KD Chart Programmer's Manual

The KD Chart Team

KD Chart Programmer's Manual

The KD Chart Team

Version 2.2

Copyright © 2001—2008 Klarälvdalens Datakonsult AB

The contents of this manual and the associated KD Chart software are the property of Klarälvdalens Datakonsult AB and are copyrighted. KD Chart is available under two different licenses, depending on the intended use of this product:

- Commercial users (i.e. people intending to develop a commercial product using KD Chart) need to order a commercial license from Klarälvdalens Datakonsult AB.
- KD Chart is also available for creating non-commercial, open-source software under the GNU General Public License, version 2.0 or 3.0 as published by the Free Software Foundation. Alternatively you may (at your option) use any later version of the GNU General Public License if such license has been publicly approved by Klarälvdalens Datakonsult AB (or its successors, if any).

It is your responsibility to decide which license type is appropriate for your intended use of KD Chart. Any reproduction of this manual and the associated KD Chart software in whole or in part that is not allowed by the applicable license is strictly prohibited without prior written permission by Klarälvdalens Datakonsult AB.

KD Chart and the KD Chart logo are trademarks or registered trademarks of Klarälvdalens Datakonsult AB in the European Union, the United States, and/or other countries. Other product and company names and logos may be trademarks or registered trademarks of their respective companies.

Table of Contents

1. Chapter 1 1
    - What You Should Know 1
    - The Structure of This Manual 1
    - What's next 2
2. Chapter 2 3
    - Overview 3
        - Code Sample 3
        - Concepts 3
        - Ownership of Components versus Parameters 5
    - KD Chart and Model/View 5
        - Code Sample 5
    - Attribute sets 6
        - Code Sample 6
    - Memory Management 7
        - Code Sample 7
    - What's Next 8
3. Chapter 3 9
    - Prerequisites 9
    - The Procedure 9
    - Two Ways To Your Chart 11
        - Widget Example 11
        - Chart Example 12
    - What's Next 12
4. Chapter 4 13
    - Cartesian Coordinate Planes 13
        - Bar Charts 14
        - Normal Bar Charts 14
        - Stacked Bar Charts 14
        - Percent Bar Charts 15
        - Code Sample 15
        - Bars Attributes 17
        - Bar Attributes Sample 18
        - Tips and Tricks 21
        - A complete Bar Example 21
        - Line Charts 25
        - Normal Line Charts 25
        - Stacked Line Charts 26
        - Percent Line Charts 26
        - Code Sample 27
        - Lines Attributes 29
        - Line Attributes Sample 30
        - Tips and Tricks 33
        - A complete Line Example 33
        - Point Charts 36
        - Point Sample Code 36
        - Points Attributes 37
        - Tips and Tricks 37
        - A complete Point Example 37
        - Area Charts 39
        - Area Sample Code 40
        - Area Attributes 40
        - Tips and Tricks 40
        - A complete Area Example 40
        - Plotter Charts 43
        - Plotter Sample Code 44

- Levey-Jennings Charts ... 44
- Polar coordinate plane ... 45
  - Pie Charts ... 45
  - Simple Pie Charts ... 45
  - Exploding Pie Charts ... 46
  - Code Sample ... 47
  - Pies Attributes ... 48
  - Pie Attributes Sample ... 49
  - Tips and Tricks ... 52
  - A complete Pie Example ... 52
  - Polar Charts ... 54
  - A Simple Polar Chart ... 54
  - Polar Attributes ... 55
- Ternary coordinate plane ... 56
  - Ternary Line Charts ... 56
- What's next ... 57

5. Chapter 5 ... 58
  - Cartesian Axis ... 58
  - Ternary Axis ... 58
  - How to configure Cartesian Axes ... 58
    - Cartesian Axes sample ... 59
  - Tips ... 60
    - Axis Example ... 60
6. Chapter 6 ... 65
  - How to configure ... 65
    - Legend Sample ... 66
  - Tips ... 67
  - What's next ... 71
7. Chapter 7 ... 72
  - How to configure ... 72
    - Headers and Footers code Sample ... 72
  - Tips ... 74
    - Headers and Footers Example ... 77
  - What's next ... 78
8. Chapter 8 ... 79
  - Attributes Model, Abstract Diagram ... 79
    - How it works ... 79
  - Data Tooltips and Comments ... 81
    - Specifying a data item tooltip ... 81
    - Specifying a fixed data item comment ... 81
  - Data Values Attributes ... 82
    - DataValue Attributes Sample code ... 83
    - Data Values Labels: Details ... 84
  - Text Attributes ... 84
    - Text Attributes Sample code ... 85
  - Markers Attributes ... 86
    - Markers Attributes Sample code ... 87
  - Value Tracker Attributes ... 88
  - Background Attributes ... 89
    - Background Attributes Sample code ... 89
  - Frame Attributes ... 90
    - Frame Attributes Sample code ... 91
  - Grid Attributes ... 91
    - Grid Attributes Sample code ... 92
  - ThreeD Attributes ... 93
    - ThreeD Attributes Example ... 94
  - Font Sizes and other Measures ... 95
    - When and how to use the Measure class ... 95

        How to specify absolute values 96
        How to specify relative values 96
    Relative and Absolute Positions 97
        What is relative positioning all about? 97
        How to specify a position 97
        Using Position and RelativePosition 97
    What's next 98
9. Chapter 9 99
    Example programs to consult 99
A. Appendix A 104

List of Figures

2.1. Scope selection for Data Value Texts 7
3.1. A Simple Widget 11
3.2. A Simple Chart 12
4.1. A Normal Bar Chart 14
4.2. A Stacked Bar Chart 15
4.3. A Percent Bar Chart 15
4.4. A Simple Bar Chart Widget 17
4.5. Bar with Configured Attributes 21
4.6. A Full featured Bar Chart 25
4.7. A Normal Line Chart 26
4.8. A Stacked Line Chart 26
4.9. A Percent Line Chart 27
4.10. A Simple Line Chart Widget 29
4.11. Line With Configured Attributes 32
4.12. A Full featured Line Chart 35
4.13. A Point Chart 36
4.14. A Full featured Point Chart 39
4.15. An Area Chart 39
4.16. A Full featured Area Chart 43
4.17. A simple Plotter diagram 44
4.18. A simple Levey-Jennings diagram 45
4.19. A Simple Pie Chart 46
4.20. An Exploding Pie Chart 46
4.21. A Simple Pie Widget 48
4.22. Pie With Configured Attributes 52
4.23. A Full featured Pie Chart 54
4.24. A Normal Polar Chart 55
4.25. A Simple Ternary Chart 56
5.1. A Simple Widget With Axis 60
5.2. Axis with configured Labels and Titles 63
6.1. A Widget with a simple Legend 67
6.2. Legend advanced example 70
7.1. A Widget with a header and a footer 74
7.2. A Chart with a configured Header 77
7.3. Headers and Footers advanced example 78
8.1. Scope selection for Data Value Texts 81
8.2. A Chart with configured Data Value Texts 84
8.3. Positioning / adjusting Data Labels 84
8.4. A Chart with a configured Header 86
8.5. A Chart with configured Data Markers 88
8.6. A Line Chart showing Value Trackers 88
8.7. A simple Bar Chart with a Background Image 90
8.8. A Chart with configured Frame Attributes 91
8.9. A Chart with configured Grid Attributes 93
8.10. A Three-D Bar Chart 95
8.11. Data value text positions relative to compass points 97
9.1. /examples/Axis/Parameters 99
9.2. /examples/Axis/Labels 99
9.3. /examples/Bars/Advanced 100
9.4. /examples/HeadersFooters/HeadersFooters/Advanced 100
9.5. /examples/Legends/LegendAdvanced 100
9.6. /examples/Lines/Advanced 101
9.7. /examples/Plotter/BubbleChart 101
9.8. /examples/ModelView/TableView 101
9.9. /examples/Pie/Advanced 102

9.10. /examples/SharedAbscissa 102
9.11. /examples/Widget/Advanced 102
9.12. /examples/Zoom/Keyboard 103
9.13. /examples/Zoom/ScrollBars 103

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 it provides lots of pointers to the many advanced features in KD Chart.

- 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 the source code.
- KD Chart also comes with an extensive Reference Manual, generated directly from the source code itself, available both as a PDF file and as browsable HTML pages. When refering to the information in this manual we will simply use the term "API Reference".

You should refer to it in conjunction with this Programmer's Manual, if your question is not covered here in the respective chapter here (or in Appendix A, *Q&A section* at the end of this manual.)

- What is KD Chart?

  KD Chart is a tool for creating business and scientific charts, and is the most powerful Qt component of its kind. Besides having all of 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 you use KD Chart for?

  KD Chart is used by a variety of programs for a variety of different purposes including 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 also no coincidence that the current version of the KOffice productivity suite uses our library.

What You Should Know

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

The Structure of This Manual

How we will proceed with presenting KD Chart?

This manual starts with an introduction to the KD Chart 2 API before going through the basic steps and methods for the user to create her own chart.

The following Chapter 4, *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 get the most out of KD Chart.

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, footers and legends as well as how to configure your chart axes is also a part of these chapters.

Chapter 9 `Advanced Charting`, will present you with KD Chart's other more advanced features and show screenshots of example programs demonstrating how set up frames and backgrounds, data value texts, axis and grids etc..Additionally it is covering features like Interactive and Multiple charts or Zooming.

We provide you with many more example programs than shown in this manual and we recommend our readers to try and run them, have a look at the code and experiment with the various settings, both by adjusting them via the user interface, and by trying out your own code modifications.

What's next

In the next chapter we introduce the KD Chart 2 API.

Chapter 2. KD Chart 2 API Introduction

Since version 2.2 KD Chart fully supports and builds on the technologies introduced with Qt 4. The charting engine makes use of the Arthur (painting) and Scribe (text rendering) frameworks to achieve high quality visual results. KD Chart 2 also integrates with the Interview framework for model/view separation and, much like Qt 4 itself, it provides a convenience Widget class for simple use cases.

Overview

The core of KD Chart 2 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 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 Bar Diagram looks like this:

Code Sample

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

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 trying out.

Concepts

For now, in order to get an overview about the KD Chart 2 API and its features, you need to understand the following base concepts:

- 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 delegate the complete coordinate calculation work to the coordinate plane.

- Each coordinate plane can have one or more diagrams associated to it. Those diagrams will share the scale provided by the plane. A chart can also 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 make it owned by the first diagram and we add it to the second diagram. The Chart layouting engine will take care of adjusting positions accordingly.

  This code is taken from `mainwindow.cpp` in `examples/SharedAbscissa/SeparateDiagrams/`, here we are using two data models, two coordinate planes, two diagrams, two ordinate axes - but just one abscissa axis:

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

m_lines2 = new LineDiagram();
m_lines2->setModel( &m_model2 );

// We call this "plane2" just for remembering, that we use it
// in addition to the plane, that's built-in by default.
plane2 = new CartesianCoordinatePlane( m_chart );

CartesianAxis *xAxis = new CartesianAxis( m_lines );
CartesianAxis *yAxis = new CartesianAxis ( m_lines );
CartesianAxis *yAxis2 = new CartesianAxis ( m_lines2 );

xAxis->setPosition ( KDChart::CartesianAxis::Top );
yAxis->setPosition ( KDChart::CartesianAxis::Left );
yAxis2->setPosition ( KDChart::CartesianAxis::Right );

m_lines->addAxis( yAxis );
m_lines2->addAxis( xAxis );
m_lines2->addAxis( yAxis2 );

m_chart->coordinatePlane()->replaceDiagram( m_lines );

plane2->replaceDiagram( m_lines2 );

m_chart->addCoordinatePlane( plane2 );
```

Note how the X axis is owned by the first diagram, but we explicitely add it to the second diagram, so it is shared between both of them.

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.

For example, to add an additional header you can use code like this:

```
HeaderFooter * additionalHeader = new HeaderFooter;
additionalHeader->setPosition( NorthWest );
// add the text and/or customize the header
// ...
chart->addHeaderFooter( additionalHeader  );
```

In the next section, we will further explain how ownership of such components is maintained.

Finally, and concluding this overview, all classes in the KD Chart 2 API are in the `KDChart` namespace, to allow concise 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`.

Ownership of Components versus Parameters

Setting up a chart consists of doing two different things: Adding components. (Diagrams, Coordinate Planes, Axes, Headers, Legends, ...) and specifying attributes (Text Attributes, Data Value Attributes, Frame Attributes, ...).

For the components please note they are typically owned by their respective container widgets. Memory management of the component classes is explained in the section called “Memory Management” further down in this chapter.

Handling of attributes is different - their values are normally copied, no pointers are passed, and the objects are owned by the one who instantiates them, please study the section called “Attribute sets” for details, this is also to be found a bit below this section in the same chapter.

KD Chart and Model/View

KD Chart 2 follows the "Interview" model/view paradigm introduced by Qt 4:

Any `KDChart::AbstractDiagram` subclass (which in turn inherits `QAbstractItemView`) can display data originating from any `QAbstractItemModel` object. In order to use your data with KD Chart diagrams, you need to either use one of Qt's built-in 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 get started quickly without having to master the complexities of the model/view framework in Qt 4. We would still advise to use `KDChart::Chart` so that you can make use of all the benefits that model/view programming brings.

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

Code Sample

Now let's look at the following lines of code where we are using `QStandardItemModel` to store the data which will be displayed by the diagram in a `KDChart::Chart` 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) );
    }
}
```

Assign the model to your diagram and display it:

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

Using `KDChart::Widget` we would use code as follow:

```
KDChart::Widget 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 that you read the API Reference of `KDChart::Chart` and `KDChart::Widget` 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 can 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 control whether there should be a background pixmap, or a solid background color. Other attribute sets include frame attributes or grid attributes. 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( chart->legend()->textAttributes() );
ta.setPen( Qt::red );
ta.setFont( QFont( "Helvetica" ) );
chart->legend()->setTextAttributes( ta );
```

Note

When ever you want to modify an attribute set make sure to use the copy constructor for instantiating your attributes object! By doing so you can be sure to not alter all of the existing configuration when modifying only your desired details of the respective attributes set.

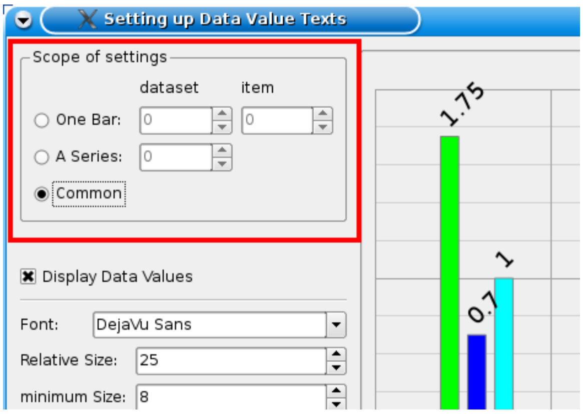
As an example, the code block shown above is just changing the font and its color, but it leaves all size settings as they have been before.

All attribute sets can be set per cell, per column or per modelindex, 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 found at the dataset level, either the model wide setting is used or 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 duplication of the fallback logic in numerous places in the library, thus avoiding unnecessary and 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 the API Reference for `KDChart::`*`[type]`*`Attributes` classes)

As an example see the upper/left part of the screenshot below demonstrating a way the scope of some attribute settings might be selected:

Figure 2.1. Scope selection for Data Value Texts



Scope of settings

dataset item

○ One Bar: 0 0

○ A Series: 0

● Common

☒ Display Data Values

Font: DejaVu Sans

Relative Size: 25

minimum Size: 8

To see how this is done please have a look at the `examples/DataValueTexts/` example program.

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 transfers ownership to the chart. Notice 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 the API Reference for `KDChart::Chart` and/or for `KDChart::Legend`)

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 described in the previous chapter, there are two ways to create a chart:

- `KDChart::Widget` provides a limited set of functions as shown in the API Reference of `KDChart::Widget`. Its purpose is to provide a convenient and simple way of displaying a chart for people who do not care about more complicated details like the Coordinate Plane and other classes provided by the KD Chart 2 API.
- The purpose of `KDChart::Chart` is to give the user access to the full power of both Qt and KD Chart.

Basically, `KDChart::Widget` has been designed for beginners, while `KDChart::Chart` is designed for experienced users and/or users who need more features and flexibility. Once again, we recommend you to check out both interfaces of those classes in order to give yourself an idea about which one of the classes best matches your needs. See the API Reference of `KDChart::Chart` and `KDChart::Widget`.

Prerequisites

As described above in the section called “KD Chart and Model/View ”, a prerequisite for using the full KD Chart API is that the data to be charted be made available through a class implementing the `QAbstractItemModel` interface. Before looking at some code, let us show you a few top-level classes of the KD Chart 2 API:

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

  A "chart" can hold several coordinate planes (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 to 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 drilling down into the details too much at this point. We will then build a complete example and create a small application displaying a chart using `KDChart::Widget` and `KDChart::Chart` 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 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 object.

```
// 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 data point within a dataset, for a dataset or for the whole diagram. See the API Reference for `KDChart::AbstractDiagram`:

```
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 in order 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 chapters. Also, we recommend you check out and run the examples shipped together with your KD Chart package.

Two Ways To Your Chart

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

Widget Example

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

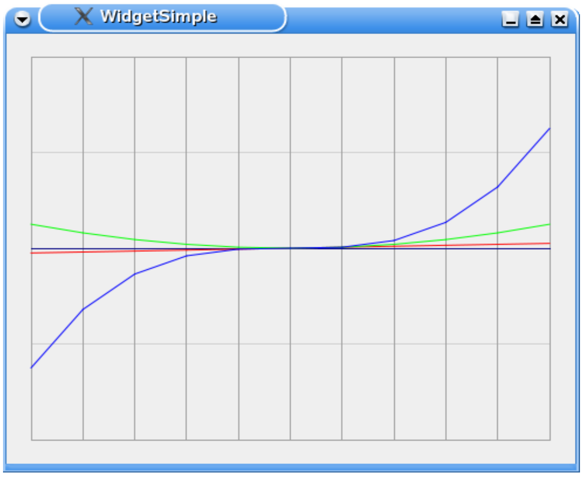
```
1
2
3
```

The result of the code above will display the simple widget presented in the screenshot below.

As you can see, the code is straightforward:

- Include the headers and bring in the Chart namespace.
- Declare your `KDChart::Widget`
- Use a `QVector` to store the data to be displayed.
- Assign the stored data to the widget, using one of the available `setDataset()` methods.

Figure 3.1. A Simple Widget



Of course, it is possible to add new elements like Title, Headers, Footers, Legends, or Axes to this simple widget as we will see later in greater detail. Notice also that the default diagram displayed by `KDChart::Widget` is a `KDChart::LineDiagram`. In the following example, we will look at how to display a Chart widget using `KDChart::Chart`.

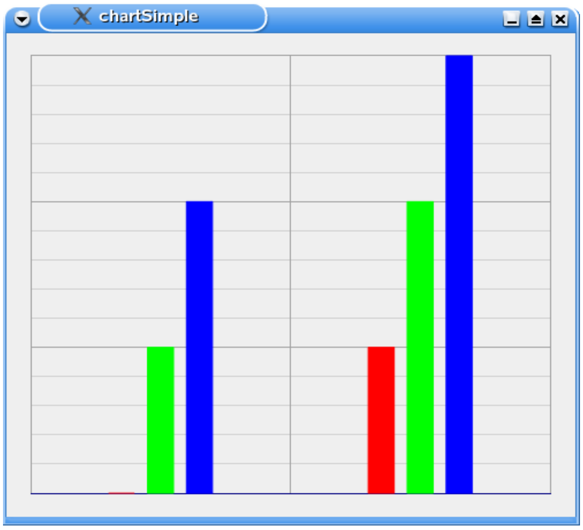
Chart Example

The following example is available at the following location of your KD Chart installation: `/examples/Bars/Simple/`

```
1
2
3
```

In this example, we are making use of `QStandardItemModel` in order to insert and store the data to be displayed by the diagram. We are also implicitly using a `KDChart::BarDiagram` to which we assign the model. See below for the resulting chart widget created by this example.

Figure 3.2. A Simple Chart



We can of course add more elements to this chart and change its default attributes as described above.

We will see in more detail how to configure those attributes (Pen, Color, etc ...) and add the various elements (Axes, Legend, Headers etc...) later.

What's Next

In the next chapter, we will describe the different available chart types (diagrams) and their coordinate planes. For each chart type, we will look at the attributes available for this particular type, and give you a few examples.

Chapter 4. Planes and Diagrams

KD Chart provides two types of planes in order to display the different types of diagrams it supports.

- A Cartesian coordinate plane, determined by a horizontal and a vertical axis, often called the x axis and y axis.
- A Polar coordinate plane which makes use of the radius and the polar angle which defines the position of a point on 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 of some sample code and/or small programs and their screenshots.

It will also give us an idea about which chart type could be appropriate for a specific purpose, and provides information about the features that are available for each type of chart. Let us first go through some important concept concerning the planes and their relation to the diagrams and the chart view itself.

Each coordinate plane can have one or more diagram associated to it. Those diagrams will share the scale provided by the plane. A chart can also have more than one coordinate plane. This makes it possible to have multiple diagrams using different scales and displayed next to, or on top of each other in the same chart.

Note

There are two ways in which planes can be caused to interact in where they are positionned layouting wise: The first is the reference plane.

By calling the `setReferenceCoordinatePlane()` method explained in the API documentation of `KDChart::AbstractCoordinatePlane` you declare the respective plane to be layouted in the same cell as the plane it is referenced too ("overlaying").

Also when planes share an axis they will be layed out in relation to each other as suggested by the position of the axis. If, for example Plane1 and Plane2 share an axis at position Left, that will result in the layout: Axis Plane1 Plane 2, vertically. If Plane1 also happens to be Plane2's reference plane, both planes are drawn over each other.

The reference plane concept allows two planes to share the same space even if none has axis, and in case there are shared axis, it is used to decide whether the planes should be painted on top of each other or layed out vertically or horizontally next to each other.

The above concept is illustrated in `examples/SharedAbscissa/OverlayedDiagrams/` and study those examples. `examples/SharedAbscissa/SeparateDiagrams/`, we recommend you

Cartesian Coordinate Planes

KD Chart uses the Cartesian coordinate system, and in particular its `KDChart::CartesianCoordinatePlane` class for displaying chart types such as lines, bars, points, etc.

In this section, we will describe and present all of the chart types using the default Cartesian coordinate plane.

In general, in order to implement a particular type of chart, just create an object of this type by calling `KDChart::`*`[type]`*`Diagram`, or if your are using `KDChart::Widget`, call its `setType()` method 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.

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

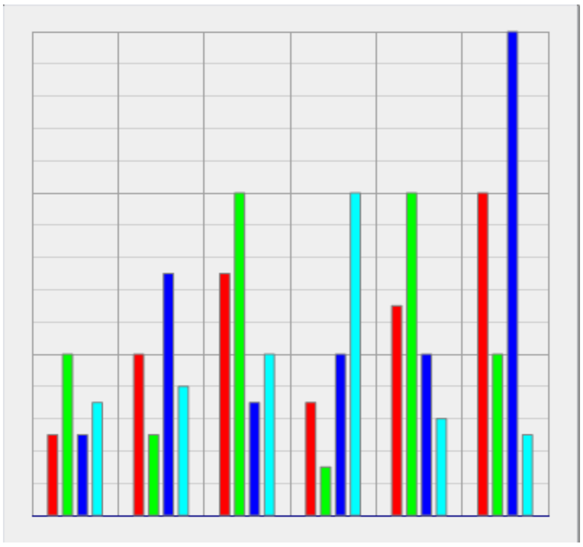
- Normal
- Stacked
- Percent

Normal Bar Charts

Tip

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

Figure 4.1. A Normal Bar Chart



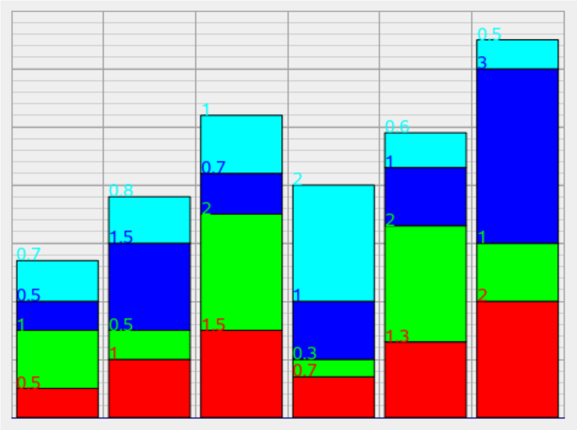
By default, a normal bar chart is displayed. You can switch to other bar chart types using `setType( Stacked )`.

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

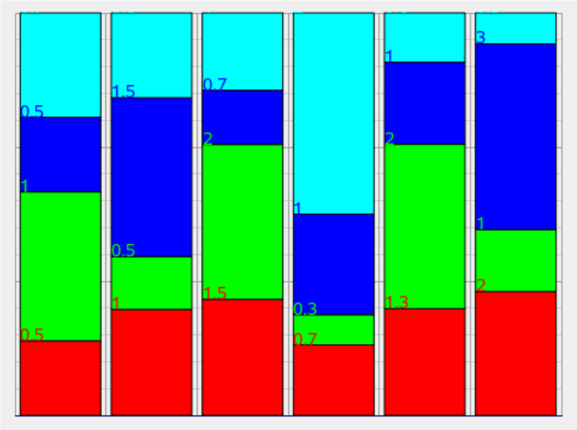


For stacked bar chart mode call `KDChart::BarDiagram::setType( Stacked )`.

Percent Bar Charts

Unlike Stacked charts Percent bar charts are not suitable for comparing the sums of the data series, but they 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 `KDChart::BarDiagram` function `setType( Percent )`.

Note

Three-dimensional look of the bars does not require a separate diagram type; you can enable it for all types (`Normal`, `Stacked`, and `Percent`) by setting its ThreeD attributes; we will describe this in codexample further on.

Code Sample

For now, let us look at the following code sample based on the `Simple Widget` example you have already seen. In this example, we show you how to configure your bar diagram and change its attributes when working with a `KDChart::Widget`.

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 later.

```
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. `KDChart::Widget` with its `setType()` and `setSubType()` methods allow us to achieve that in a very simple 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 `KDChart::Widget::setType()` We can also change the sub type of our bar chart later, e.g. by calling `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 them look nicer. This is referred to as configuring the attributes in a diagram. To do so, we configure a QPen and then assign it to our diagram. `KDChart::Widget::barDiagram()` will get a pointer to our widget diagram. As you can see, it is very easy to assign a new pen to our diagram by calling the diagram `KDChart::AbstractDiagram::setPen()` method.

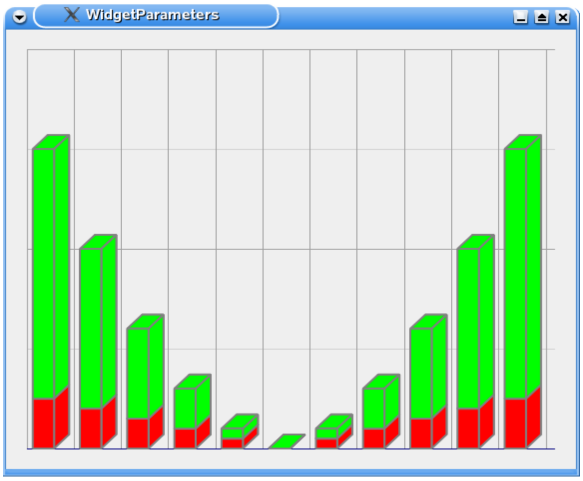
```
    //Set up your ThreeDAttributes
    //display in ThreeD mode
    ThreeDBarAttributes td(
        widget.barDiagram()->threeDBarAttributes() );
    td.setDepth(  15 );
    td.setEnabled(  true );
    widget.barDiagram()->setThreeDBarAttributes( td );
```

We want our bar chart to be displayed in 3D mode and need to configure some ThreeDBarAttributes and assign them to our diagram. Here we are configuring the depth of the 3D bars and enable 3D mode. Depth is an attribute only available to bar charts, and its setter and getter methods are implemented in the `KDChart::ThreeDBarAttributes`, whereas the `KDChart::AbstractThreeDAttributes::setEnabled()` is a generic attribute available to all chart types. Both of those attributes are made available at different levels in order to provide a better attribute structure.

```
    widget.show();

    return app.exec();
}
```

See the screenshot below to view the resulting chart displayed by the code shown above.

Figure 4.4. A Simple Bar Chart Widget



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 `KDChart::BarDiagram` making use of a `KDChart::Chart` is done in the same way as for a `KDChart::Widget`. You just need to assign the configured attributes to your bar diagram and assign it to the chart by calling `KDChart::Chart::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 the KDChartBarAttributes API Reference to get an idea about what can be configured there.

Note

KD Chart 2 API separates the attributes specific to a chart type itself and the generic attributes which are common to all chart types, for example: the setters and getters for a brush or a pen and that are accessible from the `KDChart::AbstractDiagram` 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 for this purpose `KDChart::BarDiagram::setBarAttributes()` or for example (to change the default Pen directly) by calling the `KDChart::AbstractDiagram::setPen()` method.

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

- Create a `KDChart::BarAttributes` object by calling `KDChart::BarDiagram::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 supports the following attributes for the Bar chart type. Each of those attributes can be set and retrieved in the way we describe 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 clear by looking at the following sample code that describes the above process. We recommend you compile and run the following example which is located in the `examples/Bars/Parameters/` directory of your KD Chart installation.

First of all we are including the header files and 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 specific to the Bar diagrams.

In this example we are using a `KDChart::Chart` 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 stored our data into the model, we create a diagram, in this case, we want to display a `KDChart::BarDiagram` 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 `KDChart::BarAttributes` to do so.

```
    BarAttributes ba( diagram->barAttributes() );
    //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 the KDChart::BarDiagram API Reference 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 API `KDChart::DataValueAttributes`. Those attributes are not specific to the Bar Chart types but can be used by any type of charts. The procedure is very similar.

```
    // display the values
    DataValueAttributes dva( diagram->dataValueAttributes() );
    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 `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. After it is configured as we want it, we just have to assign it to the diagram as with all other attributes.

Finally I want to paint a line around one of the datasets bars in order to bring the attention of the viewer to 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 specific 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 `KDChart::AbstractDiagram` 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 the `KDChart::AbstractDiagram` API Reference about those methods).

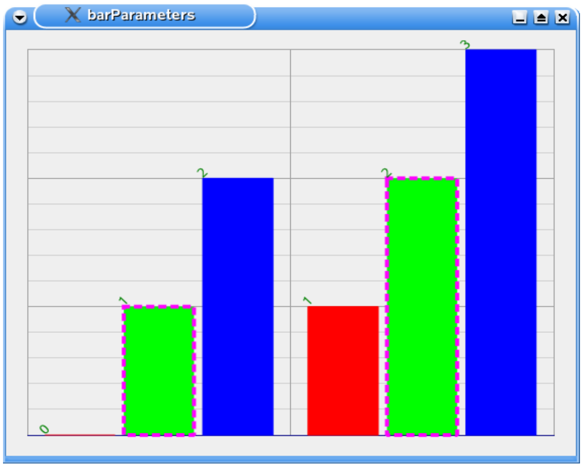
Our attribute having been configured and assigned we just need to assign the Bar diagram to our chart and conclude the implementation.

```
        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 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 `KDChart::BarDiagram::setType()` method.

Note

ThreeDAttributes for the different chart types are implemented as its own class, the same way as for the other attributes. We will talk more in details about KD Chart 2 ThreeD features in the section called “ThreeD Attributes” of Chapter 8, *Customizing your Chart*.

Tips and Tricks

In this section we want to give you some examples about how to use some of the interesting features offered by the KD Chart 2 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).
- Switch between the default (vertical) and the horizontal bar drawing mode.
- 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 `KDChart::AbstractDiagram::setPen()`, `KDChart::DataValueAttributes` and `KDChart::TextAttributes` as well as typical bar attributes only applyable to the Bar types as `KDChart::BarAttributes::setWidth()` or `KDChart::ThreeDBarAttributes`

We are making use of a `KDChart::Chart` class and also of a home made `TableModel` for convenience which is derived from `QAbstractTableModel`.

TableModel uses a simple rectangular vector of vectors to represent a data table that can be displayed in regular Qt 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 consult the "TableModel" interface and implementation files which are located in the `examples/tools/` directory of your KD Chart installation.

Let us cnow oncentrate on our Bar chart implementation 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
2
3
```

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 bar chart attributes manually . As you can see we are using a `KDChart::Chart` object (`m_chart`), a `KDChart::BarDiagram` object (`m_bars`), and our home made `TableModel` (`m_model`).

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

```
1
2
3
```

First of all we are adding our chart to the layout just like any other Qt widget. Then we 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 `KDChart::BarDiagram::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 text to be printed in different colors - according to its respective dataset's color. 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 user to display or not display the text dynamically. see `KDChart::TextAttributes::setVisible()`.

To display our diagram in threeD mode we configure its global `KDChart::ThreeDBarAttributes`. Here we are enabling or disabling the 3D look, and we adjust the depth of the bars according to user settings.

```
...
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 `KDChart::AbstractDiagram::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:

- 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.

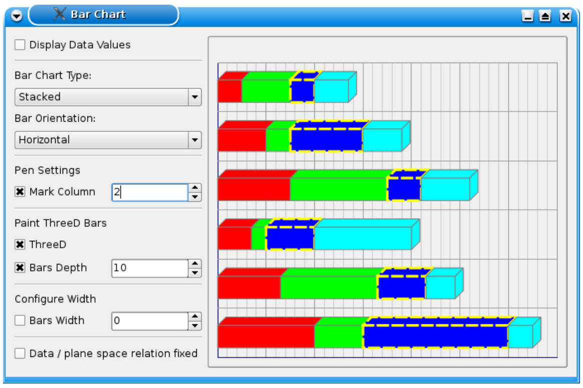
Finally we configure a typical `KDChart::BarAttributes`, the Bar Width, for the user to be able to change the width of the bars dynamically increasing or decreasing its value via the Gui.

```
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 `KDChart::BarAttributes::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



☐ Display Data Values

Bar Chart Type: Stacked

Bar Orientation: Horizontal

Pen Settings

☒ Mark Column 2

Paint ThreeD Bars

☒ ThreeD

☒ Bars Depth 10

Configure Width

☐ Bars Width 0

☐ Data / plane space relation fixed

This example is available to compile and run from the `examples/Bars/Advanced/` directory in your KD Chart installation.

Line Charts

Line charts usually show numerical values and their development over 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 with Bar types, KD Chart can generate line charts of different types. `KDChart::LineDiagram` supports the following subtypes explained below:

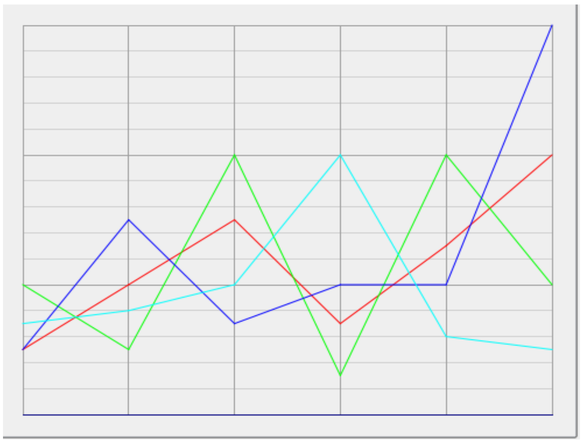
- Normal Line Chart
- Stacked Line Chart
- Percent Line chart

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



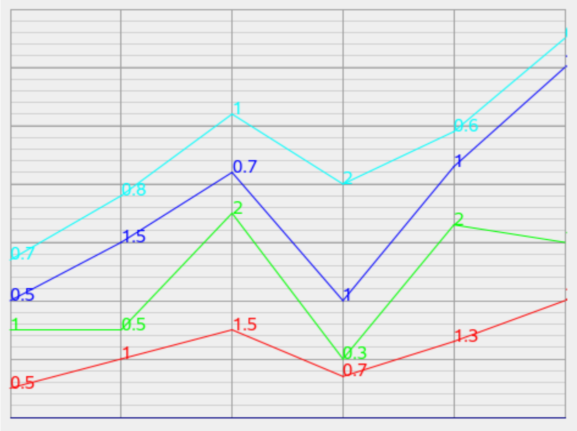
KD Chart draws normal line charts by default when in line chart mode so no method needs to be called to get one, however after having used your `KDChart::LineDiagram` to display another line chart subtype you can reset it by calling `setType( Normal )`.

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



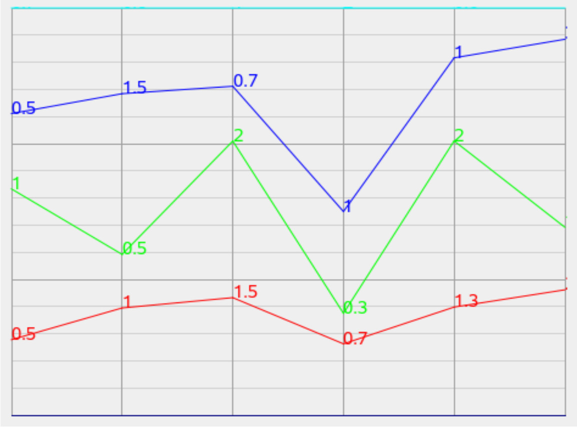
Stacked mode for line charts is activated by calling the `KDChart::LineDiagram` method `setType( Stacked )`.

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 `KDChart::LineDiagram` function `setType( Percent )`.

Note

Three-dimensional look of the lines can be enabled for all types (`Normal`, `Stacked` or `Percent`) by setting its ThreeD attributes class (see the KDChart::ThreeDLineAttributes API Reference for details). 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, see Chapter 3, *Basic steps: Create a Chart* - the section called “Widget Example”. In this example we demonstrate how to configure your line diagram and change its attributes when working with a `KDChart::Widget`.

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 using the `KDChart::Widget` with its `setSubType()` method.

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

The default sub-type being Normal for all types of charts we need to call implicitely `KDChart::Widget::setSubType()` in this case. We can also change the sub-type of our line chart further on by calling `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 draw attention to it. To do so we configure a QPen and then assign it to our diagram. `KDChart::Widget::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 `KDChart::AbstractDiagram::setPen()` method.

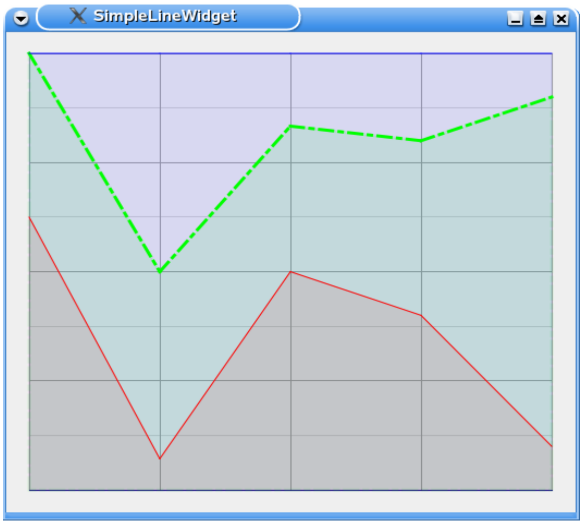
```
    //Display in Area mode
    LineAttributes ld( widget.lineDiagram()->lineAttributes() );
    ld.setDisplayArea( true );
    //configure transparency
    //it is nicer and let us
    //all the area
    ld.setTransparency( 25 );
    widget.lineDiagram()->setLineAttributes( ld );
```

The code above makes use of typical `KDChart::LineAttributes` 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 Chart Widget



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 `KDChart::LineDiagram` making use of a `KDChart::Chart` is done the same way as for a `KDChart::Widget`. You just need to assign the configured attributes to your line diagram and assign the diagram to the chart by calling `KDChart::Chart::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 `KDChart::AbstractDiagram` from which `KDChart::LineDiagram` is derived indirectly.

However to make it simple for the user we have added some convenient functions to `KDChart::LineAttributes` 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 the section called “Area Charts” in this Chapter.

`KDChart::LineDiagram` combined with its attributes and methods or combined together with `KDChart::MarkerAttributes` 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 `KDChart::LineDiagram::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 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 look at the following sample code that describes the above process. The following example which is located in 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 specific to the Line diagram.

In this example we are using a `KDChart::Chart` 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*
            }
        }

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

After having stored our data in the model, we create a diagram. In this case, we want to display a `KDChart::LineDiagram`. 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 their text and font.

```
// Display values
// 1 - Call the relevant attributes
DataValueAttributes dva( diagram->dataValueAttributes() );
// 2 - We want to configure the font and colors
//     for the data value texts.
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

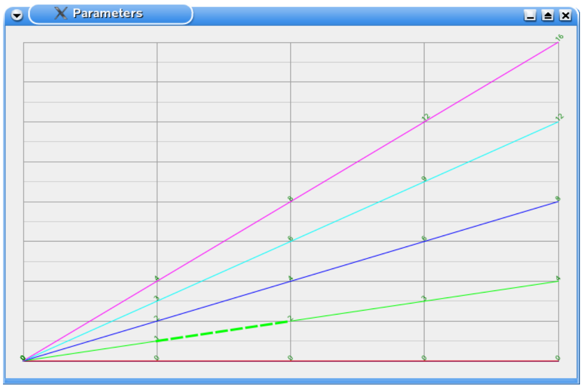
The Pen and the Brush setters and getters are implemented at a lower level in our `KDChart::AbstractDiagram` class for a cleaner code structure. Those methods are of course used by all types of diagrams and their configuration is very simple and straight forward as you can see in the above sample code.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

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 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 `KDChart::LineDiagram::setType()` method.

Note

ThreeDAttibutes for the different chart types are implemented in their own class, the same way as for the other attributes. We will talk more in detail about KD Chart 2 ThreeD features in the section called “ThreeD Attributes” of Chapter 8, *Customizing your Chart*.

Tips and Tricks

In this section we want to give you some examples of the interesting features offered by the KD Chart 2 API. We will study the code and display a screen-shot 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 `KDChart::AbstractDiagram::setPen()`, `KDChart::DataValueAttributes` and `KDChart::TextAttributes`.

We are making use of a `KDChart::Chart` class and also of a home made `TableModel` for 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 views. Additionally, it provides a method to load CSV files exported by OpenOffice Calc in the default configuration. This allows for preparation of 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 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
2
3
```

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 `KDChart::Chart` object (`m_chart`), a `KDChart::LineDiagram` object (`m_lines`), and our home made `TableModel` (`m_model`).

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

```
1
2
3
```

First of all we are adding our chart to the layout as with 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 `KDChart::BarDiagram::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 value texts 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 value texts 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 there are 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.

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

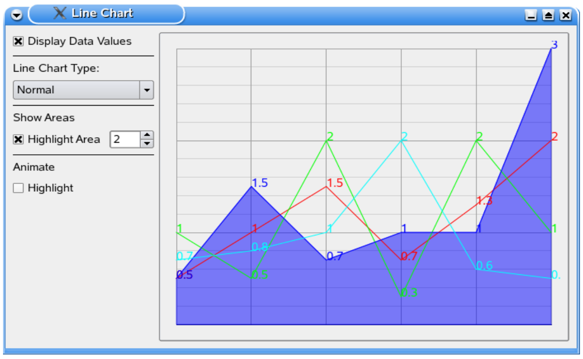
```
....
LineAttributes la = m_lines->lineAttributes( column );
if ( checked ) {
    la.setDisplayArea( true );
    la.setTransparency( opacity );
}  else {
    la.setDisplayArea( false );
}
m_lines->setLineAttributes( column, la );
...
m_chart->update();
...
```

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 in your KD Chart installation. The widget displayed by the above code is shown in the figure below.

Figure 4.12. A Full featured Line Chart



☒ Display Data Values

Line Chart Type:

Normal

Show Areas

☒ Highlight Area 2

Animate

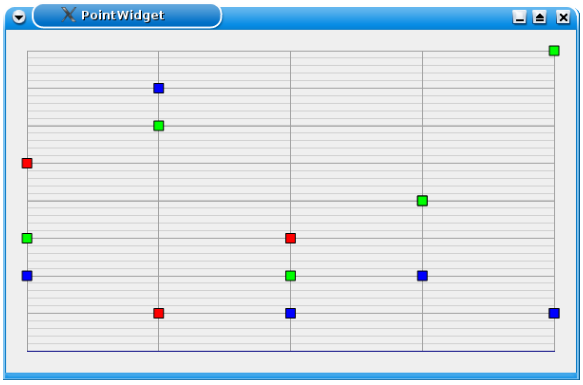
☐ Highlight

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

Point charts often are used to visualize a large amount 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 `KDChart::Widget` and a `KDChart::LineDiagram` but of course the process is very similar if we were working with a `KDChart::Chart`.

We recommend you 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 `KDChart::DataValueAttributes` and its `KDChart::MarkerAttributes` available to all other diagram types supported by KD Chart 2.

For this reason we will for now point you to the sections related to those subjects and in particular to Chapter 8, *Customizing your Chart* - the section called “Markers Attributes” or the section called “Data Values Attributes” 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 API. We will study the code and display a screen-shot showing the resulting widget.

A complete Point Example

In the following implementation we want to be able to:

- Specify the points' styles and their sizes.
- Switch between point chart line chart.
- Display the chart in Normal / Stacked / Percent mode.
- Show or hide the data value texts.

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.

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 `KDChart::Chart` object (`m_chart`), a `KDChart::LineDiagram` object (`m_lines`), and our home made `TableModel` (`m_model`).

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

Here we will not comment on the code in detail as it is similar to what we have seen before in our line chart example, but only pick out the interesting parts 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.

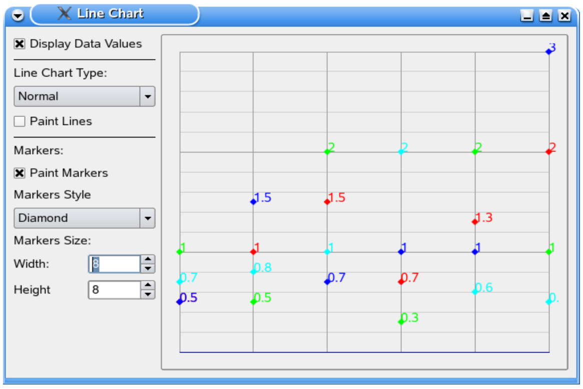
```
...
MarkerAttributes::MarkerStylesMap map;
map.insert( 0, MarkerAttributes::MarkerSquare );
map.insert( 1, MarkerAttributes::MarkerCircle );
map.insert( 2, MarkerAttributes::MarkerRing );
map.insert( 3, MarkerAttributes::MarkerCross );
map.insert( 4, MarkerAttributes::MarkerDiamond );
...
MarkerAttributes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

The user may also change the size of the marker form the GUI and this is implemented in a straight forward way by using `KDChart::MarkerAttributes` method `setMarkerSize()`.

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

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

Figure 4.14. A Full featured Point Chart



☒ Display Data Values

Line Chart Type: Normal

☐ Paint Lines

Markers:

☒ Paint Markers

Markers Style: Diamond

Markers Size:

Width: 8

Height: 8

Note

For two-dimensional data you would use the same technique as described above, but applying it to the `KDChart::Plotter` class, for details please have a look at `examples/Plotter/BubbleChart/`.

Area Charts

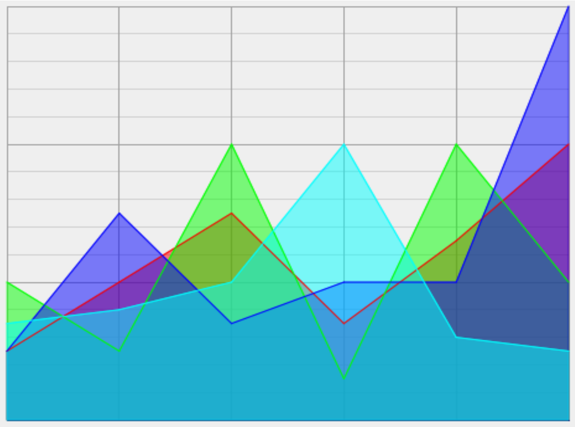
Even more than a Line Chart (of which they are attributes) an area chart can give a good visual impression of different datasets and their relation to each other. For example this chart type might be ideal for showing how several sources contributed to increasing ozone values in a conurbation during a summer's months.

Area charts are Line Charts and thus based upon several points which are connected by lines—the difference compared to the line chart is that the area below a line is filled by the respective dataset's color. This gives a clear indication 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 in the section called “Area Attributes” below. KD Chart 2 supports of course Area display for all subtypes of line charts and thus allows 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 `KDChart::LineDiagram::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 );
// Make the areas visible
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

`KDChart::LineAttributes` 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 specific 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 available, which give us full flexibility to configure our Area chart.

Tips and Tricks

In this section we will give you some examples of the interesting features offered by the KD Chart 2 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 in codexample. 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 or hide the data values texts
- Select the line chart type (Normal, Stacked, Percent)
- Display areas for each dataset on its own.

We are using a `KDChart::Chart` class and also a home made `TableModel` for convenience. It is derived from `QAbstractTableModel`.

We recommend you 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
2
3
```

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 `KDChart::Chart` object (`m_chart`), a `KDChart::LineDiagram` object (`m_lines`), and our home made `TableModel` (`m_model`).

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

```
1
2
3
```

First of all we are adding our chart to the layout as we would do with any other Qt widget. We then 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 `KDChart::BarDiagram::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 value texts 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 value texts 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:

- 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.

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

```
....
LineAttributes la( m_lines->lineAttributes( column ) );
```

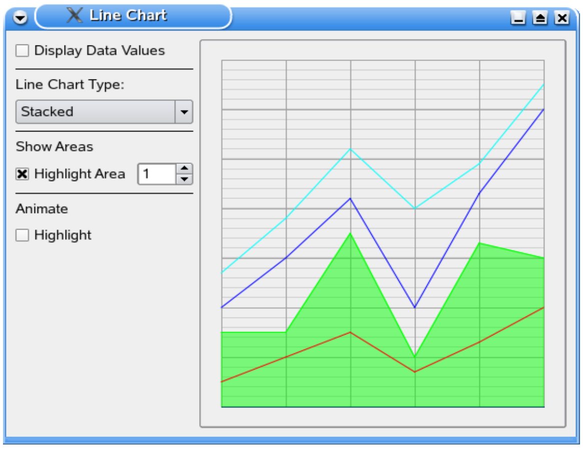
```
if ( checked ) {
    la.setDisplayArea( true );
    la.setTransparency( opacity );
} else {
    la.setDisplayArea( false );
}
m_lines->setLineAttributes( column, la );
...
m_chart->update();
...
```

This is implemented by configuring our line attributes and assigning 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 in your KD Chart installation. The widget displayed by the above code is shown in the figure below.

Figure 4.16. A Full featured Area Chart



Line Chart

☐ Display Data Values

Line Chart Type: Stacked

Show Areas

☒ Highlight Area 1

Animate

☐ Highlight

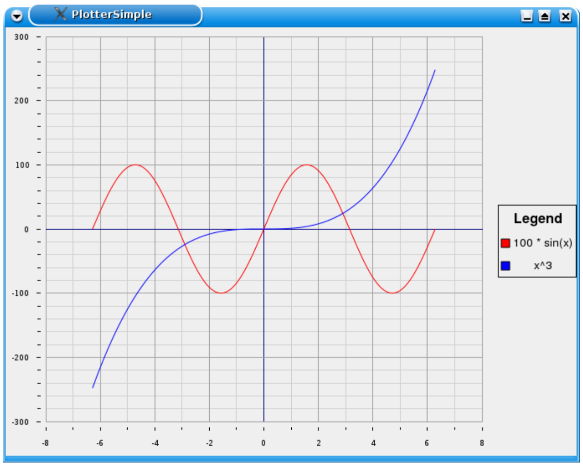
Plotter Charts

Plotter charts are almost the same as normal line diagrams with one important exception: Line diagrams always expect the values running from 1..n having step width 1. Plotters can instead of that handle free X/Y-pairs in any order and not being equidistant.

Therefore, `KDChart::Plotter` expects two columns in the model for each dataset being plotted. See the example below to find out how to use this. Apart fom that difference, please refer to the section called “Line Charts” in this manual explaining how to set the attributes for this diagram type.

The following screenshot is made from the plotter example in `examples/Plotter/Simple/`

Figure 4.17. A simple Plotter diagram



Plotter Sample Code

The following code sample is plotting a sine wave and an exponential curve from -2*pi - 2*pi consisting of 400 points on the x-axis:

```
QStandardItemModel model( points, 4 );

double x = -2 * 3.141592653589793;
for( int n = 0; n < 400; ++n ) {
    QModelIndex index = model.index( n, 0 );
    model.setData( index, QVariant( x ) );
    // the x value: x
    index = model.index( n, 1 );
    // the y value sin( x ) * 100
    model.setData( index, QVariant( sin( x ) * 100 ) );

    index = model.index( n, 2 );
    model.setData( index, QVariant( x ) );
    index = model.index( n, 3 );
    model.setData( index, QVariant( x * x * x ) );

    x += 4 * 3.141592653589793 / 399.0;
}

KDChart::Chart chart;
KDChart::Plotter plotter;
plotter.setmodel( & model );
chart.coordinatePlane()->replaceDiagram( &plotter );

chart.show();
```

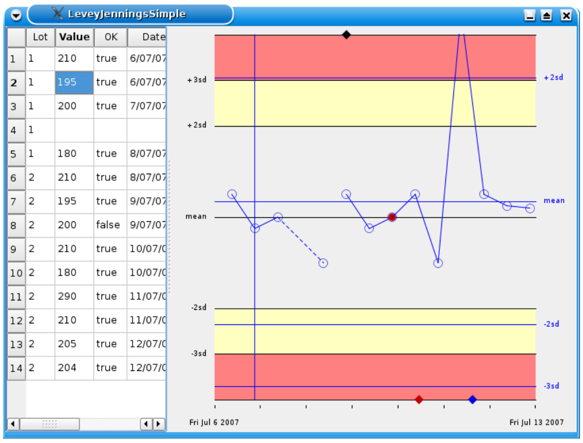
Levey-Jennings Charts

A Levey-Jennings chart is a graph on which quality control data is plotted to give a visual indication as to whether a laboratory test is working well or not.

If you are interested in using this diagram type please have a look at the API Reference for the classes `KDChart::LeveyJenningsDiagram` (derived from `KDChart::LineDiagram`) and `KDChart::LeveyJenningsAxis` (derived from `KDChart::CartesianAxis`).

The following screenshot is made from the Levey-Jennings example in `examples/LeveyJennings/Simple/`

Figure 4.18. A simple Levey-Jennings diagram



	Lot	Value	OK	Date
1	1	210	true	6/07/07
2	1	195	true	6/07/07
3	1	200	true	7/07/07
4	1			
5	1	180	true	8/07/07
6	2	210	true	8/07/07
7	2	195	true	9/07/07
8	2	200	false	9/07/07
9	2	210	true	10/07/0
10	2	180	true	10/07/0
11	2	290	true	11/07/0
12	2	210	true	11/07/0
13	2	205	true	12/07/0
14	2	204	true	12/07/0

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 and Polar.

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 `KDChart::Widget` you will need to call its `setType()` and specify the appropriate chart type. (e.g Widget::Pie, Widget::Polar etc...)

Pie Charts

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. 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.

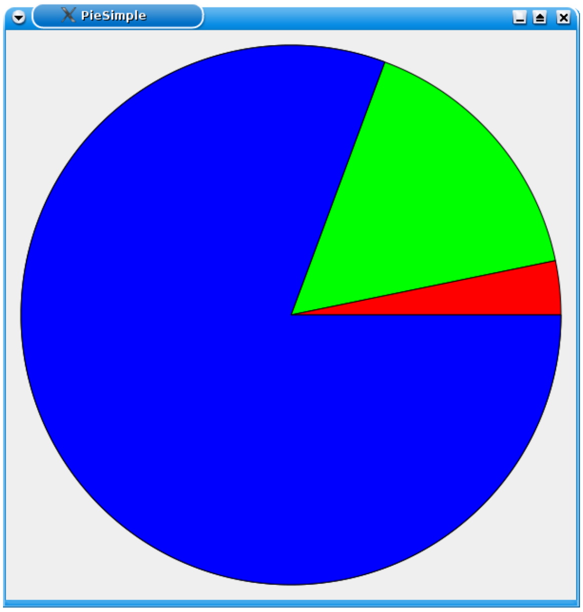
Pie charts typically consist of two or more pieces any number of which can be shown 'exploded' (shifted away from the center) at different amounts, the starting position of the first pie can be specified and your pie chart can be drawn in three-D look. Activating the pie chart mode is done by calling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

The three-dimensional look of the pies can be enabled by setting its ThreeD attributes, we will describe this in Chapter 8, *Customizing your Chart* - the section called “ThreeD Attributes” below.

Simple Pie Charts

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

Figure 4.19. A Simple Pie Chart



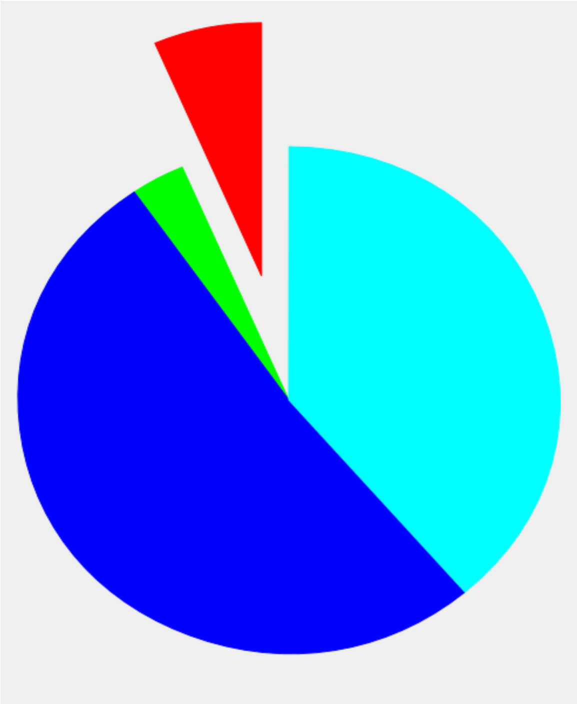
KD Chart by default draws two-dimensional pie charts when in pie chart mode so no method needs to be called to get one. We describe more in detail about how to obtain three dimensional look for a pie chart in the following the section called “Pies Attributes”.

Exploding Pie Charts

Tip

Explode individual segments to emphasize individual data.

Figure 4.20. An Exploding Pie Chart



We will go through all the configuration possibilities in the section called “Pies Attributes” 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, see Chapter 3, *Basic steps: Create a Chart* - the section called “Widget Example”. In this example we demonstrate how to configure your Pie diagram and change its attributes when working with a `KDChart::Widget`.

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 `KDChart::Widget::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 );
```

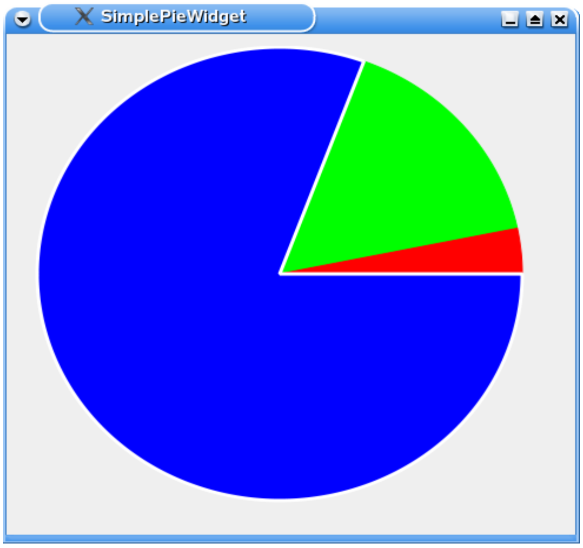
Here we are configuring the pen "attribute". As you can see it is straight forward. `KDChart::Widget::pieDiagram()` allow us to get a pointer to our widget diagram. As you can see it is very simple to assign a new pen to our diagram by calling the diagram `KDChart::AbstractDiagram::setPen()` method.

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 4.21. 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 `KDChart::PieDiagram` making use of a `KDChart::Chart` is done the same way as for a `KDChart::Widget`. You just need to assign the configured attributes to your pie diagram and assign the diagram to the chart by calling `KDChart::Chart::replaceDiagram()`.

Pies Attributes

By "Pie attributes" we are talking about all parameters that can be configured and set by the user and which are specific to the Pie Chart type. KD Chart 2 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 the `KDChart::AbstractDiagram` or `KDChart::PieAbstractDiagram`, etc ...)

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

The procedure is straight forward:

- Create a `KDChart::PieAttributes` object by calling `KDChart::PieDiagram::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 supports the following attributes for the Pie chart type. Each of those attributes can be set and retrieved in the way we describe 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 diagramsegments will show facettes instead of rounded segments. Can only be specified for the whole diagram.
- PieAttributes: set or retrieve the pie diagram Attributes. (see: `KDChart::AbstractPieDiagram`)
- ThreeDPieAttributes: set or retrieve the diagram ThreeDAttributes. (see: `KDChart::AbstractPieDiagram`)

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 `KDChart::PieAttributes::setExplode()` or `KDChart::PieAttributes::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( column ) );
// 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 detail in Chapter 8, *Customizing your Chart* - the section called “ThreeD Attributes” later on.

Pie Attributes Sample

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

First of all we are include the header files and bring in the 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 specific to the Pie types.

In this example we are using a `KDChart::Chart` 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 `KDChart::PieDiagram`. 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 specific `KDChart::PieAttributes`.

```
    // Configure some Pie specific attributes

    // explode a section
    PieAttributes pa( diagram->pieAttributes( 1 ) );
    pa.setExplodeFactor( 0.1 );

    // Assign the attributes
    // to the 2nd dataset of 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.

```
QPen sectionPen( diagram->pen( 1 ) );

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

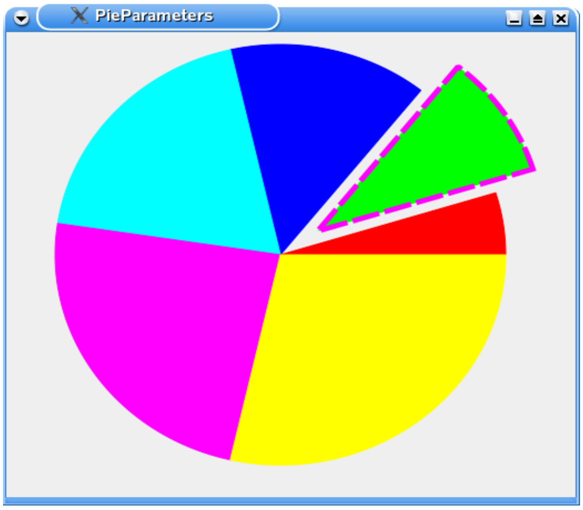
The Pen and the Brush setters and getters are implemented at a lower level in our `KDChart::AbstractDiagram` class for a cleaner code structure. Those methods are of course used by all types of diagrams and their configuration is very simple and straight forward as you can see in the above sample code.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

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

```
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 compile and run the example related to this section and located in the `examples/Pie/Parameters/` directory of your KD Chart installation.

Figure 4.22. Pie With Configured Attributes



Tips and Tricks

In this section we will to go through some examples about how to use the interesting features offered by the KD Chart 2 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 3D appearance.
- 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 `KDChart::Chart` class and also a home made `TableModel` for convenience. It is derived from `QAbstractTableModel`.

We recommend you 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
2
3
```

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 `KDChart::Chart` (`m_chart`), a `KDChart::PieDiagram` (`m_pies`), and our home made `TableModel` (`m_model`).

Note

Before displaying our Pie diagram we need to implicitely replace the default cartesian plane by a `KDChart::PolarCoordinatePlane`.

```
1
2
3
```

First of all we are adding our chart to the layout as we would 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 `KDChart::PieAttributes` 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();
```

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

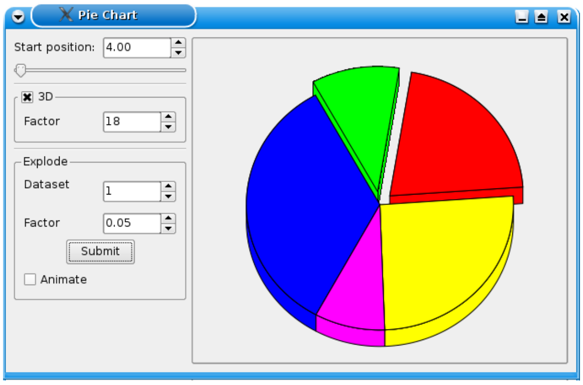
```
....
// note: We use the per-column getter method here, it will fall back
// automatically to return the global (or even the default) settings.
PieAttributes pa( m_pie->pieAttributes( column ) );
pa.setExplodeFactor( value );
m_pie->setPieAttributes( column, pa );
...
m_chart->update();
...
```

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 in your KD Chart installation. The widget displayed by the above code is shown in the figure below.

Figure 4.23. A Full featured Pie Chart



Start position: 4.00

☒ 3D

Factor 18

Explode

Dataset 1

Factor 0.05

☐ Animate

Polar Charts

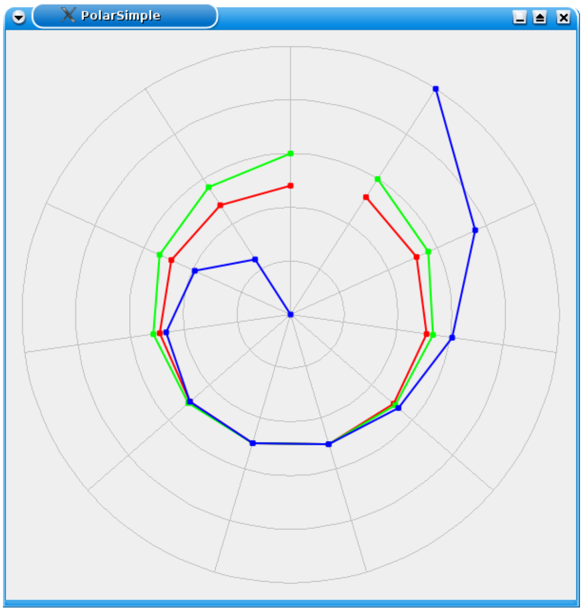
Polar charts get their name from displaying "polar coordinates" instead of Cartesian coordinates, thus they are using the `KDChart::PolarCoordinatePlane`.

To instantiate a polar chart you may call the `KDChart::Widget` function `setType(Widget::Polar)`, or create an object of type `KDChart::PolarDiagram` and assign it to your `KDChart::Chart` by calling its `replaceDiagram()` method.

A Simple Polar Chart

Compile and run the example file `examples/Polar/Simple/` to see a normal polar chart as shown below.

Figure 4.24. A Normal Polar Chart



Polar Attributes

In addition to using the generic classes `KDChart::DataValueAttributes` and `KDChart::MarkerAttributes` available to all diagram types supported by KD Chart 2 the following setter methods are provided by the `KDChart::PolarDiagram`:

- `setRotateCircularLabels( bool )` determines whether circular labels are to be rotated automatically or not: If set the labels' base lines will be adjusted to the circular grid lines.
- `setCloseDatasets( bool )` may be used to close each of the data series by connecting the last point to its respective start point.

The `KDChart::PolarCoordinatePlane` provides an additional means of configuration that might make sense for your polar chart:

- `setStartPosition( qreal )` specifies the Position of the Zero degrees value and thus the rotation of your grid.
- `setGridAttributes( bool circular )` sets the attributes to be used for grid lines drawn in circular direction (or in sagittal direction, resp.).

  In example to hide the circular grid lines you would do this:

```
....
KDChart::PolarCoordinatePlane* plane =
    static_cast< PolarCoordinatePlane* >( m_chart->coordinatePlane() );

KDChart::GridAttributes attrs( plane->gridAttributes( true ) );
attrs.setGridVisible( false );
plane->setGridAttributes( true, attrs );
....
```

These additional example files are demonstrating the methods described above: `examples/Polar/Advanced/` and `examples/Polar/Parameters/`.

Tip

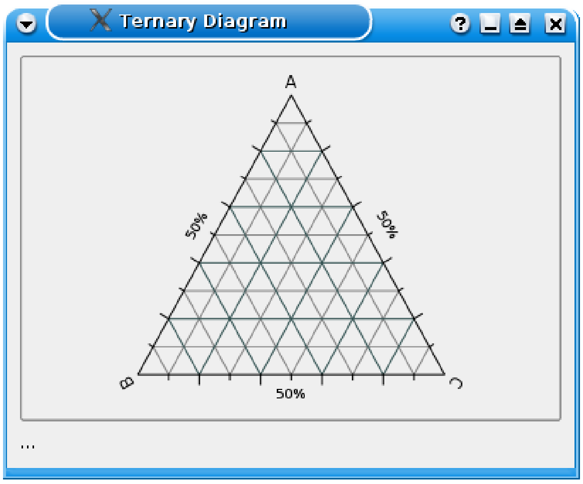
Currently only normalized polar charts can be shown: all values advance by the same number of polar degrees and there is no way to specify a data cell's angle individually. While this is ideal for some situations it is not possible to display true world map data like this since you can not specify each cell's rotation angle. Transforming your coordinates to the Cartesian system and using a Point Chart might be a solution in such cases.

Ternary coordinate plane

KD Chart has support for ternary charts and has therefore an appropriate coordinate plane. This is the class `KDChart::TernaryCoordinatePlane`.

The idea of ternary charts is to plot triple values on a triangle. Triple values are represented by three floating point values having the fixed sum 1.0. Therefore each plotted dataset needs three columns in the model.

Figure 4.25. A Simple Ternary Chart



Tip

KD Chart is using only the first two of them and calculates the third one out of those. If the sum of the first two columns is already greater than 1.0, the data triple is considered invalid and disregarded.

This section will describe the chart types which can be added to a ternary coordinate plane.

To use such a diagram, you need to create an instance of `KDChart::TernaryCoordinatePlane`. After that, you can make KD Chart using it by using `KDChart::Chart::replaceCoordinatePlane()` and add a diagram to it.

Ternary Line Charts

A ternary line charts connects all points of each dataset with a line.

Have a look at the following code example explaining how to work with it:

```
KDChart::Chart chart;
// replace the default (cartesian) coordinate plane with a ternary one
KDChart::TernaryCoordinatePlane* ternaryPlane
```

```
    = new KDChart::TernaryCoordinatePlane;
chart.replaceCoordinatePlane( ternaryPlane );
// make a ternary line diagram
KDChart::TernaryLineDiagram* diagram = new KDChart::TernaryLineDiagram;
// and replace the default diagram with it
ternaryPlane->replaceDiagram( diagram );

chart.show();
```

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 chapter.

Chapter 5. Axes

Axes are implemented at different levels in the KD Chart 2 API. KD Chart makes use of `KDChart::CartesianAxis` and `KDChart::TernaryAxis` which are derived from their common base class `KDChart::AbstractAxis`. .

The user may specify its own set of strings to be used as Axis labels with the `KDChart::AbstractAxis::setLabels()` 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 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 `KDChart::AbstractAxis::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 `KDChart::AbstractAxis`, we recommend studying the `KDChart::AbstractAxis` API Reference.

Tip

If you set a smaller number of strings than the number of 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 `KDChart::CartesianAxis` 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 specific cartesian features are implemented in `KDChart::CartesianAxis`.

Ternary Axis

The class `KDChart::TernaryAxis` is made for use with diagrams displayed in a ternary coordinate plane.

Since ternary diagrams are not rectangular but triangular, ternary axes can be added at three different positions relative to the diagram: South, East and West.

How to configure Cartesian Axes

In order to add axis to a cartesian diagram we need to use `KDChart::AbstractCartesianDiagram::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 `KDChart::AbstractDiagram::takeAxis()` method, before calling addAxis on the other diagram.

Note

`KDChart::AbstractDiagram::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 look at the following lines of code based on the `Simple Widget` example we have been demonstrating above, see Chapter 3, *Basic steps: Create a Chart* - the section called “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 `KDChart::Widget`..

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 later 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 `KDChart::Widget::setType()` method.

Now let us create our axes, 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:

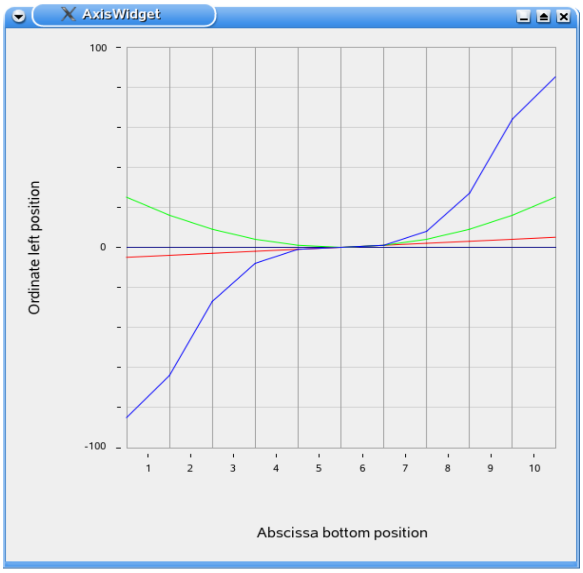
```
    widget.lineDiagram()->addAxis( xAxis );
    widget.lineDiagram()->addAxis( yAxis );
```

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/`.

In the section called “Tips” below we will present you a more elaborate example which uses `KDChart::Chart` 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 examples concening the interesting features offered by the KD Chart 2 API. We will study the code and display a screen-shot 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 `KDChart::Chart` class and also a home made `TableModel` for convenience. It is derived from `QAbstractTableModel`.

We recommend you 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
2
3
```

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

```
1
2
3
```

First of all we are adding our chart to the layout as for any other Qt widget. Load the data to be displayed 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 positioned 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 `KDChart::CartesianAxis::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 provided by in `KDChart::CartesianAxis` class, for details see its API Reference.

> Contained 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 title axis and also for our labels. The process is the same as for configuring any type of attributes, as follows:

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 by modifying the
// current settings valid for the bottom axis
// Note:
//   By default KD Chart is using the same text attributes
//   for all of its axes, so it does not matter which
//   axis we are asking in the following line of code here.
TextAttributes taLabels( bottomAxis->textAttributes() );
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

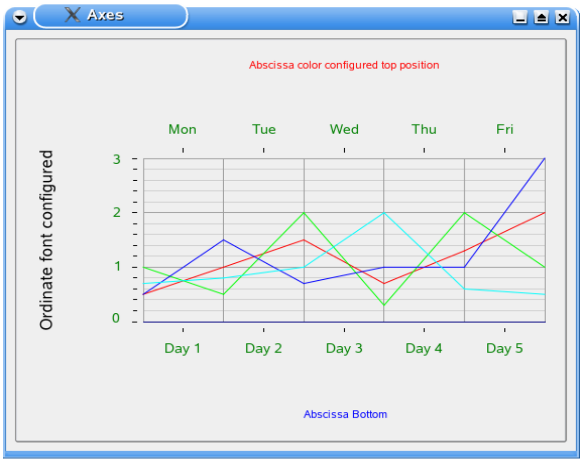
Labels specified via setLabels take precedence: if a non-empty list is passed, KD Chart will use these strings as axis labels, instead of calculating them.

Finally the last step is 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 in your KD Chart installation. We recommend checkig it out. 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.

Note

To replace the default tick marks / labels and have your own texts shown at your own positions instead please use `CartesianAxis::setAnnotations()` as shown in this piece of code:

```
QMap< double, QString > ordinateAnnotations;
ordinateAnnotations[3.3] = "three point three";
ordinateAnnotations[7.5] = "seven and a half";
ordinateAnnotations[16.0] = "sixteen";
ordinateAnnotations[-8] = "minus eight";
yAxis->setAnnotations( ordinateAnnotations );
```

Chapter 6. Legends

Legends can be drawn for all kind of diagrams and are drawn at the chart level (in relation to diagram level). We can have more than one legend per chart and add it to our chart or our widget view by using respectively `KDChart::Chart::addLegend()` or `KDChart::Widget::addLegend()`

Note

Legend is different from all other classes of 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 must call `KDChart::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 `KDChart::Legend`. Of course we also recommend that you consult its API Reerence as well as the documentation for `KDChart::Chart` and `KDChart::Widget` to have a complete idea over how to handle legends and what configurations parameters are available.

How to configure

In order to add a legend to our chart we need to use the `KDChart::Chart::addLegend()` method. The chart takes ownership of the legend and will take care of removing it by itself. The `KDChart::Chart` method above and the ones discussed in the paragraphs are similar for the `KDChart::Widget` class. In order to make the following description simpler we will only mention `KDChart::Chart` in the following paragraphs.

Tip

You may also wish to use `KDChart::Chart` `replaceLegend()` which is also available for 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.

If you want to re-use the old legend, call takeLegend and addLegend, instead of using replaceLegend.

Note

`KDChart::Chart::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 `KDChart::Legend` 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.
- Position, alignment and orientation are of course configurable.
- 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.

Note

The `KDChart::Position` class, defines positions, using compass terminology. Using this class you can specify one of nine pre-defined, logical points , in a similar way, as you would use a compass to navigate on a map.

Please consult the setters and getters methods available in the `KDChart::Legend` interface.

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 in Chapter 3, *Basic steps: Create a Chart* - the section called “Widget Example”. Through the following code we demonstrate how to add and position a Legend to your chart Widget using a KDChart::Widget.

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 `KDChartBarDiagram`. In order to be able to give a position 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 `KDChart::Widget::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 );
```

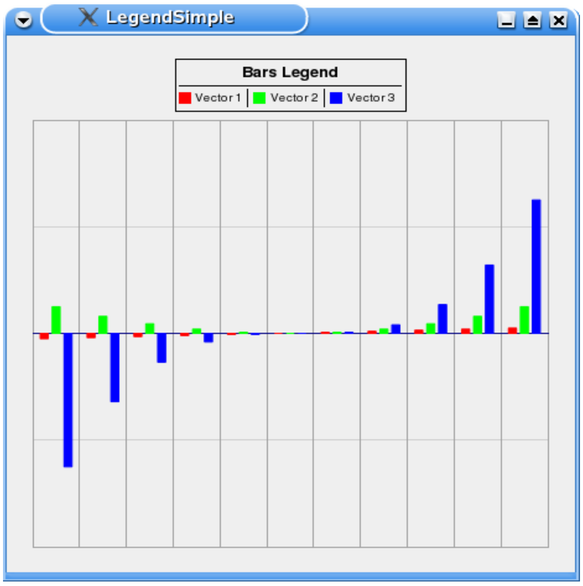
The interesting point here is how we call `KDChart::Widget::firstlegend()` to get a pointer to to our legend object and be able to set up and configure it. We will see further on in the next code example, see the section called “Tips” how to configure the elements of a legend (e.g Title text, markers,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

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

```
    widget.show();

    return app.exec();
}
```

See the screenshot below to view The resulting chart displayed by the above code.

Figure 6.1. A Widget with a simple Legend



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

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

Tips

In this section we want to give you some examples about how to use some interesting features offered by the KD Chart 2 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 include 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 handles the attributes specific to a legend, the setters and getters for the methods we have used here are implemented in the `KDChart::Legend` class. We recommend you consult its API Reference.

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 in the `KDChart::MarkerAttributes` class API Reference.

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 `KDChart::Legend::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 `KDChart::Legend::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 `KDChart::Legend::datasetCount()` which allow you to retrieve the count of the 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 attribute settings, the code is straight forward. Just create the attribute object, configure it and assign it. We recommend you have a look at the `KDChart::BackgroundAttributes` class API Reference.

Let us now configure our legend's frame:

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

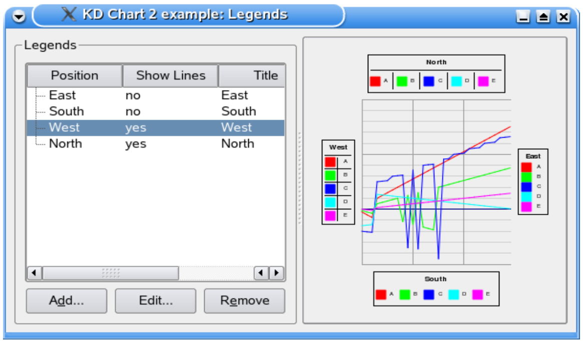
Same procedure as above. Please note the `setVisible()` method which is necessary as the default value hides those attributes.

Finally we will 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"
```

The screenshot shows the chart of the code listened above.

Figure 6.2. Legend advanced example



KD Chart 2 example: Legends

Legends

Position	Show Lines	Title
East	no	East
South	no	South
West	yes	West
North	yes	North

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 we will go through the several features and configuration possibilities available in KD Chart 2 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 consult the `KDChart::HeaderFooter` class API Reference to learn more about those features and methods.

How to configure

In order to add a header or a footer to our chart we need to use the `KDChart::Chart::addHeaderFooter()` method. The chart takes ownership and will take care of removing it by itself. This method and the ones discussed in the next paragraphs of this section are similar for the methods of the `KDChart::Widget` class. In order to make this description simpler we will only mention `KDChart::Chart` there.

Tip

You may also wish to use `KDChart::Chart` `replaceHeaderFooter()` which is also available for convenience:

The new header or footer to be used instead of the old one must not be zero. Otherwise the method will just do nothing. The second parameter of this method is the header or footer to be removed by the new one. This header or footer will be deleted automatically. If the parameter is omitted, the very first header or footer will be replaced. In case, there was no header and no footer yet, the new header or footer will just be added.

If you want to re-use the old header or footer, call takeHeaderFooter and addHeaderFooter, instead of using replaceHeaderFooter.

Note

`KDChart::Chart::takeHeaderFooter()`removes the header or footer from the chart without deleting it. The chart no longer owns the header or footer, it is the caller's responsibility to delete it.

The main configurations elements for `KDChart::HeaderFooter` are:

- Type: Either `KDChart::HeaderFooter::Header` or `KDChart::HeaderFooter::Footer`
- Position: Allow the user to define or retrieve the header or footer position using compass terminology.
- Text and text attributes can of course also be configured as we will see in the following examples.

Note

The `KDChart::Position` class defines positions using compass terminology. Using this class you can specify one of nine pre-defined, logical points in a similar way, as you would use a compass to navigate on a map. We recommend you consult its API Reference.

Headers and Footers code Sample

We will now describe those features more in depth by looking at the following sample code based on the `Simple Widget` example we have been demonstrating above in Chapter 3, *Basic steps: Create* *a Chart* - the section called “Widget Example”. Through the following code, we demonstrate how to add and position a header and a footer to a chart Widget using a `KDChart::Widget`.

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 `KDChartBarDiagram`. In order to be able to give a location (position) to our header and our footer 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 `KDChart::Widget::setType()` method in order to display a bar type diagram.

Now let us add our header and footer, set its position and its text.

```
    widget.addHeaderFooter( "A default Header - North",
                            HeaderFooter::Header, Position::North );
    widget.addHeaderFooter( "A default Footer - South",
                            HeaderFooter::Footer, Position::South );
```

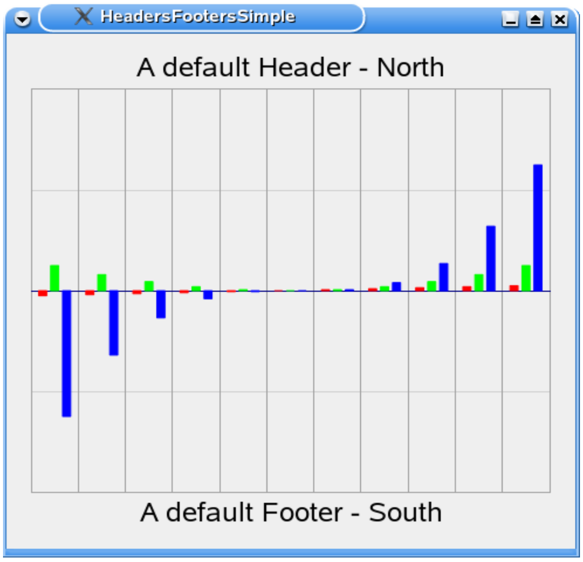
As you can see the code above is straight forward and we just need to call `KDChart::Widget::addHeaderFooter()` passing the text, type and position we want to assign to it.

Finally we conclude our small application:

```
    widget.show();

    return app.exec();
}
```

See the screenshot below to view The resulting chart displayed by the above code.

Figure 7.1. A Widget with a header and a footer



This example can be compiled and run from the following location of your KD Chart installation `examples/HeadersFooters/HeadersFootersSimple/`.

In the section called “Tips” below we will present you a more elaborate example which uses `KDChart::Chart` and where we are setting up our headers and footers (texts, background, frame 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 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 a configured header by looking at the following lines of code 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 <KDChartHeaderFooter>
#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 configure the header position as well as its text, background and frame attributes and include the header files related to those attributes on this purpose. That will allow us to make use of the methods available in these classes.

We will now add our header and set it up:

```
        // Add at one Header and set it up
        HeaderFooter* header = new HeaderFooter( &m_chart );
        header->setPosition( Position::North );
        header->setText( "A Simple Bar Chart" );
        m_chart.addHeaderFooter( header );
```

The code above handles the attributes specific to headers and footers. The setters and getters for the methods we have been used here are implemented in the `KDChart::HeaderFooter` class. We recommend you consult its API Reference.

Let us configure the header text attributes and make sure the font will be resized together with the widget in case the user resizes it.

```
        // Configure the Header text attributes
        TextAttributes hta( header->textAttributes() );
        hta.setPen( QPen(  Qt::blue ) );

        // let the header resize itself
        // together with the widget.
        // so-called relative size
        Measure m( 35.0 );
        m.setRelativeMode( header->autoReferenceArea(),
                           KDChartEnums::MeasureOrientationMinimum );
        hta.setFontSize( m );
        // min font size
        m.setValue( 3.0 );
        m.setCalculationMode( KDChartEnums::MeasureCalculationModeAbsolute );
        hta.setMinimalFontSize( m  );
        // assign
        header->setTextAttributes( hta );
```

Our header text is now displayed using a blue pen, the fonts are configured to take a relative size.

We also want to configure a white background to make it nicer, and proceed as follows:

```
        // Configure the header Background attributes
        BackgroundAttributes hba( header->backgroundAttributes() );
        hba.setBrush(  Qt::white );
        hba.setVisible( true );
        header->setBackgroundAttributes(  hba );
```

As for all types of attributes we just need to create the attribute object, configure it and assign it to our header.

The same process is applied to configure our header's frame attributes:

```
        // Configure the header Frame attributes
        FrameAttributes hfa( header->frameAttributes() );
        hfa.setPen( QPen ( QBrush( Qt::darkGray ), 2 ) );
        hfa.setVisible( true );
        header->setFrameAttributes(  hfa );
```

In the code above we assign a pen to the frame attributes in order to get a Gray line around the frame.

Note

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

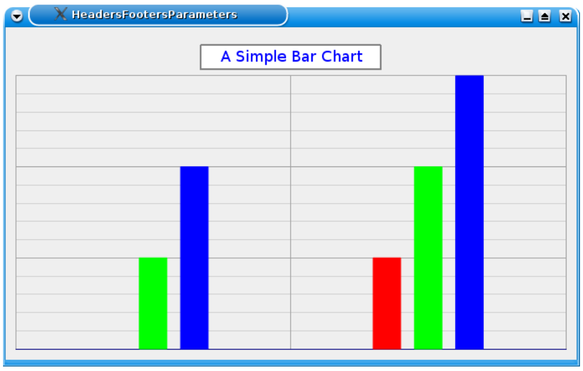
Finally, we conclude our small application.

```
        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 screenshot below to view the resulting chart displayed by the above code.

Figure 7.2. A Chart with a configured Header



We recommend you compile and run the above example. It is available at the following location: `examples/HeadersFooters/HeadersFootersParameters/`.

Headers and Footers Example

In the following implementation we want to be able to:

- Add, edit or remove headers and footers in/from our chart view.
- Configure their positions.
- Set their text
- All of the above operations should be available to the user from the GUI and performed dynamically.

In the example below we are using a `KDChart::Chart` class and also a home made `TableModel` for convenience. It is derived from `QAbstractTableModel`.

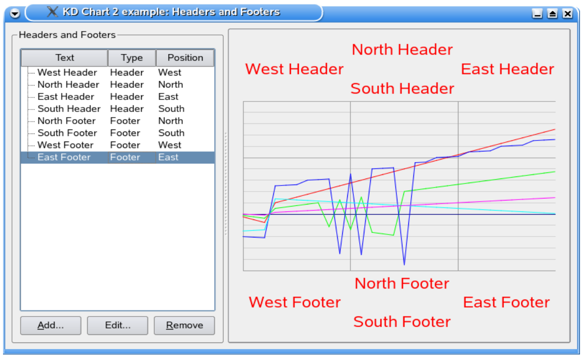
We recommend you consult the "TableModel" interface and implementation files which are located in the `examples/tools/` directory of your KD Chart installation.

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/HeadersFooters/Advanced/` directory of your installation.

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

See the screenshot below to view The resulting chart displayed by the above code.

Figure 7.3. Headers and Footers advanced example



Text	Type	Position
West Header	Header	West
North Header	Header	North
East Header	Header	East
South Header	Header	South
North Footer	Footer	North
South Footer	Footer	South
West Footer	Footer	West
East Footer	Footer	East

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

What's next

The next chapter will be dedicated to KD Chart's Attributes Model which is derived indirectly from `QAbstractProxyModel` and gives the user flexibility in customizing her chart and its component at different levels (whole diagram, per index, per row or column etc....).

Chapter 8. Customizing your Chart

Customizing your chart means configuring the attributes available for the different components of a chart (e.g diagrams, legends, headers and footers etc...). In Chapter 4, *Planes and Diagrams* we have been looking at the different attributes specific to a certain type of diagram (Line, Bar, Pie, etc...). In this chapter we will go through the details of the attributes related to the elements of a chart and also the ones common to all types of charts.

Attributes Model, Abstract Diagram

The `KDChart::AttributesModel` class is derived from `QAbstractProxyModel` and used internally by the base class for all diagrams `KDChart::AbstractDiagram` which `setAttributesModel( AttributesModel* model )` method associates an AttributesModel with a diagram.

Note

The diagram does _not_ take ownership of the AttributesModel. This should thus only be used with AttributesModels that have been explicitely created by the user. Setting an AttributesModel that is internal to another diagram will result in undefined behavior.

Let us illustrate the above assertion, the right way is:

```
// correct
AttributesModel *am = new AttributesModel( model, 0 );
diagram1->setAttributesModel( am );
diagram2->setAttributesModel( am );
```

It would be wrong to proceed as follow:

```
// Wrong
diagram1->setAttributesModel( diagram2->attributesModel() );
```

To retrieve the attribute model associated to a particular diagram, we can make use of the `KDChart::AbstractDiagram` method `attributesModel()`.

Note

By default each diagram owns its own AttributesModel, which should never be deleted. Only if a user-supplied AttributesModel has been set does the pointer returned here not belong to the diagram.

How it works

Let us make this more concrete by looking at the following methods for settings a Pen and extracted from `KDChart::AbstractDiagram`'s interface.

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

Note

`KDChart::AbstractDiagram` defines the interface, that needs to be implemented for the diagram to function within the KD Chart framework. It extends Qt's AbstractItemView.

Those methods allow us to set the Pen to be used respectively: at a given index, for a given dataset, or for all datasets in the model.

By looking at their implementations we can see how we make use of the `KDChart::AttributesModel` methods `setData()`, `setHeaderData()`, and `setModelData()` to achieve this task.

```
void AbstractDiagram::setPen( const QModelIndex& index, const QPen& pen )
{
    attributesModel()->setData(
    attributesModel()->mapFromSource( index ),
    qVariantFromValue( pen ), DatasetPenRole );
}

void AbstractDiagram::setPen( const QPen& pen )
{
    attributesModel()->setModelData(
    qVariantFromValue( pen ), DatasetPenRole );
}

void AbstractDiagram::setPen( int column,const QPen& pen )
{
    attributesModel()->setHeaderData(
    column, Qt::Vertical,
    qVariantFromValue( pen ),
    DatasetPenRole );
}
```

The above description to demonstrate how it works for almost all the attributes available for the configuranble elements of a chart, and the flexibility of this approch.

Note

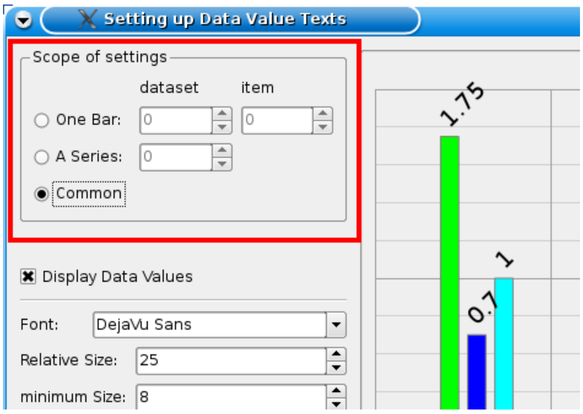
It is important to know that have three levels of precedence when setting the attributes:

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

Once you have set the attributes for a column or a cell, you can not 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 code above.

See the upper/left part of the screenshot below demonstrating a how the scope of some attribute settings might be selected:

Figure 8.1. Scope selection for Data Value Texts



○ One Bar: 0 0

○ A Series: 0

● Common

☒ Display Data Values

Font: DejaVu Sans

Relative Size: 25

minimum Size: 8

To see how this is done please have a look at the `examples/DataValueTexts/` example program.

In the next section we will have a quick look at the attributes common to all chart types and elements of a chart and learn about the way to use them.

Data Tooltips and Comments

As of version KD Chart 2.2 two roles are supported for specifying tooltips (ballon help) and/or fixed comment texts for any data item.

Specifying a data item tooltip

To have a tooltip shown for a data item, just set it at the respective cell, e.g. for a data model containing integer values you could do something like this:

```
const int row = 2;
const int column = 3;
const QModelIndex index = m_model.index(row, column, QModelIndex());
m_model.setData( index,
                 QString("<table><tr><td>Row</td><td>Column</td>"
                 "<td>Value</td></tr>"
                 "<tr><th>%1</th><th>%2</th><th>%3</th></tr></table>")
                     .arg( row )
                     .arg( column )
                     .arg( m_model.data( index ).toInt() ),
                 Qt::ToolTipRole );
```

This `setData()` method call is all you need, KD Chart and Qt will do the job for you: Once the mouse is resting over a data item (e.g. a bar) the tooltip will be shown for a while.

Specifying a fixed data item comment

To have a comment shown for a data item, just set it at the respective cell, e.g. for a data model containing integer values you could do something like this:

```
const int row = 0;
const int column = 2;
const QModelIndex index = m_model.index(row, column, QModelIndex());
m_model.setData( index,
                 QString("Value %1/%2: %3")
                     .arg( row )
                     .arg( column )
                     .arg( m_model.data( index ).toInt() ),
                 KDChart::CommentRole );
```

This `setData()` method call is all you need, KD Chart will then display a fixed comment next to the respective item (e.g. next to a bar).

Note

While tooltips may be both QML texts and normal texts, fixed comments as of yet can only be normal text. This might be changed in future versions of KD Chart depending on users' requests.

Data Values Attributes

The Data Value Attributes group all properties that can be set in relation to data value texts and if and how they are displayed. This includes things like the text attributes (font, color), what markers are used, and how many decimal digits are displayed, etc.

We recommend you consult `KDChart::DataValueAttributes`' interface to find out more in details what can be done. In this section we will describe quickly its main properties and go through a commented example that will demonstrates how to proceed in order to use and configure those attributes.

Data values can be set with some defined text, background, frame and markers. The list below gives us an overview about the most used features. We will only list the setters here and explain them. Of course each of those setters has a corresponding getter:

- setVisible(bool visible): Set whether data value texts should be displayed.
- setTextAttributes(const TextAttributes &a): Set the text attributes to use for the data value texts.
- setFrameAttributes(const FrameAttributes &a): Set the frame attributes to use for the data value text areas.
- setBackgroundAttributes(const BackgroundAttributes &a): Set the background attributes to use for the data value text areas.
- setMarkerAttributes(const MarkerAttributes &a): Set the marker attributes to use for the data values. This includes the marker type.
- void setDecimalDigits(int digits): Set how many decimal digits to use when rendering the data value texts.

The process to configure the data value attributes for a diagram is very simple, and similar to all other kind of attributes:

- Call the relevant attributes - e.g We want to configure the font and colors we need to configure the Text attributes and call them as follow: `TextAttributes ta( datavaluesattrinbutes.textAttributes() )`
- Assign the configurated attributes to your data values attributes. e.g call `datavalueattributes.setTextAttributes( ta )`.

- set them as visdible implicitly and assign them to the diagram by calling the diagram method `diagram->setDataValueAttributes()`

DataValue Attributes Sample code

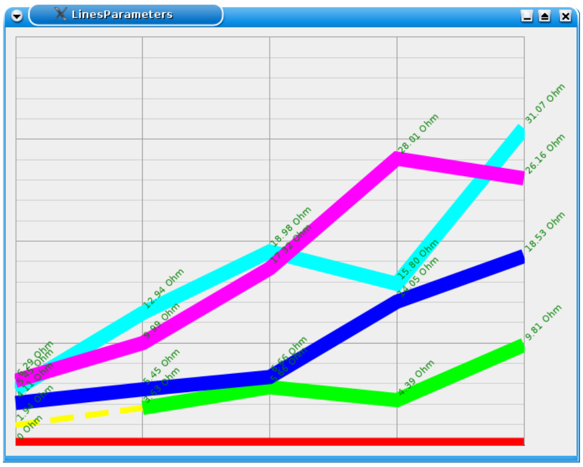
Let us make this more concrete by looking at the following lines of code which describe the above process. This example is based on the `main.cpp` file of the `examples/Lines/Parameters/` slightly modified. We recommend you compile and run this example and to study its code.

```
....
// Display values
// 1 - Call the relevant attributes
DataValueAttributes dva( diagram->dataValueAttributes() );

// 2 - We want to configure the font and colors
//     for the data value text.
TextAttributes ta( dva.textAttributes() );

// 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 );
dva.setDecimalDigits( 2 );
dva.setSuffix( " Ohm" );

// 5 - Assign to the diagram
diagram->setDataValueAttributes( dva );
        ....
// 6 - Assign the diagram to the chart
m_chart.coordinatePlane()->replaceDiagram(diagram);

// make sure there is space to display the
// data value texts at the edges of the data area
m_chart.setGlobalLeading( 15, 15, 15, 15 );
...
```

As we can see the code is straight forward and the process is similar as for setting all others types of attributes.

See the screenshot below to view The resulting chart displayed by the above code.

Figure 8.2. A Chart with configured Data Value Texts



We recommend you modifying, compiling and runing the example at the following location: `examples/Lines/Parameters/`.

Data Values Labels: Details

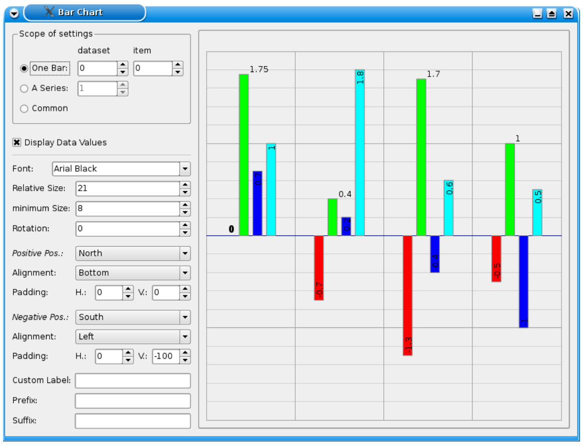
If you are interested in more details on positioning and/or customizing your data labels, have a look at the example `examples/DataValueTexts/`.

Note that all data value attributes can be configured on three different levels, in increasing hierarchy:

- Global settings to be used if no other settings have been specified.
- Dataset-specific settings to be used if no cell-specific settings have been specified.
- Cell-specific settings to be used for one single cell.

The "Scope" radio buttons and spin boxes of this example allow for selecting which data range the settings are to be applied to:

Figure 8.3. Positioning / adjusting Data Labels


Scope of settings
dataset item
● One Bar: 0 0
○ A Series: 1
○ Common

☒ Display Data Values

Font: Arial Black
Relative Size: 21
minimum Size: 8
Rotation: 0
Positive Pos.: North
Alignment: Bottom
Padding: H.: 0 V.: 0
Negative Pos.: South
Alignment: Left
Padding: H.: 0 V.: -100
Custom Label:
Prefix:
Suffix:

For information on how this is done please study the API Reference and also have a look at this file: `examples/DataValueTexts/`

Text Attributes

`TextAttributes` encapsulates settings that have to do with text. This includes font, font size, color, whether the text is rotated, etc...

We recommend studying the `KDChart::TextAttributes` API Reference to find out more in details what can be done. In this section we will describe quickly its main properties and go through a commented example that will demonstrate how to proceed in order to use and configure those attributes.

Text attributes can be set with some defined font, pen, rotation etc... The text font size can be fixed or relative (e.g it will adapt to the widget size), the list below gives us an overview of the most commonly used features. We will only list the setters here and explain them. Of course each of those setters has a corresponding getter:

- setVisible(bool visible): Set whether text attributes should be displayed.
- setFont(const QFont& font): Set the font to be used for rendering the text.
- void setFontSize(const Measure & measure): Set the size of the font used for rendering text
- setMinimalFontSize(const Measure & measure): Set the minimal size of the font used for rendering text.
- setRotation(int rotation): Set the rotation angle to use for the text.
- setPen(const QPen& pen): Set the pen to use for rendering the text.

The process to configure the text attributes any elements of a chart is very simple, and similar to all other kind of attributes:

- Call the text attributes - e.g We want to configure the font and colors we need to configure the Text attributes and call them as follow: `TextAttributes ta( header.textAttributes() )`
- Assign the configurated attributes to your header attributes. e.g call `header.setTextAttributes( ta )`.

Text Attributes Sample code

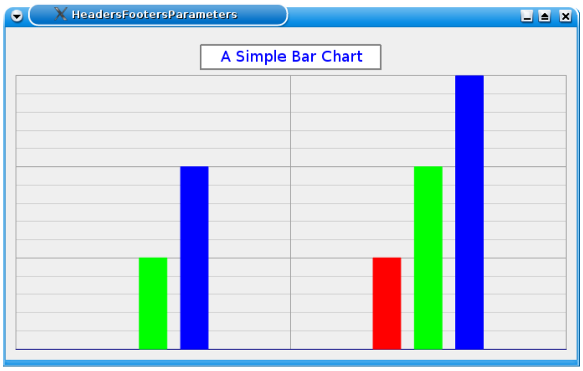
Let us now look at the following lines of code which describe the above process. This example is based on the `main.cpp` file of the `examples/HeadersFooters/HeadersFootersParameters/`. We recommend you compile and run this example and to study its code.

```
....
// Configure the Header text attributes
TextAttributes hta( header->textAttributes() );
hta.setPen( QPen(  Qt::blue ) );

// let the header resize itself
// together with the widget.
// so-called relative size
Measure m( 35.0 );
m.setRelativeMode( header->autoReferenceArea(),
KDChartEnums::MeasureOrientationMinimum );
hta.setFontSize( m );
// min font size
m.setValue( 3.0 );
m.setCalculationMode(
KDChartEnums::MeasureCalculationModeAbsolute );
hta.setMinimalFontSize( m  );

// Assign thre text attributes
// to our header.
header->setTextAttributes( hta );
...
```

As we can see the code is straight forward and the process is similar as with setting all others types of attributes.

See the screenshot below to view the resulting chart displayed by the above code.

Figure 8.4. A Chart with a configured Header



We recommend you to modify, compile and run the example at the following location: `examples/HeadersFooters/HeadersFootersParameters/`.

Markers Attributes

MarkerAttributes encapsulates settings that have to do with markers. This includes their types (square, diamond, ring etc...), size and colors. For convenience the user may also set up a map of markers.

We recommend you consult `KDChart::MarkerAttributes`' interface to find out more in detail what can be done. In this section we will describe quickly its main properties and go through a commented example that will demonstrates how to proceed in order to use and configure those attributes.

Marker attributes can be set with some defined type(s), size, color etc..., the list below gives us an overview about the most used features. We will only list the setters here and explain them - Of course each of those setters has a corresponding getter.

- setMarkerStyle(const MarkerStyle style): Set the style of the marker to be used.
- setMarkerSize(const QSizeF& size): Set the size of the marker.
- setMarkerColor(const QColor& color): Set the color of the marker.
- void setVisible(bool visible): Set whether marker attributes should be displayed.
- setMarkerStylesMap(MarkerStylesMap map): Define a map of marker to be used.

Note

As defined in the `KDChart::MarkersAttributes` class'interface the differnet marker types available are:

```
....
```

```
enum MarkerStyle { MarkerCircle  = 0,
                   MarkerSquare  = 1,
                   MarkerDiamond = 2,
                   Marker1Pixel  = 3,
                   Marker4Pixels = 4,
                   MarkerRing    = 5,
                   MarkerCross   = 6,
                   MarkerFastCross = 7 };
...
```

The process of configuring the marker attributes is very simple and similar to all other kind of attributes:

- Call the marker attributes - e.g We want to configure their types and sizes we need to configure the data values marker attributes and call them as follow: `MarkerAttributes ma( dva.markerAttributes() )`
- Assign the configurated attributes to your data values attributes. e.g call `dva.setMarkerAttributes( ma )`.

Markers Attributes Sample code

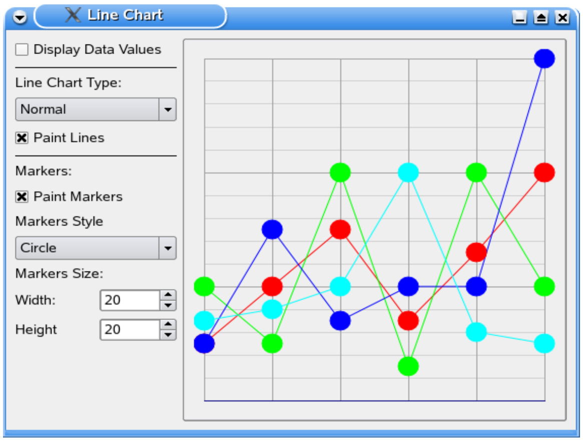
Let us make this more concrete by looking at the following lines of code which describe the above process. This example is based on the `mainwindow.cpp` file of the `examples/Axis/Parameters/`. We recommend you compile and run this example and to study its code.

```
....
// 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 );
....
// Configure markers per dataset in this example
const int colCount =
    m_lines->model()->columnCount(m_lines->rootIndex());
for ( int iColumn = 0; iColumn<colCount; ++iColumn ) {
    DataValueAttributes dva
        ( m_lines->dataValueAttributes( iColumn ) );
    MarkerAttributes ma( dva.markerAttributes() );
    ma.setMarkerStylesMap( map );
    ma.setMarkerSize( QSize( markersWidthSB->value(),
        markersHeightSB->value() ) );

    ma.setVisible(  true );

    // Assign markers attributes
    // to Data values attributes
    dva.setMarkerAttributes( ma );

    //Assign Data Values Attributes to
    //Diagram
    m_lines->setDataValueAttributes( iColumn, dva );
}
```

As we can see the code is straight forward and the process is similar as for setting all others types of attributes.

See the screenshot below to view the resulting chart displayed by the above code.

Figure 8.5. A Chart with configured Data Markers



☐ Display Data Values

Line Chart Type:
Normal

☒ Paint Lines

Markers:
☒ Paint Markers
Markers Style
Circle
Markers Size:
Width: 20
Height 20

We recommend you to modify, compile and run the example at the following location: See file: `examples/Axis/Parameters/mainwindow.cpp`.

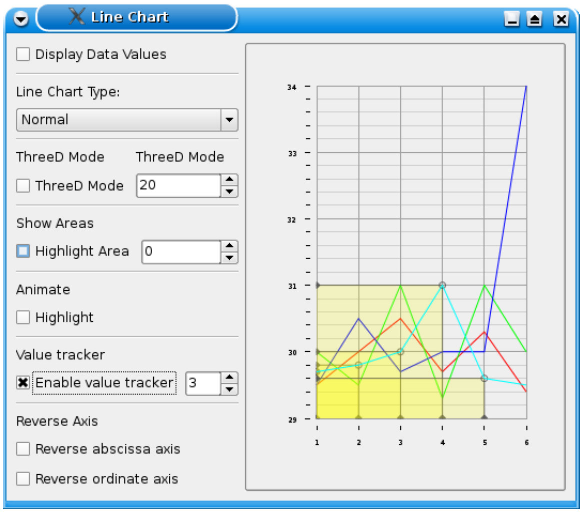
Value Tracker Attributes

Both, the `KDChart::LineDiagram` and the `KDChart::Plotter` class, provide access to `KDChart::ValueTrackerAttributes` allowing you to have extra lines drawn from a data point to one of the axes, and/or to fill the area between that line and the axis using a brush.

Please have a look at the `KDChart::ValueTrackerAttributes` interface for details on the respective setter methods.

Usage of value trackers is demonstrated in `examples/Lines/Advanced/mainwindow.cpp`, the following screenshot is taken from this example:

Figure 8.6. A Line Chart showing Value Trackers



☐ Display Data Values

Line Chart Type:
Normal

ThreeD Mode ThreeD Mode
☐ ThreeD Mode 20

Show Areas
☐ Highlight Area 0

Animate
☐ Highlight

Value tracker
☒ Enable value tracker 3

Reverse Axis
☐ Reverse abscissa axis
☐ Reverse ordinate axis

Note

As of yet, value tracker markers are just circles as shown in the screenshot and the end of the tracker lines are these small arrow heads, but to be configured via `KDChart::ValueTrackerAttributes::setMarkerSize()`. Additional setup options might be added to future versions of KD Chart depending on users' requests.

Background Attributes

Background attributes encapsulate settings that have to do with backgounds for the diverse elements of a chart view. This includes their modes (pixmap and its sub-modes and brush).

We recommend you consult `KDChart::BackgroundAttributes`'interface to find out more in details what can be done. In this section we will describe quickly its main properties and go through a commented example that will demonstrates how to proceed in order to use and configure those attributes.

The list below gives us an overview about the most used features. We will only list the setters here and explain them. Of course, each of those setters has a corresponding getter.

- setVisible(bool visible):
- setBrush(const QBrush &brush):
- setPixmapMode(BackgroundPixmapMode mode):
- setPixmap(const QPixmap &backPixmap):

Note

As defined in the `KDChart::BackgroundAttributes`' interface the different `BackgroundPixmapMode` available are:

```
....
enum BackgroundPixmapMode {
    BackgroundPixmapModeNone,
    BackgroundPixmapModeCentered,
    BackgroundPixmapModeScaled,
    BackgroundPixmapModeStretched
};
...
```

The process to configure the background attributes is very simple, and similar to all other kind of attributes:

- Call the background attributes and configure it.
- Assign the configurated attributes to the element of a chart. `element.setBackgroundAttributes( ba )`.

Background Attributes Sample code

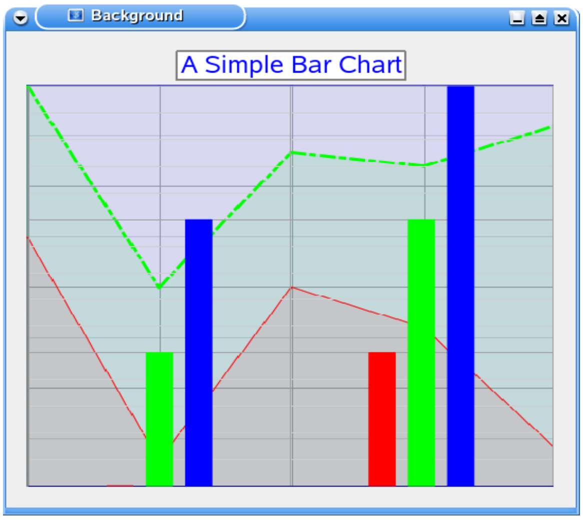
Let us make this more clear by looking at the following lines of code which describe the above process. This example is based on the `main.cpp` file of the `examples/Background/`. We recommend you compile and run this example and to study its code.

```
....
// Configure the plane's Background
BackgroundAttributes pba( diagram->coordinatePlane()->backgroundAttributes() );
pba.setPixmap(  *pixmap );
pba.setPixmapMode(
BackgroundAttributes::BackgroundPixmapModeStretched );
pba.setVisible( true );
diagram->coordinatePlane()->setBackgroundAttributes(  pba );

// Configure the Header's Background
BackgroundAttributes hba( header->backgroundAttributes() );
hba.setBrush( Qt::white );
hba.setVisible( true );
header->setBackgroundAttributes(  hba );
....
```

As we can see the code is straight forward and the process is similar as for setting all others types of attributes.

See the screenshot below to view the resulting chart displayed by the above code.

Figure 8.7. A simple Bar Chart with a Background Image



For details have a look at `examples/Background/`.

Frame Attributes

Frame attributes encapsulate settings that have to do with frames for the diverse elements of a chart view. This includes their pen and padding properties.

We recommend you consult `KDChart::FrameAttributes`' interface to find out more in details what can be done. In this section we will describe quickly its main properties and go through a commented example that will demonstrates how to proceed in order to use and configure those attributes.

The list below gives us an overview about the most used features. We will only list the setters here and explain them - Of course each of those setters has a corresponding getter.

- setVisible(bool visible):
- setPen(const QPen &pen):

- setPadding(int padding):

The process to configure the frame attributes is very simple, and similar to all other kind of attributes:

- Call the frame attributes and configure it.
- Assign the configurated attributes to the element of a chart: `element.setFrameAttributes( fa )`.

Frame Attributes Sample code

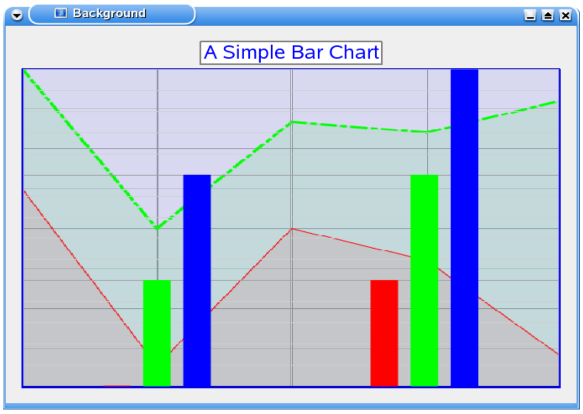
Let us make this more concrete by looking at the following lines of code which describes the above process. This example is based on the `main.cpp` file of the `examples/Background/`. We recommend you compile and run this example and to study its code.

```
....
// Configure the plane Frame attributes
FrameAttributes pfa( diagram->coordinatePlane()->frameAttributes() );
pfa.setPen( QPen ( QBrush( Qt::blue ), 2 ) );
pfa.setVisible( true );
diagram->coordinatePlane()->setFrameAttributes(  pfa );

// Configure the header Frame attributes
FrameAttributes hfa( header->frameAttributes() );
hfa.setPen( QPen ( QBrush( Qt::darkGray ), 2 ) );
hfa.setPadding( 2 );
hfa.setVisible( true );
header->setFrameAttributes(  hfa );
....
```

As we can see the code is straight forward and the process is similar as for setting all others types of attributes.

See the screenshot below to view the resulting chart displayed by the above code.

Figure 8.8. A Chart with configured Frame Attributes



We recommend you check out the example at the following location: See file: `examples/Background/`.

Grid Attributes

Grid attributes encapsulates settings that have to do with grids. This includes their pen, step width, visibility properties ...etc

We recommend you consult `KDChart::GridAttributes`' interface to find out more in details what can be done. In this section we will describe quickly its main properties and go through a commented example that demonstrates how to proceed in order to use and configure those attributes.

The list below gives us an overview about the most used features. We will only list the setters here and explain them. Of course, each of those setters has a corresponding getter.

- setGridVisible(bool visible): set whether the grid should be painted or not
- setGridStepWidth(qreal stepWidth=0.0): set the distance between the lines of the grid
- setGridPen(const QPen & pen): set the main grid pen.
- setSubGridVisible(bool visible): Specify whether the sub-grid should be displayed.
- setSubGridPen(const QPen & pen): set the sub-grid pen.
- setZeroLinePen(const QPen & pen): set the zero line pen.

The process to configure the grid attributes is very simple, and similar to all other kind of attributes:

- Call the grid attributes and configure it.
- Assign the configurated attributes to the plane using one of the setter available, e.g `CartesianCoordinatePlane::setGridAttributes ( Qt::Orientation orientation, const GridAttributes & )`. or `AbstractCoordinatePlane::setGlobalGridAttributes ( const GridAttributes & )`

Note

In case you want to set your grid attributes with orientation using the `CartesianCoordinatePlane` method above you will need to cast the result of `CartesianCoordinatePlane::coordinatePlane()` which returns a pointer to `AbstractCoordinatePlane` as shown in the following example.

Otherwise you just need to set the grid attributes globally as follow:

```
GridAttributes ga = diagram->coordinatePlane()->globalGridAttributes();
ga.setGlobalGridVisible( false );
diagram->coordinatePlane->setGlobalGridAttributes( ga );
```

Grid Attributes Sample code

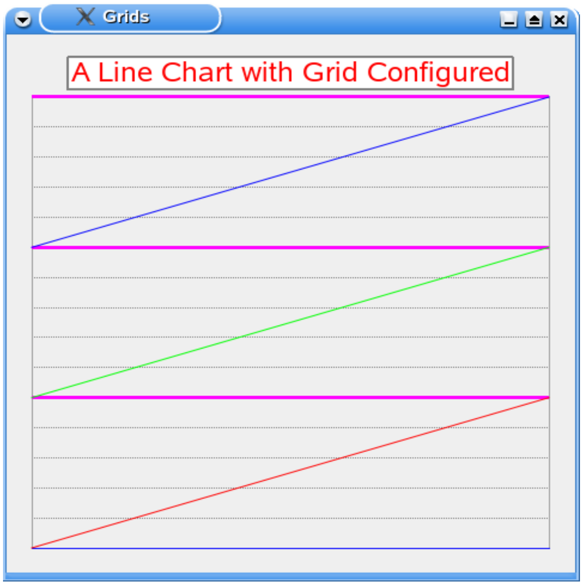
The following lines of code will show how to use grid atttributes. This example is based on the `main.cpp` file of the `examples/Grids/CartesianGrid/`. We recommend you compile and run this example and to study its code.

```
// diagram->coordinatePlane returns an abstract plane.
// if we want to specify the orientation we need to cast
// as follow
CartesianCoordinatePlane* plane =
    static_cast <CartesianCoordinatePlane*>
        ( diagram->coordinatePlane() );

// retrieve your grid attributes
// display grid and sub-grid
GridAttributes ga ( plane->gridAttributes( Qt::Vertical ) );
ga.setGridVisible(  true );
ga.setSubGridVisible( true );

// Configure a grid pen
QPen gridPen(  Qt::magenta );
gridPen.setWidth( 3 );
ga.setGridPen(  gridPen );

// Configure a sub-grid pen
QPen subGridPen( Qt::darkGray );
subGridPen.setStyle( Qt::DotLine );
ga.setSubGridPen(  subGridPen );

// Display a blue zero line
ga.setZeroLinePen( QPen( Qt::blue ) );

// Assign your grid to the plane
plane->setGridAttributes( Qt::Vertical,  ga );
```

As we can see the code is straight forward and the process is similar as for setting all others types of attributes.

See the screenshot below to view the resulting chart displayed by the above code.

Figure 8.9. A Chart with configured Grid Attributes



We recommend you modify, compile and run the example at the following location. See file: `examples/Grids/CartesianGrid/`.

ThreeD Attributes

ThreeDAttributes properties are defined at different levels in the KD Chart 2 API. We have the properties available to all types of diagram which are defined in the `KDChart::AbstractThreeDAttributes` and the ones specific to a type of diagram. At the moment we support ThreeD for Bar, Lines and Pie diagrams and the ThreeD attributes for those diagrams types are defined in their own attributes classes. We have `KDChart::ThreeDBarAttributes`, `KDChart::ThreeDLineAttributes` and `KDChart::ThreeDPieAttributes`

ThreeD attributes encapsulates settings that have to do with 3D display. This includes their depth, angle, rotation etc ... depending of the chart type we are working with.

We recommend you consult the `KDChart::ThreeDAttributes`' interface to find out more in details what can be done. In this section we will describe quickly its main properties and go through a commented example that demonstrates how to proceed in order to use and configure those attributes.

The list below gives us an overview about the most commonly used features. We will only list the setters here and explain them - Of course each of those setters has a corresponding getter.

1 - Generic (common to all diagrams) ThreeD Attributes

- setEnabled(bool enabled): set whether threeD display mode is on or off.

- setDepth(double depth): set the depth of the threeD effect (see example below).

2 - ThreeD Bar Attributes - Specific to bar diagrams.

- setAngle(uint threeDAngle): Not implemented yet

3 - ThreeD Line Attributes - Specific to line diagrams.

- setLineXRotation(const uint degrees): rotate the x coordinate.

- setLineYRotation(const uint degrees): rotate the y coordinate.

4 - ThreeD Pie Attributes - Specific to Pie diagrams.

- setUseShadowColors(bool useShadowColors): Not implemented yet

The process to configure the grid attributes is very simple, and similar to all other kind of attributes:

- Call the 3D attributes and configure it.

- Assign the configurated attributes to the diagram by calling the available method `setThreeDAttributes()` method.

ThreeD Attributes Example

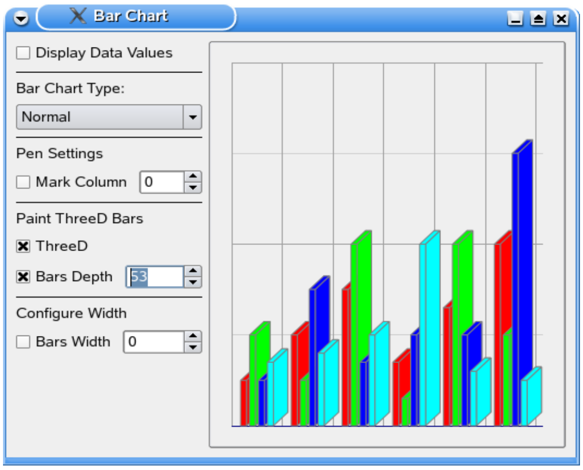
Let us make this more concrete by looking at the following lines of code which describe the above process. This example is based on the `mainwindow.cpp` file of the `examples/Bars/Advanced/`. We recommend you compile and run this example and to study its code.

```
ThreeDBarAttributes td( m_bars->threeDBarAttributes() );

td.setDepth( depthSB->value() );
td.setEnabled( true );

// Assign to the diagram
m_bars->setThreeDBarAttributes( td );
```

As we can see the code is straight forward and the process is similar as for setting all others types of attributes.

See the screenshot below to view the resulting chart displayed by the above code.

Figure 8.10. A Three-D Bar Chart



☐ Display Data Values

Bar Chart Type: Normal

Pen Settings

☐ Mark Column 0

Paint ThreeD Bars

☒ ThreeD

☒ Bars Depth 53

Configure Width

☐ Bars Width 0

We recommend you modify, compile and run the example at the following location: See file: `examples/Bars/Advanced/`.

Font Sizes and other Measures

This chapter illustrates how to use the `KDChart::Measure` class to specify sizes. Closely related to `Measure` is the `KDChart::RelativePosition` class explained in the section called "Relative and Absolute Positions" following this one.

When and how to use the Measure class

`KDChart::Measure` is used to specify absolute values or relative measures to be re-calculated at runtime according to the size of a reference area, e.g. for font sizes or to define the distance between a text and its anchor point.

- Absolute values are used to set a fixed measure, e.g. when the same font size is to be used, no matter how large the chart widget is displayed.
- Relative measures specify values that are multiplied by 1/1000 of their reference area's width (or height, resp.) at runtime. KD Chart uses this to link the default legend fonts to the chart's size: The legend is adjusted when your widget is resized.

Tip

The `KDChart::TextAttributes` class can handle both kinds of measures at the same time: You often might wish to specify a relative size via `setFontSize()` and set a fixed value via `setMinimalFontSize()` so the font will be dynamically calculated according to the area size but it will never be smaller than that specific minimum.

Being a typical value class `Measure` is commonly initialized by the copy constructor since you should modify KD Chart's pre-defined settings rather than defining new ones from scratch. File `examples/Lines/Parameters/main.cpp` shows how to do that:

```
// Retrieve the data value attrs from your diagram, and retrieve their text att
DataValueAttributes dva( diagram->dataValueAttributes() );
TextAttributes ta( dva.textAttributes() );

// Retrieve the font size and increase its value
Measure me( ta.fontSize() );
me.setValue( me.value() * 1.25 );
```

```
// Make the data value texts visible
ta.setVisible(  true );
dva.setVisible( true );

// Set the font size, set the text attrs, set the data value attrs
ta.setFontSize( me );
dva.setTextAttributes( ta );
diagram->setDataValueAttributes( dva );
```

How to specify absolute values

To specify an absolute value for a Measure that you have initialized via copy constructor please use the `setAbsoluteValue()` method:

```
Measure me( someTextAttributes.fontSize() );
me.setAbsoluteValue( 16 );
someTextAttributes.setFontSize( me );
```

If you want to declare a new Measure from scratch just set the first two constructor parameters:

```
Measure me( 16, KDChartEnums::MeasureCalculationModeAbsolute );
```

In this case you can ommit the third parameter, since the orientation setting is ignored for absolute values.

How to specify relative values

To specify a relative value for a Measure (no matter if initialized via copy constructor or not) you can use `setValue()` together with either `setRelativeMode()` or both `setReferenceArea()` and/or `setReferenceOrientation()`. So if your measure was using a fixed font size before you could say:

```
me.setValue( 25 );
me.setRelativeMode( m_chart, KDChartEnums::MeasureOrientationMinimum );
```

Note that `setRelativeMode()` is a convenience method that will implicitely enable the relative calculation mode.

When not using `setRelativeMode()` you need to explicitely call `setCalculationMode( KDChartEnums::MeasureCalculationModeRelative )`, if your Measure was not set to this mode before:

```
me.setValue( 25 );
me.setReferenceArea( m_chart );
me.setReferenceOrientation( KDChartEnums::MeasureOrientationMinimum );
me.setCalculationMode(      KDChartEnums::MeasureCalculationModeRelative );
```

In both cases the reference area must be derived from `KDChart::AbstractArea` or derived from `QWidget`. The orientation can be Horizontal, Vertical, Minimum, Maximum, the later ones meaning the area's qMin(width, height) or its qMax(), resp.

Relative and Absolute Positions

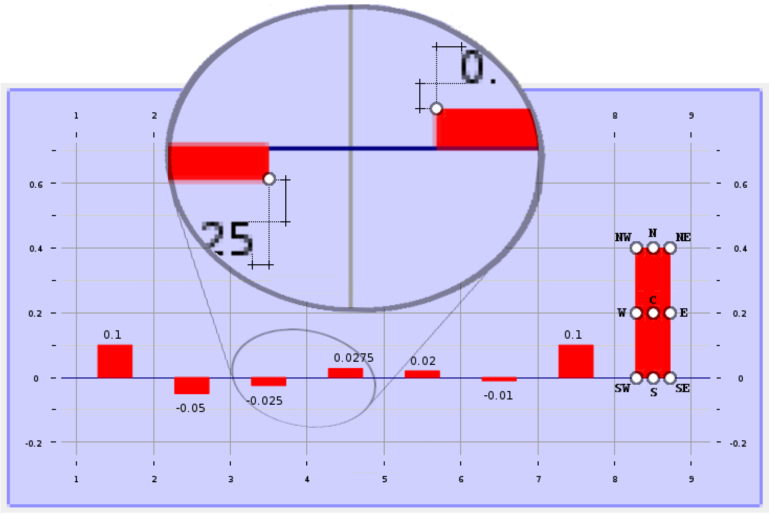
This chapter covers the `KDChart::Position` and `KDChart::RelativePosition` classes. For details on the closely related `KDChart::Measure` class see the preceeding the section called “Font Sizes and other Measures”.

What is relative positioning all about?

Introduced for floating objects in KD Chart 2.0, relative positioning is defining a point in relation to a reference point, that in turn is specified in relation to a reference area.

This illustration shows the nine position points defined for a bar. See the magnified area for the relative positioning of negative / positive data value texts.

Figure 8.11. Data value text positions relative to compass points



How to specify a position

1. If necessary name a reference area or define a set of reference points.
2. Use `KDChart::Position` to pick one of the reference area's compass points.
3. Specify padding and alignment in horizontal and vertical direction.

Using Position and RelativePosition

Illustrated on the preceeding page you have seen the most common use of these position classes: Defining the placement of data value texts in relation to their respective areas.

By default positive and negative data value texts are positioned in different ways: While positive texts would use the bar's `Position::NorthWest` their negative counterparts are located next to the `Position::SouthEast` point of the bar. Also the positive texts are using another way of alignment than the negative ones.

The reason for this is to make it easy to specify rotated data value texts: Because of different reference points and alignment, the texts will look good even when rotated without the need of adjusting other settings than just the rotation angle itself.

Being a typical value class `RelativePosition` is commonly initialized by the copy constructor since you should modify KD Chart's pre-defined settings rather than defining new ones from scratch, so you could specify non-rotated, centered texts as shown in the following code, that is using extra indentation to indicate get/set relationship:

```
// Retrieve the data value attrs from your diagram
DataValueAttributes dva( diagram->dataValueAttributes() );

    // Set the text rotation to Zero degrees
    TextAttributes ta = dva.textAttributes();
        ta.setRotation( 0 );
    dva.setTextAttributes( ta );

    // Retrieve the current position settings
    RelativePosition posPositive( dva.position( true ) );
    RelativePosition posNegative( dva.position( false ) );

        // Choose the centered position points
        posPositive.setReferencePosition( Position::North );
        posNegative.setReferencePosition( Position::South );

        // Adjust the alignment of the texts:
        // horizontally centered to their respective position points
        posPositive.setAlignment( Qt::AlignHCenter | Qt::AlignBottom );
        posNegative.setAlignment( Qt::AlignHCenter | Qt::AlignTop );

    // Set the positions
    dva.setPositivePosition( posPositive );
    dva.setNegativePosition( posNegative );

    // Make the data value texts visible
    dva.setVisible( true );

// Set the data value attrs
diagram->setDataValueAttributes( dva );
```

What's next

Advanced charting.

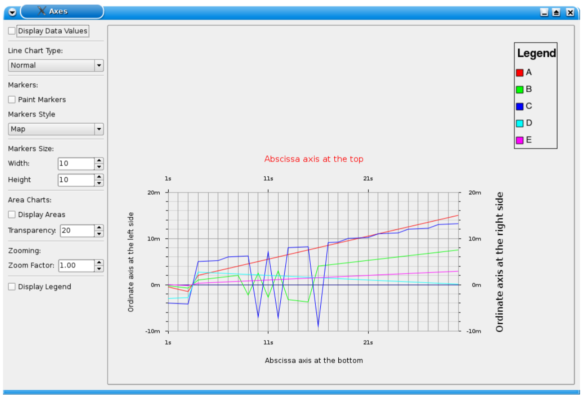
Chapter 9. Advanced Charting

In this section we are presenting some examples to demonstrate interesting features offered by the KD Chart 2 API by displaying the resulting widget and giving you a link to the directory in which you can study the example code, compile and run it.

Example programs to consult

1 - `/examples/Axis/Parameters`

Figure 9.1. /examples/Axis/Parameters



☐ Display Data Values

Line Chart Type: Normal

Markers:

☐ Paint Markers

Markers Style: Map

Markers Size:

Width: 10

Height: 10

Area Charts:

☐ Display Areas

Transparency: 20

Zooming:

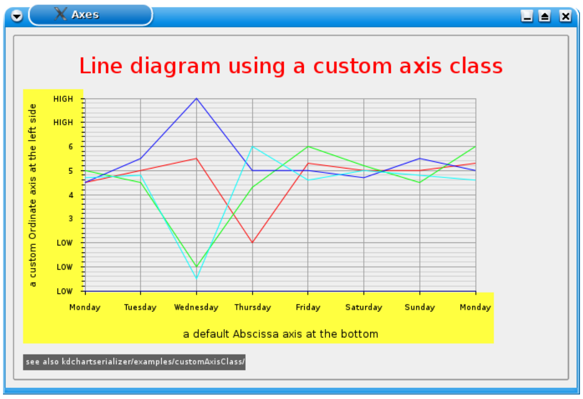
Zoom Factor: 1.00

☐ Display Legend

Axes
Legend
🟥 A
🟩 B
🟦 C
D
E
Abscissa axis at the top
1s
11s
21s
20m
10m
0m
-10m
Ordinate axis at the left side
Ordinate axis at the right side
Abscissa axis at the bottom

2 - `/examples/Axis/Labels`

Figure 9.2. /examples/Axis/Labels

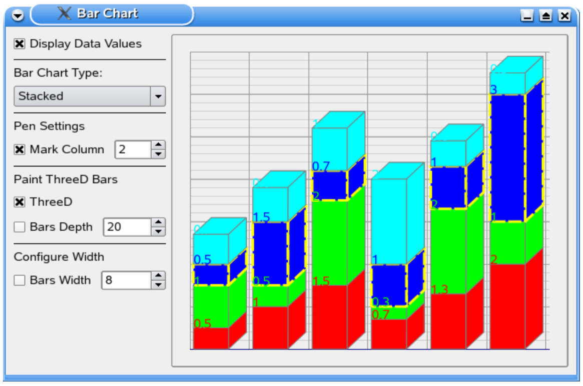


Axes
Line diagram using a custom axis class
HIGH
HIGH
6
5
4
3
LOW
LOW
LOW
a custom Ordinate axis at the left side
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday
Monday
a default Abscissa axis at the bottom

see also kdchartserializer/examples/customAxisClass/

3 - `/examples/Bars/Advanced`

Figure 9.3. /examples/Bars/Advanced



☒ Display Data Values

Bar Chart Type: Stacked

Pen Settings

☒ Mark Column 2

Paint ThreeD Bars

☒ ThreeD

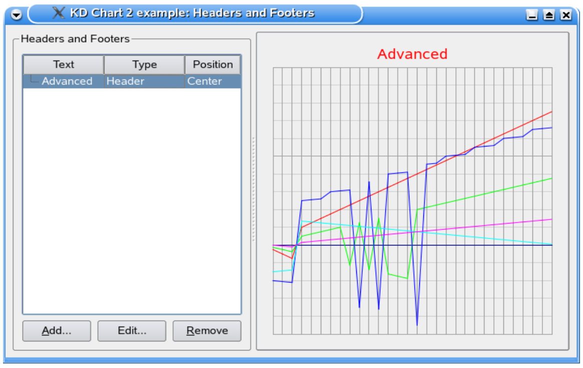
☐ Bars Depth 20

Configure Width

☐ Bars Width 8

`4 - /examples/HeadersFooters/HeadersFooters/Advanced`

Figure 9.4. /examples/HeadersFooters/HeadersFooters/Advanced

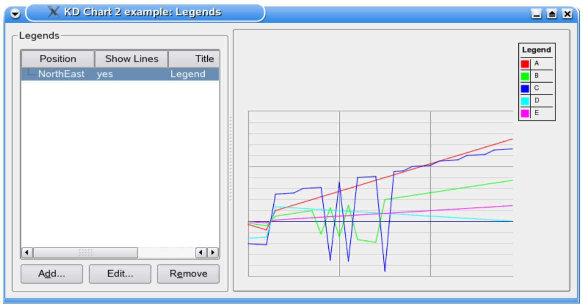


Headers and Footers

Text	Type	Position
Advanced	Header	Center

`5 - /examples/Legends/LegendAdvanced`

Figure 9.5. /examples/Legends/LegendAdvanced

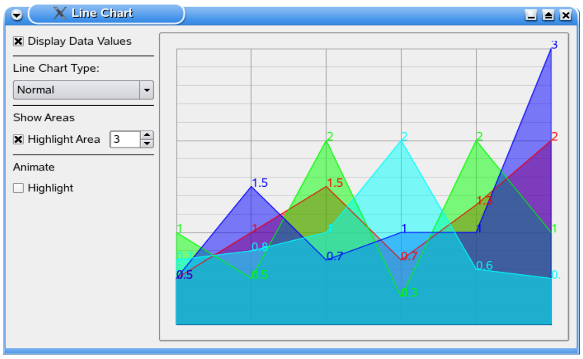


Legends

Position	Show Lines	Title
NorthEast	yes	Legend

`6 - /examples/Lines/Advanced`

Figure 9.6. /examples/Lines/Advanced



☒ Display Data Values

Line Chart Type: Normal

Show Areas

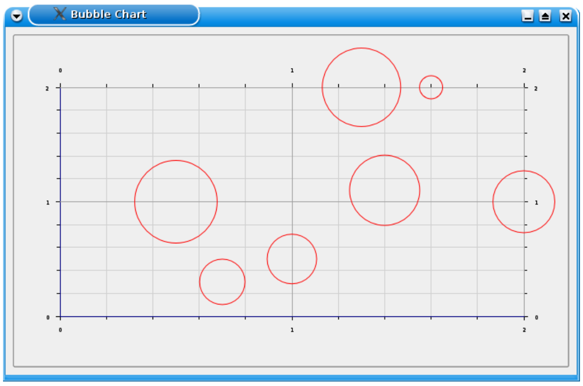
☒ Highlight Area 3

Animate

☐ Highlight

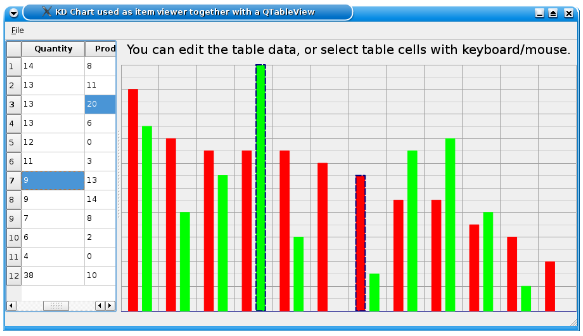
`7 - /examples/Plotter/BubbleChart`

Figure 9.7. /examples/Plotter/BubbleChart



`8 - /examples/ModelView/TableView`

Figure 9.8. /examples/ModelView/TableView

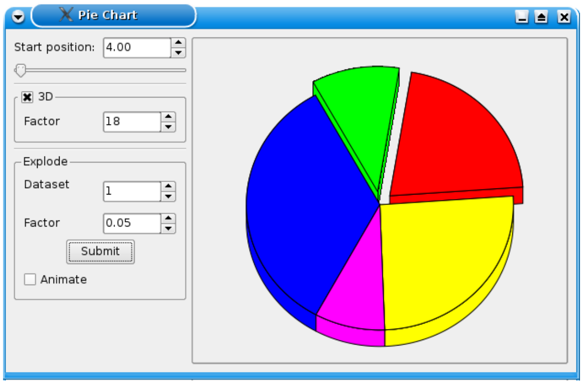


	Quantity	Prod
1	14	8
2	13	11
3	13	20
4	13	6
5	12	0
6	11	3
7	9	13
8	9	14
9	7	8
10	6	2
11	4	0
12	38	10

You can edit the table data, or select table cells with keyboard/mouse.

`9 - /examples/Pie/Advanced`

Figure 9.9. /examples/Pie/Advanced



Start position: 4.00

☒ 3D

Factor 18

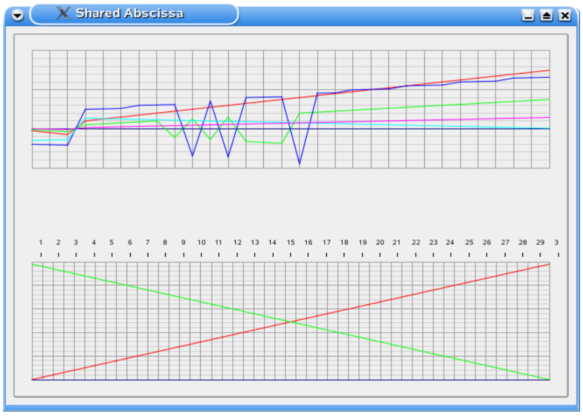
Dataset 1

Factor 0.05

☐ Animate

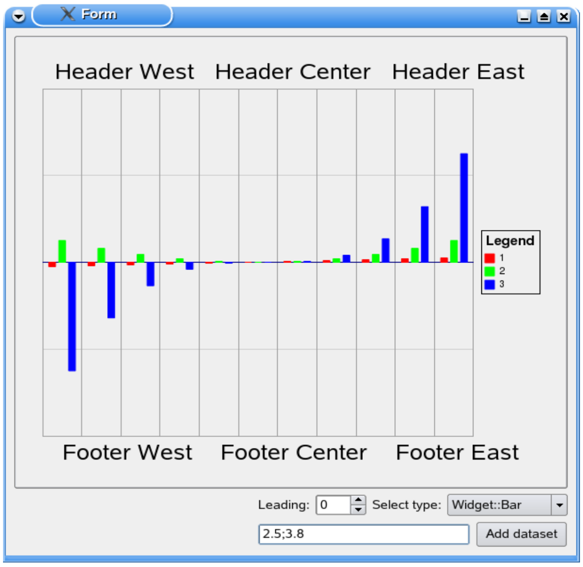
10 - `/examples/SharedAbscissa`

Figure 9.10. /examples/SharedAbscissa



11 - `/examples/Widget/Advanced`

Figure 9.11. /examples/Widget/Advanced



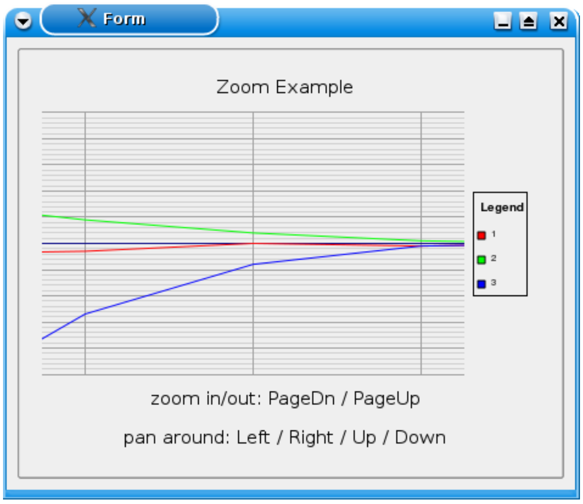
Leading: 0

Select type: Widget::Bar

2.5;3.8

12 - `/examples/Zoom/Keyboard`

Figure 9.12. /examples/Zoom/Keyboard

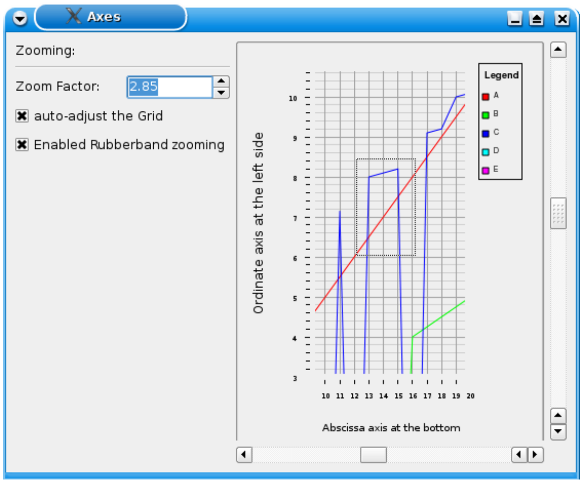


zoom in/out: PageDn / PageUp

pan around: Left / Right / Up / Down

13 - `/examples/Zoom/ScrollBars`

Figure 9.13. /examples/Zoom/ScrollBars



Zooming:

Zoom Factor: 2.85

☒ auto-adjust the Grid

☒ Enabled Rubberband zooming

Appendix A. Q&A section

A.1. Building and installing KD Chart

A.1.1. How can I build and install KD Chart from source?

Procedure to follow for building and installing KD Chart is described in file `Install.src`, please refer to that file for details.

A.1.2. How can I install the Designer Plug-in?

This can be done either manually or automatically:

- manual installation:

  ### Note

  This step is only needed if you did not install KD Chart top-level, as described in the previous answer:

  Go to the `plugins` directory of your KD Chart source installation. Run `make install` (Unix/Linux, Mac, ...), or `nmake install` (Windows)

  Now find the Plug-in file in the `lib/plugin/` directory of your KD Chart installation path: For Unix/Linux, Mac: `/usr/local/KDAB/KDChart-VERSION/lib/plugin` For Windows that is: `C:\KDAB\KDChart-VERSION\lib\plugin\` From there you can either copy it into your desired QT's plugin path, e.g. this might be `$QTDIR/plugins/designer/`, or you can set the `QT_PLUGIN_PATH` environment variable before running the designer. If set, Qt will look for plugins in the paths (separated by the system path separator) specified in the variable.

- automatic installation: This will copy the Plug-in into the QT plugin path, e.g. this might be `$QTDIR/plugins/designer/`

  Go to the `plugins` directory of your KD Chart source installation Run `make distclean` (Unix/Linux, Mac, ...), or `nmake distclean` (if using Windows) Run `qmake CONFIG +=install-qt` Run `make install` (Unix/Linux, Mac, ...), or `nmake install` (Windows)

A.2. User interaction

A.2.1. How can I connect a diagram to a `QTableView`?

As KD Chart 2 fully supports the "Interview" model/view paradigm introduced by Qt 4 connecting a diagram to a `QTableView` is as easy as using a `QItemSelectionModel`.

Have a look at the file `examples/ModelView/TableView/mainwindow.cpp` to see how this is done in the `MainWindow::setupViews()` method and/or study the Qt API Reference documentation.

A.2.2. How can I run my own code on mouse click at diagram data?

As KD Chart 2 fully supports the "Interview" model/view paradigm introduced by Qt 4 having your own `Slot` method invoked on mouse click can be achieved by using a `QItemSelectionModel` and connecting to its `selectionChanged()` signal.

Have a look at the file `examples/ModelView/TableView/mainwindow.cpp` to see how the connection is declared in the constructor. Using information in the signal's `QItemSelection` parameters any (de)selected bars are (un)marked in the `MainWindow::selectionChanged()` method.

Of course you could also show a dialog there to display additional data to the user, or you might want to fill some `QLabel` with information on items clicked ...

A.2.3. How can I let the user zoom at diagram data by rubberbanding?

As rubberbanding is explicitely supported by the `KDChart::AbstractCoordinatePlane` class you can just call its method `setRubberBandZoomingEnabled( bool )`. The plane will transparently use a `QRubberBand` in its `mousePressEvent()` for the left button and it will adjust its zoom factor setting automatically too, as well as call its parent's `update()` method.

Have a look at the file `examples/Zoom/ScrollBars/mainwindow.cpp` making use of this feature.

A.3. Storing / loading of KD Chart settings

A.3.1. How can I store KD Chart settings to a file?

This can be done by using the `KDChart::Serializer` class.

Note that `KDChart::Serializer` is dependent on your Qt library containing the `QtXml` module which provides C++ implementations of `SAX` and `DOM`, so having the serializer in a library of its own allows you to build KD Chart even if your version of Qt does not include the XML module.

To build the serializer library, just run

```
cd kdchartserializer
qmake
make   (or nmake, for Windows, resp.)
```

The examples in `kdchartserializer/examples/` show how to use the serializer and how to connect your diagram(s) to the data model(s) after the serializer has finished loading the settings.

A.4. Dynamic data / Look and Feel

A.4.1. How can I change (or add, resp.) data of an existing chart?

As KD Chart 2 fully supports the "Interview" model/view framework introduced by Qt 4 modifying your data model is automatically reflected by your views, i.e. your LineDiagram is updated and axes re-calculated if necessary.

Have a look at `examples/RealTime/main.cpp` where this is done by the `slotTimeout()` method just calling `m_model.setData()`. Of course you could also use the `insertRows()` method of the model to add new data cells, or you could remove some data ... For details see the API Reference of the Qt `QStandardItemModel` class.

A.4.2. How do I use fixed bar width so the chart gets wider when data are added?

The new method `setFixedDataCoordinateSpaceRelation( bool )` provided by the `KDChart::CartesianAxis` class can be used to lock the currently active bar width, it disables the default width adjusting so you can no longer expect all of the data to fit into the available space.

Adding more data will then keep the same bar width: The coordinate plane will grow wider, so you might consider embedding your `KDChart::Chart` (or your `KDChart::Widget`, resp.) in a `QScrollArea` to make sure all of it will fit into your application's window without that growing too large.

A.4.3. How can I make the axes area look like a contiguous region?

By using the same `QBrush` with `setBackgroundAttributes()` of both of the axes you make KD Chart show their areas as one region: An additional rectangular area will be inserted in the axes' corner to make the axes form an 'L' shaped region, as shown in `examples/Axis/Labels/`.

A.5. Contacting KD Chart Support

A.5.1. How can I get help (or report issues, resp.) on KD Chart?

To report issues/problems, or ask for help on KD Chart please send your mail with a description of your problem/question/wishes to the support address given to you with your license. Please include a description of your setup: CPU type, operating system with release number, compiler (version) used, any changes you made on libraries that are linked to ... Just include every detail that might help us set up a comparable test environment in our labs.

In most cases it will make sense to include a small sample program showing the problem you are describing: We will then reproduce the issue on our machines and either fix your sample code or adjust our own code (in case your reported issue might turn out to result from sub-optimal implementation in KD Chart).

Note

Providing us with a compilable sample program file will help us find a good solution for the problem reported, as we will be using the same code that you have been trying to use yourself.

Often the easiest way to create such a sample program could be to modify one of our programs and send the source file, e.g. if you have modified `examples/Bars/Simple/main.cpp` to show what you are trying to achieve you can just send us the `main.cpp` file and state that it is a to be used in `examples/Bars/Simple/`.