXScale();
    // We set a CategoryTimeFormat on the x-scale to handle the data
    // point indices as "time categories".
    Calendar cal = Calendar.getInstance();
    cal.set(1980, 0, 1);
    xscale.setLabelFormat(new CategoryTimeFormat(cal.getTime(),
                                                  Calendar.YEAR,
                                                  IlvTimeUnit.YEAR));
    // We also set a custom IlvValueFormat on the y-scale so that
    // labels are displayed as: value + " cm".
    IlvScale yscale = chart.getYScale(0);
    yscale.setLabelFormat(new IlvValueFormat() {
        private NumberFormat numformat = NumberFormat.getInstance();
        public String formatValue(double value) {
            return numformat.format(value) + " cm";
        }
    });
    // the gui
    JFrame frame = new JFrame("Scales Labelling");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.getContentPane().add(chart);
    frame.setSize(400,300);
    frame.setVisible(true);
}
```

You can see the result in the following figure:


Scales Labelling
Average Precipitation
140 cm
120 cm
100 cm
80 cm
60 cm
40 cm
20 cm
0 cm
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989

Scale Annotations

The complete source code of this example can be found in **<installdir>/jviews-charts86/ codefragments/chart/scale-annotation/src/ScaleAnnotation.java**.

The `IlvScale` class supports a special kind of decoration that can be used to indicate a data value on a scale as an annotation.

Such annotation objects are instances of the `IlvScaleAnnotation` class. This class allows you to draw a string representation of a data value over the scale labels. The string may be either a specific label explicitly set or the data value formatted into a string.

Here are some examples:

♦ To draw the value along the x-axis of a given data point as a formatted string, use the following code:

```
IlvAnnotation annotation = new IlvScaleAnnotation(xValue);
chart.getXScale().addAnnotation(annotation);
```

♦ To display a specific text instead of the data value, the code would be:

```
annotation.setText("Release date");
```

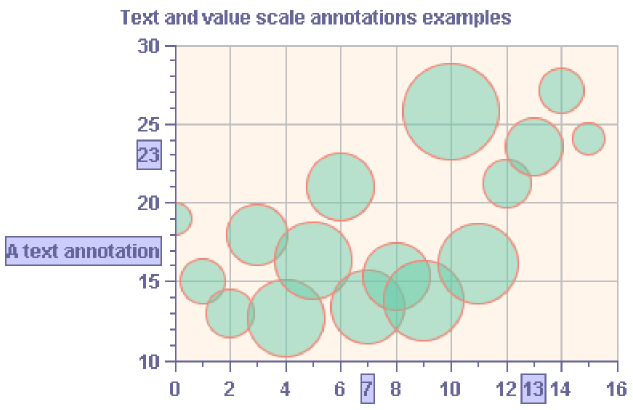
♦ To remove the scale annotation, use the following method:

```
chart.getXScale().removeAnnotation(annotation);
```

The text annotation is drawn by a dedicated `IlvScaleAnnotation` attribute, which is an instance of `IlvLabelRenderer`. This label renderer object handles all the properties related to the label drawing, like border, text color, font, and so on, and can be retrieved by invoking the `getLabelRenderer()` method.

For example, to draw the annotation as a white opaque bordered label, use the following code:

```
annotation.getLabelRenderer().setBorder(BorderFactory.createLineBorder(Color.
bl
ack));
annotation.getLabelRenderer().setOpaque(true);
annotation.getLabelRenderer().setBackground(Color.white);
```



Scales Annotations

Decorations

Explains how to draw and write decorations.

In this section

Drawing decorations

Decorations are drawn according to a drawing order. The drawing order lets you control the position of a given decoration in the drawing queue of a chart. This drawing order is used to define the position of the decoration relative to:

- ♦ Other chart decorations.
- ♦ Graphical representations of the chart data.

A chart handles decorations as an ordered list according to the decorations drawing order: decorations with the lowest drawing order are drawn first, decorations with the highest drawing order are drawn last.

The drawing order also defines whether a decoration should be drawn above or below the graphical representations of the chart data: decorations with a negative drawing order are drawn below the chart representations, while decorations with a zero or positive drawing order are drawn above the chart representations.

The `IlvChart` class defines two drawing order values that are used as default values for the `drawOrder` properties in the JViews Charts library:

- ♦ The `IlvChart.DRAW_ABOVE` value defines the first drawing order above the data graphical representations.
- ♦ The `IlvChart.DRAW_BELOW` value defines the first drawing order below the data graphical representations.

Predefined decorations

The following types of decorations are already provided by the JViews Charts library:

- `IlvDataIndicator`: an indicator for a particular value or a range of values. It is usually represented by a horizontal or vertical line or a rectangle, with an optional label.

  `IlvThresholdIndicator`: an indicator for a particular value or a range of values, together with a display of the value at the scale.

- `IlvLabelDecoration`: a simple label.

- `IlvImageDecoration`: an image or icon.

- `IlvGraphicDecoration`: a graphic object as defined in the IBM® ILOG® JViews Framework. This class is a wrapper around a graphic object of type `IlvGraphic`. See the reference manual for information about the predefined subclasses of `IlvGraphic`.

Writing a new decoration

The complete source code can be found in **`<installdir>/jviews-charts86/samples/stock/src/stock/Stripes.java`**.

The `draw` method is automatically called by an `IlvChart` to draw the decoration (provided that the decoration has been added to a chart). Depending on its drawing order property, a decoration is drawn either before or after the graphical representation of the chart.

Write a decoration that displays stripes regularly spaced, and aligned on the graduations of an associated scale. The scale type determines the stripes direction (horizontal stripes for an ordinate scale and vertical stripes for an abscissa scale). The stripes width should be equal to the stripes spacing and equal to the width of one major scale graduation. Furthermore, the stripes should be painted using a customizable fill style and the result should be independent of the chart projector.

1. Extend the `IlvChartDecoration` class.

   ```
   public class Stripes extends IlvChartDecoration
   {
     ...
   }
   ```

2. Define two data members: a reference on the associated scale and a rendering style.

   ```
   private IlvStyle fillStyle;
   IlvScale scale;
   ```

3. Initialize the scale at initialization time and pass it as a parameter to the constructor.

   ```
   public Stripes(IlvScale scale, Paint fillPaint)
   {
       this.scale = scale;
       setFillPaint(fillPaint);
   }
   ```

   The scale associated with the decoration should not change over the life of the instance.

4. Provide `set` and `get` accessors on the fill rendering style in addition to the constructor initialization. You want to allow the rendering style to be changed at any time.

   ```
   public void setFillPaint(Paint paint)
   {
       fillStyle = getFillStyle().setFillPaint(paint);
   }

   public final Paint getFillPaint()
   {
   ```

```
    return getFillStyle().getFillPaint();
}
```

Define the private `getFillStyle()` method:

```
private IlvStyle getFillStyle()
{
    if (fillStyle == null)
        fillStyle = new IlvStyle(Color.lightGray);
    return fillStyle;
}
```

Note: The `IlvStyle` class is an immutable class. Changing an attribute of an `IlvStyle` instance actually creates a copy of this instance with the specified attribute. This choice has been made to prevent unexpected side-effects when changing attributes of a shared `IlvStyle` instance.

Now everything is ready to draw the decoration.

The stripes should be aligned on the scale graduations, have a width equal to one major step of the scale graduation, and spaced from the same value. From these properties, you see that the position on the scale of a stripe depends on the position of the previous one, and that can be expressed as an interval between two values.

5. To compute the data interval covered by a stripe, use the following method to return the interval next to the `IlvDataInterval` specified as a parameter:

```
protected IlvDataInterval nextStripe(IlvDataInterval itv)
{
    IlvStepsDefinition def = scale.getStepsDefinition();
    if (itv == null) {
        itv = getAxis().getVisibleRange();
        double v = def.previousStep(itv.getMin());
        itv.setMin(def.incrementStep(v));
        itv.setMax(def.incrementStep(itv.getMin()));
    } else {
        itv.setMax(def.incrementStep(def.incrementStep(itv.getMax())));
        itv.setMin(def.incrementStep(def.incrementStep(itv.getMin())));
    }
    return itv;
}
```

Scale graduations are computed by a dedicated object, instance of the `IlvStepsDefinition` class. Since we need to compute the stripe position according to these graduations, we first get the steps definition of the scale, and then we compute the interval corresponding to the next stripe depending on the previous interval:

♦ if it is the first stripe, the interval will cover an area equal to `[visibleMin+delta, visibleMin+2*delta]`, where delta is one major step.

- if the stripe comes after a previous stripe, it should cover the area between `[previousMin+2*delta, previousMax+2*delta]`, where delta is one major step.

Now that we know the data interval corresponding to a given stripe, we can write the `draw` method. Remember, we want to be independent of the chart projector: our decoration should be used either with a Cartesian projector or a polar projector. Since we do not want to write projector-dependent code, we use the `getShape(ilog.views.chart.IlvDataWindow, java.awt.Rectangle, ilog.views.chart.IlvCoordinateSystem)` method that returns a shape (in the screen coordinate system) corresponding to a specified data window. This shape contains all the data points of the data window projected by the current chart projector.

6. Draw the decoration.

The implementation of the `draw` method is as follows:

```
public void draw(Graphics g)
{
    IlvChart chart = getChart();
    if (chart == null)
        return;

    IlvDataInterval itv = nextStripe(null);
    IlvDataWindow   w   = null;
    if (getAxis().getType() == IlvAxis.X_AXIS) {
        w = new IlvDataWindow(itv, chart.getYAxis(0).getVisibleRange());

    } else {
        w = new IlvDataWindow(chart.getXAxis().getVisibleRange(), itv);
    }

    IlvChartProjector prj        = getChart().getProjector();
    IlvCoordinateSystem coordSys = getChart().getCoordinateSystem(0);
    Rectangle plotRect           = getChart().getChartArea().getPlotRect
();

    IlvStyle style = getFillStyle();
    while (itv.getMin() < getAxis().getVisibleMax()) {
        style.fill(g, prj.getShape(w, plotRect, coordSys));
        if (getAxis().getType() == IlvAxis.X_AXIS)
            w.xRange = nextStripe(itv);
        else
            w.yRange = nextStripe(itv);
    }
}
```

Depending on the scale type (x- or y-scale), we initialize the data window corresponding to a stripe with the visible range of the corresponding axis and iterate on the stripes until the maximum visible range of the associated axis is reached.

Displaying and Writing a Grid

Describes how to display and write a grid.

In this section

What is a grid

A grid is a graphical indicator of data values. A grid is attached to an axis and is composed of:

♦ Major gridlines, with an associated rendering style.

♦ Minor gridlines, with an associated rendering style.

A grid is displayed within a chart by a dedicated object, defined by the `IlvGrid` class.

The purpose of the `IlvGrid` class is to handle the graphical representation of a grid. The default behavior of the `IlvGrid` class automatically handles the grid graphical representation according to the type of chart. For example, a Cartesian chart has a rectangular grid for both the *x*- and *y*-axis, while a polar chart has a circular *x*-grid. It also uses by default the major and minor steps of a scale to draw the major and minor ticks of the scale. By default, grids are automatically initialized when a chart is created.

You can retrieve a grid from its axis using the methods `getXGrid()` and `getYGrid(int)`.

You can change the grid of an axis using the methods `setXGrid(ilog.views.chart.IlvGrid)` and `setYGrid(int, ilog.views.chart.IlvGrid)`.

General Properties

The following table shows the properties defined for displaying a grid.

Property	Methods	Default Value
Grid visibility	`isVisible` `setVisible`	`true`
Major lines visibility	`isMajorLineVisible` `setMajorLineVisible`	`true`
Minor lines visibility	`isMinorLineVisible` `setMinorLineVisible`	`false`
Drawing order relative to the drawing of the graphical representation of data	`getDrawOrder` `setDrawOrder`	`IlvChart.DRAW_BELOW`

Major lines rendering style	`getMajorStroke`
	`setMajorStroke`
	`getMajorPaint`
	`setMajorPaint`
		`defaultGridColor`

Minor lines rendering style	`getMinorStroke`
	`setMinorStroke`
	`getMinorPaint`
	`setMinorPaint`
		`defaultGridColor`

Default gridlines color	`setDefaultGridColor` `getDefaultGridColor`	`Color.lightGray`

Grid visibility

The `visible` property defines the global grid visibility: a grid is visible when either minor or major gridlines are visible. The visibility of the major and minor gridlines is described by the `majorLineVisible` and `minorLineVisible` properties, respectively. By default, only major gridlines are visible.

Drawing order

As for all the decorations, you can control the drawing order of the grids with respect to the charts representation and to other decorations. This drawing order is defined by the `drawOrder` property. For more information on ordering decorations see *Decorations*.

Default gridline color

The gridlines are drawn by default according to the value of the `defaultGridColor` property. This is a property of the `IlvGrid` class (static) and can be easily changed to set the default gridlines rendering styles.

Writing a new grid

The complete source code can be found in **`<installdir>/jviews-charts86/samples/realtime/src/realtime/SimpleGrid.java`**.

You can create a new type of grid by subclassing the `IlvGrid` class and overriding one or both of the `draw` methods.

A grid draws its gridlines according to the gridline values. These values are used to determine the anchor point of a gridline on the associated axis and are expressed in the data coordinate system.

By default, the `draw(java.awt.Graphics)` method computes the gridline values to match the step of the scale of the associated axis and invokes the `draw(java.awt.Graphics, ilog.views.chart.util.IlvDoubleArray, boolean)` method with these values as parameter to perform the drawing operations.

Change the way the gridline values are computed by default to have gridlines equally spaced from a specified delta, and no longer snapped on the major ticks of the scale graduation.

1. Extend the `IlvGrid` class.

   ```
   class SimpleGrid extends IlvGrid
   ```

   The spacing between the two gridlines is specified at construction time as a constructor parameter and expressed in the data coordinate system by a double value.

   The grid constructor is:

   ```
   double spacing;
   public SimpleGrid(Paint majPaint, double spacing)
   {
       super(majPaint);
       this.spacing = spacing;
   }
   ```

2. Override the `draw(Graphics)` method.

   Since you want to change the way the gridline values are computed, not the way they are drawn, you only need to override the `draw` method to change its default implementation.

   This method computes the gridline values according to the current visible range, and iterates over it with a step equal to the specified spacing:

   ```
   public void draw(Graphics g)
    {
        if (getChart() == null || spacing <= 0)
            return;
        IlvDataInterval itv = getAxis().getVisibleRange();
        IlvDoubleArray gridlines = new IlvDoubleArray(16);
        double val = Math.ceil(itv.getMin()/spacing)*spacing;
        while (itv.isInside(val)) {
   ```

```
        gridlines.add(val);
        val += spacing;
    }
    if (gridlines.size() > 0)
        draw(g, gridlines, true);
}
```

The implementation is quite simple: starting from the visible range minimum bound, you iterate on the gridline values by adding the expected spacing to the previous value until the visible range maximum bound is reached. Then, if at least one gridline value has been computed, you call the `draw(Graphics, IlvDoubleArray, boolean)` method to draw the grid with your own values.

Another example of a custom `IlvGrid` subclass can be found in **`<installdir>/jviews-charts86/samples/monitor/src/monitor/MemoryMonitor.java`**. In this example, the `IlvGrid` class is extended in order to display a fixed number of gridlines. You can find the source code of this class in the **`FixedGrid.java`** file.

Displaying Data Indicator

Introduces the data indicator and its general properties.

In this section

Data Indicator

A data indicator is a graphical indicator of a data value. The data value to represent can be of different types:

- a value along the x-axis,
- a value along the y-axis,
- a data interval along the x-axis,
- a data interval along the y-axis,
- a data window.

The graphical representation of a data indicator is composed of:

- A delimiter that indicates the data area (a simple line for an x- or y-value or a more complex shape that represents a data interval or a data window that depends on the projection).
- An optional label.

Data indicators are instances of the `IlvDataIndicator` class and are handled directly by a chart. The `IlvDataIndicator` class is a subclass of `IlvChartDecoration`.

If you want to add a data indicator to a chart, use the method `addDecoration(ilog.views.chart.IlvChartDecoration)`.

If you want to remove a data indicator from a chart, use the method `removeDecoration(ilog.views.chart.IlvChartDecoration)`.

The complete source code of this example can be found in **`<installdir>/jviews-charts86/codefragments/chart/data-indicator/src/DataIndicator.java`**.

```
        // A data indictor that highlights the range [5,13] along the x-axis.
```

```
        IlvDataIndicator indic = new IlvDataIndicator(-1, new
IlvDataInterval(5,13), null);
```

```
        // set the rendering style
```

```
        indic.setStyle(INDICATOR_STYLE);
```

```
        chart.addDecoration(indic);

        // A data indicator that indicates the value 23 as a threshold line
        // along the y-axis. It displays the 'Threshold' value.
        indic = new IlvDataIndicator(0, 23, "Threshold");
        indic.setStyle(INDICATOR_STYLE);
        // change its draw order so that it is drawn ABOVE renderers
        indic.setDrawOrder(IlvChart.DRAW_ABOVE);
        // customizer its label renderer
        indic.getLabelRenderer().setOpaque(true);

indic.getLabelRenderer().setBorder(BorderFactory.createLineBorder(CHART_FOREGRO
UND_COLOR));
        indic.getLabelRenderer().setBackground(INDIC_FILL_COLOR);
        chart.addDecoration(indic);
```

General Properties

The following table shows the properties of the `IlvDataIndicator` class.

Property	Methods	Default value
Indicator type	`getType`	
Data value for which the data indicator is drawn. This property is used when the indicator type is either `X_VALUE` or `Y_VALUE`.	`setValue` `getValue`	
Data range for which the data indicator is drawn. This property is used when the indicator type is either `X_RANGE` or `Y_RANGE`.	`setRange` `getRange`	
Data window for which the data indicator is drawn. This property is used when the indicator type is `WINDOW`.	`setDataWindow` `getDataWindow`	
Text displayed over the delimiter.	`setText` `getText`	`null`
Style used to draw the data indicator.	`setStyle` `getStyle`	`chart area plot style`
Drawing order relative to the drawing of the graphical representation of data and to the other decorations.	`setDrawOrder` `getDrawOrder`	`IlvChart.DRAW_BELOW`
Visible	`setVisible` `isVisible`	`true`

Indicator type

The indicator type property is initialized at construction time depending on the constructor that is used. Setting the type does not reset the value: the last value used for the new type, if any, is used.

Data represented by the indicator

The data represented by the indicator can be changed dynamically using the corresponding `setValue(double)`/`setRange(ilog.views.chart.IlvDataInterval)`/`setDataWindow(ilog.views.chart.IlvDataWindow)` methods. If the data type to represent is different from the current one when invoking one of these methods, the indicator type is automatically updated according to the type of the data value. Furthermore, when you invoke one of these methods, you automatically update the chart drawing area.

Optional label

An optional label can be associated with a data indicator and drawn over the delimiter. The text to display can be set either at construction time or dynamically using the `setText(java.lang.String)` method.

By default, the location of the label is centered vertically or/and horizontally with respect to the data value/range/window. You can change this behavior by overriding the `computeLabelLocation(ilog.views.chart.IlvDataWindow)` method.

Drawing order

As for all the decorations, you can control the drawing order of the indicator with respect to the charts representation and to the other decorations. This drawing order is described by the `drawOrder` property. See *Decorations* for more information on ordering decorations.

Displaying an image

Describes how to display an image by using the IlvImageDecoration class.

In this section

The IlvImageDecoration class

The `IlvImageDecoration` is a predefined decoration class that displays an image within the plotting area of a chart.

An image can be drawn according to three different modes:

♦ TILED: The image is drawn as a replicated pattern in the plot area.

♦ SCALED: The image is scaled so that it fills the plot area.

♦ ANCHORED: The image is drawn at a fixed position.

The predefined position for the ANCHORED mode is defined as one of the `javax.swing.Swing` Constants compass directions.

They are the following:

♦ CENTER

♦ NORTH

♦ NORTH_EAST

♦ EAST

♦ SOUTH_EAST

♦ SOUTH

♦ SOUTH_WEST

♦ WEST

♦ NORTH_WEST

Here is an example of code that creates and adds a SCALED image decoration to a chart:

```
IlvChart chart = ...;
    try {
        java.net.URL url = new File("logo.gif").toURL();
        // The last parameter is taken into account only in ANCHORED mode.
       IlvImageDecoration deco = new                       IlvImageDecoration
(url,IlvImageDecoration.SCALED, 0);
        chart.addDecoration(deco);
    } catch (java.net.MalformedURLException e) {
        e.printStackTrace();
    }
```

Note: To preserve the image during serialization, you must use the `IlvImageDecoration` constructor that takes a `java.net.URL` as the location of the image. Other constructors do not preserve the image.

The following table shows the properties of the `IlvImageDecoration` class.

Property	Methods	Default Value
Drawing mode.	`getMode`	
Image location.	`getAnchor` `setAnchor`	`SwingConstants.CENTER`
Drawing order relative to the drawing of the graphical representations of data and to other decorations.	`getDrawOrder` `setDrawOrder`	`IlvChart.DRAW_BELOW`
Visibility.	`isVisible` `setVisible`	`true`

Drawing mode

The drawing mode property is initialized at construction time. It must be one of the TILED, SCALED or ANCHORED `IlvImageDecoration` constants. If the latter is used, a predefined position must also be specified by means of the anchor property.

Image anchor

When using the ANCHORED drawing mode, the image location within the plot area must be specified either at construction time or by means of the `setAnchor(int)` method. By default, an anchored image is drawn at the center of the plotting area.

Drawing order

You can control the drawing order of the image decoration with respect to the charts representation and to other decorations. This drawing order is defined by the `drawOrder` property. See *Decorations* for more information on ordering decorations.

Interacting With Charts

Provides detailed information on the chart interactors and explains how to handle them.

In this section

Chart Interactors

Describes the interactors defined in the JViews Charts library. For each chart interactor, you will find a table that includes the registered name of the interactor, the default key or button used for the interaction, and the action that is performed when using the interactor.

In this section

Local zoom interactor
Describes the local zoom interactor.

Edit-point interactor
Describes the edit-point interactor.

Highlight-point interactor
Describes the highlight-point interactor.

Information-view interactor
Describes the information-view interactor.

Pick-data-points interactor
Describes the pick-data-points interactor.

Treemap focus interactor
Describes the treemap focus interactor.

Introduction to the chart interactors

Each chart interactor (subclasses of `IlvChartInteractor`) implements a given type of interactive operation: scrolling, zooming, editing, or highlighting data points.

Thanks to this clean separation, chart interactors are lightweight and well-defined event-handling entities that can be easily customized.

The base class used to define the behavior of a chart in response to a given action by the user is the `IlvChartInteractor` class.

The JViews Charts package provides a comprehensive set of predefined interactors.

Some of these interactor classes inherit directly from the `IlvChartInteractor` class:

- ♦ `IlvChartZoomInteractor` allows the user to zoom in and zoom out on the data display area.
- ♦ `IlvChartXScrollInteractor` allows the user to scroll along the *x*-axis the displayed data by using the arrow keys.
- ♦ `IlvChartYScrollInteractor` allows the user to scroll along the *y*-axis the displayed data by using the arrow keys.
- ♦ `IlvChartPanInteractor` allows the user to scroll the displayed data by using the mouse.
- ♦ `IlvChartActionInteractor` allows the user to execute an `IlvChartAction` on a specified keyboard event.
- ♦ `IlvChartLocalPanInteractor` allows the user to scroll the zoomed data window of an `IlvLocalZoomAxisTransformer`.
- ♦ `IlvChartLocalReshapeInteractor` allows the user to reshape the zoomed data window of an `IlvLocalZoomAxisTransformer`.
- ♦ `IlvChartLocalZoomInteractor` allows the user to increase or decrease the zoom factor of an `IlvLocalZoomAxisTransformer`.

Other subclasses inherit from the `IlvChartDataInteractor` class, a subclass of `IlvChartInteractor` that specifically deals with interactions on the data points of the chart:

- ♦ `IlvChartEditPointInteractor` allows the user to edit a data point.
- ♦ `IlvChartHighlightInteractor` triggers an interaction event whenever the mouse moves over a data point in the data display area.
- ♦ `IlvChartInfoViewInteractor` inherits from the `IlvChartHighlightInteractor` class and displays information about a data point whenever the mouse moves over the data point in the data display area.
- ♦ `IlvChartPickInteractor` triggers an interaction event whenever a data point has been picked in the data display area.
- ♦ `IlvTreemapChartFocusInteractor` triggers an interaction event whenever a data point has been picked in the data display area.

Note: The precision that is currently used to find the data point corresponding to a given screen point is computed by an `IlvChartDataPicker` object. By default, all `IlvChartDataInteractor` subclasses use an `IlvDefaultChartDataPicker` instance that uses a Euclidian distance between two points. The way the distance is computed can be changed by overriding the `computeDistance(double, double, double, double)` method. To use your own `IlvChartDataPicker` class in your interactors, override the `createDataPicker(java.awt.event.MouseEvent)` method so that it returns an instance of your own class.

All default interactors have an associated shortcut name that allows the instantiation of the interactor class by its name. This mechanism is used by the JavaBeans to instantiate the interactor chosen by the user in the property editor of the interactors.

The following interactors are defined in the JViews Charts library:

- *Zoom interactor*
- *X-scroll interactor*
- *Y-scroll interactor*
- *Pan interactor*
- *Action interactor*
- *Local pan interactor*
- *Local reshape interactor*
- *Local zoom interactor*
- *Edit-point interactor*
- *Highlight-point interactor*
- *Information-view interactor*
- *Pick-data-points interactor*
- *Treemap focus interactor*

Zoom interactor

A zoom interactor has the following basic characteristics:

Class	`IlvChartZoomInteractor`
Registered name	“Zoom”
Default Key or Button	Left mouse button to zoom in, Shift + Left mouse button to zoom out.
Action	Lets the user trigger a zoom-in or a zoom-out command by dragging a box within the data display area of a chart. This box indicates the area to be zoomed in or zoomed out.

The zoom-in interaction is started by pressing the left mouse button by default. However, this button can be changed to any other button or key-button combination by passing the corresponding event mask as a parameter to the constructor of the zoom interactor, or by means of the `setZoomInEventMask(int)` method.

The zoom-out interaction is started by pressing the Shift Key + left mouse button by default. However, this combination can be changed to any other button or key-button combinations by passing the corresponding event mask as a parameter to the constructor of the zoom interactor, or by means of the `setZoomOutEventMask(int)` method.

The zoom-out operation is performed by starting the interaction and by dragging a box within a chart area. This box indicates the projection area of the current visible area, so that what you currently see is projected within this rectangle.

Note: You can control the zoom-out level with the size of the rectangle, as for the zoom in. For example, drawing a very small rectangle produces a big zoom out. If you have dragged a rectangle to zoom in, dragging the same rectangle in zoom-out mode returns to the previous visible data window.

Each zoom-in/zoom-out operation can be broken down into several steps to render a smooth transition between the original and the final visual state of the displayed data. When a zoom interactor instance is created, the default number of steps is set to 10. You can specify the number of steps by means of the `setAnimationStep(int)` method.

The zoom action can be performed either on a specific or both axes of a coordinate system. When an interactor instance is created, zooming along the *y*-axis is disabled by default. You change this behavior by means of the `setYZoomAllowed(boolean)` and `setXZoomAllowed (boolean)` methods.

The cursors used during zoom-in and zoom-out operations can be changed by overriding the `getZoomInCursor()` and `getZoomOutCursor()` methods to return other cursors.

X-scroll interactor

An x-scroll interactor has the following basic characteristics:

Class	`IlvChartXScrollInteractor` Inherits from `IlvChartScrollInteractor`
Registered name	“XScroll”
Default Key or Button	Left arrow key to scroll in the negative direction, right arrow key to scroll in the positive direction.
Action	Lets the user scroll through the displayed data along the x-axis.

The default keys used to scroll are the left arrow key for negative direction and right arrow key for positive direction. However, these keys can be changed by passing other key codes as parameters to the constructor of the *x*-scroll interactor, or by means of the `setPositiveDirectionKey(int)` and `setNegativeDirectionKey(int)` methods.

Y-scroll interactor

An y-scroll interactor has the following basic characteristics:

Class	`IlvChartYScrollInteractor` Inherits from `IlvChartScrollInteractor`
Registered name	“YScroll”
Default Key or Button	Up arrow key to scroll in the positive direction, down arrow key to scroll in the negative direction.
Action	Lets the user scroll through the displayed data along the y-axis.

The default keys used to scroll are the up arrow key for positive direction and bottom arrow key for negative direction. However, these keys can be changed by passing other key codes as parameters to the constructor of the y-scroll interactor, or by means of the `IlvChartScrollInteractor.setPositiveDirectionKey` and `IlvChartScrollInteractor.setNegativeDirectionKey` methods.

Pan interactor

A pan interactor has the following basic characteristics:

Class	`IlvChartPanInteractor`
Registered name	“Pan”
Default Key or Button	Right mouse button.
Action	Lets the user scroll through the displayed data by dragging the mouse in any direction.

The mouse button used to scroll through the displayed data is the right mouse button by default. However, this button can be changed to any other button or key-button combination by passing the corresponding event mask as a parameter to the constructor of the pan interactor or by means of the `setEventMask(int)` method.

The pan action can be performed either on a specific or both axes of a coordinate system. When an interactor instance is created, the panning along the *y*-axis is disabled by default. You can change this behavior by means of the `setYPanAllowed(boolean)` and `setXPanAllowed (boolean)` methods.

By default, the cursor used when the mouse is dragged is the predefined `Cursor. MOVE_CURSOR`. However, to return another cursor you can override the `getCursor()` method.

Action interactor

An action interactor has the following basic characteristics:

Class	`IlvChartActionInteractor`
Registered name	“ChartAction”
Default Key or Button	‘Z’ key to zoom in, ‘U’ key to zoom out, ‘F’ key to fit.
Action	Lets the user execute an `IlvChartAction` when a specific key is pressed.

The action to trigger when a key has been pressed is determined by means of the `getAction (java.awt.event.KeyEvent)` method. You can change the default association by overriding this method to return your own actions.

Local pan interactor

A local pan interactor has the following basic characteristics:

Class	`IlvChartLocalPanInteractor` Inherits from `IlvChartPanInteractor`
Registered name	“LocalPan”
Default Key or Button	Right mouse button.
Action	Lets the user scroll the zoomed data window of an `IlvLocalZoomAxisTransformer` by dragging the mouse in any direction.The interaction starts when the user clicks within the zoomed data window.

The mouse button used to scroll through the displayed data is the right mouse button by default. However, this button can be changed to any other button or key-button combination by passing the corresponding event mask as a parameter to the constructor of the pan interactor, or by means of the `setEventMask(int)` method.

The pan action can be performed either on a specific or both axes of a coordinate system. When an interactor instance is created, the panning along the *y*-axis is disabled by default. You can change this behavior by means of the `setYPanAllowed(boolean)` and `setXPanAllowed (boolean)` methods.

By default, the cursor used when the mouse is dragged is the predefined `Cursor.MOVE_CURSOR`. However, to return another cursor you can override the `getCursor()` method.

Local reshape interactor

A local reshape interactor has the following characteristics:

Class	`IlvChartLocalReshapeInteractor`
Registered name	“LocalReshape”
Default Key or Button	Left mouse button.
Action	Lets the user reshape the zoomed data window of an `IlvLocalZoomAxisTransformer` by dragging the zoomed area bounds when the mouse moves over one of the zoomed data window bounds.

The mouse button used to reshape the data window is the left mouse button by default. However, this button can be changed to any other button or key-button combination, by passing the corresponding event mask as a parameter to the constructor of the pan interactor or by means of the `setEventMask(int)` method.

By default, the cursor used when the mouse moves over a zoomed data window bound is the predefined `Cursor.HAND_CURSOR`. However, to return another cursor you can override the `getCursor` method.

Local zoom interactor

A local zoom interactor has the following characteristics:

Class	`IlvChartLocalZoomInteractor` Inherits from `IlvChartZoomInteractor`
Registered name	“LocalZoom”
Default Key or Button	Left mouse button.
Action	Lets the user change the zoom factor of an `IlvLocalZoomAxisTransformer` by dragging a box within the zoomed data window of the transformer. This box indicates the area to be zoomed in or zoomed out.

The zoom-in interaction is started by pressing the left mouse button by default. However, this button can be to changed to any other button or key-button combination, by passing the corresponding event mask as a parameter to the constructor of the local zoom interactor or by means of the `setZoomInEventMask(int)` method.

The zoom-out interaction is started by pressing the Shift Key + left mouse button by default. However, this combination can be changed to any other button or key-button combinations by passing the corresponding event mask as a parameter to the constructor of the zoom interactor or by means of the `setZoomOutEventMask(int)` method.

Each zoom-in/zoom-out operation can be broken down into several steps to render a smooth transition between the original and the final visual state of the displayed data. When a zoom interactor instance is created, the default number of steps is set to 10. You can specify the number of steps by means of the `setAnimationStep(int)` method.

The cursors used during zoom-in and zoom-out operations can be changed by overriding the `getZoomInCursor()` and `getZoomOutCursor()` methods to return another cursor.

Edit-point interactor

An edit-point interactor allows the user to edit a data point. It has the following basic characteristics:

Class	`IlvChartEditPointInteractor` Inherits from `IlvChartDataInteractor`
Registered name	“EditPoint”
Default Key or Button	Left mouse button.
Action	Lets the user modify a data point by dragging its graphical representation within the data display area.

The mouse button used to perform the drag operation is the left mouse button by default. However, this button can be changed by passing another button as a parameter to the constructor of the edit point interactor, or by means of the `setEventMask(int)` method.

You can use two modes for the drag operation:

♦ With the opaque mode, the data point is modified each time the mouse is dragged.

♦ With the ghost mode, the data point is modified only when the mouse button is released.

When a drag-point interactor is created, the default mode that is used for the drag operation is the ghost mode. You can specify that the opaque mode should be used by calling the `setOpaqueEdit(boolean)` method with `true` as parameter.

The data point modification can be performed either on a specific or both axes of a coordinate system. When an interactor instance is created, editing along the *y*-axis is disabled by default. You can change this behavior by means of the `setYEditAllowed(boolean)` and `setXEditAllowed(boolean)` methods with `true` as parameter.

During the interaction, the value of the data point that is currently edited can be constrained within some specified rules by means of the `validate(ilog.views.chart.IlvDoublePoints, ilog.views.chart.IlvDisplayPoint)` method. This method is called each time the mouse is dragged to validate the new data point. For example, the following code shows you how to constrain a data point to have rounded *y*-values.

```
protected void validate(IlvDoublePoints pt,IlvDisplayPoint dpt)
{
  pt.setY(0, Math.round(pt.getY(0)));
}
```

An example of `IlvChartEditPointInteractor.validate` override can be found in **`<installdir>/jviews-charts86/samples/radar/src/radar/RadarDemo.java`**. In this example, the **`EditPointInteractor`** class extends `IlvChartEditPointInteractor` to provide the following features:

♦ Display the value of the point being edited next to the mouse cursor.

- Ensures that the edited points take values that are a multiple of a given precision, by overriding the `validate` method.

Highlight-point interactor

A highlight-point interactor triggers an interaction event whenever the mouse moves over a data point in the data display area. It has the following basic characteristics:

Class	`IlvChartHighlightInteractor` Inherits from `IlvChartDataInteractor`
Registered name	“Highlight”
Default Key or Button	None.
Action	Triggers an interaction event whenever the mouse moves over a data point in the data display area.

To be notified whenever a data point has been highlighted, you have to add an interaction listener on the highlight interactor by means of the `addChartInteractionListener(ilog.views.chart.event.ChartInteractionListener)` method. Whenever the mouse moves over a data point, the `interactionPerformed(ilog.views.chart.event.ChartInteractionEvent)` method of the listener is called with a `ChartHighlightInteractionEvent` as parameter.

The `interactionPerformed(ilog.views.chart.event.ChartInteractionEvent)` method of the listener receives an event, whose `getDisplayPoint()` method returns the following:

- for all series-based chart types: the point with which it interacts, as an `IlvDisplayPoint` instance,
- for a treemap: the rectangle with which it interacts, as an `IlvDisplayObjectArea` instance.

Information-view interactor

An information-view interactor displays information about a data point whenever the user moves the mouse over the data point. It has the following basic characteristics:

Class	`IlvChartInfoViewInteractor` Inherits from `IlvChartHighlightInteractor`
Registered name	“InfoView”
Default Key or Button	None.
Action	Displays information about a data point whenever the user moves the mouse over the data point in the data display area.

The information is displayed in a `JToolTip` instance by default. However, you can change the type of the tooltip by overriding the `createToolTip()` method to return an instance of your own `JToolTip` subclass.

The text that is displayed by default is the name of the data set to which the data point belongs and the abscissa and ordinate values of the data point. This text can be redefined in a subclass by overriding the `getInfoText(ilog.views.chart.IlvDisplayPoint)` method.

By default, the text is composed of a description part and a value part. These parts can be customized individually by overriding the `getInfoTextDescriptionPart(ilog.views.chart.IlvDisplayPoint)` and `getInfoTextValuePart(ilog.views.chart.IlvDisplayPoint)` methods. The argument of these methods is an `IlvDisplayPoint`. When used within a treemap, the actual argument will be of type `IlvDisplayObjectArea`.

Pick-data-points interactor

A pick-data-points interactor triggers an event when the user selects data points in the data display area. It has the following basic characteristics:

Class	`IlvChartPickInterator` Inherits from `IlvChartDataInteractor`
Registered name	“Pick”
Default Key or Button	Left mouse button.
Action	Triggers an event when the user selects data by clicking a projected point in the data display area.

The mouse button used to perform the selection is the left mouse button by default. However, this button can be changed to any other button or key-button combination by passing the corresponding event mask as a parameter to the constructor of the selection interactor, or by means of the `setEventMask(int)` method.

The mouse event that fires the interaction event is `MOUSE_RELEASED` by default. You can change this event by overriding the `isPickingEvent(java.awt.event.MouseEvent)` method to return `true` when the interaction event should be fired.

To be notified whenever a data point, series or object has been picked, you have to add an interaction listener on the interactor by means of the `addChartInteractionListener` method. Whenever the user clicks on or near a data point or data rectangle, the `interactionPerformed` method of the listener is called with a `ChartInteractionEvent` as parameter. The `getDisplayPoint()` method returns the following:

♦ for all series-based chart types: the point that has been clicked, as an `IlvDisplayPoint` instance,

♦ for a treemap: the rectangle that has been clicked, as an `IlvDisplayObjectArea` instance.

Example:

```
// Create the interactor.
IlvChartInteractor interactor = new IlvChartPickInteractor();
// Determine what to do when the user performs an action with the
// interactor.
interactor.addChartInteractionListener(
  new ChartInteractionListener() {
   public void interactionPerformed(ChartInteractionEvent event) {
     IlvDisplayPoint point = event.getDisplayPoint();
     // In the case of a treemap:
     // IlvDisplayObjectArea area = (IlvDisplayObjectArea)point;
     ...
   }
});
```

```
// Activate the interactor.
chart.addInteractor(interactor);
```

Treemap focus interactor

A treemap chart has the notion of focus. At any time, the treemap displays a subtree of the entire tree of model objects. The root of that branch is called the current *focus* of the treemap. The focus object is displayed as a rectangle that covers almost the entire surface of the chart area.

The treemap focus interactor allows the user to change the focus. It provides a sort of drill-down. It has the following basic characteristics:

Class	`IlvTreemapChartFocusInteractor` inherits from `IlvChartDataInteractor`
Registered name	“Focus”
Default Key or Button	Left mouse button double-click.
Action	Changes the treemap focus to the object selected by the user.

By default, to change the focus you need to double-click with the left mouse button. Different mouse button actions can be obtained by choosing the appropriate constructor of `IlvTreemapChartFocusInteractor`.

When the focused subtree is not the entire tree, the user has the possibility to come back to the next enclosing focus, the parent object in the tree model, by clicking a border of the chart area that indicates this parent object.

This interactor does not keep a history of the selected focus. You can implement such a history and the appropriate Back and Forward actions by installing a `TreemapFocusListener` on the chart.

Setting an Interactor on an IlvChart

Several interactors can be used at the same time to interact with a given chart object. These interactors are managed by the `IlvChart` object on which the interactions are performed and cannot be shared among several charts.

All the predefined `IlvChartInteractor` objects are stored in an internal repository that performs associations between a string and an interactor class. When an interactor class is registered in the repository, it can be instantiated using its associated string (hereafter named “shortname”). All the interactors provided in the JViews Charts package have an associated shortname.

Two methods are available to add an interactor on an `IlvChart`:

- `void addInteractor(IlvChartInteractor)`
- `void addInteractor(String)`

The first method is used to add an interactor when an instance has explicitly been created and the second method to add an interactor using its shortname.

Note that there is priority among the interactors in the event dispatching process due to the way interactors are managed by an `IlvChart` (see *Handling interactions* for more details): the last added interactor is the last interactor to receive events.

For example, assume that you want to add a zoom interactor and an edit-point interactor. Both the interactors use the left mouse button to perform their interaction. If the zoom interactor is added before the edit-point interactor, all the left mouse button events will be sent first to the zoom interactor. Since the zoom interactor handles this event to start its interaction, the edit-point interactor will never receive a left button pressed event. To be able to use both the interactors, the correct order is to add the edit-point interactor first, and then the zoom interactor. Indeed, the edit-point interactor only handles mouse pressed event if the user clicked on a data point. If it is not the case, the event is not handled (that is, not consumed), and the chart submits it to the next interactor in the list, the zoom interactor.

The basic steps to set an interactor on a given chart object are the following:

- Create an interactor instance.
- Add the interactor to be used to the chart.

Handling interactions

Interactors are handled by a chart as an ordered list, the last element of the list is the last interactor that has been added. This order allows the user to define priorities among interactors during the event dispatching process: each time an event occurs, it is sent to all the interactors until one consumes it, beginning with the first interactor in the list.

The event dispatching process

When an event occurs on the chart area, the event is sent to all the interactors handled by the chart, beginning with the first interactor in the list. Depending on its type, the event is dispatched to the interactor by calling the `processMouseEvent(java.awt.event.MouseEvent)`, `processMouseMotionEvent(java.awt.event.MouseEvent)`, and `processKeyEvent(java.awt.event.KeyEvent)` methods. If the first interactor does not consume the event, then the event is sent to the second interactor in the list, and so on, until either one of the interactors consumes the event or the end of the list is reached. To mark an event as consumed, the interactor should call the `java.awt.AWTEvent.consume` method on the input event.

Handling events at interactor level

An `IlvChartInteractor` object can handle two types of input events:

- ♦ the mouse events,
- ♦ the key events.

To be able to handle these events, the interactor has to notify the chart that events of a given type should be caught and sent to it. This is done by means of the `enableEvents(long)` method. This method enables the specified events on the chart area, if not already done.

Filtering events

An interactor has the possibility to filter the events it receives according to their position (for example an interactor may be interested in events occurring only on the plot area). This event filtering is the purpose of the `isHandling(int, int)` method, which returns a Boolean value indicating whether the event should be processed by the interactor or not. When a chart is about to send an event to an interactor, it first invokes the interactor `isHandling` method, and depending on the returned value, propagates the event to the interactor.

Processing events

Once enabled, the events are handled by specialized `IlvChartInteractor` methods depending on their types:

- ♦ `processMouseMotionEvent(java.awt.event.MouseEvent)` for mouse motion events.
- ♦ `processMouseEvent(java.awt.event.MouseEvent)` for mouse events.
- ♦ `processKeyEvent(java.awt.event.KeyEvent)` for key events.

Note: By default, the implementation of these methods is empty. You should override them to add your own event handling code.

During the interaction, the `IlvChartInteractor` API defines the following three states:

- ♦ the interaction start,
- ♦ the interaction stop,
- ♦ the interaction abortion.

These three steps are defined at the programming level for each interactor class by calling the corresponding `startOperation(java.awt.event.MouseEvent)`, `endOperation(java.awt.event.MouseEvent)`, and `abort()` methods when appropriate.

Writing your own interactor

When writing your own interactor, you can use and extend the source code of the built-in JViews Charts interactors. You find this source code in the **`<installdir>/jviews-charts86/samples/interactor/src/`** directory.

This example is extracted from the stock sample. The source code of this example can be found in **`<installdir>/jviews-charts86/samples/stock/src/stock/ZoomScaleInteractor.java`** directory.

The `ZoomScaleInteractor` class defined in this example allows the user to zoom a given portion of the display area by selecting the area to zoom in on the scale itself instead of on the plot area.

1. Extend the `IlvChartInteractor` class.

```
public class ZoomScaleInteractor extends IlvChartInteractor
{
 ...
}
```

2. Define the attributes of the class.

```
private boolean swap = false;
protected IlvStyle style;
protected int axisIdx;
protected double start;
protected double end;
```

The `axisIdx` attribute specifies the axis index on which the zoom is performed (-1 = *x*-axis). The bounds of the zoomed area are defined by the `start` and `end` attributes. This data is updated each time the mouse is dragged.

3. Add the constructor.

```
/**
* Create new ZoomScaleInteractor associated with the specified axis
* that zoom in on a BUTTON1 event and zoom out on SHIFT+BUTTON1.
*/
public ZoomScaleInteractor(int axisIdx)
{
   super(axisIdx != -1 ? axisIdx : 0, MouseEvent.BUTTON1_MASK);
   this.axisIdx = axisIdx;
   renderer = new IlvStyle(new BasicStroke(5), IlvColor.magenta);
   ...
```

An interactor is always attached to a unique *y*-axis. Indeed, since several *y*-axes may coexist on the same chart, an interactor needs to know on which *y*-axis the interactions are performed, so that conversions from display space to data space (and vice-versa) are possible. This *y*-axis is referenced at the `IlvChartInteractor` level using its index and is the first parameter of the constructor.

We want to handle both mouse motion events and mouse click events, as well as key events (to handle cancellation using ‘ESC’ key):

```
    ...
    enableEvents(AWTEvent.MOUSE_EVENT_MASK |
                 AWTEvent.MOUSE_MOTION_EVENT_MASK |
                 AWTEvent.KEY_EVENT_MASK);
}
```

4. Filter the events.

Since the interaction is performed on the scale, the interactor is only interested in the events that occur within the scale bounds:

```
public boolean isHandling(int x, int y)
{
   return (getScale() != null) ?
     getScale().getBounds(scaleBounds).contains(x, y) : false;
}
```

5. Process the events.

In the `processMouseEvent(java.awt.event.MouseEvent)` method, we define that the interaction starts when the left mouse button has been pressed, calling the `startOperation(java.awt.event.MouseEvent)` method.

The event coordinate is converted to a data value on the scale, and the `start` and `end` attributes are initialized to the previous and next steps of the scale, respectively.

```
public void processMouseEvent(MouseEvent evt)
{
   double value;
   switch (evt.getID()) {
   case MouseEvent.MOUSE_PRESSED :
       ...
         startOperation(evt);
          value = getScale().toValue(evt.getX(), evt.getY());
             start = getScale().getStepsDefinition().previousStep(value)
;
             end = getScale().getStepsDefinition().incrementStep(start);

             drawGhost();
             evt.consume();
          ...
         break;
         ...
}
```

The `drawGhost()` method is then called. This method allows you to have a visual feedback of the interaction by temporarily drawing over the chart area.

Mouse dragged events are handled as follows:

```
public void processMouseMotionEvent(MouseEvent evt)
{
  if (evt.getID() == MouseEvent.MOUSE_DRAGGED && isInOperation()){
         drawGhost();
    double value = getScale().toValue(evt.getX(), evt.getY());
         computeStartEnd(value);
         drawGhost();
         evt.consume();
  }
}
```

First erase the previous ghost (note that it has effects only in XOR mode), then compute the new start and end values, and finally draw the new ghost.

The zoom is effectively performed when the button is released.

```
public void processMouseEvent(MouseEvent evt)
{
     ...
     case MouseEvent.MOUSE_RELEASED:
            if (!isInOperation())
                break;
            value = getScale().toValue(evt.getX(), evt.getY());
            computeStartEnd(value);
            drawGhost();
            zoomScale();
            endOperation(evt);
            evt.consume();
            break;
    ...
}
```

The new start and end values are computed, the scale is zoomed, and the interaction is set as ended. Finally, we want to cancel the interaction that is performed when the 'ESC' key is pressed:

```
public void processKeyEvent(KeyEvent evt)
{
    if (evt.getID() == KeyEvent.KEY_PRESSED &&
        evt.getKeyCode() == KeyEvent.VK_ESCAPE ) {
        if (isInOperation())
            drawGhost();
        abort();
        evt.consume();
    }
}
```

If the interactor is in operation, the ghost is erased (it has only effects in XOR mode), and the interaction state is set as aborted, with the `abort()` method implemented as follows:

```
protected void abort()
{
    super.abort();
    swap = false;
    /* Disable ghost drawing operation */
    setAllowDrawGhost(false);
}
```

6. Draw a ghost.

   A ghost can be drawn either in XOR mode or in Paint mode, depending on the value of the interactor `xorGhost` property. By default, most of the default interactors draw their ghost in paint mode. In a general manner, XOR mode should be avoided due to the poor control it gives on the drawing.

   To be able to draw a ghost, an interactor must first enable the draw ghost mechanism, calling the `setAllowDrawGhost(boolean)` method with `true` as parameter, and disable it when the interaction ends. This is performed by the `startOperation(java.awt.event.MouseEvent)` and `endOperation(java.awt.event.MouseEvent)` methods:

```
protected void endOperation(MouseEvent evt)
{
    super.endOperation(evt);
    swap = false;
    /* Disable ghost drawing operation */
    setAllowDrawGhost(false);
}

protected void startOperation(MouseEvent evt)
{
    super.startOperation(evt);
    /* Enable ghost drawing operation */
    setAllowDrawGhost(true);
}
```

Note: When overriding one of the `startOperation`, `endOperation` or `abort` methods, do not forget to call the corresponding `super` method since it performs several required internal initializations.

Configuring 3-D Rendering

Describes how to switch from a two-dimensional to a three-dimensional display and how to control the properties of 3-D rendering.

In this section

Switching to 3-D
Describes how to switch between a 2-D and a 3-D representation.

Supported features
Describes the features currently supported by a chart using 3-D rendering.

Switching to 3-D

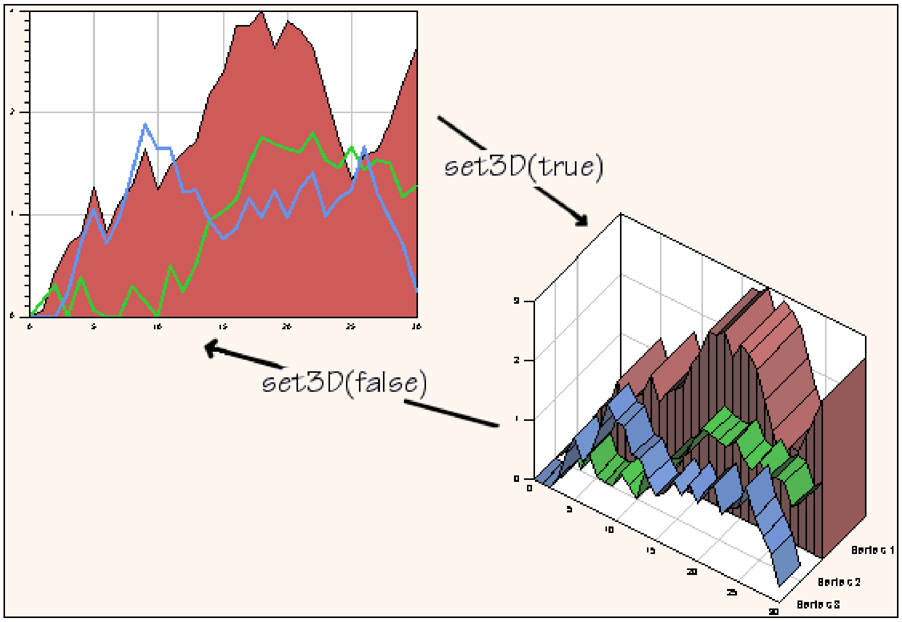
Describes how to switch between a 2-D and a 3-D representation.

In this section

3-D view methods

With the JViews Charts library, you can switch between a 2-D and a 3-D representation by simply calling the `set3D(boolean)` method. This method does not alter the structure of the chart. The chart components (header, footer, legend) and the chart elements (renderers, scales, grids, decorations, and so on) remain unchanged.

You can dynamically call this method on an existing chart, as shown in *Switching Between 2-D and 3-D*.



Switching Between 2-D and 3-D

The `is3D()` method indicates whether the chart is displayed in 3-D.

Note: Only Cartesian and Pie charts support 3-D rendering. Calling `IlvChart.set3D` on a chart that does not support this mode (for example, a Radar chart) does not raise an error, but the visual appearance of the chart will not change. Likewise, if you switch between a 3-D Cartesian chart and a Polar chart with the `setType(int)` method, the polar representation will be made in 2-D. You can find a list of the available 3-D capabilities in the section *Supported features*.

The 3D rendering features are shown in the sample located in **`<installdir>/jviews-charts86/samples/chart3d/index.html`**.

3-D view properties

The `IlvChart3DView` class represents the three-dimensional view of the chart and lets you control its visual appearance. You can retrieve the view associated with a chart by calling the `IlvChart.get3DView` method.

Note: The `get3DView()` method returns a valid `IlvChart3DView` object, even if the chart is not currently displayed in 3-D. You can thus switch between a 3-D and a 2-D representation, and still keep the properties of the 3-D display.

Projection properties

The following properties specify how points are projected in the chart view.

Property	Methods	Default Value
Elevation	`getElevation` `setElevation`	`45`
Rotation	`getRotation` `setRotation`	`35`
Automatic Scaling	`isAutoScaling` `setAutoScaling`	`true`
Zooming Factor	`getZoom` `setZoom`	`1`
Projection Type	`getProjectionType` `setProjectionType`	`IlvChart3DView.ORTHOGRAPHIC`

View angles

The elevation and rotation angles specify the location of the eye. The elevation ranges from –90 degrees to 90 degrees. The rotation angle ranges from –90 degrees to 90 degrees for Cartesian charts and can take any value for Pie charts. A dedicated interactor lets you modify these angles, as described in the section *Interactive control of the 3-D view orientation*.

Scaling factor

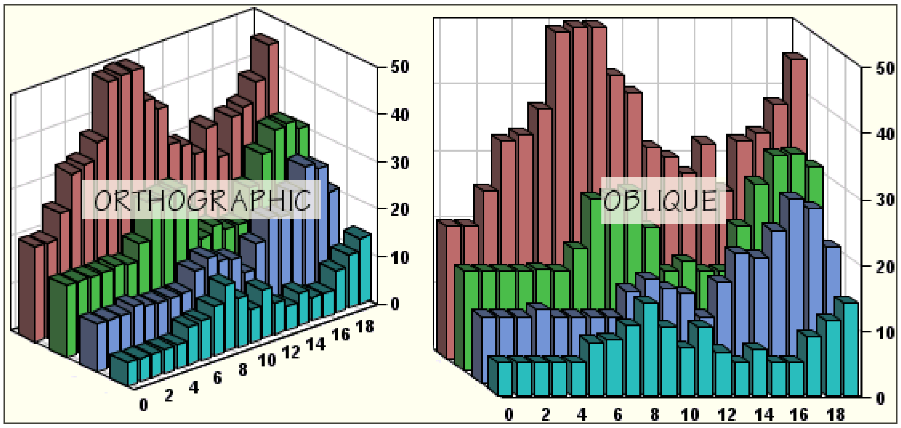
The scaling applied to the x- and y-coordinates during projection is computed as follows: if the `autoScaling` property is set to `true`, the chart tries to determine the appropriate scaling factor so that the drawing fits the rectangle of the chart area. The computed scaling factor is then multiplied by the zooming factor.

Projection type

Two projection types are available for Cartesian charts:

- Orthographic projection (`IlvChart3DView.ORTHOGRAPHIC`).
- Oblique projection (`IlvChart3DView.OBLIQUE`). This projection preserves the orthogonality of x- and y-axes.

Orthographic and Oblique Projections shows the difference between the two projection types.



Orthographic and Oblique Projections

Lighting properties

The following properties specify how lighting is shown in the chart view.

Property	Methods	Default Value
Ambient light intensity	`getAmbientLight` `setAmbientLight`	`0.1`
Light latitude	`getLightLatitude` `setLightLatitude`	`0`
Light longitude	`getLightLongitude` `setLightLongitude`	`0`

The JViews Charts library uses a simple lighting model composed of:

- A direct light, which casts parallel rays. This light is located by spherical coordinates in the projected space. The default latitude and longitude are equal to 0, which means that the light is originally located at the eye.

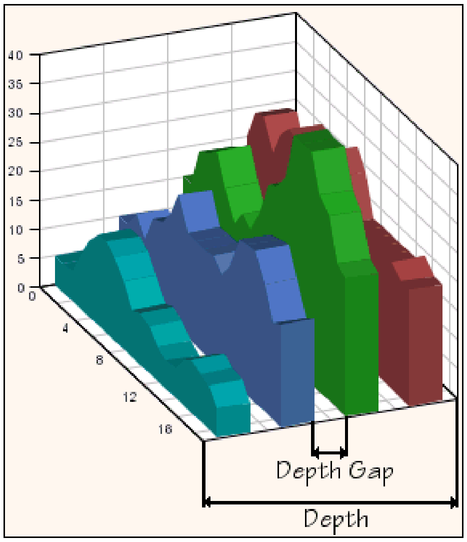
♦ An ambient light, which illuminates all the objects of a 3-D view, independently of the orientation of the rendered faces. Changing the intensity of this light prevents some parts of the chart from being too dark. The intensity ranges from 0 (no ambient light) to 1 (maximum intensity).

Controlling depth

The following properties control depth in the chapter view.

Property	Methods	Default Value
Depth	`getDepth` `setDepth`	`20`
Depth gap	`getDepthGap` `setDepthGap`	`0`

The depth property ranges from 1 to 100 and specifies the percentage of the chart depth relative to its width. The depth gap lets you control the separation between two layers, as shown in *Controlling the Depth of a Chart*.



Controlling the Depth of a Chart

Decorations along the depth axis

The chart can display annotations and gridlines along the z-axis (also referred to as the *depth axis*).

The annotations are returned by the `getZAnnotationText()` method. You can control the visibility and the appearance of annotations and grid with the following properties:

Property	Methods	Default Value
z-annotation visibility	`isZAnnotationVisible` `setZAnnotationVisible`	`true`
z-annotation renderer	`getZAnnotationRenderer` `setZAnnotationRenderer`	
z-grid visibility	`isZGridVisible` `setZGridVisible`	`true`
z-grid paint	`getZGridPaint` `setZGridPaint`	Paint of x- or y-grid
z-grid stroke	`getZGridStroke` `setZGridStroke`	Stroke of x- or y-grid

Listening to property changes

The `IlvChart3DView` class enables you to add listeners that are notified whenever a property is modified. The registration and notification use the `PropertyChangeListener` and `PropertyChangeEvent` classes from the `java.beans` package. The **ControlPanel3D.java** source file shows how listeners can be used to synchronize slider controls with the values of numeric properties.

Interactive control of the 3-D view orientation

The `IlvChart3DViewInteractor` class allows the user to interactively control the rotation and elevation angles of the view, as well as the zooming factor. This interactor can be connected to a chart as any regular chart interactor, using the `addInteractor(ilog.views.chart.IlvChartInteractor)` method:

```
chart.addInteractor(new IlvChart3DViewInteractor())
```

The `IlvChart3DViewInteractor` class provides the following interactions:

♦ You can change the rotation angle (horizontal move) or the elevation angle (vertical move) by holding the CTRL key and dragging with the left mouse button.

♦ You can change the zoom factor (zoom-out for an upward move, and zoom-in for a downward move) by holding the SHIFT key and dragging with the left mouse button.

The event masks can be specified in the constructor of the interactor, or changed with the `setAngleEventMask(int)` and `setZoomEventMask(int)` methods.

Supported features

Describes the features currently supported by a chart using 3-D rendering.

In this section

Available chart renderers

Several objects in the JViews Charts library implement the `IlvChart3DSupport` interface and provide the `has3DSupport()` method to query whether 3-D is handled:

- ♦ Drawable objects (scales, grids, and decorations)
- ♦ Renderers
- ♦ Interactors

Note: The supported features are only a subset of what is available for 2-D charts.

Once the chart is set to 3-D, all graphical elements that do not support 3-D are discarded during the rendering process. Likewise, events are not dispatched to the interactors that do not work with 3-D charts.

You can combine several Cartesian representations (bar, line, area and stair) within a single chart. More information on chart renderers can be found in *Handling Chart Renderers*.

Bar Charts

The following 3-D bar charts representations are supported:

- ♦ Superimposed bars.

  Each series is displayed in a separate layer along the depth axis. The `getZAnnotationText()` method returns the text that must be displayed next to each layer.

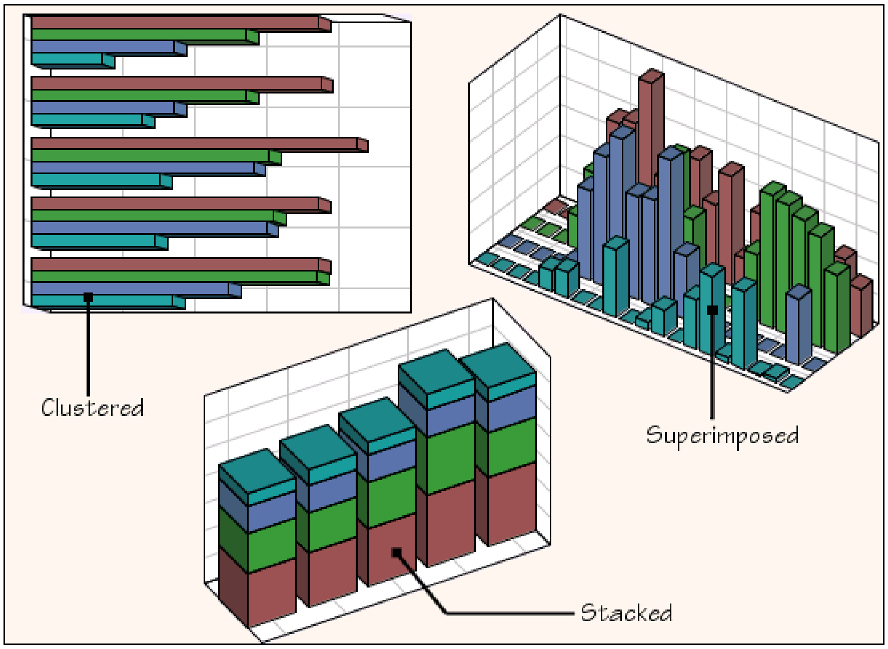
- ♦ Clustered bars.

  Bars are laid out in clusters on the same layer. No layer annotation is specified.

- ♦ Stacked bars.

  Bars are stacked on the same layer. No layer annotation is specified.

3-D Bar Charts shows some examples of 3-D bar charts.



3-D Bar Charts

Line, Area, and Stair Charts

The following continuous representations are supported:

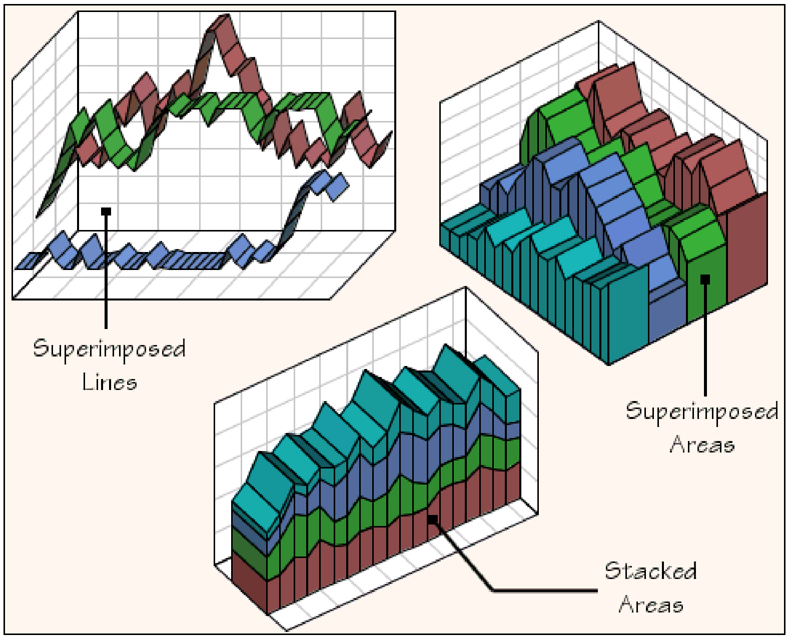
♦ Superimposed line, area, or stair charts.

  Each series is displayed in a separate layer along the depth axis. The `getZAnnotationText()` method returns the text that must be displayed next to each layer.

♦ Stacked line, area, or stair charts.

  The entire series is displayed on the same layer. No layer annotation is specified.

3-D Lines and Areas shows some examples of line and area charts.

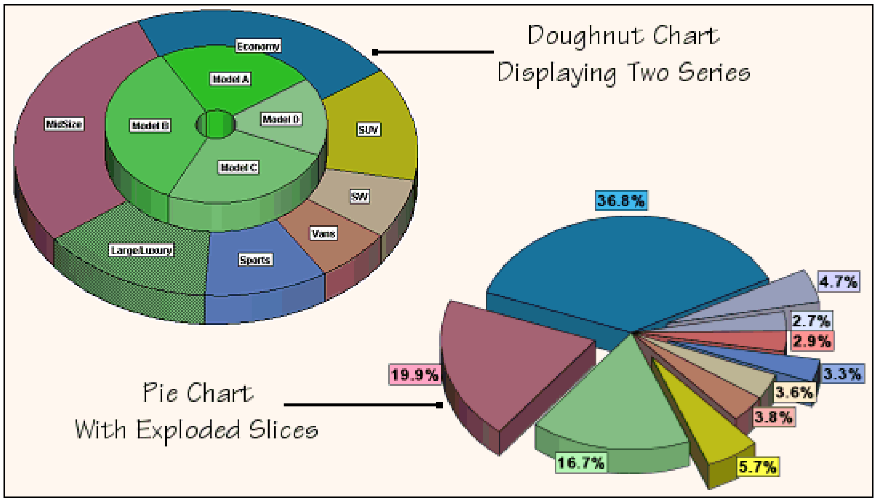


3-D Lines and Areas

For 3-D line charts, the `set3DOutlinePaint(java.awt.Paint)` method lets you specify the color of the ‘ribbon’ outline.

Pie Charts

The JViews Charts library supports 3-D pie and doughnut charts. When several series are provided, each one is displayed in a separate layer. As in 2-D, you can also explode slices. *3-D Pie Charts* shows some 3-D pie charts.



3-D Pie Charts

Available chart decorations

The JViews Charts library provides two predefined chart decorations:

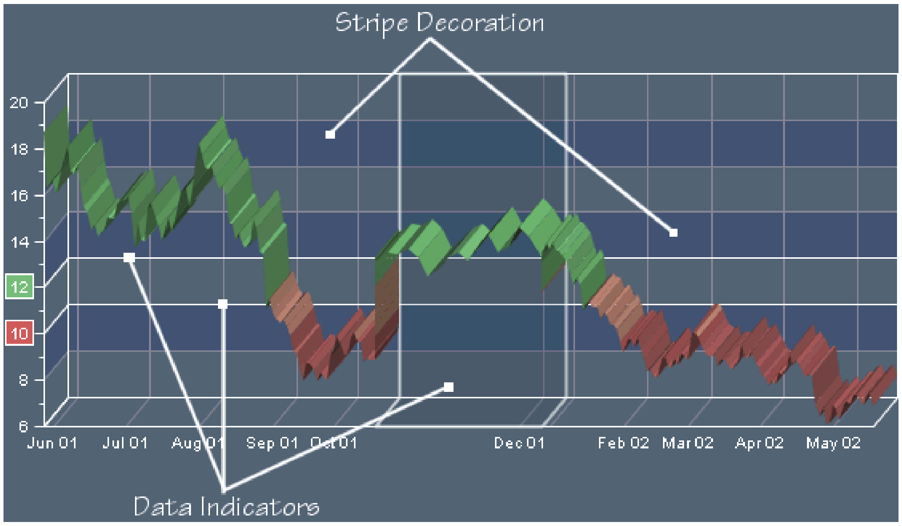
♦ Data indicators, which can also be used with a 3-D Cartesian chart.

♦ Image decorations, which are not supported in 3-D.

You can find more information on data indicators in *Displaying Data Indicator*.

You can also design decorations that can be displayed both in 2-D and in 3-D. For example, the stripe decoration implemented in **Stripes.java** uses the `getShape(ilog.views.chart.IlvDataWindow, java.awt.Rectangle, ilog.views.chart.IlvCoordinateSystem)` method. This method returns a different shape depending on whether the chart is displayed in 2-D or in 3-D.

Decorations in 3-D shows a chart using the stripe decoration and data indicators.



Decorations in 3-D

Available chart interactions

Besides the interactive control of the 3-D view, the following interactions can be used with a 3-D chart.

- Picking interactions (`IlvChartPickInteractor`). The picking mode must be set to `IlvChartData.ITEM_PICKING`..
- Highlighting interactions (`IlvChartHighlightInteractor`, `IlvChartInfoViewInteractor`). The picking mode must be set to `IlvChartData.ITEM_PICKING`..
- Scrolling interactions (`IlvChartXScrollInteractor`, `IlvChartYScrollInteractor`).

Note: Interactors that do not work with 3-D charts are simply discarded during the event dispatching process.

Unsupported operations

Trying to perform display to data projection will raise an `UnsupportedOperationException`. This concerns the following methods: `toData(ilog.views.chart.IlvDoublePoints)`,`toData(ilog.views.chart.IlvDoublePoints, java.awt.Rectangle, ilog.views.chart.IlvCoordinateSystem)`,`toDataWindow(java.awt.Rectangle, java.awt.Rectangle, ilog.views.chart.IlvCoordinateSystem)`.

Synchronization between the plotting areas of two charts is not supported (see the `synchronizeAxis(ilog.views.chart.IlvChart, int, boolean)` method).

The resizing policy specified by the `setResizingPolicy(ilog.views.chart.IlvChartResizingPolicy)` is ignored.

The Line, Area, and Stair chart renderers do not perform clipping along the y-axis. This means that you must make sure that the visual y-range contains all the displayed points. This is usually done by setting the `autoVisibleRange` property of the y-axis to `true`, which is the default value.

For the Area and Stair chart renderers, the crossing value of the x-axis is constrained to the minimum or the maximum value of the y-axis.

Using CSS Syntax in the Style Sheet

Describes CSS briefly and explains in more detail the version of CSS used inIBM® ILOG® JViews Charts and typical uses of CSS for customizing chart components, data series and points.

In this section

The origins of CSS

Cascading style sheets (CSS) are a powerful mechanism to customize HTML rendering inside a Web browser. The CSS2 specification comes from the W3C, and has now reached the status of a W3C recommendation.

The CSS syntax is a great improvement over the .Xdefault resource mechanism of the X Window System. The basic idea remains the same: matching a pattern and setting resource values. CSS is devoted to HTML rendering, matching HTML tags and setting style values. XML is another CSS target, especially as used within the SVG (Scalable Vector Graphics) recommendation from the W3C.

The CSS syntax

The style sheet syntax conforms to the CSS2 (Cascading Style Sheet level 2) specification with a few divergences.

The general template of a style rule in a style sheet is therefore:

```
selector {
  declaration1;
  declaration2;
...
}
```

For visualization purposes, the selector applies to objects in the data model, and is used for pattern-matching; the declarations apply to the corresponding graphic objects, and are used for rendering.

Declarations have the form:

```
   propertyName : value ;
```

An example of a style rule is:

```
series[name="Sales"]  {
    lineWidth: 2;
}
```

This rule sets the line width of series of name "Sales" to 2.

Style rule

A CSS document (a *style sheet*) consists of a set of *style rules*. Each rule starts with a selector and is followed by a declaration block enclosed by curly braces. The selector defines a pattern, and the declarations are applied to the objects that match the pattern.

For a full description of CSS syntax, see *http://www.w3.org/TR/REC-CSS2/*.

The simple example below shows how to apply the color red to all emphasis elements.

```
em { color : red ; }
```

where `em` is the selector, and “`color : red ;`” is a declaration.

It is possible to group several rules with the same declarations. Use a comma “,” to separate the selectors. For example:

```
em, b { color : red ; }
```

Selector

The W3C states that “A selector represents a structure. This structure can be understood for instance as a condition that determines which elements in the document tree are matched by this selector, or as a flat description of the HTML or XML fragment corresponding to that structure.”

Examples of selector:

- H3
- P.footer
- TABLE#bigtable > TR
- TABLE#bigtable TD
- node
- node[x="2"]
- node:selected
- node#subgraph1 > #id2

A selector is composed of one or more simple selectors.

Examples of simple selectors:

- H3
- P.footer
- TABLE#bigtable
- TR
- node
- node[x="2"]
- node:selected
- #id2

A simple selector is made of minimal building blocks.

Examples of minimal building blocks of selectors:

- H3
- .footer
- node
- [x="2"]
- :selected

♦ #id2

When two or more simple selectors are aggregated into a selector, they are separated by combinators. A combinator is a single character which semantics is described in Table 3.1. Extra spaces are ignored.

Combinator Symbols

Transition	Meaning
E F	Matches an F element that is descendant of an E element.
E > F	Matches an F element that is a child of an E element.
E + F	Matches an F element immediately preceded by an E element.

The minimal building blocks of a selector are listed in *Minimal Building Blocks of a Selector* . For an explanation of the Specificity column, see *Priority*.

Minimal Building Blocks of a Selector

Minimal Building Block	Matching Rule	Specificity
e	Matches any element of type e.	0-0-1
#*myid*	Matches any element with ID equal to myid.	1-0-0
.*myclass*	Matches any element with class myclass.	0-1-0
:*myclass*	Matches any element with pseudo-class myclass.	0-1-0
[*myattr*]	Matches any element with the myattr attribute that exists and <> null.	0-1-0
[*myattr*=”*warning*”]	Matches any element whose myattr attribute value is exactly equal to warning.	0-1-0
[*myattr*~=”*warning*”]	Matches any element whose myattr attribute value is a list of space-separated values, one of which is exactly equal to warning.	0-1-0
*	Matches any element.	0-0-0

For example, the following line:

```
P.pastoral.marine { color : green ; size : 10pt ; }
```

matches `<P class="pastoral marine old">`, sets the color of the paragraph to `green`, and sets the font size to `10`.

All rules start and end with an implicit “ * ” pattern. This means that a selector can match anywhere inside the hierarchy.

Declaration

Declarations are key-value couples. The separator is a colon (:). Each declaration is terminated by a semicolon (;). The key should represent a predefined graphic attribute (`foreground`, `size`, `font`, and so forth) and the value is a literal whose type depends on the key (such as red, 10pt, or serif). All key-value pairs are String. It is recommended that you quote values with quotation marks " " or single quote marks ’ ’ when the values contain nonalphanumeric characters.

Priority

The priority of the rules depends on their relative *specificity*. Specificity is computed as three numbers, a-b-c (in a number system with a large base).

- **a** is the number of ID building blocks in the selector
- **b** is the number of classes, pseudo-classes, and attributes
- **c** is the number of element types

The examples in *Priority Order Example* are in priority order, with the most specific first.

Priority Order Example

Selector	Specificity
#title > #author.full	“2-1-0”
#title	“1-0-0”
P.intro P.citation	“0-2-2”
UL OL LI.red	“0-1-3”

When two rules give the same specificity number, the order of appearance gives the priority: the last to appear has higher priority than the previous rules with the same specificity.

Priority is used is as follows: first the declarations of all rules that match the same objects are merged, and then the priority is applied only if there is a conflict (same key value) within the merged declaration block.

Cascading

Cascading consists of supplying several sources for the style. In HTML environments there are three sources: the browser, the user, and the document. Cascading fixes another weight according to the source of the style. Document style takes precedence over user style, which takes precedence over browser style when the specificity number is the same.

There are two more tokens, !important and inherit. They are used to alter the cascading priority inside declarations.

A style sheet can also import other sheets (internal cascading). The syntax is:

```
@import "[url]" ;
```

Import statements must precede the first rule in a style sheet. Priorities of the imported rules are computed as if the rules replace the import statements. Here is an example of import:

```
@import "common.css" ;
```

Inheritance

The main principle of CSS is the inheritance of declarations. Once the rules are checked against the source document, the matched declarations are sorted according to the priority order of the rules. The declarations are merged, with higher priority settings overriding lower ones in case of conflict.

The resulting set of key-value pairs represents all the declarations that the style sheet applies to a particular document.

Applying CSS to Java objects

Explains how CSS is applied to Java objects.

In this section

The CSS Engine

The CSS selector mechanism was designed to match elements in HTML or XML documents. It can also be used to match a hierarchy of Java™ objects accessible from a model interface. In this context, the CSS level 2 recommendation is transposed for the Java language and used to set Bean properties according to the Java object hierarchy and state.

The CSS declarations for each model object are sorted and used according to the application that controls the CSS engine. The declarations represent property settings on a target object. The target object concerned depends on the way the CSS engine is used.

The CSS engine has different responsibilities at load time and at run time:

♦ At load time: creating and customizing series, points and elements of the chart itself.

♦ At run time: customizing the series and points according to model changes.

Usually the left side of a declaration represents a Bean property of the graphic object. The right side is a literal and, if it needs type conversion, the method `setAsText` is invoked on the Property Editor associated with the Bean property.

The Data Model

The input data model represents the seed of the “CSS for Java” engine. It provides three important kinds of information to the CSS engine, required to resolve the selectors:

♦ The tree structure of objects, which will be exploited by selector transitions.

♦ Object type, ID, and tag (or user-defined type), which match element type, ID, and CSS classes. IDs and types are strings; CSS classes are words separated by a space character. ID is not required to be unique, although it is wise to assume so.

♦ Attribute, which matches an attribute of the same name in an attribute condition within the selector.

The target object is the graphic object associated with the model object. The declarations change property values of the graphic object that corresponds to the matching model object, thereby customizing the graphic appearance given by the rendering.

Object Types and Attribute Matching

Setting a Property Value for a Class shows a rule that matches the object of class (type) `test_Vehicle`, with the attribute `model` equal to `sport`, and sets the property `icon` of the graphic object associated with this object (defined elsewhere) to `sport-car.gif`.

Setting a Property Value for a Class

```
test_Vehicle[model=sport] {icon : "sport-car.gif";}
```

Attribute matching can be used to add dynamic behavior: a `PropertyChange` event occurring on the model can activate the CSS engine to set new property values on the graphic objects.

Color Change Behavior Dependent on an Attribute Value shows a rule that changes the color of any object of CSS class `computer` whenever the model attribute `state` is set to `down`.

Color Change Behavior Dependent on an Attribute Value

```
.computer[state = down] {color : "gray"}
```

Object Identifiers and CSS Classes

All model objects have an ID. This ID can be checked against the # selector of a rule.

A "user-defined type” can be set for an object in a property called `CSSclass`. CSS classes are not necessarily related to data model semantics; they are devices to add to the pattern-matching capabilities in the style sheet. An object belongs to only one type but can belong to several (or no) CSS classes. A check on a CSS class is a check for its presence or absence. Therefore a CSS class can be seen as an attribute without a value.

Class Name

The `class` property is a reserved keyword indicating the class name of the generated graphic object. Obviously the class declaration is applied only when there is a creation request. If the model state changes, the graphic objects are customized by applying only new declarations coming from new matching rules of the style sheet. The class declaration is then simply ignored.

The right side of a class declaration is the fully-resolved name of the Java class, loaded with the system class loader. For example:

```
#annotation {
 class : ilog.views.chart.graphic.IlvDataLabelAnnotation;
 text : "Hello World !";
}
```

By extension, the class declaration also indicates the constructor to use to initialize the object. In the example above, an `IlvDataLabelAnnotation` instance is created by invoking the default constructor, and the text bean property is set to "Hello World !".

When no default constructor exists for a class, you can specify a particular constructor, provided the following limitation is respected: only constructors that export the parameters as Bean properties are supported. In other words, the class should have the following interface:

```
public class foobar {
 public foobar(FooType prop1, BarType prop2) {...}
 public FooType getFoo() {...}
 public BarType getBar() {...}
 ...
}
```

which gives the following class declaration:

```
#afoobar {
 class : my.package.foobar(foo, bar);
 foo: ...;
 bar: ...;
}
```

When the CSS engine resolves the declaration, it first looks for the types of `foo` and `bar` Bean properties, then tries to find a constructor with parameters of these types. If such a constructor exists, it is invoked. Find below a more concrete example with a `LineBorder`:

```
#border {
 class: javax.swing.border.LineBorder(lineColor, thickness);
 lineColor: 'red';
 thickness: 2;
}
```

will use the `LineBorder(Color color, int thickness)` constructor.

Pseudo-classes

Pseudo-classes are the minimal building blocks of a selector that match model objects according to an external context. The syntax is like a CSS class but with a colon instead of a dot. For example, `series:highlighted` matches a series only if the series is highlighted. The user agent can resolve this pseudo-class at run time according to the state of each series.

A pseudo-class has the same specificity as a CSS class.

Model Indirection

The right side of a declaration resolves to a literal that is determined at run time by a *Property Editor*. However, if the literal is prefixed by @, the remainder of the string is interpreted as a model attribute name. The declaration takes the value from the model object, as shown in *Setting a Property to an Attribute Value*.

Setting a Property to an Attribute Value

```
series { lineWidth : "@width" ;}
```

The `lineWidth` property will be set to the value of the attribute called `width` in the model

Resolving URLs

Sometimes declaration values are URLs relative to the style sheet location. A special construct, standard in CSS level2, allows you to create a URL from the base URL of the current style sheet. For example:

```
imageURL : url(images/icon.gif) ;
```

This declaration extends the path of the current style sheet URL with `images/icon.gif`. This construct is very useful for creating a style sheet with images located relative to it, because the URL remains valid even if the style sheet is cascaded or imported elsewhere.

CSS Recursion

You are likely to want to specify a Java object as the value of a declaration. A simple convention allows you to recurse in the style sheet, that is, to define a new Java object which has the same style sheet but is unrelated to the current data model.

@# Construct

Prefix the value with ‘@#’ to create new Beans when required as shown in *Creating a Bean in a Declaration*.

Creating a Bean in a Declaration

```
form {
       date : "@#dateBean" ;
      title : "CSS rules" ;
}
Subobject#dateBean {
      class : ’java.util.Date’ ;
      time  : ’23849291’ ;
}
```

The ‘@#’ operator extends the current data model by adding a dummy model object as the child of the current object. The object ID of the dummy object is the remainder of the string, beyond the ‘@#’ operator. The type of the dummy object is 'Subobject'. The dummy object inherits CSS classes and attributes from its parent.

The CSS engine creates and customizes a new subobject according to the declarations it finds for the dummy object. This means, in particular, that the Java class of the subobject is determined by value of the 'class' property. The newly created subobject becomes the value of the @# expression. In the declarations for the subobject, attribute references through the @ operator refer to the attributes of the parent object.

Once the subobject is completed, the previous model is restored so that normal processing is resumed.

In the above example, a `java.util.Date` object is created, with the `time` property set to `23849291`. This new object is assigned to the `date` property of the `form` object.

@= and @+ Constructs

There are two refinements of the '@#ID' operator:

♦ '@=ID'

  Using '@=ID' instead of '@#ID' shares the instance. The first time the declaration is resolved, the object is created as with the '@#' operator. But for all subsequent accesses to the same value, '@=ID' will return the same instance, the one created the first time, without applying the rules. Note that all instances created with '@=' are cleared when a new style sheet is applied. '

♦ '@+ID'

Using ‘@+ID’ instead of ‘@#ID’ avoids unecessary objects creation. Basically ‘@+ID’ customizes only the object currently assigned to the property, unless it does not exist or its class is not the same as the one defined in the #ID rule. In this case, the object is first created, then customized, and then assigned to the property, as with an ‘@#’ construct.

The need for these refinements arises from a performance issue. The '@#' operator creates a new object each time a declaration is resolved. Usually a declaration is applied whenever a property changes. Under certain circumstances, the creation of objects may lead to expensive processing, so IBM® ILOG® JViews Charts provides an optional mechanism to minimize the creation of objects during property changes.

@| Construct

A CSS declaration value starting with "@|" is interpreted as an expression (see *Expressions*).

@ Construct

A CSS declaration value that is exactly "@" means cancel the property setting made in a previous rule. This construct is useful to prevent a property from being modified, especially when the default value is unknown. For example:

```
series {
   lineWidth : 23 ;
}

series[name="Sales"] {
   lineWidth : @ ;
}
```

These two rules say that the `lineWidth` property value should be set to 23, unless the series has the name "Sales". Without the "@" ability, the default value of `lineWidth` would have to be written down in the CSS.

Expressions

The value in a CSS declaration is usually a literal. However, it is possible to write an expression in place of a literal.

If the value begins with “@|”, then the remainder of the value is processed as an expression.

The syntax of the expressions, after the "@|" prefix, is close to the Java syntax. The expression type can be arithmetic (type int, long, float, or double), Boolean, or String. Examples:

```
@|3+2*5                -> 13
@|true&&(true||!true)  -> true
@|start+end              -> "startend"
```

An expression can refer to model attributes. The syntax is the usual one:

`@|@speed/100+@drift` -> 1/100 of the value of "speed" plus the value of "drift." "speed" and "drift" are attributes of the current object.

`'@|"name is: " + @name'`-> "name is: Bob", if the value of current object attribute "name" is "Bob." Note the use of quotes to keep the space characters.

The standard functions `abs()`, `acos()`, `asin()`, `atan()`, `ceil()`, `cos()`, `exp()`, `floor()`, `log()`, `pi`, `rint()`, `round()`, `sin()`, `sqrt()`, and `tan()` are accepted, as in, for example:

```
@|3+sin(pi/2) -> 4
```

There are some default functions: `concat`, `int`, `long`, `float`, `double`. The first one concatenates its parameters as String; the others evaluate basic numerical expressions (only the four operators +,-,*,/ are allowed).

Custom Functions

Users of CSS for Java can register their own functions, which can be part of an expression. A custom function must implement `ilog.views.util.styling.IlvCSSFunction`. This is an abstract class, but technically you should treat it just like an interface.

The signature of the main method is as follows:

- ♦ When a function is evaluated, the parameters are first resolved as subexpressions. Then the final values of parameters are passed to the `args` array.
- ♦ The parameter `type` is the expected type of the function, when known. A `null` value is possible. Implementation should take care to return an object of this type; otherwise the conversion will only be performed if it can be (that is, if it is a simple conversion between primitive types or to String).
- ♦ The other parameters are the model, node, target, and function closure at invocation time: model is the current CSS model, node is the current model object being customized, and target is the graphic object being customized, and closure is the function closure that can be set by calling `ilog.views.util.css.IlvCSSBeans.setFunctionClosure`.

If an error occurs during the `call`, the exception will be reported and the current property setting will be canceled.

Custom Function Example: Average gives an example of a CSS function, which returns the average value of its parameters.

Custom Function Example: Average

```
import ilog.views.util.styling.IlvCSSFunction;

class Average extends IlvCSSFunction {
   //default constructor
   public Average() { }

   // Returns 'avrg'
   public String getName() {
      return "avrg";
   }

   // Returns ','
   public String getDelimiters() {
      return ",";
   }

   // Returns the average of arguments
   public Object call(Object[] args, Class type, IlvCSSModel model,
                      Object node, Object target, Object closure) {
      // Assume only double, for the sake of simplicity.
      double result = 0d;
      for (int i=0; i<args.length; i++) {
         if (args[i] != null) {
            result += Double.parseDouble(args[i].toString());
         }
      }
      result /= args.length;
      return new Double(result);
   }
}
```

Calling the Custom Function Average shows an example of calling the `avrg`.

Calling the Custom Function Average

```
   elevation : @|avrg(@param1,@param2);
```

Registering Custom Functions

You must register custom functions before using them in a style sheet.

To register a function, you can simply call `registerFunction` in `ilog.views.chart.IlvChart`.

Divergences from CSS2

Java objects are not HTML documents. The CSS2 syntax remains, so that a CSS editor can still be used to create the style sheet. However, the differences lead to adaptations of the CSS mechanism so that its power can be fully exploited and directed to some specific behavior.

Cascading

Cascading is explicit: the API offers a means of cascading style sheets. However, the `!important` and `inherit` tags are not supported for the sake of simplicity.

Pseudo-classes and Pseudo-elements

In CSS2 there are predefined pseudo-classes and the notion of pseudo-elements. In JViews, when applied to Java objects, there is no predefined pseudo-class for a data set, but you can define your own and use them in style rules.

The CSS2 predefined pseudo-elements and pseudo-classes (:link, :hover, and so forth) are not implemented because they have no meaning in Java.

Attribute Matching

The attribute pattern in CSS2 makes the following checks for strings: presence [`att`], equality [`att=`val], and inclusion [`att~=`val] . The `|=` operator is disabled.

For Java objects, there are the following numeric comparators `>`, `>=`, `<>`, `<=`, `<`, with the usual semantics.

There are also equal and not-equal comparators which make the distinction between string comparison and numerical comparison:

- Equal: "A==B" is true if and only if A and B are numerically equal (for example, 10 == 10.0); use "=" to test the equality of two Strings.
- Not-equal: "A~B" is true if and only if A and B are two different Strings (for example, "10" ~ "10.0"); use "<>" to test the inequality of two numbers.

Operators Available in the Attribute Selectors

Operator	Meaning	Applicable To
A	present	strings
A=val	equals	strings
A~val	not equals	strings
A~=val	contains the word	strings
A==val	equals	numbers
A<>val	not equals	numbers
A<val	less than	numbers
A<=val	less than or equals	numbers
A>val	greater than	numbers
A>=val	greater than or equals	numbers

Syntax Enhancement

CSS for Java requires the use of quotation marks when a token contains special characters, such as dot (.), colon (:), at sign (@), pound sign (#), space ( ), and so on.

Quotes can be used almost everywhere, in particular to delimit a declaration value, a minimal building block denoting a type, or a CSS class with reserved characters.

The closing “;” is optional.

Null Value

Sometimes it makes sense to specify a null value in a declaration. By convention, `null` is a zero-length string '' or "". For example:

```
public Object call(Object[] args,Object closure,  Class type, IlvCSSModel
               model, Object target, Object closure);
series[name="Foo"] {

  color1 : '';
}
```

This will reset the color of the data series to its default value. The notation '' is also used to denote a null array for properties expecting an array of values.

Empty String

The null syntax does not allow you to specify an empty string in the style sheet. Instead, you can create an empty string, as shown in *Creating an Empty String*.

Creating an Empty String

```
chart {
  headreText : @#emptyString ;
}

Subobject#emptyString {
  class : 'java.lang.String';
}
```

Better still, you can use the sharing mechanism to avoid the creation of several strings. The `@=` construct will create the empty string the first time only and will then reuse the same instance for all other occurrences of `@#emptyString`, see *Sharing an Empty String* .

Sharing an Empty String

```
chart {
  headerText : @=emptyString ;
}

Subobject#emptyString {
  class : 'java.lang.String';
}
```

Styling

Introduces the usage of style sheets within the JViews Charts library.

In this section

Two kinds of rules

You can distinguish two sets of style rules:

♦ Rules that customize the global appearance of the chart.

  These rules are applied to the elements of the chart, such as chart area, legend, scales and grids. You can find a detailed description in the section *Styling the Chart Component*.

♦ Rules that control how individual data series or data objects are rendered.

These rules are applied to the graphical representation of data points. You can find a detailed description in *Styling the data series*.

Styles

The appearance of a chart can be dynamically controlled with cascading style sheets (CSS). Cascading style sheets are introduced in the *Using CSS Syntax in the Style Sheet.*

Applying styles

The `IlvChart` class implements the `IlvStylable` interface, which defines several methods to control the styling.

Here is an example of the typical code involved when you apply style sheets to a chart:

```
try {
  chart.setStyleSheets(new String[]{“simple.css”});
} catch (IlvStylingException x) {
  System.err.println("Cannot load style sheets: " + x.getMessage());
}
```

The `chart.setStyleSheets` method expects an array of `String` objects, which can represent either a URL, a file name, or the style sheet string directly.

Before the style sheets are applied, the chart configuration is restored to a default state, which basically corresponds to the state of a default `IlvChart` instance. This ensures that the application of two consecutive sets of style sheets does not produce an undesired cascading of styles. For more information on the operations performed to restore the state of the chart, you can refer to the documentation of the `resetStyles()` method in the Reference Manual. Please remember that some elements of the chart are re-created, such as grids or scales. As a consequence, to use Java™ code to customize your chart on top of CSS, you can:

♦ invoke the Java code *after* the style sheets are applied,

  or

♦ override `IlvChart.resetStyles`, so that it restores the chart to a different state.

You can also integrate a style sheet generated with the Designer. For more information, see Integrating your development into an application in *Using the Designer*.

Disabling styling

To disable the styling, the chart component provides two different solutions:

♦ Globally disable the styling by passing `null` to the `setStyleSheets(java.lang.String [])` method.

  This tells the chart that no style is specified and removes any overhead related to the styling. Note that this is different from setting an empty style sheet, since in this case the chart will still try to match style rules.

♦ Disable the dynamic interpretation of style rules when the data model changes.

  You can do this by means of the `setDynamicStyling(boolean)` method. When the dynamic styling is turned off, the chart will not re-apply styles if the contents of a data set are modified (for example, when new data points are added or when the values of a data point change). For efficiency reasons, it is recommended to toggle off the dynamic styling if your style sheet does not contain any rule based on the attributes of a data point or if your data model is static.

Styling the Chart Component

Describes how style sheets can be used to customize the appearance of the chart component and its subelements.

In this section

The chart renderers

Describes the `chartRenderer` selector.

Scales and grids

Describes how to reference a scale or a grid in a style rule.

Parts of the chart component

The following table lists the CSS selectors that are defined to reference the different parts of the chart component:

Selector	Description	Target Class
`chart`	The chart component	`IlvChart`
`chartArea`	The chart area component	`IlvChart.Area`
`chartLegend`	The chart legend	`IlvLegend`
`chart3Dview`	The chart 3-D view	`IlvChart3DView`
`chartRenderer`	The chart renderers	`IlvChartRenderer`
`chartScale`	The chart scales	`IlvScale`
`chartGrid`	The chart grids	`IlvGrid`

These selectors can be used to modify the Bean properties of the corresponding target class. For example, the following style rules show you how to control the borders and the colors of the chart and the chart area:

```
chart {
  foreground      : black;
  background      : lightyellow;
  opaque          : true;
  border          : @#chartBorder;
}

Subobject#chartBorder {
  class       : 'javax.swing.border.LineBorder(lineColor)';
  lineColor   : black;
}

chartArea {
  plotBackground  : oldlace;
}
```

Styling the chart

The `chart` selector identifies the Chart component and can be used to control its appearance. *Beans Properties for Chart* lists the Bean properties of the `IlvChart` class that can be set in the declarations of a CSS style rule.

Beans Properties for Chart

Property	Type	Allowed Values
`scalingFont`	Boolean	
`antiAliasing`	Boolean	
`antiAliasingText`	Boolean	
`shiftScroll`	Boolean	
`scrollRatio`	Double	
`type`	int	CARTESIAN,POLAR,RADAR,PIE
`projectorReversed`	Boolean	
`defaultColors`	Color[]	
`decorations`	Function ”decorations”	
`renderingType`	int	BAR,STACKED_BAR,STACKED100_BAR,

Property	Type	Allowed Values
		SUPERIMPOSED_BAR,AREA,STACKED_AREA,STACKED100_AR
dataSource	IlvDataSource	
optimizedRepaint	Boolean	
dynamicStyling	Boolean	
3d	Boolean	
interactors	IlvChartInteractor[] Or Function “interactors”	
plotAreaBackground	Color	
backgroundPaint	Paint	
header	JComponent	
headerText	String	
footer	JComponent	
footerText	String	
legendVisible	Boolean	
legendPosition	String	NORTH_BOTTOM, NORTH_WEST,WEST, SOUTH_WEST,SOUTH_
foreground	Color	
border	Border	

Styling the chart area

The `chartArea` selector identifies the Chart Area component and can be used to control the appearance of the portion of the chart.

Chart Area Bean Properties lists the Bean properties of the `IlvChart.Area` class that can be set in the declarations of a CSS style rule.

Chart Area Bean Properties

Property	Type
`background`	Color
`backgroundPaint`	Paint
`border`	Border
`bottomMargin`	int
`filledPlottingArea`	Boolean
`foreground`	Color
`font`	Font
`leftMargin`	int
`opaque`	Boolean
`plotBackground`	Paint
`plotStyle`	IlvStyle
`rightMargin`	int
`margins`	Insets
`topMargin`	int

Styling the chart legend

The `chartLegend` selector identifies the Chart Legend component and can be used to control its appearance. *Beans Properties for Chart Legend* lists the Bean properties of the `IlvLegend` class that can be set in the declarations of a CSS style rule.

Beans Properties for Chart Legend

Property	Type
`antiAliasing`	Boolean
`antiAliasingText`	Boolean
`border`	Border
`background`	Color
`floating`	Boolean
`floatingLayoutDirection`	int
`followChartResize`	Boolean
`font`	Font
`foreground`	Color
`interactive`	Boolean
`location`	Point
`movable`	Boolean
`paintingBackground`	Boolean
`symbolSize`	Dimension
`fsymbolTextSpacing`	int
`title`	String
`transparency`	int

Styling chart 3-D view

The `chart3DView` selector identifies the Chart 3-D view and can be used to control its appearance. *Beans Properties for Chart 3-D View* lists the Bean properties of the `IlvChart3DView` class that can be set in the declarations of a CSS style rule.

Beans Properties for Chart 3-D View

Property	Type	Allowed Values
`ZAnnotationRenderer`	IlvLabelRenderer	
`ZAnnotationVisible`	Boolean	
`ZGridVisible`	Boolean	
`ZGridStroke`	Stroke	
`ZGridPaint`	Paint	
`lightLatitude`	double	
`lightLongitude`	double	
`ambientLight`	float	
`projectionType`	int	OBLIQUE, ORTHOGRAPHIC
`elevation`	double	
`rotation`	double	
`depth`	int	
`depthGap`	int	
`zoom`	double	
`autoScaling`	Boolean	

Styling the chart grids

The chartGrid selector identifies the Chart Grid component and can be used to control its appearance. *Beans Properties for Chart Grids* lists the Bean properties of the IlvGrid class that can be set in the declarations of a CSS style rule.

Beans Properties for Chart Grids

Property	Type	Allowed Values
drawOrder	int	
majorLineVisible	Boolean	
visible	Boolean	
minorLineVisible	Boolean	
majorStroke	Stroke	
majorPaint	Paint	
minorStroke	minorPaint	

Styling the chart scale

The `chartScale` selector identifies the Chart Scale component and can be used to control its appearance. *Beans Properties for Chart Scale* lists the Bean properties of the `IlvScale` class that can be set in the declarations of a CSS style rule.

Beans Properties for Chart Scale

Property	**Type**	**Allowed Values**
`axisStroke`	Stroke	
`labelFont`	Font	
`labelColor`	Color	
`foreground`	Color	
`drawOrder`	int	
`majorTickSize`	int	
`minorTickSize`	int	
`tickLayout`	int	TICK_INSIDE, TICK_OUTSIDE, TICK_CROSS
`labelOffset`	int	
`titleOffset`	int	
`autoWrapping`	Boolean	
`labelAlignment`	int	LEFT, CENTER, RIGHT
`titleRotation`	float	
`title`	string	
`titlePlacement`	int	
`labelRotation`	float	
`skippingLabel`	Boolean	
`skipLabelMode`	int	CONSTANT_SKIP,ADAPTIVE_SKIP
`visible`	Boolean	
`axisVisible`	Boolean	
`majorTickVisible`	Boolean	
`minorTickVisible`	Boolean	
`labelVisible`	Boolean	
`autoCrossing`	Boolean	
`crossingValue`	double	
`stepsDefinition`	IlvStepsDefinition	
`category`	Boolean	
`annotations`	IlvScaleAnnotation[]	

Styling the Chart Component

You can find the files and the source code of this example in the **`<installdir>/jviews-charts86/samples/style/index.html`** directory.

This sample displays a chart that loads data from the following an XML file:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/jviews+css" href="simple.css"?>
<!DOCTYPE chartData PUBLIC '-//ILOG//JVIEWS/Chart 1.0' 'chartxml.dtd'>
<chartData version="1.0">
  <data>
    <labels>Server 1, Server 2, Server 3, Server 4</labels>
    <series id="CPU #1" type="double">
      <valuesList>90,80,65,78</valuesList>
      <property name="color" javaClass="java.awt.Paint">
        #ff9d9d-#924242
      </property>
    </series>
    <series id="CPU #2" type="double">
      <valuesList>80,55,40,60</valuesList>
      <property name="color" javaClass="java.awt.Paint">
        #78ff78-#249224
      </property>
    </series>
    <series id="CPU #3" type="double">
      <valuesList>50,35,25,20</valuesList>
      <property name="color" javaClass="java.awt.Paint">
        #98bdff-#476aa9
      </property>
    </series>
  </data>
</chartData>
```

This data corresponds to a simple log of CPU usage for several servers. The XML file contains a processing instruction that references a style sheet URL. When the XML data source is connected to the chart, the style sheet is automatically applied. Each series in the XML file defines a `color` property. Next we will see how this property can be used by a CSS declaration.

To describe the contents of a simple style sheet:

1. Specify that a bar chart is used and that the legend is visible:

   ```
   chart {
     legendVisible: true;
     renderingType: BAR;
   }
   ```

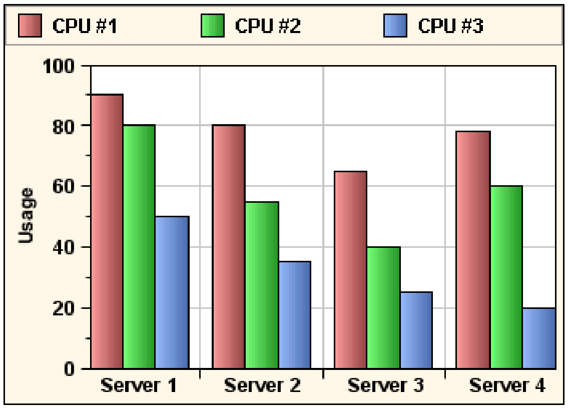
2. Tell the chart to use a category x-scale, add a title to the y-scale, and explicitly specify the data limits for the y-axis:

```
#xScale { category: true; }

#yScale { title: “Usage(%)”; titleRotation: 270;}

#yScale axis { dataMin: 0; dataMax: 100; }
```

3. Specify that the color used to render a series is equal to its `color` property, which has been defined in the XML file:

```
series { color1: @color; }
```

Style Sheet: First Example shows what the chart looks like with this style sheet:



Style Sheet: First Example

The chart renderers

The `chartRenderer` selector can be used to control the global appearance of renderers used by the chart. This selector supports an `index` attribute to select the appropriate chart renderer. For example, the following rule hides the first renderer:

```
chartRenderer[index=0] { visible : false; }
```

The purpose of the `chartRenderer` rule is to specify global settings to the top-level renderers that are accessible through the `IlvChart.getRenderer` method. For example, you can use it to set the mode of a stacked chart:

```
chartRenderer { stacked100Percent: true; }
```

Note: The `chartRenderer` rule cannot be used to customize the child renderers independently. The section *Styling the data series* explains how renderers can be customized for a specific series or a specific set of points.

Scales and grids

There are two ways to reference a specific scale, or a grid, in a style rule:

♦ You can use the `axisIndex` attribute of the `chartScale` and `chartGrid` selectors. This index equals –1 for the x-axis, or corresponds to the index of an y-axis.

♦ You can use the `xScale`, `yScale`, `xGrid` and `yGrid` CSS ID.

Here are a few examples of selector patterns:

```
// Match the x-scale.
chartScale[axisIndex=”-1”] { ... }
#xScale                    { ... }
// Match the main y-scale.
chartScale[axisIndex=”0”]  { ... }
#yScale                    { ... }
// Match the x-grid.
chartGrid[axisIndex=”-1”]  { ... }
#xGrid                     { ... }
// Match the main y-grid.
chartGrid[axisIndex=”0”]   { ... }
#yGrid                     { ... }
```

Each scale object supports an `axis` child selector, which can be used to customize the associated `IlvAxis` instance. For example, here is how you can reverse the x-axis:

#xScale axis { reversed: true; }

Styling the data series

Explains the expected selector patterns for the style rules and the properties that can be used in the declarations of these rules.

In this section

Selector patterns

Two selectors are defined to reference the chart data model:

- `series`

  Used to match the whole series (represented by `IlvDataSet` instances in the data model).

- `point`

  Used to match individual data points. The objects that match a `point` selector are direct descendants of an object that matches a `series` selector.

CSS Classes and Pseudo Classes

CSS classes can be associated with data sets through a predefined property named `CSSclass`. CSS classes can be specified in an XML file, as shown by the following XML fragment:

```
<series id="Series1">
  <valuesList>3.0,6.5,6.8,12.0</valuesList>
  <property name="CSSclass">tag</property>
</series>
```

You can also use the `setCSSClasses(ilog.views.chart.data.IlvDataSet, java.lang.String)` method to specify the CSS classes of a data set. The CSS classes can thereby be used in the selector of a style rule:

```
series.tag { ... }
```

Likewise, you can add or remove pseudoclasses by means of the `addPseudoClass(ilog.views.chart.data.IlvDataSet, java.lang.String)` and `removePseudoClass(ilog.views.chart.data.IlvDataSet, java.lang.String)` methods. There is no predefined pseudoclass for a data set, but you can define your own and use them in style rules.

For example, you can define a highlighted state for a data set:

```
// Add the “highlighted” pseudoclass to a data set.
IlvDataSetProperty.addPseudoClass(dataSet, “highlighted”);
```

Then, you can define the following rule:

```
series:highlighted { ... }
```

For more information on how to use this technique, refer to the source code in **`<installdir>/jviews-charts86/samples/css/src/css/ChartCSSDemo.java`** of the Highlighter interactor defined in the CSS demonstration.

Selector Attributes

The following attributes are defined for the `series` simple selector:

name	Description
name	The name of the data set, as returned by the `getName()` method.
index	The index of the data set in its data source.

On top of these two predefined attributes, the selector can reference any attribute accessible by the `getProperty(java.lang.Object)` method.

The following attributes are defined for the `point` simple selector:

name	Description
x	The x-value of the data point, as returned by the `getXData(int)` method.
y	The y-value of the data point, as returned by the `getYData(int)` method.
index	The index of the data point in the data set.
label	The label of the data point, as returned by the `getDataLabel(int)` method.

Here are a few examples of selector patterns that use attribute matching:

```
// Matches the series whose name is “Sales”.
series[name=”Sales”] { ... }

// Matches all the series, except the first one.
series[index<>”0”] { ... }

// Matches all data points with a positive y-value.
point[y>=0] { ... }

// For the “Sales” series, matches the data points whose y-value
// is greater than 1000. The ‘>’ transition is used to denote that
// point is a child of series.
series[name=”Sales”] > point[y>1000] { ... }
```

By using model indirection, you can reference the attributes on the right side of a declaration. Note that you can reference the attribute of a `series` within the declarations of a rule with a `point` selector.

For example, suppose that the `series` define an `overloadColor` attribute. You can define the following rule:

```
// For all data points with a y-value greater than 100, assign a color
// equal to the value of the ‘overloadColor’ attribute.
point[y>=0] {
  color1: @overloadColor;
}
```

Properties

Style sheets can be used to specify the rendering attributes of the whole data series or single data points.

Note: Properties for the `point` selector are also available for the `series` selector.

Properties for Data Points

Name	Type	Default Value
`color1`	`java.awt.Paint`	`null`
`color2`	`java.awt.Paint`	`null`
`endCap`	`int (enumerated)`	`java.awt.BasicStroke.CAP_BUTT`
`lineJoin`	`int (enumerated)`	`java.awt.BasicStroke.JOIN_BEVEL`
`lineStyle`	`float[]`	`null`
`lineWidth`	`float`	`1`
`miterLimit`	`float`	`10`
`stroke`	`java.awt.Stroke`	`null`
`annotation`	`IlvDataAnnotation`	`null`
`visible`	`boolean`	`true`

Colors

The `color1` and `color2` properties correspond to the primary and the secondary color, respectively. The meaning of these colors depends on whether the point is displayed by a filled renderer (see `isFilled()`):

♦ For renderers that are filled, the primary color corresponds to the fill color, and the secondary color corresponds to the stroke color.

♦ For renderers that are not filled, the primary color corresponds to the stroke color, and the secondary color is not used (it is, however, set as the fill color of the `IlvStyle` used by the renderer).

Stroke Style

The stroke that is used by the graphical representation of a data point can be specified either:

♦ by setting the `stroke` property.

You can do this by using an @-construct to reference a `java.awt.Stroke` instance, or

- by setting the various line attributes: `endCap`, `lineJoin`, `lineStyle`, `lineWidth` and `miterLimit`.

Note: If the `stroke` property is set, these properties are ignored.

For example, the following rules are equivalent:

```
series  {
    lineWidth: 2;
    endCap: CAP_ROUND;
    lineJoin: JOIN_ROUND;
}
```

and:

```
series  {
    stroke: @=stroke1;
}

Subobject#stroke1 {
    class : 'java.awt.BasicStroke(lineWidth, endCap, lineJoin)';
    lineWidth : 2;
    endCap : CAP_ROUND;
    lineJoin : JOIN_ROUND;
}
```

Visibility

The `visible` property allows you to toggle the visibility of data points. For example:

```
// Hide the series whose name is “CPU #1”.
series[name=”CPU #1”]  {
    visible: false;
}

// For all series, hide the points whose y-value is negative.
point[y < 0]  {
    visible: false;
}
```

Annotation

The `annotation` property lets you connect an instance of `IlvDataAnnotation` to a data point. For more information on data annotations, please refer to the section *Annotations*.

Here is an example of a rule that associates an icon with a set of data points:

```
// For the “CPU #1” series, set an icon on the points
// whose y-value is greater than 50.
series[name=”CPU #1”] > point[y>50]  {
    annotation: @=upperAnnotation;
}

Subobject#upperAnnotation {
    class: 'ilog.views.chart.graphic.IlvDefaultDataAnnotation(URL, position,
offset)';
    URL: url('gif/ok.gif');
    position: NORTH;
    offset: 2;
}
```

Properties for Data Series

On top of the data point properties, the series selector can be used to modify the Bean properties of the renderer that displays the corresponding data set. For example, you can define the following rules:

```
// Specify that the series with a “hidden” CSS class are not
// displayed by the legend.
series.hidden {
  visibleInLegend: false;
}
// Specify that a circle marker symbol must be used for all series.
// Note that this rule only affect series that are displayed by a
// renderer using marker symbols (Scatter, Bubble and Line charts).
series {
  marker: Circle;
}
```

The available properties depend on the renderer.

Properties Available for All Renderers

Name	Type	Default value	Description
visible	boolean	true	Indicates whether the renderer is visible.
name	string	null	The name of the renderer.
visibleInLegend	boolean	true	Indicates whether the renderer creates a legend item.
labeling	enumerated: DataLabel, Percent, XYValues, XValues, YValues	YValues	The data labeling mode.
labelLayout	enumerated: Outside, Centered	Centered	The layout mode for the data labels.

Properties Available for Polyline Chart Renderers

Name	Type	Default value	Description
stacked100Percent	boolean	false	If true, the sum of all y-values for a given x-value is scaled to 100.
autoTransparency	boolean	false	Indicates whether the renderer should use transparent default colors.
marker	enumerated: CIRCLE, CROSS, DIAMOND, PLUS, TRIANGLE, SQUARE, NONE	NONE	Additional marker drawn at each data point.
markerSize	int	3	Size of marker.

Properties Available for Bar Chart Renderers

Name	Type	Default value	Description
stacked100Percent	boolean	false	If true, the sum of all y-values for a given x-value is scaled to 100.
overlap	double, between 0 and 100	0	The amount by which bars overlap.
widthPercent	double, between 0 and 100	80	The amount of space available for each cluster.
autoTransparency	boolean	false	Indicates whether the renderer should use transparent default colors.

Properties Available for Scatter Chart Renderers

Name	Type	Default value	Description
marker	enumerated: CIRCLE, CROSS,	SQUARE	Marker drawn at each data

Name	Type	Default value	Description
	DIAMOND, PLUS, TRIANGLE, SQUARE, NONE		point.
markerSize	int	3	Size of marker.

Properties Available for Bubble Chart Renderers

Name	Type	Default value	Description
marker	enumerated: CIRCLE, CROSS, DIAMOND, PLUS, TRIANGLE, SQUARE, NONE	CIRCLE	Marker drawn at each data point.
minSize	int	10	Minimum size of a bubble.
maxSize	int	30	Maximum size of a bubble.

Properties Available for High/Low Chart Renderers

Name	Type	Default value	Description
overlap	double, between 0 and 100	0	The amount by which bars overlap.
widthPercent	double, between 0 and 100	80	The width of the graphical representation of a data point along the x-axis.

Properties Available for Pie Chart Renderers

Name	Type	Default value	Description
holeSize	int, between 0 and 100	0	Size of the hole in a doughnut chart.
strokeOn	boolean	true	Indicates whether the outline of the slices is drawn.

For more information on the available properties, please refer to the documentation of the corresponding renderer classes.

Note: The properties that are specified in a rule using the `series` selector usually override the settings that are specified by the `chartRenderer` rule. See the section *Styling the Chart Component* for more information.

Styling the chart data

You can find the files and the source code of the sample in **`<installdir>/jviews-charts86/codefragments/chart/styling/src/ChartCSSExample.java`**.

This sample displays a chart that loads data from the following an XML file:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/jviews+css" href="simple.css"?>
<!DOCTYPE chartData PUBLIC '-//ILOG//JVIEWS/Chart 1.0' 'chartxml.dtd'>
<chartData version="1.0">
  <data>
    <labels>Server 1, Server 2, Server 3, Server 4</labels>
    <series id="CPU #1" type="double">
      <valuesList>90,80,65,78</valuesList>
      <property name="color" javaClass="java.awt.Paint">
        #ff9d9d-#924242
      </property>
    </series>
    <series id="CPU #2" type="double">
      <valuesList>80,55,40,60</valuesList>
      <property name="color" javaClass="java.awt.Paint">
        #78ff78-#249224
      </property>
    </series>
    <series id="CPU #3" type="double">
      <valuesList>50,35,25,20</valuesList>
      <property name="color" javaClass="java.awt.Paint">
        #98bdff-#476aa9
      </property>
    </series>
  </data>
</chartData>
```

This data corresponds to a simple log of CPU usage for several servers. The XML file contains a processing instruction that references a style sheet URL. When the XML data source is connected to the chart, the style sheet is automatically applied. Each series in the XML file defines a `color` property. Next we will see how this property can be used by a CSS declaration. You are going to see how another style sheet can be applied to the same data to produce a different display.

To display a horizontal bar chart, with the color of each bar reflecting the CPU usage:

1. Reverse the Cartesian projector and the x-axis:

   ```
   chart { projectorReversed: true; }

   #xScale axis { reversed: true; }
   ```

2. Hide the ticks of the scales:

```
chartScale {
  minorTickVisible: false;
  majorTickVisible: false;
}
```

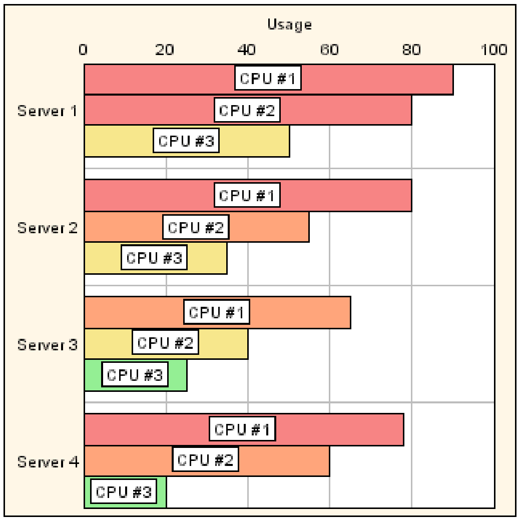
3. Specify that the color of a data point depends on its y-value:

```
point[y>0]  { color1: lightgreen; }

point[y>25] { color1: khaki; }

point[y>50] { color1: lightsalmon; }

point[y>75] { color1: lightcoral; }
```

4. For each data point, add a text annotation that is equal to the name of the corresponding data set:

```
point { annotation: @#annotation; }

Subobject#annotation {
  class :'ilog.views.chart.graphic.IlvDataLabelAnnotation';
  text  : @name;
}
```

Style Sheet: Second Example shows how the chart looks with this style sheet:



Style Sheet: Second Example

Styling the data objects

Describes the treemap chart renderer, which is used to show data objects (as opposed to data points in the other renderers).

In this section

Selector Patterns
Explains the expected selector patterns for the style rules and the properties that can be used in the declarations of these rules.

Properties
Lists the properties that can be used to customize the rendering of data models and data objects.

Selector Patterns

Style sheets can be used to specify the rendering attributes of all objects together and of individual objects.

Two types of style rule selectors are defined to reference the chart data model:

♦ objects

  Used to match the entire data source (an `IlvTreeTableDataSource`).

♦ object

  Used to match an object in the data model (available through `IlvTreeTableDataSource.getTreeModel()`).

CSS Classes and Pseudo Classes

Every object in the data model can be associated with one or more CSS classes and with one or more pseudoclasses.

Both kinds of classes can have an effect on the rendering, through style rules. But while CSS classes are meant to encode properties of the object that come directly from the model, the pseudoclasses are meant to encode application dependent state, for example in conjunction with some interactors.

The CSS classes of model objects are taken from the model. For this purpose, the renderer has methods `setCSSClassesColumn` and `setCSSClassesColumnName`, that let you specify the model column containing the CSS classes. The value in this column should be a string containing the CSS class names, in any order, and separated by spaces.

CSS classes can be used in a CSS file through the syntax:

```
object.classname { ... }
```

The CSS pseudoclasses of model objects are stored in the chart instance. You can add or remove a pseudoclass to an object (or an object to a pseudoclass) through the methods `IlvChart.addPseudoClass` and `IlvChart.removePseudoClass`. There are no predefined pseudoclasses for model objects so far. You can define your own and use them in style rules.

For example, in an interactor you might mark an object as "marked":

// Add the "marked" state to an object.

```
chart.addPseudoClass(object, "marked");
```

Then you can define a rule that highlights the marked objects:

```
object:marked { ... }
```

Selector Attributes

No attributes are defined for the `objects` selector.

The attributes defined for the `object` selector are exactly the column names of the object model. For example, if the model has two columns named 'amount2004' and 'amount2005', you can write a style rule

```
object[ amount2005 > amount2004 * 1.05 ]
```

to highlight those object where the amount increase is greater than 5%.

Additionally, the following attributes are predefined for the `object` selector:

Name	Description
`level`	The distance of the object from the model root, in the tree model. This is 0 for the root, 1 for the root children, and so on.
`focusLevel`	The distance of the treemap renderer focus (= the root of the subtree model being displayed) from the model root.
`leaf`	`true` if the object has no children objects in the model, `false` otherwise.

For a treemap display, the CSS expression `@level-@focusLevel` represents the level difference between the current object and the outermost displayed rectangle.

The following lines of CSS can therefore be used to set labels with particular attributes on each object of the specified level:

```
object [level==0] {
  annotation : "@#labelAnnotation0";
}
object [level==1] {
  annotation : "@#labelAnnotation1";
}
object [level>1] {
  annotation : "@#labelAnnotation2";
}
```

Likewise for specified levels relative to the treemap focus:

```
object [@|@level-@focusLevel==0] {
  annotation : "@#labelAnnotation0";
}
object [@|@level-@focusLevel==1] {
  annotation : "@#labelAnnotation1";
}
object [@|@level-@focusLevel>1] {
  annotation : "@#labelAnnotation2";
}
```

Properties

Properties for Individual Data Objects

Note: Properties for the `object` selector are also available for the `objects` selector.

In rules with an object selector, the following properties are available:

Name	**Type**	**Default Value**
`color1`	`java.awt.Paint`	`null`
`color2`	`java.awt.Paint`	`null`
`endCap`	`int (enumerated)`	`java.awt.BasicStroke.CAP_BUTT`
`lineJoin`	`int (enumerated)`	`java.awt.BasicStroke.JOIN_BEVEL`
`lineStyle`	`float[]`	`null`
`lineWidth`	`float`	`1`
`miterLimit`	`float`	`10`
`stroke`	`java.awt.Stroke`	`null`
`annotation`	`IlvDataAnnotation`	`null`
`visible`	`boolean`	`true`

For more details on these properties, see *Properties for Data Points*.

Properties for the Entire Data Model

In a rule with an objects selector, the following properties can be set through CSS.

- Object properties that apply to all objects. The list is the same as above.
- Renderer properties. For the treemap, the following properties are available:

Name	Type	Default value	Description
visible	boolean	true	Indicates whether the renderer is visible.
name	String	null	The name of the renderer.
visibleInLegend	boolean	true	Indicates whether the renderer creates legend items.
areaColumnName	String	null	The column chosen to determine the area of each object.
colorColumnName	String	null	The column chosen to determine the color of each rectangle.
labelColumnName	String	null	The column chosen to determine the label of each object, as shown by `IlvDataLabelAnnotation`.
CSSClassesColumnName	String	null	The column chosen to determine the CSS classes of each object.
packing	enumerated: ALTERNATING, SQUARIFIED_CORNER, SQUARIFIED, BAR	ALTERNATING	The rectangle packing.
primaryDirection	enumerated: RIGHT, LEFT, TRAILING, TOP, BOTTOM	TRAILING	The primary direction of arrangement of the rectangles.
secondaryDirection	enumerated: RIGHT, LEFT TRAILING, TOP, BOTTOM	BOTTOM	The secondary direction of arrangement of the rectangles.
leadingMarginProportion	double, between 0 and 1	0.02	Leading margin proportion.
maximumLeadingMargin	double	2	Maximum leading margin, in pixels.
trailingMarginProportion	double, between 0 and 1	0.01	Trailing margin proportion.
maximumTrailingMargin	double	2	Maximum trailing margin, in pixels.
bottomMarginProportion	double, between 0 and 1	0.02	Bottom margin proportion.
maximumBottomMargin	double	2	Maximum bottom margin, in pixels.
topMarginProportion	double, between 0 and 1	0.05	Top margin proportion.
maximumTopMargin	double	30	Maximum top margin, in pixels.
marginReductionFactor	double, between 0 and 1	1.0	Describes the margin proportions of a nested rectangle, compared to the

Name	Type	Default value	Description
			margin proportions of the rectangle that contains it.
useMarginsOnRoot	boolean	true	Indicates whether margins also apply to the outermost rectangle.
focusParentMarginProportion	double, between 0 and 1	0.02	Focus parent margin proportion.
maximumFocusParentMargin	double	5	Maximum focus parent margin, in pixels.
colorScheme	enumerated: CONSTANT, DEPTH, SEQUENTIAL, SEQUENTIAL_HUE, CYCLIC_SEQUENTIAL_HUE, QUALITATIVE, DIVERGING_RED_GREEN, DIVERGING_GREEN_RED, DIVERGING_BLUE_YELLOW, DIVERGING_YELLOW_BLUE, AVERAGE_RED_GREEN, AVERAGE_GREEN_RED, AVERAGE_BLUE_YELLOW, AVERAGE_YELLOW_BLUE	CONSTANT	The color scheme.
colorForNaN	Color	#AAAAAA	The color used when a value in the color column is NaN.
strokeThreshold	int	0	The minimum size of a rectangle, under which the

Name	Type	Default value	Description
			stroke of a rendering style is not used.
annotation	IlvDataAnnotation	null	An annotation for all the tree nodes.
labelAlignmentX	enumerated: LEFT, CENTER, RIGHT, LEADING, TRAILING	LEADING	The horizontal alignment of each label within the rectangle to which it belongs.
labelAlignment Y	enumerated: BOTTOM, CENTER, TOP	TOP	The vertical alignment of each label within the rectangle to which it belongs.
labelFormat	java.text.Format	null	The format for labels associated with rectangles.
annotationClipping	boolean	true	Indicates whether annotations are clipped to fit in their corresponding rectangle.
annotationVisibility	non-negative integer or one of ALL, SMART, NONE	ALL	The visibility mode for annotations.
legendLabelFormat	java.text.Format	null	The format for labels associated with colors in the legend.

Integrating a chart customizer into your application

Describes how to create a chart customizer that can be integrated into your application.

In this section

Creating a chart customizer

It may happen that you need to modify the appearance of your chart at run time. To do this, you can adopt one of the following solutions:

- Write Java™ code to manipulate the chart renderers directly, as described in *Handling Chart Renderers*.
- Load different predefined style sheets.
- Use chart customizers, that is, interactive GUI components that allow you to handle the parameters of the chart appearance.

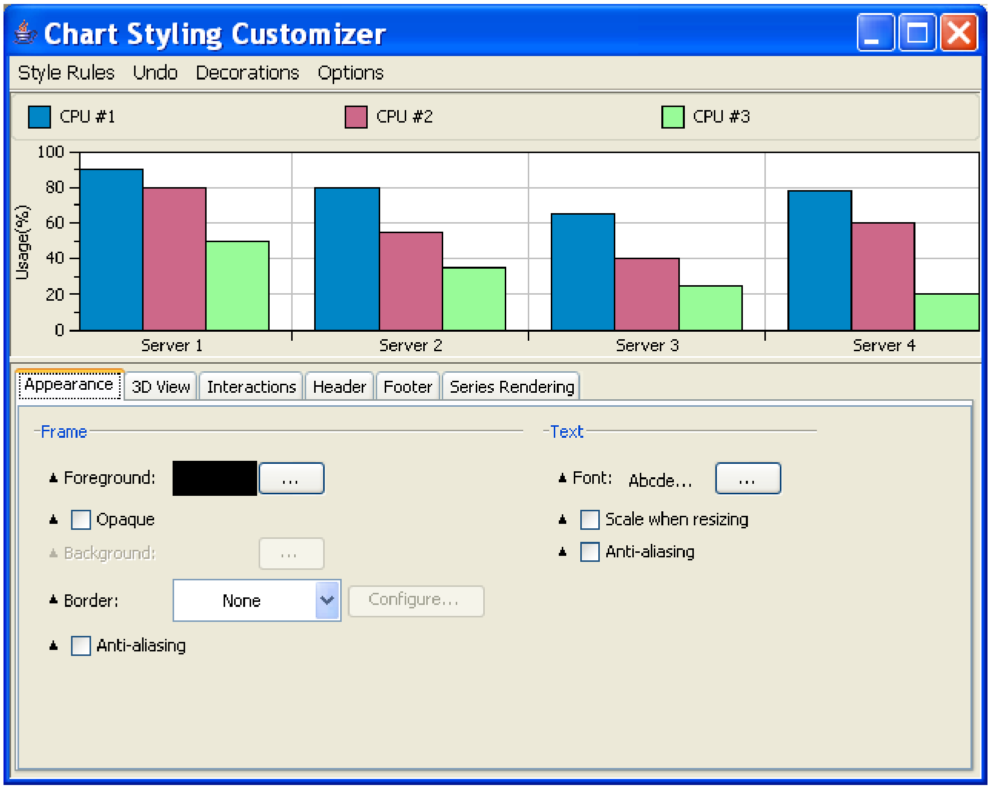


chart customizer

IBM® ILOG® JViews Charts provides predefined chart customizers that can be customized and easily integrated into a JViews Charts application. These chart customizers are based on the CSS styling technology of the chart. Therefore, it is possible to create an application that:

- loads one or multiple style sheets to customize the chart,
- fine-tunes the customization after styling by using the interactive chart customizers.

The complete source code of this example can be found in **`<installdir>/jviews-charts86/samples/customizer/src/ChartCustomizerExample.java`**.

To create a chart customizer:

1. Import the following packages to create a chart:
   a. import `ilog.views.chart.*`
   b. import `ilog.views.chart.data.xml.*`
2. Import the packages that contain the customizer framework for JViews Charts:
   a. import `ilog.views.chart.customizer.*`
   b. import `ilog.views.chart.customizer.swing.*`
3. Create the chart and load your style sheet:

```
IlvChart chart = new IlvChart();
try {
chart.setDataSource(...);
chart.setStyleSheets(...);
} catch (ilog.views.util.styling.IlvStylingException x) {
...
}
```

4. Create the chart customizer associated with this chart.

```
IlvChartCSSCustomizerPanel customizer =
new IlvChartCSSCustomizerPanel(chart);
```

5. Create the chart customizer panel.

   The chart customizer panel is a `JPanel`. It can be displayed in a dialog or in a frame or it can be added to a `JComponent`. In this example, you are going to display it in a split pane.

   Make the chart appear in the upper part of the split pane and the chart customizer in the lower part:

```
JSplitPane splitPane = new JSplitPane(JSplitPane.VERTICAL_SPLIT);
frame.getContentPane().add(splitPane, BorderLayout.CENTER);
splitPane.setTopComponent(chart);
splitPane.setBottomComponent(customizer);
```

6. Set the style rule that will be handled by the chart customizer:

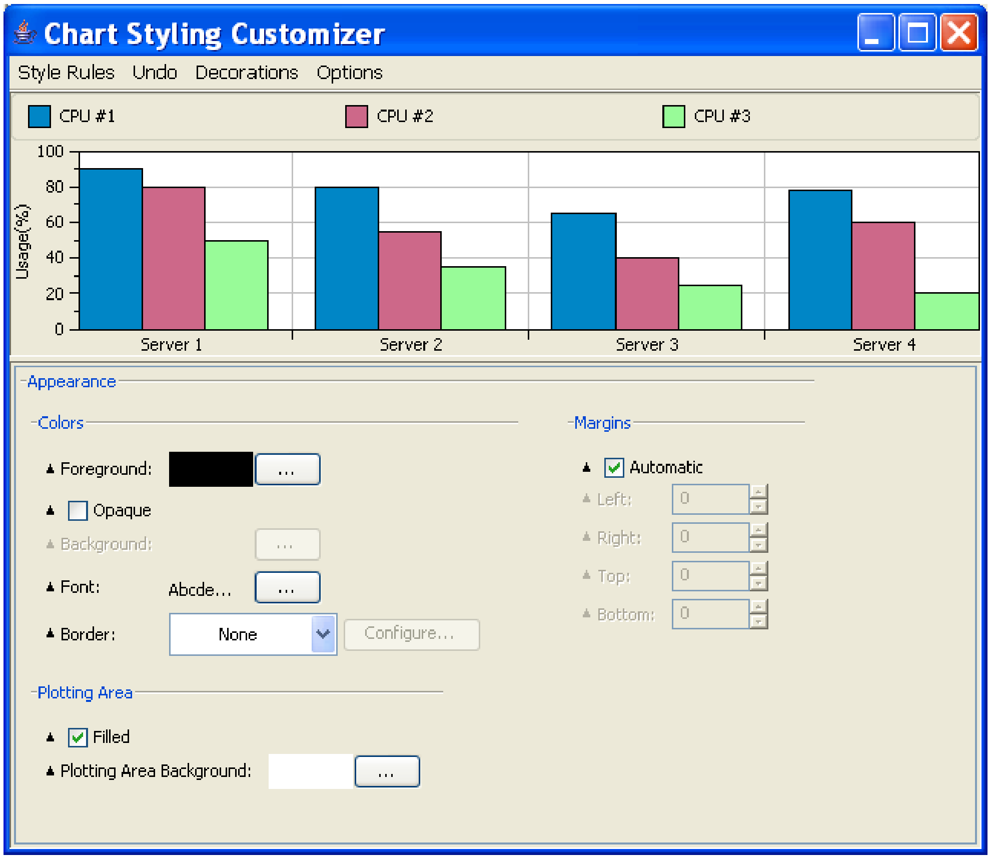
```
customizer.setRule("chartArea");
```

In this case, the chart customizer shows all the controls (buttons, fields, and boxes) that allow you to customize the chart area interactively. The parameter passed to the method `setRule` of the chart customizer is the selector of a style rule.

Possible values are: `chart`, `chartArea`, `chartLegend`, `chart3Dview`, `chartScale`, `chartGrid`, `series`, `point`.

Complex selectors such as `chartScale[axisIndex="-1"]` are also allowed, provided that the chart actually has such a scale.

You can find more details about selectors in *Styling the Chart Component*.



Foreground:

☐ Opaque

Background:

Font: Abcde...

Border: None

☑ Automatic

Left: 0

Right: 0

Top: 0

Bottom: 0

☑ Filled

Plotting Area Background:

These steps are sufficient if you want to reuse the predefined chart customizers. Otherwise, you can modify the chart customizers according to your needs. For more details, see *Customizing the chart customizer*.

Note: A chart customizer for treemaps is currently not available.

Main classes of the chart customizer

- `IlvChartCSSCustomizerPanel`

  This class is the panel that displays the chart customizer.

Occasionally, you may need to use the following classes:

- `IlvChartCSSAdapter`IlvChart

  This class is used to manipulate the CSS of the chart and to retrieve information about the CSS.

- `IlvRuleCustomizerLogic`

  When you set a style rule in the panel of the chart customizer, an `IlvRuleCustomizerLogic` is selected to handle this rule. The `IlvRuleCustomizerLogic` class contains the logic of what happens when you click a control. It knows which parameters are customizable for the corresponding style rule and how the CSS declaration values in this style rule need to look like.

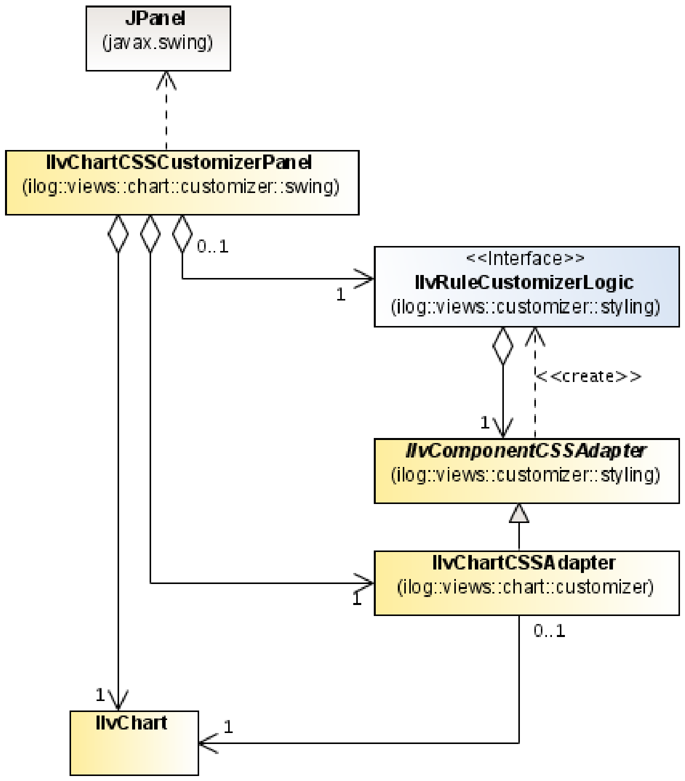

JPanel
(javax.swing)
IlvChartCSSCustomizerPanel
(ilog::views::chart::customizer::swing)
0..1
1
<<Interface>>
IlvRuleCustomizerLogic
(ilog::views::customizer::styling)
<<create>>
1
IlvComponentCSSAdapter
(ilog::views::customizer::styling)
IlvChartCSSAdapter
(ilog::views::chart::customizer)
1
0..1
1
IlvChart
1

Chart Customizer Main Classes

The chart customizer panel

The chart customizer panel (`IlvChartCSSCustomizerPanel` is a Swing panel that displays the GUI elements used to customize the chart.

The chart customizer panel is attached to a chart CSS adapter. The CSS adapter can be obtained by using the method `IlvChartCSSAdapter getAdapter()`.

To set the target of the chart customizer, you can use one of the following methods:

- `void setRule(String selector)`

  Receives the selector of a style rule as a string. It searches for an `IlvRuleCustomizerLogic` that matches the selector, sets this `IlvRuleCustomizerLogic` in the chart customizer panel and removes any existing `IlvRuleCustomizerLogic`.

- `void setRule(IlvCSSRule rule)`

  Receives the style rule directly. It searches for an `IlvRuleCustomizerLogic` that matches the style rule and sets this `IlvRuleCustomizerLogic` in the chart customizer panel.

- `void setRuleCustomizerLogic(IlvRuleCustomizerLogic ruleCustomizerLogic)`

  Receives the `IlvRuleCustomizerLogic` of a style rule and sets it in the chart customizer panel.

To retrieve the current `IlvRuleCustomizerLogic`, use the method `IlvRuleCustomizerLogic getRuleCustomizerLogic()`. The chart customizer panel updates its controls automatically when the parameters of the chart change.

It is rarely necessary to update the panel explicitly. If you need to do so, use one of the following methods:

- `void update()`

- `void update(boolean updateValue, boolean updateEnabled, boolean updateVisible)`

  This method allows you to specify whether one of the following elements should be updated:

  - values of the parameters displayed in the chart customizer panel
  - enabled state of the controls
  - visibility of the controls

The Chart CSS adapter

The chart CSS adapter (`IlvChartCSSAdapter` allows you to retrieve information about the CSS of the chart component. The CSS of a chart component can be very complex and can contain many auxiliary style rules. To simplify the situation, only the main style rules should be passed to the panel (such as `chartArea`, `chartLegend`, `chart3Dview`), since all tiny auxiliary style rules are handled automatically. Basically, the chart CSS adapter provides information about the selectors that can be passed to the method:

```
customizerPanel.setRule(selector)
```

To retrieve all possible selectors, you can use the following method:

```
String[] getCustomizableRuleSelectors()
```

The resulting string array contains all selectors that are currently active in the chart.

If you have an object that is part of the chart, for example an `IlvGrid` object, which can be the x-grid or the y-grid object, you can query which selector fits to this object. The method `String getSingleObjectCustomizableRuleSelector(Object object)` returns the corresponding selector.

Decoration rules specify the image decorations or the data indicators of the chart. Since an arbitrary number of decorations can be added to the chart, it is often necessary to retrieve in particular the selectors of the decoration rules of the chart. You can do this by using the method:

```
String[] getDecorationRuleSelectors()
```

Furthermore, the chart CSS adapter allows you to retrieve the instances of `IlvRuleCustomizerLogic` that fit a particular selector, by using the method

```
IlvRuleCustomizerLogic getRuleCustomizerLogic(String selector)
```

The returned `IlvRuleCustomizerLogic` is initialized with the style rule that corresponds to the input selector.

The IlvRuleCustomizerLogic class

The instances of the class `IlvRuleCustomizerLogic` represent the logic of how a style rule can be customized.

When you call the method `customizerPanel.setRule(selector)`, the `IlvRuleCustomizerLogic` appropriate to this selector is set in the chart customizer panel.

The chart CSS adapter provides only a fixed set of `IlvRuleCustomizerLogic` classes, and each `IlvRuleCustomizerLogic` provides only a fixed set of chart parameters that can be manipulated.

The available `IlvRuleCustomizerLogic` classes allow you to control a large set of parameters suitable for common cases. All `IlvRuleCustomizerLogic` classes can be retrieved from the chart component adapter by using the following code:

```
IlvRuleCustomizerLogic[] getRuleCustomizerLogics()
```

The same `IlvRuleCustomizerLogic` is reused for multiple style rules of the same type. For example, the `IlvRuleCustomizerLogic` of an image decoration can be reused for all image decorations. To test whether a style rule fits an `IlvRuleCustomizerLogic`, you can use the following method:

```
boolean match(IlvCSSRule rule)
```

If an `IlvRuleCustomizerLogic` is activated (which happens automatically when setting the style rule of the customizer panel), the method `IlvCSSRule getMainRule()` returns the style rule that is currently handled by an instance of the `IlvRuleCustomizerLogic`. Furthermore, every `IlvRuleCustomizerLogic` fires property change events whenever the main style rule has changed. If you want to listen to these events, you can set a property change listener on the `IlvRuleCustomizerLogic` by using the following method:

```
void addPropertyChangeListener(PropertyChangeListener listener)
```

Undo manager into the chart customizer

The chart customizer provides no undo manager by itself. Usually, an application has already a central undo manager. The chart customizer can be easily integrated into a standard Swing undo manager (`javax.swing.undo.UndoManager`).

```
UndoManager undoManager = new UndoManager();
```

The chart customizer fires a `javax.swing.event.UndoableEditEvent` whenever a change that cannot be undone occurs in the chart customizer. You can listen to these events in the following way:

```
customizer.addUndoableEditListener(undoListener);
```

When the undo listener receives an event, it adds the corresponding undo edit to the undo manager and updates the `undoManager`:

```
private UndoableEditListener undoListener = new UndoableEditListener() {
  public void undoableEditHappened(UndoableEditEvent e) {
    undoManager.addEdit(e.getEdit());
    ... code that updates the buttons and menu items that trigger an undo/redo,

    for example, enable the undo button to indicate undo is now possible, or
update the tooltip of the undo button ...
  }
};
```

To trigger an undo, you can use the standard Swing mechanism, that is, implement a menu item or button that calls the method `undoManager.undo()` when an action is performed.

Customizing the chart customizer

If you want to define your chart customizer for a particular style rule in your own application, these are the possible customizations:

- It is possible to remove buttons, fields, and boxes from the chart customizer.
- It is possible to rearrange the existing buttons, fields, and boxes. For example, you can change the order of the GUI elements in the tabbed panes.
- It is not possible to create new buttons, fields, and boxes that control a completely different parameter. The chart customizer can control a well-defined set of JViews Charts parameters. Rendering parameters that are not part of this set cannot be handled.

When you select a style rule for the chart customizer by using the method `customizer.setRule("chartArea")`, the panel of the chart customizer displays the controls that are needed to customize this style rule.

This is controlled by an XML file that is associated with the corresponding `IlvRuleCustomizerLogic`. You can change this XML file to modify the controls that appear in the panel.

Examples of XML files are located in the directory `<installdir>/jviews-charts86/codefragments/chart/customizer/data/examples`.

It is possible to adapt the XML file according to your needs. The format is described in *XML specification of the chart customizer*.

For internationalization purposes, each XML file is associated with a property file containing the text (labels, tooltips) that should appear in the chart customizer. This property file must be located in the same location as the XML configuration file.

For example, the `IlvChartAreaCustomizerForm.xml` file that describes the appearance of the chart customizer for the chart area and the `selectorIlvChartAreaCustomizerForm.properties` file that is the corresponding property file must be placed in the same directory.

To make your chart customizer use the XML and property specifications you have modified, it is possible to use one of the following methods:

- `customizer.setConfigFileBaseURL(myDirectoryURL);`

  This solution is applicable when you modify the contents of the XML or property specifications without altering the file names.

- `customizer.setRule("chartArea", myXMLConfigFileURL);`

  This solution forces you to maintain yourself the association between colors and file names, but allows you to use different configurations for the same selector. For example, you might use separate configuration files (and therefore also separate chart customizers) for the colors of the chart area and for the margins of the chart area, although both would be associated with the selector `chartArea`.

XML specification of the chart customizer

Example of XML File

```
<!-- Customizer Form for IlvChart.Area -->
 <form bundles="ChartAreaForm">
   <group title="AppearanceGroup">
     <group title="ColorsGroup">
       <property name="foreground"
        displayedName="foregroundDisplayedName"
        tooltip="foregroundTooltip"/>
       <property name="opaque"
        displayedName="opaqueDisplayedName"
        tooltip="opaqueTooltip"/>
       <property name="backgroundPaint"
        displayedName="backgroundDisplayedName"
        tooltip="backgroundTooltip">
        <enabled>
         <when property="opaque" value="true"/>
        </enabled>
       </property>
       <property name="border"
        displayedName="borderDisplayedName"
        tooltip="borderTooltip"/>
       <property name="font"
        displayedName="fontDisplayedName"
        tooltip="fontTooltip"/>
     </group>
   </group>
 </form>
```

The GUI elements are specified between `<form>` and `</form>` elements.

For internationalization purposes, you can pass a resource bundle to the `<form>` element.

The resource bundle is used to retrieve all texts (labels, tooltips) that should be displayed in the chart customizer. Furthermore, the specification consists of groups (`<group>`) and properties (`<property>`).

The `<group>` element specifies a section in the chart customizer. Usually, a group has a title that indicates the purpose of the section. Groups can be nested, that is, a group can contain other groups. Finally, a group consists of a list of properties. In the chart customizer, you can consider a group as a panel with a titled order that groups together a collection of controls (buttons, fields, and boxes).

The `<property>` element specifies a property of the corresponding chart renderer that should be customized. For example, to customize the foreground color of the chart area, specify `<property name="foreground"/>`.

There is only a limited set of properties for each `IlvRuleCustomizerLogic`, you cannot specify arbitrary properties that do not exist. In the chart customizer, the `<property>` element produces a control element (button, text field, checkbox, listbox, and so on) that allows you to edit this property. For example, the specification `<property name="foreground"/>` in the example above creates a color editor control that allows you to change the foreground color of a chart area. This control appears with a label in the chart customizer. You can specify the label and the tooltip of the control as follows:

```
<property name="foreground" displayedName="foregroundDisplayedName"
tooltip="foregroundTooltip"/>
```

The keys `foregroundDisplayedName` and `foregroundTooltip` are looked up in the resource bundle of the chart customizer form. If the keys are not found, the string `foregroundDisplayedName` is used as label, and the string `foregroundTooltip` is used as tooltip.

It might happen that a control of a property should only be enabled if another property has a certain value. This is possible with subspecifications in the properties. For example, the following specification means that the control of the `backgroundPaint` should only be enabled if the `opaque` property has the value `true`.

```
<property name="backgroundPaint">
 <enabled>
  <when property="opaque" value="true"/>
 </enabled>
</property>
```

XML Tags and Attributes Available in the Specification of a chart customizer describes all XML tags that are available in the specification of a chart customizer.

XML Tags and Attributes Available in the Specification of a chart customizer

Element	Attribute	Description
`form`		Root element. Defines the chart customizer.
	`bundles`	Optional, but recommended. Defines a list of resource bundles that provide string resources for the `form`. The items of the list are separated by a comma and are relative pathnames. These pathnames are resolved into full pathnames using the URL of the `form` as a reference or the URL resolvers added to the application.
	`title`	Optional, but recommended. Defines a title for the entire chart customizer.
`group`		Parent: form or group. Defines a group of any combinations of property, help and group elements.
	`title`	Optional, but recommended.

Element	Attribute	Description
		Defines the title of the group. Typically, this will be shown either as the name of a tab in a tabbed pane or in the title border of a `BorderedPanel`.
	`awtLayoutManager`	Optional. Defines the fully-qualified class name of the layout manager to be used for this group instead of the default one in AWT/Swing chart customizers. It must be a public class (accessible in the classpath) implementing `java.awt.LayoutManager` and providing a default constructor or a constructor taking a `java.awt.Container` argument (the target panel).
	`dummy`	Optional. If the value is `true`, the GUI does not show any border around the group nor a title.
	`overlaid`	Optional. If the value is `true`, the elements contained in this group (subgroups or properties) are displayed in overlaid mode. Typically, this attribute is set when you want only one set of elements to be visible, depending on certain conditions. See also the `visible` element.
`property`		Parent: form or group. This element defines one property (usually corresponding to a JavaBean property, but not necessarily) of the customized target.
	`name`	Required. Defines the name of the property. Typically, this is the name of the JavaBean property, but in complex situations it can be a different identifier. This must not be internationalized.
	`targetId`	Optional. Defines an identifier for the target to which the property belongs. Typically, this is used in complex chart customizers where subtargets have their own properties. This must not be internationalized.
	`displayedName`	Optional, but recommended. Defines the string that appears as a label near the GUI component and allows you to customize the property.
	`tooltip`	Optional, but recommended. Defines the string that appears as a tooltip over the GUI component and allows you to customize the property.
	`minValue`	Optional for numeric properties, ignored for non-numeric. Defines the minimum value that the user is allowed to enter. If both

Element	Attribute	Description
		`minValue` and `maxValue` are specified, the GUI will typically show a slider in addition to the text field and the spinner.
	`maxValue`	Optional for numeric properties, ignored for non-numeric. Defines the maximum value that the user is allowed to enter. If both `minValue` and `maxValue` are specified, the GUI will typically show a slider in addition to the text field and the spinner.
	`incValue`	Optional for numeric properties, ignored for non-numeric. Defines the increment for the spinner usually associated with the text field.
	`choices`	Optional for tagged properties, ignored otherwise. Defines the list of values to be presented to the user, typically in a combo box. Its value must be a comma-separated list of strings (for example, the names of the combo items) and values. For example: `MyResource.FirstItem.Name=3,MyResource.SecondItem.Name=4`. Note that, by default, this is retrieved from the property descriptor. You only need the attribute if you want to override this information.
	`displayedNullValue`	Optional for properties of non-primitive type, ignored otherwise. Defines the string representation of the `null` value of the property. For example, it can be "null", "(null)", "(none)", "(unspecified)", or an empty string. If the attribute is not specified, the text is retrieved from `propertyEditor.getAsText`.
	`index`	Optional for indexed properties, ignored otherwise. It is used when the chart customizer needs to allow the customization of a given element of the array of an indexed property.
	`preferredWidth`	Optional. Defines a preferred width of the GUI control used for the property, to be used instead of the default one. You can specify either the value directly, or a resource key.
	`preferredHeight`	Optional. Defines a preferred height of the GUI control used for the property, to be used instead of the default one. You can specify either the value directly, or a resource key.
	`propertyEditor`	Optional. Defines the fully-qualified class name of the property editor to be used for this property instead of the default one. It must be a public class (accessible in the classpath) implementing `java.beans.PropertyEditor` and providing a default constructor.
`help`		Parent: form, group, or property.

Element	Attribute	Description
		Defines help contents for its parent element. This can be either a text area directly embedded in the chart customizer, or a button allowing to open a standalone help window to display an HTML file.
	`text`	Optional. Defines the string to be shown embedded in the chart customizer, typically as a text area.
	`title`	Optional. Defines the string to be shown as title of the help window instead of the default title. Ignored if the `text` attribute is present (that is, for help text in a text area embedded in the chart customizer).
	`page`	Optional. Defines the relative URL of an HTML help page. Ignored if the `text` attribute is present.
	`buttonIcon`	Optional. Defines the relative URL of the image to be shown instead of the default icon used to open a standalone help window. Ignored if the `text` attribute is present.
	`tooltip`	Optional, but recommended. Defines the string that is shown as tooltip over the GUI component (text area or button for opening a standalone help window.)
`enabled`		Parent: `group` or `property`. Defines a condition for enabling/disabling a group or a property depending on the value of one or several properties of the target, or depending on some external conditions. It must have the following subelements: `when`, `or`, `and`, or `not`.
`visible`		Parent: `group` or `property`. Defines a condition for controlling the visibility of a group or of a property depending on the value of one or several properties of the target, or depending on some external conditions. It must have the following subelements: `when`, `or`, `and`, or `not`.
`when`		Parent: `enabled`, `visible`, `or`, `and`, or `not`. Defines the rules for enabling/disabling or changing the visibility of a group or a property depending on the value of one or several properties of the target, or depending on some external conditions.

Element	Attribute	Description
		It must have either the attribute `condition`, or the attribute `property` associated with `value`.
	`condition`	Required if `property` associated with `value` are not present. Defines the identifier of a condition that is implemented in Java code.
	`property`	Required if `condition` is not present. Defines the name of the property. The enabled/disabled state or the visibility depends on the value of this property. For properties where a `targetId` is not specified, use the `name` attribute. Otherwise, use the `targetId` attribute.
	`value`	Required if `condition` is not present. Defines the value of the property specified by the `property` attribute.
`and`		Parent: `enabled`, `visible`, `and`, `or`, or `not`. Defines an `and` logical operation for its subelements. It must have the following subelements: `when`, `or`, `and`, or `not`.
`or`		Parent: `enabled`, `visible`, `and`, `or`, or `not`. Defines an `or` logical operation for its subelements. It must have the following subelements: `when`, `or`, `and`, or `not`.
`not`		Parent: `enabled`, `visible`, `and`, `or`, or `not` Defines a `not` logical operation for its subelements. It must have the following subelements: `when`, `or`, `and`, or `not`.

Property Editors

The chart customizer uses special property editors for some types, such as `Boolean`, `Integer`, `Double`, `Date`, `Point`, `Font`, `Color`. These may collide with the look and feel of other parts of your application.

By default, when you use a chart customizer, IBM® ILOG® JViews Charts installs these property editors in the global property editor registry, in the class `java.beans.PropertyEditorManager`. As a consequence, other parts of your application will get the IBM® ILOG® JViews look and feel.

If you want to prevent this, you can set the system property `ilog.propagatesPropertyEditors` to `false`. By doing this, your application will not be affected by the IBM® ILOG® JViews look and feel, but the chart customizer will keep using this look and feel. To set the system property `ilog.propagatesPropertyEditors` to `false` you can either use the command line option

```
-Dilog.propagatesPropertyEditors=false
```

or call `System.setProperty("ilog.propagatesPropertyEditors", "false")` before instantiating any chart customizer.

Using Load-On-Demand

Describes how the Load-On-Demand mechanism works and how you can interact with it.

In this section

Framework structure
Describes the set of interfaces and classes that define and implement the load-on-demand mechanism.

The tile controller
Describes the tile controller mechanism.

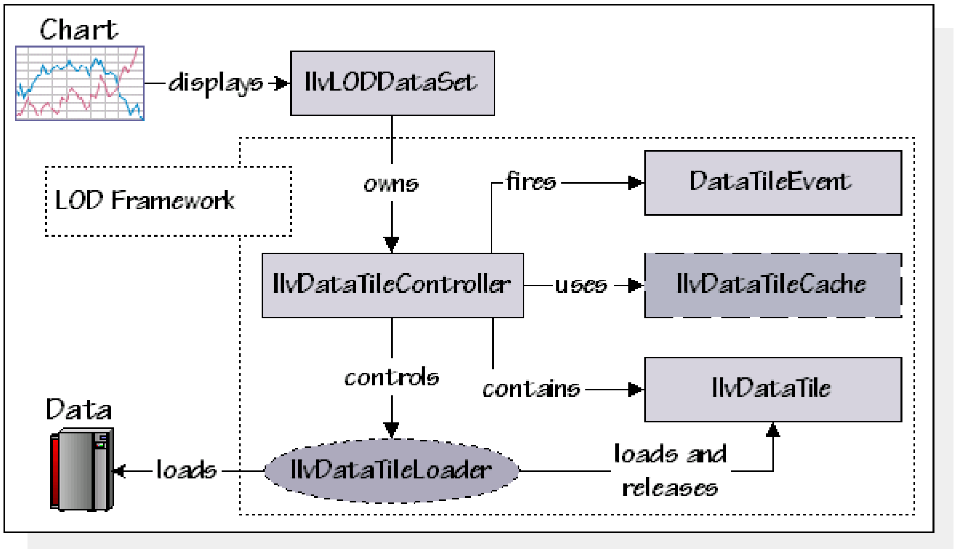
The tile cache
Describes the strategy for caching and releasing tiles.

How to listen to events
Describes how to listen to the events generated by the load-on-demand mechanism.

How to use LOD with your data
Explains the tile loader implementation available in the DataTileLoader.java file.

Framework structure

The load-on-demand mechanism integrates seamlessly into the chart data model through the `IlvLODDataSet` class, which can be connected to a chart as any regular data set. `IlvLODDataSet` actually acts as a bridge between the data model and the load-on-demand mechanism, which is defined and implemented by a set of interfaces and classes as described in the following figure:



The Load-On-Demand Framework

All these interfaces and classes are located in the `ilog.views.chart.data.lod` package.

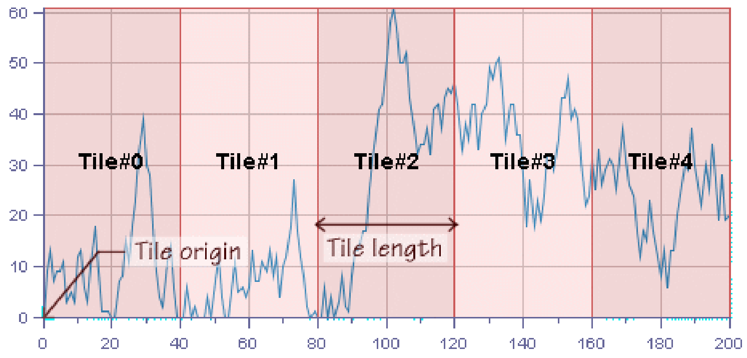
The tile controller

A tile controller is an instance of the `IlvDataTileController` class, which implements the load-on-demand strategy. To do so, the tile controller divides the *x*-axis into a number of equal intervals called *tiles*. This operation is performed according to two parameters specified in the controller constructor:

- The *tile origin*, which represents the minimum *x*-value of the first tile.
- The *tile length*, which represents the extent of a tile along the *x*-axis.

Each tile is represented by an instance of the `IlvDataTile` class. It is identified by a specific integer that corresponds to its location relative to the tile origin.

Tiling of the Visible Range of a Chart illustrates the tiling of the visible range of a chart:



Tiling of the Visible Range of a Chart

Each time a tile becomes visible due to a change in the visible range of an *x*-axis, the tile controller notifies the tile loader to fetch the data attached to this tile. Each loaded tile is locked by the object that triggered the loading request, and unlocked by the same object when data is no longer used. This ensures that the data actually displayed is kept in memory, while the data that is no longer in the visible range of the chart is put in a cache for potential reuse.

Each load-on-demand data set owns a tile controller. When these data sets are connected to a chart, the load-on-demand mechanism is activated so that the changes in the visible range of the *x*-axis of the considered chart are translated into loading requests (it is said that load-on-demand is *event-driven*). This translation is performed by the `axisRangeChanged (ilog.views.chart.IlvAxis)` method, whose default implementation is to request the loading of tiles that intersect the new visible range of the corresponding axis. You can override this method to pre-load neighboring data.

For example, a data set may expand the new visible range by a given factor before sending the loading request, thus anticipating further scrolling or zooming.

Although tile loading is event-driven, you can also explicitly use the `requestLoading(ilog.views.chart.IlvDataInterval, java.lang.Object)` method to load data for a given *x*-interval. In the same way, the `lock(java.lang.Object)` and `unlock(java.lang.Object)` methods let you manually lock or unlock tiles to prevent them from being released.

The `IlvDataTileController` is the core class of the load-on-demand mechanism, while `IlvLODDataSet` is the bridge to the chart data model. Each load-on-demand data set must be created with its tile controller, as shown in the following code extract:

```
IlvDataTileLoader loader = ...;
IlvDataTileCache  cache  = ...;
double tileOrigin        = 0;
double tileLength        = 200;
int    tileCapacity      = 200;

IlvDataTileController ctrl
= new IlvDataTileController(loader, cache, tileOrigin,
                                  tileLength, tileCapacity);

IlvLODDataSet dataSet = new IlvLODDataSet(ctrl);
```

The *tileLength* parameter represents the extent of a tile along the *x*-axis.

The *tileOrigin* parameter represents the minimum *x*-value of the first tile.

The *tileCapacity* parameter represents the maximum number of data points a tile can hold. The controller uses this capacity to provide indexed access to data points through the data set.

The tile cache

The abstract class `IlvDataTileCache` specifies the methods that can be used to implement a strategy for caching and releasing tiles. The API of this class is based on a callback model, in which the controller invokes some methods when specific events occur. The following methods are concerned:

♦ `tileCached(ilog.views.chart.data.lod.IlvDataTile)`

  Called when the controller decides to put a tile in the cache.

♦ `tileRetrieved(ilog.views.chart.data.lod.IlvDataTile)`

  Called when the controller wants to recover a tile.

♦ `tileAboutToLoad(ilog.views.chart.data.lod.IlvDataTile)`

  Called when the controller has requested the loading of a tile.

The JViews Charts library provides a default cache implementation in the `IlvDefaultDataTileCache` class. This class implements a memory-sensitive cache where tiles are released whenever the application requires memory to be freed. You can control the load of this cache by specifying minimum and maximum capacities. For more information on the meaning of these capacities, please refer to the Reference Manual.

Note: The `IlvDefaultDataTileCache` class tries to release tiles in a LRU (Least Recently Used) order so that it first unloads the tiles that have been visited the least recently. However, since the memory-sensitive part is based on an external mechanism, this order may not always be verified.

Although this default cache implementation offers a good and ready-to-use solution in general cases, you may still wish to subclass `IlvDataTileCache` and implement more efficient cache strategies that take into account application-specific criteria for choosing when and which tiles should be kept or released.

The tile loader

Just as `IlvLODDataSet` is the bridge between the load-on-demand mechanism and the chart data model, the tile loader performs the connection between this mechanism and the actual data. The `IlvDataTileLoader` interface specifies the methods needed for this connection. The tile controller uses the tile loader as it uses the tile cache: methods are called when specific events occur. The `IlvDataTileLoader` interface includes the following methods:

♦ `load(ilog.views.chart.data.lod.IlvDataTile)`

  Called to load the contents of a new tile.

♦ `release(ilog.views.chart.data.lod.IlvDataTile)`

  Called to release a tile.

♦ `getXRange()` and `getYRange()`

Returns the limits of the *x*- and *y*-values provided by the loader.

The process of loading a tile is usually divided into three main steps:

1. Fetch the data corresponding to the tile.

   This operation is based on the *x*-range of the corresponding tile, which can be accessed with the `getRange()` method.

2. Fill the tile with the data.

   This operation is performed by means of the `setData(ilog.views.chart.IlvDoublePoints)` method.

3. Notify the tile loader that loading is complete.

   This last operation is performed by calling the `loadComplete()` method.

The load-on-demand framework does not contain any concrete implementation of the `IlvDataTileLoader` interface, as such implementations depend on the origin of the data.

How to listen to events

The load-on-demand mechanism generates events that describe changes in the status of data tiles. Such events are sent by the tile controller and are implemented by the `DataTileEvent` class. To be notified of these events, you can add a `DataTileListener` to the corresponding controller by using the following code:

```
controller.addDataTileListener(new DataTileListener() {
  public void dataTileChanged(DataTileEvent evt)
  {
    System.out.println(evt);
  }
});
```

In this example, all the changes to the data tiles are printed to the standard output.

The generated events occur after the following actions:

- The loading of a tile starts or completes.
- A tile is added to or retrieved from the cache.
- A tile is released.
- An error occurred while loading a tile.

For a list of the event types related to each of these actions, please refer to the Reference Manual.

The `isAdjusting()` method allows you to know whether an event is part of a series of several other events (for example, changing the visible range of a chart can trigger several loading events). This mechanism allows you to postpone event handling until the sequence ends. When this happens, an `ADJUSTMENT_END` event is sent.

The file **`CachedTileListener.java`** of the load-on-demand example shows you how listeners can be used to monitor changes in the cached status of data tiles.

How to use LOD with your data

The data connection is performed by means of the `IlvDataTileLoader` interface.

The complete source file can be found in **`<installdir>/jviews-charts86/samples/lod/src/lod/DataTileLoader.java`**.

The tile loader that you are going to consider must be in accordance with the following specifications:

♦ Data is stored in a binary file. This file contains 32-bit integer records that represent the *y*-values of the data points. The *x*-value of a data point is equal to its index in the file.

♦ The loading of the tiles must be performed in a separate thread.

♦ The loading can be interrupted.

First you need to study the data members of the loader class.

```
public class DataTileLoader implements IlvDataTileLoader
{
    private IlvDataInterval xRange    = new IlvDataInterval();
    private IlvDataInterval yRange    = new IlvDataInterval();
    private LinkedList      tileQueue = new LinkedList();
    private File            file;
    private LoadThread      loadThread;
    private long            lag;
```

♦ The `xRange` and `yRange` members are used to store the limits of the values provided by the loader. These intervals are returned by the `DataTileLoader.getXRange` and `DataTileLoader.getYRange` methods.

♦ The `tileQueue` list is used as a queue to hold tiles whose loading has been requested.

♦ The `file` attribute points to the file from which our loader will retrieve the data.

♦ The `loadThread` member references the thread that the loader will use to load the data.

♦ The `lag` attribute is used for demonstration purposes to simulate a lag in data loading (the lag will correspond to a duration in milliseconds during which the loading thread will sleep.)

The constructor of our loader has the following implementation:

```
public DataTileLoader(File file)
{
  parseDataFile(file);
  this.file = file;
  loadThread = new LoadThread();
  loadThread.start();
}
```

The `LoadThread.parseDataFile` method simply reads the data file contents to initialize the `xRange` and `yRange` members. The constructor also spawns the loading thread.

```
private class LoadThread extends Thread {
  RandomAccessFile rf = null;
  IlvDataTile      tile;
  LoadThread()
  {
    super("TileLoadingThread");
  }
```

The `LoadThread` class defines two data members:

- ♦ `tile` corresponds to the tile being currently processed by the thread.
- ♦ `rf` represents the `RandomAccessFile` instance that is used to retrieve data from the loader data file.

The `run` method of this thread has the following implementation:

```
public void run()
{
  while (true) {
    try {
      rf = null;

      //==[1]==//
      tile = popTile();
      if (tile == null) continue;

      //==[2]==//
      rf = new RandomAccessFile(DataTileLoader.this.file, "r");

      //==[3]==//
      IlvDataInterval itv = tile.getRange();
      long start = (long) Math.floor(itv.getMin());
  if (start < 0) continue;
      rf.seek(start*4);
      int count = (int) itv.getLength()+1;
      IlvDataPoints dataPts = new IlvDataPoints(count);

      // ... Read the data from the file and fill dataPts.

      //==[4]==//
      tile.setData(dataPts);
      dataPts.dispose();
      tile.loadComplete();
      tile = null;

      //==[5]==//
      } catch (InterruptedException e) {
        // Can be because of cancelLoading.
      } catch (IOException e) {
        // Can be because of cancelLoading.
      } finally {
        // ... Close the file.
      }
 }
```

The `run` method of the thread consists of an infinite loop whose first instruction is to call the `popTile` method of the loader ([1]):

```
private synchronized IlvDataTile popTile()
{
  try {
    while(tileQueue.size() == 0)
      wait(); // Put loading thread in waiting state.
  } catch (InterruptedException x) {
    return null;
  }
  return (IlvDataTile) tileQueue.removeFirst();
}
```

Calling this method returns the first tile in the loading queue, or puts the thread into a waiting state if no tile is available. Once the thread has retrieved the tile, a `RandomAccessFile` object is created ([2]), and the range of the tile is used to read the corresponding data from the file ([3]). Since the *x*-value of a data point corresponds to the record number, you can easily determine the offset of the first relevant *y*-value within the file. Note that if *x*-values were stored in the file, you should have used a more sophisticated algorithm such as a binary search.

Finally, the read data is used to fill the tile, and the `loadComplete()` method is called to notify that the loading is complete ([4]). This whole procedure can be interrupted with the `LoadingThread.cancelLoading` method, which interrupts the thread and closes the descriptor on the data file. Both cases are handled in the `run` method so that it actually cancels the loading of the current tile and resumes the loading process, without having to restart the thread ([5]).

Now that you have seen how the loading thread works, you will look at the implementation of the main methods of the `IlvDataTileLoader` interface, namely the `load(ilog.views.chart.data.lod.IlvDataTile)` and `release(ilog.views.chart.data.lod.IlvDataTile)` methods.

```
public synchronized void load(IlvDataTile tile) throws Exception
{
  pushTile(tile);
}

private synchronized void pushTile(IlvDataTile tile)
{
  tileQueue.addLast(tile);
  if (tileQueue.size() == 1)
    notify(); // Wake up loading thread.
}
```

The `load(ilog.views.chart.data.lod.IlvDataTile)` method simply appends the tile at the end of the loading queue, and wakes up the loading thread if no tile was previously available for processing (as you have seen in the description of the `popTile` method the loading thread remains in a waiting state until a loading request is issued).

```
public synchronized void release(IlvDataTile tile)
{
  if (loadThread.tile == tile) {
    // The tile is currently processed
    // by the loading thread: cancel.
    loadThread.cancelLoading();
  } else {
    // Just remove the tile from the queue.
    tileQueue.remove(tile);
  }
  tile.setData(null);
}
```

The `release(ilog.views.chart.data.lod.IlvDataTile)` method cancels the loading request for a given tile, which can be either in the queue or currently being processed by the thread.

In conclusion, it is possible to distinguish two main parts in our tile loader implementation:

♦ The actual data access, which in our case consists of randomly accessing the contents of a file.

♦ The handling of loading requests and the threading technique, which provide an asynchronous and interruptible mechanism.

The `DataTileLoader` class could be reused with a different data source. For example, data could be retrieved with database queries instead of file access. Also, the implementation of the loading queue could be improved so that it takes into account priorities, instead of following a First-In-First-Out policy.

This implementation highlights two main concerns that should be kept in mind while designing a tile loader:

♦ Since load-on-demand is event-driven, the loading of tiles must be threaded so that it does not hinder the remainder of the application.

♦ In order to be efficient, a tile loader must be able to quickly retrieve the data associated with a given tile. This usually means it is possible either to have random access to data or to issue requests that are processed by smart back-end programs, such as database servers.

Using JViews Charts JavaBeans

Describes the IlvChart bean, the data source beans and how to use the JavaBeans™ with an IDE.

In this section

The IlvChart bean

Describes the properties defined by the `IlvChart` bean, in addition to the inherited `JComponent` properties.

In this section

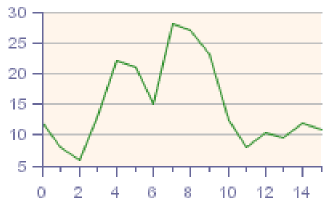
General properties

Property	Description	Default value
xGridVisible	Controls the abscissa grid visibility	true
yGridVisible	Controls the ordinate grid visibility	true
header	The text to display on top of the chart area	empty
footer	The text to display at the bottom of the chart area	empty
antiAliasing	Controls the antialiasing mode inside the chart area	false
antiAliasingText	Controls the antialiasing mode of text inside the chart area	false
chartAreaBorder	The border to set on the chart area component	null
plotAreaBackground	The background color of the plot area	white

Grid visibility

The `xGridVisible` and `yGridVisible` properties indicate whether the abscissa grid or the ordinate grid should be drawn. A grid is drawn below the graphical representations of the data, and match the major steps location of their associated scale. By default, both grids are visible.

Grid Visibility shows a chart with a hidden abscissa grid, and a visible ordinate grid:

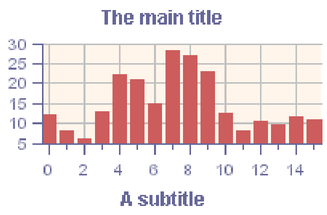


Grid Visibility

Header and footer text

The `header` and `footer` properties indicate that a text can be displayed on top of the chart area, and at the bottom of the chart area, respectively. This string can be expressed as simple text or as HTML text. By default, these properties are set to an empty string.

Header and Footer shows a chart with the `header` property set to “The main Title” and the `footer` property set to “A subtitle”:



Header and Footer

Antialiasing modes

The antialiasing mechanism allows you to have a nice display rendering by reducing stair effects on the display area. The `IlvChart` bean supports two antialiasing modes by means of two properties:

- `antiAliasing`

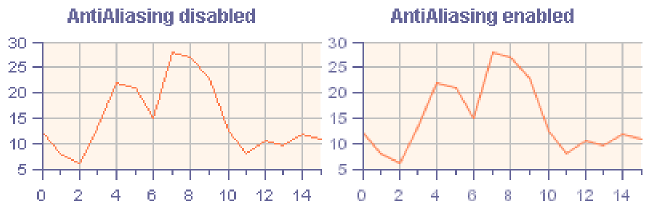
  Controls the rendering quality of all the drawings performed within the chart area, except text drawings.

- `antiAliasingText`

  Controls the rendering quality of all the text drawings performed within the chart area.

Since the antialiasing mechanism may be a time-consuming process, both modes are disabled by default.

Antialiasing shows two charts: the first one has the `antiAliasing` property set to `false`, the second one has the `antiAliasing` property set to `true`:

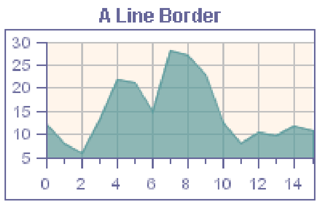


Antialiasing

Chart area border

The `IlvChart` bean allows you to apply a `javax.swing.Border` instance of the chart area component by means of the `chartAreaBorder` property. By default, no border is set.

A Line Border Instance shows a chart area with a `LineBorder` instance:



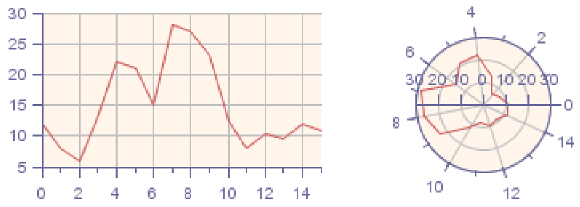
A Line Border Instance

Plotting area background

The plotting area is defined as the region where data is projected and displayed. The bounds of this rectangle are computed according to the drawing rectangle (the chart area bounds minus its internal insets) and the chart area internal margins.

The `plotAreaBackground` property describes the background color of this plotting rectangle, and is set to white by default.

Plotting Area Background shows two charts with a plotting area background color set to "255, 245, 235":



Plotting Area Background

Legend properties

Property	Description	Default value
legendVisible	Controls the legend visibility	false
legendPosition	Controls the legend position within the chart	North

Legend visibility

The `IlvChart` bean may be associated with a legend to display information about the drawn data. When this legend is enabled, all the graphical representations displayed within the chart have a corresponding legend item.

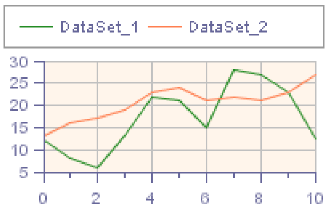
The legend is activated by means of the `legendVisibility` property, and is disabled by default.

Legend position

The chart legend is displayed at predefined positions within the chart around the chart area. These anchor positions are defined relative to the chart area, and the possible values are: `North`, `North West`, `West`, `South West`, `South`, `South East`, `East`, and `North East`.

The legend anchor position is defined by the `legendPosition` property, and is set to `North` by default.

Legend shows a chart with a `legendVisible` property set to `true` and the `legendPosition` property set to `North`:



Legend

Axis properties

Property	Description	Default Value
projectorReversed	Controls the projector configuration	false
xAxisReversed	Controls the abscissa axis orientation	false
yAxisReversed	Controls the ordinate axis orientation	false

Axis configuration

The `IlvChart` beans allow you to customize the chart axis configuration by means of three different properties that work together to define the coordinate system configuration:

- `projectorReversed`

  Allows you to change the meaning of the abscissa and ordinate coordinates of a point. For example, a reversed Cartesian projector will project *x*-data values along the *y*-axis of the screen.

- `xAxisReversed`

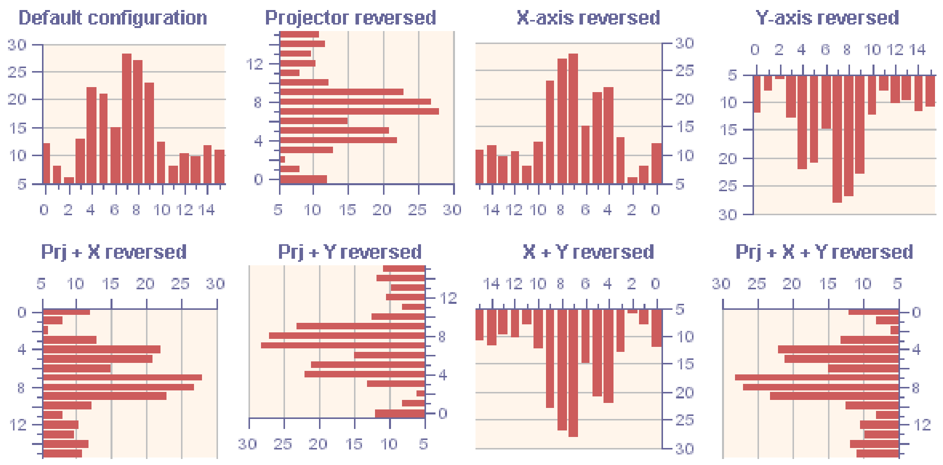
  Allows you to change the orientation of the *x*-axis.

- `yAxisReversed`

  Allows you to change the orientation of the *y*-axis.

The combination of these three properties determines the full configuration of the chart coordinate system.

Axis shows the possible configurations and their results:



Axis

Scale properties

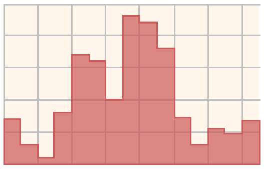
Property	Description	Default value
xScaleVisible	Controls the abscissa scale visibility	true
yScaleVisible	Controls the ordinate scale visibility	true
xScaleTitle	Defines the abscissa scale title	none
yScaleTitle	Defines the ordinate scale title	none
xScaleTitleRotation	Defines the abscissa scale title rotation	0
yScaleTitleRotation	Defines the ordinate scale title rotation	0

Scales visibility

A scale is defined as the graphical representation of an axis. Scale objects are directly handled by the `IlvChart` bean that creates one scale per axis.

The visibility of the chart scales is controlled by means of the `xScaleVisible` and `yScaleVisible` properties. By default, both *x*- and *y*-scales are visible.

Scales Visibility shows a chart with the hidden scales:



Scales Visibility

Scales title

The `IlvChart` bean allows you to associate a title with each scale of the chart. A scale title is defined by the following elements:

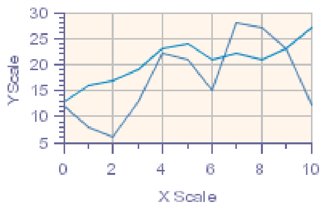
- the text

  Displayed through the `xScaleTitle` and `yScaleTitle` properties that are initialized to an empty string by default.

- a rotation angle

  Defined through the `xScaleTitleRotation` and `yScaleTitleRotation` properties that are initialized to 0.

Scales Title shows a chart with the `xScaleTitle` property set to “X Scale”, the `yScaleTitle` set to “Y Scale”, and the `yScaleTitleRotation` property set to -90:



Scales Title

Graphical representation properties

Property	Description	Default value
type	Defines the chart type	Cartesian
renderingType	Defines the type of the graphical representation of a data model	Polyline

To determine how a data model is rendered on the screen, the `IlvChart` beans define the following two properties:

♦ `type`

Determines how data is projected (using either a Cartesian or polar projection). The possible values are: Cartesian, Polar, Radar, and Pie.

Chart Types shows the possible values of the `type` property:

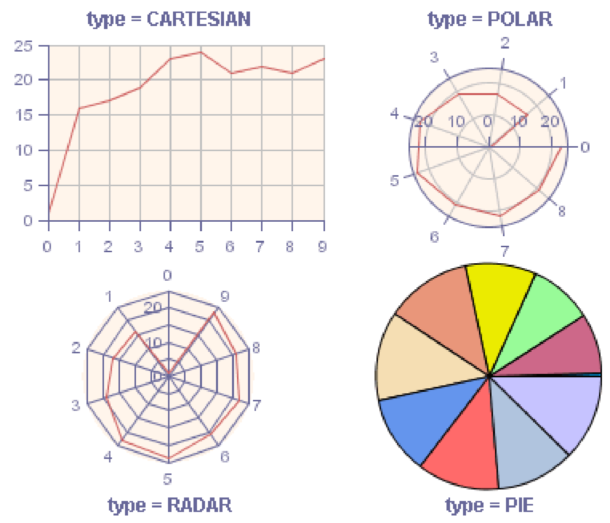


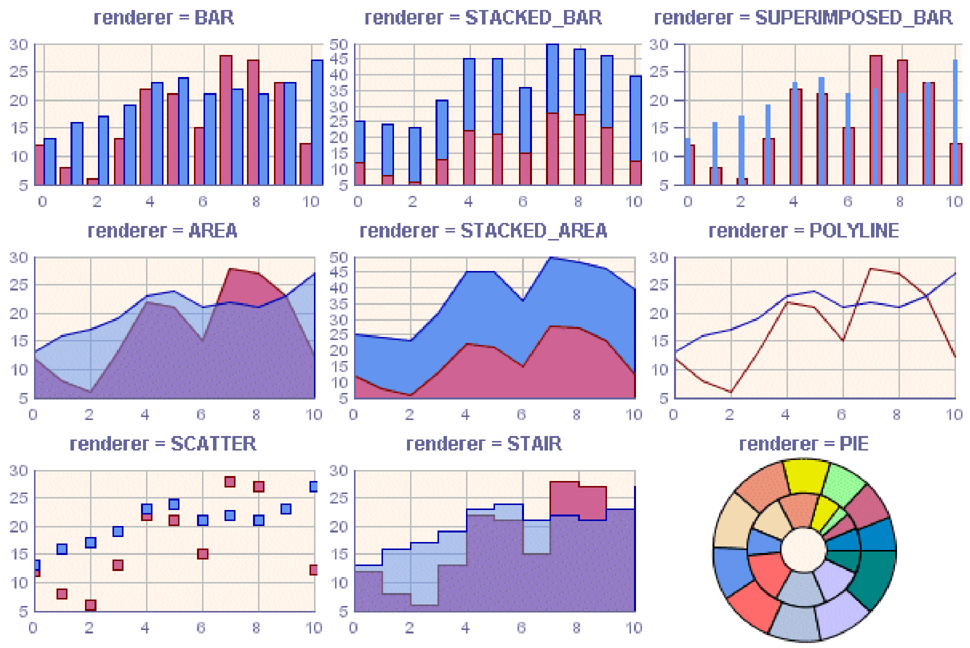
Chart Types

♦ `renderingType`

defines the graphical representation of a data model (as polylines, bars, stairs, and so on.). The possible values are: Bar, Stacked Bar, SuperimposedBar, Area, Stacked Area, Polyline, Scatter, Stair, and Pie.

Note: The Pie Renderer type is essentially used with a Pie chart.

Renderers shows the Cartesian charts with the different graphical representations:


renderer = BAR
renderer = STACKED_BAR
renderer = SUPERIMPOSED_BAR
renderer = AREA
renderer = STACKED_AREA
renderer = POLYLINE
renderer = SCATTER
renderer = STAIR
renderer = PIE

Renderers

Interaction Properties

Property	Description	Default value
interactors	Defines the possible interactions that are handled by this chart bean	none
xScrollBar	Defines the scroll bar associated with the x-axis	none

Interactor properties

The JViews Charts library provides a built-in mechanism to handle the user's interactions on an `IlvChart` bean. User's interactions are performed by means of objects called *interactors*.

The interactors define atomic behaviors (like zooming or panning) that can be combined to achieve complex interactions.

Interactors are handled by an `IlvChart` bean by means of the `interactors` property, defined as an array of interactors.

The predefined interactors available to the `IlvChart` bean are: Zoom, Information-View, Pan, Highlight-Point, X-Scroll, Y-Scroll, Edit-Point, Action, and 3DView.

For more information on these interactors, you can refer to *Interacting With Charts*.

xScrollBar property

The `xScrollBar` property allows you to connect a `JScrollBar` object to the *x*-axis of an `IlvChart` bean. Once connected to an `IlvChart` bean, the scroll bar is able to scroll the visible range of the chart, and is updated following any modifications of the visible range.

Data properties

You can import data in an IlvChart by means of a data source. A data source is an object that holds the data sets to be displayed in a chart. An IlvChart bean is connected to a data model through its data source property.

You can find more information on the data model architecture in *Using the Data Model*.

Data source beans

Describes the two default data source beans provided with the JViews Charts library.

In this section

The IlvXMLDataSource bean

The `IlvXMLDataSource` bean allows you to import data from an XML file, and to display it in an `IlvChart` bean. The data file should be conform with the JViews Charts DTD as defined in *Using the Data Model*.

Property	Description	Default value
filename	The URL of the data file	none

The location of the XML file to load is defined by the `filename` property. The location can be expressed either as a file name or as a URL.

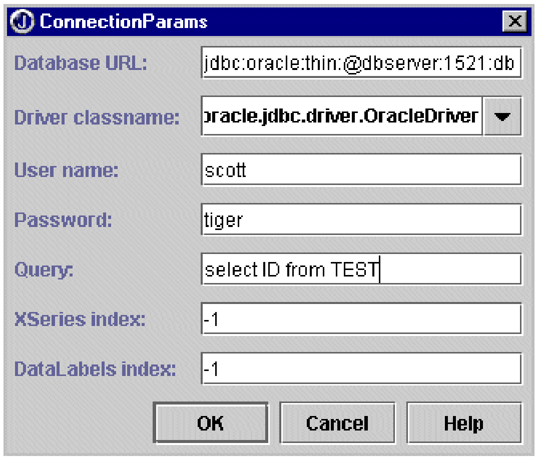
You can find more information in XML File Format in *Using the Designer*.

The IlvJDBCDataSource bean

The `IlvJDBCDataSource` bean allows you to import a data model from a database using the JDBC API.

You can find more information on the JDBC API at *http://java.sun.com/jdbc*.

The `IlvJDBCDataSource` bean is configured according to an initialization string defined by means of a dedicated property editor. The property editor is accessible by editing the data source `connectionParams` property in your IDE, and appears as in *JDBC Editor*:



Database URL:	jdbc:oracle:thin:@dbserver:1521:db
Driver classname:	oracle.jdbc.driver.OracleDriver
User name:	scott
Password:	tiger
Query:	select ID from TEST
XSeries index:	-1
DataLabels index:	-1

JDBC Editor

General properties

Property	Description	Default value
databaseURL	The database URL of the form `jdbc:subprotocol:subname`	none
driverClassname	The full class name of the JDBC Driver	none
user	The database user	none
password	The user's password	none
query	The SQL query to execute	none
xSeriesIndex	The x-series column index in the query	-1
dataLabelsIndex	The index of the data labels column in the query	-1

Connection parameters

The connection to the database is performed by a JDBC driver that requires specific information to connect to the database. This information is defined by the following properties:

- `databaseURL`

  Identifies the data source as a JDBC URL. A JDBC URL is used to find the appropriate driver able to establish the connection. This URL is of the form `jdbc:subprotocol:subname`, where `<subprotocol>` is the name of a database connectivity mechanism and `<subname>` a database vendor-dependent parameter used to identify the data source. For example, a JDBC URL for an Oracle database using the Oracle JDBC thin driver looks like:

  ```
  jdbc:oracle:thin:@myserver:1521:mydatabase
  ```

  Please refer to your database documentation for more information on how to open a connection using JDBC.

- `driverClassname`

  Defines the full class name of the JDBC driver to use, for example `oracle.jdbc.driver.OracleDriver`.

- `user`

  Defines the database user who opens the connection.

- `password`

  Defines the user's password.

Data sets parameters

Data sets are built from a `ResultSet` resulting from the execution of a query. Once the query has been executed, the result set is parsed, and data is extracted to initialize the corresponding data sets.

The following properties are involved in the data set creation:

- `query`

  Defines the query to execute, expressed as an SQL statement.

- `xSeriesIndex`

  Specifies the index of the column in the result set that contains *x*-values.

  Depending on your data, you may or may not have defined specific *x*-series for the data points. If you do not specify any *x*-series, then the index of the data points in their data set is used. By default, no *x*-series column is specified, and the property is set to -1.

- `dataLabelsIndex`

  Specifies the index of the column in the result set that contains data labels. By default, no data label column is specified and the property is set to -1.

The IlvSwingTableDataSource

The `IlvSwingTableDataSource` allows you to import data from a Swing table model (instance of `javax.swing.TableModel`) and to display it in an `IlvChart` bean.

Note: To be properly imported, a table model must contain data whose type is compatible with the supported data type.

Refer to the section *Data Source Classes* for more information on the supported data type.

Property	Description	Default value
dataLabelsIndex	The index of the data labels column in the table model.	-1
seriesType	The type of the series order (by rows or by columns).	COLUMN
tableModel	The `javax.swing.TableModel` instance to attach.	null
xSeriesIndex	The x-series column index in the table model.	-1

TableModel parameters

To determine how a `TableModel` is imported as a chart data model, the `IlvSwingTableDataSource` defines the following properties:

- `tableModel`: Defines the `javax.swing.TableModel` instance to connect.
- `seriesType`: Defines the type of the series. The `IlvSwingTableDataSource` supports two types of series ordering: by columns or by rows.

Data sets parameter

Once a table model has been connected to an `IlvSwingTableDataSource`, the corresponding data sets are created and initialized according to the following properties:

- `dataLabelsIndex`: Specifies the index of the column in the result set that contains data labels. By default, no data label column is specified and the property is set to -1.
- `xSeriesIndex`: Specifies the index of the column in the result set that contains x-values.

Depending on your data, you may or may not have defined specific *x*-series for the data points. If you do not specify any *x*-series, then the index of the data points in their data set is used. By default, no *x*-series column is specified, and the property is set to -1.

Using the JavaBeans with an IDE

The application loads the data from an XML data file, and displays it in a Cartesian chart by means of an area renderer. The zooming and panning support will also be added to allow the user's interaction on the chart.

In this section it is assumed that:

♦ You have successfully added the JViews Charts JavaBean to your IDE.

  For more information on installing new JavaBeans in the IDE, refer to the documentation of your IDE.

♦ The IDE is configured to use the JViews Charts library. In particular, this example requires that the JAXP library (`jaxp.jar` and `crimson.jar`) is accessible.

♦ You are familiar with the manipulation of JavaBeans.

The following example is developed using Borland® JBuilder® . For other IDEs the procedures are the same.

For clarity, the project creation step has been ignored, which is too IDE-dependent.

Loading the data model.

1. Display the JViews Charts beans on the beans toolbar.
2. Click the `IlvXMLDataSource` bean icon , and add it to your project.

   This component is a non-GUI JavaBean, and it should not appear in the GUI design interface.
3. Select the data source bean, and edit the `filename` property to enter the data file location.

   For this example, the file can be found in `<installdir>/jviews-charts86/samples/basic/webpages/data.xml`.

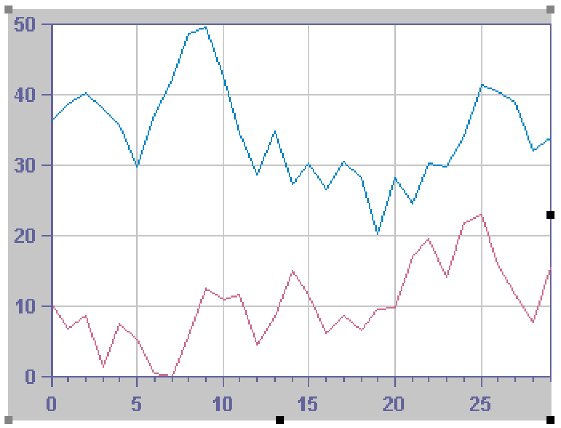
   At this time, the data source has loaded the data from the XML data file.

Creating the chart.

1. Check that the main frame of the application has a `BorderLayout` instance as layout manager.
2. Display the JViews Charts beans on the toolbar .
3. Click the `IlvChart` bean icon, and then click inside the main frame of your application.

   At this point, an `IlvChart` bean instance is created and displayed inside the frame (Note that a default data model is created at design time).
4. Check that the `IlvChart` bean is added to the frame with the CENTER constraint.

   The main frame of our application should appear as in *Beans (1)* (since data is created randomly, the data model may differ).

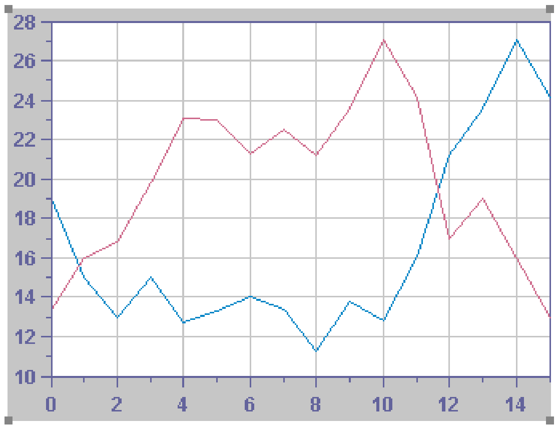


Beans (1)

Connecting the data source.

- ♦ Select the `IlvChart` beans in your frame, and edit the `datasource` property to select the instance of the `IlvXMLDataSource` you have just created.

  The chart displays the data model contained in the data source, and appears as in *Beans (2)*:



Beans (2)

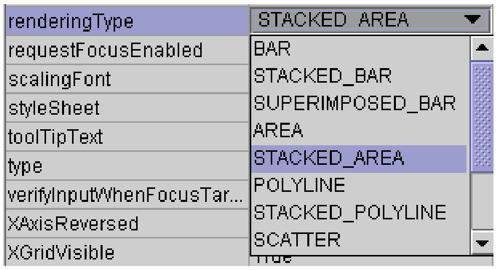
Configuring the chart.

Now that the `IlvChart` is created, you are going to modify its appearance as follows:

- ♦ Represent the data model by means of an area renderer.
- ♦ Activate/enable the antialiasing mode.
- ♦ Add a legend on top of the chart area.

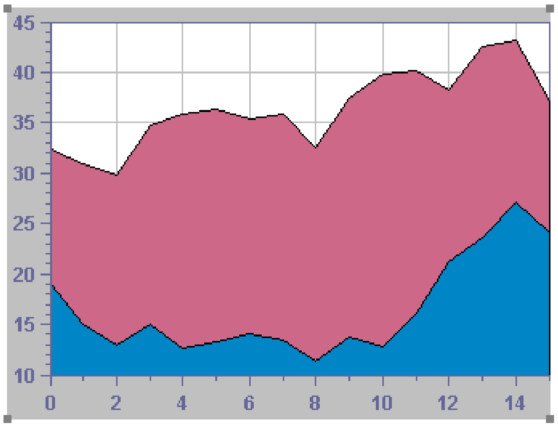
- Add support for zooming and panning the user’s interactions.
- Add a title to our chart.
- Add a horizontal scroll bar to our frame to be able to scroll the visible range of the chart in addition to the pan interactor.

1. From the property list, edit the `renderingType` property and choose `STACKED_AREA` in the combo box.


renderingType | STACKED_AREA

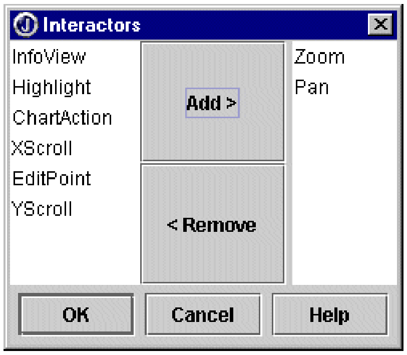
Beans (3)

The `IlvChart` appears now with area renderers.



Beans (4)

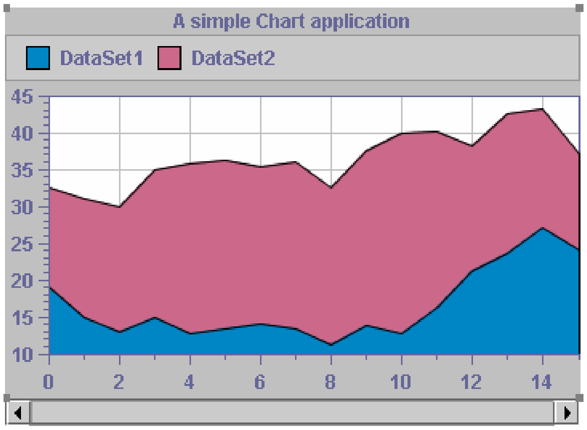
2. Set the `antiAliasing` property to `true`.
3. Set the `legendVisible` property to `true`.
4. From the property list, click the Interactors editor property (usually accessible by clicking a “...” button).
5. In the Interactors property editor, add the Zoom and Pan interactors:



Beans (5)

6. Set the `header` property to “A simple Chart application”.
7. From the Swing component palette of your IDE, select a `JScrollBar` and add it to the frame with a `SOUTH` layout constraint.
8. Select the scroll bar bean, and set its orientation property to `HORIZONTAL`.
9. Select the chart bean, and edit its `XScrollBar` property to select the scroll bar instance that you have just created.

Our application is now completed and ready to be run. The frame should appear as follows:



Beans (6)

Check that all the required Java™ libraries have been added to the project class path, compile the project, and run it.

Using JViews products in Eclipse RCP applications

The Standard Widget Toolkit (SWT) is the window toolkit of the Eclipse™ development environment and the Eclipse Rich Client Platform (RCP). This topic shows you how to display charts embedded in an SWT window.

Installing the JViews runtime plugin

IBM® ILOG® JViews Charts provides jar files in the form of a pre-packaged Eclipse plugin. The name of this package is `ilog.views.eclipse.chart.runtime`.

In order to install the IBM® ILOG® JViews Eclipse plugins, you need to install from the local site as shown below.

For Eclipse 3.3:

1. Launch your Eclipse installation.
2. Go to **Help/Software Updates/Find And Install**.
3. In the `Install/Update` dialog box, click **Search for new features to install**.
4. Define a New Local Site with the directory `<installdir>/jviews-framework86/tools/ilog.views.eclipse.update.site`.
5. Select the features you want to install.

For Eclipse 3.4:

1. Launch your Eclipse installation.
2. Go to **Help/Software Updates** and select the **Available Software** tab.
3. Add a new local site: Click **Add Site**, then **Local** and specify the directory `<installdir>/jviews-framework86/tools/ilog.views.eclipse.update.site`
4. Select the features you want to install, and press the **Install** button.

This installation also installs some examples. See Installing and using Eclipse samples for more information.

In your applications, you need the `ilog.views.eclipse.chart.runtime` plugin and its dependencies:

- `ilog.views.eclipse.chart.runtime`
- `ilog.views.eclipse.framework.runtime`
- `ilog.views.eclipse.utilities.runtime`

Providing access to class loaders

Many services in JViews need to look up a resource. Since the classical way to provide access to resources is a classloader, JViews uses classloaders for this purpose. But in Eclipse/RCP applications, each plugin corresponds to a classloader, and the JViews classloader sees only its own resources, not the application resources. To fix this problem, you can register plugin classloaders with JViews through the `IlvClassLoaderUtil.registerClassLoader` function. Each resource lookup then considers the registered classloaders and, if the plugins are configured accordingly, also considers the dependencies of the registered classloaders.

The code for doing this is usually located in a plugin activator class. For example:

```
public class MyPluginActivator extends AbstractUIPlugin
{
    /**
     * This method is called upon plugin activation
     */
    public void start(BundleContext context) throws Exception {
      super.start(context);
      IlvClassLoaderUtil.registerClassLoader(getClass().getClassLoader());
    }

    /**
     * This method is called when the plugin is stopped
     */
    public void stop(BundleContext context) throws Exception {
      super.stop(context);
      IlvClassLoaderUtil.unregisterClassLoader(getClass().getClassLoader());
    }

  }
```

The overriding of `stop()` is necessary so that, when the plugin gets unloaded, JViews gets notified about the plugin that is going to stop and can drop references to its resources or instances of its classes. The activator plugin is usually also the place where `IlvProductUtil.registerApplication` is called. See section Before you start deploying an application for an example.

The bridge between AWT/Swing and SWT

The Standard Widget Toolkit (SWT) is the window toolkit of the Eclipse development environment and the Eclipse Rich Client Platform (RCP).

JViews provides an `IlvSwingControl` class that encapsulates a Swing JComponent in an SWT widget. It allows you to use `IlvChart` or `IlvLegend` objects in an SWT window, together with other SWT or JFace controls. In this way, it provides a bridge between the AWT/Swing windowing system and the SWT windowing system.

The following code shows how to create a bridge object:

```
Composite parent = ...;
IlvChart chart = new IlvChart();
ControlSWTchart = new IlvSwingControl(parent, SWT.NONE, chart);
```

Using `IlvSwingControl` instead of the native SWT_AWT class has the following benefits:

- Simplicity: it is easier to use, since you do not have to worry about the details of the `Component` hierarchy (see *http://java.sun.com/javase/6/docs/api/java/awt/Component.html*).

- Portability: `IlvSwingControl` also works on platforms that do not have SWT_AWT, like X11/Motif® and MacOS® X 10.4.

- Less flickering: on Linux®/Gtk, flickering is reduced.
- Popup menus: popup menus can be positioned on each `Component` inside the AWT component hierarchy. For details of components, see *http://java.sun.com/javase/6/docs/api/java/awt/Component.html*.
- Better size management: the size management between SWT and AWT (`LayoutManager`) is integrated.
- Focus: it provides a workaround for a focus problem on Microsoft® Windows® platforms.

Note: The `IlvSwingControl` bridge is not supported on all platforms. It is only supported on Windows, UNIX® with X11 (Linux, Solaris™, AIX®, HP-UX®), and MacOS X 10.4 or later.

The `IlvSwingControl` bridge does not support arbitrary JComponents. Essentially, components that provide text editing are not supported. See `IlvSwingControl` for a precise description of the limitations.

Threading modes

You can handle the SWT-Swing user interface events in one or two threads.

Note: Single-thread mode is incompatible with AWT/Swing Dialogs. If you use single-thread mode, you cannot use AWT `Dialogs`, Swing `JDialogs`, or modal `JInternalFrames` in your application. There are also some other limitations. See the class `IlvEventThreadUtil` for a precise description of the limitations.

- Two-thread mode

  The SWT events are handled in the SWT event thread and AWT/Swing events are handled in the AWT/Swing event thread. This is the default mode.

  You can switch between the two threads by using the SWT method `Display.asyncExec()` and the AWT method `EventQueue.invokeLater()`.

  If your application uses this mode, you must be careful to:

  - Make API calls on SWT widgets only in the SWT event thread. Otherwise, you will get `SWTExceptions` of type `ERROR_THREAD_INVALID_ACCESS`.
  - Make API calls on JComponents, which include `IlvChart` and `IlvLegend`, only in the AWT/Swing event thread. Otherwise, you risk deadlocks.

    You can switch between the two threads by using the SWT method `Display.asyncExec()` and the AWT method `EventQueue.invokeLater()`.

- Single-thread mode

  In single-thread mode, SWT and AWT/Swing events are handled in the same thread.

Single-thread mode reduces the risk of producing deadlocks.

Enable this mode by calling `setAWTThreadRedirect` or `enableAWTThreadRedirect()` early during initialization.

The following example shows how to enable single-thread mode:

```
// Switch single-event-thread mode during a static initialization.
      static {
          IlvEventThreadUtil.enableAWTThreadRedirect();
      }
```

This mode reduces the risk of producing deadlocks. If you are using JComponents other than `IlvChart` and `IlvLegend` in your application, your JComponents must use the method `isDispatchThread()` rather than *`EventQueue.isDispatchThread()`* or *`SwingUtilities.isEventDispatchThread()`*.

Note: This mode is incompatible with AWT/Swing Dialogs. If you use single-thread mode, you cannot use AWT Dialogs, Swing JDialogs, or modal JInternalFrames in your application. There are also some other limitations. See the class `IlvEventThreadUtil` for a precise description of the limitations.

Printing

Describes how to print charts in two different modes: in a flow and with a custom document structure.

In this section

Printing a chart in a flow

Describes how to print `IlvChart` objects in a printing flow that contains other printable objects. This mode supports complex page format (paragraph alignment, local fonts, and so on) and is particularly well suited for reporting.

In this section

Flow

A flow consists of a list of printable objects of different types (text, charts, manager, and so on) that are printed sequentially in a document.

Flow objects are instances of the `ilog.views.util.print.IlvFlow` class and are directly obtained from the printable document by means of the `getFlow()` method.

Note: There is only one flow per document.

A flow object is responsible for creating the document pages and defining their layout by positioning the printable object according to the pages, paragraphs, and text formats defined in the flow.

Creating a flow involves the following steps:

1. Ccreate a document and a printing controller.
2. Get the document flow object using the `IlvPrintableDocument.getFlow` method.
3. Add the printable objects to the flow using the appropriate `IlvFlow.add` methods.

The IlvChartFlowObject class

The printing of an `IlvChart` in a flow is performed by a specialized printable object, which is an instance of the `IlvChartFlowObject` class.

This class allows you to specify the size of a printed chart in two different ways:

♦ As a fixed size in paper coordinates.

  The fixed size must be expressed in the page coordinate system. It is specified as an `ilog.views.util.print.IlvUnit.Dimension` object and might be of any units supported by the `IlvUnit` class.

  For example, the following code prints an `IlvChart`, which is 7 cm wide and 5 cm high:

```
IlvChartFlowObject flowChart =
        new IlvChartFlowObject(chart,
                               new IlvUnit.Dimension(7,5, IlvUnit.CM);
```

♦ As a percentage of the page dimensions.

  In this mode, one dimension is specified as a percentage of the corresponding page dimension, while the other one is automatically computed according to a specified aspect ratio. You can specify the size either in the constructor or by means of the `setPercentWidth(int, float)` and `setPercentHeight(int, float)` methods.

Printing a chart in a flow

The complete source code of this example can be found in **`<installdir>/jviews-charts86/codefragments/chart/printing/src/PrintFlowExample.java`**.

Note: Only the code related to the printing is shown.

Creating the document and the printing controller.

♦ Use the following code:

```
IlvChart chart = ...;
IlvPrintableDocument document =
    new IlvPrintableDocument("ChartInFlow");
IlvPrintingController controller =
    new IlvPrintingController(document);
```

Constructing the flow.

♦ Get the flow object of the document you have just created:

```
IlvFlow flow = document.getFlow();
```

Populate it.

1. Add a title, with a center alignment.

```
IlvFlow.TextStyle style = new IlvFlow.TextStyle();
style.setAlignment(IlvFlow.TextStyle.CENTER_ALIGNMENT);
style.setFont(new Font("Dialog", Font.PLAIN, 20));
flow.setTextStyle(style);
flow.add("A Chart in a Flow");
```

2. Add a chart in a new paragraph.

   Its size will take 60% of the page width, with an aspect ratio between the width and height equal to 0.75:

```
flow.newLine();
IlvChartFlowObject printChart =
    new IlvChartFlowObject(chart, flow, 60, 0, .75f);
flow.add(printChart, IlvFlow.BOTTOM_ALIGNMENT);
```

3. Add text below the chart as a new paragraph, left justified.

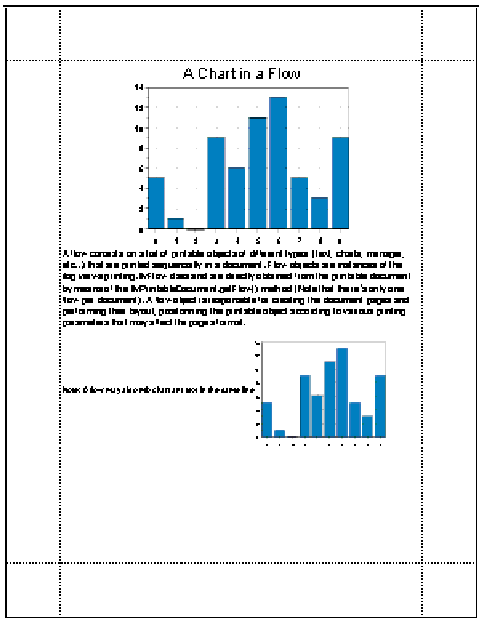
```
style.setAlignment(IlvFlow.TextStyle.LEFT_ALIGNMENT);
style.setFont(new Font("Dialog", Font.PLAIN, 12));
flow.setTextStyle(style);
flow.newLine();
String text = ...;
flow.add(text);
```

4. Add a new paragraph containing a chart and text on the same line.

   The chart is 6 cm wide and 5 cm high, centered vertically on the text baseline.

```
style.setFont(new Font("Dialog", Font.PLAIN, 10));
flow.setTextStyle(style);
flow.newLine();
text = ...;
flow.add(text);
printChart =
  new IlvChartFlowObject(chart, flow, new IlvUnit.Dimension(6,5, IlvUnit.
CM));
flow.add(printChart, IlvFlow.CENTER_BASELINE);
```

Here is a picture showing the final result:



A flow consists of a list of printable objects of different types (text, charts, images, etc.) that are printed sequentially in a document. Flow objects are instances of the ilog.views.printing.IlvFlow class and are directly obtained from the printable document by means of the IlvPrintableDocument.getFlow() method (Note that there's only one flow per document). A flow object is responsible for creating the document pages and performing their layout, positioning the printable object according to various printing parameters that may affect the page format.

Now, this flow displays both a chart and text in the same line

Printing a chart with a custom document structure

Describes how to print an `IlvChart` with optional additional objects in a document (single or multipage) but without the automatic formatting capabilities that the flow mode offers.

In this section

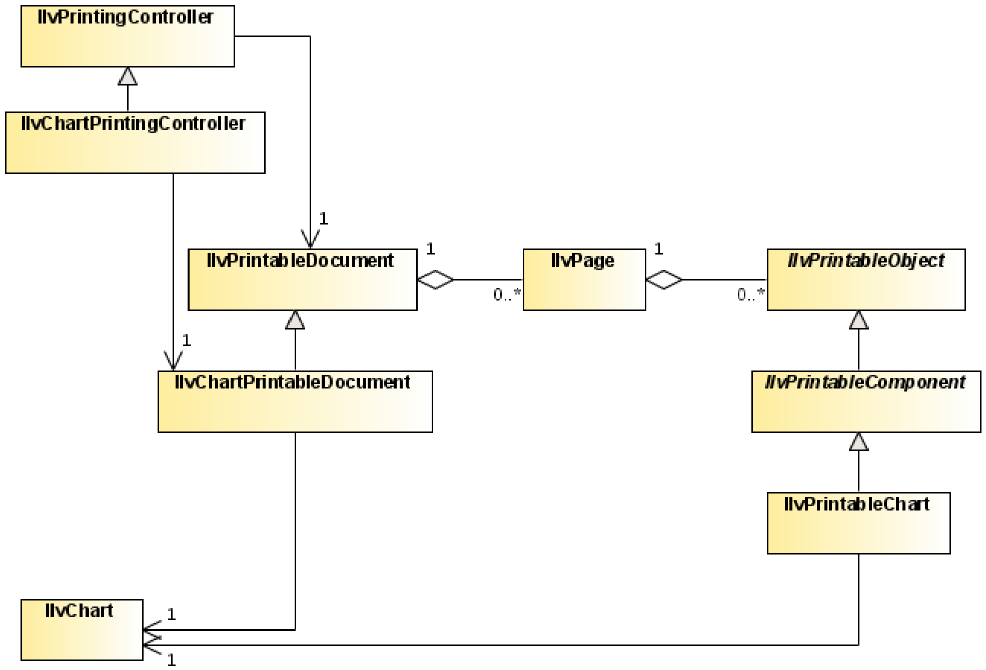
The classes involved

As opposed to the flow printing mode, where both the document and the printing controller were instances of generic classes of the `ilog.views.util.print` package, the custom mode requires specialized subclasses to handle specific chart properties. These classes are:

♦ A chart printing controller, instance of the `IlvChartPrintingController` class.

♦ A chart printable document, instance of the `IlvChartPrintableDocument` class.

♦ A chart printable object, instance of the `IlvPrintableChart` class.



The Classes Involved When Printing a Chart

The IlvChartPrintingController class

The printing controller controls the printing process. It initiates the printer job, handles the setup and the preview dialog boxes, and configures the document accordingly.

The IlvChartPrintableDocument class

The printable document stores the printed document structure and defines a set of parameters to customize the printing (the printed data window, the part of the chart to be printed, how the chart fits on the page, and so on). It is responsible for creating and populating the pages.

The following table lists the `IlvChartPrintableDocument` properties.

Property	Methods	Default Value
Printed data window	`getDataWindow` `setDataWindow`	
Number of pages (for multipage document)	`getMultiPageCount` `setMultiPageCount`	
Repeat x-scale title on all pages	`getRepeatTitle` `setRepeatTitle`	`false`
Repeat y-scales on all pages	`getRepeatYScales` `setRepeatYScales`	`false`
Resize mode	`getResizeMode` `setResizeMode`	`FIT_TO_DIMENSION`
Scaling Alignment	`getScalingAlignment` `setScalingAlignment`	XMID_YMID
Resolution scale factor	`getResolutionScaleFactor` `setResolutionScaleFactor`	`MEDIUM_RESOLUTION`
Printed component	`isPrintingChart` `setPrintingChart`	

Printed Data Window

This property specifies the data window to print. By default, the printed data window corresponds to the data range of the chart default coordinate system.

Number of Pages for Multipage Document

The JViews Charts printing framework provides support for multipage printing through the `multiPageCount` property.

A multipage printing consists of dividing the x-range of the printed data window in successive intervals with respect to the number of pages. You can change the way the data window of a page is computed by overriding the `computeDataWindow(int, int)` method.

Repeat X-Scale Title on all Pages

When the x-scale title placement is other than 0 or 100 (that is, the axis extremities), this property sets whether the title has to be printed on each page of the document.

Repeat Y-Scales on all Pages

This property sets whether y-scales with a minimum or maximum crossing value (that is, positioned on the axis extremities) need to be printed on each page of the document.

Resize Mode

To determine the printed chart size, the `IlvChartPrintableDocument` defines two resizing modes:

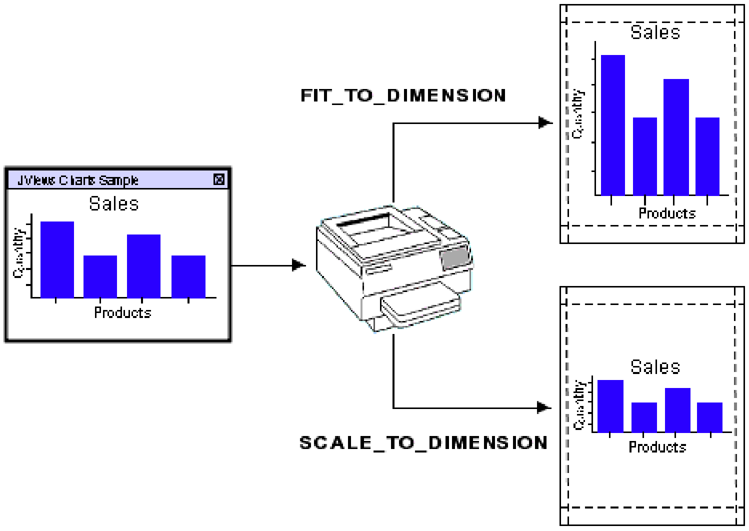
- Fit to the dimension of the printable area.

  The chart is expanded to fill the printable area in both dimensions. The original proportions between the chart objects are not preserved. This mode corresponds to the `IlvChartPrintableDocument.FIT_TO_DIMENSION` constant.

- Scale to the dimension of the printable area.

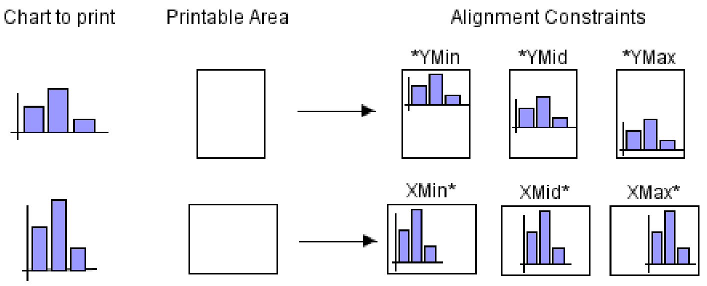
  The chart is expanded in both dimensions proportionally, until the nearest page margins are reached. The original proportions between the chart objects are preserved. This mode corresponds to the `IlvChartPrintableDocument.SCALE_TO_DIMENSION` constant.

The following figure illustrates the result of these two modes:



Scaling Alignment

When the resize mode property is set to `SCALE_TO_DIMENSION`, the `scalingAlignment` property indicates the method to use when the aspect ratio of the chart does not match the aspect ratio of the printable area. The following figure shows the result for the different values depending on the destination printable area. The default value is `XMID_YMID`, that is, the chart is centered within the printable area.



Scaling Alignment

Resolution Scale Factor

Because the printer resolution is usually higher than the screen resolution, there may be some discrepancies between the screen appearance and the printing result. This is particularly true for labels, whose font size may not be adapted for printing. To reduce this effect, an additional scale transform may be applied on the printing graphics context when drawing the chart area.

There are three predefined resolution scale factors:

- High (`IlvChartPrintableDocument.HIGH_RESOLUTION`)
- Medium (`IlvChartPrintableDocument.MEDIUM_RESOLUTION`)
- Low (`IlvChartPrintableDocument.IDENTITY_RESOLUTION`)

where the low resolution scale factor is the identity transform.

The default value is Medium.

Printed Component

This property defines which chart, or chart area, is printed. Printing a chart means that all the components added to an `IlvChart` object will be printed. This includes the header, the footer, and the legend. Conversely, printing only the chart area means that all the other chart components, including the legend, will be ignored.

The default value is `true`. This means that the whole chart is printed.

Note: All these properties are also accessible in the Chart Setup tab of the Page Setup dialog box.

The IlvPrintableChart class

The printable object prints the `IlvChart` within a region of the printable area of an `IlvPage`. The `resizeMode` document property determines how the chart fills the region.

How it works

This can be done either:

- ♦ by code

  using the `print(boolean)` method,

  or

- ♦ from a GUI request

  using the setup or preview dialog box, by means of the `printPreview(java.awt.Window)` and `setupDialog(java.awt.Window, boolean, boolean)` methods.

When a printing task is initiated, the document associated with the printing controller is prepared for printing: the pages are initialized with the printable objects and added to the document.

Handling pages

The pages of a chart document are instances of the `IlvPage` class. They handle a collection of printable objects, instances of `IlvPrintableObject`.

Populating a page

The pages are created in an `IlvChartPrintableDocument` by means of the `createPages()` factory method. By default, a page contains one `IlvPrintableChart`. If you want to add additional printable objects to the page, you may override the `createPages` method.

For example, assume that you want to add a border on the page. The `ilog.views.util.print` package provides the `IlvPrintableRectangle` class to print a rectangle. What you have to do is to add such an object to the page after the chart has been added, by subclassing `IlvChartPrintableDocument` and overriding the `createPages` method to add an instance of `IlvPrintableRectangle`.

The code is:

```
class MyChartDocument extends IlvChartPrintableDocument {
  protected IlvPage[] createPages() {
    IlvPage[] pages = super.createPages();
    for (int i=0 ; i<pages.length ; ++page)
     pages[i].addPrintableObject(new IlvPrintableRectangle(getImageableBounds
()));
    return pages;
  }
}
```

First invoke the method of the mother class, so that the chart is added in the first position. Then, add a new instance of `IlvPrintableRectangle` to each page.

Computing the data window of a page

As mentioned in *Number of Pages for Multipage Document*, a chart can be printed as a multipage document. Such a document creates as many pages as specified, and divides the data window to print into as many successive data windows. Each page prints its corresponding data window. By default, the splitting is performed along the data window x-range, with a y-range equal to the printed data window y-range. This computation is done in the `computeDataWindow` method, which you can override to apply your own policy.

Printing the chart: the printing context

An `IlvChartPrintableDocument` prints, in an `IlvPage`, a given data window of the data model of an `IlvChart`, as it would be displayed by this chart.

To avoid affecting the appearance of the printed chart by modifying its configuration, the JViews Charts printing framework introduces the concept of *printing context* through the abstract `IlvChartPrintContext` class. The purpose of a printing context is to define a drawing configuration for a chart, independent of its current state. There is one printing context per printable chart, and each context defines the printing of each page.

Mainly, a chart context defines:

- the print range of a given axis,
- the visibility of the drawable objects of the printed chart,
- the visibility of the title of a scale.

The print range of each axis of a chart is handled by the abstract `getRange(int)` method. This method returns the range to print for a specified axis, and a default implementation is provided by the `IlvDefaultChartPrintContext` class. The default policy returns the corresponding range of the printed data window of the current page (see the section *Computing the data window of a page*) for the default coordinate system axis and the data range of the axis for other coordinate systems.

The visibility of the chart drawable objects during a printing may be defined independently of the current chart state by means of the `isVisible(ilog.views.chart.IlvChartDrawable)` method. The default policy takes care of only y-scales, their visibility depending on the value of the document `repeatYScales` property and on the scales anchor.

Finally, the scale title visibility is checked using the `isTitleVisible(ilog.views.chart.IlvScale)` method. The default implementation takes care of only the x-scale title, checking whether the title must be repeated on each scale according to the title placement along the axis and to the document settings.

Customizing the printing of a chart

This example is extracted from the `tablemodel` demo. You can find the complete source code of this example in **`<installdir>/jviews-charts86/samples/tablemodel/src/tablemodel/TableModelDemo.java`**.

Note: Only the code related to the printing is shown.

The requirements are:

- ♦ a title at the top of the page,
- ♦ a chart in the top-middle part of the page,
- ♦ the table at the bottom middle part of the page.

The title is printed using an `IlvPrintableLabel` object. This object is added to the top of the page, centered horizontally.

The chart is printed using an `IlvPrintableChart`. It fills horizontally the printable area of the page, and extends vertically half of the printable area height of the page.

Finally, the table is rendered in two parts: first the header, using an `IlvPrintableTableHeader` and the table itself, using an `IlvPrintableTable`.

Creating the document class.

- ♦ Use the following code:

```
public class TablePrintableDocument extends IlvChartPrintableDocument
{
    private JTable table;

    public TablePrintableDocument(String name,
                                  JTable table,
                                  IlvChart chart)
    {
        super("test_chart",
              chart,
              false,
              1,
              false,
        new IlvDataWindow(chart.getXAxis().getDataRange(),
                          chart.getYAxis(0).getDataRange()),
              new PageFormat());
        this.table = table;
        // Initialize the page format.
        getPageFormat().setPaper(new Paper());
        getPageFormat().setOrientation(PageFormat.LANDSCAPE);
    }
}
```

Populating the document pages with the label and the table.

1. Add additional objects to a page of a document by overriding the `createPages()` method:

```
  protected IlvPage[] createPages()
  {
    IlvPage[] allPages = super.createPages();
    IlvUnit.Rectangle pageArea = getImageableBounds();
    for (int i=0; i<allPages.length; ++i) {
      IlvPage page = allPages[i];
      ...
    }
    return allPages;
  }
```

2. Add a printable label, placed at the top of the chart and centered horizontally:

```
    for (int i=0; i<allPages.length; ++i) {
      IlvPage page = allPages[i];
      if (title != null) {
        IlvPrintableLabel prnLabel = new IlvPrintableLabel(title,
titleArea);
        page.addPrintableObject(prnLabel);
      }
      ...
    }
```

3. Add the table header:

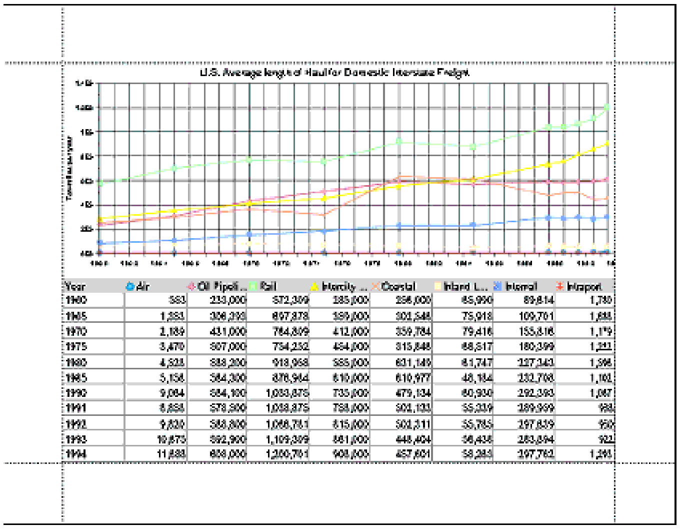
```
double chartHeight = getChartHeight(pageArea);
    double headerX = pageArea.getX();
    double headerY = titleArea.getY()+titleArea.getHeight()+chartHeight;

    JTableHeader header = table.getTableHeader();
    double headerHeight = header.getHeight();
    double headerWidth = pageArea.getWidthAs(IlvUnit.POINTS);
    if (headerHeight == 0 )
      headerHeight = header.getPreferredSize().height;
    int startCol = pageIdx*table.getModel().getColumnCount() /
getMultiPageCount();
    int endCol = (pageIdx+1)*table.getModel().getColumnCount() /
getMultiPageCount()-1;
    IlvUnit.Rectangle headerArea = new IlvUnit.Rectangle(headerX,
                                                         headerY,
                                                         headerWidth,
                                                         headerHeight,
                                                         IlvUnit.POINTS)
```

```
;
    page.addPrintableObject(new IlvPrintableTableHeader(header,
                                                          headerArea,
                                                          startCol,
                                                          endCol));
```

4. Add the table itself:

```
double tableY = headerY + headerHeight;
   IlvUnit.Rectangle printArea =
        new IlvUnit.Rectangle(headerX,
                              tableY,
                              headerWidth,
                              pageArea.getHeightAs(IlvUnit.POINTS)
-chartHeight-2*headerHeight,
                              IlvUnit.POINTS);
   page.addPrintableObject(new IlvPrintableTable(table,
                                                 printArea,
                                                 startCol,
                                                 endCol,
                                                 0,
                                                table.getModel().
getRowCount() -1));
    ...
```

The following picture shows the final result:



Year	● Air	◆ Oil Pipeli...	🟩 Rail	▲ Intercity ...	✕ Coastal	🟨 Inland L...	▨ Internal	✱ Intraport
1960	583	233,000	572,309	285,000	256,000	65,990	89,814	1,780
1965	1,333	306,393	697,878	359,000	301,548	75,918	109,701	1,688
1970	2,189	431,000	764,809	412,000	359,784	79,416	155,816	1,179
1975	3,470	507,000	754,252	454,000	315,848	88,517	180,399	1,211
1980	4,528	588,200	918,958	555,000	631,149	81,747	227,343	1,398
1985	5,156	564,300	876,984	610,000	610,977	48,184	232,708	1,102
1990	9,064	584,100	1,033,875	735,000	479,134	60,930	292,393	1,087
1991	8,858	578,900	1,038,875	758,000	501,133	55,239	289,999	988
1992	9,820	588,800	1,066,781	815,000	502,311	55,785	297,839	950
1993	10,673	592,900	1,109,309	861,000	448,404	58,438	283,894	922
1994	11,888	608,000	1,200,701	908,000	457,601	58,284	297,762	1,291

Generating PDF

The PDF generation uses the standard *XSL-FO* format as intermediate format:

`IlvChart` ---> XSL-FO ---> PDF

The XSL-FO format allows you to fine-tune the details of the page layout. The generation of XSL-FO from an `IlvChart` is done through one of the following APIs:

- `IlvChart.paintToFO`
- `IlvFOUtil.paintToFO`

This step requires the availability of `batik-jviews-x.y.jar` in the `CLASSPATH`.

The second step, the conversion from FO to PDF, uses the Apache™ *FOP* package. It requires the presence of the following `.jars` in the CLASSPATH:

- `fop-0.20.5.jar`
- `fop-avalon-framework-0.20.5.jar`
- `fop-batik-0.20.5.jar`

The XSL-FO generation is usable in both multithreading modes (see Choosing the multithreading mode). For the *using event thread* mode (that is, typically a Swing application) you will use `IlvChart.paintToFO` or `IlvFOUtil.paintToFO`, while for the *current thread* mode (that is, typically in a Web server) you will use `IlvChart.paintToFOCurrentThread`, or apply `IlvFOUtil.paintToFO` to a chart with `isUsingEventThread() = true`.

You can find a sample showing the PDF generation in `samples/tablemodel/`.

Note: JViews Charts does not support the generation of interactive PDF documents (PDF with embedded JavaScript™).

Index