

# JIDE TreeMap Developer Guide

---

## Contents

<b>PURPOSE OF THIS DOCUMENT .....</b>	<b>2</b>
<b>TREEMAPS BASIC .....</b>	<b>2</b>
<b>JIDE TREEMAP FEATURES .....</b>	<b>2</b>
<b>DATA SOURCE .....</b>	<b>3</b>
<b>JIDE TREEMAP QUICK START .....</b>	<b>3</b>
<b>JIDE TREEMAP ARCHITECTURE .....</b>	<b>5</b>
DEFAULTTREEMAPMODEL .....	5
DEFAULTTREEMAPVIEW .....	5
DEFAULTTREEMAPCONTROLLER .....	5
<b>JIDE TREEMAP CUSTOMIZATION .....</b>	<b>5</b>
CONFIGURE THE SIZE .....	6
ASSIGNING COLORS .....	6
ADDING A THIRD DIMENSION .....	6
HIERARCHICAL GROUPING .....	7
LABELING .....	7
LAYOUT ALGORITHMS .....	8
APPEARANCE AND RENDERING OPTIONS .....	8
CUSTOMIZING TEXTUAL DISPLAY .....	9
TOOLTIP CONTENT CUSTOMIZATION .....	9
<b>INTERACTING WITH JIDE TREEMAP .....</b>	<b>10</b>
PROBING AND SELECTION .....	10
<b>EXTENDING JIDE TREEMAP .....</b>	<b>10</b>
CONTEXT MENU CUSTOMIZATION .....	10
<b>HANDLING DYNAMIC DATA.....</b>	<b>10</b>
<b>SCALABILITY .....</b>	<b>10</b>
<b>WHAT'S NEXT.....</b>	<b>11</b>

## Purpose of This Document

Welcome to the *JIDE TreeMap*. *JIDE TreeMap* is a Java/Swing implementation of the treemap visualization technique. This developer guide is designed for developers who want to learn how to use *JIDE TreeMap* in their applications.

*JIDE TreeMap* depends on features and components provided by *JIDE Grids*. So if you never used *JIDE Grids* before, we strongly recommend you read *JIDE Grids Developer Guide* first or at least refer to it while reading this developer guide.

## Treemaps Basic

Treemaps graphically represent information about objects by dividing the display into **areas (typically rectangles) that are proportional to the size of each object**. It also enables the display of hierarchical information by nesting each area into subregions. It was invented and first used in the early 1990s by Ben Shneiderman at the University of Maryland for the management of the disk space of his server. Treemaps display rows of data as groups of squares that can be arranged, sized and colored to graphically reveal underlying data patterns. This visualization technique allows users to explore and easily recognize complicated data relationships.

## JIDE TreeMap Features

*JIDE TreeMap* comes with an extended set of features, including:

- Support for popular Swing *TableModel*
- Flexible configuration of size, color, and labels of the treemap elements
- Complete set of layout algorithm (including solid squarified and aesthetically-pleasing circular layout algorithm)
- Cushion rendering to reveal hierarchy intuitively
- Zoomable user interface, including drilling
- Many options to fine-tune the appearance of the display
- Flexible hierarchy definition to create custom aggregation schemes
- Filtering support
- Details on demand provided with popups
- Useful for small datasets already, but scales to 100'000s of data objects

## Data Source

You may read your data from any data sources such as a database table, a file, a piece of data in memory. The data should be in tabular format that can be converted to *TableModel* as defined in Java Swing. From *JIDE TreeMap* point of view, the only data it will accept is the *TableModel*. As long as you convert your raw data to *TableModel*, you can use it in *JIDE TreeMap*. Should you want to convert a *JDBC ResultSet*, *JIDE Data Grids* provides two handy table models to make the conversion easy: *ResultSetTableModel* and *DatabaseTableModel*.

## JIDE TreeMap Quick Start

This section contains some examples that demonstrate how easy it is to get up and running with *JIDE TreeMap*. The following sections provide much more detail about how to configure your charts and which features are available, but most developers are eager to get something working quickly, so here is a quick working example.

```
public class Hello {
    public static void main(String[] args) {
        // Defining the data, column names and types
        Object[][] data = new Object[][]{
            {"Hello", 12, 3.0},
            {"from", 11, 4.0},
            {"the", 9, 5.0},
            {"TreeMap", 8, 6.0},
            {"World!", 7, 7.0},
        };
        Object[] columnNames = new Object[]{"Name", "Value", "Strength"};
        final Class[] columnTypes = new Class[]{String.class, Integer.class, Double.class};

        // Creating a standard Swing TableModel
        TableModel tableModel = new DefaultTableModel(data, columnNames) {
            public Class<?> getColumnClass(int columnIndex) {
                return columnTypes[columnIndex];
            }
        };

        // Creating the TreeMap
        TreeMap treeMap = new TreeMap(tableModel);

        // Tuning the appearance of the TreeMap
        treeMap.setAlgorithm(AlgorithmFactory.SQUARIFIED);
        treeMap.setSizeByName("Value");
        treeMap.setColor(2);
        treeMap.setBackgroundByName("Name");
        treeMap.setLabels();

        // Creating a frame to display
        final JFrame frame = new JFrame("Hello from the TreeMap World!");
        frame.setSize(600, 600);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add(treeMap);
        frame.setLocationRelativeTo(null);
        frame.setVisible(true);
    }
}
```

```
}
```

The code above will produce the output below. One can see that the area of each rectangle is proportional to the “Value” column and the colors assigned depending on the “Strength” column. The values of the “Name” column are used to label each rectangle by filling the entire width and scaling the font accordingly. While the content of this example is meaningless, it highlights the key principles and one should be able to extrapolate the use of such a visualization technique when applied to budget, sales, quality control, fraud detection, and financial data and any other areas where getting the big picture reveals pattern.



## JIDE TreeMap Architecture

The *com.jidesoft.treemap.TreeMap* class acts as a façade to the *JIDE TreeMap* model-view-controller (MVC) architecture. In brief, the controller collects user input, the model manipulates application data, and the view presents results to the user. This class wraps a *TreeMapModel*, *TreeMapView*, and *TreeMapController* interfaces together. It allows easy loading of the data and customization of the most common settings. By default, it uses the following implementations of these interfaces:

### DefaultTreeMapModel

The default implementation of the *TreeMapModel* interface uses a standard Swing *TableModel* as the underlying data holder. The default implementation of the *TreeMapModel* interface will automatically map the size to the first numerical column, the color to the second numerical column, the labels to the first categorical column, and the group by to the second categorical column.

If your data model is not in tabular form or you want to customize how the data are fed into the *TreeMap*, then you can extend the *AbstractTreeMapModel* class. Key methods to implement are *getChildren()*, *getParent()* and *getValueAt()*.

### DefaultTreeMapView

The default implementation of the *TreeMapView* interface will automatically coalesce multiple changes to the model to a single repaint operation and has the possibility of updating the display progressively should very large datasets be displayed.

### DefaultTreeMapController

The default implementation of the *TreeMapController* interface supports probing, selection, zooming, panning, and drilling operations. Probing is achieved by simply moving the mouse over the shape of interest and a popup will then display detailed information about that item. Selection is done through left-mouse click and lasso operations by pressing down the Alt key while dragging the mouse. Zooming can comfortably be done using a mouse wheel or zooming gesture of a trackpad. Finally, drilling is supported using the Page-Down and Page-Up keys.

## JIDE TreeMap Customization

JIDE TreeMap has been designed with customizability in mind. The most common settings can be changed through the *TreeMap* façade. Should this not be enough, the *TreeMapSettings* class can be obtained using *TreeMap.getModel().getSettings()* and provides access to global options, while *TreeMapFieldSettings* provides access to field-specific options. Default settings can be changed using *TreeMapSettings.getDefaultFieldSettings()* and can be overridden on a field-by-field basis using *TreeMapSettings.getFieldSettings(TreeMapField)*. This provides, for example, with a mean of using different layout algorithm at each hierarchy level.

## Configure the size

As the treemap visualization technique attempts to keep the area of the shapes on the screen proportional to some data values, it is key to specify which column in the *TableModel* should be used as a proxy for the size. This can be specified using one of the following two methods, depending on whether you prefer to use the column index or the column name:

```
treeMap.setSize(2);
treeMap.setSizeByName("Value");
```

The specified column should contain numerical values of type Integer, Float, or Double. If the value is null, then it will simply not be included in the treemap layout. Since values can be negative but shapes cannot have negative areas, the default behavior is to use the absolute value. To override this behavior, it is possible to use another scaling scheme, such as:

```
treeMap.setScale(ScaleFactory.getInstance().get("Original"));
```

In this case, negative values will not be included in the treemap layout.

## Assigning colors

How each shape should be colored can be defined by using a colormap that will convert numerical and categorical values into colors according to a defined scale or dictionary. Which column to use should be defined using its index or name:

```
treeMap.setColor(2);
treeMap.setColorByName("Strength");
```

## Creating colormaps

To override the default colormap that is assigned to each column, it is possible to set a customized one, for example:

```
treeMap.getModel().getSettings().getFieldSettings(treeMap.getModel().getTreeMapField(2)).setColorMap(
ColorMapFactory.getInstance().createSequentialColorMap(0, 100));
```

The *ColorMapFactory* class provides a range of static methods for creating standard colormaps for categorical, sequential, and diverging values. The *PaletteFactory* gives access to a wide range of predefined color gradients.

A *Colormap* contains both an *Interval* and a *Palette*, which are conjointly used to map the actual values into the normalized range (0..1) used to retrieve the colors defined in the palette. To create a custom colormap and palette that associate -200 to red, 0 to white, and 200 to green, you need to:

```
MutableColorMap colorMap = new SimpleColorMap(new ClosedInterval(-200, 400),
    new InterpolatedPalette(new InterpolatedPalette.Entry(0, Color.red),
    new InterpolatedPalette.Entry(0.5, Color.white),
    new InterpolatedPalette.Entry(1, new Color(0, 128, 0)))
);
```

You can then customize it further by assigning special colors to values that fall outside of the range defined by the Interval by using the *setUnderColor()* and *setOverColor()* methods, or to missing values with the *setNullColor()* method.

## Adding a third dimension

The shapes can extend to the third dimension by specifying the column index or name containing their relative height:

```
treeMap.setHeight(2);
treeMap.setHeightByName("Strength");
```

In conjunction, the maximum relative height (as a percentage of the overall treemap size) of the shapes can be controlled through:

```
treeMap.getModel().getSettings().setMaximumHeight(0.05);
```

## Hierarchical grouping

Treemap visualization is a very effective method when used with hierarchical data. To define how each row of the original *TableModel* should be grouped and sub-grouped, the list of columns indices or names can be defined:

```
treeMap.setGroupBy(4, 5);
treeMap.setGroupByNames("Region", "Department");
```

Should your data have an unbalanced hierarchy, you can specify the hierarchal placement of each row in the *TableModel* by having a column of type *Path*.

## Labeling

Labels containing values from the original *TableModel* can be incorporated within each shape, either as a list where each value is displayed below one another:

```
treeMap.setLabels(1, 2);
treeMap.setLabelsByName("Value", "Strength");
```

and/or by filling the entire area with one value:

```
treeMap.setBackground(0);
treeMap.setBackgroundByName("Name");
```

## Layout Algorithms

*JIDE TreeMap* includes a wide range of treemap layout algorithm:

*BINARY\_TREE*: Uses a static binary tree layout

*SLICE*: Original slice-and-dice treemap algorithm, which has excellent stability properties but leads to high aspect ratios

*SQUARIFIED*: the aspect ratio of each rectangle is kept as close as possible to a square

*STRIP*: An ordered squarified treemap algorithm

*PIVOT\_BY\_SPLIT\_SIZE*: Pivot by split size

*CIRCULAR*: Circular treemap layout

The one to use can easily be set using the following method call:

```
treeMap.setAlgorithm(AlgorithmFactory.SQUARIFIED);
```

## Appearance and Rendering Options

Many options are provided to tune the appearance of the resulting treemap visualization. The most common is to define whether the shapes should be rendered with a cushion effect that highlight the hierarchical placement or using a solid color with an optional border:

```
treeMap.setRendering(RenderingFactory.CUSHION);
treeMap.setRendering(RenderingFactory.FLAT);
treeMap.setRendering(RenderingFactory.FLAT_NO_BORDER);
```

Another important customization area is to define how the headers of the various groups be displayed, for example by setting the placement of the header and font to use:

```
treeMap.setLabeling(LabelingFactory.TOP_LABELING);
treeMap.setHeaderFont(new Font("Tahoma", Font.BOLD, 16));
```

Similarly, the font to use to display the labels can be specified as well:

```
treeMap.setLabelingFont(new Font("Tahoma", Font.ITALIC, 18));
```

Finally, the color used for probing and selection can be customized using:

```
treeMap.getModel().getSettings().setProbingColor(new Color(200, 200, 0));
treeMap.getModel().getSettings().setSelectionColor(Color.orange);
```

## Customizing textual display

It is possible to fine tune the appearance and the positioning of the headers, labels, and tooltips by customizing their respective renderers: *TreeMapHeaderRenderer*, *TreeMapLabelRenderer*, and *TreeMapTooltipRenderer*. We provide default implementations for each of them that allow to customize, for example, the visual effect (drop shadow, glow), the vertical and horizontal alignment of the text, the minimum number of character to display, and how text should be truncated:

```
final DefaultTreeMapHeaderRenderer headerRenderer = new DefaultTreeMapHeaderRenderer();
headerRenderer.setEffect(DefaultTreeMapHeaderRenderer.Effect.Glow);
headerRenderer.setRendering(DefaultTreeMapHeaderRenderer.Rendering.Truncate);
headerRenderer.setHorizontalAlignment(DefaultTreeMapLabelRenderer.CENTER);
headerRenderer.setVerticalAlignment(DefaultTreeMapLabelRenderer.TOP);

treeMap.getView().setHeaderRenderer(headerRenderer);
```

The foreground, background and effect colors, as well as the font, have to be customized through the settings:

```
treeMap.getModel().getSettings().getDefaultFieldSettings().setHeaderForeground(new Color(156, 156, 156));
treeMap.getModel().getSettings().getDefaultFieldSettings().setHeaderEffectColor(new Color(50, 50, 50));
treeMap.getModel().getSettings().getDefaultFieldSettings().setHeaderBackground(new Color(96, 96, 96));

treeMap.setHeaderFont(new Font("Tahoma", Font.BOLD, 16));
```

Setting the appropriate Format to each column can specify how values are formatted:

```
treeMap.getModel().setFormat(3, new DecimalFormat("#,##0.00 $bil"));
```

## Tooltip content customization

The content of the tooltip can be changed by enabling/disabling the display of particular values and labels:

```
treeMap.getModel().getSettings().setShowPopup(treeMap.getModel().getTreeMapField(7), true);
treeMap.getModel().getSettings().getFieldSettings(treeMap.getModel().getTreeMapField(7)).setShowLabel(true);
```

## Interacting with JIDE TreMap

### Probing and selection

The selected nodes can be retrieved using *TreeMapModel.getSelection()*. The node currently under the mouse can be accessed using *TreeMapModel.getProping()*. It is possible to turn off multiple selection by using the *DefaultTreeMapController.setMultipleSelectionEnabled()* method.

## Extending JIDE TreeMap

### Context menu customization

The *TreeMapController* controls the content of the popup menu. It can be extended with additional entries using:

```
treemap.getController().getPopupMenu().insert(new JSeparator(), 0);
treemap.getController().getPopupMenu().insert(new AbstractAction("Do something") {
    public void actionPerformed(ActionEvent e) {
        // Place code to do something here
    }
}, 0);
```

## Handling dynamic data

There is a *DefaultTreeMapModel.setTableModel()* method that can be used to swap the current *TableModel*. Best is however to have the table model send appropriate events on data update instead of replacing the entire model, but both are supported. There is a demo (*DynamicPortfolioDemo*) that exemplifies both approaches. Changing the structure of the data, such as the number of columns, their names and types, is currently not supported.

## Scalability

*JIDE TreeMap* has been designed with performance in mind. Datasets containing 100'000s of data objects can be handled. To assess the effectiveness of the treemap view, one can display timing information using the *TreeMapView.setShowTiming(true)*.

Drawing text on screen is an expensive operation and this is the main consideration to take into account when faced with performance issues. Reducing the amount of labels to be displayed can typically solve this. Better yet is the possibility of progressively showing the labels in an iterative or incremental manner and without blocking the user interface. This mode, which is disabled by default, can be enabled using:

```
treemap.getView().setProgressive(TreeMapView.Progressive.Incremental)
```

Further speed improvements can be obtained by disabling anti-aliasing, the cushion effect, and the use of the 3<sup>rd</sup> dimension.

## What's Next

*JIDE TreeMap* is still in beta. There are still many features that we want to add to this product but haven't got a chance to do so.

- 1) Provide demo to demonstrate scalability of *JIDE TreeMap*.
- 2) Improve documentation for creating custom colormaps.
- 3) Add Voronoi treemap layout algorithm.
- 4) Include *TreeTable* wrapper.
- 5) Show how to encode *Path* information for single cell value.
- 6) Provide support for animated transitions.

If you have any feedbacks and suggestions, please feel free to email us or post on the forum.