KLARÄLVDALENS DATAKONSULT AB

Programmer's Manual

KD CHART 1.1

Table of Contents

List of Figures

Chapter 1. Introduction

KD Chart is Klarälvdalens Datakonsult AB's charting package for Qt applications. This is the KD Chart Programmer's Manual. It will get you started with creating your charts and provides lots of pointers to its many advanced features.

- Depending on your KD Chart version, you will find different `INSTALL` files that explain how to install KD Chart on your platform and a step by step description about how to build it from sources.
- KD Chart also comes with an extensive Reference Manual generated directly from the source code itself.

You should refer to it in conjunction with this Programmer's Manual.

- What is KD Chart?

  KD Chart is a tool for creating business and engineering charts, and is the most powerful Qt component of its kind. Besides having all the standard features, it also enables the developer to design and manage a large number of axes and provide sophisticated means of layout customization. Since all configuration settings have reasonable defaults you can usually get by with setting only a handful of parameters and relying on the defaults for the rest.

- What can we use KD Chart for?

  KD Chart is used by a variety of programs for many different purposes.

  The above example shows how KD Chart is used for visualizing flood events in a river; other samples on our web site at `http://www.kdab.net/kdchart` show how KD Chart is used for monitoring seismic activity. It is no coincidence that the current version of the KOffice productivity suite uses our library.

  * Display a view with small diagrams and arrows showing how the main classes work together

What You Should Know

You should be familiar with writing Qt applications, as well as have a working C++ knowledge. When you are in doubt about how a Qt4 class mentioned in this Programmer's Guide works, please check the Qt4 reference documentation or a good book about Qt4. A more in-depth introduction to the API can be found in the file `doc/KD-Chart-2.0-API-Introduction`. Also to browse KD Chart API Reference documentation start with this file: `doc/refman/index.html`.

The Structure of This Manual

How we will proceed to present KD Chart for Qt4?

This manual starts with an introduction to KD Chart2.0 API before going through the basic steps and methods for the user to create hers own chart.

The following Chapter 4 `Coordinate planes and Diagrams` will provide the reader with more details about the different chart types supported and the information you need to know in order to use KD Chart's features the best way.

The subsequent chapters contain more advanced customizing material like how to specify colors, fonts and other attributes if you don't want to use KD Chart's default settings. How to create and display headers and footers, legends and configure your chart axes.

Chapter 9 `Advanced Charting`, will guide you through KD Chart other more advanced features and describe the way to use them (frames and backgrounds, data values, axis and grid manipulations etc...). It will also show in details different interesting features like Interactive and Multiple charts or Zooming.

We provide you with lots of sample code combined with screen-shots that show the resulting display. We recommend our readers to try and run the sample code and experiment with the various settings.

What's next

In the next chapter we introduce you to KD Chart 2.0 new API.

Chapter 2. KD Chart 2 API Introduction

Version 2.0 of KD Chart fully supports and builds on the technologies introduced with Qt 4. The charting engine makes use of the Arthur and Scribe painting and text rendering frameworks to achieve high quality visual results. KD Chart 2.0 also integrates with the Interview framework for model/view separation and, much like Qt 4 itself, provides a convenience Widget class for those cases where that is too complex.

Overview

KD Chart 2.0 overall API strives for maximum consistency with the concepts and API style found in Qt 4. Of course, this means breaking source compatibility in several places, but like Trolltech, we have made a conscious decision that it would be better to clean up the API now, than to carry it with us into the next KD Chart generation.

Note

Wherever possible, compatibility methods and classes have been, or will be, provided.

The core of KD Chart's 2.0 API is the `KDChart::Chart` class. It encapsulates the canvas onto which the individual components of a chart are painted, manages them and provides access to them. There can be more than one `KDChart::Diagram` on a `KDChart::Chart`, how they are laid out is determined by which axes, if any, they share (more on axes further below).

`KDChart::Diagram` subclasses for the various types of charts are provided, such as `KDChart::PieDiagram`, and users can subclass `KDChart::AbstractDiagram` (or one of the other Diagram classes starting with Abstract, which are designed to be base classes) to implement custom chart types. A typical use of a simple BarDiagram looks like this:

Code Sample

```
using namespace KDChart;
......
BarDiagram *bars = new BarDiagram;
bars->setModel( &m_model );
chart->coordinatePlane()->replaceDiagram( bars );
.....
```

You could also use the following way, to prevent your compiler from complaining about instance bars not being free'ed:

```
BarDiagram *bars = static_cast <BarDiagram *>
(myChart->replaceDiagram( new BarDiagram ));
//now call the setter methods ...
```

In Chapter 3 `Basic steps: Create a Chart` we will make this somewhat abstract description more concrete by looking at some complete examples (Widget and Charts), which we recommend you to compile and run.

For now, in order for you to have an understanding about KD Chart 2.0 API and its features to know the following:

- Each diagram has an associated Coordinate Plane (Cartesian by default), which is responsible for the translation of data values into pixel positions. It defines the scale of the diagram, and all axes that are associated with it. This makes implementing diagram subclasses (types) much easier, since the drawing code can leave all of the coordinate calculation work to the coordinate plane.

- Each coordinate plane can have one or more diagram associated to it. Those diagrams will share the scale provided by the plan. Also a chart can have more than one coordinate plane. This makes it possible to have multiple diagrams (e.g a line and a bar chart) using the same or different scales and displayed next to, or on top of each other in the same chart.

- To share an axis among two planes (and also diagrams) we just need to add it to both diagrams. The Chart lay-outing engine will take care of adjusting positions accordingly.

A chart can also have a number of optional components such as Legends, Headers/Footers or custom `KDChart::Area` subclasses that implement user-defined elements. The API for manipulating these is similar for all of them.

To add an additional header for example, you may proceed as follow:

```
HeaderFooter * additionalHeader = new HeaderFooter;
additionalHeader->setPosition( NorthWest );
chart->addHeaderFooter( additionalHeader  );
```

We will explain further on how ownership of such components is handled (next section).

Finally and to conclude this overview, all classes in the KD Chart 2 API are in the "KDChart" namespace, to allow short and clear class names, while still avoiding name clashes. Unless you prefer to use the "KDChart::" prefix on all class names in your code, you can add the following line at the top of your implementation files, to make all names in the "KDChart" namespace available in that file:

```
using namespace KDChart;
```

Like Qt, KD Chart provides STL-style forwarding headers, allowing you to omit the ".h" suffix when including headers. To bring the bar diagram header into your implementation file, you could therefore write:

```
#include <KDChartBarDiagram>
or
#include <KDChartBarDiagram.h>
```

Note

File names of header and implementation files all have the "KDChart" prefix in the name. The definition of `KDChart::BarDiagram` is thus located in the file `KDChartBarDiagram.h`.

KD Chart and Interview

KD Chart 2.0 follows the Interview model/view paradigm introduced by Qt 4:

Any `KDChart::AbstractDiagram` subclass (since that is derived from `QAbstractItemView`) can display data coming from any `QAbstractItemModel`. In order to use your data with KD Chart diagrams, you need to either use one of Qt's builtin models to manage it, or provide the `QAbstractItemModel` interface on top of your already existing data storage by implementing your own model that talks to that underlying storage.

`KDChart::Widget` is a convenience class that provides a simpler, but less flexible way of displaying data in a chart. It stores the data it displays itself, and thus does not need a `QAbstractItemModel`. It should be sufficient for many basic charting needs but it is not meant to handle very large amounts of data or to make use of user-supplied chart types.

`KDChart::Widget` is provided in order to allow getting started easily without having to master the complexities of the new Interview framework in Qt 4. We would still advise to use KDChart::Chart so that you can make use of all the benefits that Interview brings you once you have mastered it.

To understand the relationship between `KDChart::View` and `KDChart::Widget` better, compare for example `KDChart::View` and `KDChart::Widget` to `QListView` and `QListWidget` in the Qt 4 documentation. You will clearly notice the parallels.

Code Sample

Let us make this more concrete by looking at the following lines of code where we are

using `QStandardItemModel` to store the data which will be displayed by the diagram in a `KDChartChart` widget.

```
// set up your model
m_model.insertRows( 0, 2, QModelIndex() );
m_model.insertColumns(  0,  3,  QModelIndex() );
for (int row = 0; row < 3; ++row) {
   for (int column = 0; column < 3; ++column) {
      QModelIndex index =
      m_model.index(row, column, QModelIndex());
      m_model.setData(index, QVariant(row+1 * column) );
   }
}
```

In order to assign the model above to your diagram and display it you would proceed as follow:

```
KDChart::BarDiagram* diagram = new KDChart::BarDiagram;
diagram->setModel(&m_model);
m_chart.coordinatePlane()->replaceDiagram(diagram);
```

Using `KDChartWidget` we would proceed as follow:

```
KDChartWidget widget;
QVector< double > vec0,  vec1;
vec0 << -5 << -4 << -3 << -2 << -1 << 0 ...;
vec1 << 25 << 16 << 9 << 4 << 1 << 0 ...;
widget.setDataset( 0, vec0, "Linear" );
widget.setDataset( 1, vec1, "Quadratic" );
widget.show();
```

We recommend you to consult `KDChartChart.h` and `KDChartWidget.h` to learn more about those classes and what they can do. Also compile and run the complete examples that describe very simply the two ways you may use to display a Chart.

Attribute sets

The various components of a chart such as legends or axes have attribute sets associated with them that define the way they are laid out and painted. For example, both the chart itself and all areas have a set of `KDChart::BackgroundAttributes`, which govern whether there should be a background pixmap, or a solid background color. Other attribute sets include FrameAttributes or GridAttributes. The default attributes provide reasonable, unintrusive settings, such as no visible background and no visible frame.

These attribute sets are passed by value, they are intended to be used much like Qt's QPen or QBrush. As shown below:

Code Sample

```
KDChart::TextAttributes ta;
ta.setPen( Qt::red );
ta.setFont( QFont( "Helvetica" ) );
chart->legend()->setTextAttributes( ta );
```

All attribute sets can be set per cell, per column or per model, and only be queried per cell. Access at the cell level only ensures that the proper fallback hierarchy can be observed. If there is a value set at cell level, it will be used, otherwise the dataset (column) level is checked. If nothing was set at dataset level, the model wide setting is used, and if there is none either, the default values will be applied. All of this happens automatically, so that the code using these values only has to ask the cell for its attributes, and will get the correct values. This avoids duplicating the fallback logic all over the library and the application, and avoids (expensive) error handling.

When using attributes sets, you need to be aware of this fallback hierarchy, because e.g. per-cell changes will hide per-column changes. (see files /src/KDChart*Attributes.h)

Memory management

As a general rule, everything in a KDChart::Chart is owned by the chart. Manipulation of the built-in components of a chart, such as for example a legend, happens through mutable pointers provided by the view, but those components can also be replaced.

Code Sample

Let us make this more concrete by looking at the following lines of code.

```
// set the built-in (default) legend visible
m_chart->legend()->setPosition( North );

// replace the default legend with a custom one
//the chart view will take ownership of the allocated
//memory and free the old legend
KDChart::Legend *myLegend =
m_chart->replaceLegend( new Legend );
```

Similarly, inserting new components into the view chart up their ownership. Note that the same procedure has to be applied for a diagram too.

```
// add an additional legend, chart takes ownership
chart->addLegend( Legend );
```

Removing a component does not de-allocate it. If you "take" a component from a chart or diagram, you are responsible for freeing it as appropriate.

(see files /src/{KDChartChart.h, KDChartLegend.h})

Notice how this pointer-based access to the components of a chart is different from the value-based usage of the attribute classes; the latter can be copied around freely, and are meant to be transient in your code; they will be copied internally as necessary. The reason for the difference, of course, is polymorphism.

What's next

Basic steps: Create a Chart or a Widget.

Chapter 3. Basic steps: Create a Chart

As specified in the above Chapter, there are two ways to create a chart:

- `KDChart::Widget` is providing a limited set of functions as can be seen in `KDChartWidget.h`. Its purpose is a convenient and simple way of getting a chart, for people who do not want to learn about the new Qt Interview concept or who do not care about more complicated details like the Coordinate Plane and other classes provided by KD Chart 2 API.

- `KDChart::Chart` purpose is to allow the user to use the full power of both the new Qt and the new KD Chart.

Basically, `KDChart::Widget` has been designed for starters, while `KDChart::Chart` is destinated for the experienced user and/or for users who need more features and flexibility. Once again we recommend you to read both interfaces for those classes in order to make yourself an opinion about what you would need while developping your application.(See `KDChartWidget.h` and `KDChartChart.h`).

Prerequisite

As described above (Section KD Chart and Interview) a prerequisite for using KD Chart's full API is that the data to be charted is provided by you through a class implementing the `QAbstractItemModel` interface. Before looking at code lines, let us present you a few top level classes of the KD Chart 2 API:

- The "chart" is the central widget acting as a container for all of the charting elements, including the diagrams themselves, its class is called `KDChart::Chart`.

  A "chart" can hold several coordinate planes (e.g cartesian and polar coordinates are supported at the moment) each of which can hold several diagrams.

- The "coordinate plane" (often called the "plane") represents the entity that is responsible for mapping the values into positions on the widget. The plane is also showing the (sub-)grid lines. There can be several planes per chart.

- The "diagram" is the actual plot (bars, lines and other chart types) representing the data.There can be several diagrams per coordinate plane.

The Procedure

Let us go through the general procedure for creating a chart, without taking care about the details. We will then build complete example and create a small application displaying a chart using `KDChartWidget` and `KDChartChart` respectively.

First of all we need to include the appropriate headers, and bring in the "KDChart" namespace:

```
#include <KDChartChart>
#include <KDChartLineDiagram>
using namespace KDChart;

//Add the widget to your layout like for any other QWidget:
QHBoxLayout* chartLayout = new QHBoxLayout( chartFrame );
m_chart = new Chart();
chartLayout->addWidget( m_chart );
```

In this example, we will create a single line diagram, and use the default Cartesian coordinate plane, which is already contained in an empty Chart.

```
// Create a line diagram and associate the data model to it
m_lines = new LineDiagram();
m_lines->setModel( &m_model );

// Replace the default diagram of the default coordinate
// plane with your newly created one.
// Note that the plane takes ownership of the diagram,
// so you are not allowed to delete it.
m_chart->coordinatePlane()->replaceDiagram( m_lines );
```

Adding elements such as axes or legends is straightforward as well:

```
CartesianAxis *yAxis = new CartesianAxis ( m_lines );
yAxis->setPosition ( KDChart::CartesianAxis::Left );

// the diagram takes ownership of the Axis
m_lines->addAxis( yAxis );

legend = new Legend( m_lines, m_chart );
m_chart->addLegend( legend );
```

You can adjust and fine-tune various aspects of the diagrams, planes, legends, etc...

Much like Qt itself, KD Chart uses a value-based approach to these attributes. In the case of diagrams, most aspects can be adjusted at different levels of granularity. The QPen that is used for drawing datasets (lines, bars, etc...) can be set either for one datapoint within a dataset, for a dataset or for the whole diagram. See file `KDChartAbstractDiagram.h`:

```
void setPen( const QModelIndex& index, const QPen& pen );
void setPen( int dataset, const QPen& pen );
void setPen( const QPen& pen );
```

To use a dark gray color for all lines in your example chart, you would write:

```
QPen pen;
pen.setColor( Qt::darkGray );
pen.setWidth( 1 );
m_lines->setPen( pen );
```

Attributes that form logical groupings are combined into collection classes, such as `GridAttributes`, `DataValueAttributes`, `TextAttributes`, etc....

This makes it possible to keep sets of such properties around and swap them in one step, based on program state. However, you might often want to adjust just one or a few of the default settings, rather than specifying a complete new set. Thus in most cases, using the copy constructor of the settings class might be appropriate, so to use a special font for drawing a legend, for example, you would just write:

```
TextAttributes ta( legend->textAttributes() );
ta.setFont( myfont );
legend->setTextAttributes( ta );
```

We will continue with more examples and more detailed information about all those points in the next sections and the next chapters. Also we recommend you to consult and run the examples sent together with your KD Chart distribution package.

Two ways

We will now go through the basic steps for creating a simple chart widget using first `KDChart::Widget` and then `KDChart::Chart`, that will give us an overview about how to proceed in both cases.

Widget Example

We recommend you to read, compile and run the following example. It is available at the following location of your KD Chart installation: `examples/Widget/Simple`.

```
 1                        /****************************************************************
    ** Copyright (C) 2006 Klarälvdalens Datakonsult AB.  All rights reserved.
    **
 5  ** This file is part of the KD Chart library.
    **
    ** This file may be distributed and/or modified under the terms of the
    ** GNU General Public License version 2 as published by the Free Software
    ** Foundation and appearing in the file LICENSE.GPL included in the
10  ** packaging of this file.
    **
    ** Licensees holding valid commercial KD Chart licenses may use this file in
    ** accordance with the KD Chart Commercial License Agreement provided with
    ** the Software.
```

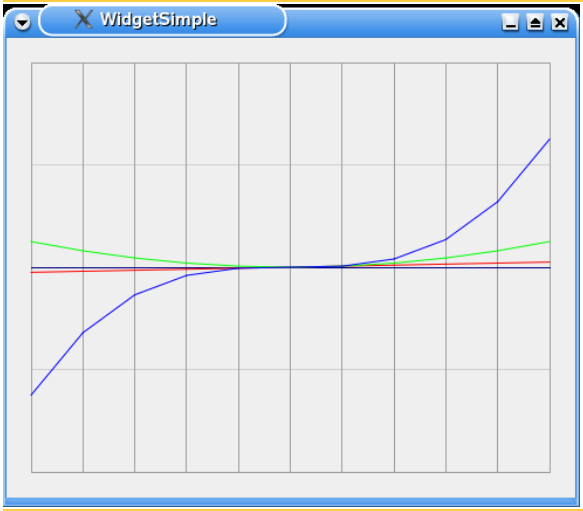
```
15  **
    ** This file is provided AS IS with NO WARRANTY OF ANY KIND, INCLUDING THE
    ** WARRANTY OF DESIGN, MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
    **
    ** See http://www.kdab.net/kdchart for
20  **   information about KDChart Commercial License Agreements.
    **
    ** Contact info@kdab.net if any conditions of this
    ** licensing are not clear to you.
    **
25  **********************************************************************/

   #include <QApplication>
   #include <KDChartWidget>

30
   using namespace KDChart;

   int main( int argc, char** argv ) {
       QApplication app( argc, argv );
35
       Widget widget;
       widget.resize( 600, 600 );

       QVector< double > vec0,  vec1,  vec2;
40
       vec0 << -5 << -4 << -3 << -2 << -1 << 0
            << 1 << 2 << 3 << 4 << 5;
       vec1 << 25 << 16 << 9 << 4 << 1 << 0
            << 1 << 4 << 9 << 16 << 25;
45     vec2 << -125 << -64 << -27 << -8 << -1 << 0
            << 1 << 8 << 27 << 64 << 125;

       widget.setDataset( 0, vec0, "Linear" );
       widget.setDataset( 1, vec1, "Quadratic" );
50     widget.setDataset( 2, vec2, "Cubic" );

       widget.show();

       return app.exec();
55 }
```

The result of the the code above will display the very simple widget presented in the screen-shot below.

As we can see the code code is straight forward:

- Include the headers and bring the Chart namespace.
- Declare your `KDChartWiget`
- Use a `QVector` to store the data to be displayed.
- Assign the stored data to the widget, using one of the available `setDataset` method to do so.

Figure 3.1. A Simple Widget



Of course it is possible to add new elements like Title, Headers, Footers, Legends or Axes ...etc to this very simple widget as we will see later on more in details. Notice also that the default diagram displayed by a `KDChartWidget` is a `KDChartLineDiagram`. In the following example we will look at the way to display a Chart widget using `KDChartChart`.

Chart Example

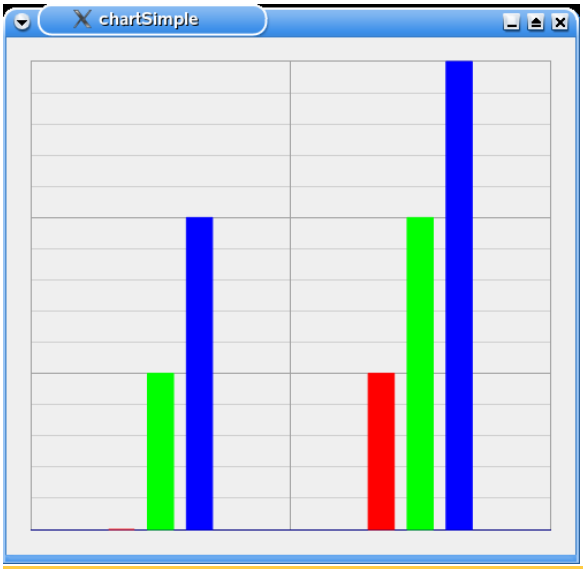
We recommend you to read, compile and run the following example. It is available at the following location of your KD Chart installation: `/examples/Charts/simple`

```
 1
                      #include <QtGui>
   #include <KDChartChart>
   #include <KDChartBarDiagram>
 5
   class ChartWidget : public QWidget {
     Q_OBJECT
   public:
10   explicit ChartWidget(QWidget* parent=0)
       : QWidget(parent)
     {

       m_model.insertRows( 0, 2, QModelIndex() );
15     m_model.insertColumns(  0,  3,  QModelIndex() );
       for (int row = 0; row < 3; ++row) {
               for (int column = 0; column < 3; ++column) {
                   QModelIndex index = m_model.index(row, column, QModelIndex());
                   m_model.setData(index, QVariant(row+1 * column) );
20             }
       }

       KDChart::BarDiagram* diagram = new KDChart::BarDiagram;
       diagram->setModel(&m_model);
25
       m_chart.coordinatePlane()->replaceDiagram(diagram);
```

```
        QVBoxLayout* l = new QVBoxLayout(this);
        l->addWidget(&m_chart);
30      setLayout(l);
    }

  private:
    KDChart::Chart m_chart;
35  QStandardItemModel m_model;
  };

  int main( int argc, char** argv ) {
      QApplication app( argc, argv );
40
      ChartWidget w;
      w.show();

      return app.exec();
45 }

  #include "main.moc"
```

In this example we are making use of a `QStandardItemModel` to insert and store the data to be displayed by the diagram. We are also implicitely using a `KDChartBarDiagram` to which we assign the model. See below the resulting chart widget displayed by this implementation.

Figure 3.2. A Simple Chart



We can of course add more elements to this chart and change its default attributes as described above in the section intituled `The Procedure`.

We will see more in details how to configure those attributes (Pen, Color, etc ...)and add the divers elements (Axes, Legend, Headers etc...) further on.

What's next

In the next chapter we describe the different chart types (diagrams) available and their coordinate planes. For each chart type we will study the attributes available for this special type and give a few example to make it clear.

Chapter 4. Planes and Diagrams

KD Chart supports at the moment two types of plane in order to display the different types of diagrams it supports.

- A Cartesian coordinate plane, formed by a horizontal axis and a vertical axis, often labeled the x-axis and y-axis.
- A Polar coordinate plane which make use of the radius or the polar angle, that define the position of a point in a plane.

  This chapter tells you how to change the chart type from the default to any one of the other types. All of the chart types provided by KD Chart are presented here with the help some sample code and/or small programs and their screen shots.

  It will also give us an idea about which chart type might be appropriate for a specific purpose and provides information on the features that are available for each type of chart.

Cartesian coordinate plane

KD Chart uses the Cartesian coordinate system, and in particular its `KDChart::CartesianCoordinatePlane` class for displaying chart types like (e.g: Lines, Bars, Points, etc...).

In this section we will describe and present each chart type which uses the default Cartesian coordinate plane.

In general to implement a particular type of chart, just create an object of this type by calling `KDChart[type]Diagram`, or if your are using `KDChartWidget` you will need to call its `setType()` and specify the appropriate chart type. (e.g Widget::Bar, Widget::Line etc...)

Bar Charts

Tip

Bar charts are the most common type of charts and can be used for visualizing almost any kind of data. Like the Line Charts, the bar charts can be the ideal choice to compare multiple series of data.

A good example for using a bar chart would be a comparison of the sales figures in different departments, perhaps accompanied by a High/Low Chart showing each day's key figures.

Your Bar Chart can be configured with the following (sub-)types as described in details in the following sections:

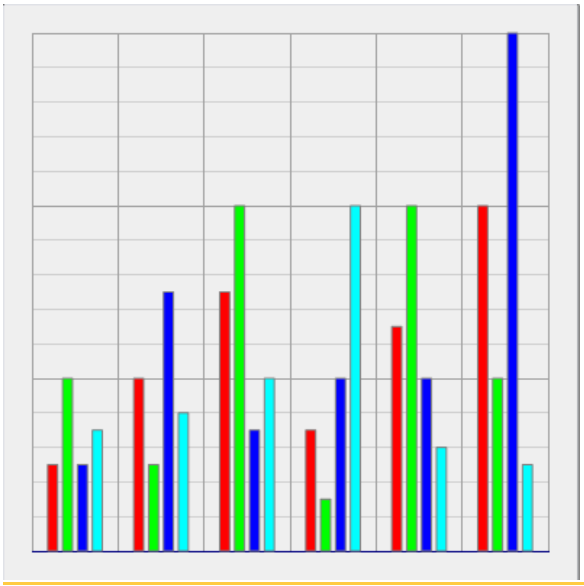
- Normal
- Stacked
- Percent

Normal Bar Charts

Tip

In a normal bar chart, each individual value is displayed as a bar by itself. This flexibility allows to compare both the values in one series and values of different series.

Figure 4.1. A Normal Bar Chart



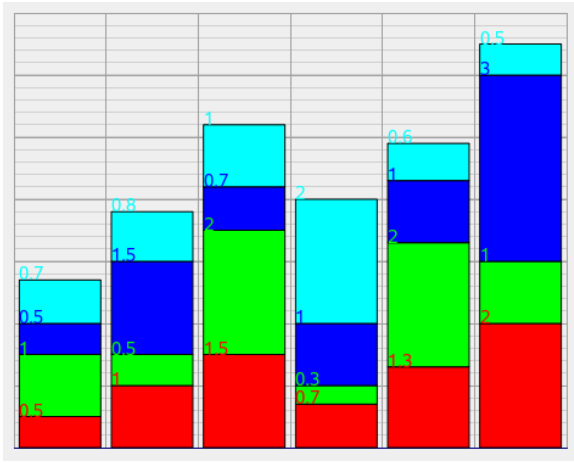
KD Chart's default type is the normal bar chart so no method needs to be called to get one, however after having used your `KDChartBarDiagram`. To display another sub-type you can return to the normal one by calling `setType( Normal )`.

Stacked Bar Charts

Tip

Stacked bar charts focus on comparing the sums of the individual values in each data series, but also show how much each individual value contributes to its sum.

Figure 4.2. A Stacked Bar Chart



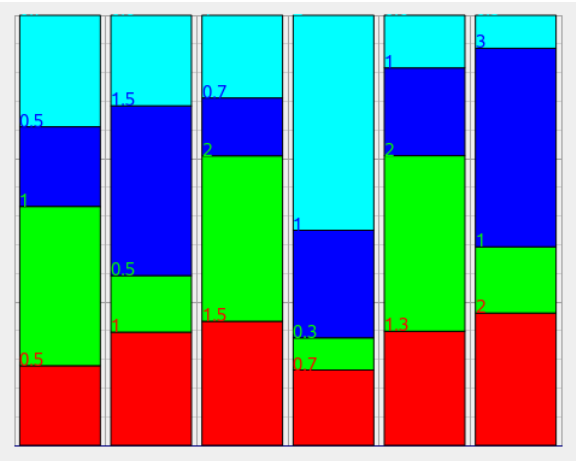
Stacked mode for bar charts is activated by calling the `KDChartBarDiagram` function `setType( Stacked )`.

Percent Bar Charts

Tip

Unlike stacked bar charts, percent bar charts are not suitable for comparing the sums of the data series, but rather focus on the respective contributions of their individual values.

Figure 4.3. A Percent Bar Chart



Percent: Percentage mode for bar charts is activated by calling the `KDChartBarDiagram` function `setType( Percent )`.

Note

Three-dimensional look of the bars is no special feature you can enable it for all types (`Normal`, `Stacked` or `Percent`) by setting its ThreeD attributes, we will describe that in the "Bars Attributes" section further on.

Code Sample

For now let us make the above description more concrete by looking at the following code sample based on the `Simple Widget` example we have been demonstrating above. In this example we show how to configure your bar diagram and change its attributes when working with a `KDChartWidget`.

First include the appropriate headers and bring in the "KDChart namespace":

```
#include <QApplication>
#include <KDChartWidget>
#include <KDChartBarDiagram>
#include <QPen>

using namespace KDChart;
```

We need to include `KDChartBarDiagram` in order to be able to configure some of its attributes as we will see further on.

```
int main( int argc, char** argv ) {
QApplication app( argc, argv );
```

```
Widget widget;
// our Widget can be configured
// as any Qt Widget
widget.resize( 600, 600 );
// store the data and assign it
QVector< double > vec0,  vec1;
vec0 << 5 << 4 << 3 << 2 << 1 << 0
<< 1 << 2 << 3 << 4 << 5;
vec1 << 25 << 16 << 9 << 4 << 1 << 0
<< 1 << 4 << 9 << 16 << 25;
widget.setDataset( 0, vec0, "vec0" );
widget.setDataset( 1, vec1, "vec1" );
```

We want to change the default line chart type to a bar chart type. In this case we also want to display it in stacked mode. `KDChartWidget` with its `setType` and `setSubType` methods allow us to achieve that the easy way.

```
widget.setType( Widget::Bar , Widget::Stacked );
```

The default type being Normal type for the widget, we need to implicitely pass the second parameter when calling `KDChartWidget::setType()` We can also change the sub-type of our bar chart further on by calling for example `setSubType( Widget::Percent )`.

```
//Configure a pen and draw a line
//surrounding the bars
QPen pen;
pen.setWidth( 2 );
pen.setColor(  Qt::darkGray );
// call your diagram and set the new pen
widget.barDiagram()->setPen(  pen );
```

In the above code our intention is to draw a gray line around the bars to make it nicer. That is what we call configuring the attributes in a diagram. To do so we configure a QPen and then assign it to our diagram. `KDChartWidget::barDiagram()` allow us to get a pointer to our widget diagram. As you can see it is very simple to assign a new pen to our diagram by calling the diagram `KDChartAbstractDiagram::setPen()` method.

```
//Set up your ThreeDAttributes
//display in ThreeD mode
ThreeDBarAttributes td;
td.setDepth(  15 );
td.setEnabled(  true );
widget.barDiagram()->setThreeDBarAttributes( td );
```

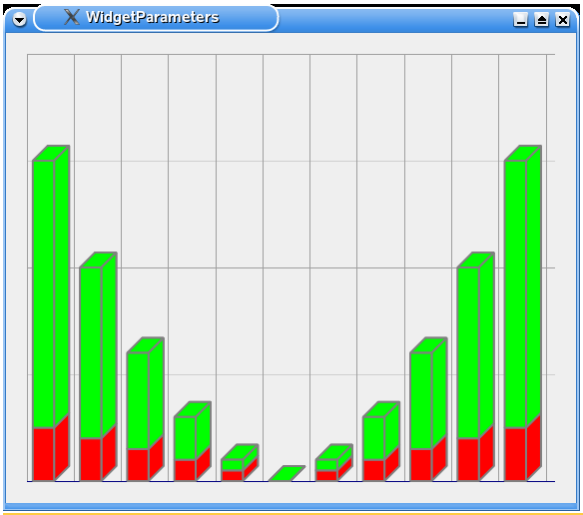
We want our bar chart to be displayed in ThreeD mode and need to configure some ThreeDBarAttributes and assing them to our diagram. Here we are configuring the depth of the ThreeD bars and enabling ThreeD mode. Depth is an attribute specific to Bar charts and its setter and getter are implemented into the `KDChartThreeDBarAt-`

tributes, when the KDChartAbstractThreeDAttributes::setEnabled() is a generic attributes available to all chart types. Both of those attribute are set very simply but are implemented at different level for a better code structure.

```
                widget.show();

                return app.exec();
                }
```

See the screen-shot below to view The resulting chart displayed by the above code.

Figure 4.4. A Simple Bar ChartWidget



This example can be compiled and run from the following location of your KD Chart installation examples/Widget/Parameters

Note

Configuring the attributes for a KDChartBarDiagram making use of a KDChartChart is done the same way as for a KDChartWidget. You just need to assign the configured attributes to your bar diagram and assign it to the chart by calling KDChartChart::replaceDiagram().

Bars Attributes

By "Bars attributes" we are talking about all parameters that can be configured and set by the user and which are specifics to the Bar Chart type. The "getters" and "setters" for

those attributes can be consulted by looking at `KDChartBarAttributes.h` to get an idea about what can be configured there.

Note

KD Chart 2.0 API separates the attributes specifics to a chart type itself and the generic attributes which are common to all chart types as for example the setters and getters for a brush or a pen and that are accessible from the `KDChartAbstractDiagram` interface.

All those attributes have a reasonnable default value that can simply be modified by the user by calling one of the diagram set function implemented on this purpose `KDChartBarDiagram::setBarAttributes()` or for example (to change the default Pen) directly by calling the `KDChartAbstractDiagram::setPen()` method.

The procedure is straight forward on both cases. Let us discuss the types specifics attributes first:

- Create a `KDChart::BarAttributes` object by calling `KDChartBarDiagram::barAttributes`.
- Configure this object using the setters available.
- Assign it to your Diagram with the help of one of the setters available in `KDChart::BarDiagram`. All the attributes can be configured to be applied for the whole diagram, for a column, or at a specified index (`QModelIndex`).

KD Chart 2.0 supports the following attributes for the Bar chart type. Each of those attributes can be set and retrieved the way we describe it in our example below:

- BarWidth: Specifies the width of the bars
- GroupGapFactor: Configure the gap between groups of bars.
- BarGapFactor: Configure the gap between Bars within a group
- DrawSolidExcessArrow: Specify whether the arrows showing excess values should be drawn solidly or split.

Bar Attributes Sample

Let us make this more concrete by looking at the following sample code that describes the above process. We recommand you to compile and run the following example which is located into the `examples/Bars/Parameters` directory of your KD Chart installation.

First of all we are including the header files we need and bring KD Chart namespace.

```
#include <QtGui>
#include <KDChartChart>
#include <KDChartBarDiagram>
#include <KDChartDataValueAttributes>

using namespace KDChart;
```

We have included `KDChartDataValueAttributes` to be able to display our data values. Those attributes are of course used by all types of charts and are not specifical to the Bar diagrams.

In this example we are using a `KDChartChart` class as well as a `QStandardItemModel` in order to store the data which will be assigned to our diagram

```
class ChartWidget : public QWidget {
Q_OBJECT
public:
explicit ChartWidget(QWidget* parent=0)
: QWidget(parent)
{

m_model.insertRows( 0, 2, QModelIndex() );
m_model.insertColumns(  0,  3,  QModelIndex() );
for (int row = 0; row < 3; ++row) {
  for (int column = 0; column < 3; ++column) {
    QModelIndex index = m_model.index(row, column, QModelIndex());
    m_model.setData(index, QVariant(row+1 * column) );
  }
}

BarDiagram* diagram = new KDChart::BarDiagram;
diagram->setModel(&m_model);
```

After having store our data into the model, we create a diagram, in this case, we want to display a `KDChartBarDiagram` and assing the model to our diagram. The procedure is of course similar for all types of diagrams.

We are no ready to configure our bar specifics attributes using a `KDChartBarAttributes` to do so.

```
BarAttributes ba;
//set the bar width and
//implicitely enable it
ba.setFixedBarWidth( 500 );
ba.setUseFixedBarWidth( true );
//configure gab between values
//and blocks
ba.setGroupGapFactor( 0.50 );
ba.setBarGapFactor( 0.125 );

//assign to the diagram
diagram->setBarAttributes(  ba );
```

We want to configure our bars width so that they get displayed a bit larger. The Width of a bar is calculated automatically depending on the gaps between each bar and the gaps between groups of bars as well as the space available horizontally in the plane. So those values interact with each other so that your bars does not exceed the plane surface horizontally. Here we are increasing the value of my bars width and at the same time set some lower values for the gaps. Which will give us larger bars

Note

After having configured our attributes we need to assign the `BarAttributes` object to the diagram. This can be done for the whole diagram, at a specific index or for a column. See `KDChartBarDiagram.h` and look at the methods available there to find out those setters and getters.

We will now display the data values related to each bar making use of KD Chart 2.0 API `KDChartDataValueAttributes`. Those attributes are not specifics to the Bar Chart types but can be used by any type of charts. The procedure is very similar.

```
// display the values
   DataValueAttributes dva;
   TextAttributes ta = dva.textAttributes();
   //rotate if you wish
   //ta.setRotation( 0 );
   ta.setFont( QFont( "Comic", 9 ) );
   ta .setPen( QPen( QColor( Qt::darkGreen ) ) );
   ta.setVisible( true );
   dva.setTextAttributes( ta );
   dva.setVisible( true );
   diagram->setDataValueAttributes( dva );
```

We could have displayed the data values without caring about settings its `KDChartTextAttributes`, but we wanted to do so in order to demonstrate this feature too. Notice that you have to implicitely enable your attributes (DataValue and Text) by calling their `setVisible()` methods. After it is configured as we want it is just to assign to the diagram as for all other attributes.

Finally I want to paint a ligne around one of the datasets bars. In order to keep the attention of the public on this specific set of data. To do so I need to change the default pen used by my bars for this data set exclusively. Of course we could also have changed the pen for all datasets or for a specifical index or value.

```
 //draw a surrounding line around bars
QPen linePen;
linePen.setColor( Qt::magenta );
linePen.setWidth( 4 );
linePen.setStyle( Qt::DotLine );
//draw only around a dataset
//to draw around all the bars
// call setPen( myPen );
diagram->setPen( 1,  linePen );
```

Note

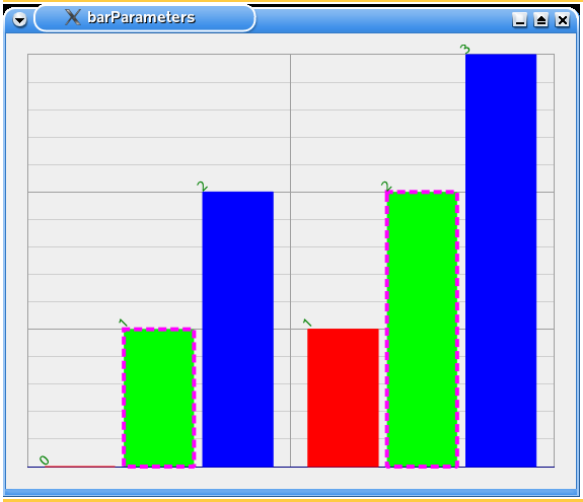
The Pen and the Brush setters and getters are implemented at a lower level in our `KDChartAbstractDiagram` class for a cleaner code structure. Those methods are of course used by all types of diagram and their configuration is very simple and straight forward as you can see in the above sample code. Create a Pen, configure it, call one of the setters methods available (See `KDChartAbstractDiagram.h` about those methods).

Our attribute having been configured and assigned we just need to assign the Bar diagram to our chart and conclude the implementation.

```
                m_chart.coordinatePlane()->replaceDiagram(diagram);

                QVBoxLayout* l = new QVBoxLayout(this);
                l->addWidget(&m_chart);
                setLayout(l);
                }

                private:
                Chart m_chart;
                QStandardItemModel m_model;
                };

                int main( int argc, char** argv ) {
                QApplication app( argc, argv );

                ChartWidget w;
                w.show();

                return app.exec();
                }

                #include "main.moc"
```

The above procedure can be applied to any of the supported attributes relative to the chart types. The resulting display of the code we have gone through can be seen in the following screen-shot. We also recommend you to compile and run the example related to this section and located in the `examples/Bars/Parameters` directory of your KD Chart installation.

Figure 4.5. Bar with Configured Attributes



The subtype of a bar chart (Normal, Stacked or Percent) is not set via its attribute class, but directly by using the diagram `KDChartBarDiagram::setType` method.

Note

ThreeDAttibutes for the different chart types are implemented has an own class, the same way as for the other attributes. We will talk more in details about KD Chart 2.0 ThreeD features in the ThreeD section, Chapter 5 - Customizing your Chart.

Tips and Tricks

In this section we want to give you some example about how to use some interesting features offered by the KD Chart 2.0 API. We will study the code and display a screen-shot showing the resulting widget.

A complete Bar Example

In the following implementation we want to be able to:

- Display the data values.
- Change the bar chart subtype (Normal, percent, Stacked).
- Select a column and mark it by changing the generic pen attributes.
- Display in ThreeD mode and change the Bars depth dynamically.

- Change the Bars width dynamically.

To do so we will need to use several types of attributes. Generics one available to all chart types (e.g `KDChartAbstractDiagram::setPen()`, `KDCHartDataValueAttributes` and `KDChartTextAttributes` as well as typical bar attributes only applyable to the Bar types as `KDChartBarAttributes::setWidth()` or `KDChartThreeDBarAttributes`

We are making use of a `KDChartChart` class and also of an home made `TableModel` for the convenience and derived from `QAbstractTableModel`.

TableModel uses a simple rectangular vector of vectors to represent a data table that can be displayed in regular Qt Interview views. Additionally, it provides a method to load CSV files exported by OpenOffice Calc in the default configuration. This allows to prepare test data using spreadsheet software.

It expects the CSV files in the subfolder ./modeldata. If the application is started from another location, it will ask for the location of the model data files.

We recommend you to consult the "TableModel" interface and implementation files which are located in the `examples/tools` directory of your KD Chart installation.

Let us concentrate on our Bar chart implementation for now and consult the following files: other needed files like the ui, pro , qrc ,CSV and main.cpp files can be consulted from the `examples/Bars/Advanced` directory of your installation.

```
 1
                      /****************************************************************
    ** Copyright (C) 2006 Klarälvdalens Datakonsult AB.  All rights reserved.
    **
 5  ** This file is part of the KD Chart library.
    **
    ** This file may be distributed and/or modified under the terms of the
    ** GNU General Public License version 2 as published by the Free Software
    ** Foundation and appearing in the file LICENSE.GPL included in the
10  ** packaging of this file.
    **
    ** Licensees holding valid commercial KD Chart licenses may use this file in
    ** accordance with the KD Chart Commercial License Agreement provided with
    ** the Software.
15  **
    ** This file is provided AS IS with NO WARRANTY OF ANY KIND, INCLUDING THE
    ** WARRANTY OF DESIGN, MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
    **
    ** See http://www.kdab.net/kdchart for
20  **   information about KDChart Commercial License Agreements.
    **
    ** Contact info@kdab.net if any conditions of this
    ** licensing are not clear to you.
    **
25  **********************************************************************/

   #ifndef MAINWINDOW_H
   #define MAINWINDOW_H

30 #include "ui_mainwindow.h"
   #include <TableModel.h>

   namespace KDChart {
```

```
        class Chart;
35      class BarDiagram;
    }

    class MainWindow : public QWidget, private Ui::MainWindow
    {
40      Q_OBJECT

    public:
        MainWindow( QWidget* parent = 0 );

45  private slots:

        void on_barTypeCB_currentIndexChanged( const QString & text );
        void on_paintValuesCB_toggled( bool checked );
        void on_paintThreeDBarsCB_toggled( bool checked );
50      void on_markColumnCB_toggled( bool checked );
        void on_markColumnSB_valueChanged( int i );
        void on_threeDDepthCB_toggled( bool checked );
        void on_depthSB_valueChanged( int i );
        void on_widthCB_toggled( bool checked );
55      void on_widthSB_valueChanged( int i );

    private:
        KDChart::Chart* m_chart;
        KDChart::BarDiagram* m_bars;
60      TableModel m_model;
    };


    #endif /* MAINWINDOW_H */
65
```

In the above code we bring up the KDChart namespace as usual and declare our slots. The prupose is to let the user configure its bar chart attributes manually . As you can see we are using a KDChartChart object (m_chart), a KDChartBarDiagram object (m_bars), and our home made TableModel (m_model).

The implementation is also straight forward as we will see below:

```
 1
                        /****************************************************************
     ** Copyright (C) 2006 Klarälvdalens Datakonsult AB.  All rights reserved.
     **
 5   ** This file is part of the KD Chart library.
     **
     ** This file may be distributed and/or modified under the terms of the
     ** GNU General Public License version 2 as published by the Free Software
     ** Foundation and appearing in the file LICENSE.GPL included in the
10   ** packaging of this file.
     **
     ** Licensees holding valid commercial KD Chart licenses may use this file in
     ** accordance with the KD Chart Commercial License Agreement provided with
     ** the Software.
15   **
     ** This file is provided AS IS with NO WARRANTY OF ANY KIND, INCLUDING THE
     ** WARRANTY OF DESIGN, MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
     **
     ** See http://www.kdab.net/kdchart for
20   **   information about KDChart Commercial License Agreements.
     **
     ** Contact info@kdab.net if any conditions of this
     ** licensing are not clear to you.
     **
```

```
**********************************************************************/

#include "mainwindow.h"

#include <KDChartChart>
#include <KDChartDatasetProxyModel>
#include <KDChartAbstractCoordinatePlane>
#include <KDChartBarDiagram>
#include <KDChartTextAttributes>
#include <KDChartDataValueAttributes>
#include <KDChartThreeDBarAttributes>


#include <QDebug>
#include <QPainter>

using namespace KDChart;

MainWindow::MainWindow( QWidget* parent ) :
    QWidget( parent )
{
    setupUi( this );

    QHBoxLayout* chartLayout = new QHBoxLayout( chartFrame );
    m_chart = new Chart();
    chartLayout->addWidget( m_chart );

    m_model.loadFromCSV( ":/data" );

    // Set up the diagram
    m_bars = new BarDiagram();
    m_bars->setModel( &m_model );

    QPen pen(  m_bars->pen() );
    pen.setColor( Qt::darkGray );
    pen.setWidth( 1 );
    m_bars->setPen( pen );
    m_chart->coordinatePlane()->replaceDiagram( m_bars );
}


void MainWindow::on_barTypeCB_currentIndexChanged( const QString & text )
{
    if ( text == "Normal" )
        m_bars->setType( BarDiagram::Normal );
    else if ( text == "Stacked" )
        m_bars->setType( BarDiagram::Stacked );
    else if ( text == "Percent" )
        m_bars->setType( BarDiagram::Percent );
    else
        qWarning (" Does not match any type");

    m_chart->update();
}


void MainWindow::on_paintValuesCB_toggled( bool checked )
{
    Q_UNUSED( checked );
    // We set the DataValueAttributes on a per-column basis here,
    // because we want the texts to be printed in different
    // colours - according to their respective dataset's colour.
    const QFont font(QFont( "Comic", 10 ));
    const int colCount = m_bars->model()->columnCount();
    for ( int iColumn = 0; iColumn<colCount; ++iColumn ) {
        QBrush brush( m_bars->brush( iColumn ) );
        DataValueAttributes a( m_bars->dataValueAttributes( iColumn ) );
        TextAttributes ta( a.textAttributes() );
        ta.setRotation( 0 );
```

```
            ta.setFont( font );
            ta .setPen( QPen( brush.color() ) );
            if ( checked )
                ta.setVisible( true );
            else
                ta.setVisible( false );

            a.setTextAttributes( ta );
            a.setVisible( true );
            m_bars->setDataValueAttributes( iColumn, a);
        }

        m_chart->update();
    }

    void MainWindow::on_paintThreeDBarsCB_toggled( bool checked )
    {
        ThreeDBarAttributes td( m_bars->threeDBarAttributes() );
        double defaultDepth = td.depth();
        if ( checked ) {
            td.setEnabled( true );
            if ( threeDDepthCB->isChecked() )
                td.setDepth( depthSB->value() );
            else
                td.setDepth( defaultDepth );
        } else {
            td.setEnabled( false );
        }
        m_bars->setThreeDBarAttributes( td );
        m_chart->update();
    }

    void MainWindow::on_markColumnCB_toggled( bool checked )
    {
        const int column = markColumnSB->value();
        QPen pen( m_bars->pen( column ) );
        if ( checked ) {
            pen.setColor( Qt::yellow );
            pen.setStyle( Qt::DashLine );
            pen.setWidth( 3 );
            m_bars->setPen( column, pen );
        }  else {
            pen.setColor( Qt::darkGray );
            pen.setStyle( Qt::SolidLine );
            pen.setWidth( 1 );
            m_bars->setPen( column, pen );
        }
        m_chart->update();
    }

    void MainWindow::on_depthSB_valueChanged( int i )
    {
         Q_UNUSED( i );

        if ( threeDDepthCB->isChecked() && paintThreeDBarsCB->isChecked() )
            on_paintThreeDBarsCB_toggled( true );
    }

    void MainWindow::on_threeDDepthCB_toggled( bool checked )
    {
         Q_UNUSED( checked );

        if ( paintThreeDBarsCB->isChecked() )
            on_paintThreeDBarsCB_toggled( true );
    }

    void MainWindow::on_markColumnSB_valueChanged( int i )
    {
        QPen pen( m_bars->pen( i ) );
```

```
165     markColumnCB->setChecked( pen.color() == Qt::yellow );
    }

    void MainWindow::on_widthSB_valueChanged( int value )
    {
170     if (  widthCB->isChecked() ) {
            BarAttributes ba( m_bars->barAttributes() );
            ba.setFixedBarWidth( value );
            ba.setUseFixedBarWidth( true );
            m_bars->setBarAttributes( ba  );
175     }
        m_chart->update();
    }

    void MainWindow::on_widthCB_toggled( bool checked )
180 {
        if (  checked ){
            on_widthSB_valueChanged( widthSB->value() );
        }else{
            BarAttributes ba( m_bars->barAttributes() );
185         ba.setUseFixedBarWidth( false );
            m_bars->setBarAttributes( ba  );
            m_chart->update();
        }
    }
190
```

First of all we are adding our chart to the layout as for any other Qt widget. Load the data to be display into our model, and assign the model to our bar diagram. We also want to configure a Pen and surround the displayed bars by a darkGray line to make it somewhat nicer. Finally we assign the diagram to our chart.

```
                    //draw a surrounding line around bars
                   QHBoxLayout* chartLayout = new QHBoxLayout( chartFrame );
                    m_chart = new Chart();
                    chartLayout->addWidget( m_chart );

                    m_model.loadFromCSV( ":/data" );

                    // Set up the diagram
                    m_bars = new BarDiagram();
                    m_bars->setModel( &m_model );

                    QPen pen;
                    pen.setColor( Qt::darkGray );
                    pen.setWidth( 1 );
                    m_bars->setPen( pen );

                    m_chart->coordinatePlane()->replaceDiagram( m_bars );
```

The user should be able to change the default sub-type via a combo box from the GUI. This can be done by using `KDChartBarDiagram::setType()` as shown below and by updating the view.

```
                ....
                if ( text == "Normal" )
                   m_bars->setType( BarDiagram::Normal );
                   else if ( text == "Stacked" )
                   m_bars->setType( BarDiagram::Stacked );
                ....
```

```
m_chart->update();
```

We set the DataValueAttributes on a per-column basis here, because we want the texts to be printed in different colours - according to their respective dataset's colour. The user will be able to display or hide the values.

```
...
const QFont font(QFont( "Comic", 10 ));
const int colCount = m_bars->model()->columnCount();
for ( int iColumn = 0; iColumn<colCount; ++iColumn ) {
    QBrush brush( m_bars->brush( iColumn ) );
    DataValueAttributes a( m_bars->dataValueAttributes( iColumn ) );
    TextAttributes ta( a.textAttributes() );
    ta.setRotation( 0 );
    ta.setFont( font );
    ta .setPen( QPen( brush.color() ) );
    if ( checked )
    ta.setVisible( true );
    else
    ta.setVisible( false );

    a.setTextAttributes( ta );
    a.setVisible( true );
    m_bars->setDataValueAttributes( iColumn, a);
}

m_chart->update();
....
```

As you can see in the above code we are changing the default values for `DataValuesAttributes` `TextAttributes`. Also we allow the usert to display or not the texts dynamically. see `KDChartTextAttributes::setVisible()`.

In order to be able to display our diagram in threeD mode we need to bring `KDChartThreeDBarAttributes`, and configure it. Here we are enabling or disabling and change its Depth parameter according to the user interaction.

```
...
ThreeDBarAttributes td( m_bars->threeDBarAttributes() );
double defaultDepth = td.depth();
if ( checked ) {
td.setEnabled( true );
  if ( threeDDepthCB->isChecked() )
    td.setDepth( depthSB->value() );
  else
    td.setDepth( defaultDepth );
} else {
  td.setEnabled( false );
}
m_bars->setThreeDBarAttributes( td );
m_chart->update();
...
```

ThreeDBarAttributes are as simple to use as all other Attributes types. Our next lines of code will make use of the generic `KDChartAbstractDiagram::setPen()` available

to all diagram types, to allow the user to mark a column or reset it to the original Pen interactively.

```
...
const int column = markColumnSB->value();
QPen pen( m_bars->pen( column ) );
if ( checked ) {
pen.setColor( Qt::yellow );
pen.setStyle( Qt::DashLine );
pen.setWidth( 3 );
m_bars->setPen( column, pen );
}  else {
pen.setColor( Qt::darkGray );
pen.setStyle( Qt::SolidLine );
pen.setWidth( 1 );
m_bars->setPen( column, pen );
}
m_chart->update();
...
```

Note

It is important to know that have three levels of precedence when setting the attributes: Which means that once you have set the attributes for a

- Global: Weak
- Per column: Medium
- Per cell: Strong

the attributes: Which means that once you have set the attributes for a column or a cell, you will not be able to change those settings by calling the "global" method to reset it to another value, but instead call the per column or per index setter. As demonstrated in the above code.

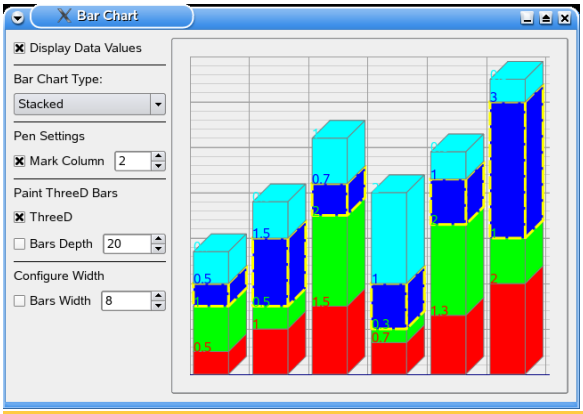
Finally we configure a typical `KDChartBarAttributes`, the Bar Width, for the user to be able to change the width of the bars dynamically increasing or decreasing its value via the Gui.

```
if (  widthCB->isChecked() ) {
  BarAttributes ba( m_bars->barAttributes() );
  ba.setFixedBarWidth( value );
  ba.setUseFixedBarWidth( true );
  m_bars->setBarAttributes( ba  );
}
m_chart->update();
```

Here we are making use of the `KDChartBarAttributes::setUseFixedBarWidth()` method to enable or disable the effect. The Bar Width value being passed by the value of a Spin Box.

See how this widget having some attributes enabled is displayed in the following screen-shot.

Figure 4.6. A Full featured Bar Chart



This example is available to compile and run from the `examples/Bars/Advanced` directory into your KD Chart installation. We recommend you to run it.

Line Charts

Tip

Line charts usually show numerical values and their development in time. Like the Bar Charts they can be used to compare multiple series of data.

An example might be the development of stock values over a longer period of time or the water level rise on several gauges.

As for Bar types, KD Chart can generate line charts of different kind of line charts. `KDChartLineDiagram` supports the following subtypes explained below:

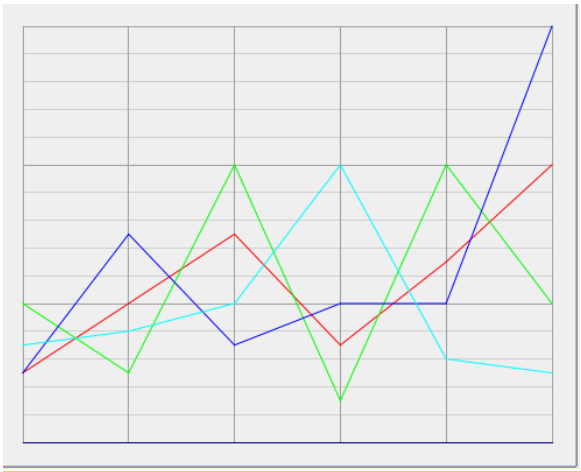
- Normal Line Chart
- Stacked Line Chart
- Percent Line chart

Normal Line Charts

Tip

Normal line charts are the most common type of line charts and are used when the datasets are compared to each other individually. For example, if you want to visualize the development of sales figures over time for each department separately, you might have one line per department.

Figure 4.7. A Normal Line Chart



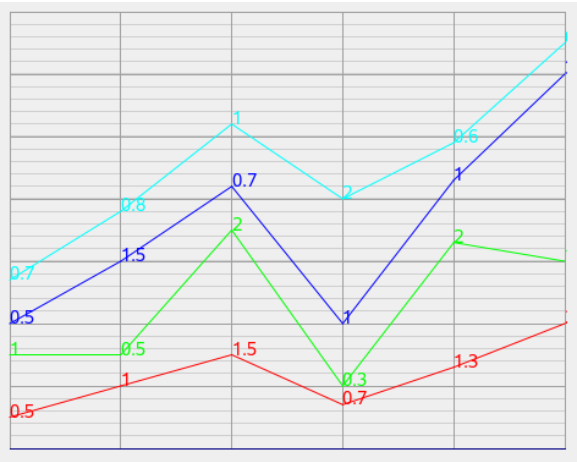
KD Chart draws normal line charts by default when in line chart mode so no method needs to be called to get one, however after having used your `KDChartLineDiagram` to display another line chart subtype you can reset it by calling `setType( Normal )`.

Stacked Line Charts

Tip

Stacked line charts allow you to compare the development of a series of values summarized over all datasets. You could use this if you are only interested in the development of total sales figures in your company, but have the data split up by department.

Figure 4.8. A Stacked Line Chart



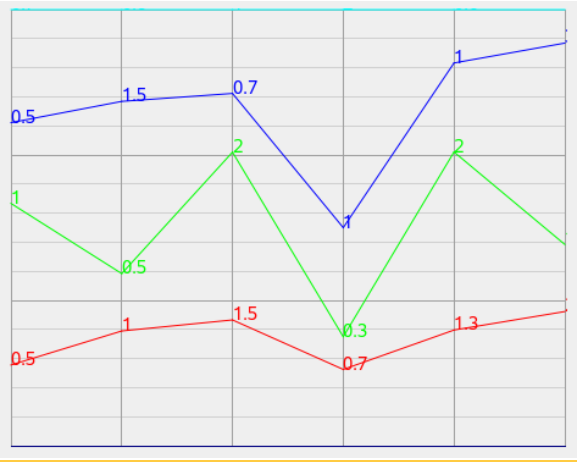
Stacked mode for line charts is activated by calling the `KDChartLineDiagram` method `setType( Stacked )`.

Percent Line Charts

Tip

Percent line charts show how much each value contributes to the total sum, similar to percent bar charts.

Figure 4.9. A Percent Line Chart



Percent: Percentage mode for line charts is activated by calling the KDChartLineDiagram function setType(Percent).

Note

Three-dimensional look of the lines is no special feature you can enable it for all types (Normal, Stacked or Percent) by setting its ThreeD attributes class (see KDChartThreeDLineAttributes.h to consult its interface). We will describe it more in details in the "Line Attributes" section further on.

Code Sample

For now let us make the above description more concrete by looking at the following code sample based on the Simple Widget example we have been demonstrating above (Chapter 3 - Two Ways - Widget Example). In this example we demonstrate how to configure your line diagram and change its attributes when working with a KDChartWidget.

First include the appropriate headers and bring in the "KDChart namespace":

```
#include <QApplication>
#include <KDChartWidget>
#include <KDChartLineDiagram>
#include <QPen>

using namespace KDChart;
```

We need to include KDChartLineDiagram in order to be able to configure some of its attributes as we will see further on.

```
int main( int argc, char** argv ) {
QApplication app( argc, argv );
Widget widget;
// our Widget can be configured
// as any Qt Widget
widget.resize( 600, 600 );
// store the data and assign it
QVector< double > vec0,  vec1;
vec0 << 5 << 1 << 3 << 4 << 1;
vec1 << 3 << 6 << 2 << 4 << 8;
vec2 << 0 << 7 << 1 << 2 << 1;
widget.setDataset( 0, vec0, "vec0" );
widget.setDataset( 1, vec1, "vec1" );
widget.setDataset( 2, vec2, "vec2" );
widget.setSubType(  Widget::Percent );
```

We dont need to change the default chart type as Line Charts is the default . In this case we also want to display it in percent mode. KDChartWidget with its setSubType

method allow us to achieve that the easy way.

```
widget.setSubType( Widget::Percent );
```

The default sub-type being Normal for all types of charts we need to call implicitely `KDChartWidget::setSubType()` in this case. We can also change the sub-type of our line chart further on by calling for example `setSubType( Widget::Stacked )` or reset its default value by calling `setSubType( Widget::Normal )`.

```
//Configure a pen and draw
 //a dashed line for column 1
 QPen pen;
 pen.setWidth( 3 );
 pen.setStyle( Qt::DashDotLine );
 pen.setColor(  Qt::green );
 // call your diagram and set the new pen
 widget.lineDiagram()->setPen(  1 , pen );
```

In the above code our intention is to draw a new style of line for this specific dataset in order to keep the attention of the public on it. That is what we call configuring an attribute. In this case the pen attribute. To do so we configure a QPen and then assign it to our diagram. `KDChartWidget::lineDiagram()` allow us to get a pointer to our widget diagram. As you can see it is very simple to assign a new pen to our diagram by calling the diagram `KDChartAbstractDiagram::setPen()` method.

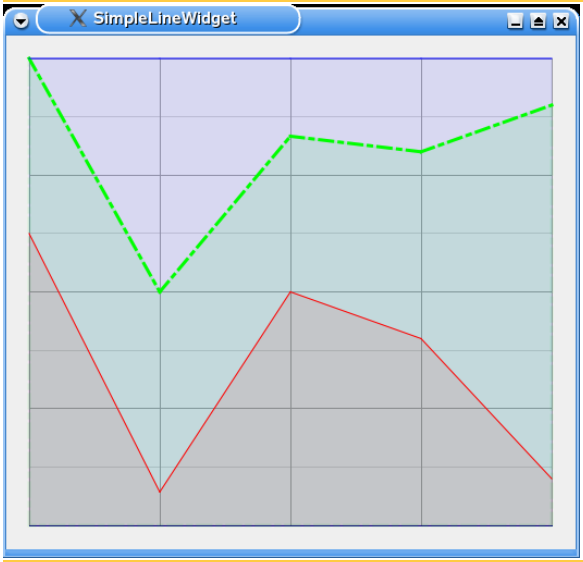
```
//Display in Area mode
LineAttributes ld;
ld.setDisplayArea( true );
//configure transparency
//it is nicer and let us
//all the area
ld.setTransparency( 25 );
widget.lineDiagram()->setLineAttributes( ld );
```

The code above makes use of typical `KDChartLineAttributes` and let us diplay the areas as well as set up the color transparency which is very helpfull when displaying a normal chart type where the areas can hide each other. Finally we conclude our small example:

```
widget.show();

return app.exec();
}
```

See the screen-shot below to view The resulting chart displayed by the above code.

Figure 4.10. A Simple Line ChartWidget



This example can be compiled and run from the following location of your KD Chart installation `examples/Lines/SimpleLineWidget`

Note

Configuring the attributes for a `KDChartLineDiagram` making use of a `KDChartChart` is done the same way as for a `KDChartWidget`. You just need to assign the configured attributes to your line diagram and assign the diagram to the chart by calling `KDChartChart::replaceDiagram()`.

Lines Attributes

There are only a few attributes specific to a line chart as it is using a Pen to draw the lines. Pen and Brush are generic attributes common to all types of diagrams and are handled by `KDChartAbstractDiagram` from which `KDChartLineDiagram` is derived indirectly.

However to make it simple for the user we have added some convenient functions to the `KDChartLineAttributes` in order to be able to display Areas and set transparency for all subtypes of a line chart. We will go through those methods further on in our Area charts section in this Chapter.

`KDChartLineDiagram` combined with its attributes or combined together with `KDChartMarkerAttributes` let us display the line chart subtypes as described above as well as Area Charts and Point charts the easy way. We will of course present all those alternatives with some sample code and ready to use examples in the next sections.

The use of LineAttributes is as simple as for the other chart types:

- Create a `KDChart::LineAttributes` object by calling `KDChartLineDiagram::lineAttributes`.
- Configure this object using the setters available.
- Assign it to your Diagram with the help of one of the setters available in `KDChart::LineDiagram`. All the attributes can be configured to be applied for the whole diagram, for a column, or at a specified index (`QModelIndex`).

KD Chart 2.0 supports the following attributes for the Line chart type. Each of those attributes can be set and retrieved the way we describe it in our example below:

- Display area: paint the area for a dataset.
- Area transparency: set the transparency for the displayed area color.

Line Example

Let us make this more concrete by looking at the following lines of code.

```
// 1 - Create a Line Attributes object
LineAttributes la = m_lines->lineAttributes( index );
// 2 - Display area at this index
la.setDisplayArea( true );
// 3 - Configure the transparency e.g. using the value
// returned by a Spin box
la.setTransparency( transparencySB->value() );
// 3 - Assign to your diagram
m_lines->setLineAttributes( index, la );
```

Of course the attributes can be set or retrieved for a column, at a given index, or for the whole diagram as for all the other chart types.

Tips and Tricks

In this section we want to give you some example about how to use some interesting features offered by the KD Chart 2.0 API. We will study the code and display a screenshot showing the resulting widget.

A complete Line Example

In the following implementation we want to be able to:

- Display the data values.
- Change the bar chart subtype (Normal, percent, Stacked).
- Change the line color and style for a dataset or a section of a dataset
- Display Areas and set their tranparency for one or several dataset(s).

```
 1
                        /****************************************************************
    ** Copyright (C) 2006 Klarälvdalens Datakonsult AB.  All rights reserved.
    **
 5  ** This file is part of the KD Chart library.
    **
    ** This file may be distributed and/or modified under the terms of the
    ** GNU General Public License version 2 as published by the Free Software
    ** Foundation and appearing in the file LICENSE.GPL included in the
10  ** packaging of this file.
    **
    ** Licensees holding valid commercial KD Chart licenses may use this file in
    ** accordance with the KD Chart Commercial License Agreement provided with
    ** the Software.
15  **
    ** This file is provided AS IS with NO WARRANTY OF ANY KIND, INCLUDING THE
    ** WARRANTY OF DESIGN, MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
    **
    ** See http://www.kdab.net/kdchart for
20  **   information about KDChart Commercial License Agreements.
    **
    ** Contact info@kdab.net if any conditions of this
    ** licensing are not clear to you.
    **
25  **********************************************************************/

   #ifndef MAINWINDOW_H
   #define MAINWINDOW_H

30 #include "ui_mainwindow.h"
   #include <TableModel.h>

   namespace KDChart {
       class Chart;
35     class DatasetProxyModel;
       class LineDiagram;
       class LineAttributes;
       class CartesianAxis;
       class Legend;
40 }

   class MainWindow : public QWidget, private Ui::MainWindow
   {
       Q_OBJECT
45
   public:
       MainWindow( QWidget* parent = 0 );


50
   private slots:

       void on_lineTypeCB_currentIndexChanged( const QString & text );
       void on_paintLegendCB_toggled( bool checked );
```

```
55      void on_paintValuesCB_toggled( bool checked );
        void on_paintMarkersCB_toggled( bool checked );
        void on_markersStyleCB_currentIndexChanged( const QString & text );
        void on_markersWidthSB_valueChanged( int i );
        void on_markersHeightSB_valueChanged( int i);
60      void on_displayAreasCB_toggled( bool checked );
        void on_transparencySB_valueChanged( int i );
        void on_zoomFactorSB_valueChanged( double factor );
        void on_hSBar_valueChanged( int value );
        void on_vSBar_valueChanged( int value );
65
    private:
        KDChart::Chart* m_chart;
        TableModel m_model;
70      KDChart::DatasetProxyModel* m_datasetProxy;
        KDChart::LineDiagram* m_lines;
        KDChart::Legend* m_legend;
        // mutable KDChart::CartesianAxis xAxis;
        //mutable KDChart::CartesianAxis yAxis;
75
    };


    #endif /* MAINWINDOW_H */
80
```

More explanation h file?

```
 1
                        /****************************************************************************
     ** Copyright (C) 2006 Klarälvdalens Datakonsult AB.  All rights reserved.
     **
 5   ** This file is part of the KD Chart library.
     **
     ** This file may be distributed and/or modified under the terms of the
     ** GNU General Public License version 2 as published by the Free Software
     ** Foundation and appearing in the file LICENSE.GPL included in the
10   ** packaging of this file.
     **
     ** Licensees holding valid commercial KD Chart licenses may use this file in
     ** accordance with the KD Chart Commercial License Agreement provided with
     ** the Software.
15   **
     ** This file is provided AS IS with NO WARRANTY OF ANY KIND, INCLUDING THE
     ** WARRANTY OF DESIGN, MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
     **
     ** See http://www.kdab.net/kdchart for
20   **   information about KDChart Commercial License Agreements.
     **
     ** Contact info@kdab.net if any conditions of this
     ** licensing are not clear to you.
     **
25   **********************************************************************/

    #include "mainwindow.h"

    #include <KDChartChart>
30  #include <KDChartAbstractCoordinatePlane>
    #include <KDChartLineDiagram>
    #include <KDChartLineAttributes>
    #include <KDChartTextAttributes>
    #include <KDChartDataValueAttributes>
35  #include <KDChartThreeDLineAttributes>
    #include <KDChartMarkerAttributes>
    #include <KDChartLegend>
```

```
   #include <QDebug>
40 #include <QPainter>

   using namespace KDChart;

   MainWindow::MainWindow( QWidget* parent ) :
45     QWidget( parent )
   {
       setupUi( this );

       QHBoxLayout* chartLayout = new QHBoxLayout( chartFrame );
50     m_chart = new Chart();
       chartLayout->addWidget( m_chart );
       hSBar->setVisible( false );
       vSBar->setVisible( false );

55     m_model.loadFromCSV( ":/data" );

       // Set up the diagram
       m_lines = new LineDiagram();
       m_lines->setModel( &m_model );
60     //CartesianAxisList List = m_lines->axesList();
       CartesianAxis *xAxis = new CartesianAxis( m_lines );
       CartesianAxis *yAxis = new CartesianAxis ( m_lines );
       CartesianAxis *axisTop = new CartesianAxis ( m_lines );
       CartesianAxis *axisRight = new CartesianAxis ( m_lines );
65     xAxis->setPosition ( KDChart::CartesianAxis::Bottom );
       yAxis->setPosition ( KDChart::CartesianAxis::Left );
       axisTop->setPosition( KDChart::CartesianAxis::Top );
       axisRight->setPosition( KDChart::CartesianAxis::Left);//Right );

70     xAxis->setTitleText ( "Abscissa axis at the bottom" );
       yAxis->setTitleText ( "Ordinate axis at the left side" );
       axisTop->setTitleText ( "Abscissa axis at the top" );
       axisRight->setTitleText ( "Ordinate axis at the right side" );
       TextAttributes taTop ( xAxis->titleTextAttributes () );
75     taTop.setPen( QPen( Qt::red ) );
       axisTop->setTitleTextAttributes ( taTop );
       TextAttributes taRight ( xAxis->titleTextAttributes () );
       Measure me( taRight.fontSize() );
       me.setValue( me.value() * 1.5 );
80     taRight.setFontSize( me );
       axisRight->setTitleTextAttributes ( taRight );

       m_lines->addAxis( xAxis );
       m_lines->addAxis( yAxis );
85     m_lines->addAxis( axisTop );
       m_lines->addAxis( axisRight );
       m_chart->coordinatePlane()->replaceDiagram( m_lines );

       // Set up the legend
90     m_legend = new Legend( m_lines, m_chart );
       m_chart->addLegend( m_legend );
       m_legend->hide();
   }

95 void MainWindow::on_lineTypeCB_currentIndexChanged( const QString & text )
   {
       if ( text == "Normal" )
           m_lines->setType( LineDiagram::Normal );
       else if ( text == "Stacked" )
100        m_lines->setType( LineDiagram::Stacked );
       else if ( text == "Percent" )
           m_lines->setType( LineDiagram::Percent );
       else
           qWarning (" Does not match any type");
105
       m_chart->update();
   }
```

```
110 void MainWindow::on_paintLegendCB_toggled( bool checked )
    {
        m_legend->setVisible( checked );
        m_chart->update();
    }
115
    void MainWindow::on_paintValuesCB_toggled( bool checked )
    {
        // We set the DataValueAttributes on a per-column basis here,
        // because we want the texts to be printed in different
120     // colours - according to their respective dataset's colour.
        const QFont font(QFont( "Comic", 10 ));
        const int colCount = m_lines->model()->columnCount();
        for ( int iColumn = 0; iColumn<colCount; ++iColumn ) {
            QBrush brush( m_lines->brush( iColumn ) );
125         DataValueAttributes a( m_lines->dataValueAttributes( iColumn ) );
            if ( ! paintMarkersCB->isChecked() ) {
                MarkerAttributes ma( a.markerAttributes() );
                ma.setVisible( false );
                a.setMarkerAttributes( ma );
130         }
            TextAttributes ta( a.textAttributes() );
            ta.setRotation( 0 );
            ta.setFont( font );
            ta .setPen( QPen( brush.color() ) );
135         if ( checked )
                ta.setVisible( true );
            else
                ta.setVisible( false );

140         a.setTextAttributes( ta );
            a.setVisible( true );
            m_lines->setDataValueAttributes( iColumn, a);
        }
        m_chart->update();
145 }


    void MainWindow::on_paintMarkersCB_toggled( bool checked )
    {
150     // first: Specify global settings!
        DataValueAttributes attrs( m_lines->dataValueAttributes() );

        MarkerAttributes ma( attrs.markerAttributes() );
        MarkerAttributes::MarkerStylesMap map;
155     map.insert( 0, MarkerAttributes::MarkerSquare );
        map.insert( 1, MarkerAttributes::MarkerCircle );
        map.insert( 2, MarkerAttributes::MarkerRing );
        map.insert( 3, MarkerAttributes::MarkerCross );
        map.insert( 4, MarkerAttributes::MarkerDiamond );
160     ma.setMarkerStylesMap( map );

        switch ( markersStyleCB->currentIndex() ) {
            case 0:
                break;
165         case 1:
                ma.setMarkerStyle( MarkerAttributes::MarkerCircle );
                break;
            case 2:
                ma.setMarkerStyle( MarkerAttributes::MarkerSquare );
170             break;
            case 3:
                ma.setMarkerStyle( MarkerAttributes::MarkerDiamond );
                break;
            case 4:
175             ma.setMarkerStyle( MarkerAttributes::Marker1Pixel );
                break;
            case 5:
```

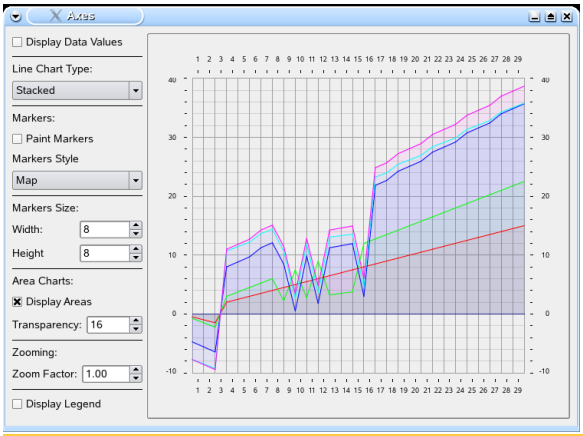
```
                ma.setMarkerStyle( MarkerAttributes::Marker4Pixels );
                break;
            case 6:
                ma.setMarkerStyle( MarkerAttributes::MarkerRing );
                break;
            case 7:
                ma.setMarkerStyle( MarkerAttributes::MarkerCross );
                break;
            case 8:
                ma.setMarkerStyle( MarkerAttributes::MarkerFastCross );
                break;
        }
        ma.setMarkerSize( QSize( markersWidthSB->value(), markersHeightSB->value() ) );

        if ( checked )
            ma.setVisible( true );
        else
            ma.setVisible( false );

        attrs.setMarkerAttributes( ma );


        // Note: We set the global attributes now ( == before the following loop),
        //       so that inside of that loop the getter functions
        //       can fall-back to these global settings,
        //       if no cell-specific / column-specific attributes were set
        //       for a cell (or for a column, resp.).
        m_lines->setDataValueAttributes( attrs );


        // next: Specify column- / cell-specific attributes!
        const int rowCount = m_lines->model()->rowCount();
        const int colCount = m_lines->model()->columnCount();
        for ( int iColumn = 0; iColumn<colCount; ++iColumn ) {
            // Specify column-specific attributes!
            if ( markersStyleCB->currentIndex() == 0 ) {
                // retrieve column specific attributes
                // or fall back to global settings:
                DataValueAttributes colAttributes( m_lines->dataValueAttributes( iColumn
                MarkerAttributes ma( colAttributes.markerAttributes() );
                ma.setMarkerStyle( ma.markerStylesMap().value( iColumn ) );
                colAttributes.setMarkerAttributes( ma );
                // set column specific attributes:
                m_lines->setDataValueAttributes( iColumn, colAttributes );
            }

            // Specify cell-specific attributes for some values!
            for ( int j=0; j< rowCount; ++j ) {
                const QModelIndex index( m_lines->model()->index( j, iColumn, QModelIndex
                const QBrush brush( m_lines->brush( index ) );
                const double value = m_lines->model()->data( index ).toDouble();
                /* Set a specific color - marker for a specific value */
                if ( value == 8 ) {
                    // retrieve cell specific attributes
                    // or fall back to column settings
                    // or fall back to global settings:
                    DataValueAttributes yellowAttributes( m_lines->dataValueAttributes( i
                    MarkerAttributes yellowMarker( yellowAttributes.markerAttributes() );
                    yellowMarker.setMarkerColor( Qt::yellow );
                    yellowAttributes.setMarkerAttributes( yellowMarker );
                    // set cell specific attributes:
                    m_lines->setDataValueAttributes( index, yellowAttributes );
                }
            }
        }

        m_chart->update();
    }
```

```
    void MainWindow::on_markersStyleCB_currentIndexChanged( const QString & text )
    {
250     Q_UNUSED( text );
        if ( paintMarkersCB->isChecked() )
            on_paintMarkersCB_toggled( true );
    }

255
    void MainWindow::on_markersWidthSB_valueChanged( int i )
    {
        Q_UNUSED( i );
        markersHeightSB->setValue( markersWidthSB->value() );
260     if ( paintMarkersCB->isChecked() )
            on_paintMarkersCB_toggled( true );
    }

    void MainWindow::on_markersHeightSB_valueChanged( int /*i*/ )
265 {
        markersWidthSB->setValue( markersHeightSB->value() );
        if ( paintMarkersCB->isChecked() )
            on_paintMarkersCB_toggled( true );
    }
270

    void MainWindow::on_displayAreasCB_toggled( bool checked )
    {
        LineAttributes la( m_lines->lineAttributes() );
275     if ( checked  ) {
            la.setDisplayArea( true );
            la.setTransparency( transparencySB->value() );
        } else
            la.setDisplayArea( false );
280     m_lines->setLineAttributes( la );
        m_chart->update();
    }

    void MainWindow::on_transparencySB_valueChanged( int alpha )
285 {
        LineAttributes la( m_lines->lineAttributes() );
        la.setTransparency( alpha );
        m_lines->setLineAttributes( la );
        on_displayAreasCB_toggled( true );
290 }

    void MainWindow::on_zoomFactorSB_valueChanged( double factor )
    {
        if ( factor > 1 ) {
295         hSBar->setVisible( true );
            vSBar->setVisible( true );
        } else {
            hSBar->setValue( 500 );
            vSBar->setValue( 500 );
300         hSBar->setVisible( false );
            vSBar->setVisible( false );
        }
        m_chart->coordinatePlane()->setZoomFactorX( factor );
        m_chart->coordinatePlane()->setZoomFactorY( factor );
305     m_chart->update();
    }

    void MainWindow::on_hSBar_valueChanged( int hPos )
    {
310     m_chart->coordinatePlane()->setZoomCenter( QPointF(hPos/1000.0, vSBar->value()/10
        m_chart->update();
    }

    void MainWindow::on_vSBar_valueChanged( int vPos )
315 {
        m_chart->coordinatePlane()->setZoomCenter( QPointF( hSBar->value()/1000.0, vPos/1
        m_chart->update();
```

```
    }
320
```

More explanation cpp file?

Figure 4.11. A Full featured Line Chart

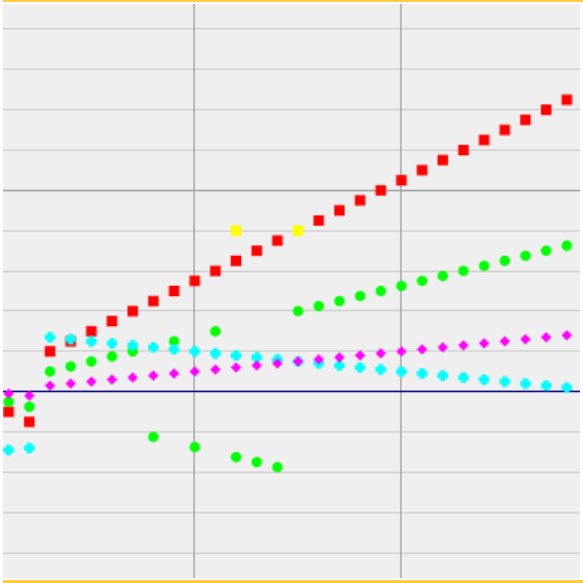


Point Charts

Tip

Point charts often are used to visualize a big number of data in one or several datasets. A well known point chart example is the historical first Herzsprung-Russel diagram from 1914 where circles represented stars with directly measured parallaxes and crosses were used for guessed values of stars from star clusters similar to the following simple chart.

Figure 4.12. A Point Chart



Unlike the other chart types in KD Chart the point chart is not a type of its own but actually a special kind of Line Chart. The resulting display is obtained by painting markers instead of lines as we will see in the following code sample.

The process for creating a point chart is very simple as described below:

- Set up a line chart and set its pen to Qt::NoPen.
- Configure and display its data values markers.

Point Example

The following code sample is going through the process above in order to create a point chart.

```
// 1 - Set up a line chart and set its pen to Qt::NoPen
m_lines = new LineDiagram();
m_lines->setPen( Qt::NoPen );
// 2 - Configure and display its data values markers
// we will see more in details how to handle DataValueAttribute
// and its MarkerAttributes
DataValueAttributes dva;
MarkerAttributes ma = dva.markerAttributes();
ma->setVisible( true );
// store several styles if needed
// we can use a style for each dataset for example
MarkerAttributes::MarkerStylesMap map;
map.insert( 0, MarkerAttributes::MarkerSquare );
map.insert( 1, MarkerAttributes::MarkerCircle );
....etc
```

```
ma.setMarkerStylesMap( map );
dva.setMarkerAttributes( ma );
dva.setVisible( true );
//3 - Paint different markers style for each dataset
const int colCount = m_lines->model()->columnCount();
for ( int i = 0; i < colCount; ++i ) {
    ma.setMarkerStyle( ma.markerStylesMap().value(i) );
    colAttributes.setMarkerAttributes( ma );
}
// Assign to the diagram
m_lines->setDataValueAttributes( i, dva );
```

After assigning your markersattributes setting to the diagram you may need to run `KDChartChart::update` as follow, in order to force a re-paint depending of your implementation.

```
m_chart->update();
```

We recommend you to run the complete example presented in the following Tips section.

Points Attributes

As you have probably deduced from the section above, point charts are line charts configured with no pen to avoid displaying the lines and using the generic classes `KDChartDataValueAttributes` and its `KDChartMarkerAttributes` available to all other diagram types supported by KD Chart 2.0.

For this reason we will for now point you to the sections related to those subject and in particular to Chapter 5 - Customizing your Chart - Section Markers or Chapter 9 - Advanced Charting - Section Data Value Manipulation and finalize this section by implementing a full featured point chart in the Tips section below.

Tips and Tricks

In this section we want to give you some example about how to use some interesting features offered by the KD Chart 2.0 API. We will study the code and display a screenshot showing the resulting widget.

A complete Point Example

In the following implementation we want to be able to:

- Be able to configure the points styles, color and size.

- Display data values or hide it.
- Display areas for each dataset on its own
- Shift between points and lines charts

```
 1                       /****************************************************************
    ** Copyright (C) 2006 Klarälvdalens Datakonsult AB.  All rights reserved.
    **
 5  ** This file is part of the KD Chart library.
    **
    ** This file may be distributed and/or modified under the terms of the
    ** GNU General Public License version 2 as published by the Free Software
    ** Foundation and appearing in the file LICENSE.GPL included in the
10  ** packaging of this file.
    **
    ** Licensees holding valid commercial KD Chart licenses may use this file in
    ** accordance with the KD Chart Commercial License Agreement provided with
    ** the Software.
15  **
    ** This file is provided AS IS with NO WARRANTY OF ANY KIND, INCLUDING THE
    ** WARRANTY OF DESIGN, MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
    **
    ** See http://www.kdab.net/kdchart for
20  **   information about KDChart Commercial License Agreements.
    **
    ** Contact info@kdab.net if any conditions of this
    ** licensing are not clear to you.
    **
25  **********************************************************************/

   #ifndef MAINWINDOW_H
   #define MAINWINDOW_H

30 #include "ui_mainwindow.h"
   #include <TableModel.h>

   namespace KDChart {
       class Chart;
35     class DatasetProxyModel;
       class LineDiagram;
       class LineAttributes;
       class CartesianAxis;
       class Legend;
40 }

   class MainWindow : public QWidget, private Ui::MainWindow
   {
       Q_OBJECT
45
   public:
       MainWindow( QWidget* parent = 0 );


50
   private slots:

       void on_lineTypeCB_currentIndexChanged( const QString & text );
       void on_paintLegendCB_toggled( bool checked );
55     void on_paintValuesCB_toggled( bool checked );
       void on_paintMarkersCB_toggled( bool checked );
       void on_markersStyleCB_currentIndexChanged( const QString & text );
       void on_markersWidthSB_valueChanged( int i );
       void on_markersHeightSB_valueChanged( int i);
60     void on_displayAreasCB_toggled( bool checked );
       void on_transparencySB_valueChanged( int i );
```

```
        void on_zoomFactorSB_valueChanged( double factor );
        void on_hSBar_valueChanged( int value );
        void on_vSBar_valueChanged( int value );

    private:
        KDChart::Chart* m_chart;
        TableModel m_model;
        KDChart::DatasetProxyModel* m_datasetProxy;
        KDChart::LineDiagram* m_lines;
        KDChart::Legend* m_legend;
        // mutable KDChart::CartesianAxis xAxis;
        //mutable KDChart::CartesianAxis yAxis;
    };


    #endif /* MAINWINDOW_H */
```

More explanation h file?

```
                        /****************************************************************
     ** Copyright (C) 2006 Klarälvdalens Datakonsult AB.  All rights reserved.
     **
     ** This file is part of the KD Chart library.
     **
     ** This file may be distributed and/or modified under the terms of the
     ** GNU General Public License version 2 as published by the Free Software
     ** Foundation and appearing in the file LICENSE.GPL included in the
     ** packaging of this file.
     **
     ** Licensees holding valid commercial KD Chart licenses may use this file in
     ** accordance with the KD Chart Commercial License Agreement provided with
     ** the Software.
     **
     ** This file is provided AS IS with NO WARRANTY OF ANY KIND, INCLUDING THE
     ** WARRANTY OF DESIGN, MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
     **
     ** See http://www.kdab.net/kdchart for
     **   information about KDChart Commercial License Agreements.
     **
     ** Contact info@kdab.net if any conditions of this
     ** licensing are not clear to you.
     **
     **********************************************************************/

    #include "mainwindow.h"

    #include <KDChartChart>
    #include <KDChartAbstractCoordinatePlane>
    #include <KDChartLineDiagram>
    #include <KDChartLineAttributes>
    #include <KDChartTextAttributes>
    #include <KDChartDataValueAttributes>
    #include <KDChartThreeDLineAttributes>
    #include <KDChartMarkerAttributes>
    #include <KDChartLegend>

    #include <QDebug>
    #include <QPainter>

    using namespace KDChart;

    MainWindow::MainWindow( QWidget* parent ) :
```

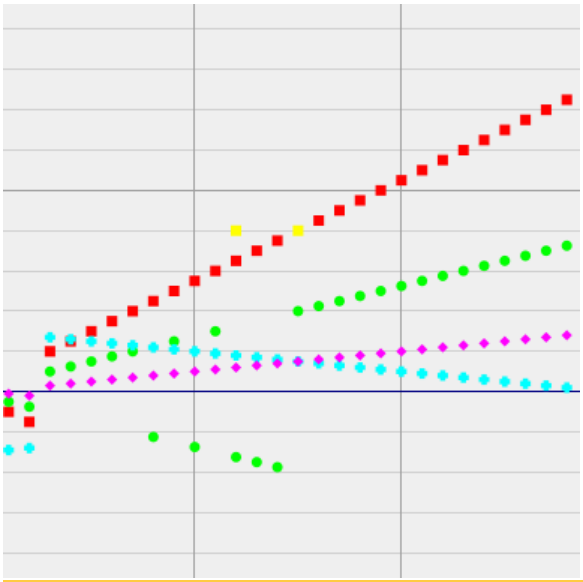
```
    QWidget( parent )
{
    setupUi( this );

    QHBoxLayout* chartLayout = new QHBoxLayout( chartFrame );
    m_chart = new Chart();
    chartLayout->addWidget( m_chart );
    hSBar->setVisible( false );
    vSBar->setVisible( false );

    m_model.loadFromCSV( ":/data" );

    // Set up the diagram
    m_lines = new LineDiagram();
    m_lines->setModel( &m_model );
    //CartesianAxisList List = m_lines->axesList();
    CartesianAxis *xAxis = new CartesianAxis( m_lines );
    CartesianAxis *yAxis = new CartesianAxis ( m_lines );
    CartesianAxis *axisTop = new CartesianAxis ( m_lines );
    CartesianAxis *axisRight = new CartesianAxis ( m_lines );
    xAxis->setPosition ( KDChart::CartesianAxis::Bottom );
    yAxis->setPosition ( KDChart::CartesianAxis::Left );
    axisTop->setPosition( KDChart::CartesianAxis::Top );
    axisRight->setPosition( KDChart::CartesianAxis::Left);//Right );

    xAxis->setTitleText ( "Abscissa axis at the bottom" );
    yAxis->setTitleText ( "Ordinate axis at the left side" );
    axisTop->setTitleText ( "Abscissa axis at the top" );
    axisRight->setTitleText ( "Ordinate axis at the right side" );
    TextAttributes taTop ( xAxis->titleTextAttributes () );
    taTop.setPen( QPen( Qt::red ) );
    axisTop->setTitleTextAttributes ( taTop );
    TextAttributes taRight ( xAxis->titleTextAttributes () );
    Measure me( taRight.fontSize() );
    me.setValue( me.value() * 1.5 );
    taRight.setFontSize( me );
    axisRight->setTitleTextAttributes ( taRight );

    m_lines->addAxis( xAxis );
    m_lines->addAxis( yAxis );
    m_lines->addAxis( axisTop );
    m_lines->addAxis( axisRight );
    m_chart->coordinatePlane()->replaceDiagram( m_lines );

    // Set up the legend
    m_legend = new Legend( m_lines, m_chart );
    m_chart->addLegend( m_legend );
    m_legend->hide();
}

void MainWindow::on_lineTypeCB_currentIndexChanged( const QString & text )
{
    if ( text == "Normal" )
        m_lines->setType( LineDiagram::Normal );
    else if ( text == "Stacked" )
        m_lines->setType( LineDiagram::Stacked );
    else if ( text == "Percent" )
        m_lines->setType( LineDiagram::Percent );
    else
        qWarning (" Does not match any type");

    m_chart->update();
}


void MainWindow::on_paintLegendCB_toggled( bool checked )
{
    m_legend->setVisible( checked );
    m_chart->update();
}
```

```
115
    void MainWindow::on_paintValuesCB_toggled( bool checked )
    {
        // We set the DataValueAttributes on a per-column basis here,
        // because we want the texts to be printed in different
120     // colours - according to their respective dataset's colour.
        const QFont font(QFont( "Comic", 10 ));
        const int colCount = m_lines->model()->columnCount();
        for ( int iColumn = 0; iColumn<colCount; ++iColumn ) {
            QBrush brush( m_lines->brush( iColumn ) );
125         DataValueAttributes a( m_lines->dataValueAttributes( iColumn ) );
            if ( ! paintMarkersCB->isChecked() ) {
                MarkerAttributes ma( a.markerAttributes() );
                ma.setVisible( false );
                a.setMarkerAttributes( ma );
130         }
            TextAttributes ta( a.textAttributes() );
            ta.setRotation( 0 );
            ta.setFont( font );
            ta .setPen( QPen( brush.color() ) );
135         if ( checked )
                ta.setVisible( true );
            else
                ta.setVisible( false );

140         a.setTextAttributes( ta );
            a.setVisible( true );
            m_lines->setDataValueAttributes( iColumn, a);
        }
        m_chart->update();
145 }


    void MainWindow::on_paintMarkersCB_toggled( bool checked )
    {
150     // first: Specify global settings!
        DataValueAttributes attrs( m_lines->dataValueAttributes() );

        MarkerAttributes ma( attrs.markerAttributes() );
        MarkerAttributes::MarkerStylesMap map;
155     map.insert( 0, MarkerAttributes::MarkerSquare );
        map.insert( 1, MarkerAttributes::MarkerCircle );
        map.insert( 2, MarkerAttributes::MarkerRing );
        map.insert( 3, MarkerAttributes::MarkerCross );
        map.insert( 4, MarkerAttributes::MarkerDiamond );
160     ma.setMarkerStylesMap( map );

        switch ( markersStyleCB->currentIndex() ) {
            case 0:
                break;
165         case 1:
                ma.setMarkerStyle( MarkerAttributes::MarkerCircle );
                break;
            case 2:
                ma.setMarkerStyle( MarkerAttributes::MarkerSquare );
170             break;
            case 3:
                ma.setMarkerStyle( MarkerAttributes::MarkerDiamond );
                break;
            case 4:
175             ma.setMarkerStyle( MarkerAttributes::Marker1Pixel );
                break;
            case 5:
                ma.setMarkerStyle( MarkerAttributes::Marker4Pixels );
                break;
180         case 6:
                ma.setMarkerStyle( MarkerAttributes::MarkerRing );
                break;
            case 7:
                ma.setMarkerStyle( MarkerAttributes::MarkerCross );
```

```
185             break;
            case 8:
                ma.setMarkerStyle( MarkerAttributes::MarkerFastCross );
                break;
        }
190     ma.setMarkerSize( QSize( markersWidthSB->value(), markersHeightSB->value() ) );

        if ( checked )
            ma.setVisible( true );
        else
195         ma.setVisible( false );

        attrs.setMarkerAttributes( ma );


200     // Note: We set the global attributes now ( == before the following loop),
        //       so that inside of that loop the getter functions
        //       can fall-back to these global settings,
        //       if no cell-specific / column-specific attributes were set
        //       for a cell (or for a column, resp.).
205     m_lines->setDataValueAttributes( attrs );


        // next: Specify column- / cell-specific attributes!
        const int rowCount = m_lines->model()->rowCount();
210     const int colCount = m_lines->model()->columnCount();
        for ( int iColumn = 0; iColumn<colCount; ++iColumn ) {
            // Specify column-specific attributes!
            if ( markersStyleCB->currentIndex() == 0 ) {
                // retrieve column specific attributes
215             // or fall back to global settings:
                DataValueAttributes colAttributes( m_lines->dataValueAttributes( iColumn
                MarkerAttributes ma( colAttributes.markerAttributes() );
                ma.setMarkerStyle( ma.markerStylesMap().value( iColumn ) );
                colAttributes.setMarkerAttributes( ma );
220             // set column specific attributes:
                m_lines->setDataValueAttributes( iColumn, colAttributes );
            }

            // Specify cell-specific attributes for some values!
225         for ( int j=0; j< rowCount; ++j ) {
                const QModelIndex index( m_lines->model()->index( j, iColumn, QModelIndex
                const QBrush brush( m_lines->brush( index ) );
                const double value = m_lines->model()->data( index ).toDouble();
                /* Set a specific color - marker for a specific value */
230             if ( value == 8 ) {
                    // retrieve cell specific attributes
                    // or fall back to column settings
                    // or fall back to global settings:
                    DataValueAttributes yellowAttributes( m_lines->dataValueAttributes( i
235                 MarkerAttributes yellowMarker( yellowAttributes.markerAttributes() );
                    yellowMarker.setMarkerColor( Qt::yellow );
                    yellowAttributes.setMarkerAttributes( yellowMarker );
                    // set cell specific attributes:
                    m_lines->setDataValueAttributes( index, yellowAttributes );
240             }
            }
        }

        m_chart->update();
245 }


    void MainWindow::on_markersStyleCB_currentIndexChanged( const QString & text )
    {
250     Q_UNUSED( text );
        if ( paintMarkersCB->isChecked() )
            on_paintMarkersCB_toggled( true );
    }
```

```
255
    void MainWindow::on_markersWidthSB_valueChanged( int i )
    {
        Q_UNUSED( i );
        markersHeightSB->setValue( markersWidthSB->value() );
260     if ( paintMarkersCB->isChecked() )
            on_paintMarkersCB_toggled( true );
    }

    void MainWindow::on_markersHeightSB_valueChanged( int /*i*/ )
265 {
        markersWidthSB->setValue( markersHeightSB->value() );
        if ( paintMarkersCB->isChecked() )
            on_paintMarkersCB_toggled( true );
    }
270

    void MainWindow::on_displayAreasCB_toggled( bool checked )
    {
        LineAttributes la( m_lines->lineAttributes() );
275     if ( checked  ) {
            la.setDisplayArea( true );
            la.setTransparency( transparencySB->value() );
        } else
            la.setDisplayArea( false );
280     m_lines->setLineAttributes( la );
        m_chart->update();
    }

    void MainWindow::on_transparencySB_valueChanged( int alpha )
285 {
        LineAttributes la( m_lines->lineAttributes() );
        la.setTransparency( alpha );
        m_lines->setLineAttributes( la );
        on_displayAreasCB_toggled( true );
290 }

    void MainWindow::on_zoomFactorSB_valueChanged( double factor )
    {
        if ( factor > 1 ) {
295         hSBar->setVisible( true );
            vSBar->setVisible( true );
        } else {
            hSBar->setValue( 500 );
            vSBar->setValue( 500 );
300         hSBar->setVisible( false );
            vSBar->setVisible( false );
        }
        m_chart->coordinatePlane()->setZoomFactorX( factor );
        m_chart->coordinatePlane()->setZoomFactorY( factor );
305     m_chart->update();
    }

    void MainWindow::on_hSBar_valueChanged( int hPos )
    {
310     m_chart->coordinatePlane()->setZoomCenter( QPointF(hPos/1000.0, vSBar->value()/10
        m_chart->update();
    }

    void MainWindow::on_vSBar_valueChanged( int vPos )
315 {
        m_chart->coordinatePlane()->setZoomCenter( QPointF( hSBar->value()/1000.0, vPos/1
        m_chart->update();
    }

320
```

Figure 4.13. A Full featured Point Chart



Area Charts

Tip

Even more than a Line Chart (of which they are attributes) an area chart can give a good visual impression of different datasets and their relation to each other.

For example the area chart type might be the best choice for showing how several sources contributed to increasing ozone values in a conurbation during a summer's months.

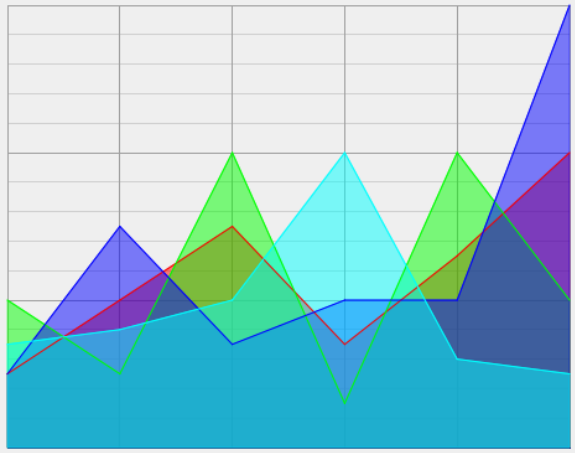
Area charts are Line Charts and thus based upon several points which are connected by lines—the difference to the line chart is that the area below a line is filled by the respective dataset's color. This gives a clear appreciation of each dataset's relative values.

In order to make it possible to see all points, since some are covered by another dataset's area, we have introduced an attribute which allow the user to configure the level of transparency (more about that into the Attributes paragraph of this section. KD Chart 2.0 supports of course Area display for all subtypes of line charts and thus allow also the user to display the non-overlapping line types. The following types can be displayed

very simply in Area mode:

- Normal Line Area
- Stacked Line Area.
- Percent Line Area.

Figure 4.14. An Area Chart



Note

KD Chart uses the term "area" in two different ways which can be distinguished easily:

- In this chapter it stands for a special chart type or even more accurately as a line diagram attribute.
- In other context it can also point to the different (normally rectangular) parts of a chart like for example the *legend area* or the *headers area*.

This varying usage of the word "area" should Not cause a lack of clarity: In the context of this special section on *area charts* the word is clear, in the rest of the manual it just means a part of a chart.

Displaying the area for a dataset or the whole diagram is straight forward:

- Create a LineAttribute object by calling `KDChartLineDiagram::lineAttributes`
- Display it. You can also configure the level of transparency.

Area Example

Let us make this more concrete by looking at the following lines of code:

```
// Create a LineAttribute object
LineAttributes la = m_lines->lineAttributes( index );
// set Display implicitely
la.setDisplayArea( true );
// Assign to the diagram
m_lines->setLineAttributes( index, la );
```

Of course Brush and Pen settings as well as all other configurable attributes accessible by the diagram itself can be set, which give the user a lot of flexibility (display or hide data values, markers, lines, configure colors etc ...).

area Attributes

There are no specifical attributes related to the Area chart. As explained above Area charts displaying is implemented as a Line Attribute. Of course the generic attributes common to all chart types are availables, which give us full flexibility to configure our area chart.

Tips and Tricks

In this section we want to give you some example about how to use some interesting features offered by the KD Chart 2.0 API. We will study the code and display a screenshot showing the resulting widget.

A complete Area Example

In the following implementation we want to be able to:

- Display data values or markers and hide
- Shift line types (Normal, Stacked, Percent)
- Display areas for each dataset on its own and for the whole diagram

- run the area animation

```
/****************************************************************
** Copyright (C) 2006 Klarälvdalens Datakonsult AB.  All rights reserved.
**
** This file is part of the KD Chart library.
**
** This file may be distributed and/or modified under the terms of the
** GNU General Public License version 2 as published by the Free Software
** Foundation and appearing in the file LICENSE.GPL included in the
** packaging of this file.
**
** Licensees holding valid commercial KD Chart licenses may use this file in
** accordance with the KD Chart Commercial License Agreement provided with
** the Software.
**
** This file is provided AS IS with NO WARRANTY OF ANY KIND, INCLUDING THE
** WARRANTY OF DESIGN, MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
**
** See http://www.kdab.net/kdchart for
**   information about KDChart Commercial License Agreements.
**
** Contact info@kdab.net if any conditions of this
** licensing are not clear to you.
**
**********************************************************************/

#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include "ui_mainwindow.h"
#include <TableModel.h>

namespace KDChart {
    class Chart;
    class LineDiagram;
}

class MainWindow : public QWidget, private Ui::MainWindow
{
    Q_OBJECT

public:
    MainWindow( QWidget* parent = 0 );

private slots:

    void on_lineTypeCB_currentIndexChanged( const QString & text );
    void on_paintValuesCB_toggled( bool checked );
    void on_animateAreasCB_toggled( bool checked );
    void on_highlightAreaCB_toggled( bool checked );
    void on_highlightAreaSB_valueChanged( int i );
    void setHighlightArea( int column, int opacity, bool checked, bool doUpdate );
    void slot_timerFired();

private:
    KDChart::Chart* m_chart;
    KDChart::LineDiagram* m_lines;
    TableModel m_model;
    int m_curColumn;
    int m_curOpacity;
};


#endif /* MAINWINDOW_H */

```

More explanation h file?

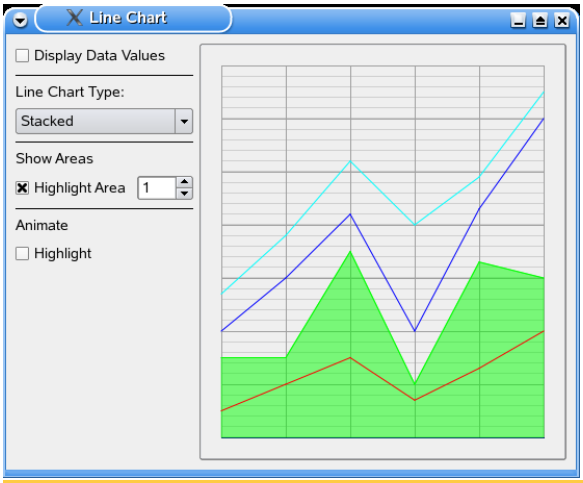
```
 1
                        /****************************************************************
    ** Copyright (C) 2006 Klarälvdalens Datakonsult AB.  All rights reserved.
    **
 5  ** This file is part of the KD Chart library.
    **
    ** This file may be distributed and/or modified under the terms of the
    ** GNU General Public License version 2 as published by the Free Software
    ** Foundation and appearing in the file LICENSE.GPL included in the
10  ** packaging of this file.
    **
    ** Licensees holding valid commercial KD Chart licenses may use this file in
    ** accordance with the KD Chart Commercial License Agreement provided with
    ** the Software.
15  **
    ** This file is provided AS IS with NO WARRANTY OF ANY KIND, INCLUDING THE
    ** WARRANTY OF DESIGN, MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
    **
    ** See http://www.kdab.net/kdchart for
20  **   information about KDChart Commercial License Agreements.
    **
    ** Contact info@kdab.net if any conditions of this
    ** licensing are not clear to you.
    **
25  **********************************************************************/

   #include "mainwindow.h"

   #include <KDChartChart>
30 #include <KDChartDatasetProxyModel>
   #include <KDChartAbstractCoordinatePlane>
   #include <KDChartLineDiagram>
   #include <KDChartTextAttributes>
   #include <KDChartDataValueAttributes>
35

   #include <QDebug>
   #include <QPainter>
   #include <QTimer>
40
   using namespace KDChart;

   MainWindow::MainWindow( QWidget* parent ) :
       QWidget( parent )
45 {
       setupUi( this );

       m_curColumn = -1;
       m_curOpacity = 0;
50
       QHBoxLayout* chartLayout = new QHBoxLayout( chartFrame );
       m_chart = new Chart();
       chartLayout->addWidget( m_chart );

55     m_model.loadFromCSV( ":/data" );

       // Set up the diagram
       m_lines = new LineDiagram();
       m_lines->setModel( &m_model );
60     m_chart->coordinatePlane()->replaceDiagram( m_lines );

       // Instantiate the timer
       QTimer *timer = new QTimer(this);
       connect(timer, SIGNAL(timeout()), this, SLOT(slot_timerFired()));
```

```
    timer->start(40);
}

void MainWindow::on_lineTypeCB_currentIndexChanged( const QString & text )
{
    if ( text == "Normal" )
        m_lines->setType( LineDiagram::Normal );
    else if ( text == "Stacked" )
        m_lines->setType( LineDiagram::Stacked );
    else if ( text == "Percent" )
        m_lines->setType( LineDiagram::Percent );
    else
        qWarning (" Does not match any type");

    m_chart->update();
}

void MainWindow::on_paintValuesCB_toggled( bool checked )
{
    qDebug() << "MainWindow::on_paintValuesCB_toggled("<<checked<<")";
    const int colCount = m_lines->model()->columnCount(m_lines->rootIndex());
    for ( int iColumn = 0; iColumn<colCount; ++iColumn ) {
        DataValueAttributes a( m_lines->dataValueAttributes( iColumn ) );
        QBrush brush( m_lines->brush( iColumn ) );
        TextAttributes ta( a.textAttributes() );
        ta.setRotation( 0 );
        ta.setFont( QFont( "Comic", 10 ) );
        ta.setPen( QPen( brush.color() ) );

        if ( checked )
            ta.setVisible( true );
        else
            ta.setVisible( false );
        a.setVisible( true );
        a.setTextAttributes( ta );
        m_lines->setDataValueAttributes( iColumn, a );
    }
    m_chart->update();
}

void MainWindow::on_animateAreasCB_toggled( bool checked )
{
    if( checked ){
        highlightAreaCB->setCheckState( Qt::Unchecked );
        m_curColumn = 0;
    }else{
        m_curColumn = -1;
    }
    highlightAreaCB->setEnabled( ! checked );
    highlightAreaSB->setEnabled( ! checked );
    // un-highlight all previously highlighted columns
    const int colCount = m_lines->model()->columnCount();
    for ( int iColumn = 0; iColumn<colCount; ++iColumn )
        setHighlightArea( iColumn, 127, false, false );
    m_chart->update();
    m_curOpacity = 0;
}

void MainWindow::slot_timerFired()
{
    if( m_curColumn < 0 ) return;
    m_curOpacity += 5;
    if( m_curOpacity > 255 ){
        setHighlightArea( m_curColumn, 127, false, false );
        m_curOpacity = 0;
        ++m_curColumn;
        if( m_curColumn >= m_lines->model()->columnCount(m_lines->rootIndex()) )
            m_curColumn = 0;
    }
    setHighlightArea( m_curColumn, m_curOpacity, true, true );
```

```
135 }

    void MainWindow::setHighlightArea( int column, int opacity, bool checked, bool doUpda
    {
        LineAttributes la = m_lines->lineAttributes( m_lines->model()->index( 0, column,
140     if ( checked ) {
            la.setDisplayArea( true );
            la.setTransparency( opacity );
        }  else {
            la.setDisplayArea( false );
145     }
        m_lines->setLineAttributes( column, la );
        if( doUpdate )
            m_chart->update();
    }
150
    void MainWindow::on_highlightAreaCB_toggled( bool checked )
    {
        setHighlightArea( highlightAreaSB->value(), 127, checked, true );
    }
155
    void MainWindow::on_highlightAreaSB_valueChanged( int i )
    {
        Q_UNUSED( i );
        if ( highlightAreaCB->isChecked() )
160         on_highlightAreaCB_toggled( true );
        else
            on_highlightAreaCB_toggled( false);
    }

165
```

More explanation?

Figure 4.15. A Full featured Area Chart



High/Low Charts

Tip

High/low charts could be useful if your value's evolution during a time period should be visualized looking at small segments of this time. Often they are used to display a share's opening and closing value per day, as well as each day's maximum and minimum. Other possible uses of high/low charts include the changing temperature during the months: open/close would stand for a month's first and last day then.

Typically high/low charts are used to display one value evolving over the time—for showing more than one value consider using another chart type or use a second `KDChartWidget` positioned next to the first one. Another option might be to show a line chart for all of your values and draw special attention to one of them by adding an extra high/low chart featuring this value.The high/low chart could be positioned above the line chart—sharing its abscissa axis.

High/low charts are the most simple ones of KD Chart's statistical chart types and unlike the sophisticated Box&Whisker Charts they do not calculateeach dataset's high/low/open/close values but it is your obligation to pass these numbers into the data cells. This is done for optimization: most likely you have these figures anyway, if not it is extremely easy to determine them.

The advantage of not stuffing all your data values into KD Chart is that such very small datasets (holding just four cells each) enable your high/low charts to be displayed very quickly and even when representing a large number of days they will not consume too much memory.

High Low Charts can be configured with the following sub types as described in details below: Simple and Open/Close.

Simple High/Low Charts

Tip

Simple high/low charts (not showing open nor close values) can be useful to give an overview about an item's largest and smallest values per time-segment. You can use such a chart if your values beginning and end values are not interesting because the important information is represented by its minima and maxima: e.g. this might be the highest/lowest temperature reached in a year where your chart might show the evolution of these values during a span of some 100 years.

KD Chart by default draws normal high/low charts without showing any open or close marks when in High/Low Chart mode so no methods need to be called to get such charts, however after having used your to display another high/low chart type you can call to return to the default subtype.

Open/Close High/Low Charts

Tip

Extended open/close high/low charts provide a view on the start and end values as well as on the maximal and minimal values of a specific item per time-segment. This kind of chart is useful if your item's start and end value is of special importance too, e.g. typically such charts are used to visualize the evolution of per-day figures in a financial share value diagram displaying one or several weeks.

To make draw the open and close lines too you call the "classname" function "methodname". In case you are not interested in the open value (e.g. because this would always be identical to the respective previous close value) just use `HiLoClose` instead.

High/Low Charts with Custom Data Labels

Tip

Adding data labels to your High/Low chart can be useful if your chart is displaying a rather limited number items: otherwise it might be a better idea to draw the lines without showing the data labels but show a few labels instead using "method".

Box and Whisker Charts

This chart is called box&whisker because of its look: a box in the middle and two lines looking like whiskers on each side. While the box surrounds the center half of your spreading values, the upper and lower whisker ends are framing all or most of the values, except from a few outliers indicated separately.

Box&whisker charts provide detailed descriptive statistics of a variable: The height of the box shows how close together the main part of your values are and the length of the whiskers indicate how far the other values spread.

Tip

Since box&whisker charts give an overview of your values distribution (plus their mean and average value) they can be used for a quick estimate without looking into your statistical tables. An example might be a comparison of the number of failures in a device, perhaps three datasets of some 30 computer chips each to compare their errors at three different temperatures. Each of the data cells containing the number of failures

shown by one chip your chart would show three boxes: you might expect both a lower error number (with the box being drawn nearer to the abscissa axis) and less variation (smaller box and whiskers) at lower temperature.

Box&whisker charts are the most sophisticated ones of KD Chart's statistical chart types: unlike the simple High/Low Charts *they calculate* each dataset's statistical values themselves.

You can activate the box&whisker chart mode by calling the "classname" function "methodname"

Normal Box&Whisker Charts

Tip

Normal box&whisker charts are used to get a quick overview about your spreading data and see the show outliers.

Box&Whisker Charts without Outliers

Tip

If outlier values would dilute the message of your chart or simply be visually unpleasant, you can turn them off as described here.

To suppress drawing of any markers for *outliers* or *extremes* just set their size to zero by calling "methodname" as demonstrated by the tutorial file "filename", of course you could also use a value different from zero to set them to a specific size: "methodename" would increase their dynamic size by factor two since -25 is the default setting (a quarter of the box width), while "methodname" would set them to a fixed size of 15pt ignoring the size of the widget.

Note

This special mode might require some additional explanation to tell your users that the outliers and extremes are suppressed: otherwise the chart might be interpreted wrongly since it looks like *there are no* outliers but all values are within the inner fences or the box area.

Box&Whisker Charts with Statistics

Tip

By showing statistical values in your box&whisker charts, you can make your charts even more expressive - at the possible expense of readability.

Box&whisker charts can print up to ten different statistical figures next to the respective boxes—each of which can be shown using specific fonts, text color,and background colors.

Printing of a statistical value is enabled by passing one of the `KDChart::Params::BWStatVal` enum's descriptive names (`UpperOuterFence`, `UpperInnerFence`, `Quartile3`, `Median`, `Quartile1`, `LowerInnerFence`, `LowerOuterFence`, `MaxValue`, `MeanValue`, `MinValue`) to the following function:

Parameters:

`statValue`	one of the enum values listed above
`active`	set to true to have the `statValue` printed using either the default fond and color or the settings specified by the following parameters
`font`	if not zero the font will be used for this statistical value
`size`	if not zero this value will be interpreted as percent of the actual box width: font size will be calculated dynamically then instead of using the `font` parameter's fixed size
`color`	the text color of this statistical value
`brush`	the color of the background of this statistical value, if set to Qt::NoBrush the background is *not* erased before the text is printed

Compile and run the tutorial file to see a box&whisker chart featuring all possible statistical value texts using custom font settings and a special background color for the boxes

Note

No fence values are printed for the middle series in our sample because they are outside of the chart's data area. This can occur if all values of a series are in the inner fences and the box. Since in this case there is no need to show the fence values the range of the ordinate axis is *not* extended but the chart uses the available space to have more room for displaying the other datasets which otherwise would have less vertical space to draw their boxes and whiskers.

Polar coordinate plane

KD Chart makes use of the Polar coordinate system, and in particular its `KDChart::PolarCoordinatePlane` class for displaying chart types like Pie, Polar and Ring.

In this section we will describe and present each of the chart types which uses the Polar coordinate plane.

In general to implement a particular type of chart, just create an object of this type by calling `KDChart[type]Diagram`, or if your are using `KDChartWidget` you will need to call its `setType()` and specify the appropriate chart type. (e.g Widget::Pie, Widget::Polar etc...)

Pie Charts

Tip

Pie charts can be used to visualize the relative values of a few data cells (typically 2..20 values). Larger amounts of items can be hard to distinguish in a pie chart, so a Percent Bar Chart might fit your needs better then. Pie charts are most suitable if one of the data elements covers at least one forth, preferably even more of the total area.

A good example is the distribution of market shares among products or vendors.

While pie charts are nice for displaying *one* dataset there is a complementary chart type you might choose to visualize several datasets: the Ring Chart.

Note

The Ring Charts section describes a circular *multi-dataset* chart type.

Pie charts typically consist of two or more pieces any number of which can be shown 'exploded' (shifted away from the center) at different amounts, starting position of the first pie can be specified and your pie chart can be drawn in three-D look. Activating the pie chart mode is done by calling the `KDChartWidget` function `setType( KDChartWidget::Pie )` or by creating an object of this type using the `KDChartPieDiagram` class.

Note

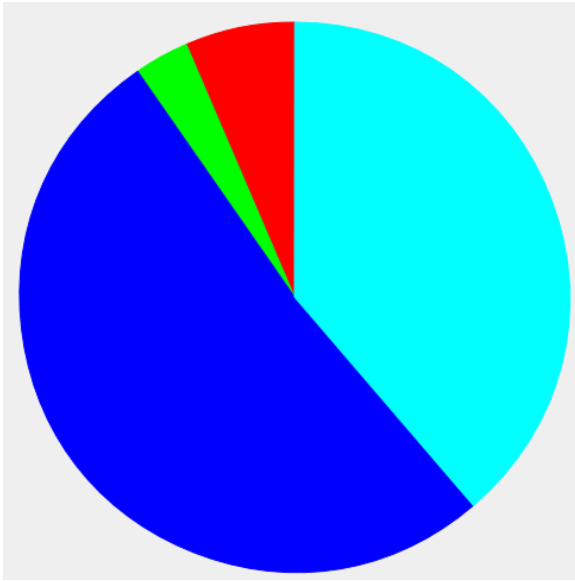
Three-dimensional look of the pies is no special feature you can enable by setting its ThreeD attributes, we will describe that more in details Chapter 5 - Customizing your Chart - ThreeD section further on.

Simple Pie Charts

Tip

A simple pie chart shows the data without emphasizing a special item.

Figure 4.16. A Simple Pie Chart



KD Chart by default draws two-dimensional pie charts when in pie chart mode so no method needs to be called to get one. We are describing more in details about how to obtain three dimensional look for a pie chart in the following Pie Attributes section.

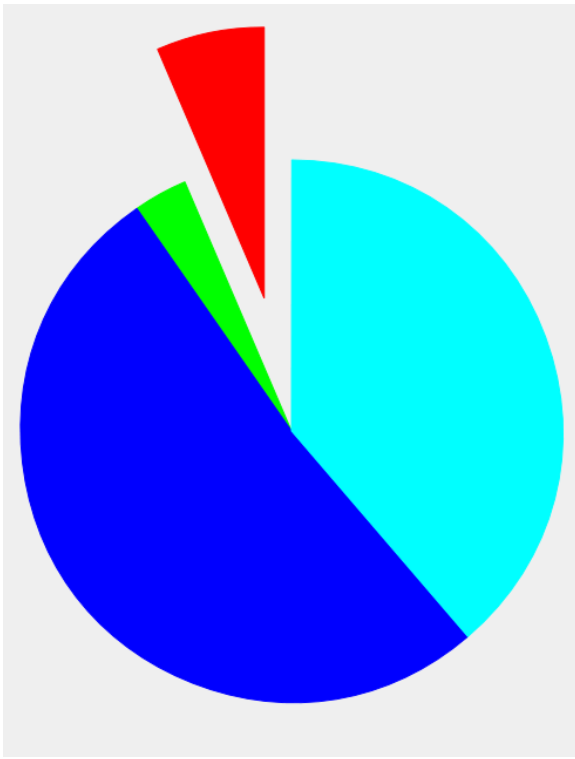
Compile and run the example "examples/Pie" to build a normal pie chart as presented above

Exploding Pie Charts

Tip

Exploding individual segments allows to emphasize individual data.

Figure 4.17. An Exploding Pie Chart



We will go through all the configuration possibilities in the Pie Attributes section below.

Pies Attributes

By "Pie attributes" we are talking about all parameters that can be configured and set by the user and which are specifics to the Pie Chart type. KD Chart 2.0 API separates the attributes specifics to a chart type itself and the generic attributes which are common to all chart types as for example the setters and getters for a brush or a pen.

All those attributes have a reasonnable default value that can simply be modified by the user by calling one of the diagram set function implemented on this purpose `KDChart-PieDiagram::setPieAttributes()`.

The procedure is straight forward:

- Create a `KDChart::PieAttributes` object by calling `KDChartPieDiagram::pieAttributes`.
- Configure this object using the setters available.
- Assign it to your Diagram with the help of one of the setters available in `KDChart::PieDiagram`. All the attributes can be configured to be applied for the whole diagram, for a column, or at a specified index (`QModelIndex`).

KD Chart 2.0 supports the following attributes for the Pie chart type. Each of those attributes can be set and retrieved the way we describe it in our example below:

- Explode: Enable/Disable exploding one or several datasets
- Explode factor: Set the explode factor for a dataset or for all slices
- StartPosition: Set the starting angle for the first dataset
- ThreeDPieAttributes: Enable/Disable threeD

To get a pie chart like the one presented above (having one or several of the pieces separated from the others in *exploded* mode) you would have to set its attributes and implicitely enable exploding, by calling `KDChartPieAttributes::setExplode` and then use the available methods to set the exploding factor for a particular dataset. as shown in the following code sample

```
// 1 - Create a PieAttribute object
PieAttributes pa = m_pie->PieAttributes( index );
// 2 - Enable exploding, point to a dataset and give the
// explode factor passing the dataset number and the factor
pa.setExplode( true );
pa.setExplodeFactor( 2, 0.5 );
// 3 - Assign to your diagram
m_pie->setPieAttributes( index, pa);
```

Note

Three-dimensional look of the pies can be enable and configured by setting its ThreeD attributes the same way as we are setting the PieAttributes in the code sample above, we will describe that more in details Chapter 5 - Customizing your Chart - ThreeD section further on.

Tips and Tricks

In this section we want to go through some examples about how to use some interesting features offered by the KD Chart 2.0 API. We will study the code and display a screen-shot showing the resulting widget.

A complete Pie Example

In the following implementation we want to be able to:

- Display the data values.
- Configure the Start position .
- Display a Pie chart and shift between normal and threeD look.
- Enable or disable the use of shadow colors when in threeD mode
- Set the surrounding lines width and color
- Explode one or several slices.
- Run an animation (exploding).

```
 1
                      /****************************************************************
    ** Copyright (C) 2006 Klar#vdalens Datakonsult AB.  All rights reserved.
    **
 5  ** This file is part of the KD Chart library.
    **
    ** This file may be distributed and/or modified under the terms of the
    ** GNU General Public License version 2 as published by the Free Software
    ** Foundation and appearing in the file LICENSE.GPL included in the
10  ** packaging of this file.
    **
    ** Licensees holding valid commercial KD Chart licenses may use this file in
    ** accordance with the KD Chart Commercial License Agreement provided with
    ** the Software.
15  **
    ** This file is provided AS IS with NO WARRANTY OF ANY KIND, INCLUDING THE
    ** WARRANTY OF DESIGN, MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
    **
    ** See http://www.kdab.net/kdchart for
20  **   information about KDChart Commercial License Agreements.
    **
    ** Contact info@kdab.net if any conditions of this
    ** licensing are not clear to you.
    **
25  **********************************************************************/

   #ifndef MAINWINDOW_H
   #define MAINWINDOW_H

30 #include "ui_mainwindow.h"
   #include <TableModel.h>

   class QTimer;
```

```
namespace KDChart {
    class Chart;
    class DatasetProxyModel;
    class PieDiagram;
}

class MainWindow : public QWidget, private Ui::MainWindow
{
    Q_OBJECT

public:
    MainWindow( QWidget* parent = 0 );

private slots:
    // start position
    void on_startPositionSB_valueChanged( double pos );
    void on_startPositionSL_valueChanged( int pos );

    // explode
    void on_explodeSubmitPB_clicked();
    void on_animateExplosionCB_toggled( bool toggle );
    void setExplodeFactor( int column, double value );

    // animation
    void slotNextFrame();

    // 3D
    void on_threeDGB_toggled( bool toggle );
    void on_threeDFactorSB_valueChanged( int factor );

private:
    KDChart::Chart* m_chart;
    TableModel m_model;
    KDChart::DatasetProxyModel* m_datasetProxy;
    KDChart::PieDiagram* m_pie;
    QTimer* m_timer;

    int m_currentFactor;
    int m_currentDirection;
    int m_currentSlice;
};

#endif /* MAINWINDOW_H */


```

More explanation h file?

```
/****************************************************************
** Copyright (C) 2006 Klar#vdalens Datakonsult AB.  All rights reserved.
**
** This file is part of the KD Chart library.
**
** This file may be distributed and/or modified under the terms of the
** GNU General Public License version 2 as published by the Free Software
** Foundation and appearing in the file LICENSE.GPL included in the
** packaging of this file.
**
** Licensees holding valid commercial KD Chart licenses may use this file in
** accordance with the KD Chart Commercial License Agreement provided with
** the Software.
**
** This file is provided AS IS with NO WARRANTY OF ANY KIND, INCLUDING THE
** WARRANTY OF DESIGN, MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
**
```

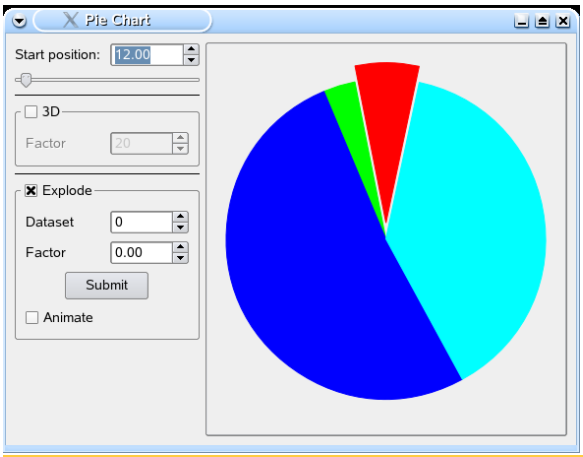
```
     ** See http://www.kdab.net/kdchart for
20   **   information about KDChart Commercial License Agreements.
     **
     ** Contact info@kdab.net if any conditions of this
     ** licensing are not clear to you.
     **
25   **********************************************************************/

   #include "mainwindow.h"

   #include <KDChartChart>
30 #include <KDChartAbstractCoordinatePlane>
   #include <KDChartPieDiagram>
   #include <KDChartTextAttributes>
   #include <KDChartDataValueAttributes>
   #include <KDChartMarkerAttributes>
35 #include <KDChartLegend>
   #include <KDChartPieAttributes>
   #include <KDChartThreeDPieAttributes>

   #include <QDebug>
40 #include <QPainter>
   #include <QTimer>

   using namespace KDChart;

45 MainWindow::MainWindow( QWidget* parent ) :
       QWidget( parent ), m_currentFactor( 0 ), m_currentDirection( 1 ), m_currentSlice(
   {
       setupUi( this );

50     QHBoxLayout* chartLayout = new QHBoxLayout( chartFrame );
       m_chart = new Chart();
       chartLayout->addWidget( m_chart );
       hSBar->setVisible( false );
       vSBar->setVisible( false );
55
       m_model.loadFromCSV( ":/data" );

       // Set up the diagram
       PolarCoordinatePlane* polarPlane = new PolarCoordinatePlane( m_chart );
60     m_chart->replaceCoordinatePlane( polarPlane );
       m_pie = new PieDiagram();
       m_pie->setModel( &m_model );
       m_chart->coordinatePlane()->replaceDiagram( m_pie );

65     m_timer = new QTimer( this );
       connect( m_timer, SIGNAL( timeout() ), this, SLOT( slotNextFrame() ) );
   }

   void MainWindow::on_startPositionSB_valueChanged( double pos )
70 {
       const int intValue = static_cast<int>( pos );
       startPositionSL->blockSignals( true );
       startPositionSL->setValue( intValue );
       startPositionSL->blockSignals( false );
75     // note: We use the global getter method here, it will fall back
       //       automatically to return the default settings.
       PieAttributes attrs( m_pie->pieAttributes() );
       attrs.setStartPosition( pos );
       m_pie->setPieAttributes( attrs );
80     update();
   }

   void MainWindow::on_startPositionSL_valueChanged( int pos )
   {
85     double doubleValue = static_cast<double>( pos );
       startPositionSB->blockSignals( true );
       startPositionSB->setValue( doubleValue  );
       startPositionSB->blockSignals( false );
```

```
        // note: We use the global getter method here, it will fall back
        //       automatically to return the default settings.
        PieAttributes attrs( m_pie->pieAttributes() );
        attrs.setStartPosition( pos );
        m_pie->setPieAttributes( attrs );
        update();
}

    void MainWindow::on_explodeSubmitPB_clicked()
    {
        setExplodeFactor( explodeDatasetSB->value(), explodeFactorSB->value() );
        update();
    }

    void MainWindow::setExplodeFactor( int column, double value )
    {
        // note: We use the per-column getter method here, it will fall back
        //       automatically to return the global (or even the default) settings.
        PieAttributes attrs( m_pie->pieAttributes( column ) );
        attrs.setExplodeFactor( value );
        m_pie->setPieAttributes( column, attrs );
}

    void MainWindow::on_animateExplosionCB_toggled( bool toggle )
    {
        if( toggle )
            m_timer->start( 100 );
        else
            m_timer->stop();
    }

void MainWindow::slotNextFrame()
    {
        m_currentFactor += ( 1 * m_currentDirection );
        if( m_currentFactor == 0 || m_currentFactor == 5 )
            m_currentDirection = -m_currentDirection;

        if( m_currentFactor == 0 ) {
            setExplodeFactor( m_currentSlice, 0.0 );
            m_currentSlice++;
            if( m_currentSlice == 4 )
                m_currentSlice = 0;
        }

        setExplodeFactor(
            m_currentSlice,
            static_cast<double>( m_currentFactor ) / 10.0 );
        update();
    }

    void MainWindow::on_threeDGB_toggled( bool toggle )
{
        // note: We use the global getter method here, it will fall back
        //       automatically to return the default settings.
        ThreeDPieAttributes attrs( m_pie->threeDPieAttributes() );
        attrs.setEnabled( toggle );
        attrs.setDepth( threeDFactorSB->value() );
        m_pie->setThreeDPieAttributes( attrs );
        update();
    }

void MainWindow::on_threeDFactorSB_valueChanged( int factor )
    {
        // note: We use the global getter method here, it will fall back
        //       automatically to return the default settings.
        ThreeDPieAttributes attrs( m_pie->threeDPieAttributes() );
        attrs.setEnabled( threeDGB->isChecked() );
        attrs.setDepth( factor );
        m_pie->setThreeDPieAttributes( attrs );
        update();
```

```
    }
160
```

More explanation cpp file?

Figure 4.18. A Full featured Pie Chart



Ring Charts

Tip

While a Pie Chart might be a good choice when displaying a single data series, using a ring chart might be ideal for visualizing a small amount of data cells stored in several datasets: ring charts can show both the relative values of the cells compared to their dataset's total value *and* the relation of the series totals compared to each other. This is done by using relative ring widths as shown below.

Ring charts (like Pie Charts) typically consist of two or more segments any number of which can be shown 'exploded' (shifted away from the center). To activate the ring chart mode simple call the `KDChartWidget` function `setType(Widget::Ring)` or create an object of type `KDChartRingDiagram`

Simple Ring Charts

Tip

If you do not care about the relative size of the sums of values in each datasets, simple ring charts are is the good choice.

KD Chart by default draws non-exploded rings with equal thickness when in ring chart mode so no methods need to be called.

Compile and run the example file "SimpleRing" to see obtain normal ring chart.

Ring Charts with Relative Thickness

Tip

Looking similar to a tree's annual rings these charts might be a good choice for displaying several years-related volumes of data like sales numbers. The segments could stand for the different product lines in a sortiment of goods. Looking at a ring's thicknesses you then could see the change in sales for *all* of your goods while a segment's length would show you how much this product line has contributed to the respective year's total turnover—compared to your other products.

Relative thickness mode is activated by calling the `KDChartRingAttributes` function `setRelativeThickness( true )` where each ring represents one dataset and the ring widths show the relations of the dataset totals to each other.

Compile and run the example RingRelativeThickness to see a ring chart featuring three datasets: the thickness of the middle ring shows clearly that this series represents the biggest total value.

Ring Charts Exploding Segments

Tip

Explode one or several of the segments of your ring chart to emphasize the respective data cell(s). You might use this for drawing attention to sales figures below a critical level of for highlighting a very successful item.

To have one or several segments of your ring chart shown exploded you need to to set its attributes and implicitely enable exploding, by calling `KDChartPieAttributes::setExplode` and then use the available methods to set the exploding factor for a particular dataset. as shown in the Ring Attribute section coming next.

In case you want to apply *different* explode factors to the segments just call the `KDChartRingAttributes::setExplodeFactors()` function and pass to it a `KD-`

`ChartRingAttributes::ExplodeFactorsMap` with one `double` value for every segment. The following ring chart has exploded all segments with values less than their ring's average while the smallest value is exploded even more.

Tip

You will find all you need to know for configuring your ring chart in our Ring Attribute section below.

Compile and run the example RingExploded to see a ring chart featuring both relative thickness of the rings and differently exploded segments on the outer ring.

Rings Attributes

By "Ring Attributes" we are talking about all parameters that can be configured and set by the user and which are specifics to the Ring Chart type. KD Chart 2.0 API separates the attributes specifics to a chart type itself and the generic attributes which are common to all chart types as for example the setters and getters for a brush or a pen.

All those attributes have a reasonnable default value that can simply be modified by the user by calling one of the diagram set function implemented on this purpose `KDChartRingDiagram::setRingAttributes()`.

The procedure is straight forward:

- Create a `KDChart::RingAttributes` object by calling `KDChartRingDiagram::ringAttributes`.
- Configure this object using the setters available.
- Assign it to your Diagram with the help of one of the setters available in `KDChart::RingDiagram`. All the attributes can be configured to be applied for the whole diagram, for a column, or at a specified index (`QModelIndex`).

KD Chart 2.0 supports the following attributes for the Ring chart type. Each of those attributes can be set and retrieved the way we describe it in our example below:

- Not implemented yet
- Not implemented yet
- Not implemented yet
- Not implemented yet

To get a ring chart like the one presented above (having one or several of the pieces sep-

arated from the others in *exploded* mode) you would have to set its attributes and implicitely enable exploding, by calling `KDChartRingAttributes::setExplode` and then use the available methods to set the exploding factor for a particular dataset. as shown in the following code sample

```
// 1 - Create a RingAttributes object
RingAttributes ra = m_ring->RingAttributes( index );
// 2 - Enable exploding, point to a dataset and give the
// explode factor passing the dataset number and the factor
ra.setExplode( true );
ra.setExplodeFactor( 2, 0.5 );
// 3 - Assign to your diagram
m_ring->setRingAttributes( index, pa);
```

Note

Only segments that are located on the *outer* ring can be exploded.

Tips and Tricks

In this section we want to go through some examples about how to use some interesting features offered by the KD Chart 2.0 API. We will study the code and display a screenshot showing the resulting widget.

A complete Ring Example

In the following implementation we want to be able to:

- Display the data values.
- Display a ring using most of its attributes
- Run an animation (exploding).

```
 1
                     /****************************************************************
    ** Copyright (C) 2006 Klar#vdalens Datakonsult AB.  All rights reserved.
    **
 5  ** This file is part of the KD Chart library.
    **
    ** This file may be distributed and/or modified under the terms of the
    ** GNU General Public License version 2 as published by the Free Software
    ** Foundation and appearing in the file LICENSE.GPL included in the
10  ** packaging of this file.
    **
    ** Licensees holding valid commercial KD Chart licenses may use this file in
    ** accordance with the KD Chart Commercial License Agreement provided with
    ** the Software.
15  **
```

```
     ** This file is provided AS IS with NO WARRANTY OF ANY KIND, INCLUDING THE
     ** WARRANTY OF DESIGN, MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
     **
     ** See http://www.kdab.net/kdchart for
     **   information about KDChart Commercial License Agreements.
     **
     ** Contact info@kdab.net if any conditions of this
     ** licensing are not clear to you.
     **
     **********************************************************************/

    #ifndef MAINWINDOW_H
    #define MAINWINDOW_H

    #include "ui_mainwindow.h"
    #include <TableModel.h>

    class QTimer;
    namespace KDChart {
        class Chart;
        class DatasetProxyModel;
        class PieDiagram;
    }

    class MainWindow : public QWidget, private Ui::MainWindow
    {
        Q_OBJECT

    public:
        MainWindow( QWidget* parent = 0 );

    private slots:
        // start position
        void on_startPositionSB_valueChanged( double pos );
        void on_startPositionSL_valueChanged( int pos );

        // explode
        void on_explodeSubmitPB_clicked();
        void on_animateExplosionCB_toggled( bool toggle );
        void setExplodeFactor( int column, double value );

        // animation
        void slotNextFrame();

        // 3D
        void on_threeDGB_toggled( bool toggle );
        void on_threeDFactorSB_valueChanged( int factor );

    private:
        KDChart::Chart* m_chart;
        TableModel m_model;
        KDChart::DatasetProxyModel* m_datasetProxy;
        KDChart::PieDiagram* m_pie;
        QTimer* m_timer;

        int m_currentFactor;
        int m_currentDirection;
        int m_currentSlice;
    };

    #endif /* MAINWINDOW_H */
```

More explanation h file?

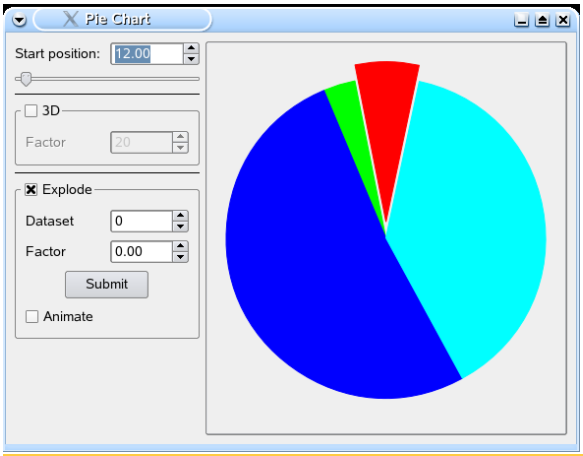
```
 1
                      /****************************************************************
    ** Copyright (C) 2006 Klar#vdalens Datakonsult AB.  All rights reserved.
    **
 5  ** This file is part of the KD Chart library.
    **
    ** This file may be distributed and/or modified under the terms of the
    ** GNU General Public License version 2 as published by the Free Software
    ** Foundation and appearing in the file LICENSE.GPL included in the
10  ** packaging of this file.
    **
    ** Licensees holding valid commercial KD Chart licenses may use this file in
    ** accordance with the KD Chart Commercial License Agreement provided with
    ** the Software.
15  **
    ** This file is provided AS IS with NO WARRANTY OF ANY KIND, INCLUDING THE
    ** WARRANTY OF DESIGN, MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
    **
    ** See http://www.kdab.net/kdchart for
20  **   information about KDChart Commercial License Agreements.
    **
    ** Contact info@kdab.net if any conditions of this
    ** licensing are not clear to you.
    **
25  ****************************************************************/

   #include "mainwindow.h"

   #include <KDChartChart>
30 #include <KDChartAbstractCoordinatePlane>
   #include <KDChartPieDiagram>
   #include <KDChartTextAttributes>
   #include <KDChartDataValueAttributes>
   #include <KDChartMarkerAttributes>
35 #include <KDChartLegend>
   #include <KDChartPieAttributes>
   #include <KDChartThreeDPieAttributes>

   #include <QDebug>
40 #include <QPainter>
   #include <QTimer>

   using namespace KDChart;

45 MainWindow::MainWindow( QWidget* parent ) :
       QWidget( parent ), m_currentFactor( 0 ), m_currentDirection( 1 ), m_currentSlice(
   {
       setupUi( this );

50     QHBoxLayout* chartLayout = new QHBoxLayout( chartFrame );
       m_chart = new Chart();
       chartLayout->addWidget( m_chart );
       hSBar->setVisible( false );
       vSBar->setVisible( false );
55
       m_model.loadFromCSV( ":/data" );

       // Set up the diagram
       PolarCoordinatePlane* polarPlane = new PolarCoordinatePlane( m_chart );
60     m_chart->replaceCoordinatePlane( polarPlane );
       m_pie = new PieDiagram();
       m_pie->setModel( &m_model );
       m_chart->coordinatePlane()->replaceDiagram( m_pie );

65     m_timer = new QTimer( this );
       connect( m_timer, SIGNAL( timeout() ), this, SLOT( slotNextFrame() ) );
   }

   void MainWindow::on_startPositionSB_valueChanged( double pos )
70 {
```

```
        const int intValue = static_cast<int>( pos );
        startPositionSL->blockSignals( true );
        startPositionSL->setValue( intValue );
        startPositionSL->blockSignals( false );
    75  // note: We use the global getter method here, it will fall back
        //       automatically to return the default settings.
        PieAttributes attrs( m_pie->pieAttributes() );
        attrs.setStartPosition( pos );
        m_pie->setPieAttributes( attrs );
    80  update();
    }

    void MainWindow::on_startPositionSL_valueChanged( int pos )
    {
    85  double doubleValue = static_cast<double>( pos );
        startPositionSB->blockSignals( true );
        startPositionSB->setValue( doubleValue  );
        startPositionSB->blockSignals( false );
        // note: We use the global getter method here, it will fall back
    90  //       automatically to return the default settings.
        PieAttributes attrs( m_pie->pieAttributes() );
        attrs.setStartPosition( pos );
        m_pie->setPieAttributes( attrs );
        update();
    95 }

    void MainWindow::on_explodeSubmitPB_clicked()
    {
        setExplodeFactor( explodeDatasetSB->value(), explodeFactorSB->value() );
   100  update();
    }

    void MainWindow::setExplodeFactor( int column, double value )
    {
   105  // note: We use the per-column getter method here, it will fall back
        //       automatically to return the global (or even the default) settings.
        PieAttributes attrs( m_pie->pieAttributes( column ) );
        attrs.setExplodeFactor( value );
        m_pie->setPieAttributes( column, attrs );
   110 }

    void MainWindow::on_animateExplosionCB_toggled( bool toggle )
    {
        if( toggle )
   115      m_timer->start( 100 );
        else
            m_timer->stop();
    }

   120 void MainWindow::slotNextFrame()
    {
        m_currentFactor += ( 1 * m_currentDirection );
        if( m_currentFactor == 0 || m_currentFactor == 5 )
            m_currentDirection = -m_currentDirection;
   125
        if( m_currentFactor == 0 ) {
            setExplodeFactor( m_currentSlice, 0.0 );
            m_currentSlice++;
            if( m_currentSlice == 4 )
   130          m_currentSlice = 0;
        }

        setExplodeFactor(
            m_currentSlice,
   135      static_cast<double>( m_currentFactor ) / 10.0 );
        update();
    }

    void MainWindow::on_threeDGB_toggled( bool toggle )
   140 {
```

```
        // note: We use the global getter method here, it will fall back
        //       automatically to return the default settings.
        ThreeDPieAttributes attrs( m_pie->threeDPieAttributes() );
        attrs.setEnabled( toggle );
        attrs.setDepth( threeDFactorSB->value() );
        m_pie->setThreeDPieAttributes( attrs );
        update();
    }

void MainWindow::on_threeDFactorSB_valueChanged( int factor )
    {
        // note: We use the global getter method here, it will fall back
        //       automatically to return the default settings.
        ThreeDPieAttributes attrs( m_pie->threeDPieAttributes() );
        attrs.setEnabled( threeDGB->isChecked() );
        attrs.setDepth( factor );
        m_pie->setThreeDPieAttributes( attrs );
        update();
    }
```

More explanation cpp file?

Figure 4.19. A Full featured Ring Chart



Polar Charts

Tip

Polar charts got their name from displaying "polar coordinates" instead of Cartesian coordinates. Currently only normalized polar charts can be shown: all values advance by the same number of polar degrees and there is no way to specify a data cell's angle individually. While this is ideal for

some situations it is not possible to display true world map data like this since you can not specify each cell's rotation angle. Transforming your coordinates to the Cartesian system and using a Point Chart might be a solution in such cases.

To activate the ring chart mode simply call the `KDChartWidget` function `setType(Widget::Polar)` or create an object of type `KDChartPolarDiagram`

A Simple Polar Charts

Just like the Line Charts to which they can be compared the polar chart type is divided into three sub types which can be activated by calling `setType( KDChartPolarDiagram::Normal )` or `Stacked` or `Percent`.

Compile and run the example file "PolarSimple" to see a normal polar chart as shown below.

Note

Data Values are shown by default in polar charts even if drawing of the markers is suppressed by `setPolarMarker( false )`, you can hide them by calling the `KDChartPolarAttributes` function `setPrintDataValues( false )`.

Polars Attributes

Text ...

Tips and Tricks

Some tips and tricks

A complete Polar Example

Let us make this more concrete by looking at the following lines of code.

More explanation?

What's next

Customizing your chart - Tips

Chapter 5. Customizing your Chart

Introduction text

Colors

Text ...

Colors Example

Let us make this more concrete by looking at the following lines of code.

More explanation?

Fonts

Text ...

Font Example

Let us make this more concrete by looking at the following lines of code.

More explanation?

Markers

Text ...

Markers Example

Let us make this more concrete by looking at the following lines of code.

More explanation?

ThreeD

Text ...

ThreeD Example

Let us make this more concrete by looking at the following lines of code.

More explanation?

Tips

Text ...

A cool Example

Let us make this more concrete by looking at the following lines of code.

More explanation?

What's next

Headers and footers.

Chapter 6. Header and Footers

Introduction text

How to configure

Text ...

Headers and Footers Example

Let us make this more concrete by looking at the following lines of code.

More explanation?

Tips

Text ...

A cool headers and footers Example

Let us make this more concrete by looking at the following lines of code.

More explanation?

What's next

Legends.

Chapter 7. Legends

Introduction text

How to configure

Text ...

Legend Example

Let us make this more concrete by looking at the following lines of code.

More explanation?

Tips

Text ...

Legend Example

Let us make this more concrete by looking at the following lines of code.

More explanation?

What's next

Axes.

Chapter 8. Axes

Introduction text

How to configure

Text ...

Axes Example

Let us make this more concrete by looking at the following lines of code.

More explanation?

Tips

Text ...

A cool Axes Example

Let us make this more concrete by looking at the following lines of code.

More explanation?

What's next

Advanced Charting.

Chapter 9. Advanced Charting

Introduction text

Frame and Background

Text ...

Frame and Background Example

Let us make this more concrete by looking at the following lines of code.

More explanation?

Data Value Manipulation

Text ...

Data Value Manipulation Example

Let us make this more concrete by looking at the following lines of code.

More explanation?

Axis Manipulation

Text ...

Axis Manipulation Example

Let us make this more concrete by looking at the following lines of code.

More explanation?

Grid Manipulation

Text ...

Grid Manipulation Example

Let us make this more concrete by looking at the following lines of code.

More explanation?

Interactive Charts

Text ...

Interactive Chart Example

Let us make this more concrete by looking at the following lines of code.

More explanation?

Multiple Charts

Text ...

Multiple Chart Example

Let us make this more concrete by looking at the following lines of code.

More explanation?

Zooming

Text ...

Zooming Example

Let us make this more concrete by looking at the following lines of code.

More explanation?

What's next

FAQ.

Appendix A. Q&A section

Note

This section will grow further according to the most frequently asked questions to our support.