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.
```

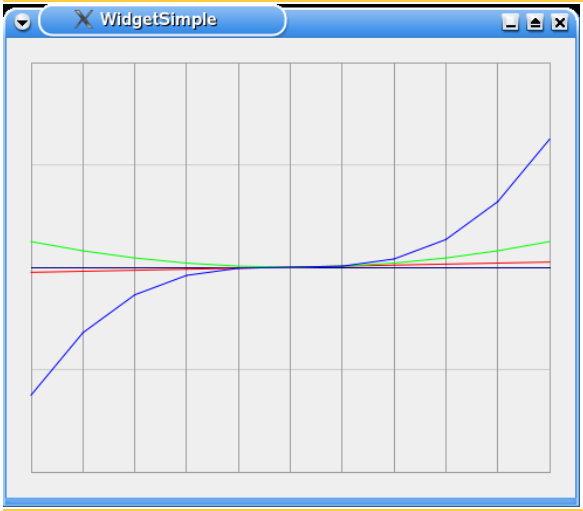
```
15  **
    ** This file is provided AS IS with NO WARRANTY OF ANY KIND, INCLUDING THE
    ** WARRANTY OF DESIGN, MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
    **
    ** See http://www.kdab.net/kdchart for
20  **   information about KDChart Commercial License Agreements.
    **
    ** 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

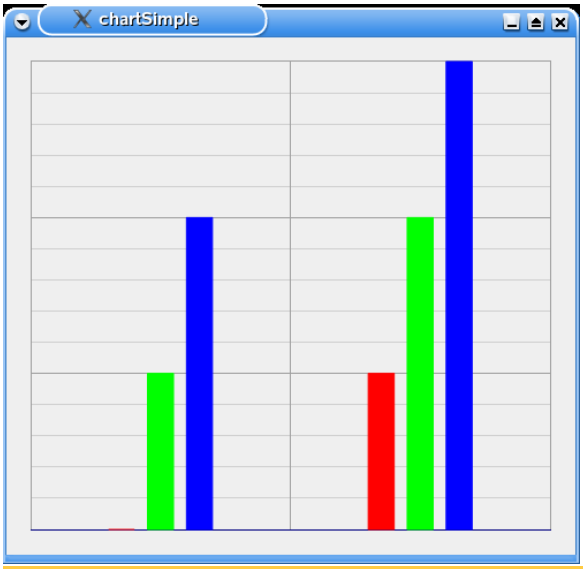
We recommend you to read, compile and run the following example. It is available at the following location of your KD Chart installation: `/examples/Charts/simple`

```
 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:

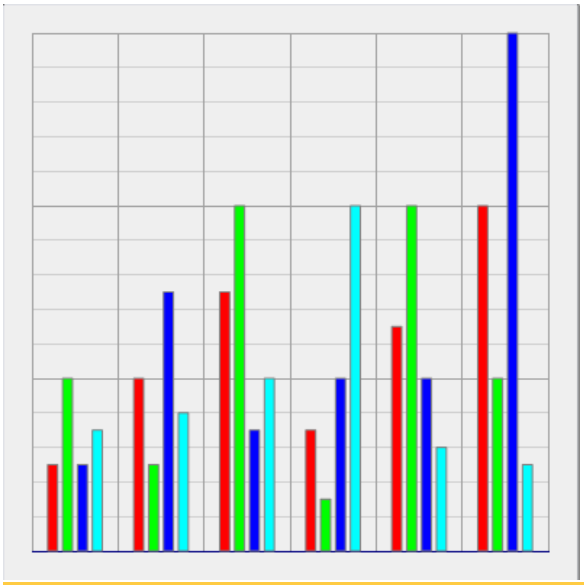
- Normal
- Stacked
- Percent

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



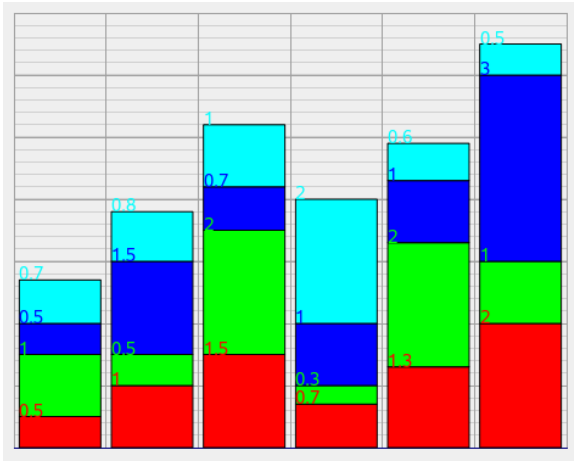
KD Chart's default type is the normal bar chart so no method needs to be called to get one, however after having used your `KDChartBarDiagram`. To display another sub-type you can return to the normal one by calling `setType( Normal )`.

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



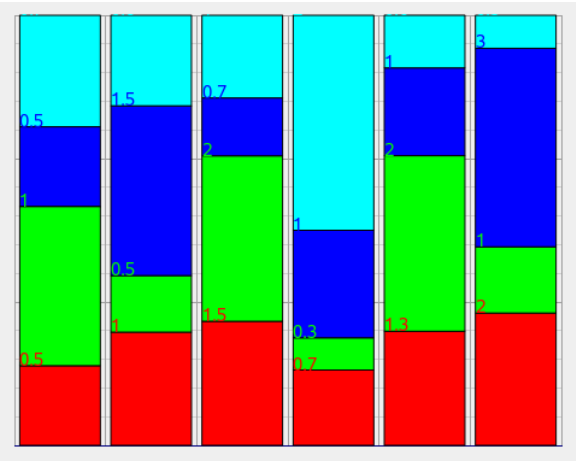
Stacked mode for bar charts is activated by calling the `KDChartBarDiagram` function `setType( Stacked )`.

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 );
```

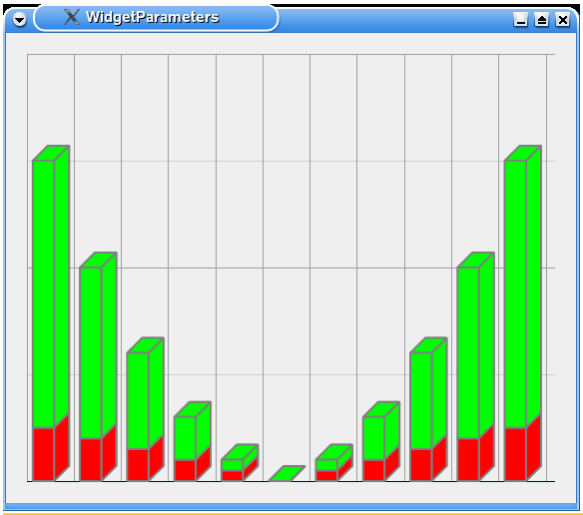
We want our bar chart to be displayed in ThreeD mode and need to configure some ThreeDBarAttributes and assing them to our diagram. Here we are configuring the depth of the ThreeD bars and enabling ThreeD mode. Depth is an attribute specific to Bar charts and its setter and getter are implemented into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

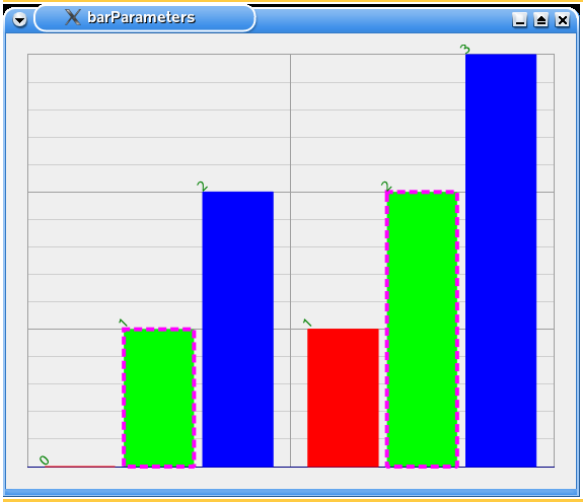
The Pen and the Brush setters and getters are implemented at a lower level in our `KDChartAbstractDiagram` class for a cleaner code structure. Those methods are of course used by all types of diagram and their configuration is very simple and straight forward as you can see in the above sample code. 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.

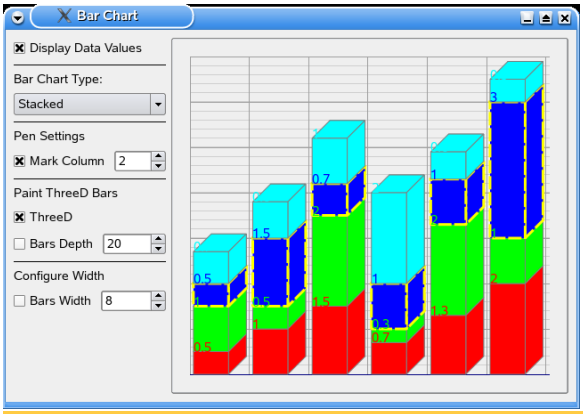
Finally we configure a typical `KDChartBarAttributes`, the Bar Width, for the user to be able to change the width of the bars dynamically increasing or decreasing its value via the Gui.

```
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:

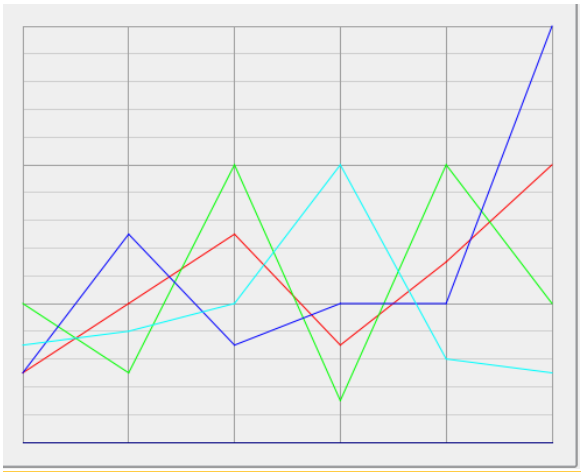
- Normal Line Chart
- Stacked Line Chart
- Percent Line chart

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



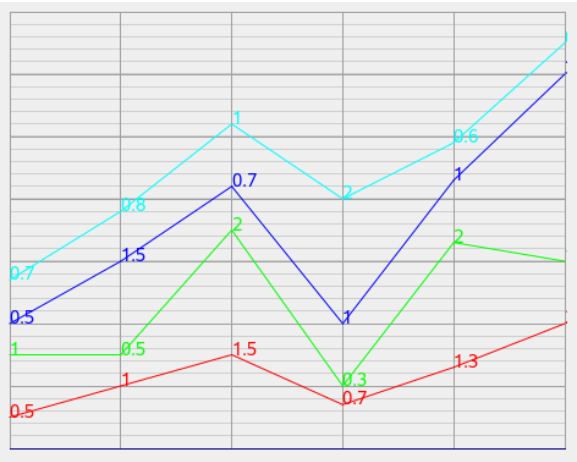
KD Chart draws normal line charts by default when in line chart mode so no method needs to be called to get one, however after having used your `KDChartLineDiagram` to display another line chart subtype you can reset it by calling `setType( Normal )`.

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



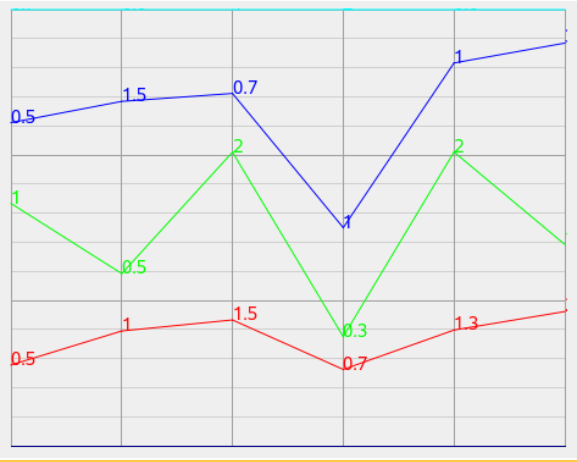
Stacked mode for line charts is activated by calling the `KDChartLineDiagram` method `setType( Stacked )`.

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.

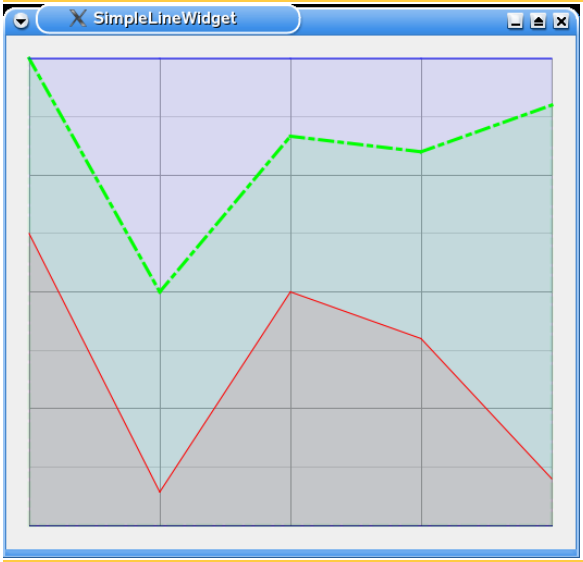
```
//Display in Area mode
LineAttributes ld;
ld.setDisplayArea( true );
//configure transparency
//it is nicer and let us
//all the area
ld.setTransparency( 25 );
widget.lineDiagram()->setLineAttributes( ld );
```

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 and methods 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:

- MissingValuesPolicy: Specifies how missing values will be shown in a line diagram.
- Display area: paint the area for a dataset.
- Area transparency: set the transparency for the displayed area color.

Note

All other attributes as ThreeDLineAttributes (specific to line charts), or MarkerAttributes, DataValueAttributes and TextAttributes ..etc.. available to all types of charts are of course also available to the line charts types and sub-types.

Line 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/Lines/Parameters` directory of your KD Chart installation.

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

```
#include <QtGui>
#include <KDChartChart>
#include <KDChartLineDiagram>
#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 Line diagram.

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( 0,5, QModelIndex() );
m_model.insertColumns( 0,5, QModelIndex() );

for( int i = 0; i < 5; ++i ) {
  for( int j = 0; j < 5; ++j ) {
    m_model.setData( m_model.index( i,j,QModelIndex() ), (double)i*j
  }
}

LineDiagram* diagram = new LineDiagram;
diagram->setModel(&m_model);
```

After having stored our data into the model, we create a diagram. In this case, we want to display a `KDChartLineDiagram`. As always we need to assign the model to our diagram. This procedure is of course similar for all types of diagrams.

We are now ready to configure our attributes. We want to display the data values and configure the text and font for those.

```
// Display values
// 1 - Call the relevant attributes
DataValueAttributes dva( diagram->dataValueAttributes() );
// 2 - We want to configure the font and colors
//     for the data values text.
TextAttributes ta( dva.textAttributes() );
//rotate if you wish
//ta.setRotation( 0 );
// 3 - Set up your text attributes
ta.setFont( QFont( "Comic", 6 ) );
ta .setPen( QPen( QColor( Qt::darkGreen ) ) );
ta.setVisible( true );
// 4 - Assign the text attributes to your
//     DataValuesAttributes
dva.setTextAttributes( ta );
dva.setVisible( true );
// 5 - Assign to the diagram
diagram->setDataValueAttributes( dva );
```

As for all attributes we call them by using the relevant method available from our diagram interface, here `diagram->dataValueAttributes()`. The second step is to set it up with our own values and finally we assign it to our diagram.

We could have displayed the data values without caring about settings its `KDChart-`

`TextAttributes`, 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 before we assign it to the diagram.

Note

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

We want to configure the Pen in order to draw a section of a line (dataset) differently. e.g. We want to focus the attention of the reader on this particular section.

```
// Draw a the section of a line differently.
// 1 - Retrieve the pen for the dataset and change
//    its style.
//    This allow us to keep the line original color.
QPen linePen(  diagram->pen( 1 ) );
linePen.setWidth( 3 );
linePen.setStyle( Qt::DashLine );
// 2 - Change the Pen for a section within a line
while assigning it to the diagram
diagram->setPen( m_model.index( 1, 1, QModelIndex() ), linePen );
```

Of course we could also have changed the pen for a single or all datasets as well. See how we call the pen for this very dataset before changing its style and width. This is done to keep its original color for consistancy. Alos

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 or get a Pen , configure it, call one of the setters methods available (See `KDChartAbstractDiagram.h` about those methods).

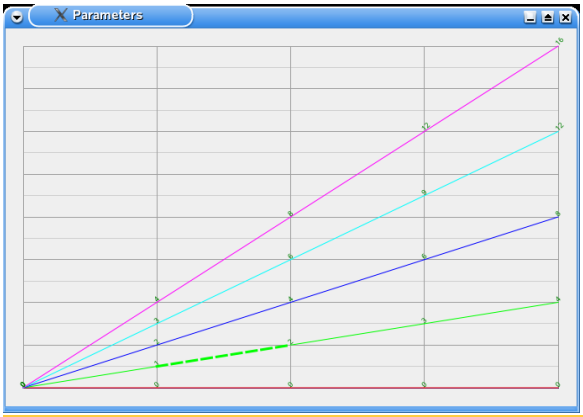
Our attribute having been configured and assigned we just need to assign our line diagram to our chart and conclude the implementation.

```
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 for all 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/Lines/Parameters` directory of your KD Chart installation.

Figure 4.11. Line With Configured Attributes



The subtype of a line chart (Normal, Stacked or Percent) is not set via its attribute class, but directly by using the diagram `KDChartLineDiagram::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 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 line chart subtype (Normal, percent, Stacked).
- Display Areas for one or several for one or several dataset(s).
- Run a small animation highlighting the areas one after the other.

To do so we will need to use several types of attributes and methods, as `KDChartAbstractDiagram::setPen()`, `KDCHartDataValueAttributes` and `KDChartTextAttributes`.

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 Line 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/Lines/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 LineDiagram;
   }

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

   public:
       MainWindow( QWidget* parent = 0 );

45 private slots:

       void on_lineTypeCB_currentIndexChanged( const QString & text );
       void on_paintValuesCB_toggled( bool checked );
       void on_animateAreasCB_toggled( bool checked );
50     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();

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


   #endif /* MAINWINDOW_H */
65
```

In the above code we bring up the KDChart namespace as usual and declare our slots. The purpose is to let the user configure its line chart attributes manually . As you can see we are using a KDChartChart object (m_chart), a KDChartLineDiagram object (m_lines), 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
    ** 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 <KDChartLineDiagram>
#include <KDChartTextAttributes>
#include <KDChartDataValueAttributes>


#include <QTimer>

using namespace KDChart;

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

    m_curColumn = -1;
    m_curOpacity = 0;

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

    m_model.loadFromCSV( ":/data" );

    // Set up the diagram
    m_lines = new LineDiagram();
    m_lines->setModel( &m_model );
    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 )
    {
 80     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 ) );
 85         TextAttributes ta( a.textAttributes() );
            ta.setRotation( 0 );
            ta.setFont( QFont( "Comic", 10 ) );
            ta.setPen( QPen( brush.color() ) );

 90         if ( checked )
                ta.setVisible( true );
            else
                ta.setVisible( false );
            a.setVisible( true );
 95         a.setTextAttributes( ta );
            m_lines->setDataValueAttributes( iColumn, a );
        }
        m_chart->update();
    }
100
    void MainWindow::on_animateAreasCB_toggled( bool checked )
    {
        if( checked ){
            highlightAreaCB->setCheckState( Qt::Unchecked );
105         m_curColumn = 0;
        }else{
            m_curColumn = -1;
        }
        highlightAreaCB->setEnabled( ! checked );
110     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 );
115     m_chart->update();
        m_curOpacity = 0;
    }

    void MainWindow::slot_timerFired()
120 {
        if( m_curColumn < 0 ) return;
        m_curOpacity += 5;
        if( m_curOpacity > 255 ){
            setHighlightArea( m_curColumn, 127, false, false );
125         m_curOpacity = 0;
            ++m_curColumn;
            if( m_curColumn >= m_lines->model()->columnCount(m_lines->rootIndex()) )
                m_curColumn = 0;
        }
130     setHighlightArea( m_curColumn, m_curOpacity, true, true );
    }

    void MainWindow::setHighlightArea( int column, int opacity, bool checked, bool doUpda
    {
135     LineAttributes la = m_lines->lineAttributes( m_lines->model()->index( 0, column,
        if ( checked ) {
            la.setDisplayArea( true );
            la.setTransparency( opacity );
        }  else {
140         la.setDisplayArea( false );
        }
        m_lines->setLineAttributes( column, la );
        if( doUpdate )
            m_chart->update();
145 }
```

```
    void MainWindow::on_highlightAreaCB_toggled( bool checked )
    {
        setHighlightArea( highlightAreaSB->value(), 127, checked, true );
150 }

    void MainWindow::on_highlightAreaSB_valueChanged( int i )
    {
        Q_UNUSED( i );
155     if ( highlightAreaCB->isChecked() )
            on_highlightAreaCB_toggled( true );
        else
            on_highlightAreaCB_toggled( false);
    }
160
```

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 line diagram. We also want to set up a `QTimer` to be able to run our animation. Finally we assign the diagram to our chart.

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

 m_model.loadFromCSV( ":/data" );

 // Set up the diagram
 m_lines = new LineDiagram();
 m_lines->setModel( &m_model );
 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);
 ...
```

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_lines->setType( LineDiagram::Normal );
else if ( text == "Stacked" )
   m_lines->setType( LineDiagram::Stacked );
else if ( text == "Percent" )
   m_lines->setType( LineDiagram::Percent );
....
m_chart->update();
```

We want the user to be able to display or hide the data values from the GUI, and also change the default font for our data values labels to make it nicer.

```
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();
```

In the code above, we make sure our data values labels will be painted using the dataset default color by retrieving the brush for each dataset and assigning the color of the brush to the pen.

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.

The user should be able to display the area for one or several dataset.

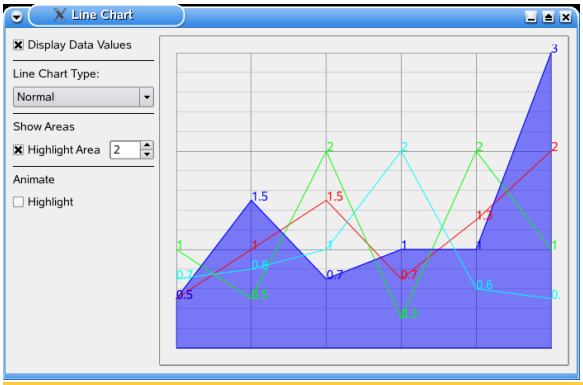
```
....
LineAttributes la =
m_lines->lineAttributes(
m_lines->model()->index( 0, column, m_lines->rootIndex() ) );
if ( checked ) {
 la.setDisplayArea( true );
 la.setTransparency( opacity );
}  else {
 la.setDisplayArea( false );
}
m_lines->setLineAttributes( column, la );
...
m_chart->update();
...
```

This is implemented by configuring our line attributes and assign them by dataset to the diagram, as shown above.

The same procedure is used for us to be able to run our animation. You can of course learn more about this part of the code which is more related to Qt programming by consulting `examples/Lines/Advanced/mainwindow.cpp`.

This example is available to compile and run from the `examples/Lines/Advanced` directory into your KD Chart installation. We recommend you to run it. The widget displayed by the above code is shown in the figure below.

Figure 4.12. A Full featured Line Chart



Note

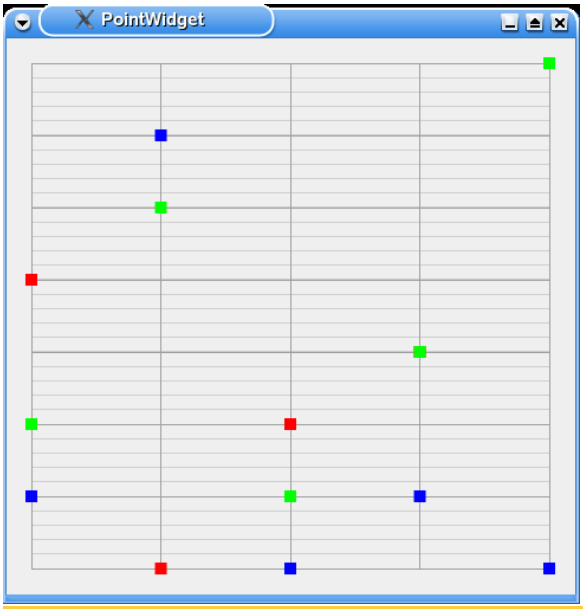
The following sections about Point charts and Area are tightly related to line charts. Point charts are line diagrams with Markers (lines themselves are not painted). Area charts are also line charts with the area below the lines, filled by the respective dataset's color.

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 4.13. A Point Chart



Note

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 diagram and configure its pen to Qt::NoPen.
- Display its data values marker attributes.

Point Sample Code

The following code sample is going through the process described above to obtain a very simple point chart. It is based on the `examples/Widget/Simple` which code has been slightly modified to display a Point diagram.

```
                    ...
```

```
//  Hide the lines
widget.lineDiagram()->setPen(  Qt::NoPen );
// Set up the Attributes
DataValueAttributes dva( widget.lineDiagram()->dataValueAttributes()
MarkerAttributes ma(  dva.markerAttributes() );
TextAttributes ta(  dva.textAttributes() );
ma.setVisible( true );
// display values or not
ta.setVisible(  false );
dva.setTextAttributes(  ta );
dva.setMarkerAttributes(  ma );
dva.setVisible( true );

widget.lineDiagram()->setDataValueAttributes( dva );
```

This sample code is making use of a `KDChartWidget` and a `KDChartLineDiagram` but of course the process is very similar if we were working with a `KDChartChart`.

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 subjects 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.
- Shift between points and lines charts

We are using a `KDChartChart` class and also an home made `TableModel` for the convenience. It is 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 Line 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/Lines/PointChart` 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 LineDiagram;
   }

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

   public:
       MainWindow( QWidget* parent = 0 );

45 private slots:
```

```
        void on_lineTypeCB_currentIndexChanged( const QString & text );
        void on_paintValuesCB_toggled( bool checked );
        void on_paintMarkersCB_toggled( bool checked );
        void on_paintLinesCB_toggled( bool checked );
        void on_markersStyleCB_currentIndexChanged( const QString & text );
        void on_markersWidthSB_valueChanged( int i );
        void on_markersHeightSB_valueChanged( int i);

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

    #endif /* MAINWINDOW_H */


```

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

The implementation is similar to the line chart implementation presented earlier:

```
 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 <KDChartLineDiagram>
    #include <KDChartTextAttributes>
    #include <KDChartDataValueAttributes>


35  using namespace KDChart;

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

```
    {
40      setupUi( this );

        QHBoxLayout* chartLayout = new QHBoxLayout( chartFrame );
        m_chart = new Chart();
        chartLayout->addWidget( m_chart );
45
        m_model.loadFromCSV( ":/data" );

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

    }

55  void MainWindow::on_lineTypeCB_currentIndexChanged( const QString & text )
    {
        if ( text == "Normal" )
            m_lines->setType( LineDiagram::Normal );
        else if ( text == "Stacked" )
60          m_lines->setType( LineDiagram::Stacked );
        else if ( text == "Percent" )
            m_lines->setType( LineDiagram::Percent );
        else
            qWarning (" Does not match any type");
65
        m_chart->update();
    }

    void MainWindow::on_paintValuesCB_toggled( bool checked )
70  {
        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 ) );
75          TextAttributes ta( a.textAttributes() );
            ta.setRotation( 0 );
            ta.setFont( QFont( "Comic", 10 ) );
            ta.setPen( QPen( brush.color() ) );

80          if ( checked )
                ta.setVisible( true );
            else
                ta.setVisible( false );
            a.setVisible( true );
85          a.setTextAttributes( ta );
            m_lines->setDataValueAttributes( iColumn, a );
        }
        m_chart->update();
    }
90

    void MainWindow::on_paintLinesCB_toggled(  bool checked )
    {
        const int colCount = m_lines->model()->columnCount(m_lines->rootIndex());
95      for ( int iColumn = 0; iColumn<colCount; ++iColumn ) {
            DataValueAttributes a( m_lines->dataValueAttributes( iColumn ) );
            QBrush lineBrush( m_lines->brush( iColumn ) );
            if ( checked ) {
                QPen linePen( lineBrush.color() );
100             m_lines->setPen(  iColumn,  linePen );
            }
            else
                m_lines->setPen( iColumn,  Qt::NoPen );
        }
105         m_chart->update();
    }
```

```
    void MainWindow::on_paintMarkersCB_toggled( bool checked )
{
        // set up a map with different marker styles
        MarkerAttributes::MarkerStylesMap map;
        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 );


        const int colCount = m_lines->model()->columnCount(m_lines->rootIndex());
        for ( int iColumn = 0; iColumn<colCount; ++iColumn ) {
            DataValueAttributes dva( m_lines->dataValueAttributes( iColumn ) );
            TextAttributes ta ( dva.textAttributes() );
            if (  paintValuesCB->isChecked() )
                ta.setVisible(  true );
            else
                ta.setVisible( false );
            MarkerAttributes ma( dva.markerAttributes() );
            ma.setMarkerStylesMap( map );
            ma.setMarkerSize( QSize( markersWidthSB->value(),
                                     markersHeightSB->value() ) );

            switch ( markersStyleCB->currentIndex() ) {
            case 0:
                break;
            case 1:
                ma.setMarkerStyle( MarkerAttributes::MarkerCircle );
                break;
            case 2:
                ma.setMarkerStyle( MarkerAttributes::MarkerSquare );
                break;
            case 3:
                ma.setMarkerStyle( MarkerAttributes::MarkerDiamond );
                break;
            case 4:
                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.setVisible(  true );
            dva.setTextAttributes( ta );
            dva.setMarkerAttributes( ma );

            if (  checked )
                dva.setVisible( true );
            else
                dva.setVisible(  false );
            m_lines->setDataValueAttributes( iColumn, dva );
        }

        m_chart->update();
    }

    void MainWindow::on_markersStyleCB_currentIndexChanged( const QString & text )
    {
```

```
        Q_UNUSED( text );
180     if ( paintMarkersCB->isChecked() )
            on_paintMarkersCB_toggled( true );
    }


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

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

200
```

Here we will not comment in details the code as it is similar to what we have seen before in our line chart example, but only pick up the interesting part of it.

In order to get a point chart we paint or hide the lines by setting our line diagram pen:

```
                    void MainWindow::on_paintLinesCB_toggled(  bool 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 lineBrush( m_lines->brush( iColumn ) );
                          if ( checked ) {
                            QPen linePen( lineBrush.color() );
                            m_lines->setPen(  iColumn,  linePen );
                          }
                          else
                            m_lines->setPen( iColumn,  Qt::NoPen );
                          }
                    m_chart->update();
                    }
```

We need to retrieve the pen color before resetting it to its original value, and do that by looping through the datasets.

Note

It is important to know that have three levels of precedence when setting the attributes:

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

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.

For us to be able to store different Markers style we make use of `MarkerAttributes::MarkerStylesMap map` which is very convenient in this case.

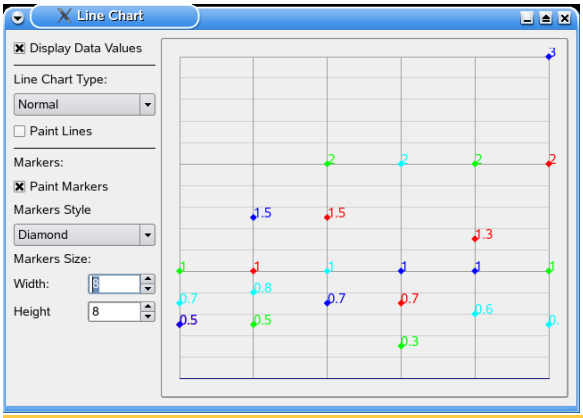
```
...
MarkerAttributes::MarkerStylesMap map;
map.insert( 0, MarkerAttributes::MarkerSquare );
map.insert( 1, MarkerAttributes::MarkerCircle );
map.insert( 2, MarkerAttributes::MarkerRing );
map.insert( 3, MarkerAttributes::MarkerCross );
map.insert( 4, MarkerAttributes::MarkerDiamond );
...
MarkerAttributes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

The user may also change the size of the marker form the GUI and this is implemented straight forward by using `KDChartMarkerAttributes` method `setMarkerSize()`.

```
ma.setMarkerSize( QSize( markersWidthSB->value(),
                         markersHeightSB->value() ) );
```

This example is available to compile and run from the `examples/Lines/PointChart` directory into your KD Chart installation. We recommend you to run it. The widget displayed by the above code is shown in the figure below.

Figure 4.14. 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.

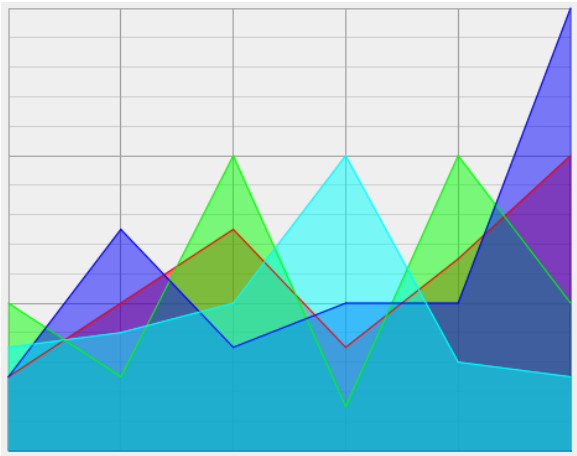
For example the area chart type might be the best choice for showing how several sources contributed to increasing ozone values in a conurbation during a summer's months.

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 4.15. 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 Sample Code

Let us make this more concrete by looking at the following lines of code and reproduce the process described above:

```
// 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 ...).

Note

`KDChartLineAttributes` can be set for the whole diagram, for a dataset or for a specific index (see sample code above), as for any other attributes.

Area Attributes

There are no specifical attributes related to the Area chart. As explained above Area charts display mode 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 screen-shot showing the resulting widget.

A complete Area Example

Note

This example has already been presented in details `Section - A Complete Line Example`. You dont need to go through it, if you already have studied the section above.

In the following implementation we want to be able to:

- Display data values
- Shift line types (Normal, Stacked, Percent)

- Display areas for each dataset on its own and for the whole diagram

We are using a `KDChartChart` class and also an home made `TableModel` for the convenience. It is 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 Line 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/Lines/Advance` 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 LineDiagram;
   }

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

```
    public:
        MainWindow( QWidget* parent = 0 );

45  private slots:

        void on_lineTypeCB_currentIndexChanged( const QString & text );
        void on_paintValuesCB_toggled( bool checked );
        void on_animateAreasCB_toggled( bool checked );
50      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();

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


    #endif /* MAINWINDOW_H */
65
```

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

The implementation is similar to the line chart implementation presented earlier:

```
 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 <KDChartLineDiagram>
    #include <KDChartTextAttributes>
    #include <KDChartDataValueAttributes>
```

```
35 #include <QTimer>

   using namespace KDChart;

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

       m_curColumn = -1;
45     m_curOpacity = 0;

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

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

       // Instantiate the timer
       QTimer *timer = new QTimer(this);
60     connect(timer, SIGNAL(timeout()), this, SLOT(slot_timerFired()));
       timer->start(40);
   }

   void MainWindow::on_lineTypeCB_currentIndexChanged( const QString & text )
65 {
       if ( text == "Normal" )
           m_lines->setType( LineDiagram::Normal );
       else if ( text == "Stacked" )
           m_lines->setType( LineDiagram::Stacked );
70     else if ( text == "Percent" )
           m_lines->setType( LineDiagram::Percent );
       else
           qWarning (" Does not match any type");

75     m_chart->update();
   }

   void MainWindow::on_paintValuesCB_toggled( bool checked )
   {
80     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 ) );
85         TextAttributes ta( a.textAttributes() );
           ta.setRotation( 0 );
           ta.setFont( QFont( "Comic", 10 ) );
           ta.setPen( QPen( brush.color() ) );

90         if ( checked )
               ta.setVisible( true );
           else
               ta.setVisible( false );
           a.setVisible( true );
95         a.setTextAttributes( ta );
           m_lines->setDataValueAttributes( iColumn, a );
       }
       m_chart->update();
   }
100
   void MainWindow::on_animateAreasCB_toggled( bool checked )
   {
```

```
        if( checked ){
            highlightAreaCB->setCheckState( Qt::Unchecked );
105         m_curColumn = 0;
        }else{
            m_curColumn = -1;
        }
        highlightAreaCB->setEnabled( ! checked );
110     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 );
115     m_chart->update();
        m_curOpacity = 0;
    }

    void MainWindow::slot_timerFired()
120 {
        if( m_curColumn < 0 ) return;
        m_curOpacity += 5;
        if( m_curOpacity > 255 ){
            setHighlightArea( m_curColumn, 127, false, false );
125         m_curOpacity = 0;
            ++m_curColumn;
            if( m_curColumn >= m_lines->model()->columnCount(m_lines->rootIndex()) )
                m_curColumn = 0;
        }
130     setHighlightArea( m_curColumn, m_curOpacity, true, true );
    }

    void MainWindow::setHighlightArea( int column, int opacity, bool checked, bool doUpda
    {
135     LineAttributes la = m_lines->lineAttributes( m_lines->model()->index( 0, column,
        if ( checked ) {
            la.setDisplayArea( true );
            la.setTransparency( opacity );
        }  else {
140         la.setDisplayArea( false );
        }
        m_lines->setLineAttributes( column, la );
        if( doUpdate )
            m_chart->update();
145 }

    void MainWindow::on_highlightAreaCB_toggled( bool checked )
    {
        setHighlightArea( highlightAreaSB->value(), 127, checked, true );
150 }

    void MainWindow::on_highlightAreaSB_valueChanged( int i )
    {
        Q_UNUSED( i );
155     if ( highlightAreaCB->isChecked() )
            on_highlightAreaCB_toggled( true );
        else
            on_highlightAreaCB_toggled( false);
    }
160
```

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 line diagram. We also want to set up a `QTimer` to be able to run our animation. Finally we assign the diagram to our chart.

```
                    ...
```

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

 m_model.loadFromCSV( ":/data" );

 // Set up the diagram
 m_lines = new LineDiagram();
 m_lines->setModel( &m_model );
 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);
 ...
```

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_lines->setType( LineDiagram::Normal );
else if ( text == "Stacked" )
   m_lines->setType( LineDiagram::Stacked );
else if ( text == "Percent" )
   m_lines->setType( LineDiagram::Percent );
....
m_chart->update();
```

We want the user to be able to display or hide the data values from the GUI, and also change the default font for our data values labels to make it nicer.

```
 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();
```

In the code above, we make sure our data values labels will be painted using the dataset default color by retrieving the brush for each dataset and assigning the color of the brush to the pen.

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.

The user should be able to display the area for one or several dataset.

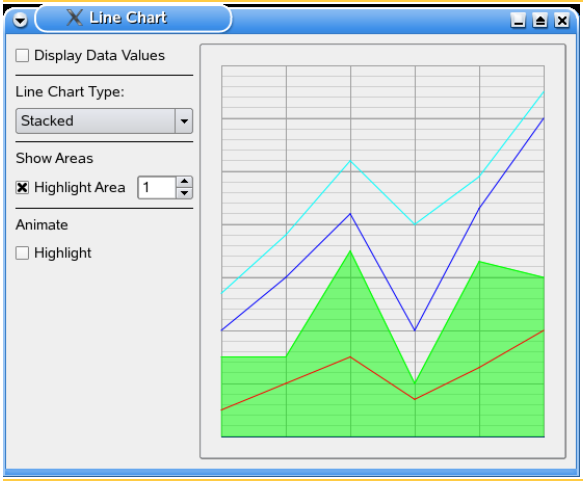
```
....
LineAttributes la =
m_lines->lineAttributes(
m_lines->model()->index( 0, column, m_lines->rootIndex() ) );
if ( checked ) {
 la.setDisplayArea( true );
 la.setTransparency( opacity );
}  else {
 la.setDisplayArea( false );
}
m_lines->setLineAttributes( column, la );
...
m_chart->update();
...
```

This is implemented by configuring our line attributes and assign them by dataset to the diagram, as shown above.

The same procedure is used for us to be able to run our animation. You can of course learn more about this part of the code which is more related to Qt programming by consulting `examples/Lines/Advanced/mainwindow.cpp`.

This example is available to compile and run from the `examples/Lines/Advanced` directory into your KD Chart installation. We recommend you to run it. The widget displayed by the above code is shown in the figure below.

Figure 4.16. A Full featured Area Chart



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, a circular multi dataset chart type described in the `Ring Charts` section furhter on.

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

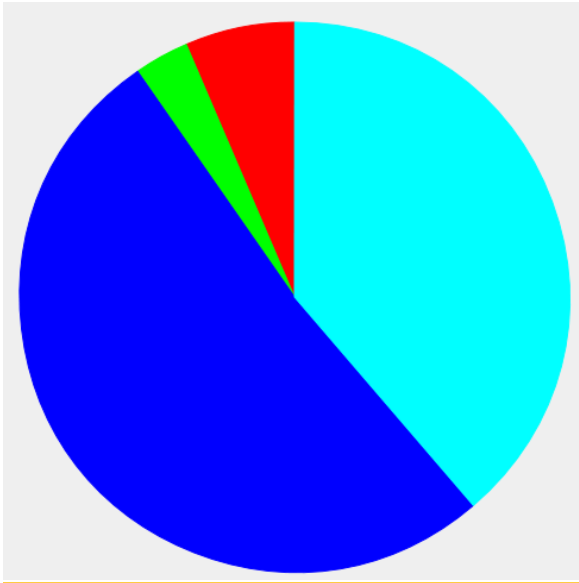
Three-dimensional look of the pies is no special feature you can enable by setting its ThreeD attributes, we will describe that more in details Chapter 5 - Customizing your Chart - ThreeD section further on.

Simple Pie Charts

Tip

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

Figure 4.17. A Simple Pie Chart



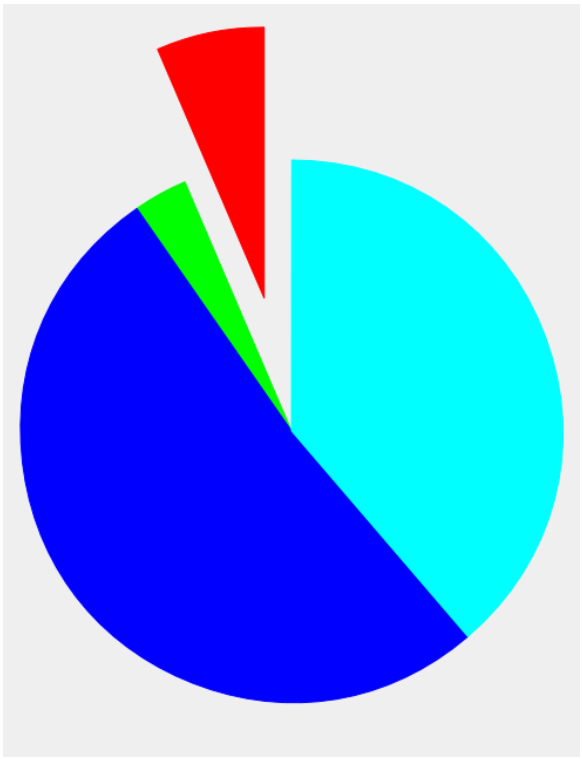
KD Chart by default draws two-dimensional pie charts when in pie chart mode so no method needs to be called to get one. We are describing more in details about how to obtain three dimensional look for a pie chart in the following Pie Attributes section.

Exploding Pie Charts

Tip

Explode individual segments to emphasize individual data.

Figure 4.18. An Exploding Pie Chart



We will go through all the configuration possibilities in the Pie Attributes section below, but let us study some code sample first.

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 Pie 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 <KDChartPieDiagram>
#include <QPen>

using namespace KDChart;
```

We need to include `KDChartPieDiagram` 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.setType(  Widget::Pie );
```

We just need to change the default chart type (Line Charts) by calling the `KDChartWidget::setType` method.

Now let us configure a Pen to draw a line arount the Pie and its section

```
QPen piePen( widget.pieDiagram()->pen() );
piePen.setWidth( 3 );
piePen.setColor( Qt::yellow );
// call your diagram and set the new pen
widget.pieDiagram()->setPen( 2, piePen );
```

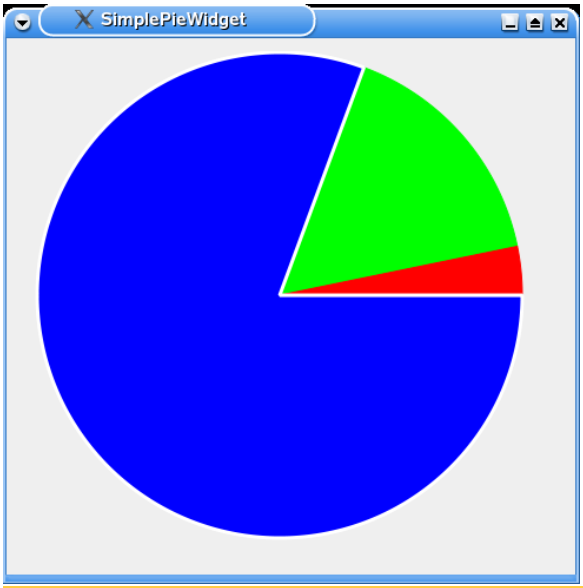
Here we are configuring the pen "attribute". As you can see it is straight forward. `KDChartWidget::pieDiagram()` allow us to get a pointer to our widget diagram. As you can see it is very simple to assign a new pen to our diagram by calling the diagram `KDChartAbstractDiagram::setPen()` method.

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 4.19. A Simple Pie Widget



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

Note

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

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 (See `KDChartAbstractDiagram` or `KDChartPieAbstractDiagram` etc...

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 `KDChartPieDiagram::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 pie piece(s)
- Explode factor: The explode factor is a qreal between 0 and 1, it is interpreted as a percentage of the total available radius.
- StartPosition: Set the starting angle for the first dataset. Can only be specified for the whole diagram.
- Granularity: Set the granularity: the smaller the granularity the more your diagram-segments will show facettes instead of rounded segments. Can only be specified for the whole diagram.
- PieAttributes: set or retrieve the pie diagram Attributes. (see: `KDChartAbstractPieDiagram`)
- ThreeDPieAttributes: set or retrieve the diagram ThreeDAttributes. (see: `KDChartAbstractPieDiagram`)

Tip

The default explode factor is 10 percent; use `setExplodeFactor` to specify a different factor. This is a convenience function: Calling `setExplode( true )` does the same as calling `setExplodeFactor( 0.1 )`, and calling `setExplode( false )` does the same as calling `setExplodeFactor( 0.0 )`.

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 by calling `KDChartPieAttributes::setExplode` or `KDChartPieAttributes::setExplodeFactor` if you want to change the explode factore default value and then use the available methods to assing those attributes to your diagram as shown in the following code sample

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

Pie 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/Lines/Parameters` directory of your KD Chart installation.

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

```
#include <QtGui>
#include <KDChartChart>
#include <KDChartPieDiagram>
#include <KDChartPieAttributes>

using namespace KDChart;
```

We have included `KDChartPieAttributes` to be able to configure exploding for one of the pie slice. Those attributes are specifical to the Pie types.

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.

```
m_model.insertRows( 0, 1, QModelIndex() );
m_model.insertColumns(  0,  6,  QModelIndex() );
for (int row = 0; row < 1; ++row) {
  for (int column = 0; column < 6; ++column) {
```

```
            QModelIndex index =
        m_model.index(row, column, QModelIndex());
            m_model.setData(index, QVariant(row+1 * column+1) );
          }
        }
        // We need a Polar plane for the Pie type
        PolarCoordinatePlane* polarPlane =
        new PolarCoordinatePlane( &m_chart );
        // replace the default Cartesian plane with
        // our Polar plane
        m_chart.replaceCoordinatePlane( polarPlane );

        // assign the model to our pie diagram
        PieDiagram* diagram = new PieDiagram;
        diagram->setModel(&m_model);
```

After having stored our data into the model, we create a need to replace the default Cartesian plane against a Polar plane before creating our Pie diagram. In this case, we want to display a `KDChartPieDiagram`. As always we need to assign the model to our diagram. This procedure is of course similar for all types of diagrams.

We are now ready to configure our attributes. We want to explode a section and configure a Pen to surround it. Let us begin with the specifical `PieAttributes`.

```
        // Configure some Pie specifical attributes

        // explode a section
        PieAttributes pa(  diagram->pieAttributes() );
        pa.setExplodeFactor( 0.1 );

        // Assign the attributes
        // to the diagram
        diagram->setPieAttributes( 1,  pa );
```

As for all attributes we call them by using the relevant method available from our diagram interface, here `diagram->PieAttributes()`. The second step is to set it up with our own values and finally we assign it to our diagram. In the above code we explode the second slice (dataset) in our Pie.

Note

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

We want to configure the Pen in order to draw a surrounding line around the exploded section (dataset) to focus the attention of the reader on this particular section.

```
        // Configure a generic attribute
        // available to all chart types
        QPen sectionPen;
        sectionPen.setWidth( 5 );
```

```
sectionPen.setStyle( Qt::DashLine );
sectionPen.setColor( Qt::magenta );

diagram->setPen( 1, sectionPen );
```

Of course we could also have changed the pen for all datasets as well.

Note

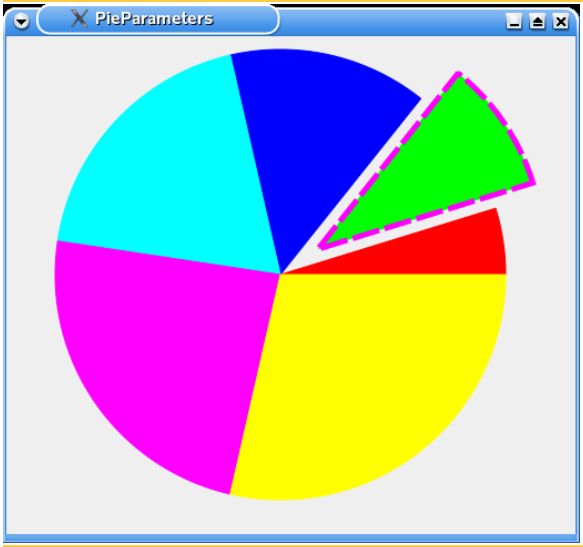
The Pen and the Brush setters and getters are implemented at a lower level in our `KDChartAbstractDiagram` class for a cleaner code structure. Those methods are of course used by all types of diagram and their configuration is very simple and straight forward as you can see in the above sample code.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

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

```
 // Assign our diagram to the Chart
m_chart.coordinatePlane()->replaceDiagram(diagram);

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

The above procedure can be applied to any of the supported attributes for all 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/Pie/Parameters` directory of your KD Chart installation.

Figure 4.20. Pie With Configured Attributes



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 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:

- Configure the Start position .
- Display a Pie chart and shift between normal and threeD look.
- Explode one or several slices and set a surrounding line around exploded sections
- Run an animation (exploding).

In the example below we are using a `KDChartChart` class and also an home made `TableModel` for the convenience. It is 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 Pie 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/Pie/Advanced` directory of your installation.

```
 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 {
35     class Chart;
       class PieDiagram;
   }

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

   public:
       MainWindow( QWidget* parent = 0 );
45
```

```
    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::PieDiagram* m_pie;
        QTimer* m_timer;

        int m_currentFactor;
        int m_currentDirection;
        int m_currentSlice;
    };


#endif /* MAINWINDOW_H */
```

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

Note

Before diplaying our Pie diagram we need to implicitely replace the default cartesian plane by a Polar plane.

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

```
**   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 <KDChartPieDiagram>
#include <KDChartPieAttributes>
#include <KDChartThreeDPieAttributes>

#include <QDebug>
#include <QTimer>

using namespace KDChart;

MainWindow::MainWindow( QWidget* parent ) :
    QWidget( parent ), m_currentFactor( 0 ), m_currentDirection( 1 ), m_currentSlice(
{
    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
    PolarCoordinatePlane* polarPlane = new PolarCoordinatePlane( m_chart );
    m_chart->replaceCoordinatePlane( polarPlane );
    m_pie = new PieDiagram();
    m_pie->setModel( &m_model );
    m_chart->coordinatePlane()->replaceDiagram( m_pie );

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

void MainWindow::on_startPositionSB_valueChanged( double pos )
{
    const int intValue = static_cast<int>( pos );
    startPositionSL->blockSignals( true );
    startPositionSL->setValue( intValue );
    startPositionSL->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 );
    m_chart->update();
}

void MainWindow::on_startPositionSL_valueChanged( int pos )
{
    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 );
    m_chart->update();
}
```

```
90
    void MainWindow::on_explodeSubmitPB_clicked()
    {
        setExplodeFactor( explodeDatasetSB->value(), explodeFactorSB->value() );
        m_chart->update();
95  }

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

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

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

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

    void MainWindow::on_threeDGB_toggled( bool toggle )
135 {
        // 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 );
140     attrs.setDepth( threeDFactorSB->value() );
        m_pie->setThreeDPieAttributes( attrs );
        m_chart->update();
    }

145 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() );
150     attrs.setEnabled( threeDGB->isChecked() );
        attrs.setDepth( factor );
        m_pie->setThreeDPieAttributes( attrs );
        m_chart->update();
    }
155
```

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 pie diagram. We also want to set up a `QTimer` to be able to run our animation. Finally we assign the diagram to our chart.

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

 m_model.loadFromCSV( ":/data" );

 // Set up the plane
 PolarCoordinatePlane* polarPlane = new PolarCoordinatePlane( m_chart
 m_chart->replaceCoordinatePlane( polarPlane );

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

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

The user should be able to change the start position from the GUI. This can be implemented by using `KDChartPieAttributes` as shown below and by updating the view.

```
....
PieAttributes pa( m_pie->pieAttributes() );
pa.setStartPosition( pos );
m_pie->setPieAttributes( pa );
m_chart->update();
....
```

We want the user to be able to shift between 3D mode display or the normal standard display from the GUI.

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

Note

It is important to know that have three levels of precedence when setting the attributes:

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

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.

We want the user to be able to explode one or several slice(s) (dataset) and to configure the exploding factor.

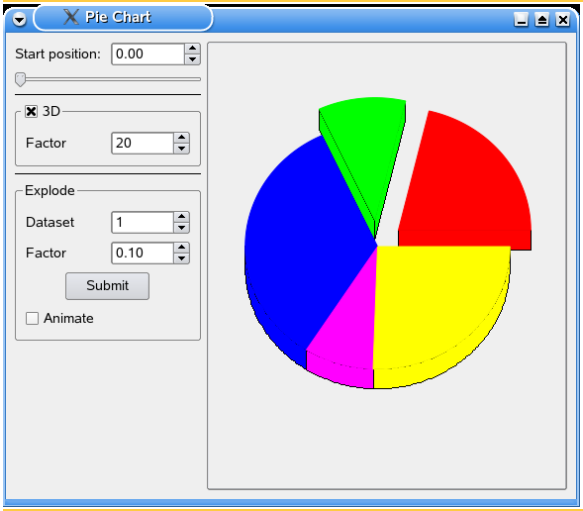
```
....
// note: We use the per-column getter method here, it will fall back
// automatically to return the global (or even the default) settings.
PieAttributes pa( m_pie->pieAttributes( column ) );
pa.setExplodeFactor( value );
m_pie->setPieAttributes( column, pa );
...
m_chart->update();
...
```

This is implemented by configuring our pie attributes and assign them by dataset to the diagram, as shown above.

The same procedure is used for us to be able to run our animation. You can of course learn more about this part of the code which is more related to Qt programming by consulting `examples/Pie/Advanced/mainwindow.cpp`.

This example is available to compile and run from the `examples/Pie/Advanced` directory into your KD Chart installation. We recommend you to run it. The widget displayed by the above code is shown in the figure below.

Figure 4.21. A Full featured Pie Chart



What's next

For our diagram to be useful we need to be able to display its axis. That will be the subject of our next section.

Chapter 5. Axes

Axes are implemented at different levels in the KD Chart 2.0 API. KD Chart make use of Cartesian axis - see `KDChartCartesianAxis` and Polar Axis which are derived from the base class for axes `KDChartAbstractAxis`. .

The user may specify its own set of strings to be used as Axis labels with the `KDChartAbstractAxis::setLabels( const QStringList )` method.

Note

Labels specified via setLabels take precedence: If a non-empty list is passed, KD Chart will use these strings as axis labels, instead of calculating them. By passing an empty QStringList you can reset the default behaviour.

For the convenience we can also specify short labels in our own set of string to be used as axis labels,in case the normal labels are too long by using `KDChartAbstractAxis::setShortLabels( const QStringList )`

Axis values and labels text attributes can also be configured. Thus the labels of all of your axes in all of your diagrams within that Chart will be drawn in same font size, by default.

The setters and getters for axis labels and their text attributes are implemented in the axis base class `KDChartAbstractAxis`, we recommend you to study its interface - See `KDChartAbstractAxis.h`.

Tip

If you a smaller number of strings than the number od labels drawn at this axis, KD Chart will iterate over the list, repeating the strings, until all labels are drawn.

As an example you could specify the seven days of the week as abscissa labels, which would be repeatedly used then.

Cartesian Axis

The class `KDChartCartesianAxis` is used together with the diagrams displayed in a cartesian coordinate plane and contains the setters and getters related to the axis specifics to those chart types.

It allows the user to set and retrieve the position (top, bottom, left or right), or the type (abscissa, ordinate) of the axis, assign or retrieve a title and its text attributes. That is

where the axis are painted.

The setters and getters for those specifics cartesian features are implemented in `KDChartCartesianAxis`, we recommend you to study its interface - See `KDChartCaretesianAxis.h`.

How to configure

In order to add axis to a cartesian diagram we need to use `KDChartAbstractCartesianDiagram::AddAxis()` method. The diagram takes ownership of the axis and will delete it by itself.

To gain back ownership (e.g. for assigning the axis to another diagram) use the `KDChartAbstractDiagram::takeAxis()` method, before calling addAxis on the other diagram.

Note

`KDChartAbstractDiagram::takeAxis()`Removes the axis from the diagram, without deleting it. The diagram no longer owns the axis, so it is the caller's responsibility to delete the axis.

Cartesian Axes sample

Let us make the above description more concrete by looking at the following lines of code based on the `Simple Widget` example we have been demonstrating above (`Chapter 3 - Two Ways - Widget Example`). In this example we demonstrate how to add an X axis and a Y axis to your diagram and set the Axis titles when working with a `KDChartWidget`..

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

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

using namespace KDChart;
```

We need to include `KDChartLineDiagram` in order to be able to add the Axis 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" );
```

Note

We don't need to change the default chart type (Line Charts) by calling the `KDChartWidget::setType` method.

Now let us create our axis, position them and set their titles:

```
CartesianAxis *xAxis = new CartesianAxis( widget.lineDiagram() );
CartesianAxis *yAxis = new CartesianAxis (widget.lineDiagram() );
xAxis->setPosition ( CartesianAxis::Bottom );
yAxis->setPosition ( CartesianAxis::Left );
xAxis->setTitleText ( "Abscissa bottom position" );
yAxis->setTitleText ( "Ordinate left position" );
```

And add them to our diagram which will take the ownership:

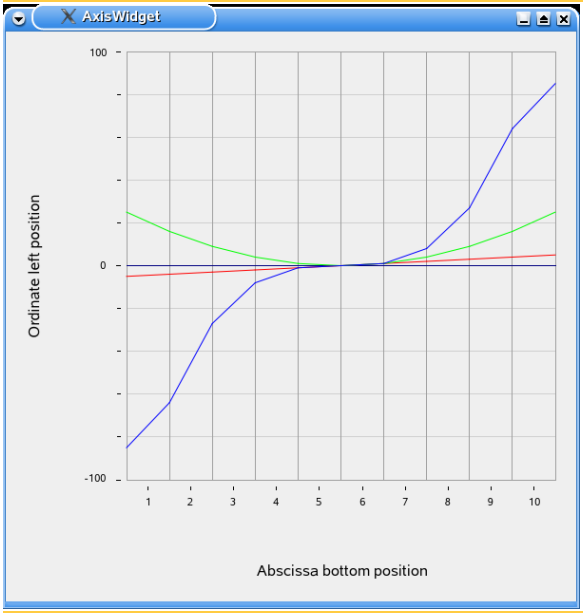
```
widget.lineDiagram()->addAxis( xAxis );
widget.lineDiagram()->addAxis( yAxis );
```

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 5.1. A Simple Widget With Axis



This example can be compiled and run from the following location of your KD Chart installation `examples/Axis/Widget`, we recommend you to do so.

In the Tips section below we will present you a more elaborate example which uses `KDChartChart` and where we are configuring our axis title text attributes. We also use our own labels and their shortened version.

Tips

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.

Axis Example

In the following implementation we want to be able to:

- Add axes at different positions.
- Set the axis title and configure their text attributes.
- Use our own labels and their shortened versions.
- Configure our labels text attributes.

In the example below we are using a `KDChartChart` class and also an home made `TableModel` for the convenience. It is 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 diagram _with_ axis 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/Axis/Chart` 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 LineDiagram;
   }

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

   public:
       MainWindow( QWidget* parent = 0 );

45
```

```
    private:
        KDChart::Chart* m_chart;
        TableModel m_model;
        KDChart::LineDiagram* m_lines;
};


#endif /* MAINWINDOW_H */

```

In the above code we bring up the KDChart namespace as usual. As you can see we are using a KDChartChart object (m_chart), a KDChartLineDiagram object (m_lines), and our home made TableModel (m_model).

```
                        /****************************************************************
    ** 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 <KDChartLineDiagram>
#include <KDChartTextAttributes>

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 );
```

```
        // create and position axis
        CartesianAxis *topAxis = new CartesianAxis( m_lines );
        CartesianAxis *leftAxis = new CartesianAxis ( m_lines );
 55     CartesianAxis *bottomAxis = new CartesianAxis ( m_lines );
        topAxis->setPosition ( CartesianAxis::Top );
        leftAxis->setPosition ( CartesianAxis::Left );
        bottomAxis->setPosition ( CartesianAxis::Bottom );

 60     // set axis titles
        topAxis->setTitleText ( "Abscissa color configured top position" );
        leftAxis->setTitleText ( "Ordinate font configured" );
        bottomAxis->setTitleText ( "Abscissa Bottom" );

 65     // configure titles text attributes
        TextAttributes taTop ( topAxis->titleTextAttributes () );
        taTop.setPen( QPen( Qt::red ) );
        topAxis->setTitleTextAttributes ( taTop );

 70     TextAttributes taLeft ( leftAxis->titleTextAttributes () );
        Measure me( taLeft.fontSize() );
        me.setValue( me.value() * 1.5 );
        taLeft.setFontSize( me );
        leftAxis->setTitleTextAttributes ( taLeft );
 75
        TextAttributes taBottom ( bottomAxis->titleTextAttributes () );
        taBottom.setPen(  QPen( Qt::blue ) );
        bottomAxis->setTitleTextAttributes ( taBottom );

 80     // configure labels text attributes
        TextAttributes taLabels;
        taLabels.setPen(  QPen( Qt::darkGreen ) );
        topAxis->setTextAttributes( taLabels );
        leftAxis->setTextAttributes( taLabels );
 85     bottomAxis->setTextAttributes( taLabels );

    // set the following to 0, to see the default Abscissa labels
    // (== X headers, as read from the data file)
    #if 1
 90     // configure labels and their shortened versions
        QStringList daysOfWeek;
        daysOfWeek << "Monday" << "Tuesday" << "Wednesday"
                   << "Thursday" << "Friday" ;
        topAxis->setLabels( daysOfWeek );
 95
        QStringList shortDays;
        shortDays << "Mon" << "Tue" << "Wed"
                  << "Thu" << "Fri";
        topAxis->setShortLabels( shortDays );
100
        QStringList bottomLabels;
        bottomLabels << "Day 1" << "Day 2" << "Day 3"
                     << "Day 4" << "Day 5";
        bottomAxis->setLabels( bottomLabels );
105
        QStringList shortBottomLabels;
        shortBottomLabels << "D1" << "D2" << "D3"
                     << "D4" << "D5";
        bottomAxis->setShortLabels( shortBottomLabels );
110 #endif

        // add axis
        m_lines->addAxis( topAxis );
        m_lines->addAxis( leftAxis );
115     m_lines->addAxis( bottomAxis );

        // assign diagram to chart view
        m_chart->coordinatePlane()->replaceDiagram( m_lines );
    }
120
```

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 diagram.

```
...
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 );
...
```

We want to display three axis, respectively positionned at the top, left and bottom side of our diagram. This is straight forward:

```
....
CartesianAxis *topAxis = new CartesianAxis( m_lines );
CartesianAxis *leftAxis = new CartesianAxis ( m_lines );
CartesianAxis *bottomAxis = new CartesianAxis ( m_lines );
topAxis->setPosition ( CartesianAxis::Top );
leftAxis->setPosition ( CartesianAxis::Left );
bottomAxis->setPosition ( CartesianAxis::Bottom );
....
```

In the code above we are declaring our axis and make use of `KDChartCartesianAxis::setPosition()` to give their location.

Let us now define the title text for each of those axis:

```
...
topAxis->setTitleText ( "Abscissa color configured top position" );
leftAxis->setTitleText ( "Ordinate font configured" );
bottomAxis->setTitleText ( "Abscissa Bottom" );
...
```

`setTitleText()` and `setTitleTextAttributes()` are implemented in `KDChartCartesianAxis`, we recommend you to consult its interface (see `KDChartCartesianAxis.h`

In this example and to demonstrate the text configuration for the title and the labels we want to have a different configuration for each of our tilte axis and also for our labels. The process is the same as for configuring any type of attributes, as follow:

Create an attribute object, configure it and assign it.

```
...
 // configure titles text attributes
```

```
TextAttributes taTop ( topAxis->titleTextAttributes () );
// color configuration
taTop.setPen( QPen( Qt::red ) );
// assign to the axis
topAxis->setTitleTextAttributes ( taTop );

TextAttributes taLeft ( leftAxis->titleTextAttributes () );
// Font configuration
Measure me( taLeft.fontSize() );
me.setValue( me.value() * 1.5 );
taLeft.setFontSize( me );
leftAxis->setTitleTextAttributes ( taLeft );

TextAttributes taBottom ( bottomAxis->titleTextAttributes () );
taBottom.setPen(  QPen( Qt::blue ) );
bottomAxis->setTitleTextAttributes ( taBottom );

// configure labels text attributes
TextAttributes taLabels;
taLabels.setPen(  QPen( Qt::darkGreen ) );
topAxis->setTextAttributes( taLabels );
leftAxis->setTextAttributes( taLabels );
bottomAxis->setTextAttributes( taLabels );
...
```

We want our top and bottom axis to display different types of labels as well as to make sure those labels will be shortened in case the normal labels are too long (see setShortLabels()).

```
// configure labels and their shortened versions
QStringList daysOfWeek;
daysOfWeek << "Monday" << "Tuesday" << "Wednesday"
<< "Thursday" << "Friday" ;
topAxis->setLabels( daysOfWeek );

QStringList shortDays;
shortDays << "Mon" << "Tue" << "Wed"
<< "Thu" << "Fri";
topAxis->setShortLabels( shortDays );

QStringList bottomLabels;
bottomLabels << "Day 1" << "Day 2" << "Day 3"
<< "Day 4" << "Day 5";
bottomAxis->setLabels( bottomLabels );

QStringList shortBottomLabels;
shortBottomLabels << "D1" << "D2" << "D3"
<< "D4" << "D5";
bottomAxis->setShortLabels( shortBottomLabels );
```

Note

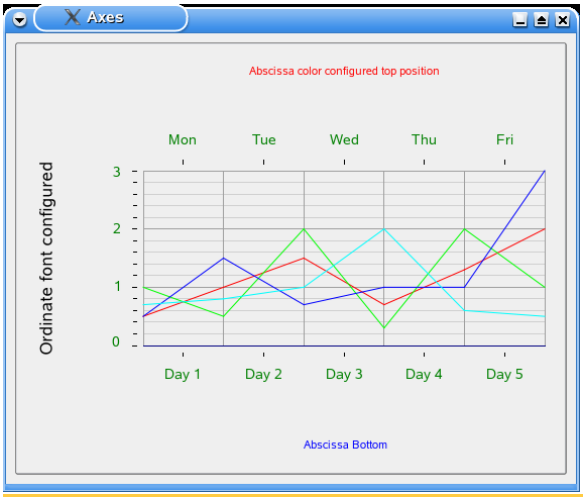
Labels specified via setLabels take precedence: if a non-empty list is passed, KD Chart will use these strings as axis labels, instead of calculating them.

Finally the last step i to assign our axis to the diagram and the diagram to our chart view.

```
 // add axis
m_lines->addAxis( topAxis );
m_lines->addAxis( leftAxis );
m_lines->addAxis( bottomAxis );

// assign diagram to chart view
m_chart->coordinatePlane()->replaceDiagram( m_lines );
```

This example is available to compile and run from the `examples/Axis/Chart` directory into your KD Chart installation. We recommend you to run it. The widget displayed by the above code is shown in the figure below.

Figure 5.2. Axis with configured Labels and Titles



Several ready to run examples related to axis are available at the following location `examples/Axis`, we recommend you to run them all and consult their implementation.

What's next

Legends are also an important element. In the next section we will describe how to add and configure your chart legend.

Chapter 6. Legends

Legends can be drawn for all kind of diagrams and are drawn on chart level (in relation to diagram level). We can have more than one legend per chart, using `KDChart-Chart::addLegend()`.

Note

Legend is different from all other classes ofd KD Chart, since it can be displayed outside of the Chart's area. If we want to, we can embedd the legend into your own widget, or into another part of a bigger grid, into which we might have inserted the Chart.

On the other hand, please note that we need to (MUST) call `Chart::addLegend()` to get our legend positioned at the correct position in our chart in case we want to display the legend inside of the chart which is probably true for most cases.

Let us go through the main configuration features offered by `KDChartLegend`. Of course we also recommend you to consult its interface see `KDChartLegend.h`.

How to configure

In order to add a legend to our chart we need to use the `KDChart-Chart::AddLegend()` method. The charttakes ownership of the legend and will take care of removing it by itself.

Tip

You may also wish to use `KDChartChart::replaceLegend( Legend newLegend , Legend oldLegend)` which is also available for the convenience:

The old legend will be deleted automatically. If its parameter is omitted, the very first legend will be replaced. In case, there was no legend yet, the new legend will just be added.

Please consult the interface for `KDChartChart` in order to get accustomed to those methods related to legends. See `KDChartChart.h`.

Note

`KDChartChart::takeLegend()`Removes the legend from the chart without deleting it. The chart no longer owns the legend, it is the caller's responsibility to delete the legend.

The main configurations elements for `KDChartLegend` are:

- ReferenceArea: Specifies or retrieve the reference area for font size of title text and for font size of the item texts.
- Diagrams: Add, retrieve, replace or remove diagrams associated to the legends.
- Show Lines: Paint lines between the different items of a legend.
- Title, Markers and Text attributes can be set, as well as colors and spacing.

Please consult the setters and getters methods available in the `KDChartLegend` interface. See `KDChartLegend.h`.

Legend Sample

We will now describe those features a more concrete way by looking at the following sample code based on the `Simple Widget` example we have been demonstrating above `Chapter 3 - Two Ways - Widget Example`. Through the following code we demonstrate how to add and position a Legend to your chart Widget using a KDChartWidget.

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

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

using namespace KDChart;
```

In this sample code we want to display a bar chart and need to include our `KDChartBarDiagram` class. In order to be able to give a location (position) to our legend in the widget view we also include `KDChartPosition`.

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

Widget widget;
widget.resize( 600, 600 );

QVector< double > vec0,  vec1,  vec2;

vec0 << -5 << -4 << -3 << -2 << -1 << 0
<< 1 << 2 << 3 << 4 << 5;
vec1 << 25 << 16 << 9 << 4 << 1 << 0
<< 1 << 4 << 9 << 16 << 25;
vec2 << -125 << -64 << -27 << -8 << -1 << 0
<< 1 << 8 << 27 << 64 << 125;
```

```
widget.setDataset( 0, vec0, "v0" );
widget.setDataset( 1, vec1, "v1" );
widget.setDataset( 2, vec2, "v2" );
widget.setType( Widget::Bar );
```

Note

We need to change the default chart type (Line Charts) by calling the `KDChartWidget::setType` method in order to display a bar type diagram.

Now let us add our legend, set its position and orientation, its title and dataset labels text:

```
widget.addLegend(Position::North);
widget.firstLegend()->setOrientation( Qt::Horizontal );
widget.firstLegend()->setTitleText( "Bars Legend" );
widget.firstLegend()->setText( 0,  "Vector 1" );
widget.firstLegend()->setText( 1,  "Vector 2" );
widget.firstLegend()->setText( 2,  "Vector 3" );
widget.firstLegend()->setShowLines(  true );
```

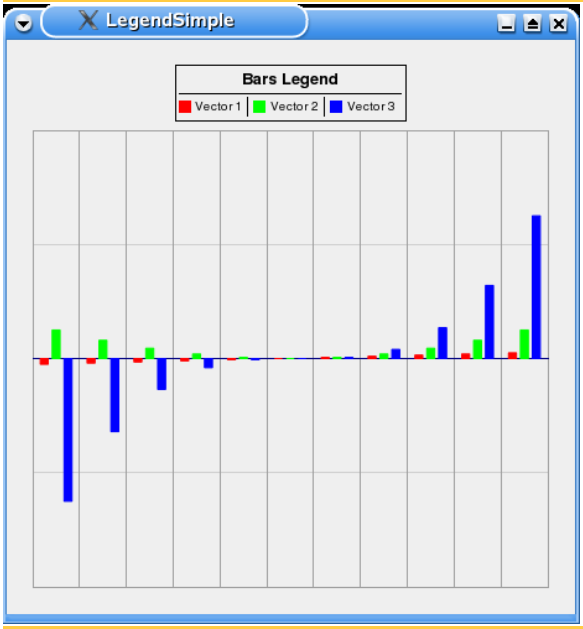
The interesting point here is how we call `KDChartWidget::firstlegend()` to get a pointer to to our legend object and be able to set up and configure it. We will see further on in the next code example (see - Section Tips - how to configure the elements of a legend (e.g Title text, markers, etc....).

Finally we conclude our small application by runnig the usual lines of code.

```
 widget.show();

return app.exec();
```

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

Figure 6.1. A Widget With a simply configured Legend



This example can be compiled and run from the following location of your KD Chart installation `examples/Legends/LegendSimple`, we recommend you to do so.

In the Tips section below we will present you a more elaborate example which uses `KDChartChart` and where we are setting up our legend elements (title, texts, markers, etc...).

Tips

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.

Before we go through this example, let us study a very simple chart implementation with its legend by looking at the following line of codes which we will comment.

First and as we always do, we set up a model, declare our diagram, and assign the model to it and the diagram to our chart after having included the relevant header files.

```
#include <QtGui>
#include <KDChartChart>
#include <KDChartBarDiagram>
#include <KDChartLegend>
#include <KDChartPosition>
#include <KDChartBackgroundAttributes>
#include <KDChartFrameAttributes>

using namespace KDChart;
```

```
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 BarDiagram;
    diagram->setModel(&m_model);

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

We will set the legend position as well as its Background and Frame attributes and includes those header files on this purpose. That will allow us to make use of the methods available in those classes.

We will now add a legend and set it up (positions, orientations, etc...):

```
// Add a legend and set it up
Legend* legend = new Legend( diagram, &m_chart );
legend->setPosition( Position::NorthEast );
legend->setAlignment( Qt::AlignCenter );
legend->setShowLines( false );
legend->setTitleText( tr( "Bars" ) );
legend->setOrientation( Qt::Vertical );
m_chart.addLegend( legend );
```

The code above handle the attributes specific to a legend, the setters and getters for the methods we have been used here are implemented into the `KDChartLegend` class. We recommend you to consult its interface. See `KDChartLegend.h`.

Set the Legend marker attributes. We want each dataset's marker to have its own marker style.

```
// Configure the items markers
MarkerAttributes lma;
lma.setMarkerStyle( MarkerAttributes::MarkerDiamond );
legend->setMarkerAttributes( 0,  lma );
lma.setMarkerStyle( MarkerAttributes::MarkerCircle );
legend->setMarkerAttributes( 1,  lma );
```

Markers are assigned per dataset as you can see above. You can learn more about the marker styles and the methods available to configure markers into the `MarkerAttributes` class interface. See `KDChartMarkerAttributes.h`.

Let us now configure our legend's items text:

```
// Configure labels for Legend's items
legend->setText( 0,  "Series 1" );
legend->setText( 1,  "Series 2" );
legend->setText( 2,  "Series 3" );
```

Each dataset can be assigned its own text. We want to change their pen color for demonstrating this feature and also to make our legend nicer. We proceed as follow and configure their text attributes.

```
TextAttributes lta;
lta.setPen( QPen( Qt::darkGray ) );
legend->setTextAttributes(  lta );
```

Text attributes configuration and assignment is done as for all other types of attribute. Create a text attribute object, configure it and assign it. In this case we assign it to our legend by using its method `KDChartLegend::setTextAttributes()`.

Tip

If we wish to paint a surrounding line round our legend markers we just need to configure a pen and assign it to our legend by calling `KDChartLegend::setPen()`. See the following code sample that demonstrate that.

```
// Configure a pen to surround
// the markers with a border
QPen markerPen;
markerPen.setColor(  Qt::darkGray );
markerPen.setWidth( 2 );
// Pending Michel use datasetCount() here as soon
// as it is fixed
for (  uint i = 0; i < legend->datasetCount(); i++ )
    legend->setPen( i,  markerPen );
```

Note

Mind the call to `KDChartLegend::datasetCount()` which allow you to retrieve the number of dataset and simply loop through it.

We want to make our legend more readable by setting a white background inside its frame.

```
 // Add a background to your legend
BackgroundAttributes ba;
ba.setBrush(  Qt::white );
ba.setVisible( true );
```

```
legend->setBackgroundAttributes(  ba );
```

As for all attributes settings the code is straight forward, just create the attribute object, configure it and assign it. We recommend you to have a look at the `KDChartBackgroundAttributes` interface. See `KDChartBackgroundAttributes.h`

Let us now configure our legend's frame:

```
 FrameAttributes fa;
fa.setPen( markerPen );
fa.setPadding( 5 );
fa.setVisible( true );
legend->setFrameAttributes(  fa );
```

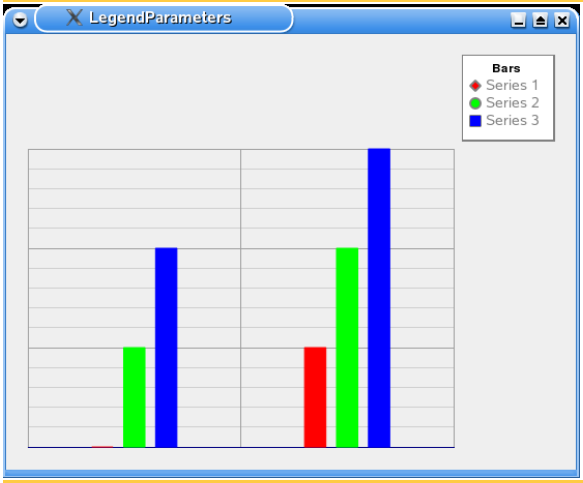
Same procedure as above. Please note the `setVisible()` method which is necessary as the default value hide those attributes.

Finally we need to conclude our small application.

```
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"
```

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

Figure 6.2. A Chart with configured Legend



We recommend you to compile and run the above example, it is available at the following location: `examples/Legends/LegendParameters` .

Legend Example

In the following implementation we want to be able to:

- Add, edit or remove Legends in/from our chart view.
- Configure its position, and a few attributes.
- Set its Title
- All of the above operations should be available to the user from the GUI and performed dinamically.

In the example below we are using a `KDChartChart` class and also an home made `TableModel` for the convenience. It is 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 diagram _with_ axis 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/Legends/LegendAdvanced` 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 <QDialog>
   #include <QMap>

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

   namespace KDChart {
       class Chart;
       class LineDiagram;
40 }

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

   private slots:
50     void on_addLegendPB_clicked();
       void on_editLegendPB_clicked();
       void on_removeLegendPB_clicked();
       void on_legendsTV_itemSelectionChanged();

55 private:
       void initAddLegendDialog( DerivedAddLegendDialog& conf,
                                 Qt::Alignment alignment ) const;

       KDChart::Chart* m_chart;
60     TableModel m_model;
       KDChart::LineDiagram* m_lines;
       QMap<Qt::Alignment, QString> alignmentMap;
   };
```

```
65
   #endif /* MAINWINDOW_H */
```

In the above code we bring up the KDChart namespace as usual. As you can see we are using a KDChartChart object (m_chart), a KDChartLineDiagram object (m_lines), and our home made TableModel (m_model).

```
 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 <KDChartLegend>
   #include <KDChartPosition>
   #include <KDChartLineDiagram>
   #include <KDChartTextAttributes>
   #include <QComboBox>
35 #include <QLineEdit>

   class LegendItem : public QTreeWidgetItem
   {
   public:
40     LegendItem( KDChart::Legend* legend, QTreeWidget* parent ) :
           QTreeWidgetItem( parent ), m_legend( legend ) {}

       KDChart::Legend* legend() const { return m_legend; }

45 private:
       KDChart::Legend* m_legend;
   };


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

55     QHBoxLayout* chartLayout = new QHBoxLayout( chartFrame );
       m_chart = new KDChart::Chart();
```

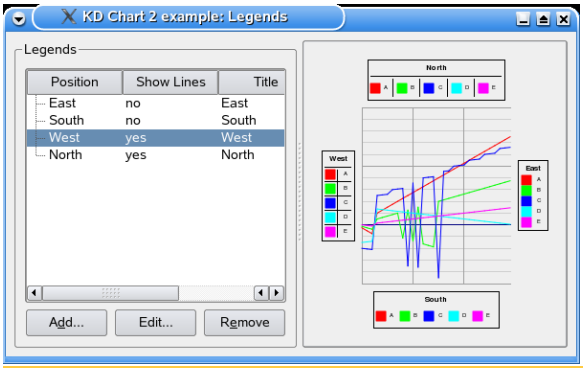
```
        chartLayout->addWidget( m_chart );

        m_model.loadFromCSV( ":/data" );

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

        // Add at least one legend for starters
        KDChart::Legend* legend = new KDChart::Legend( m_lines, m_chart );
        legend->setPosition( KDChart::Position::NorthEast );
        legend->setAlignment( Qt::AlignCenter );
        legend->setShowLines( false );
        legend->setTitleText( tr( "Legend" ) );
        legend->setOrientation( Qt::Vertical );
        m_chart->addLegend( legend );
        legend->show();

        LegendItem* newItem = new LegendItem( legend, legendsTV );
        newItem->setText( 0, tr( "NorthEast" ) );
        newItem->setText( 1, tr( "no" ) );
        newItem->setText( 2, tr( "Legend" ) );
        newItem->setText( 3, tr( "Vertical" ) );
        newItem->setText( 4, tr("Center") );

        alignmentMap[ Qt::AlignTop     | Qt::AlignLeft]    = tr("Top + Left");
        alignmentMap[ Qt::AlignTop     | Qt::AlignHCenter] = tr("Top + HCenter");
        alignmentMap[ Qt::AlignTop     | Qt::AlignRight]   = tr("Top + Right");
        alignmentMap[ Qt::AlignVCenter | Qt::AlignRight]   = tr("VCenter + Right");
        alignmentMap[ Qt::AlignBottom  | Qt::AlignRight]   = tr("Bottom + Right");
        alignmentMap[ Qt::AlignBottom  | Qt::AlignHCenter] = tr("Bottom + HCenter");
        alignmentMap[ Qt::AlignBottom  | Qt::AlignLeft]    = tr("Bottom + Left");
        alignmentMap[ Qt::AlignVCenter | Qt::AlignLeft]    = tr("VCenter + Left");
        alignmentMap[ Qt::AlignCenter]                     = tr("Center");

        m_chart->update();
    }

    void MainWindow::initAddLegendDialog( DerivedAddLegendDialog& conf,
                                          Qt::Alignment alignment ) const
    {
        conf.titleTextED->setFocus();
        conf.positionCO->addItems( KDChart::Position::printableNames(false) );
        QMap<Qt::Alignment, QString>::const_iterator i = alignmentMap.constBegin();
        while (i != alignmentMap.constEnd()) {
            conf.alignmentCO->addItem( i.value() );
            ++i;
        }
        const int idx = conf.alignmentCO->findText( alignmentMap[ alignment ] );
        if( idx > -1 )
            conf.alignmentCO->setCurrentIndex( idx );
    }

    void MainWindow::on_addLegendPB_clicked()
    {
        DerivedAddLegendDialog conf;
        initAddLegendDialog( conf, Qt::AlignCenter );
        if( conf.exec() == QDialog::Accepted ) {
            KDChart::Legend* legend = new KDChart::Legend( m_lines, m_chart );
            m_chart->addLegend( legend );
            legend->setPosition(
                KDChart::Position::fromPrintableName( conf.positionCO->currentText() ) );
            // get the alignment
            Qt::Alignment alignment = Qt::AlignCenter;
            const QString selectedAlignment( conf.alignmentCO->currentText() );
            QMap<Qt::Alignment, QString>::const_iterator i = alignmentMap.constBegin();
            while (i != alignmentMap.constEnd()) {
                if (  i.value() == selectedAlignment ){
```

```
                    alignment = i.key();
                    break;
                }
                ++i;
            }
            legend->setAlignment( alignment );
            legend->setShowLines( conf.showLinesCB->isChecked() );
            legend->setTitleText( conf.titleTextED->text() );
            legend->setOrientation( ( conf.orientationCO->currentIndex() == 0 ) ? Qt::Ver
            legend->show();
            LegendItem* newItem = new LegendItem( legend, legendsTV );
            newItem->setText( 0, conf.positionCO->currentText() );
            newItem->setText( 1, conf.showLinesCB->isChecked() ? tr("yes") : tr("no") );
            newItem->setText( 2, conf.titleTextED->text() );
            newItem->setText( 3, conf.orientationCO->currentText() );
            newItem->setText( 4, selectedAlignment );
            m_chart->update();
        }
}


    void MainWindow::on_editLegendPB_clicked()
    {
        if ( legendsTV->selectedItems().size() == 0 ) return;
        LegendItem* item = static_cast<LegendItem*>( legendsTV->selectedItems().first() )
        KDChart::Legend* legend = item->legend();
        DerivedAddLegendDialog conf;
        initAddLegendDialog( conf, legend->alignment() );
        conf.showLinesCB->setChecked( legend->showLines() );
        conf.titleTextED->setText( legend->titleText() );

        // In this example we are using legend position names, that match
        // exactly the names in KDChart::Legend::LegendPosition,
        // so we can use a shortcut here, to set the respective name in
        // the dialog's list, and we need no error checking for findText():
        conf.positionCO->setCurrentIndex(
                conf.positionCO->findText( legend->position().printableName() ) );
        conf.orientationCO->setCurrentIndex( (legend->orientation()==Qt::Vertical)?0:1 );

        if( conf.exec() == QDialog::Accepted ) {
            //legend->setPosition( (KDChart::Legend::LegendPosition)conf.positionCO->curr
            legend->setPosition(
                KDChart::Position::fromPrintableName( conf.positionCO->currentText() ) );
            // get the alignment
            Qt::Alignment alignment = Qt::AlignCenter;
            const QString selectedAlignment( conf.alignmentCO->currentText() );
            QMap<Qt::Alignment, QString>::const_iterator i = alignmentMap.constBegin();
            while (i != alignmentMap.constEnd()) {
                if (  i.value() == selectedAlignment ){
                    alignment = i.key();
                    break;
                }
                ++i;
            }
            legend->setAlignment( alignment );
            legend->setShowLines( conf.showLinesCB->isChecked() );
            legend->setTitleText( conf.titleTextED->text() );
            legend->setOrientation( ( conf.orientationCO->currentIndex() == 0 ) ? Qt::Ver
            item->setText( 0, conf.positionCO->currentText() );
            item->setText( 1, conf.showLinesCB->isChecked() ? tr("yes") : tr("no") );
            item->setText( 2, conf.titleTextED->text() );
            item->setText( 3, conf.orientationCO->currentText() );
            item->setText( 4, selectedAlignment );
            m_chart->update();
        }
    }


    void MainWindow::on_removeLegendPB_clicked()
```

```
    {
        if ( legendsTV->selectedItems().size() == 0 ) return;
        QList<QTreeWidgetItem*> items = legendsTV->selectedItems();
200     for( QList<QTreeWidgetItem*>::const_iterator it = items.begin();
             it != items.end(); ++it ) {
            delete static_cast<LegendItem*>( (*it) )->legend();
            delete (*it);
        }
205     m_chart->update();
    }

    void MainWindow::on_legendsTV_itemSelectionChanged()
    {
210     removeLegendPB->setEnabled( legendsTV->selectedItems().count() > 0 );
        removeLegendPB->setEnabled( legendsTV->selectedItems().count() == 1 );
    }
```

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

Figure 6.3. Legend advanced example



This ready to run example is available at the following location `examples/Legends/LegendAdvanced` of your KD Chart installation, we recommend you to study its code, compile and run it.

What's next

You can also add headers and/or footers to your chart to make it more understandable. In the next section w e will go through the several features and configuration possibilities available in KD Chart 2.0 about "Headers and Footers".

Chapter 7. Header and Footers

Headers and Footers can be added and configured in several ways. That will be the subject of this section where we will go through the main features and methods available. Of course we recommend you to consult the `KDChartHeaderFooter` class interface to learn more about those features and methods. See `KDChartHeaderFooter.h`

How to configure

text

Headers and Footers code Sample

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

Customizing your chart.

Chapter 8. Customizing your Chart

Introduction text

KD Chart Attributes

Text ...

Attributes Example

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

More explanation?

Data Values Attributes

Text ...

DataValue Attributes Example

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

More explanation?

Text Attributes

Text ...

Text Attributes Example

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

More explanation?

Markers Attributes

Text ...

Marker Attributes Example

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

More explanation?

BackGround Attributes

Text ...

Background Attributes Example

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

More explanation?

Grid Attributes

Text ...

Grid Attributes Example

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

More explanation?

Frame Attributes

Text ...

Frame Attributes Example

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

More explanation?

ThreeD Attributes

Text ...

ThreeD Attributes Example

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

More explanation?

Tips

Text ...

Customizing 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

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.