Programmers Manual

KD Chart

KDAB
KLARÄLVDALENS DATAKONSULT AB

Table of Contents

List of Figures

List of Tables

Chapter 1. Introduction

Welcome to the KD Chart Programmer's Manual. KD Chart is Klarälvdalens Datakonsult AB's charting package for Qt applications. This manual will get you started creating your own charts. It covers the fundamentals of coding with KD Chart and provides plenty of tips for advanced programmers.

- Depending on your version of KD Chart, you will find a unique `INSTALL` file containing instructions on how to install KD Chart on your platform. Each instruction set also includes step-by-step description of how to build KD Chart directly from the source code.
- KD Chart also comes with an extensive "API Reference" Manual (generated from the source code itself). It is available both as a PDF file and as browsable HTML pages.

The "API Reference" is an excellent resource for topics not covered in the Programmer's Manual. Both the Programmer's Manual and API Reference are designed to be used in conjunction with each other. If you have a question not covered in the following chapters, check the API reference for a solution (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. It is the most powerful Qt component of its kind. In addition to all Qt's standard features, it provides developers with a sophisticated way to customize layouts and to design and manage large numbers of axes. Since all configuration settings have practible defaults, you can usually get by with tweaking just a few parameters and then 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. For example, one application uses KD Chart to visualize flood events in a river. Another uses KD Chart for monitoring seismic activity. The current version of the KOffice productivity suite also uses our library. For other examples, visit our web site at `http://www.kdab.com/kdchart/2.4/`

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/KD-`

`Chart-2.0-API-Introduction`. Also to browse KD Chart API Reference start with the file `doc/refman/index.html` or `http://docs.kdab.com/kdchart/`.

The Structure of This Manual

Where do we start?

This manual begins with an introduction to the KD Chart 2 API then goes through the basic steps and methods for the user to create her own chart.

Chapter 4, *Planes and Diagrams* will provide the reader with details about the different chart types supported and more information about how to make the most out of KD Chart.

Each subsequent chapter covers more advanced material for customizing charts, such as: how to specify colors, fonts and other attributes. If you prefer not to use KD Chart's default settings, These chapters will go over topics like: how to create and display headers, footers and legends, as well as, how to configure your chart axes.

Chapter 9, `Advanced Charting`, presents more of KD Chart's advanced features and shows screenshots of example programs. We demonstrate how set up frames, backgrounds, data value texts, axis and grids etc... Additionally, it covers features like Interactive and Multiple charts and Zooming.

We provide you with many more example programs than shown in this manual. We recommend that our readers try them out 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

Version 2.4 of KD Chart builds on technologies introduced with Qt 4. The charting engine uses the Arthur (painting) and Scribe (text rendering) frameworks to achieve high quality visual results. KD Chart 2 also integrates the Interview framework for model/view separation and, much like Qt 4 itself, it provides a convenient Widget class for simple-use cases.

Overview

The core of KD Chart 2 API is the `KDChart::Chart` class. It creates the canvas onto which the individual components of a chart are painted. It manages them and it 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` contains subclasses for various types of charts, such as `KDChart::PieDiagram`. 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. Implementing a simple Bar Diagram looks like this:

Code Sample

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

The code example may seem abstract. In Chapter 3, *Basic steps: Create a Chart*, we will look at some complete examples of Widget and Charts to clarify how to implement the chart classes.

Concepts

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

- Each diagram drawn by KD Chart has an associated Coordinate Plane (Cartesian by default). Every calculation necessary for drawing each diagram is done by the coordinate plane. The Coordinate Plane translates data values into pixel information. This makes implementing diagram subclasses (types) much easier, since the Coordinate Plane defines the scale of the diagram and all axes associated with it.

- Each coordinate plane can have one or more diagrams linked to it. In which case,

those diagrams will share the scale provided by the master plane. Also, a chart may contain more than one coordinate plane. This makes it possible to have multiple diagrams (e.g. a line and a bar chart), using the different scales, within the same master chart. By using a combination of these methods, diagrams can be displayed, at varying scales, next to, or on top of each other in the final drawing output by KD Chart.

- A Coordinate Plane can share an axis with a second Coordinate plane. By linking the axis of the second Coordinate plane to an axis of the first, the first owns the second. By this method you can use the KD Chart engine to adjust the position and scale of the second diagram proportionately to the first.

  This code is taken from `mainwindow.cpp` in `examples/SharedAbscissa/SeparateDiagrams/`, Here we use two data models, two coordinate planes, two diagrams, and 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. Then we explicitly add the axis to the second diagram so that it is shared between both of them.

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

For example, to add additional headers, 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 explain how ownership of such components is maintained.

All classes in the KD Chart 2 API are included in the `KDChart` namespace. This allows concise class names while avoiding name clashes. Instead of using the `KDChart::` prefix with every class name in your code, add the `KDChart` namespace to the beginning of your implementation file:

```
using namespace KDChart;
```

Like Qt, KD Chart provides STL-style forwarding headers, allowing you to omit the `.h` suffux. To include the bar diagram header in your implementation file, write:

```
#include <KDChartBarDiagram>
or, if you prefer:
#include <KDChartBarDiagram.h>
```

Note

Header and implementation files all have the `KDChart` prefix in the name. For example, the definition of `KDChart::BarDiagram` is 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 Section , “Memory Management” further down in this chapter.

Handling attributes is different. Their values are normally copied. No pointers are passed and the objects are owned by the one who instantiates them. Study Section , “Attribute sets” for details.

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. This can be done by implementing your own model that

talks to that underlying storage.

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

`KDChart::Widget` allows you to get started quickly without having to master the complexities of the model/view framework in Qt 4. To make use of all the benefits model/view programming, we advise you use `KDChart::Chart`.

In better understand the relationship between `KDChart::View` and `KDChart::Widget`, compare `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. Here we use `QStandardItemModel` to store data to 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 follows:

```
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 the widget classes and what they can do. You will learn a lot by compiling and running the associated examples. The API reference de-

scribes, very simply, both ways you can use to display a Chart.

Attribute sets

Components of a chart, such as legends or axes, have attribute sets associated with them that define how they are laid out and painted. For example, both the chart itself and all areas have a set of `KDChart::BackgroundAttributes` that control whether there should be a background pixmap or a solid background color. Other attribute sets determine frame 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

Whenever you modify an attribute set, make sure to use the copy constructor for instantiating your attributes object. By doing so, you will not alter your entire existing configuration, only the desired changes in the attributes set.

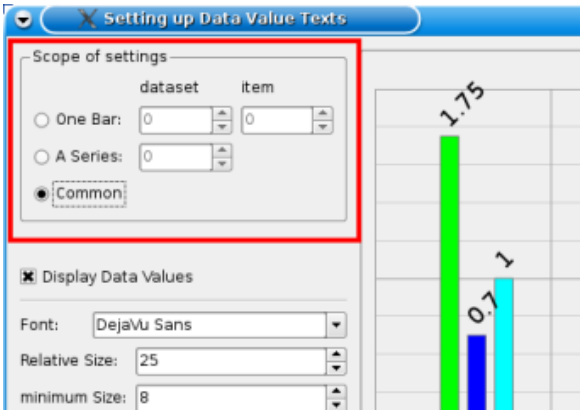
For example, the code block shown above, just changes the font and its color, but leaves all size settings the same.

Attributes can be set per cell, per column or per modelindex, but can only be queried per cell. Restricting access to the cell level ensures the proper fallback hierarchy. When 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, the model-wide setting is used. Or, if there is no value given, a default value will be applied. All of this happens automatically. 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. Per-cell changes will hide per-column changes. (see the API Reference for `KDChart::`*`[type]`*`Attributes` classes)

The example below demonstrates how the scope of some attribute settings might be selected:

Figure 2.1. Scope selection for Data Value Texts



For a closer look, check out 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 (for example: a legend) happens through mutable pointers provided by the view. However, those components may be replaced.

Code Sample

Let's elaborate 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`). The widget provides a convenient and simple way of displaying a chart without worrying over complicated details like the Coordinate Plane and other classes provided by the KD Chart 2 API.
- `KDChart::Chart` gives the user access to the full power of both Qt and KD Chart.

Essentially, `KDChart::Widget` is intended for beginners, while `KDChart::Chart` is designed for experienced 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 of which set of classes best match your needs. See the API Reference of `KDChart::Chart` and `KDChart::Widget`.

Prerequisites

As described above in Section , “KD Chart and Model/View”, in order to use the full KD Chart API, all data-to-be-charted must be made available through a class implementing the `QAbstractItemModel` interface. Before looking at some code, let's look at 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. Each of which can hold several diagrams. Currently, cartesian and polar coordinates are supported.
- The "coordinate plane" - often called the "plane" - represents the entity responsible for mapping the values to positions on the widget. The plane also shows the grid and subgrid lines. There can be several planes per chart.
- The "diagram" is the actual plot representing the data: bars, lines and other chart types. There can be several diagrams per coordinate plane.

The Procedure

Let's go through the general procedure for creating a chart. Then we will build a complete example. We will create a small application that displays a chart using `KDChart::Widget` and `KDChart::Chart`.

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 );
```

Now we will create a single line diagram using the default Cartesian coordinate plane 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 the diagrams, planes, legends, etc. Much like Qt itself, KD Chart uses a value-based approach to these attributes. With diagrams, most aspects can be adjusted at different levels of granularity. The QPen used for drawing datasets - lines, bars, etc - can be set for the whole program, a dataset, or one data point within a dataset. 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 are combined into collection classes that form logical groupings, such as: `GridAttributes`, `DataValueAttributes`, `TextAttributes`, etc....

This makes it possible to swap entire sets of properties, in one step, based on the program state. However, more often you will only want to adjust a few of the default settings. So in most cases, using the copy constructor of the settings class will suffice. For example, to use a special font for drawing a legend, you would write:

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

As we continue with the examples in the following sections and chapters, we will cover these points in more detail. Also, we recommend you check out and run the examples that shipped with your KD Chart package.

Two Ways To Create Your Chart

Now we'll go through the basic steps of creating a simple chart widget. First we'll use `KDChart::Widget` and then `KDChart::Chart`. This will give us a good overview of 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
   #include <QApplication>
   #include <KDChartWidget>

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

       KDChart::Widget widget;
       widget.resize( 600, 600 );
10
       QVector< double > vec0,  vec1,  vec2;

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

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

       widget.show();
25
       return app.exec();
   }
```

Compiling 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

Ofcourse, you can add other elements like a Title, Headers, Footers, Legends, or Axes to this simple widget. We will cover that later in greater detail. Also, notice 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

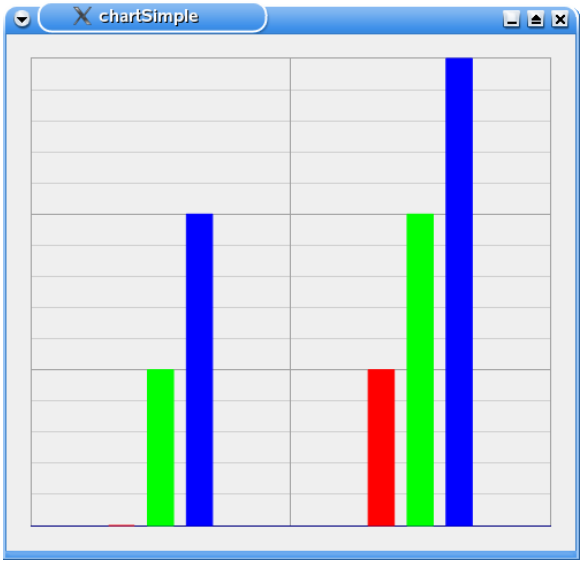
The following example is available at the following location of your KD Chart installation: `/examples/Bars/Simple/`

```
 1
   #include <QtGui>
   #include <QtSvg/QSvgGenerator>
   #include <KDChartChart>
 5 #include <KDChartBarDiagram>

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

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

        /*
        {
        // show a comment at one data item:
            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 );
        }
        */

        KDChart::BarDiagram* diagram = new KDChart::BarDiagram;
        diagram->setModel(&m_model);
        diagram->setPen( QPen( Qt::black, 0 ) );

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

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

        /*
        // render chart to a SVG
        QSvgGenerator generator;
        generator.setFileName("/home/kdab/chart.svg");
        generator.setSize(QSize(300, 300));
        generator.setViewBox(QRect(0, 0, 300, 300));
        generator.setTitle(tr("SVG Chart"));
        QPainter painter;
        painter.begin(&generator);
        painter.setRenderHint(QPainter::Antialiasing);
        m_chart.paint(&painter, generator.viewBox());
        painter.end();
        */
    }

    private:
      KDChart::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"
```

In this example, we used `QStandardItemModel` to insert and store data displayed by the diagram. We also implicitly used `KDChart::BarDiagram` by assigning the model to it. The resulting chart widget is below:

Figure 3.2. A Simple Chart



We can add more elements to this chart and change its default attributes as as in the widget example.

Later we will add various elements (Axes, Legend, Headers etc...) and configure thier attributes (Pen, Color, etc ...).

What's Next

In Chapter 4, we will describe the various chart types (diagrams) and their coordinate planes. In each example, we will also cover "chart type" attributes.

Chapter 4. Planes and Diagrams

KD Chart provides two types of planes to display diagrams:

- 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. We will use sample code, small programs and screenshots to present the chart types provided by KD Chart.

We will also learn what chart types are appropriate for a what purposes and what information is available for each type of chart. First, let's go through the chart view as well as some important concepts concerning planes and their relation to diagrams.

It is possible to have multiple diagrams within one chart. Each coordinate plane can have one or more diagrams associated to it, in which case, the diagrams will share the scale provided by the plane. Also, a chart may contain more than one coordinate plane. This way you can display multiple diagrams at different scales in the same chart, next to or, on top of one another.

Note

You can use the "reference plane" to control how planes are positioned by the layout engine by calling the `setReferenceCoordinatePlane()` method.

The reference plane concept allows two planes to share the same space even if neither has a shared axis. You may declare the respective plane to be drawn in the same cell as the plane it is referenced to. This is called "overlaying". When planes share an axis, they will typically be laid out in relation to each other as suggested by the position of the axis. If, for example, Plane1 and Plane2 share an axis to thier left, the lay out engine will position the axis to the left with Plane1 above Plane2. If Plane1 also happens to be Plane2's reference plane, both planes will be drawn over each other.

This concept is illustrated in `examples/SharedAbscissa/OverlayedDiagrams/` and `examples/SharedAbscissa/SeparateDiagrams/`, we recommend you study these examples. More information on the reference plane can be found the API documentation of `KDChart::AbstractCoordinatePlane`.

Cartesian Coordinate Planes

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

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

To implement a particular type of chart, simply create an object of its type. Call `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 common for visualizing almost any kind of data. Like Line Charts, the bar charts are ideal for comparing 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:

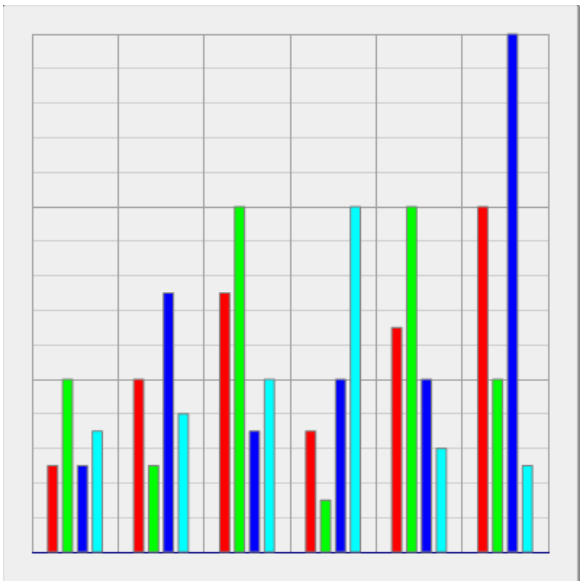
- Normal
- Stacked
- Percent

Normal Bar Charts

Tip

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

Figure 4.1. A Normal Bar Chart



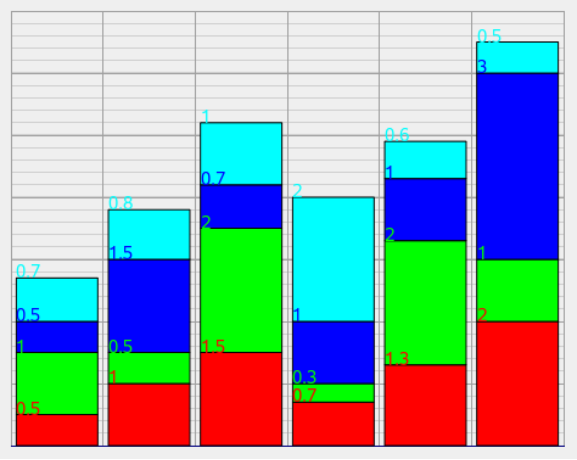
By default, KDChart displays normal bar chart. You can switch to other bar chart types using `setType( Stacked )`.

Stacked Bar Charts

Tip

Stacked bar charts are useful for comparing the sums of the individual values in each data series. They also show how much each individual value contributes to its sum.

Figure 4.2. A Stacked Bar Chart

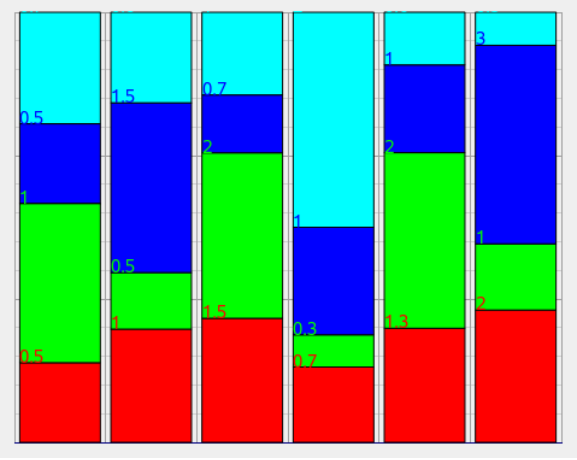


For stacked bar chart mode, call `KDChart::BarDiagram::setType( Stacked )`.

Percent Bar Charts

Unlike Stacked charts, Percent bar charts are not suitable for comparing the sums of the data series. Rather, they 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 `KD-`

`Chart::BarDiagram` function `setType( Percent )`.

Note

Three-dimensional look of the bars does not require a separate diagram type. You can enable "ThreeD" attributes for all types (`Normal`, `Stacked`, and `Percent`). We will describe this in codexample further on.

Code Sample

Look at the following code sample based on the `Simple Widget` you have already seen. In this example, we will configure the bar diagram and change its attributes with `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 must include `KDChartBarDiagram` in order 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. We also want to display it in stacked mode. `KDChart::Widget`, with its `setType()` and `setSubType()` methods, allows us make those configurations in a very simple way.

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

The default type is "Normal" for the widget. We need to implicitly pass the second parameter when calling `KDChart::Widget::setType()`. We can also change the subtype of our bar chart later 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, we want to draw a gray line around the bars to make them look nicer. This is referred to as "configuring the attributes" in a diagram. We configure a QPen and then assign it to our diagram. `KDChart::Widget::barDiagram()` will get a pointer to our widget diagram. We assign the new pen to our diagram by calling the diagram `KDChart::AbstractDiagram::setPen()` method.

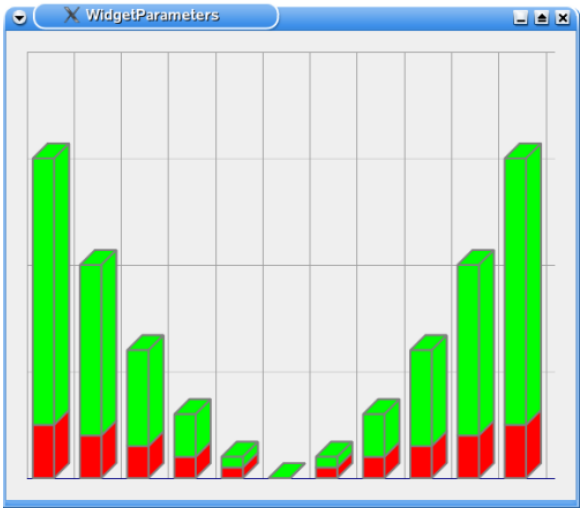
```
    //Set up your ThreeDAttributes
    //display in ThreeD mode
    ThreeDBarAttributes td(
        widget.barDiagram()->threeDBarAttributes() );
    td.setDepth(  15 );
    td.setEnabled(  true );
    widget.barDiagram()->setThreeDBarAttributes( td );
```

To display our bar chart in 3D mode, we need to configure some ThreeDBarAttributes and assign them to the diagram. The code above shows how we configure the depth of the 3D bars and enable 3D mode. Depth is an attribute only available to bar charts. 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 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` with `KDChart::Chart` is done in the same way as a `KDChart::Widget`. Simply assign the configured attributes to your bar diagram and then assign it to the chart by calling `KDChart::Chart::replaceDiagram()`.

Bars Attributes

"Bars Attributes" refers to all parameters specific to the Bar Chart type that are configured and set by the user. To get an idea of what can be configured - the "getters" and "setters" for those attributes - consult the KDChartBarAttributes API Reference.

> ## Note
>
> KD Chart 2 API separates the attributes of a specific chart type from the generic attributes common to all chart types. For example, the setters and getters for a brush, or a pen, are accessible from the `KDChart::AbstractDiagram` interface.

All attributes are preset to a reasonable default value. Each value may be modified by the user. You can call one of the diagram "set functions" like `KD-`

`Chart::BarDiagram::setBarAttributes()`, or to change a default directly, like the "Pen", 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 the object using the setters available.
- Assign the change to your Diagram using one of the setters available in `KDChart::BarDiagram`. 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:

- 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

The following sample code describes the above process. Compile and run the following example located in the `examples/Bars/Parameters/` directory of your KD Chart installation.

At the top of the file we include the header files and the KD Chart namespace.

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

using namespace KDChart;
```

We include `KDChartDataValueAttributes` so that we can display our data values. These attributes are 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 we will be assigning 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 storing our data in the model, we create a diagram. In this case, we want to display the data in a `KDChart::BarDiagram`. (The procedure is similar for displaying data in all types of diagrams.)

Now we are ready to configure specifics attributes of the bar diagram using `KDChart::BarAttributes`.

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

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

We want to configure the bars' width so that they will display a bit larger. The width of a bar is calculated automatically based on the gaps between each bar, the gaps between groups of bars, and the space available horizontally in the plane. The values interact with each other so that your bars do not exceed the plane surface horizontally. In the code example, we are increasing the value of the bars' width and also setting some lower values for the gaps. This will give us larger bars.

Note

After the attributes are configured, we need to assign the `BarAttributes` object to the diagram. You can do this for the whole diagram, at a specific index, or for a column. See the KDChart::BarDiagram API Reference for methods available on configuring those setters and getters.

We will now use the KD Chart 2 API, `KDChart::DataValueAttributes`, to display data values related to each bar. Showing data values is not exclusive to Bar Chart types, but can be used with any of the chart types. In all cases, 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 display the data values without adjusting `KDChart::TextAttributes`, but in this case, we want to demonstrate another way to customize your chart's output: You can implicitly enable your attributes, DataValue and Text, by calling their `setVisible()` methods. Once the attributes are configured, we simply assign them to the diagram. This same processs works with all other attributes as well.

Finally, I want to paint a line around one of the datasets bars. This way, I can highlight a specific data set. KDChart allows us to change the pen for all datasets or for specific indexes or values. But right now, I need to change the default pen for this data set exclusively.

```
    //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

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

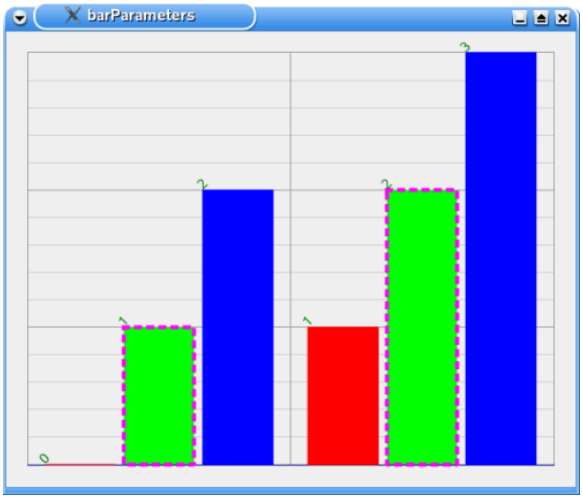
Now that we've configured and assigned the attribute, we just need to include the Bar diagram in our chart to finish the implementation.

```
    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"
```

You can use the above series of steps to configure any of KDChart's attributes. The code we have gone through produces the diagram in the following screen-shot. For more on attributes, try compiling and running the examples 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 it is set 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 Section , “ThreeD Attributes” of Chapter 8, *Customizing your Chart*.

Tips and Tricks

This section will illustrate some interesting features offered in the KD Chart 2 API. We will study a code sample and then look a screen-shot with the resulting widget.

A Complete Bar Example

In the following implementation we want 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`.)

Bar Attributes - applyable only to Bar types (e.g. `KDChart::BarAttributes::setWidth()` or `KDChart::ThreeDBarAttributes`.)

We will use a `KDChart::Chart` class and a home made `TableModel` derived from `QAbstractTableModel`.

TableModel uses a simple rectangular vector of vectors to represent a data table. It can be displayed in regular Qt views. Additionally, it provides a method, in the default configuration, to load CSV files exported by OpenOffice Calc. This allows us 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 now concentrate 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
   #ifndef MAINWINDOW_H
   #define MAINWINDOW_H

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

   namespace KDChart {
       class Chart;
10     class BarDiagram;
   }

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

   public:
       MainWindow( QWidget* parent = 0 );

20 private slots:

       void on_barTypeCB_currentIndexChanged( const QString & text );
       void on_barOrientationCB_currentIndexChanged( const QString & text );
       void on_paintValuesCB_toggled( bool checked );
25     void on_paintThreeDBarsCB_toggled( bool checked );
       void on_markColumnCB_toggled( bool checked );
       void on_markColumnSB_valueChanged( int i );
       void on_threeDDepthCB_toggled( bool checked );
       void on_depthSB_valueChanged( int i );
30     void on_widthCB_toggled( bool checked );
       void on_widthSB_valueChanged( int i );
       void on_fixPlaneSizeCB_toggled( bool checked );

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

40
   #endif /* MAINWINDOW_H */
```

In the above code, we bring up the `KDChart` namespace, as usual, and declare our slots. The user configures the 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:

```
 1
   #include "mainwindow.h"

   #include <KDChartChart>
 5 #include <KDChartDatasetProxyModel>
   #include <KDChartAbstractCoordinatePlane>
   #include <KDChartBarDiagram>
   #include <KDChartTextAttributes>
   #include <KDChartDataValueAttributes>
10 #include <KDChartThreeDBarAttributes>
   #include <KDChartBackgroundAttributes>


   #include <QDebug>
15 #include <QPainter>

   using namespace KDChart;
```

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

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

    m_model.loadFromCSV( ":/data" );

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

    // define custom color for the borders of the bars
    QPen pen(  m_bars->pen() );
    pen.setColor( Qt::black );
    pen.setWidth( 0 );
    m_bars->setPen( pen );
    m_chart->coordinatePlane()->replaceDiagram( m_bars );

    // define custom colours for the bars
    QList<QColor> bcolor;
    bcolor.append(Qt::darkGreen);
    bcolor.append(Qt::green);
    bcolor.append(Qt::darkRed);
    bcolor.append(Qt::red);
    for (int row=0; row < m_model.columnCount(); ++row) {
        m_bars->setBrush(row, QBrush(bcolor[row]));
    }

    // Configure the plane's Background
    BackgroundAttributes pba;
    //pba.setBrush( QBrush(QColor(0x20,0x20,0x60)) );
    pba.setVisible( true );
    m_chart->coordinatePlane()->setBackgroundAttributes(  pba );

    m_chart->setGlobalLeadingTop( 20 );
}


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

    m_chart->update();
}
void MainWindow::on_barOrientationCB_currentIndexChanged(const QString & text)
{
    if ( text == "Vertical" )
        m_bars->setOrientation( Qt::Vertical );
    else if ( text == "Horizontal" )
        m_bars->setOrientation( Qt::Horizontal );
    else
        qWarning (" Does not match any orientation");

    m_chart->update();
}


void MainWindow::on_paintValuesCB_toggled( bool checked )
{
    Q_UNUSED( checked );
    // We set the DataValueAttributes on a per-column basis here,
    // because we want the texts to be printed in different
```

```
        // colours - according to their respective dataset's colour.
 95     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 ) );
100         TextAttributes ta( a.textAttributes() );
            ta.setRotation( 0 );
            ta.setFont( font );
            ta .setPen( QPen( brush.color() ) );
            if ( checked )
105             ta.setVisible( true );
            else
                ta.setVisible( false );

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

        m_chart->update();
115 }


    void MainWindow::on_paintThreeDBarsCB_toggled( bool checked )
    {
120     ThreeDBarAttributes td( m_bars->threeDBarAttributes() );
        double defaultDepth = td.depth();
        if ( checked ) {
            td.setEnabled( true );
            if ( threeDDepthCB->isChecked() )
125             td.setDepth( depthSB->value() );
            else
                td.setDepth( defaultDepth );
        } else {
            td.setEnabled( false );
130     }
        m_bars->setThreeDBarAttributes( td );
        m_chart->update();
    }

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

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

        if ( threeDDepthCB->isChecked() && paintThreeDBarsCB->isChecked() )
            on_paintThreeDBarsCB_toggled( true );
    }
160
    void MainWindow::on_threeDDepthCB_toggled( bool checked )
    {
         Q_UNUSED( checked );

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

```
    void MainWindow::on_markColumnSB_valueChanged( int i )
{
        QPen pen( m_bars->pen( i ) );
        markColumnCB->setChecked( pen.color() == Qt::yellow );
    }

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

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

    void MainWindow::on_fixPlaneSizeCB_toggled( bool checked )
    {
    CartesianCoordinatePlane* plane =
            qobject_cast<CartesianCoordinatePlane*>( m_chart->coordinatePlane() );
        if( plane == 0 )
            return;
        plane->setFixedDataCoordinateSpaceRelation( checked );
    // just a little adjustment:
        // Reset the zoom settings to their initial values
        // when the releation-adjust checkbox is unchecked:
        if( ! checked ){
            m_chart->coordinatePlane()->setZoomFactorX( 1.0 );
        m_chart->coordinatePlane()->setZoomFactorY( 1.0 );
            m_chart->coordinatePlane()->setZoomCenter( QPointF(0.5, 0.5) );
        }
        m_chart->update();
    }
```

First, add our chart to the layout - just like any other Qt widget. Then, load the data to be display in the model. Assign the model to our bar diagram. To make the display a little more visually appealing, configure a Pen to draw a 'darkGray' line around the displayed bars. Finally, 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 from the GUI. via a combo box from the GUI. This can be done via a combo box. Use `KDChart::BarDiagram::setType()`, as shown below, then update the view:

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

Because we want the text to be printed in different colors, set the DataValueAttributes on a per-column basis - each according to its respective dataset's color. The user will then 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`. We are also allowing 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`. To enable or disable the 3D look, adjust the depth of the bars according to the 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, as all other Attributes types, are easy to use. The next lines of code show how to use the generic `KDChart::AbstractDiagram::setPen()`. It's available to all diagram types. This allows 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 note that there are three levels of precedence when setting the attributes: Once you have set the attributes for a column or a cell,

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

ting the attributes: 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 re-set the value, you must call the setter "per column" or "per index" - as demonstrated in the above code.

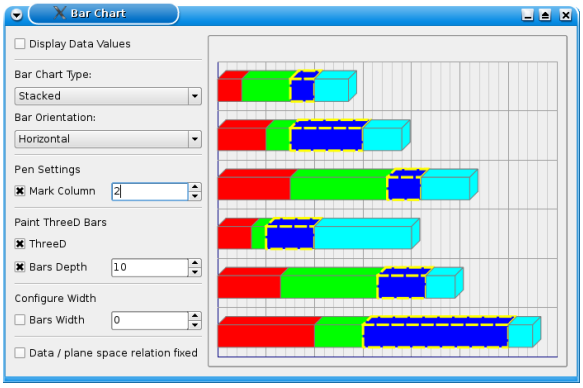
Finally, we configure a typical `KDChart::BarAttributes`, the Bar Width, so the user can change the width of the bars dynamically from the GUI.

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

Here we make use of the `KDChart::BarAttributes::setUseFixedBarWidth()` method to enable the effect. The Bar Width value being passed by the value of a Spin Box.

This screenshot shows how a widget with some attributes enabled is displayed in a full featured bar chart:

Figure 4.6. A Full featured Bar Chart



Compile and run this example for yourself. Look in the `examples/Bars/Advanced/` directory of 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.

For example, you might chart the development of stock values over a long period of time or plot the water level rise on several gauges.

As with Bar types, KD Chart generates line charts of different types. `KDChart::LineDiagram` supports the following subtypes explained below:

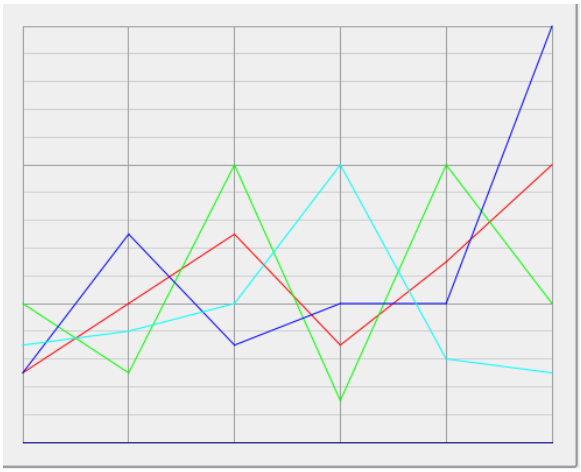
- Normal Line Chart
- Stacked Line Chart
- Percent Line chart

Normal Line Charts

Tip

Normal line charts are the most common type of line charts. They are used when comparing independent datasets to each other. For example, if you want to visualize the development of sales figures for separate departments over time, you might chart one line per department in the same graph.

Figure 4.7. A Normal Line Chart



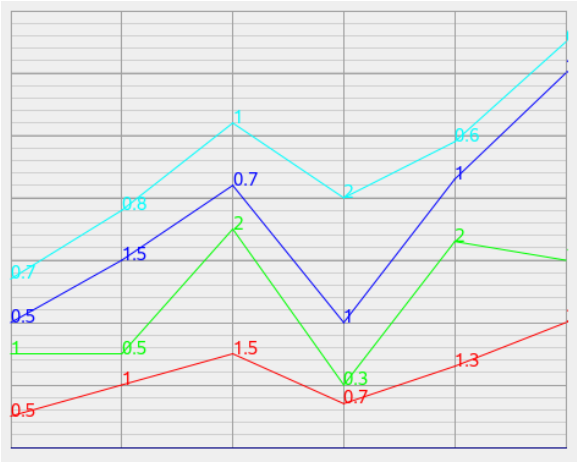
When in line chart mode, by default, KD Chart draws a normal line chart. No special method needs to be called to get a normal line chart. However, after using your `KDChart::LineDiagram` to display a different line chart subtype, you must reset the chart mode by calling `setType( Normal )`.

Stacked Line Charts

Tip

With stacked line charts you can chart the development of a series with values displayed over summarized datasets. This is useful if you are 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



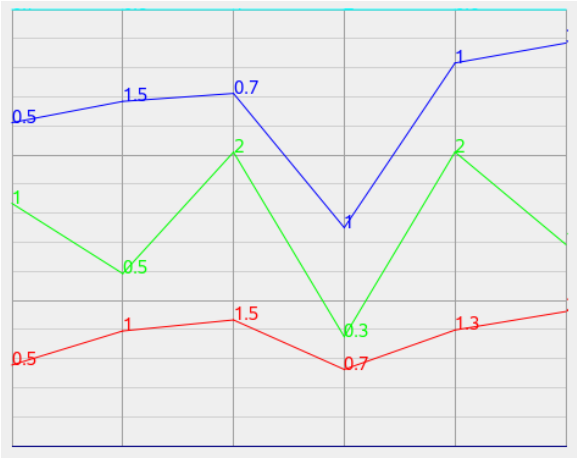
For stacked mode, call the `KDChart::LineDiagram` method `setType( Stacked )`.

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

The 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, in more detail, in the "Line Attributes" section further on.

Code Sample

In this next example we will demonstrate how to configure a line diagram and change its attributes when working with a `KDChart::Widget`. The following code sample is based on the `Simple Widget` we have been using above, see Chapter 3, *Basic steps: Create a Chart* - Section , “Widget Example”.

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` so that we can configure some of its attributes 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 don't need to change the default chart type (as Line Charts is the default). And in this case, we want to display the chart in percent mode using the `KDChart::Widget` with its `setSubType()` method.

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

Because the default sub-type is "Normal" for all types of charts, in this case, we need to call, implicitely, the `KDChart::Widget::setSubType()`. We can change the sub-

type of our line chart later 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 );
```

Say we want to draw attention to a particular data set. We can do this by creating a new style of line for the dataset we want highlight. In the above code, 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, we can easily 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 );
```

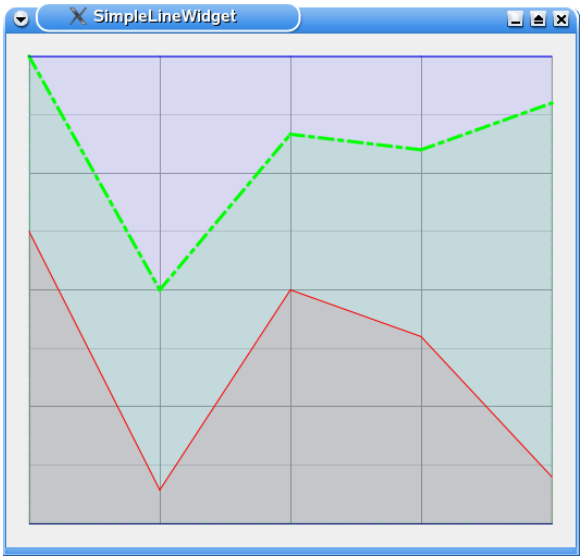
This code shows a typical use of `KDChart::LineAttributes`. We can diplay the areas as well as configure the color transparency, which is helpful when displaying a normal chart type where the areas can hide each other.

Finally, we conclude our small example:

```
    widget.show();

    return app.exec();
}
```

See the screen-shot below to view the resulting chart:

Figure 4.10. A Simple Line Chart Widget



This example can be compiled and run from the KD Chart installation in: `examples/Lines/SimpleLineWidget/`

Note

Configuring the attributes for a `KDChart::LineDiagram` (making use of a `KDChart::Chart`) is done the same way as a `KDChart::Widget`. You just need to assign the configured attributes to your line diagram, then assign the diagram to the chart by calling `KDChart::Chart::replaceDiagram()`.

Lines Attributes

Because KDChart uses Pen to draw lines, there are only a few attributes to configure specifically on a line chart. Pen and Brush are generic attributes common to all types of diagrams and they are handled by `KDChart::AbstractDiagram` (from which `KDChart::LineDiagram` is indirectly derived).

So, to make it simple for the user, we have added some convenient functions to `KDChart::LineAttributes` for displaying Areas and setting transparencies for all the subtypes of a line chart. We will go through those methods further on in Section , “Area Charts” in this Chapter.

`KDChart::LineDiagram` makes displaying line charts easy. You can modify LineDiagram attributes and methods, combine attributes with `KDChart::MarkerAttributes`, generate line chart subtypes (as described above), or display Area Charts and Point charts. In the next sections, we will present each of these options with sample code and ready-to-use examples.

Configuring LineAttributes is similar to configuring the other chart types:

- Create a `KDChart::LineAttributes` object by calling `KDChart::LineDiagram::lineAttributes()`.
- Configure the object using the setters available.
- Assign it to your Diagram with the help of one of the setters available in `KDChart::LineDiagram`. All attributes can be configured, and 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 the which can be set and retrieved as described below:

- MissingValuesPolicy: specify 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 colored areas to be displayed.

Note

All other attributes, such as ThreeDLineAttributes (specific to line charts), MarkerAttributes, DataValueAttributes and TextAttributes, etc., are also available to line charts and their sub-types.

Line Attributes Sample

The following sample code shows how to configure Line Attributes. (This example can be found in the `examples/Lines/Parameters/` directory of your KD Chart installation).

First, we include the header files and bring in the KD Chart namespace.

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

using namespace KDChart;
```

We have included `KDChartDataValueAttributes` to display our data values. Those attributes can be used by all types of charts and are not specific to the Line diagram.

Next, we use a `KDChart::Chart` class and a `QStandardItemModel` to store the data which will be assigned to our diagram.

```
class ChartWidget : public QWidget {
Q_OBJECT
public:
    explicit ChartWidget(QWidget* parent=0)
    : QWidget(parent)
    {
        m_model.insertRows( 0,5, QModelIndex() );
        m_model.insertColumns( 0,5, QModelIndex() );

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

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

After we have 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 similar for all types of diagrams.

Now we are ready to configure our attributes. We want to display the data values and configure the 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()`. Now we need to set it up with our own values and then assign it to our diagram.

We could have displayed the data values without caring about setting its `KDChart::TextAttributes`, but we want to demonstrate customizable features too. Notice that you must implicitly enable your attributes (DataValue and Text) by calling their `setVisible()` methods before you assign it to the diagram.

Note

After configuring the attributes, we need to assign them to the diagram. You can do this 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 each attribute's setters and getters.

If we want to focus the reader's attention on a particular section, we can configure the Pen to draw a dataset (section of a line) differently:

```
// 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 );
```

We can, likewise, change the pen for all datasets. Notice how we call the pen for this particular dataset before changing its style and width. This is done to keep the line's original color for consistancy.

Note

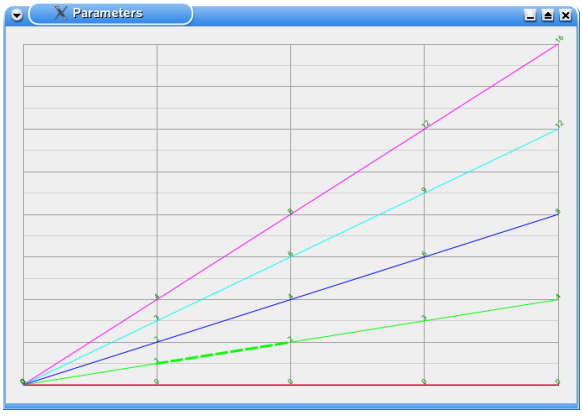
The Pen and the Brush setters and getters are implemented at a lower level in our `KDChart::AbstractDiagram` class for a cleaner code structure. These methods are used by all types of diagrams. Their configuration is very simple and straight forward as you can see in the sample code above. Create or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

Now that we've configured and assigned our attributes, 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 supported attributes for all chart types. The widget, resulting from the code we have gone through, can be seen in the following screen-shot. We recommend you compile and run the example related to this section. It is located in the `examples/Lines/Parameters/` directory of your KD Chart installation.

Figure 4.11. Line With Configured Attributes



The subtypes of a line chart (Normal, Stacked or Percent) are not set via an attribute class, rather they are set 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 the other attributes. We will cover more KD Chart 2 ThreeD features in Section , “ThreeD Attributes” of Chapter 8, *Customizing your Chart*.

Tips and Tricks

In this section, we want to give you some examples of 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: `KDChart::AbstractDiagram::setPen()`, `KDChart::DataValueAttributes` and `KDChart::TextAttributes`.

We will use the `KDChart::Chart` class and also a homemade `TableModel` that 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. It also provides, in the default configuration, a method to load CSV files exported by OpenOffice Calc. This allows us to prepare the test data using spreadsheet software.

TableModel 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 located in the `examples/tools/` directory of your KD Chart installation.

For now, let's concentrate on our Line chart implementation. Consult the following files:

(Other needed files, like the ui, pro , qrc, CSV and main.cpp, can be found in the `examples/Lines/Advanced/` directory of your installation.)

```
 1
   #ifndef MAINWINDOW_H
   #define MAINWINDOW_H

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

   namespace KDChart {
       class Chart;
10     class LineDiagram;
   }

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

   public:
       MainWindow( QWidget* parent = 0 );

20 private:
       bool eventFilter(QObject* target, QEvent* event);

   private slots:

25     void on_lineTypeCB_currentIndexChanged( const QString & text );
       void on_paintValuesCB_toggled( bool checked );
       void on_centerDataPointsCB_toggled( bool checked );
       void on_threeDModeCB_toggled( bool checked );
       void on_depthSB_valueChanged( int i );
30     void on_animateAreasCB_toggled( bool checked );
       void on_highlightAreaCB_toggled( bool checked );
       void on_highlightAreaSB_valueChanged( int i );
       void setHighlightArea( int row, int column, int opacity,
                              bool checked, bool doUpdate );
35     void on_trackAreasCB_toggled( bool checked );
       void on_trackAreasSB_valueChanged( int i );
       void setTrackedArea( int column, bool checked, bool doUpdate );
       void slot_timerFired();
       void on_reverseHorizontalCB_toggled( bool checked );
40     void on_reverseVerticalCB_toggled( bool checked );

   private:
       KDChart::Chart* m_chart;
       KDChart::LineDiagram* m_lines;
45     TableModel m_model;
       int m_curRow;
       int m_curColumn;
       int m_curOpacity;
   };
50

   #endif /* MAINWINDOW_H */


55
```

In the above code, we bring up the `KDChart` namespace, as usual, and declare our slots. This lets the user configure the 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:

```
 1
   #include "mainwindow.h"

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

```
#include <KDChartThreeDLineAttributes>

#include <QTimer>
#include <QMouseEvent>

using namespace KDChart;

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

    m_curColumn = -1;
    m_curOpacity = 0;

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

    m_model.loadFromCSV( ":/data" );

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

    CartesianAxis *xAxis = new CartesianAxis( m_lines );
    CartesianAxis *yAxis = new CartesianAxis ( m_lines );
    xAxis->setPosition ( KDChart::CartesianAxis::Bottom );
    yAxis->setPosition ( KDChart::CartesianAxis::Left );
    m_lines->addAxis( xAxis );
    m_lines->addAxis( yAxis );

    m_chart->coordinatePlane()->replaceDiagram( m_lines );
    m_chart->setGlobalLeading( 20,  20,  20,  20 );
    // Instantiate the timer
    QTimer *timer = new QTimer(this);
    connect(timer, SIGNAL(timeout()), this, SLOT(slot_timerFired()));
    timer->start(30);

    //Change the cursor to IBeamCursor inside Chart widget.
    m_chart->setCursor(Qt::IBeamCursor);

    //Install event filter on Chart to get the mouse position
    m_chart->installEventFilter(this);
}

/**
  Event filter for getting mouse position
*/
bool MainWindow::eventFilter(QObject* target, QEvent* event)
{
    if (target == m_chart) {
        if (event->type() == QEvent::MouseMove) {
            QMouseEvent* mouseEvent = static_cast<QMouseEvent *>(event);
            qDebug() << "Mouse position " << mouseEvent->pos();
        }
    }
    return QWidget::eventFilter(target, event);
}

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

    m_chart->update();
}
```

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

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

105 void MainWindow::on_centerDataPointsCB_toggled( bool checked )
    {
        m_lines->setCenterDataPoints( checked );
        m_chart->update();
    }
110
    void MainWindow::on_animateAreasCB_toggled( bool checked )
    {
        if( checked ){
            highlightAreaCB->setCheckState( Qt::Unchecked );
115         m_curRow = 0;
            m_curColumn = 0;
        }else{
            m_curColumn = -1;
        }
120     highlightAreaCB->setEnabled( ! checked );
        highlightAreaSB->setEnabled( ! checked );
        // un-highlight all previously highlighted columns
        const int rowCount = m_lines->model()->rowCount();
        const int colCount = m_lines->model()->columnCount();
125     for ( int iColumn = 0; iColumn<colCount; ++iColumn ){
            setHighlightArea( -1, iColumn, 127, false, false );
            for ( int iRow = 0; iRow<rowCount; ++iRow )
            //    m_lines->resetLineAttributes( cellIndex );
                setHighlightArea( iRow, iColumn, 127, false, false );
130     }
        m_chart->update();
        m_curOpacity = 0;
    }

135 void MainWindow::slot_timerFired()
    {
        if( m_curColumn < 0 ) return;
        m_curOpacity += 8;
        if( m_curOpacity > 255 ){
140         setHighlightArea( m_curRow, m_curColumn, 127, false, false );
            m_curOpacity = 5;
            ++m_curRow;
            if( m_curRow >= m_lines->model()->rowCount(m_lines->rootIndex()) ){
                m_curRow = 0;
145             ++m_curColumn;
                if( m_curColumn >=
                    m_lines->model()->columnCount( m_lines->rootIndex() ) )
                    m_curColumn = 0;
            }
150     }
        setHighlightArea( m_curRow, m_curColumn, m_curOpacity, true, true );
    }

    void MainWindow::setHighlightArea( int row, int column, int opacity,
155                                    bool checked, bool doUpdate )
    {
        if( row < 0 ){
```

```
            // highlight a complete dataset
            LineAttributes la = m_lines->lineAttributes( column );
160         if ( checked ) {
                la.setDisplayArea( true );
                la.setTransparency( opacity );
            }  else {
                la.setDisplayArea( false );
165         }
            m_lines->setLineAttributes( column, la );
        }else{
            // highlight two segments only
            if( row ){
170             QModelIndex cellIndex( m_lines->model()->index(
                        row-1, column, m_lines->rootIndex() ) );
                if ( checked ) {
                    LineAttributes la( m_lines->lineAttributes( cellIndex ) );
                    la.setDisplayArea( true );
175                 la.setTransparency( 255-opacity );
                    // set specific line attribute settings for this cell
                    m_lines->setLineAttributes( cellIndex, la );
                }  else {
                    // remove any cell-specific line attribute settings
180                 // from the indexed cell
                    m_lines->resetLineAttributes( cellIndex );
                }
            }
            if( row < m_lines->model()->rowCount(m_lines->rootIndex()) ){
185             QModelIndex cellIndex( m_lines->model()->index(
                        row, column, m_lines->rootIndex() ) );
                if ( checked ) {
                    LineAttributes la( m_lines->lineAttributes( cellIndex ) );
                    la.setDisplayArea( true );
190                 la.setTransparency( opacity );
                    // set specific line attribute settings for this cell
                    m_lines->setLineAttributes( cellIndex, la );
                }  else {
                    // remove any cell-specific line attribute settings
195                 // from the indexed cell
                    m_lines->resetLineAttributes( cellIndex );
                }
            }
        }
200     if( doUpdate )
            m_chart->update();
    }

    void MainWindow::on_highlightAreaCB_toggled( bool checked )
205 {
        setHighlightArea( -1, highlightAreaSB->value(), 127, checked, true );
    }

    void MainWindow::on_highlightAreaSB_valueChanged( int i )
210 {
        Q_UNUSED( i );
        if ( highlightAreaCB->isChecked() )
            on_highlightAreaCB_toggled( true );
        else
215         on_highlightAreaCB_toggled( false);
    }

    void MainWindow::on_threeDModeCB_toggled( bool checked )
    {
220     ThreeDLineAttributes td( m_lines->threeDLineAttributes() );
        td.setDepth( depthSB->value() );
        if ( checked )
            td.setEnabled(  true );
        else
225         td.setEnabled(  false );

        m_lines->setThreeDLineAttributes( td );

        m_chart->update();
230 }

    void MainWindow::on_depthSB_valueChanged( int i )
```

```
    {
        Q_UNUSED( i );
235     if ( threeDModeCB->isChecked() )
            on_threeDModeCB_toggled( true );
    }

    void MainWindow::on_trackAreasCB_toggled( bool checked )
240 {
        setTrackedArea( trackAreasSB->value(), checked, true );
    }

    void MainWindow::on_trackAreasSB_valueChanged( int i )
245 {
        Q_UNUSED( i );
        on_trackAreasCB_toggled( trackAreasCB->isChecked() );
    }

250 void MainWindow::setTrackedArea( int column, bool checked, bool doUpdate )
    {
        const int rowCount    = m_model.rowCount(    m_lines->rootIndex() );
        const int columnCount = m_model.columnCount( m_lines->rootIndex() );
        for( int i = 0; i < rowCount; ++i ) {
255         for( int j = 0; j < columnCount; ++j ) {
                QModelIndex cellIndex( m_model.index( i, j,
                                       m_lines->rootIndex() ) );
                ValueTrackerAttributes va(
                        m_lines->valueTrackerAttributes( cellIndex ) );
260             va.setEnabled( checked && j == column );
                va.setAreaBrush( QColor( 255, 255, 0, 50 ) );
                m_lines->setValueTrackerAttributes( cellIndex, va );
            }
        }
265     if( doUpdate )
            m_chart->update();
    }

    void MainWindow::on_reverseHorizontalCB_toggled( bool checked )
270 {
        static_cast<KDChart::CartesianCoordinatePlane*>(
                m_chart->coordinatePlane() )->setHorizontalRangeReversed(
                    checked );
    }
275
    void MainWindow::on_reverseVerticalCB_toggled( bool checked )
    {
        static_cast<KDChart::CartesianCoordinatePlane*>(
                m_chart->coordinatePlane() )->setVerticalRangeReversed(
280                 checked );
    }
```

First, we add our chart to the layout (as with any other Qt widget), then load the data to be displayed into our model, and then, assign the model to our line diagram. We also want to set up a `QTimer` 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, then 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. We also want to allow the user to change the default font for our data value texts (to make it look 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 set the paint color for data value texts to a default dataset-color. This is done by retrieving the brush for each dataset and assigning the color of the brush to the pen.

Note

There are three levels of precedence when setting the attributes:

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

This means that once you set the attributes for specific a column or cell, you will not be able to change those settings by calling the "global" method. To change the attributes value, you'd have to call the setter per column or per index, as demonstrated in the above code.

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

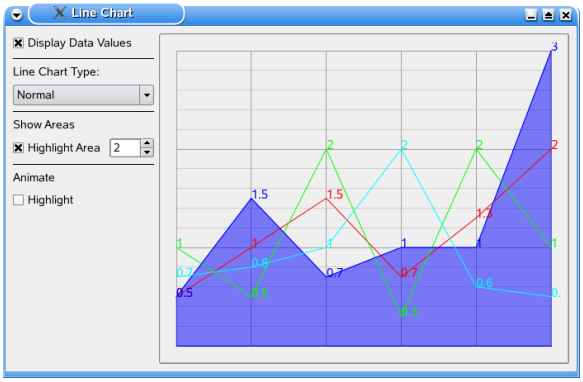
```
....
LineAttributes la = m_lines->lineAttributes( column );
if ( checked ) {
    la.setDisplayArea( true );
    la.setTransparency( opacity );
}  else {
    la.setDisplayArea( false );
}
m_lines->setLineAttributes( column, la );
...
m_chart->update();
...
```

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 running our animation. You can 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, in 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

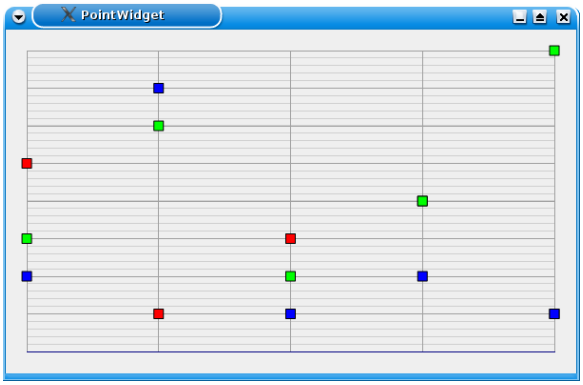


The following sections, covering Point charts and Area, are closely related to line charts. Point charts are line diagrams with Markers but the lines themselves are not painted. Area charts are line charts where the area below the lines filled with the respective dataset's color.

Point Charts

Point charts often are used to visualize large amounts of data. This can be done 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 its own type, but a special kind of Line Chart. As we will see in the following code sample, instead of drawing lines, KD Chart paints markers to display data.

Creating a point chart is easy:

- 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 paints a very simple point chart. It is based on the `examples/Widget/Simple/`. The 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 uses a `KDChart::Widget` and a `KDChart::LineDiagram` but 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

You've probably figured out, from the section above, that point charts are line charts configured with no pen (and thus no lines). This allows us to use the generic classes, `KDChart::DataValueAttributes` and its `KDChart::MarkerAttributes`, which are available to all diagram types supported by KD Chart 2.

For this reason, we will cover sections related to those subjects, in particular Chapter 8, *Customizing your Chart* - Section , “Markers Attributes” or Section , “Data Values Attributes”, and then finish section by implementing a full featured point chart.

Tips and Tricks

We want to give you some examples of 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 Point Example

In the following implementation we want to be able to:

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

Lets concentrate on our Line chart implementation for now. Consult the files below. Other needed files like the ui, pro , qrc ,CSV and main.cpp can be located in the `examples/Lines/PointChart/` directory of your installation.

```
 1
   #ifndef MAINWINDOW_H
   #define MAINWINDOW_H

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

   namespace KDChart {
       class Chart;
10     class LineDiagram;
   }

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

   public:
       MainWindow( QWidget* parent = 0 );

20 private slots:

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

30 private:
       KDChart::Chart* m_chart;
       KDChart::LineDiagram* m_lines;
       TableModel m_model;
   };
35

   #endif /* MAINWINDOW_H */
```

```
40
```

In the above code, we bring up the `KDChart` namespace as usual and declare our slots. This lets the user configure the 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
   #include "mainwindow.h"

   #include <KDChartChart>
 5 #include <KDChartLineDiagram>
   #include <KDChartTextAttributes>
   #include <KDChartDataValueAttributes>
   #include <KDChartMarkerAttributes>

10
   using namespace KDChart;

   MainWindow::MainWindow( QWidget* parent ) :
       QWidget( parent )
15 {
       setupUi( this );

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

       m_model.loadFromCSV( ":/data" );

       // Set up the diagram
25     m_lines = new LineDiagram();
       m_lines->setModel( &m_model );
       m_chart->coordinatePlane()->replaceDiagram( m_lines );
       m_chart->setGlobalLeading( 20, 20, 20, 20 );

30     on_paintLinesCB_toggled( false );
       on_paintMarkersCB_toggled( true );
   }

   void MainWindow::on_lineTypeCB_currentIndexChanged( const QString & text )
35 {
       if ( text == "Normal" )
           m_lines->setType( LineDiagram::Normal );
       else if ( text == "Stacked" )
           m_lines->setType( LineDiagram::Stacked );
40     else if ( text == "Percent" )
           m_lines->setType( LineDiagram::Percent );
       else
           qWarning (" Does not match any type");

45     m_chart->update();
   }

   void MainWindow::on_paintValuesCB_toggled( bool checked )
   {
50     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() );
55         ta.setRotation( 0 );
           ta.setFont( QFont( "Comic" ) );
           ta.setPen( QPen( brush.color() ) );

           if ( checked )
```

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

void MainWindow::on_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();
}


void MainWindow::on_paintMarkersCB_toggled( bool checked )
{
    // set up a map with different marker styles
    MarkerAttributes::MarkerStylesMap map;
    map.insert( 0, MarkerAttributes::MarkerSquare );
    map.insert( 1, MarkerAttributes::MarkerCircle );
    map.insert( 2, MarkerAttributes::MarkerRing );
    map.insert( 3, MarkerAttributes::MarkerCross );
    map.insert( 4, MarkerAttributes::MarkerDiamond );


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

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

```
135             ma.setMarkerStyle( MarkerAttributes::MarkerCross );
                break;
            case 8:
                ma.setMarkerStyle( MarkerAttributes::MarkerFastCross );
                break;
140         }

            QPen markerPen( lineBrush.color() );
            ma.setPen( markerPen );
            ma.setVisible(  true );
145         dva.setTextAttributes( ta );
            dva.setMarkerAttributes( ma );

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

155     m_chart->update();
    }


    void MainWindow::on_markersStyleCB_currentIndexChanged( const QString & text )
160 {
        Q_UNUSED( text );
        if ( paintMarkersCB->isChecked() )
            on_paintMarkersCB_toggled( true );
    }
165

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

175 void MainWindow::on_markersHeightSB_valueChanged( int /*i*/ )
    {
        markersWidthSB->setValue( markersHeightSB->value() );
        if ( paintMarkersCB->isChecked() )
            on_paintMarkersCB_toggled( true );
180 }
```

In the next examples, we will not comment on the code in great detail as it is similar to what we have already seen in the line chart code. We will highlight relevant variances.

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. We do that by looping through the datasets.

Note

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

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

attributes: This 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. Instead, you must call the setter per column or per index, as demonstrated in the above code.

To store different Markers styles, we make use of `MarkerAttributes::MarkerStylesMap map()`.

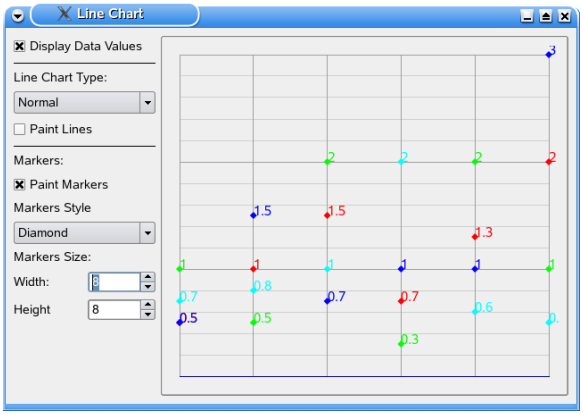
```
...
MarkerAttributes::MarkerStylesMap map;
map.insert( 0, MarkerAttributes::MarkerSquare );
map.insert( 1, MarkerAttributes::MarkerCircle );
map.insert( 2, MarkerAttributes::MarkerRing );
map.insert( 3, MarkerAttributes::MarkerCross );
map.insert( 4, MarkerAttributes::MarkerDiamond );
...
MarkerAttributes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

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

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

To compile and run this example, check the `examples/Lines/PointChart/` directory in your KD Chart installation. The resulting widget is displayed in the figure below.

Figure 4.14. A Full featured Point Chart



Note

For two-dimensional data you would apply the same technique, as described above, to the `KDChart::Plotter` class. For details, please look at `examples/Plotter/BubbleChart/`.

Area Charts

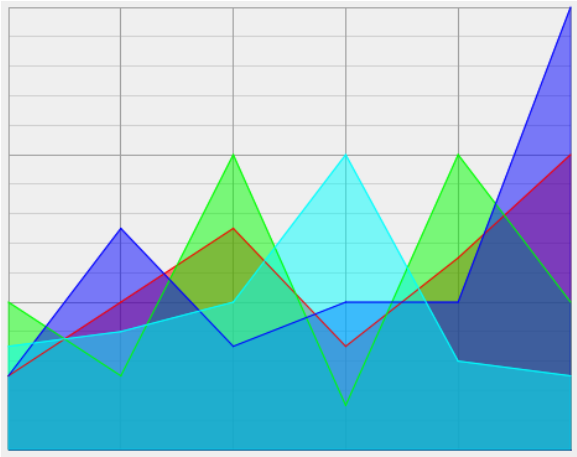
Even more than a Line Chart, an area chart can give a good visual impression of how different datasets relate to each other. This chart type would be ideal for showing, for example, how several sources contributed to increasing ozone values in a conurbation during a summer's months.

Area charts are just like Line Charts. They are made up of several points connected by lines. The difference being that in an area chart, the space below the line is filled with the dataset color. This makes each dataset's relative values easy to analyze.

Since some dataset areas overlap, we have introduced an attribute that allows the user to configure the level of transparency (more about that in Section , “Area Attributes” below). This makes it easier to see all the chart's points. KD Chart 2 allows "Area" display for all line chart subtypes. This way, users can display non-overlapping line types as well. 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.

- In this chapter, it stands for a special chart type or, even more accurately, as a line diagram attribute.
- In the rest of the chapters, it refers to different parts of a chart (normally rectangular) like the *legend area* or the *headers area*.

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 may also configure the level of transparency.

Area Sample Code

Lets look at some lines of code using an area chart:

```
// 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 );
```

The Brush and Pen settings, as well as all other configurable attributes, can also be set. This gives the user a lot of flexibility (to 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).

Area Attributes

There are not any special attributes specific to the "Area chart". As explained above, the Area chart display mode, itself, is actually implemented as an attribute of Line Chart. This provides us with generic attributes common to all chart types thereby providing us with full flexibility to configure our Area chart.

Tips and Tricks

In this section, we will give you some examples of interesting features offered by the KD Chart 2 API. We will study the code and display a screenshot of the resulting widget.

A Complete Area Example

Note

This example has already been presented in details in codexample. You do not 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 as well aa home made `TableModel` for convenience. The "TableModel" is derived from `QAbstractTableModel`.

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

For now, let's concentrate on our Line chart implementation. In addition to the examples below, the other needed files (like the ui, pro , qrc ,CSV and main.cpp files) can be found in the `examples/Lines/Advanced/` directory of your installation.

```
 1
   #ifndef MAINWINDOW_H
   #define MAINWINDOW_H

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

   namespace KDChart {
       class Chart;
10     class LineDiagram;
   }

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

   public:
       MainWindow( QWidget* parent = 0 );

20 private:
       bool eventFilter(QObject* target, QEvent* event);

   private slots:

25     void on_lineTypeCB_currentIndexChanged( const QString & text );
       void on_paintValuesCB_toggled( bool checked );
       void on_centerDataPointsCB_toggled( bool checked );
       void on_threeDModeCB_toggled( bool checked );
       void on_depthSB_valueChanged( int i );
30     void on_animateAreasCB_toggled( bool checked );
       void on_highlightAreaCB_toggled( bool checked );
       void on_highlightAreaSB_valueChanged( int i );
       void setHighlightArea( int row, int column, int opacity,
                              bool checked, bool doUpdate );
35     void on_trackAreasCB_toggled( bool checked );
       void on_trackAreasSB_valueChanged( int i );
       void setTrackedArea( int column, bool checked, bool doUpdate );
       void slot_timerFired();
       void on_reverseHorizontalCB_toggled( bool checked );
40     void on_reverseVerticalCB_toggled( bool checked );

   private:
       KDChart::Chart* m_chart;
       KDChart::LineDiagram* m_lines;
45     TableModel m_model;
       int m_curRow;
       int m_curColumn;
       int m_curOpacity;
   };
50

   #endif /* MAINWINDOW_H */
```

```
55
```

In the above code, we bring up the `KDChart` namespace, as usual, and declare our slots. This allows the user configure his 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
   #include "mainwindow.h"

   #include <KDChartChart>
 5 #include <KDChartLineDiagram>
   #include <KDChartTextAttributes>
   #include <KDChartDataValueAttributes>
   #include <KDChartThreeDLineAttributes>

10
   #include <QTimer>
   #include <QMouseEvent>

   using namespace KDChart;
15
   MainWindow::MainWindow( QWidget* parent ) :
       QWidget( parent )
   {
       setupUi( this );
20
       m_curColumn = -1;
       m_curOpacity = 0;

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

       m_model.loadFromCSV( ":/data" );

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

       CartesianAxis *xAxis = new CartesianAxis( m_lines );
35     CartesianAxis *yAxis = new CartesianAxis ( m_lines );
       xAxis->setPosition ( KDChart::CartesianAxis::Bottom );
       yAxis->setPosition ( KDChart::CartesianAxis::Left );
       m_lines->addAxis( xAxis );
       m_lines->addAxis( yAxis );
40
       m_chart->coordinatePlane()->replaceDiagram( m_lines );
       m_chart->setGlobalLeading( 20,  20,  20,  20 );
       // Instantiate the timer
       QTimer *timer = new QTimer(this);
45     connect(timer, SIGNAL(timeout()), this, SLOT(slot_timerFired()));
       timer->start(30);

       //Change the cursor to IBeamCursor inside Chart widget.
       m_chart->setCursor(Qt::IBeamCursor);
50
       //Install event filter on Chart to get the mouse position
       m_chart->installEventFilter(this);
   }

55 /**
     Event filter for getting mouse position
   */
   bool MainWindow::eventFilter(QObject* target, QEvent* event)
   {
```

```
    if (target == m_chart) {
        if (event->type() == QEvent::MouseMove) {
            QMouseEvent* mouseEvent = static_cast<QMouseEvent *>(event);
            qDebug() << "Mouse position " << mouseEvent->pos();
        }
    }
    return QWidget::eventFilter(target, event);
}

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

    m_chart->update();
}

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

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

void MainWindow::on_centerDataPointsCB_toggled( bool checked )
{
    m_lines->setCenterDataPoints( checked );
    m_chart->update();
}

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

```
void MainWindow::slot_timerFired()
{
    if( m_curColumn < 0 ) return;
    m_curOpacity += 8;
    if( m_curOpacity > 255 ){
        setHighlightArea( m_curRow, m_curColumn, 127, false, false );
        m_curOpacity = 5;
        ++m_curRow;
        if( m_curRow >= m_lines->model()->rowCount(m_lines->rootIndex()) ){
            m_curRow = 0;
            ++m_curColumn;
            if( m_curColumn >=
                m_lines->model()->columnCount( m_lines->rootIndex() ) )
                m_curColumn = 0;
        }
    }
    setHighlightArea( m_curRow, m_curColumn, m_curOpacity, true, true );
}

void MainWindow::setHighlightArea( int row, int column, int opacity,
                                   bool checked, bool doUpdate )
{
    if( row < 0 ){
        // highlight a complete dataset
        LineAttributes la = m_lines->lineAttributes( column );
        if ( checked ) {
            la.setDisplayArea( true );
            la.setTransparency( opacity );
        }  else {
            la.setDisplayArea( false );
        }
        m_lines->setLineAttributes( column, la );
    }else{
        // highlight two segments only
        if( row ){
            QModelIndex cellIndex( m_lines->model()->index(
                    row-1, column, m_lines->rootIndex() ) );
            if ( checked ) {
                LineAttributes la( m_lines->lineAttributes( cellIndex ) );
                la.setDisplayArea( true );
                la.setTransparency( 255-opacity );
                // set specific line attribute settings for this cell
                m_lines->setLineAttributes( cellIndex, la );
            }  else {
                // remove any cell-specific line attribute settings
                // from the indexed cell
                m_lines->resetLineAttributes( cellIndex );
            }
        }
        if( row < m_lines->model()->rowCount(m_lines->rootIndex()) ){
            QModelIndex cellIndex( m_lines->model()->index(
                    row, column, m_lines->rootIndex() ) );
            if ( checked ) {
                LineAttributes la( m_lines->lineAttributes( cellIndex ) );
                la.setDisplayArea( true );
                la.setTransparency( opacity );
                // set specific line attribute settings for this cell
                m_lines->setLineAttributes( cellIndex, la );
            }  else {
                // remove any cell-specific line attribute settings
                // from the indexed cell
                m_lines->resetLineAttributes( cellIndex );
            }
        }
    }
    if( doUpdate )
        m_chart->update();
}

void MainWindow::on_highlightAreaCB_toggled( bool checked )
{
    setHighlightArea( -1, highlightAreaSB->value(), 127, checked, true );
}

void MainWindow::on_highlightAreaSB_valueChanged( int i )
```

```
{
    Q_UNUSED( i );
    if ( highlightAreaCB->isChecked() )
        on_highlightAreaCB_toggled( true );
    else
        on_highlightAreaCB_toggled( false);
}

void MainWindow::on_threeDModeCB_toggled( bool checked )
{
    ThreeDLineAttributes td( m_lines->threeDLineAttributes() );
    td.setDepth( depthSB->value() );
    if ( checked )
        td.setEnabled(  true );
    else
        td.setEnabled(  false );

    m_lines->setThreeDLineAttributes( td );

    m_chart->update();
}

void MainWindow::on_depthSB_valueChanged( int i )
{
    Q_UNUSED( i );
    if ( threeDModeCB->isChecked() )
        on_threeDModeCB_toggled( true );
}

void MainWindow::on_trackAreasCB_toggled( bool checked )
{
    setTrackedArea( trackAreasSB->value(), checked, true );
}

void MainWindow::on_trackAreasSB_valueChanged( int i )
{
    Q_UNUSED( i );
    on_trackAreasCB_toggled( trackAreasCB->isChecked() );
}

void MainWindow::setTrackedArea( int column, bool checked, bool doUpdate )
{
    const int rowCount    = m_model.rowCount(    m_lines->rootIndex() );
    const int columnCount = m_model.columnCount( m_lines->rootIndex() );
    for( int i = 0; i < rowCount; ++i ) {
        for( int j = 0; j < columnCount; ++j ) {
            QModelIndex cellIndex( m_model.index( i, j,
                                   m_lines->rootIndex() ) );
            ValueTrackerAttributes va(
                    m_lines->valueTrackerAttributes( cellIndex ) );
            va.setEnabled( checked && j == column );
            va.setAreaBrush( QColor( 255, 255, 0, 50 ) );
            m_lines->setValueTrackerAttributes( cellIndex, va );
        }
    }
    if( doUpdate )
        m_chart->update();
}

void MainWindow::on_reverseHorizontalCB_toggled( bool checked )
{
    static_cast<KDChart::CartesianCoordinatePlane*>(
            m_chart->coordinatePlane() )->setHorizontalRangeReversed(
                checked );
}

void MainWindow::on_reverseVerticalCB_toggled( bool checked )
{
    static_cast<KDChart::CartesianCoordinatePlane*>(
            m_chart->coordinatePlane() )->setVerticalRangeReversed(
                checked );
}
```

First off, we added our chart to the layout (the same as we would do with any other Qt widget). Then we 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 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. To do this, use `KDChart::BarDiagram::setType()`, as shown below, and update 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 to allow the user to display or hide data values from the GUI. We can let him to change the default font for the data value texts. This helps to make everything look 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 the text displaying our data values will be painted with the default color. We retrieve the brush for each dataset and assign a brush color to the pen.

Note

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

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

attributes: This 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 set it to another value, you must call the setter per column or per index, as demonstrated in the above code.

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

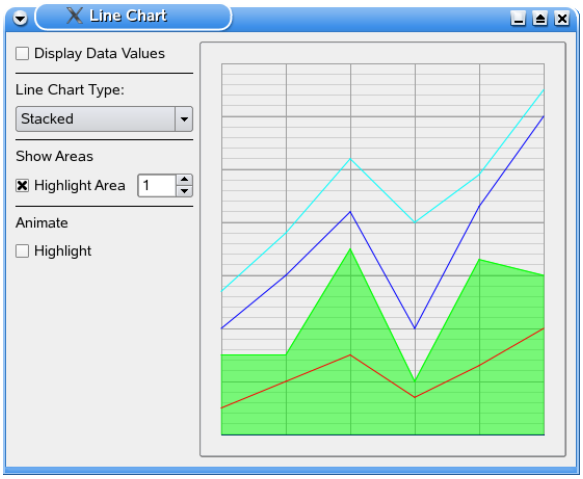
```
....
LineAttributes la( m_lines->lineAttributes( column ) );
if ( checked ) {
    la.setDisplayArea( true );
    la.setTransparency( opacity );
} else {
    la.setDisplayArea( false );
}
m_lines->setLineAttributes( column, la );
...
m_chart->update();
...
```

This can be implemented by configuring our line attributes and assigning them, by data-set, to the diagram (as shown above).

The same procedure can be used for running our animation. To learn more about this part of the code (which is more related to Qt programming), consult `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 resulting from the above code is shown in the figure below.

Figure 4.16. A Full featured Area Chart



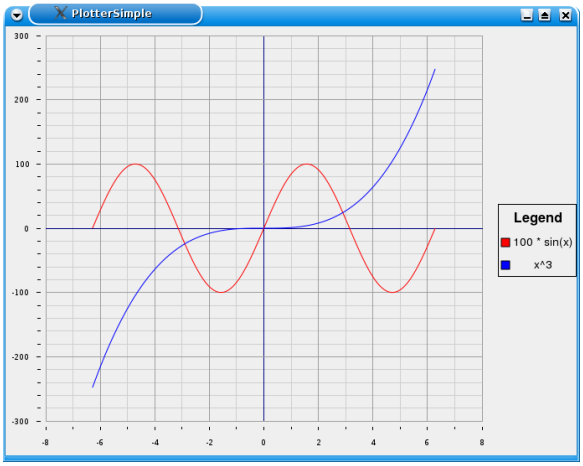
Plotter Charts

Plotter charts are almost the same as normal line diagrams with one important exception: line diagrams always expect values running from 1..n having step width 1; plotters, on the other hand, can handle free X/Y-pairs in any order and do not have to be equidistant.

Therefore, `KDChart::Plotter` expects two columns in a model for each dataset being plotted. We will cover plotter charts more closely in the sample below. Setting attributes is exactly the same. For more information on setting attribues for plotter charts, refer to Section , “Line Charts” in this manual.

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

Figure 4.17. A simple Plotter diagram



Plotter Sample Code

The following code sample plots 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

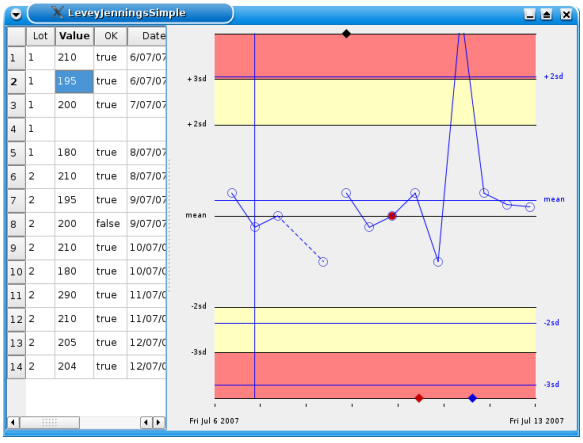
A Levey-Jennings chart graphs quality control data. It shows us, visually, whether a laboratory test is working well or not.

If you are interested in using this diagram type, have a look at the classes `KDChart::LeveyJenningsDiagram` (derived from `KDChart::LineDiagram`) and `KD-`

`Chart::LeveyJenningsAxis` (derived from `KDChart::CartesianAxis`) in the API reference.

The following screenshot shows the Levey-Jennings example in `examples/LeveyJennings/Simple/`

Figure 4.18. A simple Levey-Jennings diagram



The Polar Coordinate Plane

KD Chart uses the Polar coordinate system. The `KDChart::PolarCoordinatePlane` class is used for displaying Pie and Polar chart types.

In this section, we will describe and present each of the chart types that use the Polar coordinate plane.

In general, when implementing a particular type of chart, you must create an object of its type by calling `KDChart::[type]Diagram`. Or, if you are using `KDChart::Widget`, you will need to call `setType()` and specify the appropriate chart type (Widget::Pie, Widget::Polar, etc...).

Pie Charts

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

A good application of a Pie Chart might be to show the distribution of market shares among a group of products or vendors.

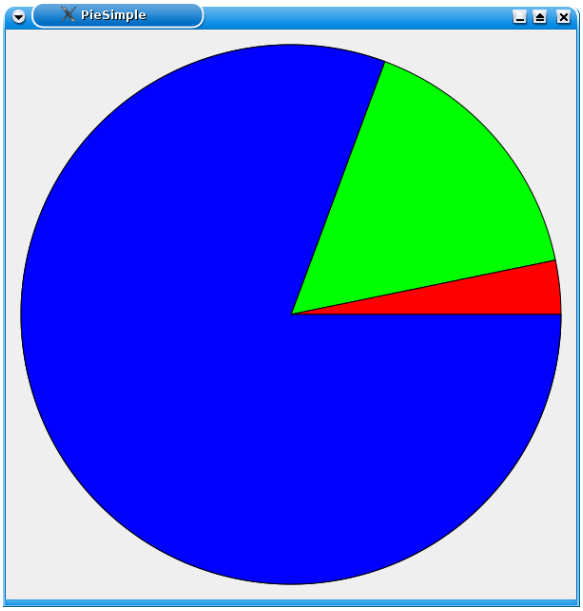
Pie charts typically consist of two or more pieces; any number of which can be shown 'exploded' (shifted away from the center with varying gaps). The starting position of the first pie segment can be specified. To activate pie chart mode, you can either call the `KDChart::Widget` method with `setType( KDChart::Widget::Pie )`; or you can create an object using the `KDChart::PieDiagram` class.

You can also display your pie chart with the three-D look by setting its ThreeD attributes. We will describe this in Chapter 8, *Customizing your Chart* - Section , “ThreeD Attributes” below.

Simple Pie Charts

A simple pie chart shows data without emphasizing a specific item.

Figure 4.19. A Simple Pie Chart



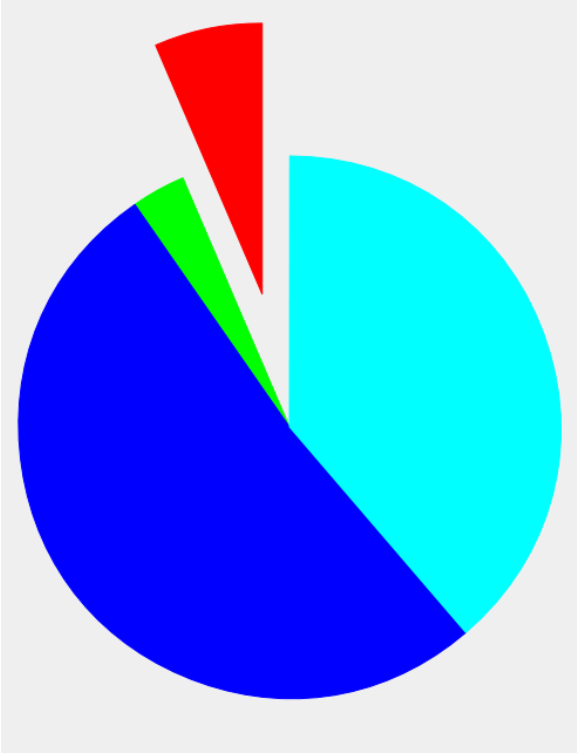
KD Chart by default draws two-dimensional pie charts when in "pie chart mode" (so no method needs to be called to get one). We will describe later, in more detail, how to create the three dimensional look in Section , “Pies Attributes”.

Exploding Pie Charts

Tip

Explode individual segments to emphasize a particular data set.

Figure 4.20. An Exploding Pie Chart



We will go through all the configuration possibilities in Section , “Pies Attributes” below; but first, let's study some sample code.

Code Sample

Let's take a look at a code sample based on the `Simple Widget` we've been using above. see Chapter 3, *Basic steps: Create a Chart* - Section , “Widget Example”. Now, we are going to demonstrate how to configure a Pie diagram and change its attributes when working with `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` so that we can configure the pie chart's 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 need to change the default chart type from Line Chart (the default) to Pie Chart by calling the `KDChart::Widget::setType()` method.

Now, configure a Pen to draw a line around 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 );
```

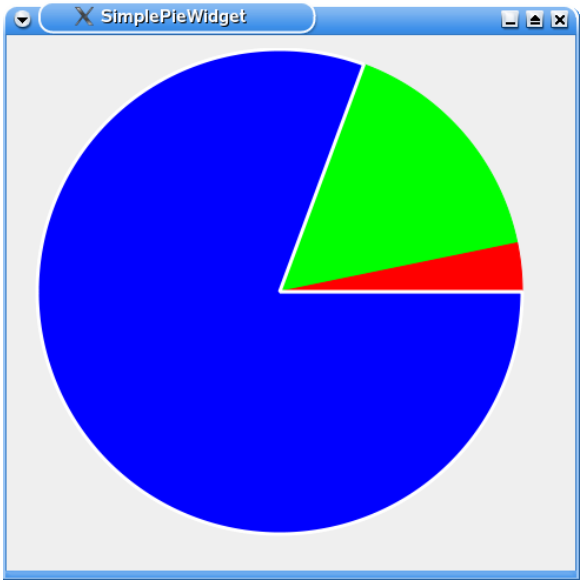
Here, we are configuring the pen "attribute". As you can see, it's pretty straight forward. `KDChart::Widget::pieDiagram()` lets us get a pointer to our widget diagram. To assign the new pen to our diagram, call the diagram `KDChart::AbstractDiagram::setPen()` method.

Finally, we conclude our little example:

```
    widget.show();
    return app.exec();
}
```

See the screenshot below for resulting chart.

Figure 4.21. A Simple Pie Widget



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

> **Note**
>
> Configuring attributes for a `KDChart::PieDiagram` that uses `KDChart::Chart` is done the same way as a `KDChart::Widget`. You just need to assign the configured attributes to your pie diagram, then 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 as well as those specific to the Pie Chart type. KD Chart 2 API separates the attributes specific to a chart type from the generic attributes. Generic attributes are com-

mon to all chart types - for example, the setters and getters for a brush or a pen (See the `KDChart::AbstractDiagram`, `KDChart::PieAbstractDiagram`, etc).

Those attributes that have reasonable default values can simply be modified by the user by calling one of the pie diagram set functions - `KDChart::PieDiagram::setPieAttributes()`.

The procedure is straight forward:

- Create a `KDChart::PieAttributes` object by calling `KDChart::PieDiagram::pieAttributes()`.
- Configure the object using the setters available.
- Assign the object to your Diagram using one of the setters available in `KDChart::PieDiagram`. Every attribute can be configured and applied to the whole diagram, to a column, or to a specified index (`QModelIndex`).

KD Chart 2 supports the attributes listed below for the Pie chart type. In the next section, we will learn how each of those attributes can be set and retrieved.

- Explode: Enable/Disable exploding pie piece(s)
- Explode factor: The explode factor is a qreal between 0 and 1. We read the decimal as a percentage of the total available radius.
- StartPosition: Set the starting angle for the first dataset. It can only be specified for the whole diagram.
- Granularity: Set the granularity: the smaller the granularity, the more your diagram-segments will show facettes instead of rounded segments. It 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 (with 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 factor's default value, use the available methods for assigning 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. This is done in the same way we set the PieAttributes in the code sample above. We will describe that more in detail later in Chapter 8, *Customizing your Chart* - Section , “ThreeD Attributes”.

Pie Attributes Sample

Let's look at some sample code that describes the above process. We recommend that you compile and run the example with us. It is located in the `examples/Lines/Parameters/` directory of your KD Chart installation.

First, we want to include the header files and bring in the KD Chart namespace.

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

using namespace KDChart;
```

We include `KDChartPieAttributes` so that we can configure an exploding pie slice. These attributes are specific to the Pie types.

In the next example, we use a `KDChart::Chart` class and a `QStandardItemModel` to store the data that 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 we have stored our data in the model, we need to replace the default Cartesian plane against a Polar plane. This has to be done 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 similar for all diagram types.

We are now ready to configure our attributes. We want to explode a section of the Pie and configure a Pen to draw an outline around it. Lets begin with the `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, it's `diagram->pieAttributes()`. The second step is to set the attributes with our own values. Then we'll assign it to our diagram. In the code above, we explode the second slice (dataset) in our Pie.

Note

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

We want to configure the Pen to draw a line around the exploded section. This helps focus the reader's attention on that particular dataset.

```
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

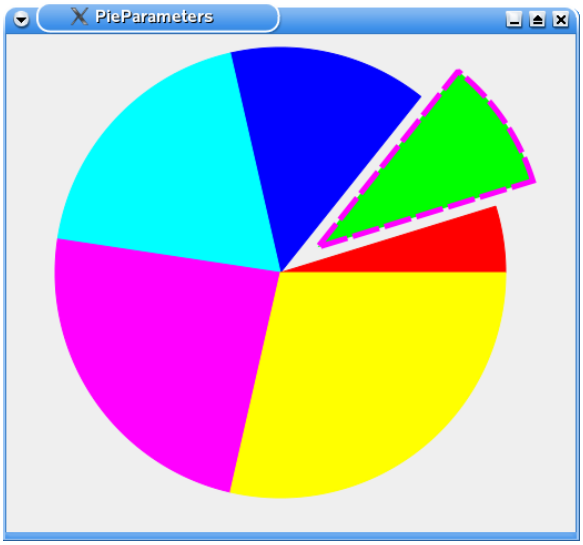
The Pen and the Brush setters and getters are implemented at a lower level in our `KDChart::AbstractDiagram` class to create a cleaner code structure. The AbstractDiagram methods are used by all types of diagrams. Their configuration is simple and straight forward (as you can see in the sample code).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

Once our attributes are configured and assigned, we just need to assign the 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);
```

We can apply the procedure above to any supported attributes for all the chart types. The resulti of the code we have gone through can be seen in the following screenshot. 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 the next section we will be going through some examples that use interesting features offered by the KD Chart 2 API. We go over the code and display a screenshot of 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 a homemade `TableModel` derived from `QAbstractTableModel`.

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

Let's concentrate on our Pie chart implementation and consult the files below. Other needed files - like the ui, pro , qrc ,CSV and main.cpp - can be found in the `examples/Pie/Advanced/` directory of your installation.

```
 1
   #ifndef MAINWINDOW_H
   #define MAINWINDOW_H

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

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

   class MainWindow : public QWidget, private Ui::MainWindow
15 {
       Q_OBJECT

   public:
       MainWindow( QWidget* parent = 0 );
20
   private slots:
       // start position
       void on_startPositionSB_valueChanged( double pos );
       void on_startPositionSL_valueChanged( int pos );
25
       // explode
       void on_explodeSubmitPB_clicked();
       void on_animateExplosionCB_toggled( bool toggle );
```

```
        void setExplodeFactor( int column, double value );

        // animation
        void slotNextFrame();

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

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

        int m_currentFactor;
        int m_currentDirection;
        int m_currentSlice;
    };


#endif /* MAINWINDOW_H */
```

In the code above, we bring up the `KDChart` namespace, as usual, and declare our slots. This allows the user configure 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 with a `KDChart::PolarCoordinatePlane`.

```

    #include "mainwindow.h"

    #include <KDChartChart>
    #include <KDChartPieDiagram>
    #include <KDChartPieAttributes>
    #include <KDChartThreeDPieAttributes>

    #include <QDebug>
    #include <QTimer>

    using namespace KDChart;

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

        QHBoxLayout* chartLayout = new QHBoxLayout( chartFrame );
        m_chart = new Chart();
        m_chart->setGlobalLeadingLeft( 5 );
        m_chart->setGlobalLeadingRight( 5 );
        chartLayout->addWidget( m_chart );
        hSBar->setVisible( false );
        vSBar->setVisible( false );

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

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

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

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

    void MainWindow::on_startPositionSL_valueChanged( int pos )
    {
        double doubleValue = static_cast<double>( pos );
        startPositionSB->blockSignals( true );
        startPositionSB->setValue( doubleValue  );
        startPositionSB->blockSignals( false );
        static_cast<PolarCoordinatePlane*>( m_chart->coordinatePlane()
                                          )->setStartPosition( pos );
        m_chart->update();
    }

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

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

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

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

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

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

```
    }

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

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

```

First, we add our chart to the layout as we would any other Qt widget. Then we 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 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 then 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 the 3D-mode display and the 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();
```

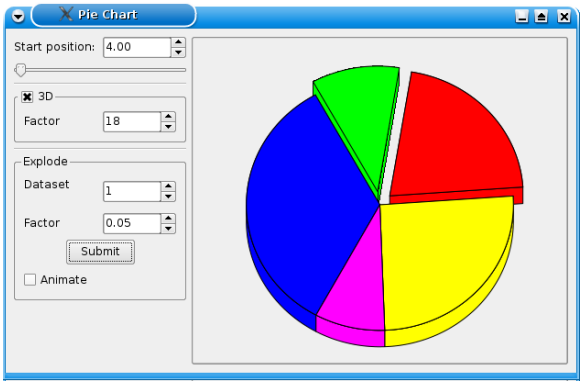
We want the user to be able to explode one or several slices (datasets). We also want to allow the user to configure the exploding factor. We implement this by configuring our pie attributes and assigning them by dataset to the diagram, as shown below.

```
....
// 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();
...
```

We can use the same procedure to run our animation. To learn more about coding the animation, which is more related to Qt programming, consult `examples/Pie/Advanced/mainwindow.cpp`.

You can compile and run this example from the `examples/Pie/Advanced/` directory in your KD Chart installation. The figure below shows the widget we created in the code example.

Figure 4.23. A Full featured Pie Chart



Polar Charts

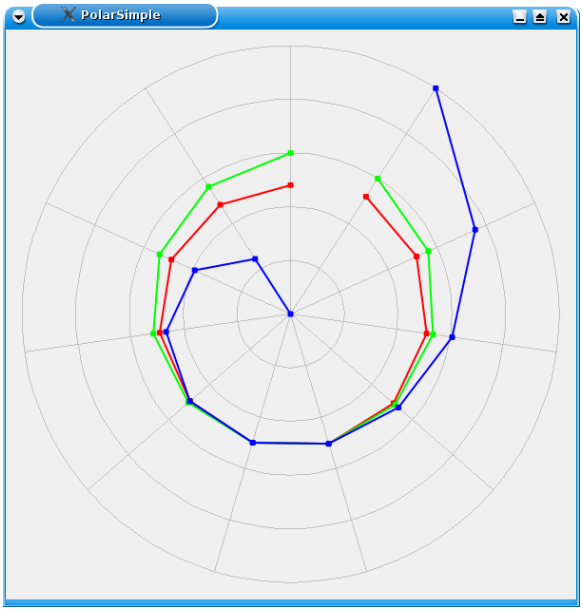
Polar charts get their name from displaying "polar coordinates" instead of Cartesian coordinates. They use the `KDChart::PolarCoordinatePlane`.

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

A Simple Polar Chart

Compile and run the example file in `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 rotated automatically or not. If set, the labels' base lines will be adjusted in reference to the circular grid lines.
- `setCloseDatasets( bool )` may be used to close each of the data series by connecting the last points to thier respective start points.

The `KDChart::PolarCoordinatePlane` provides an additional means of configuration that may 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 a circular direction (or in sagittal direction, resp.).

  For 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 demonstrate 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, its not possible to display true world map data since you can not specify each cell's rotation angle. Transforming your coordinates to the Cartesian system and using a Point Chart may be a solution in such cases.

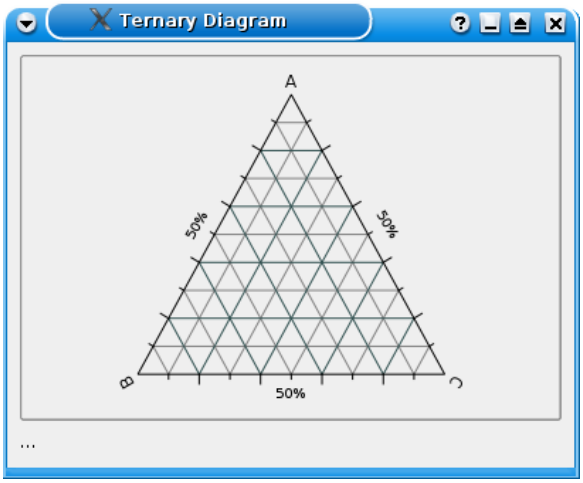
Ternary Coordinate Plane

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

The idea of a ternary chart is to plot triple values on a triangle. Triple values are represented by three floating point values totalling the fixed sum 1.0. Each plotted dataset

needs three columns in the model.

Figure 4.25. A Simple Ternary Chart



Tip

KD Chart uses the first two values to calculate the third. If the sum of the first two columns is already greater than 1.0, the data triple is considered invalid and disregarded.

This section describes the chart types that can be added to a ternary coordinate plane.

To use the diagram, create an instance of `KDChart::TernaryCoordinatePlane`. Then you can make KD Chart use it by calling `KDChart::Chart::replaceCoordinatePlane()` and adding the diagram to it.

Ternary Line Charts

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

The following code example explains 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 uses `KDChart::CartesianAxis` and `KDChart::TernaryAxis`. Both are derived from their common base class: `KDChart::AbstractAxis`.

The user may specify his own set of strings to use 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 reset it to the default behaviour.

For convenience, we can also specify short labels in our own set of strings. These can be used as axis labels if normal labels are too long. Use `KDChart::AbstractAxis::setShortLabels( const QStringList ).`

General text attributes for axis values and labels may also be configured. This way, the labels of all of your axes, in all of your diagrams (within the Chart), can be drawn in a default font size.

The setters and getters, for axis labels and their text attributes, are implemented in the axis base class `KDChart::AbstractAxis`. For an in depth look, 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.
>
> For example, you could specify the seven days of the week as abscissa labels. Then those labels could be repeatedly used.

Cartesian Axis

The class `KDChart::CartesianAxis` is used along with displayed diagrams in a cartesian coordinate plane and it contains the setters and getters related to the axis specifics to those chart types.

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

The setters and getters for specific cartesian features are implemented in `KDChart::CartesianAxis`.

Ternary Axis

Use the `KDChart::TernaryAxis` class for 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

To add axis to a cartesian diagram, we need to use the `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's look at the following lines of code based on the `Simple Widget` we have be working with above (see Chapter 3, *Basic steps: Create a Chart* - Section , “Widget Example”). In this example, we will demonstrate how to add an X axis, and a Y axis, to your diagram. We will also set the Axis titles in 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` so that we can 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 Chart) by calling the `KDChart::Widget::setType()` method.

Now, let's 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):

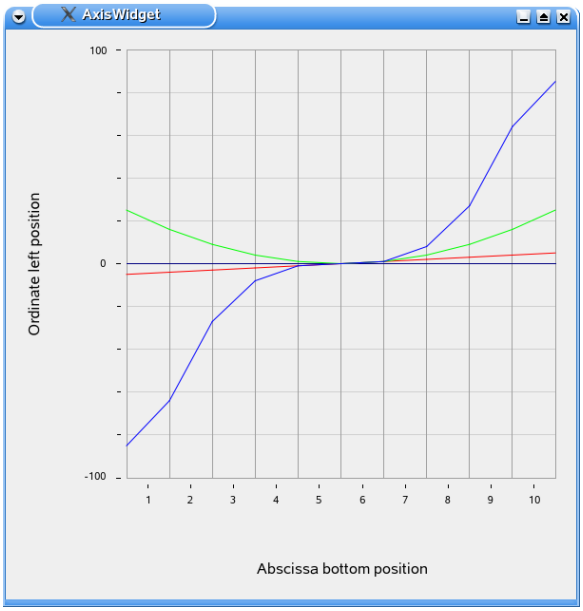
```
    widget.lineDiagram()->addAxis( xAxis );
    widget.lineDiagram()->addAxis( yAxis );
```

Finally, we conclude our small example:

```
    widget.show();

    return app.exec();
}
```

See the screenshot below for the resulting chart:

Figure 5.1. A Simple Widget With Axis



This example may be compiled and run from the following location of your KD Chart installation: `examples/Axis/Widget/`

In Section , “Tips”, we will present you a more elaborate example that uses `KDChart::Chart`. We will configure our axis title text attributes as well as create our own labels (and their shortened version).

Tips

In this section, we want to give you some examples showing interesting features offered by the KD Chart 2 API. We will study some code and show a screenshot of 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
   #ifndef MAINWINDOW_H
   #define MAINWINDOW_H

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

   namespace KDChart {
       class Chart;
10     class BarDiagram;
   }

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

   public:
       MainWindow( QWidget* parent = 0 );

20
   private:
       KDChart::Chart* m_chart;
       TableModel m_model;
       KDChart::BarDiagram* m_lines;
25 };


   #endif /* MAINWINDOW_H */

30
```

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
   #include "mainwindow.h"

   #include <KDChartChart>
 5 #include <KDChartBarDiagram>
   #include <KDChartTextAttributes>
   #include <KDChartRulerAttributes>
   #include <KDChartFrameAttributes>

10 using namespace KDChart;

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

       QHBoxLayout* chartLayout = new QHBoxLayout( chartFrame );
```

```
        m_chart = new Chart();
        m_chart->setGlobalLeading( 10,  10,  10,  10 );
        chartLayout->addWidget( m_chart );
        hSBar->setVisible( false );
        vSBar->setVisible( false );

        m_model.loadFromCSV( ":/data" );

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

        // create and position axis
        CartesianAxis *topAxis = new CartesianAxis( m_lines );
        CartesianAxis *leftAxis = new CartesianAxis ( m_lines );
        RulerAttributes rulerAttr = topAxis->rulerAttributes();
        rulerAttr.setTickMarkPen( 0.9999999, QPen( Qt::red ) );
        rulerAttr.setTickMarkPen( 2.0, QPen( Qt::green ) );
        rulerAttr.setTickMarkPen( 3.0, QPen( Qt::blue ) );
        rulerAttr.setShowMinorTickMarks(true);
        //rulerAttr.setShowMajorTickMarks(false);
        topAxis->setRulerAttributes( rulerAttr );
        CartesianAxis *rightAxis = new CartesianAxis ( m_lines );
        CartesianAxis *bottomAxis = new CartesianAxis ( m_lines );
        topAxis->setPosition ( CartesianAxis::Top );
        leftAxis->setPosition ( CartesianAxis::Left );
        rightAxis->setPosition ( CartesianAxis::Right );
        bottomAxis->setPosition ( CartesianAxis::Bottom );

    // set the margin that should be used between the displayed labels and the ticks to zero
    #if 0
        RulerAttributes ra = bottomAxis->rulerAttributes();
        ra.setLabelMargin(0);
        bottomAxis->setRulerAttributes( ra );
    #endif

    // show a red frame around the bottom axis
    #if 0
        FrameAttributes fa( bottomAxis->frameAttributes() );
        fa.setPen( QPen(QBrush(QColor("#ff0000")),1.0) );
        fa.setVisible( true );
        bottomAxis->setFrameAttributes( fa );
    #endif

        // set axis titles
        topAxis->setTitleText ( "Abscissa color configured top position" );
        leftAxis->setTitleText ( "left Ordinate: fonts configured" );
        rightAxis->setTitleText ( "right Ordinate: default settings" );
        bottomAxis->setTitleText ( "Abscissa Bottom" );
        topAxis->setTitleSize(1.1);
        topAxis->setTitleSpace(.2);

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

        TextAttributes taLeft ( leftAxis->titleTextAttributes () );
        taLeft.setRotation( 180 );
        Measure me( taLeft.fontSize() );
        me.setValue( me.value() * 0.8 );
        taLeft.setFontSize( me );

    // Set the following to 1, to hide the left axis title
    //  - no matter if a title text is set or not
    #if 0
        taLeft.setVisible( false );
    #endif
        leftAxis->setTitleTextAttributes ( taLeft );

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

        // configure labels text attributes
```

```
        TextAttributes taLabels( topAxis->textAttributes() );
        taLabels.setPen(  QPen( Qt::darkGreen ) );
 95     taLabels.setRotation( 90 );
        topAxis->setTextAttributes( taLabels );
        leftAxis->setTextAttributes( taLabels );
        bottomAxis->setTextAttributes( taLabels );

100
    // Set the following to 0, to see the default Abscissa labels
    // (== X headers, as read from the data file)
    #if 1
        // configure labels and their shortened versions
105     QStringList daysOfWeek;
        daysOfWeek << "M O N D A Y" << "Tuesday" << "Wednesday"
                   << "Thursday" << "Friday" ;
        topAxis->setLabels( daysOfWeek );

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

115     QStringList bottomLabels;
        bottomLabels << "Team A" << "Team B" << "Team C";
        bottomAxis->setLabels( bottomLabels );

        QStringList shortBottomLabels;
120     shortBottomLabels << "A" << "B";
        bottomAxis->setShortLabels( shortBottomLabels );
    #endif

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

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

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 set-

ShortLabels()).

```
// 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

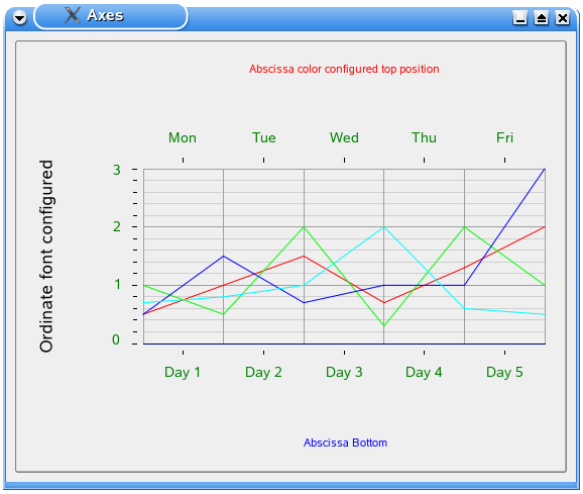
Labels specified via setLabels take precedence: if a non-empty list is passed, KD Chart will use these strings as axis labels, instead of calculating them.

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* - Section , “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 `KDChart-BarDiagram`. 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 );
```

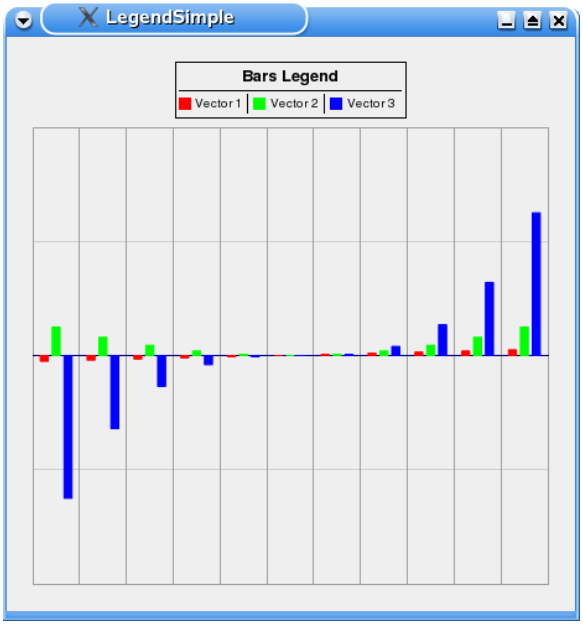
The interesting point here is how we call `KDChart::Widget::firstlegend()` to get a pointer to to our legend object and be able to set up and configure it. We will see further on in the next code example, see Section , “Tips” how to configure the elements of a legend (e.g Title text, markers, etc.).

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

```
    widget.show();

    return app.exec();
}
```

See the 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 Section , “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 );
```

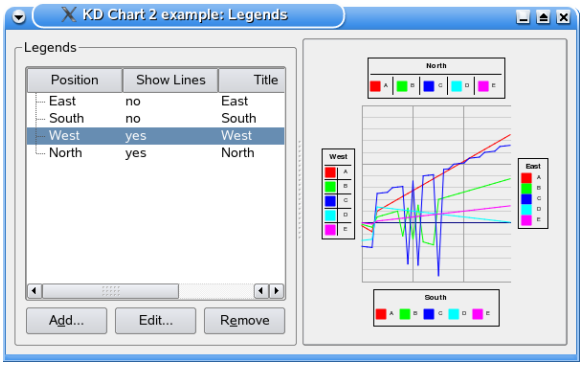
Same procedure as above. Please note the `setVisible()` method which is necessary as the default value hides those attributes.

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



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* - Section , "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 dia-

gram.

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 );
```

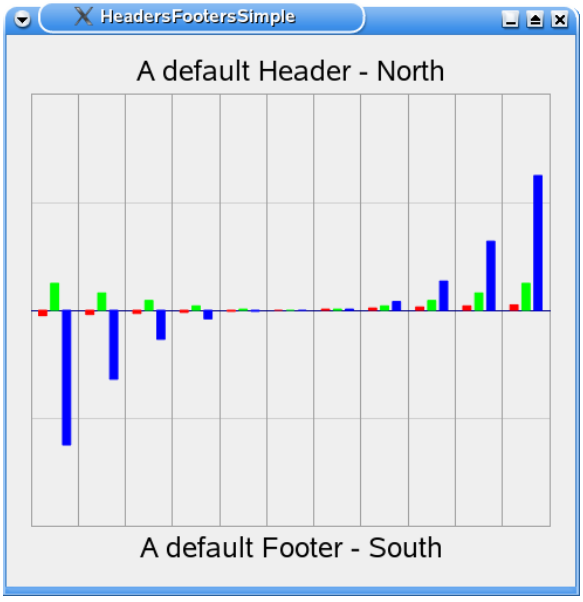
As you can see the code above is straight forward and we just need to call `KDChart::Widget::addHeaderFooter()` passing the text, type and position we want to assign to it.

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 Section , “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

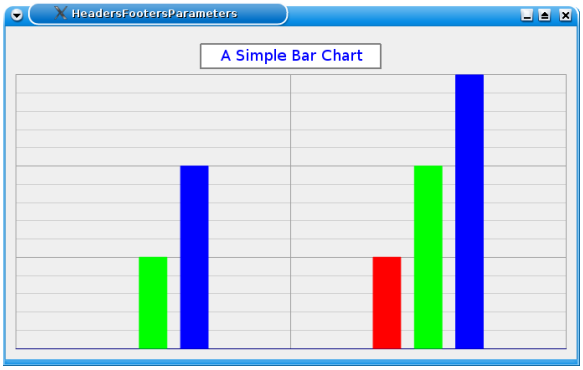
Same procedure as above. Please note the `setVisible()` method which is necessary as the default value hides the attributes above.

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.

```
 1
   #ifndef MAINWINDOW_H
   #define MAINWINDOW_H

 5 #include <QDialog>
   #include <QMap>

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

   namespace KDChart {
       class Chart;
       class DatasetProxyModel;
15     class LineDiagram;
   }

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

   public:
       MainWindow( QWidget* parent = 0 );

25 private slots:
       void on_addHeaderPB_clicked();
       void on_editHeaderPB_clicked();
       void on_removeHeaderPB_clicked();
       void on_headersTV_itemSelectionChanged();
30
   private:
       void setupAddHeaderDialog( QDialog* dlg,
                                  Ui::AddHeaderDialog& conf ) const;

35     KDChart::Chart* m_chart;
       TableModel m_model;
       KDChart::DatasetProxyModel* m_datasetProxy;
       KDChart::LineDiagram* m_lines;
   };
40

   #endif /* MAINWINDOW_H */
```

```
45
```

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`).

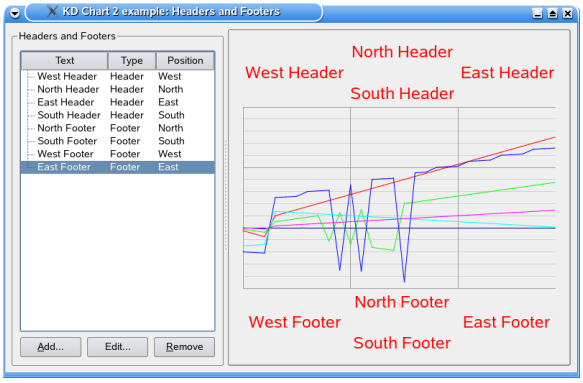
```
 1
   #include "mainwindow.h"

   #include <KDChartChart>
 5 #include <KDChartHeaderFooter>
   #include <KDChartPosition>
   #include <KDChartCartesianCoordinatePlane>
   #include <KDChartLineDiagram>
   #include <KDChartTextAttributes>
10 #include <KDChartDatasetProxyModel>
   #include <QComboBox>
   #include <QLineEdit>
   #include <QPen>

15 class HeaderItem : public QTreeWidgetItem
   {
   public:
       HeaderItem( KDChart::HeaderFooter* header, QTreeWidget* parent ) :
           QTreeWidgetItem( parent ), m_header( header ) {}
20
       KDChart::HeaderFooter* header() const { return m_header; }

   private:
       KDChart::HeaderFooter* m_header;
25 };

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

       QHBoxLayout* chartLayout = new QHBoxLayout( chartFrame );
       m_chart = new KDChart::Chart();
       chartLayout->addWidget( m_chart );
35
       m_model.loadFromCSV( ":/data" );

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

       m_chart->update();
   }
45

   void MainWindow::setupAddHeaderDialog( QDialog* dlg,
                                          Ui::AddHeaderDialog& conf )const
   {
50     conf.setupUi( dlg );
       conf.textED->setFocus();

       // Note: Header/Footer position can be Center but it can not be Floating
       const QStringList labels = KDChart::Position::printableNames( KDChart::Position::IncludeC
55     const QList<QByteArray> names = KDChart::Position::names( KDChart::Position::IncludeCente

       for ( int i = 0, end = qMin( labels.size(), names.size() ) ;
             i != end ;
             ++i )
60         conf.positionCO->addItem( labels[i], names[i] );
   }


   void MainWindow::on_addHeaderPB_clicked()
65 {
       QDialog dlg;
```

```
        Ui::AddHeaderDialog conf;
        setupAddHeaderDialog( &dlg, conf );
        conf.typeCO->setCurrentIndex( 0 ); // let us start with "Header"
    70  conf.positionCO->setCurrentIndex( 0 );
        if( dlg.exec() ) {
            KDChart::HeaderFooter* headerFooter
                    = new KDChart::HeaderFooter( m_chart );
            m_chart->addHeaderFooter( headerFooter );
    75      headerFooter->setText( conf.textED->text() );
            KDChart::TextAttributes attrs( headerFooter->textAttributes() );
            attrs.setPen( QPen( Qt::red ) );
            headerFooter->setTextAttributes( attrs );
            headerFooter->setType(
    80          conf.typeCO->currentText() == "Header"
                ? KDChart::HeaderFooter::Header
                : KDChart::HeaderFooter::Footer );
            headerFooter->setPosition(
                KDChart::Position::fromName( conf.positionCO->itemData(
    85              conf.positionCO->currentIndex() ).toByteArray() ) );
            //headerFooter->show();
            HeaderItem* newItem = new HeaderItem( headerFooter, headersTV );
            newItem->setText( 0, conf.textED->text() );
            newItem->setText( 1, headerFooter->type()
    90                           == KDChart::HeaderFooter::Header
                                 ? tr("Header")
                                 : tr("Footer") );
            newItem->setText( 2, conf.positionCO->currentText() );
            m_chart->update();
    95  }
    }


    void MainWindow::on_editHeaderPB_clicked()
   100 {
        if ( headersTV->selectedItems().size() == 0 ) return;
        HeaderItem* item =
                static_cast<HeaderItem*>( headersTV->selectedItems().first() );
        KDChart::HeaderFooter* headerFooter = item->header();
   105  QDialog dlg;
        Ui::AddHeaderDialog conf;
        setupAddHeaderDialog( &dlg, conf );
        conf.textED->setText( headerFooter->text() );
        conf.typeCO->setCurrentIndex(
   110      headerFooter->type() == KDChart::HeaderFooter::Header
                ? 0 : 1 );
        conf.positionCO->setCurrentIndex(
            conf.positionCO->findText(headerFooter->position().printableName()));
        if( dlg.exec() ) {
   115      headerFooter->setText( conf.textED->text() );
            headerFooter->setType(
                conf.typeCO->currentText() == "Header"
                ? KDChart::HeaderFooter::Header
                : KDChart::HeaderFooter::Footer );
   120      headerFooter->setPosition(
                KDChart::Position::fromName( conf.positionCO->itemData(
                    conf.positionCO->currentIndex() ).toByteArray() ) );
            item->setText( 0, conf.textED->text() );
            item->setText( 1, headerFooter->type()
   125                        == KDChart::HeaderFooter::Header
                               ? tr("Header")
                               : tr("Footer") );
            item->setText( 2, conf.positionCO->currentText() );
            m_chart->update();
   130  }
    }



   135
    void MainWindow::on_removeHeaderPB_clicked()
    {
        if ( headersTV->selectedItems().size() == 0 ) return;
        QList<QTreeWidgetItem*> items = headersTV->selectedItems();
   140  for( QList<QTreeWidgetItem*>::const_iterator it = items.begin();
             it != items.end(); ++it )
```

```
        {
            KDChart::HeaderFooter* headerFooter
                    = static_cast<HeaderItem*>( (*it) )->header();
145 #if 0
            // Note: Despite it being owned by the Chart, you *can* just
            //       delete the header: KD Chart will notice that and
            //       it will adjust its layout ...
            delete headerFooter;
150 #else
            // ... but the correct way is to first take it, so the Chart
            // is no longer owning it:
            m_chart->takeHeaderFooter( headerFooter );
            // ... and then delete it:
155         delete headerFooter;
    #endif
            delete (*it);
        }
        m_chart->update();
160 }


    void MainWindow::on_headersTV_itemSelectionChanged()
    {
165     removeHeaderPB->setEnabled( headersTV->selectedItems().count() > 0 );
        editHeaderPB->setEnabled( headersTV->selectedItems().count() == 1 );
    }
```

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

Figure 7.3. Headers and Footers advanced example



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

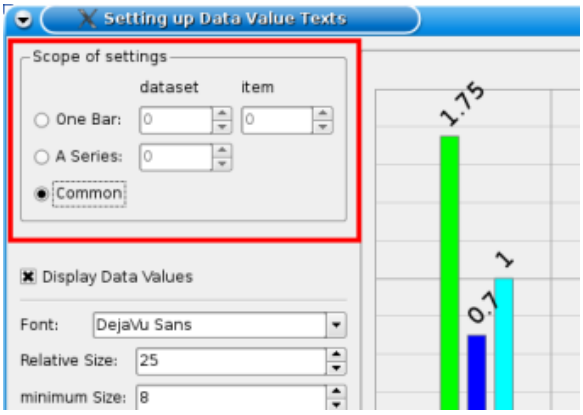
It is important to know that have three levels of precedence when setting the attributes:

- 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



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 2.4 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.

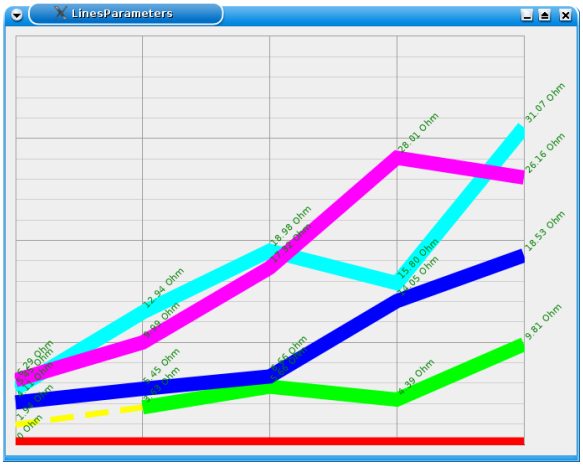
```
....
// Display values
// 1 - Call the relevant attributes
DataValueAttributes dva( diagram->dataValueAttributes() );

// 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

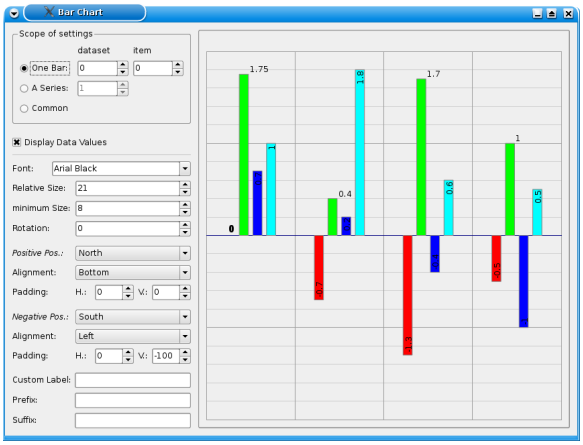
If you are interested in more details on positioning and/or customizing your data labels, have a look at the example `examples/DataValueTexts/`.

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



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.

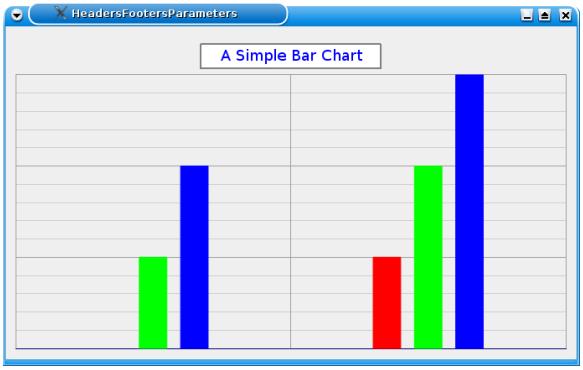
```
....
// Configure the Header text attributes
TextAttributes hta( header->textAttributes() );
hta.setPen( QPen(  Qt::blue ) );

// 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.

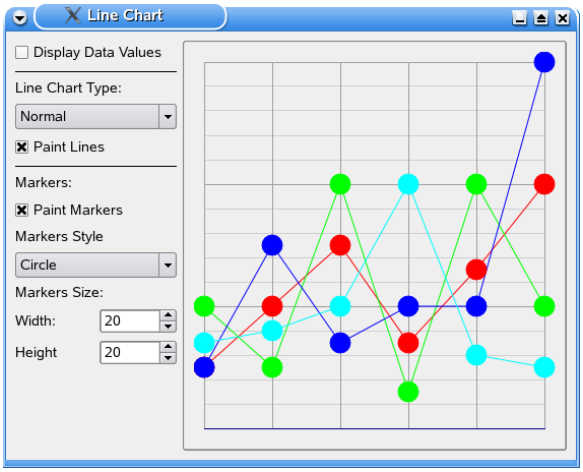
```
// set up a map with different marker styles
MarkerAttributes::MarkerStylesMap map;
map.insert( 0, MarkerAttributes::MarkerSquare );
map.insert( 1, MarkerAttributes::MarkerCircle );
map.insert( 2, MarkerAttributes::MarkerRing );
map.insert( 3, MarkerAttributes::MarkerCross );
....
// Configure markers 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



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

Note

To change a marker's *color* please use the following special logic as shown in file `examples/Polar/Parameters/main.cpp`:

```
const QModelIndex index = diagram->model()->index( 1, 2, QModelIndex() );
```

```
DataValueAttributes dva( diagram->dataValueAttributes( index ); );
MarkerAttributes ma( dva.markerAttributes() );

// This is the canonical way to adjust a marker's color:
// By default the color is invalid so we use an explicit fallback
// here to make sure we are getting the right color, as it would
// be used by KD Chart's built-in logic too:

QColor semiTrans( ma.markerColor() );

if( ! semiTrans.isValid() )
    semiTrans = diagram->brush( index ).color();

semiTrans.setAlpha(164);
ma.setMarkerColor( semiTrans );
dva.setMarkerAttributes( ma );

diagram->setDataValueAttributes( index, dva);
```

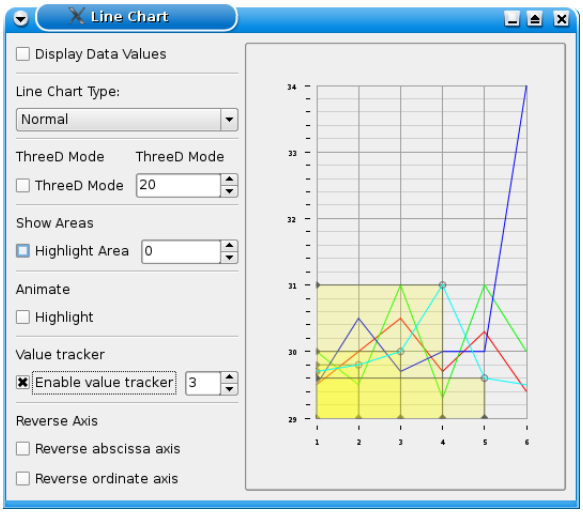
Value Tracker Attributes

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



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

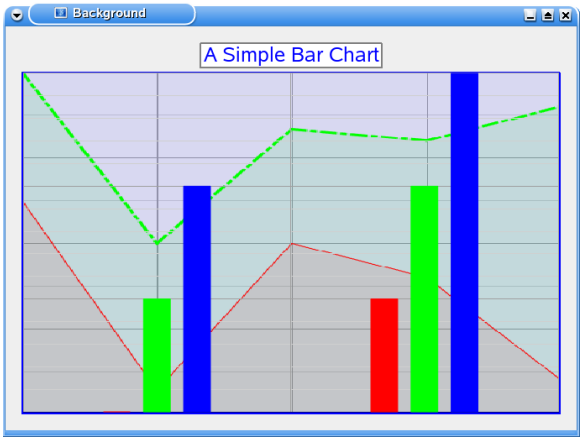
Let us make this more concrete by looking at the following lines of code which describes the above process. This example is based on the `main.cpp` file of the `examples/Background/`. We recommend you compile and run this example and to study its code.

```
....
// 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.

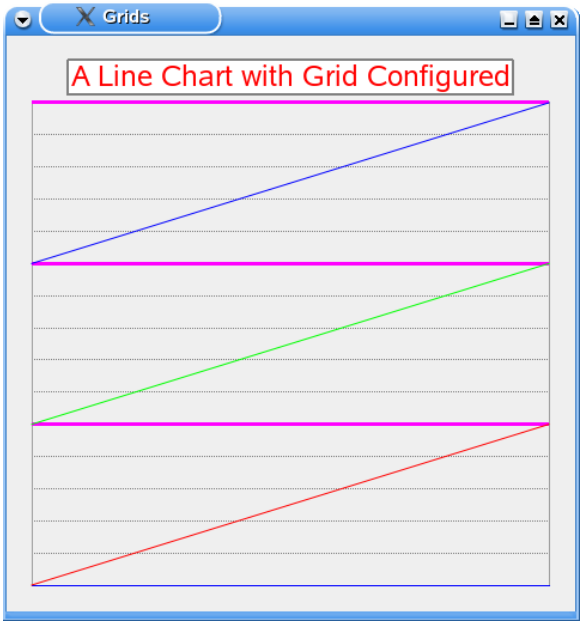
```
// diagram->coordinatePlane returns an abstract plane.
// if we want to specify the orientation we need to cast
// as follow
CartesianCoordinatePlane* plane =
    static_cast <CartesianCoordinatePlane*>
        ( diagram->coordinatePlane() );

// 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

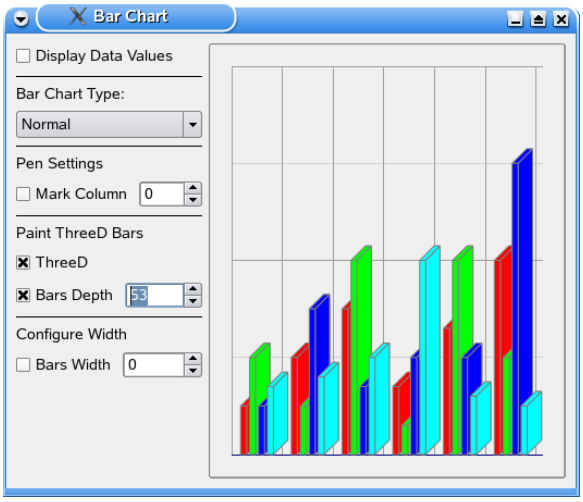
Let us make this more concrete by looking at the following lines of code which describe the above process. This example is based on the `mainwindow.cpp` file of the `examples/Bars/Advanced/`. We recommend you compile and run this example and to study its code.

```
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



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 Section , “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 attrs
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 or 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

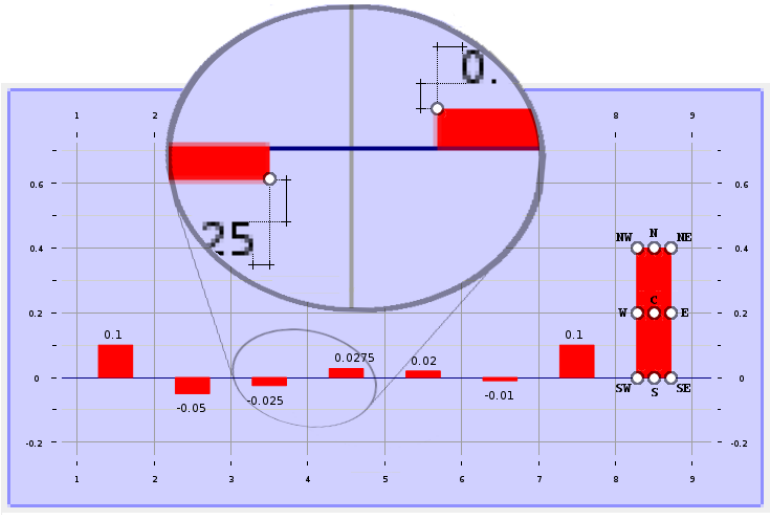
This chapter covers the `KDChart::Position` and `KDChart::RelativePosition` classes. For details on the closely related `KDChart::Measure` class see the preceeding Section , “Font Sizes and other Measures”.

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.

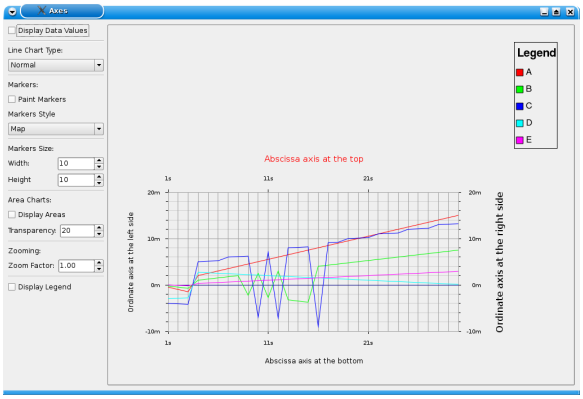
Chapter 9. Advanced Charting

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

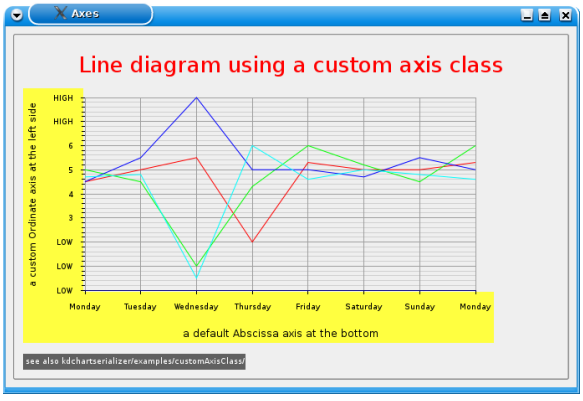
1 - `/examples/Axis/Parameters`

Figure 9.1. /examples/Axis/Parameters



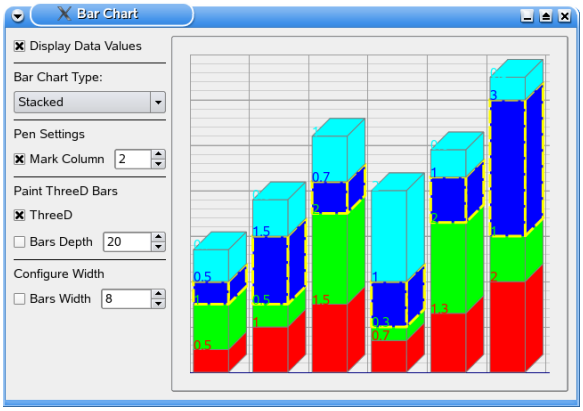
2 - `/examples/Axis/Labels`

Figure 9.2. /examples/Axis/Labels



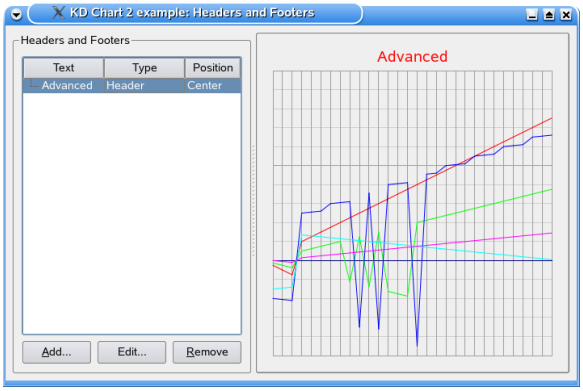
3 - `/examples/Bars/Advanced`

Figure 9.3. /examples/Bars/Advanced



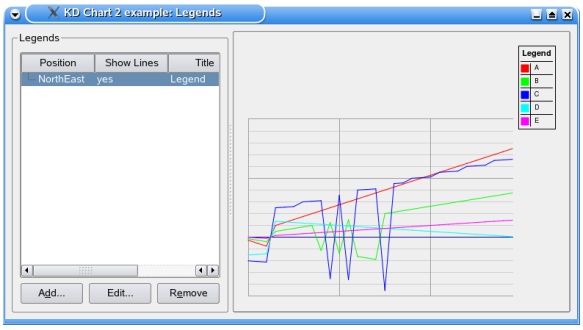
4 - `/examples/HeadersFooters/HeadersFooters/Advanced`

Figure 9.4. /examples/HeadersFooters/HeadersFooters/Advanced



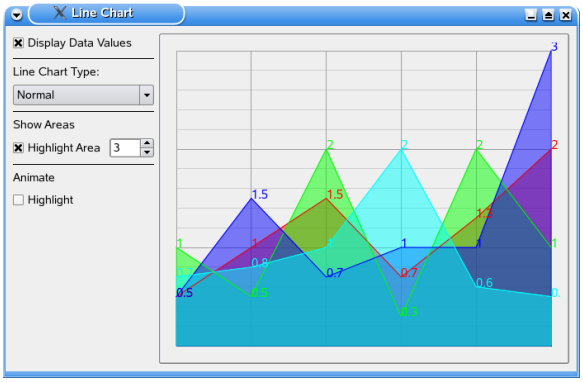
5 - `/examples/Legends/LegendAdvanced`

Figure 9.5. /examples/Legends/LegendAdvanced



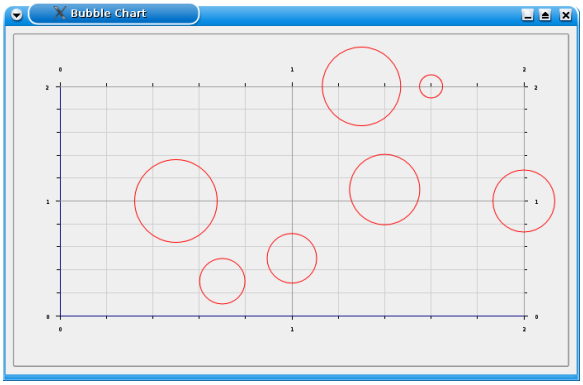
6 - `/examples/Lines/Advanced`

Figure 9.6. /examples/Lines/Advanced



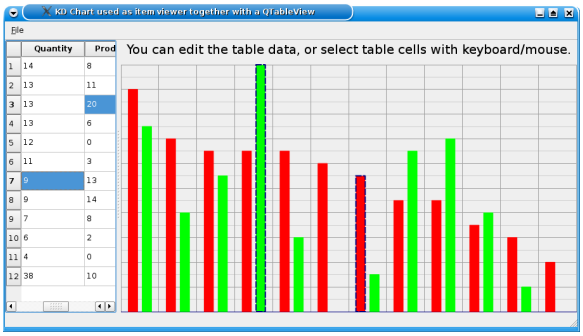
7 - `/examples/Plotter/BubbleChart`

Figure 9.7. /examples/Plotter/BubbleChart



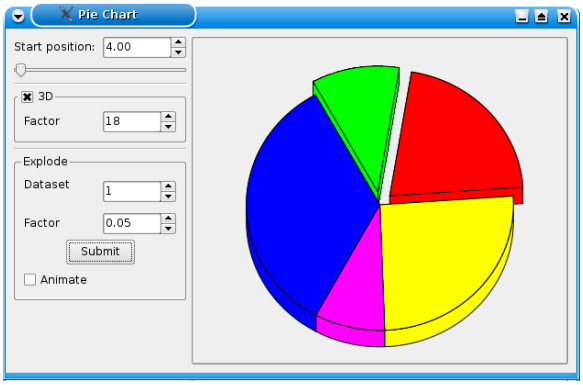
8 - `/examples/ModelView/TableView`

Figure 9.8. /examples/ModelView/TableView



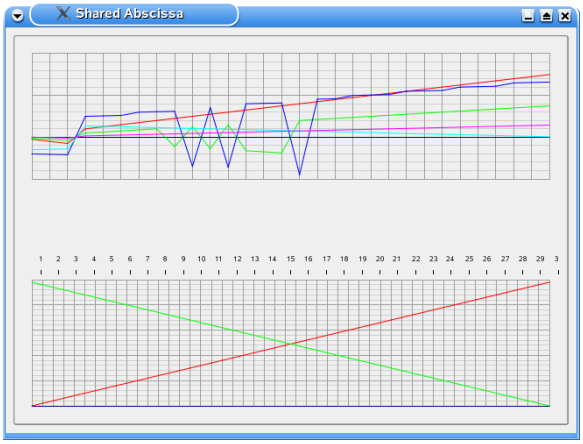
9 - `/examples/Pie/Advanced`

Figure 9.9. /examples/Pie/Advanced



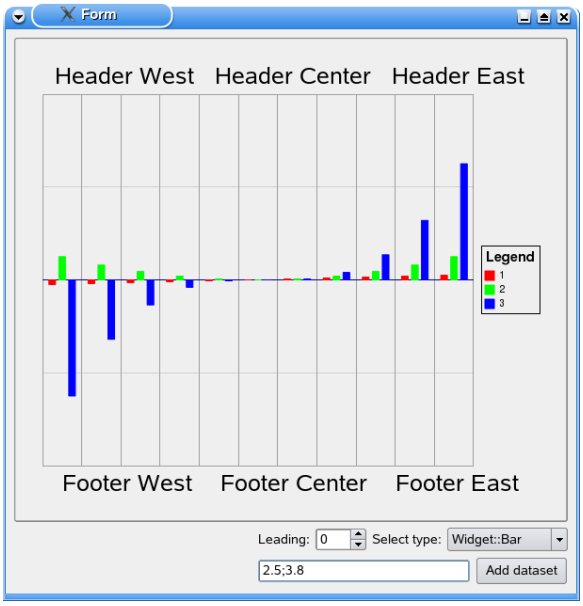
10 - `/examples/SharedAbscissa`

Figure 9.10. /examples/SharedAbscissa



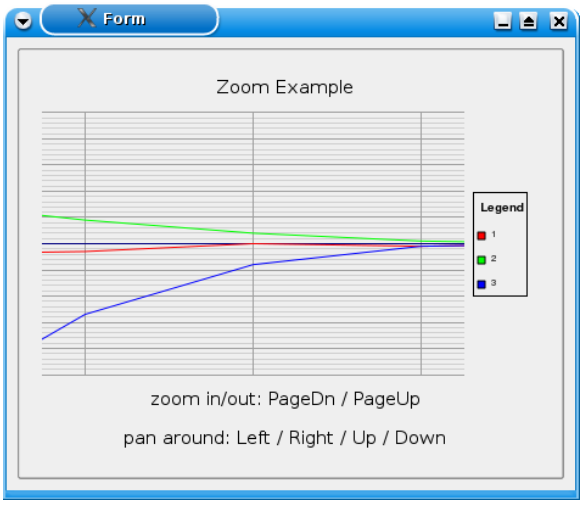
11 - `/examples/Widget/Advanced`

Figure 9.11. /examples/Widget/Advanced



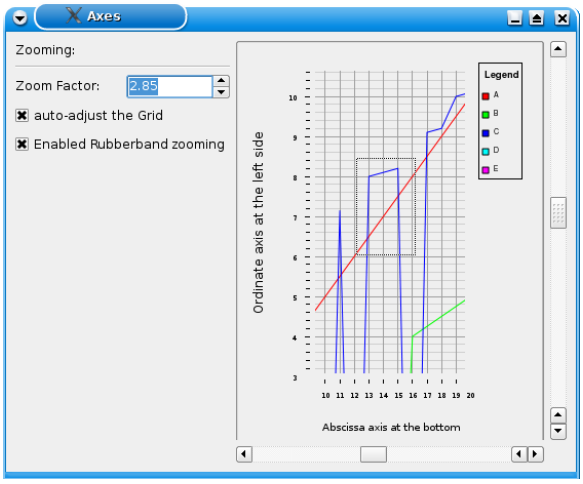
12 - `/examples/Zoom/Keyboard`

Figure 9.12. /examples/Zoom/Keyboard



13 - `/examples/Zoom/ScrollBars`

Figure 9.13. /examples/Zoom/ScrollBars



Chapter 10. Gantt Charts

A Gantt chart is a horizontal bar chart used to schedule and track different tasks, for example, a software project.

It is constructed with a horizontal axis showing a timeline which can be divided into several smaller parts, such as hours, days, weeks, etc. A Gantt chart also has a vertical axis containing the different tasks included in the project. These tasks can be sub-items to other tasks or simply stand-alone tasks.

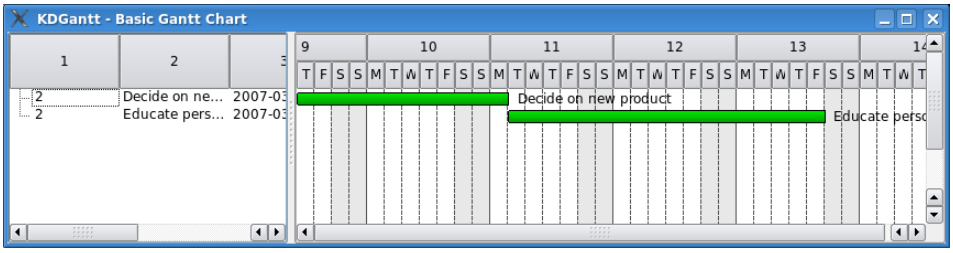
Gantt Chart Examples

The following two Gantt charts show a few of the possibile uses of KD Chart. A more detailed description of the parts shown in these examples can be found in later chapters.

The complete source code for these two examples will be shown in the next chapter, where we describe more closely how they work.

A Basic Gantt Chart

Figure 10.1. A Basic Gantt Chart



This is a screenshot of an application running KD Chart. As you can see, the main widget is separated into two views — one view containing the tasks, and the other view containing the actual schedule with the timeline.

The left view in the screenshot above contains, as stated, all the scheduled tasks. All tasks have a name, type, a start date and an end date. Some tasks, such as milestones, only have one date, because these are not actionable tasks, but rather a note on certain events, such as a delivery.

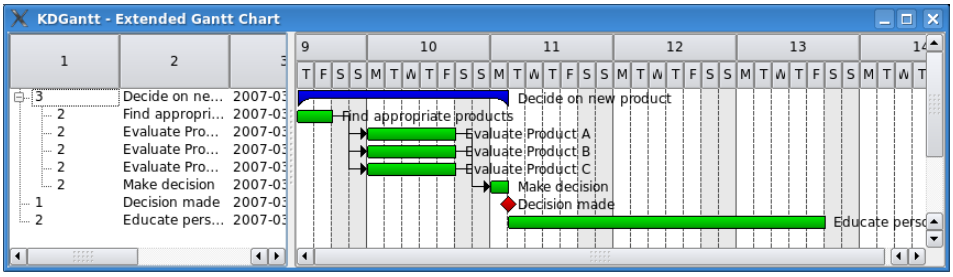
In the right view, we have the actual schedule. In this simple chart, we only have two tasks which are performed sequentially.

A Gantt chart is often better the more verbose it is. In the following section, we will extend the simple chart we have seen so far to show more details about the performed

tasks.

An Extended Gantt Chart

Figure 10.2. An Extended Gantt Chart



In the above screenshot, you can see examples of subtasks as well as task links. We have also changed the type of the first task to a KDGantt::TypeSummary. Different item types are described further in Section , “Basic Usage, Working With Items”.

The task links used in the Gantt chart above show dependencies between the tasks. You cannot evaluate the different products until you know which products to evaluate, and that is discovered during the "Find appropriate products" task. Also, you cannot decide on which product to use until all alternatives are evaluated. The different usage alternatives for task links are shown in more detail in Section , “Working With Constraints” where we will even show a few code examples on how you use them.

We have added an event item as well, showing when the actual decision event should take place. This is an object of type KDGantt::TypeEvent, with the start time set to the end time of the "Make decision"-task.

Your First Own Gantt Chart

Now we will have a look at how to create a first basic Gantt chart, first outlining the general procedure, then later showing a code example.

The creation of a Gantt chart is a very easy and straightforward procedure:

1. Create an object of type `KDGantt::View`. This is the actual Gantt view, consisting of the tree view to the left and the schedule on the right. From this object you can access, apart from the tree view and the schedule, the underlying model that stores the data, as well as the constraint model and the grid.

2. Create a model, and add items to this model. These items are the entries in the Gantt chart, consisting of a type, name, start date, end date and, optionally, comple-

tion percentage. There are five different types to choose among: KDGantt::TypeNone, KDGantt::TypeEvent, KDGantt::TypeTask, KDGantt::TypeSummary and KDGantt::TypeUser. These five types are described in more detail in Section , “Basic Usage, Working With Items”. You will need to set the start and end times of these items, to have them displayed in the Gantt view. How to do this will be explained in the following examples.

Examples

To make the discussion more practical, we will now show you a few code examples. The output of these examples is shown in Figure 10.1 and Figure 10.2.

A Basic Gantt Chart

The following code is also available in a complete example in `step01a.cpp`.

```
 1
   #include <QApplication>
   #include <QStandardItemModel>

 5 #include <KDGanttView>
   #include <KDGanttDateTimeGrid>

   class MyStandardItem : public QStandardItem {
   public:
10     MyStandardItem( const QVariant& v ) : QStandardItem()
       {
           setData( v, Qt::DisplayRole );
       }
       MyStandardItem( const QString& v ) : QStandardItem()
15     {
           setData( v, Qt::DisplayRole );
       }
   };

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

       QStandardItemModel model;❶
25     model.appendRow( QList<QStandardItem*>()❷
                        << new MyStandardItem( QString( "Decide on new product" ) )
                        << new MyStandardItem( KDGantt::TypeTask )
                        << new MyStandardItem( QDateTime( QDate( 2007, 3, 1 ) ) )
                        << new MyStandardItem( QDateTime( QDate( 2007, 3, 13 ) ) )
30                      << new MyStandardItem( QString::null ) );

       model.appendRow( QList<QStandardItem*>()
                        << new MyStandardItem( QString( "Educate personel" ) )
                        << new MyStandardItem( KDGantt::TypeTask )
35                      << new MyStandardItem( QDateTime( QDate( 2007, 3, 13 ) ) )
                        << new MyStandardItem( QDateTime( QDate( 2007, 3, 31 ) ) )
                        << new MyStandardItem( QString::null ) );

       KDGantt::DateTimeGrid grid;❸
40     grid.setDayWidth( 16 );

       KDGantt::View view;❹
       view.setGrid( &grid );
       view.setModel( &model );
45
       view.setWindowTitle( "KDGantt - Basic Gantt Chart" );
```

```
       view.show();

       return app.exec();
50 }
```

❶ The model is created, and in the following lines the data is added to the model. You can of course use your own model for this instead of using `QStandardItemModel`.

❷ The data is added to the model, row by row. As you can see, the first column of the row contains the type of entry, in this case a `KDGantt::TypeTask`, and the following columns contains the caption, start date, end date, and completion percent, in that order.

❸ As we are dealing with quite long tasks we will need to modify the grid in order to be able to see the entire schedule without scrolling. In this case, we set the day width of the grid to 16 pixels. This modifies the lower scale on the schedule, while the higher scale is calculated automatically.

❹ Last, we create the actual view, set it up with the grid, supply the model, give it a title and show it.

An Extended Gantt Chart

The following example is a more verbose version of the Gantt chart above. It can also be found in `step01b.cpp`.

```
 1
   #include <QApplication>
   #include <QStandardItemModel>

 5 #include <KDGanttView>
   #include <KDGanttDateTimeGrid>
   #include <KDGanttConstraintModel>

   class MyStandardItem : public QStandardItem {
10 public:
       MyStandardItem( const QVariant& v ) : QStandardItem()
       {
           setData( v, Qt::DisplayRole );
       }
15     MyStandardItem( const QString& v ) : QStandardItem()
       {
           setData( v, Qt::DisplayRole );
       }
   };
20
   int main( int argc, char* argv[] )
   {
       QApplication app( argc, argv );

25     QStandardItemModel model;

       QStandardItem* topitem = new MyStandardItem( QString( "Decide on new❶ product" ) );

       topitem->appendRow( QList<QStandardItem*>()
30                 << new MyStandardItem( QString( "Find appropriate products" ) )
                   << new MyStandardItem( KDGantt::TypeTask )
                   << new MyStandardItem( QDateTime(QDate(2007, 3, 1)) )
                   << new MyStandardItem( QDateTime(QDate(2007, 3, 3)) )
                   );
35
       topitem->appendRow( QList<QStandardItem*>()
                   << new MyStandardItem( QString( "Evaluate Product A" ) )
                   << new MyStandardItem( KDGantt::TypeTask )
```

```
                    << new MyStandardItem( QDateTime(QDate(2007, 3, 5)) )
                    << new MyStandardItem( QDateTime(QDate(2007, 3, 10)) )
                    );

        topitem->appendRow( QList<QStandardItem*>()
                    << new MyStandardItem( QString( "Evaluate Product B" ) )
                    << new MyStandardItem( KDGantt::TypeTask )
                    << new MyStandardItem( QDateTime(QDate(2007, 3, 5)) )
                    << new MyStandardItem( QDateTime(QDate(2007, 3, 10)) )
                    );

        topitem->appendRow( QList<QStandardItem*>()
                    << new MyStandardItem( QString( "Evaluate Product C" ) )
                    << new MyStandardItem( KDGantt::TypeTask )
                    << new MyStandardItem( QDateTime(QDate(2007, 3, 5)) )
                    << new MyStandardItem( QDateTime(QDate(2007, 3, 10)) )
                    );

        topitem->appendRow( QList<QStandardItem*>()
                    << new MyStandardItem( QString( "Make decision" ) )
                    << new MyStandardItem( KDGantt::TypeTask )
                    << new MyStandardItem( QDateTime(QDate(2007, 3, 12)) )
                    << new MyStandardItem( QDateTime(QDate(2007, 3, 13)) )
                    );

        model.appendRow( QList<QStandardItem*>()
                    << topitem
                    << new MyStandardItem( KDGantt::TypeSummary ) );

        model.appendRow( QList<QStandardItem*>()❷
                    << new MyStandardItem( QString( "Decision made" ) )
                    << new MyStandardItem( KDGantt::TypeEvent )
                    << new MyStandardItem( QDateTime(QDate(2007, 3, 13)) )
                    << new MyStandardItem( QDateTime(QDate(2007, 3, 13)) )
                    );

        model.appendRow( QList<QStandardItem*>()
                    << new MyStandardItem( QString( "Educate personel" ) )
                    << new MyStandardItem( KDGantt::TypeTask )
                    << new MyStandardItem( QDateTime(QDate(2007, 3, 13)) )
                    << new MyStandardItem( QDateTime(QDate(2007, 3, 31)) )
                    << new MyStandardItem( QString::null ) );

        KDGantt::ConstraintModel cmodel;❸
        QModelIndex pidx = model.index( 0, 0 );
        cmodel.addConstraint( KDGantt::Constraint( model.index( 0, 0, pidx ),
                                                   model.index( 1, 0, pidx ) ) );
        cmodel.addConstraint( KDGantt::Constraint( model.index( 0, 0, pidx ),
                                                   model.index( 2, 0, pidx ) ) );
        cmodel.addConstraint( KDGantt::Constraint( model.index( 0, 0, pidx ),
                                                   model.index( 3, 0, pidx ) ) );
        cmodel.addConstraint( KDGantt::Constraint( model.index( 1, 0, pidx ),
                                                   model.index( 4, 0, pidx ) ) );
        cmodel.addConstraint( KDGantt::Constraint( model.index( 2, 0, pidx ),
                                                   model.index( 4, 0, pidx ) ) );
        cmodel.addConstraint( KDGantt::Constraint( model.index( 3, 0, pidx ),
                                                   model.index( 4, 0, pidx ) ) );

        KDGantt::DateTimeGrid grid;
        grid.setDayWidth( 16 );

        KDGantt::View view;
        view.setGrid( &grid );
        view.setModel( &model );
        view.setConstraintModel( &cmodel );

        view.setWindowTitle( "KDGantt - Extended Gantt Chart" );
        view.show();

        return app.exec();
    }
```

❶ To gather several tasks in one task, we use a KDGantt::TypeSummary entry, and add five rows of child entries to it. A summary entry does not need a start date and end date as these are automatically calculated from the children.

❷ To indicate that a deadline has been reached at a certain date, we use a KDGantt::TypeEvent in the Gantt chart. This has the same start date and end date and is shown as a diamond shape in the chart.

❸ The `KDGantt::ConstraintModel` class is used to set up constraints between entries. One single entry can depend on many entries, and vice versa—several entries can depend on one entry. This is done on the following lines, adding constraints between the `find` entry and the `evaluate` entries, and also from the `evaluate` entries to the `decision` entry. More on constraints can be found in Section , “Working With Constraints”.

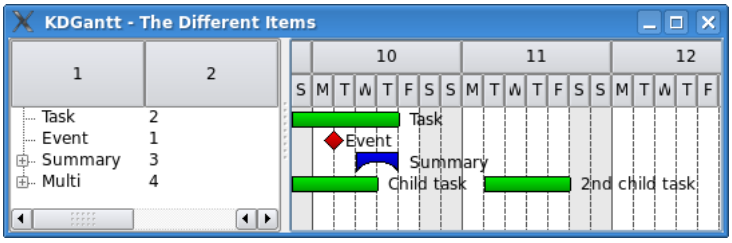
Basic Usage, Working With Items

Here we will will focus on using KD Chart with the `KDGantt::View` class. For more advanced concepts, such as working with the `KDGantt::GraphicsView` please see Section , “Working With The GraphicsView”.

Introduction to Items

There are three different items available in KD Chart: Task, Event, and Summary items, represented by the enumeration values KDGantt::TypeTask, KDGantt::TypeEvent, KDGantt::TypeSummary and KDGantt::TypeMulti. There are also two special values, KDGantt::TypeNone and KDGantt::TypeUser. You can see a typical example of item usage in Figure 10.3.

Figure 10.3. The Different Items



These items are all created very similarly, it is just a matter of setting the correct type of the item. Each item is then given a name and, in the case of tasks and events, a start date and end date. Summary items do not have any dates set, as these are calculated from their children. Summary items are also not visible unless they have any visible children, such as tasks or events.

When adding several items at once, which could be the case when a file is loaded, redrawing the Gantt view after every addition might become very slow. To avoid this, you can use `KDGantt::View::setUpdatesEnabled( false )` , which will cause the

update of the Gantt view to be suspended until you set the value to `true` again.

It is also possible to change the appearance of the different items, details and code examples are available in Section , “Customizing Items”.

Tasks

Task items are used to model the planned tasks in a Gantt chart. By default, they are drawn as a green rectangle which spans from the start date to the end date. If the time is not specified, or if the start time equals the end time, nothing will be drawn in the schedule view, however the entry will still be visible in the tree view.

Events

Events are used to schedule certain events in, for example, a software project. Events need both a start time and an end time, but only the start time will be used for the drawing. The item itself is drawn as a red diamond shape, but both the brush and pen used to draw it can be changed, as is explained further in Section , “Customizing Items”.

Summaries

Summary items are used to summarize several tasks into one, and has calculates its time from its children's start/end times. Because of this, a summary item takes only a caption, but not a start date or an end date. If a summary item is added to the model, but never receives any children, it will not be visible in the schedule view, only in the tree view. Its children can be of any type, such as a task, an event or a summary.

By default, summary items are blue and drawn as a line with one triangle in each end.

Multi items

Multi items are similar to summary items as they both have several tasks as children and the start and end dates are calculated automatically. The difference is when using multi items, all the tasks are drawn on the same line in the graphics view.

Adding Items

The data for KD Chart is stored in a model, and initially given to the view by using `KDGantt::View::setModel()`. When working with `KDGantt::View`, each row in the model represents an item in the Gantt chart, and the different columns are bound to store one type of data, as seen in Table 10.1 which, as a Gantt chart, would look similar to that in Figure 10.1.

Table 10.1. Item data table

Caption	Item Type	Start Date	End Date	Completion Percentage
Decide on new product	KDGantt::TypeTask	2007-03-01 08:00	2007-03-13 17:00	20
Educate staff	KDGantt::TypeTask	2007-03-14 08:00	2007-03-31 17:00	0

The following code example comes from `step02a.cpp`, and the result is shown in Figure 10.3.

```
class MyStandardItem : public QStandardItem {❶
public:
    MyStandardItem( const QVariant& v ) : QStandardItem()
    {
        setData( v, Qt::DisplayRole );
    }
    MyStandardItem( const QString& v ) : QStandardItem()
    {
        setData( v, Qt::DisplayRole );
    }
};

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

    QStandardItemModel model;
    model.appendRow( QList<QStandardItem*>()❷
                     << new MyStandardItem( QString( "Task" ) )
                     << new MyStandardItem( KDGantt::TypeTask )
                     << new MyStandardItem( QDateTime( QDate( 2007, 3, 4 ) ) )
                     << new MyStandardItem( QDateTime( QDate( 2007, 3, 9 ) ) ) );

    model.insertRow( 1 );
    model.setData( model.index( 1, 0 ), QString( "Event" ) );❸
    model.setData( model.index( 1, 1 ), KDGantt::TypeEvent );
    model.setData( model.index( 1, 2 ), QDateTime( QDate( 2007, 3, 6 ) ) );
    model.setData( model.index( 1, 3 ), QDateTime( QDate( 2007, 3, 6 ) ) );

    // ...
}
```

❶ `MyStandardItem` is simply used for convenience when adding a row in one single call, instead of having to call `setData()` on five `QStandardItem` objects.

❷ Here, we add one entire Gantt chart item in one single call, by creating a list of `MyStandardItem` objects, each representing a specific data entry in the model.

❸ Another method of adding data would be to call `QAbstractItemModel::setData()` for each of the indices.

If you are adding child items to a summary item or multi item, simply use the model index of the summary or multi item as the parent index in the call to `QAbstractItemModel::index()`, instead of the default value. Or, if you are constructing your items from `QStandardItem` objects, use the object representing the summary or multi item, and call `QStandardItem::appendRow()`.

Customizing Items

There are two ways of customizing the appearance of items in KD Chart, either using the already present `ItemDelegate` or implementing your own custom `ItemDelegate` subclass. In this section we will cover the former; implementing your own custom `ItemDelegate` is covered in Section , “Creating Your Own ItemDelegate”.

Changing the Brush

Different types of items have different default colors. The initial default color is set to red for event items, green for tasks and blue for summaries.

The basic way of changing the color of an item type is to use a different `QBrush` when drawing it. This `QBrush` is then supplied to KD Chart using `KDGantt::ItemDelegate::setDefaultBrush()` , which takes two arguments: the type for which you wish to change the brush, and the actual brush.

This will change the brush, and hence the color, for all items of a certain type, including already created items.

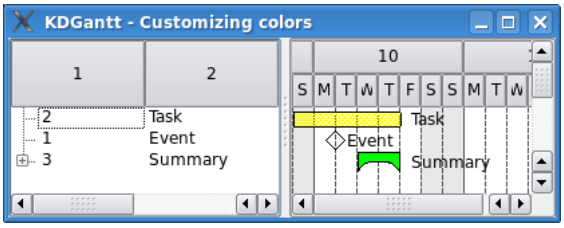
This code example comes from `step02b.cpp`, and the result is shown in Figure 10.4.

```
// Added a task, an event and a summary item, with a child

QBrush myTaskBrush( Qt::yellow, Qt::Dense3Pattern );❶
QBrush myEventBrush( Qt::NoBrush );
QBrush mySummaryBrush( Qt::green );
view.graphicsView()->itemDelegate()->setDefaultBrush( KDGantt::TypeTask,
                                                      myTaskBrush );    ❷
view.graphicsView()->itemDelegate()->setDefaultBrush( KDGantt::TypeEvent,
                                                      myEventBrush );
view.graphicsView()->itemDelegate()->setDefaultBrush( KDGantt::TypeSummary,
                                                      mySummaryBrush );
```

❶ Setting up the brushes for customizing the different items. The tasks will be yellow, with a moderately dense pattern, the events will be transparent while the summary items will be green.

❷ Setting the default brush on the `ItemDelegate`. As mentioned, this will change the brush for all existing items of this particular type.

Figure 10.4. Customizing the Brush



Changing the Pen

Just as with item colors, you can change the line style for each item type, simply by providing `KDGantt::ItemDelegate` with a `QPen` object. The default pen for all items is black, with a width of 1 pixel.

To change the pen for an item type, use `KDGantt::ItemDelegate::setDefaultPen()` . This method takes two arguments: the item type and the `QPen` object.

Note

Changing the pen for an item not only changes the line color in the schedule view, it also changes the color of the caption text in the schedule view, as these are drawn using the same pen.

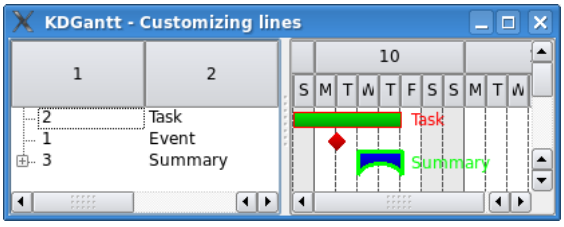
This code example comes from `step02c.cpp`, and the result is shown in Figure 10.5.

```
// Added a task, an event and a summary item, with a child

QPen myTaskPen( Qt::red ); ❶
QPen myEventPen( Qt::NoPen );
QPen mySummaryPen( Qt::green, 3 );
view.graphicsView()->itemDelegate()->setDefaultPen( KDGantt::TypeTask,
                                                    myTaskPen );      ❷
view.graphicsView()->itemDelegate()->setDefaultPen( KDGantt::TypeEvent,
                                                    myEventPen );
view.graphicsView()->itemDelegate()->setDefaultPen( KDGantt::TypeSummary,
                                                    mySummaryPen );
```

❶ Setting up the pens for customizing the different items. The tasks will be drawn with a red line, the events will be drawn without line while the summary items will have a green line with a width of 3 pixels.

❷ Setting the default pen on the `ItemDelegate`. As you can see in the screenshot below, this also affects the caption in the schedule view for each of the items.

Figure 10.5. Customizing Lines



Start and End Time

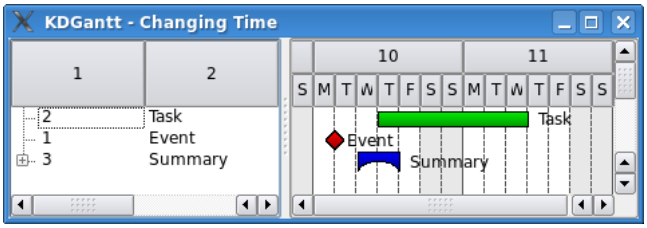
Changing the start and end time of an item is simply a matter of modifying the data within the model, specifically in the third and fourth column. This is done by using `QAbstractItemModel::setData()`, and giving it the index for the start or end date entries, as well as the new date.

As mentioned earlier in this manual, the start date and end date for summary items are calculated automatically and should not be set. Also, events only take the start date into consideration when drawing, although they still need a valid end date. Hence, the only items that require both a start date and an end date are task items.

The complete source code for this example is in `step02d.cpp`. The result can be seen in Figure 10.6, and as seen compared to Figure 10.3 the task item has been moved and extended.

```
// A task added to the first row of the model

model.setData( model.index( 0, 2 ), QDateTime( QDate( 2007, 3, 8 ) ) );
model.setData( model.index( 0, 3 ), QDateTime( QDate( 2007, 3, 15 ) ) );
```

Figure 10.6. Customizing Start and End Times



Working With Constraints

Constraints are most often used to show dependencies between tasks. We saw an example of this in the introductory chapters, in Figure 10.2. Using constraints makes it easier for the reader to see quickly which tasks need to be completed before the next task can be started.

In this section, we will try to explain this a bit further, by means of a few code examples on how you can use constraints.

Introduction To Constraints

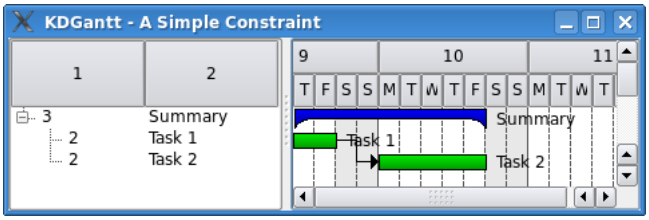
A constraint is an object of `KDGantt::Constraint`, containing references to the indices of the two tasks it is linking together. One `KDGantt::Constraint` can only link two tasks, however, several `KDGantt::Constraints` can reference the same tasks at the same time. This enables one task to be dependant on several others, and several

tasks to be dependant on one task.

To have KD Chart observe a constraint, the `KDGantt::Constraint` object must be added to a `KDGantt::ConstraintModel`.

The example in Figure 10.7 shows links between items of the type KDGantt::TypeTask but it works just the same with items of the types KDGantt::TypeEvent and KDGantt::TypeSummary.

Figure 10.7. A Simple Constraint



Adding Constraints

As mentioned in the introduction, constraints are objects of `KDGantt::Constraint` added to a `KDGantt::ConstraintModel` using `KDGantt::ConstraintModel::addConstraint()` . This constraint model is then passed into the view by calling `KDGantt::View::setConstraintModel()`.

The `KDGantt::Constraint` object is made up of two `QModelIndex`'s, each representing an item in the Gantt chart with the second index depending on the first index.

The complete source code for this example is in `step02e.cpp`. The result can be seen in Figure 10.7

```
QStandardItemModel model;

// Added summary on index 0,0

// Added two tasks on index 0,0 and 0,1 as the children of the summary

KDGantt::ConstraintModel cmodel;
QModelIndex pidx = model.index( 0, 0 );
cmodel.addConstraint( KDGantt::Constraint( model.index( 0, 0, pidx ),
                                           model.index( 1, 0, pidx ) ) );
view.setModel( &model );
view.setConstraintModel( &cmodel );
```

Customizing Constraints

To customize the look of the constraints you will need to implement your own `KDGantt::ItemDelegate` and override `paintConstraintItem()`. More on how to

do this is covered in Section , “Creating Your Own ItemDelegate”.

Working With the Grid

There are a number of options you can change in the grid, the most important being the scale. There are three different values for the scale, ScaleAuto, ScaleHour and ScaleDay. Changing the scale does not change the width of the days, but rather the second row in the header. As can be seen in Figure 10.8 and Figure 10.9 the width stays the same although we changed the scale.

Figure 10.8. Using Day Scale

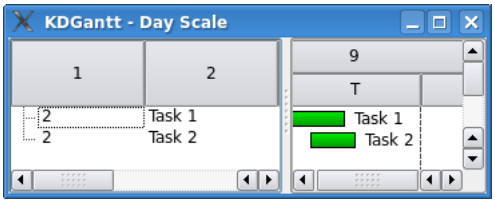
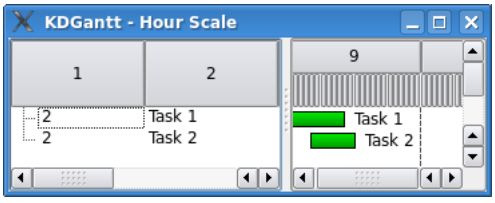


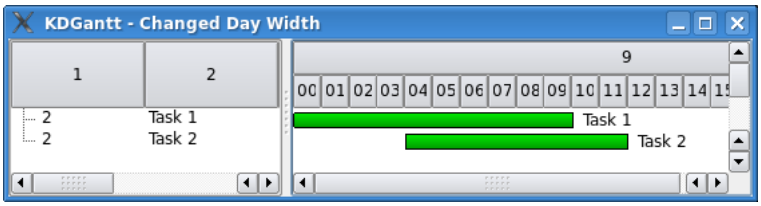
Figure 10.9. Using Hour Scale



Changing the width of the days would enable the user to see the items in the chart with finer granularity than before. Instead of looking at an item spanning 2 hours of a day from a view where you can see an entire week, you can, by increasing the day width, see exactly during which hours the item takes place. This is done by calling `KDGantt::DateTimeGrid::setDayWidth()` , giving it a value representing the actual pixel width of a day.

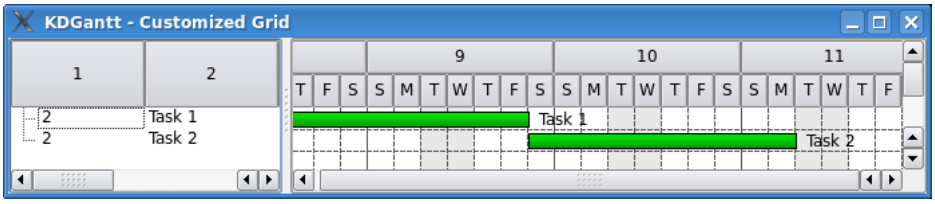
The screenshot shown in Figure 10.10 is based on the same data as the one in Figure 10.9, and is also using the hour scale. However, the day width has been changed to 500 pixels, making it possible to see at which hours the tasks begin and end.

Figure 10.10. Changed Day Width



You can also modify the week shown in the grid, both by setting the start day of the week as well as setting which days are free. The former is done by calling `KDGantt::DateTimeGrid::setWeekStart()`, and the latter by calling `KDGantt::DateTimeGrid::setFreeDays()`. The start day of the week will have a solid left line drawn in the grid, instead of a dashed line which is the case for all other week days. Also, quite naturally, the week scale will use it as week delimiter. The free days will be drawn with a grey background instead of the default white.

Figure 10.11. Customized Grid



The following code example can be found in `step02f.cpp`, and the results can be seen in Figure 10.11.

```
// ... set up the model, added two tasks

KDGantt::DateTimeGrid grid;
grid.setScale( KDGantt::DateTimeGrid::ScaleDay );
grid.setDayWidth( 20 );
grid.setWeekStart( Qt::Sunday );
grid.setFreeDays( QSet<Qt::DayOfWeek>() << Qt::Tuesday << Qt::Wednesday );
grid.setRowSeparators( true );

// ... set up the view
```

User Interaction

KD Chart has a set of built-in user interactions. For example, a user can click and drag an item in the chart to move it, or grab either end of it to change the start date or end date. It is also possible to create a constraint between two items by clicking-and-dragging.

Besides these default implementations, you can of course add your own user interactions by connecting to various signals in `KDGantt::GraphicsView` and

`KDGantt::GraphicsScene`. Please refer to the reference manual for details on each class.

Working With The GraphicsView

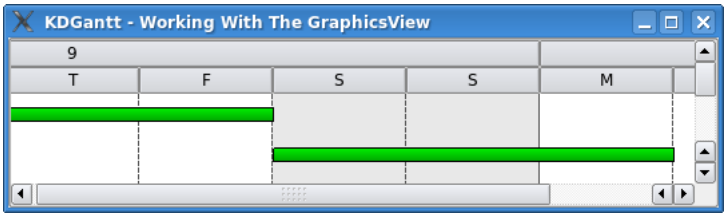
Here we will give you a more in-depth knowledge of KD Chart, and how you can work with it to fit your needs. We will also introduce a new way of adding items to the Gantt chart, rather than using `KDGantt::View`.

Furthermore, this chapter will show you how you can modify all items in the Gantt chart more extensively than offered by the available API.

The `KDGantt::GraphicsView` is the component visible to the right in all screenshots so far, where all the tasks, events and summaries are drawn. You are able to use this directly instead of using `KDGantt::View`, but this introduces some new concepts when it comes to adding items to the Gantt chart. You also have greater responsibility to make sure everything needed is initialized, as this was handled by the `KDGantt::View` earlier.

As described in Section , “Basic Usage, Working With Items”, when adding items to the view you need to make sure the correct data is put into the correct column of the model. This is not the case when using `KDGantt::GraphicsView` directly, now all data goes to index `0,0` but with different data item roles. The item roles used are ItemTypeRole, StartTimeRole, EndTimeRole and TaskCompletionRole for each type of data.

Figure 10.12. Only Using GraphicsView



The following code example is also available in `step03a.cpp`. The result can be seen in Figure 10.12

```
 1
                #include <QApplication>
   #include <QStandardItemModel>
 5 #include <QPointer>

   #include <KDGanttGraphicsView>
   #include <KDGanttDateTimeGrid>
   #include <KDGanttAbstractRowController>
10                                                                    ❶
   class MyRowController : public KDGantt::AbstractRowController {
   private:
       static const int ROW_HEIGHT ;
       QPointer<QAbstractItemModel> m_model;
15
```

```
   public:
       MyRowController()
       {
       }
20
       void setModel( QAbstractItemModel* model )
       {
           m_model = model;
       }
25
       /*reimp*/int headerHeight() const { return 40; }

       /*reimp*/ bool isRowVisible( const QModelIndex& ) const { return true;}
       /*reimp*/ bool isRowExpanded( const QModelIndex& ) const { return false; }
30     /*reimp*/ KDGantt::Span rowGeometry( const QModelIndex& idx ) const
       {
           return KDGantt::Span( idx.row()*ROW_HEIGHT, ROW_HEIGHT );
       }
       /*reimp*/ int maximumItemHeight() const {
35         return ROW_HEIGHT/2;
       }
       /*reimp*/int totalHeight() const {
           return m_model->rowCount()* ROW_HEIGHT;
       }
40
       /*reimp*/ QModelIndex indexAt( int height ) const {
           return m_model->index( height/ROW_HEIGHT, 0 );
       }

45     /*reimp*/ QModelIndex indexBelow( const QModelIndex& idx ) const {
           if ( !idx.isValid() )return QModelIndex();
           return idx.model()->index( idx.row()+1, idx.column(), idx.parent() );
       }
       /*reimp*/ QModelIndex indexAbove( const QModelIndex& idx ) const {
50         if ( !idx.isValid() )return QModelIndex();
           return idx.model()->index( idx.row()-1, idx.column(), idx.parent() );
       }

   };
55
   const int MyRowController::ROW_HEIGHT = 30;
                                                          ❷
   int main( int argc, char* argv[] )
   {
60     QApplication app( argc, argv );

       QStandardItemModel model;

       QStandardItem* item = new QStandardItem();
65     item->setData( KDGantt::TypeTask, KDGantt::ItemTypeRole );
       item->setData( QString( "Decide on new product" ) );
       item->setData( QDateTime( QDate( 2007, 3, 1 ) ), KDGantt::StartTimeRole );
       item->setData( QDateTime( QDate( 2007, 3, 3 ) ), KDGantt::EndTimeRole );

70     QStandardItem* item2 = new QStandardItem();
       item2->setData( KDGantt::TypeTask, KDGantt::ItemTypeRole );
       item2->setData( QString( "Educate personel" ) );
       item2->setData( QDateTime( QDate( 2007, 3, 3 ) ), KDGantt::StartTimeRole );
       item2->setData( QDateTime( QDate( 2007, 3, 6 ) ), KDGantt::EndTimeRole );
75                                          ❸
       model.appendRow( item );
       model.appendRow( item2 );

       MyRowController rowController;
80     rowController.setModel( &model );

       KDGantt::GraphicsView graphicsView;
       graphicsView.setRowController( &rowController );
       graphicsView.setModel( &model );
85
       graphicsView.setWindowTitle( "KDGantt - Working With The GraphicsView" );
       graphicsView.show();

       return app.exec();
90 }
```

❶ One important part of setting up the `KDGantt::GraphicsView` is that you need to implement your own row controller as a subclass of `KDGantt::AbstractRowController`. The row controller is used by `KDGantt::GraphicsView` to navigate through the data model and also to determine the row geometries. For more information on each method you will need to override, please refer to the reference manual.

❷ Two new items are created and added to the model. As you can see, we only need to create one `QStandardItem` object per item in the Gantt chart with all the data in different roles, compared to when using `KDGantt::View` where we needed four or five `QStandardItem` objects for each item in the Gantt chart.

❸ We then create our `KDGantt::GraphicsView` and set the required member variables on it—the model containing the items to be drawn as well as the custom row controller.

Setting Up The Grid

Working with the grid on a standalone `KDGantt::GraphicsView` is no different from what we have described in Section , “Working With the Grid” , except that instead setting it up with `KDGantt::View::setGrid()`, you have to use `KDGantt::GraphicsView::setGrid()`.

Warning

You should never need to call either `KDGantt::AbstractGrid::setModel()` or `KDGantt::AbstractGrid::setRootIndex()` from client code. These are for internal use only. Doing so will cause an assert in client code.

A complete code example where the grid is used with a standalone `KDGantt::GraphicsView` can be found in `step03b.cpp`.

Creating Your Own `ItemDelegate`

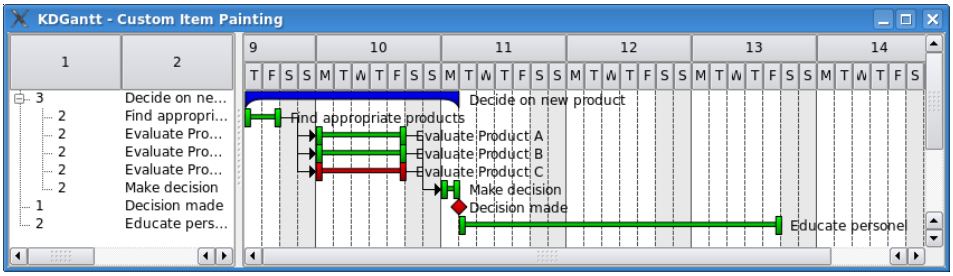
In order to customize items and constraints more extensively than offered by the basic API you will need to implement your own custom `ItemDelegate`. This is then installed on the graphics view by using `KDGantt::GraphicsView::setItemDelegate()`. In this section, we will have a closer look on exactly what you can change with your own `ItemDelegate`, as well as show some code examples doing just that.

Customizing Items

In Section , “Customizing Items”, we have shown how you can change the brush and pen for all the items in the Gantt chart. By overriding

`KDGantt::ItemDelegate::paintGanttItem()` you can customize the look of your items even further as you are then responsible for painting them.

Figure 10.13. Custom Item Painting



In the following code example, we change the look of tasks, while events and summaries are painted by the base class. For the complete code, please refer to `step03c.cpp`. The result can be seen in Figure 10.13.

```
using namespace KDGantt;
class MyCustomItemDelegate : public ItemDelegate {
    void paintGanttItem( QPainter* painter,❶
                         const StyleOptionGanttItem& opt,
                         const QModelIndex& idx )
    {
        painter->setRenderHints( QPainter::Antialiasing );
        if ( !idx.isValid() ) return;
        ItemType type = static_cast<ItemType>(
                            idx.model()->data( idx, ItemTypeRole ).toInt() );
        QString txt = idx.model()->data( idx, Qt::DisplayRole ).toString();
        QRectF itemRect = opt.itemRect;
        QRectF boundingRect = opt.boundingRect;
        boundingRect.setY( itemRect.y() );
        boundingRect.setHeight( itemRect.height() );

        QBrush brush = defaultBrush( type );
        if ( opt.state & QStyle::State_Selected ) {❷
            QLinearGradient selectedGrad( 0., 0., 0.,
                                          QApplication::fontMetrics().height() );
            selectedGrad.setColorAt( 0., Qt::red );
            selectedGrad.setColorAt( 1., Qt::darkRed );

            brush = QBrush( selectedGrad );
        }

        painter->setPen( defaultPen( type ) );
        painter->setBrush( brush );
        painter->setBrushOrigin( itemRect.topLeft() );

        switch( type ) {
        case TypeTask:
            if ( itemRect.isValid() ) {
                QRectF r = itemRect;
                r.translate( 0., r.height() / 3. );
                r.setHeight( 2. * r.height() / 9. );

                painter->translate( 0.5, 0.5 );
                painter->drawRect( r );❸

                QRectF leftRect = itemRect;
                leftRect.setWidth( 5 );
                painter->drawRect( leftRect );
```

```
                QRectF rightRect = itemRect;
                rightRect.setWidth( 5 );
                rightRect.translate( r.width() - 5, 0 );
                painter->drawRect( rightRect );

                Qt::Alignment ta;
                switch( opt.displayPosition ) {
                case StyleOptionGanttItem::Left: ta = Qt::AlignLeft; break;
                case StyleOptionGanttItem::Right: ta = Qt::AlignRight; break;
                case StyleOptionGanttItem::Center: ta = Qt::AlignCenter; break;
                }
                painter->drawText( boundingRect, ta, txt );
            }
            break;
        default:
            KDGantt::ItemDelegate::paintGanttItem( painter, opt, idx );
            break;
        }
    }
};
```

❶ As we are changing the look of Gantt items, we need to override `paintGanttItem()`. If we were to change the appearance of constraints, we would have to override `paintConstraintItem()` instead.

❷ One thing that we changed compared to the default painting of the task items, is that selected items are filled with a red gradient instead of having the line width doubled.

❸ The second thing we changed was the actual look of the task items. By default, they are drawn as a rectangle spanning from the start date to the end date. This is still the case, however we have decreased the height of the rectangle, and added two vertical rectangles at the beginning and end of the task, to symbolize the fact that you, as a user, can click and drag the edge of the rectangle to change the start date and end date.

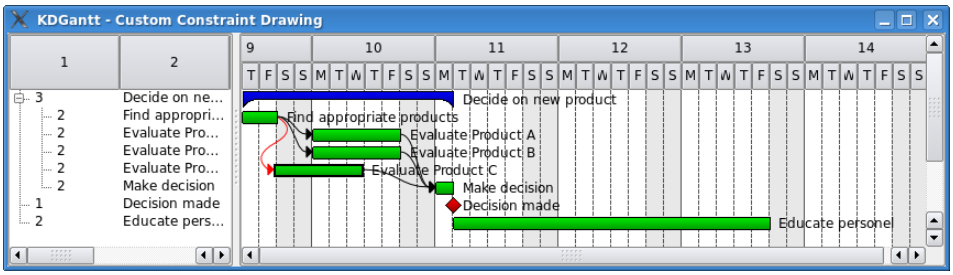
The process of customizing the appearance of summary and event items are of course similar, it is up to your imagination what they can look like.

Customizing Constraints

As mentioned in the previous section, it is also possible to change the apperance of the constraints. This is achieved by overriding `KDGantt::ItemDelegate::paintConstraintItem()` , providing your own code for drawing it.

The default look of a constraint is an arrow drawn from the end of one item to the beginning of another, as seen in e.g. Figure 10.13. In the following example, we will change the way the connection is drawn from straight lines to a Bezier curve. The complete code example is available in `step03d.cpp` and the results can be seen in Figure 10.14. Please note that the "Evaluate Product C" task has been moved manually to demonstrate what the constraint looks like when the start and end times overlap.

Figure 10.14. Custom Constraint Drawing




```
class MyCustomItemDelegate : public ItemDelegate {

    static const qreal TURN = 10.;

    void paintConstraintItem( QPainter* painter,❶
                              const QStyleOptionGraphicsItem& opt,
                              const QPointF& start, const QPointF& end )
    {
        Q_UNUSED( opt );
        painter->setRenderHints( QPainter::Antialiasing );
        qreal midx = ( end.x()-start.x() )/2. + start.x();
        qreal midy = ( end.y()-start.y() )/2. + start.y();

        if ( start.x() <= end.x() ) {
            painter->setPen( Qt::black );
            painter->setBrush( Qt::black );
        } else {
            painter->setPen( Qt::red );
            painter->setBrush( Qt::red );
        }
        if ( start.x() > end.x()-TURN ) {
            QPainterPath path( start );❷
            path.quadTo( QPointF( start.x() + TURN * 2., ( start.y() + midy ) / 2. ),
                         QPointF( midx, midy ) );
            path.quadTo( QPointF( end.x() - TURN * 2., ( end.y() + midy ) / 2. ),
                         QPointF( end.x() - TURN / 2., end.y() ) );

            QBrush brush = painter->brush();
            painter->setBrush( QBrush() );
            painter->drawPath( path );
            painter->setBrush( brush );

            QPolygonF arrow;
            arrow << end
                  << QPointF( end.x()-TURN/2., end.y()-TURN/2. )
                  << QPointF( end.x()-TURN/2., end.y()+TURN/2. );
            painter->drawPolygon( arrow );
        } else {
            painter->setBrush( QBrush() );
            QPainterPath path( start );❷
            path.cubicTo( QPointF( midx, start.y() ),
                          QPointF( midx, end.y() ),
                          QPointF( end.x()-TURN/2., end.y() ) );
            painter->drawPath( path );

            painter->setBrush( Qt::black );

            QPolygonF arrow;
            arrow << end
                  << QPointF( end.x()-TURN/2., end.y()-TURN/2. )
                  << QPointF( end.x()-TURN/2., end.y()+TURN/2. );
            painter->drawPolygon( arrow );
        }
    }
};
```

❶ When customizing the appearance of constraints, you will need to override `KDGantt::ItemDelegate::paintConstraintItem()`.

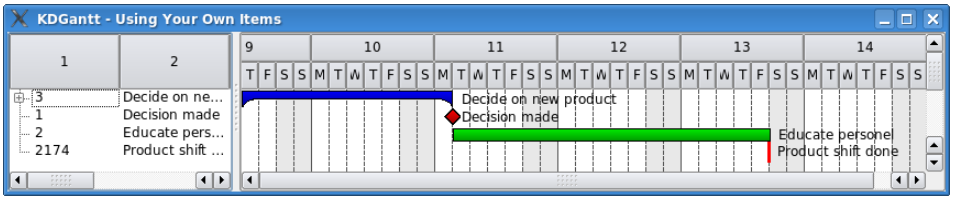
❷ What we have changed here was the drawing of the actual constraint link. We have set it up for two situations: when the start time and end time of the two tasks overlap, and when they do not. The painter path is then simply drawn using `QPainter::drawPath()`.

Creating Your Own Items

Besides changing the appearance of items and constraints, implementing your own `ItemDelegate` also has the advantage that you can add your own custom items to the Gantt chart. This is done similarly to customizing the items, but instead of changing the way an already existing item type is drawn, you simply look for your own type. This type must have a value greater than KDGantt::TypeUser, as the values below are reserved for KD Chart.

The following example shows how you can create a fourth item type, Deadline, used for showing a deadline in the Gantt chart. The complete code example can be found in `step03e.cpp` and the results can be seen in Figure 10.15.

Figure 10.15. Using Your Own Items




```
enum MyItemType {❶
    TypeDeadline = KDGantt::TypeUser + 1174
};

using namespace KDGantt;
class MyCustomItemDelegate : public ItemDelegate
{
    void paintGanttItem( QPainter* painter,
                         const StyleOptionGanttItem& opt,
                         const QModelIndex& idx )
    {
        if ( !idx.isValid() ) return;
        ItemType type = static_cast<ItemType>(
                            idx.model()->data( idx, ItemTypeRole ).toInt() );
        QString txt = idx.model()->data( idx, Qt::DisplayRole ).toString();
        QRectF itemRect = opt.itemRect;
        QRectF boundingRect = opt.boundingRect;
        boundingRect.setY( itemRect.y() );
        boundingRect.setHeight( itemRect.height() );

        switch( type ) {
        case TypeDeadline:❷
            if ( opt.boundingRect.isValid() ) {
                QPen pen( Qt::red );
                pen.setWidth( 3 );
                painter->setPen( pen );
                QRect r = opt.rect;
```

```
                QLineF line( 0., 0., 0., r.height() );
                painter->drawLine( line );

                painter->setPen( Qt::black );
                Qt::Alignment ta;
                switch( opt.displayPosition ) {
                case StyleOptionGanttItem::Left: ta = Qt::AlignLeft; break;
                case StyleOptionGanttItem::Right: ta = Qt::AlignRight; break;
                case StyleOptionGanttItem::Center: ta = Qt::AlignCenter; break;
                }
                painter->drawText( boundingRect, ta, txt );
            }
            break;
        default:
            KDGantt::ItemDelegate::paintGanttItem( painter, opt, idx );
            break;
        }
    }
};
```

❶ We declare our new type which is later used to distinguish between this type and the types supplied by KD Chart.

❷ When we discover that the painting code for our item is called, we simply draw it—in this case a 3 pixel thick vertical red line to show a deadline in the Gantt chart. The line takes up all the vertical space on the row it is set.

Appendix A. Q&A section

Building and installing KD Chart

A.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.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)

User interaction

A.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.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.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.

Storing / loading of KD Chart settings

A.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.

Dynamic data / Look and Feel

A.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.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.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/`.

Contacting KD Chart Support

A.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 kdchart-support@kdab.com. 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/`.