Qt in Education

Custom Widgets and Painting

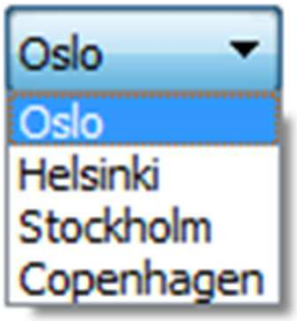
digia

digia

What is a Widget?

- Look

Feel

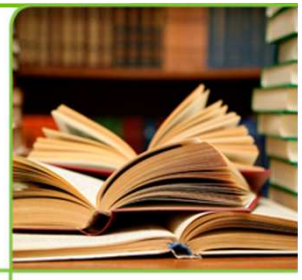


- API

Public Functions

	QComboBox (QWidget * *parent* = 0)
	~QComboBox ()
void	**addItem** (const QString & *text*, const QVariant & *userData* = QVariant())
void	**addItem** (const QIcon & *icon*, const QString & *text*, const QVariant & *userData* = QVariant())
void	**addItems** (const QStringList & *texts*)
QCompleter *	**completer** () const
int	**count** () const
int	**currentIndex** () const
QString	**currentText** () const
bool	**duplicatesEnabled** () const
int	**findData** (const QVariant & *data*, int *role* = Qt::UserRole, Qt::MatchFlags *flags* = Qt::MatchExactly | Qt::Mat
int	**findText** (const QString & *text*, Qt::MatchFlags *flags* = Qt::MatchExactly | Qt::MatchCaseSensitive) const
bool	**hasFrame** () const
virtual void	**hidePopup** ()
QSize	**iconSize** () const
void	**insertItem** (int *index*, const QString & *text*, const QVariant & *userData* = QVariant())
void	**insertItem** (int *index*, const QIcon & *icon*, const QString & *text*, const QVariant & *userData* = QVariant())
void	**insertItems** (int *index*, const QStringList & *list*)
InsertPolicy	**insertPolicy** () const
void	**insertSeparator** (int *index*)

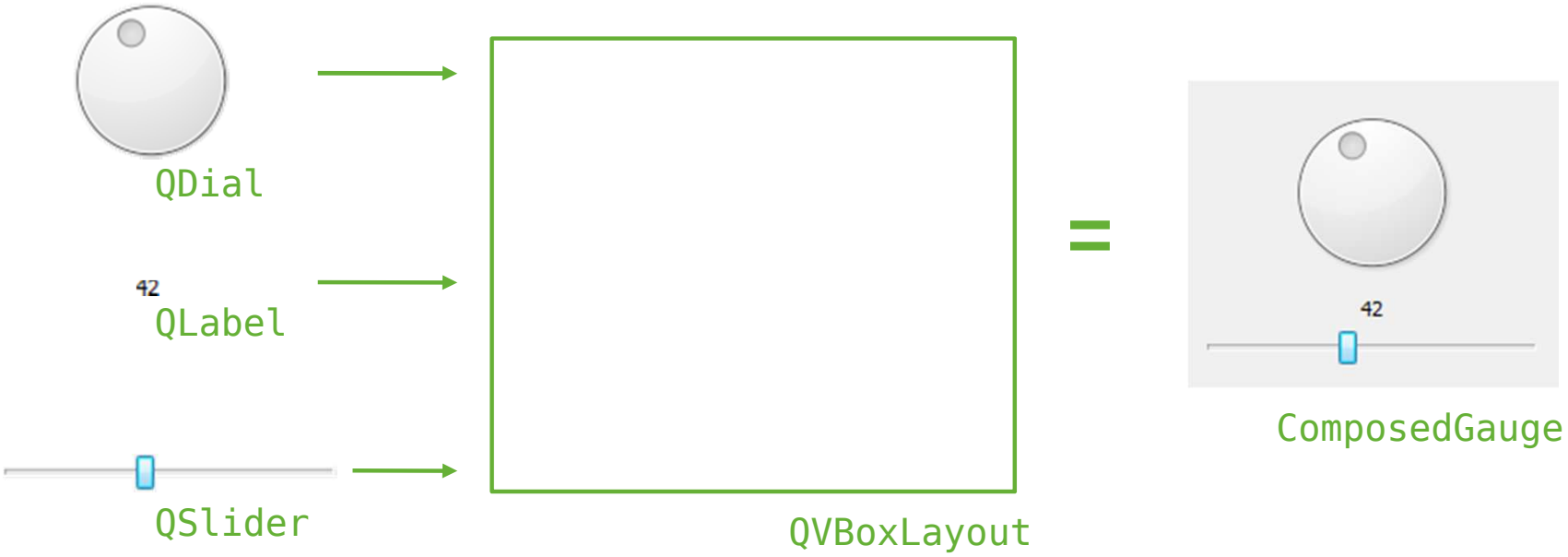
Custom Widgets



- Custom widgets can make or break a user experience
  - Custom widgets can enhance the look and feel
  - Custom widgets can help brand a user interface
  - Custom widgets are almost always a part of a non-trivial application
  - Beware – users know how the standard widgets work

Composing Widgets

- Composing widgets is an easy way to build re-usable widgets from existing components

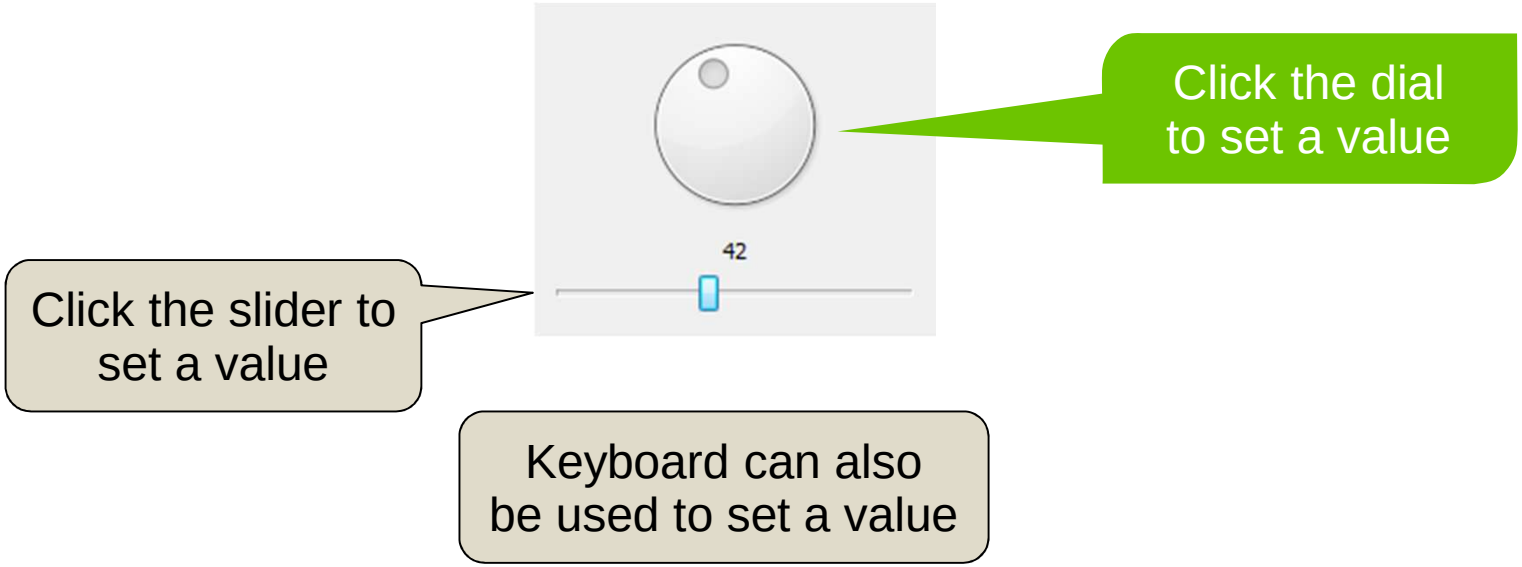


Composing Widgets

```
ComposedGauge::ComposedGauge(QWidget *parent) :
    QWidget(parent)
{
    QVBoxLayout *layout = new QVBoxLayout(this);

    QDial *dial = new QDial();
    QLabel *label = new QLabel();
    m_slider = new QSlider();

    layout->addWidget(dial);
    layout->addWidget(label);
    layout->addWidget(m_slider);

    m_slider->setOrientation(Qt::Horizontal);
    label->setAlignment(Qt::AlignCenter);

    ...
```

Look and Feel

- When composing widgets, the look and feel is inherited from the widgets used



Addressing the Feel

```
ComposedGauge::ComposedGauge(QWidget *parent) :
    QWidget(parent)
{
    ...

    connect(dial, SIGNAL(valueChanged(int)),
            m_slider, SLOT(setValue(int)));
    connect(m_slider, SIGNAL(valueChanged(int)),
            dial, SLOT(setValue(int)));
    connect(m_slider, SIGNAL(valueChanged(int)),
            label, SLOT(setNum(int)));

    dial->setFocusPolicy(Qt::NoFocus);

    dial->setValue(m_slider->value());
    label->setNum(m_slider->value());

    ...
```

API

- Wrapping the composed widgets in a task specific API makes the widget easy to (re)use

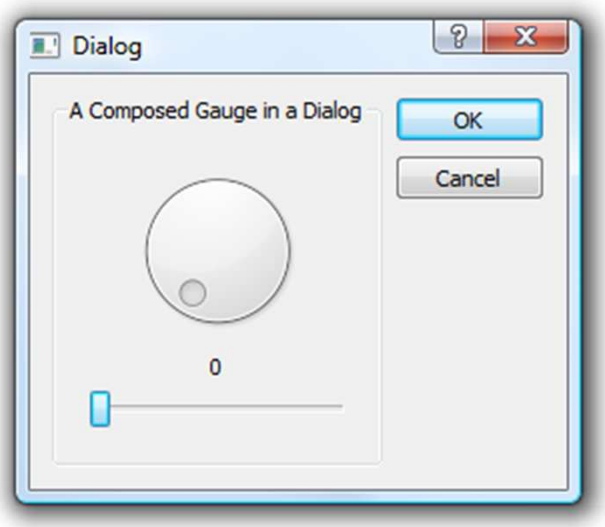
```
class ComposedGauge : public QWidget
{
Q_OBJECT
Q_PROPERTY(int value READ value WRITE setValue)
public:
    explicit ComposedGauge(QWidget *parent = 0);

    int value() const;

public slots:
    void setValue(int);

signals:
    void valueChanged(int);

private:
    QSlider *m_slider;
};
```

Implementing the API

- The `QSlider` holds the actual value

```
ComposedGauge::ComposedGauge(QWidget *parent) :
    QWidget(parent)
{
    ...

    connect(m_slider, SIGNAL(valueChanged(int)),
            this, SIGNAL(valueChanged(int)));
}

int ComposedGauge::value() const
{
    return m_slider->value();
}

void ComposedGauge::setValue(int v)
{
    m_slider->setValue(v);
}
```

Using the Widget



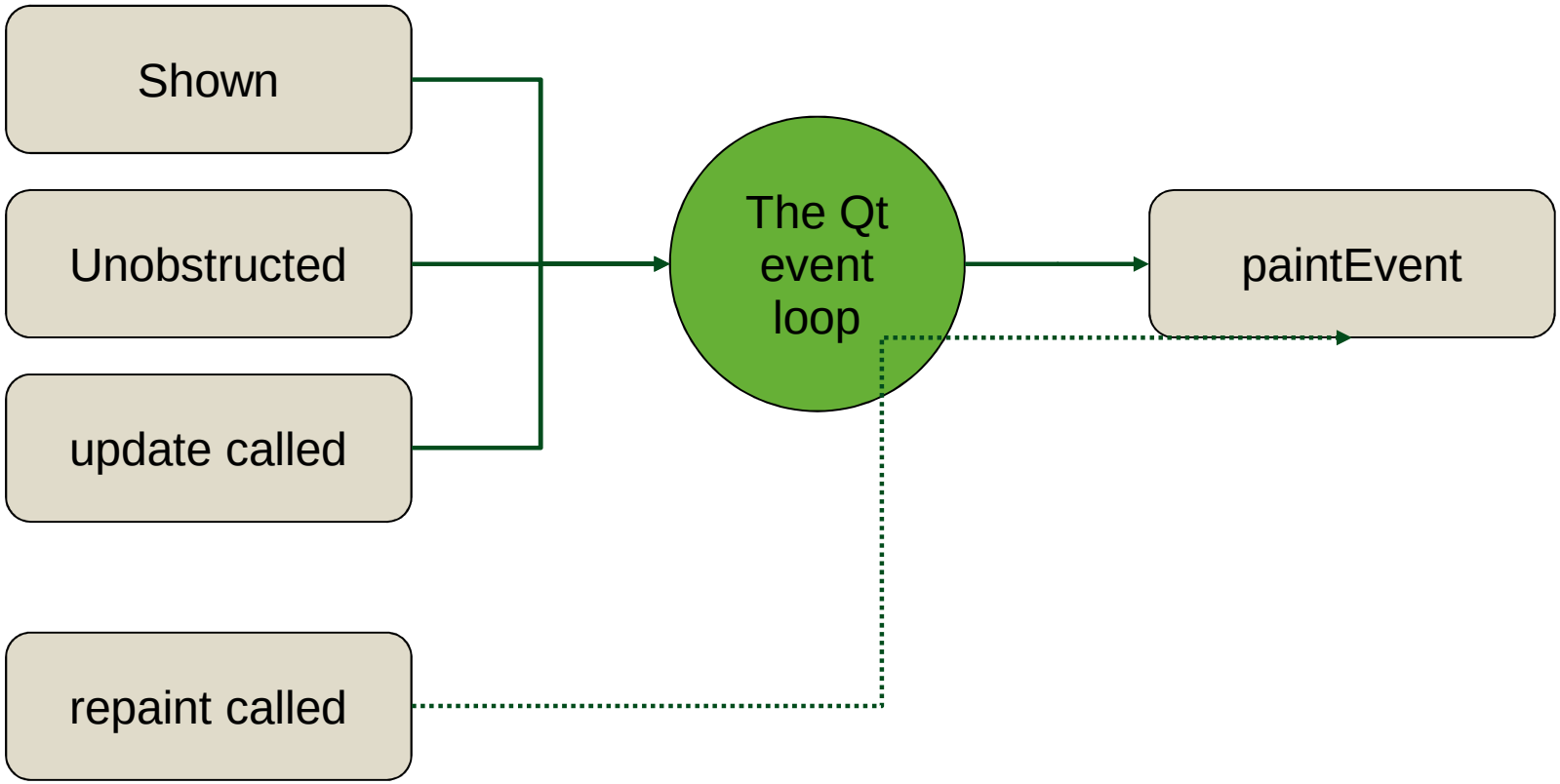

```
...

ComposedGauge *gauge = new ComposedGauge();

layout->addWidget(gauge);
connect(gauge, SIGNAL(valueChanged(int)), ... );

...
```

Custom Widgets

- To create truly custom widgets you must
  - Handle painting yourself
  - Handle events
    - Keyboard
    - Mouse
    - Resize
    - etc
  - Handle size hints and size policies

Custom Painting

- Painting is handled through the `paintEvent`



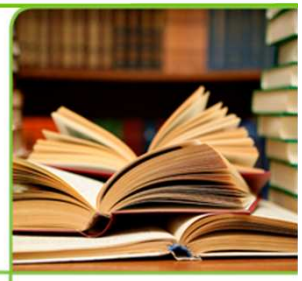
Custom Painting

- To handle paint events, simply override the `paintEvent` function and instantiate a `QPainter`

```
class MyWidget : public QWidget
{
    ...

protected:
    void paintEvent(QPaintEvent*);
```

```
void MyWidget::paintEvent(QPaintEvent *ev)
{
    QPainter p(this);

    ...
```

QPainter

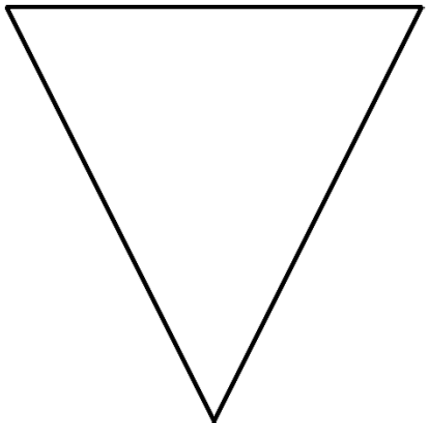


- `QPainter` objects paint on `QPaintDevice` objects
  - `QWidget`
  - `QImage` – hardware independent, for modifying
  - `QPixmap` – off-screen, for showing on screen
  - `QPrinter`
  - `QPicture` – records and replays painter commands
  - `QSvgGenerator` – records painter commands and stores them as SVG files

QPainter

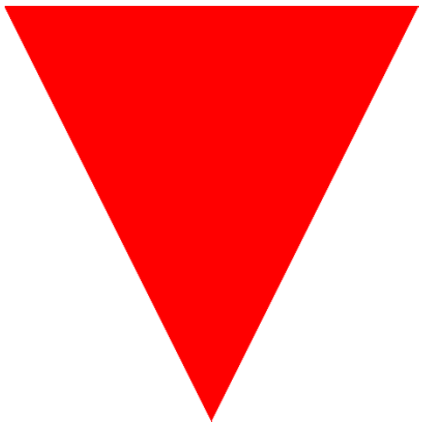
- A `QPainter` can be used to draw any shape
- Outlines are stroked using a `QPen`

```
QPainter p( ... );
QPen pen(Qt::black, 5);
p.setPen(pen);
p.drawPolygon(polygon);
```



- Interiors are filled using a `QBrush`

```
QPainter p( ... );
p.setPen(Qt::NoPen);
p.setBrush(Qt::red);
p.drawPolygon(polygon);
```



QColor

- QColor is used to represent colors

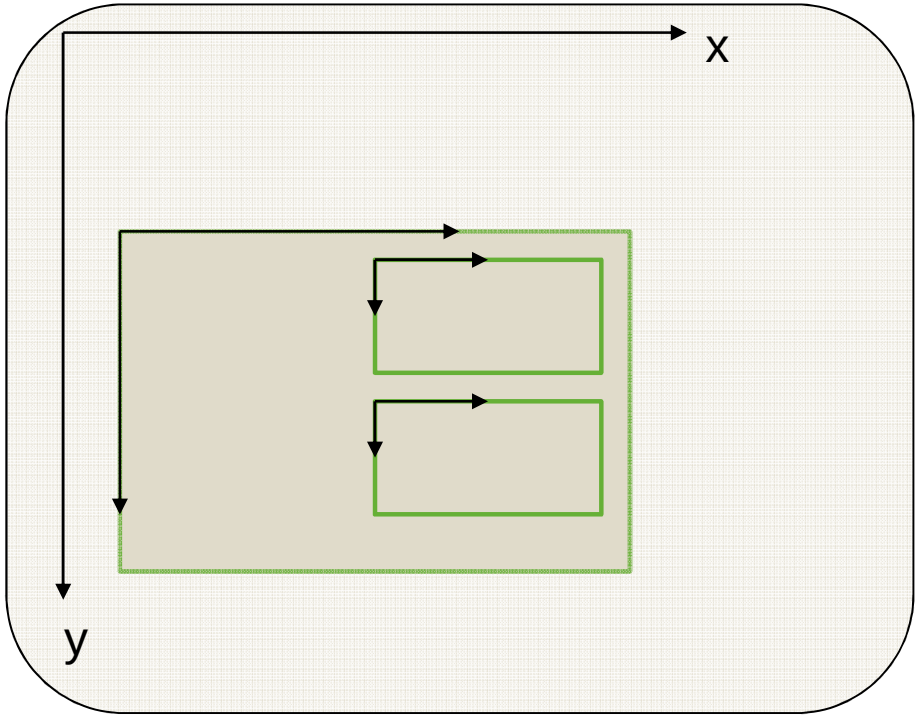
```
QColor c = QColor(red, green, blue, alpha=255);
```

- The arguments `red`, `green`, `blue` and `alpha` are specified in the range `0` to `255`
- The `alpha` setting controls the transparency
  - `255`, the color is opaque
  - `0`, the color is transparent

Coordinates

- The X-axis grows right
- The Y-axis grows downwards

- Coordinates can be
  - global
  - local (to a widget)



Coordinates

- Qt uses classes for points, sizes and rectangles
  - `QPoint` – a point (`x`, `y`)
  - `QSize` – a size (`width`, `height`)
  - `QRect` – a point and size (`x`, `y`, `width`, `height`)

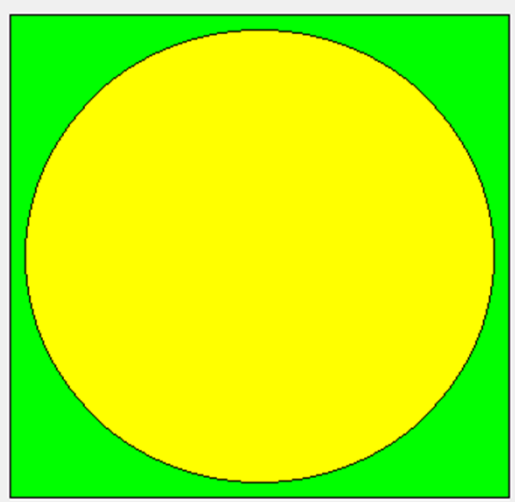
  Functions `topLeft`, `topRight`, `bottomLeft`, `bottomRight` and `size`

- `QPointF`/`QSizeF`/`QRectF` for floating point coord's

Basic Painting

- This is a trivial `paintEvent` implementation
  - Notice that the default pen is black

```
void RectWithCircle::paintEvent(QPaintEvent *ev)
{
    QPainter p(this);

    p.setBrush(Qt::green);
    p.drawRect(10, 10, width()-20, height()-20);
    p.setBrush(Qt::yellow);
    p.drawEllipse(20, 20, width()-40, height()-40);
}
```



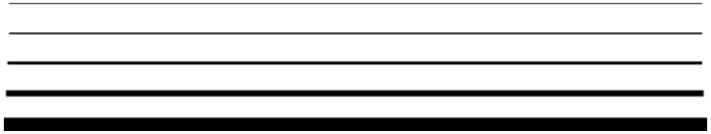
Convenient overloading

- Most draw-functions have multiple ways to provide coordinates and settings

```
drawRect(QRectF r);
  drawRect(QRect r);
  drawRect(int x, int y, int w, int h);


drawPoint(QPointF p);
  drawPoint(QPoint p);
  drawPoint(int x, int y);
```

Basic Shapes

- QPainter::drawPoint
- QPainter::drawLine

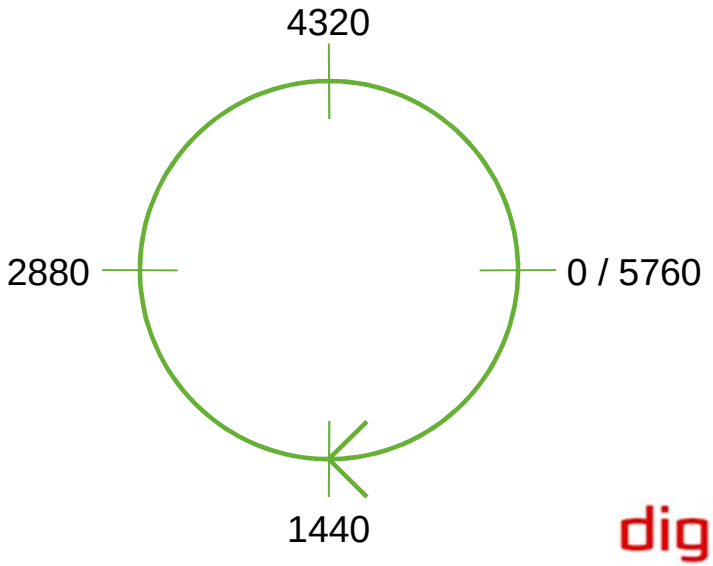


- QPainter::drawRect

- QPainter::drawRoundedRect

Basic Shapes

- `QPainter::drawEllipse` 
- `QPainter::drawArc` 
- `QPainter::drawPie` 
- The arc and pie angles are specified as 16ths of degrees, zero degrees at three o'clock growing clock-wise



Painting Text

- QPainter::drawText

```
QPainter p(this);

QFont font("Helvetica");
p.setFont(font);
p.drawText(20, 20, 120, 20, 0, "Hello World!");

font.setPixelSize(10);
p.setFont(font);
p.drawText(20, 40, 120, 20, 0, "Hello World!");

font.setPixelSize(20);
p.setFont(font);
p.drawText(20, 60, 120, 20, 0, "Hello World!");

QRect r;
p.setPen(Qt::red);
p.drawText(20, 80, 120, 20, 0, "Hello World!", &r);
```

Hello World!

Hello World!

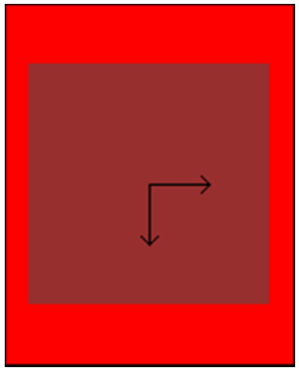
Hello World!

Hello World!

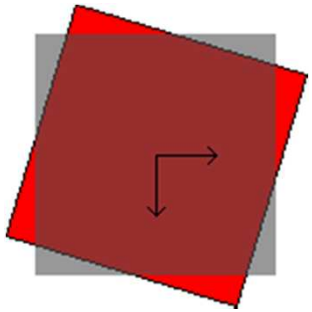
The rectangle `r` represents the extent of the text

Transformations

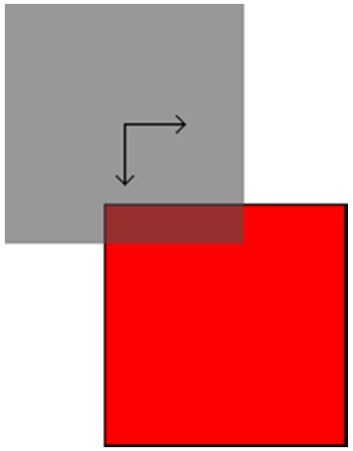
- QPainter::scale



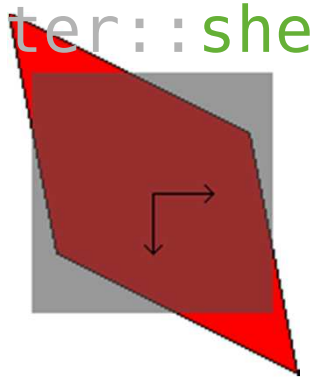
- QPainter::rotate



- QPainter::translate

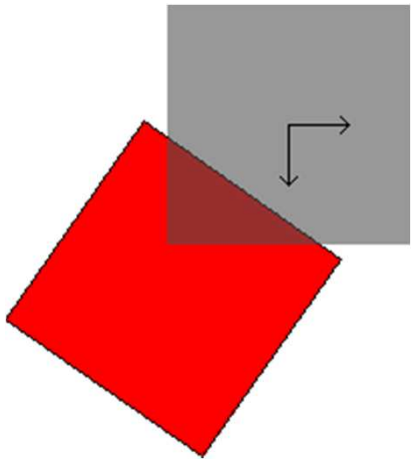


- QPainter::shear

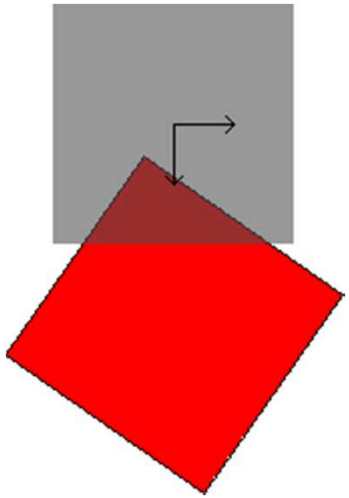


Transformations

- Order of transformations is important
- Origin is important when scaling, rotating and shearing



```
p.translate(0, 100);
p.rotate(35);

p.drawRect(-60, -60, 120, 120);
```



```
p.rotate(35);
p.translate(0, 100);

p.drawRect(-60, -60, 120, 120);
```

Transformations

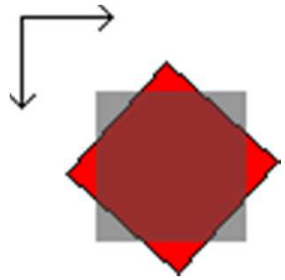
- Using `save` and `restore`, transformation states can be kept on a stack
- Example, rotating around an arbitrary point

```
QPoint rotCenter(50, 50);
qreal angle = 42;

p.save();
p.translate(rotCenter);
p.rotate(angle);
p.translate(-rotCenter);

p.setBrush(Qt::red);
p.setPen(Qt::black);
p.drawRect(25,25, 50, 50);

p.restore();

p.setPen(Qt::NoPen);
p.setBrush(QColor(80, 80, 80, 150));
p.drawRect(25,25, 50, 50);
```

Apply transformations

Red rectangle

Gray rectangle

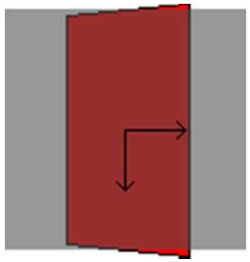


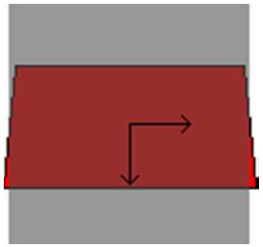
digia

2.5D Transformations

- When rotating, it is possible to rotate about any axis, creating a 3D effect

```
QTransform t;
t.rotate(60, Qt::YAxis);

painter.setTransform(t, true);
```





A Gauge

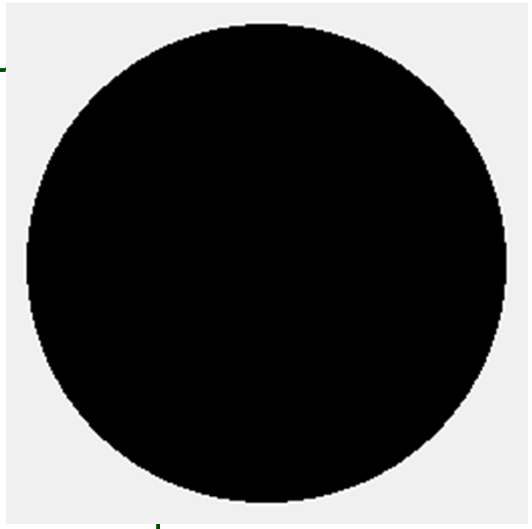
- An example of a custom widget: `CircularGauge`
- Works as the `CombinedGauge`, but is a truly custom widget
  - Same API as `CombinedGauge`, the value property
  - Custom painting
  - Can interact with
    - keyboard
    - mouse

A Gauge

- Painting the gauge background

```
void CircularGauge::paintEvent(QPaintEvent *ev)
{
    QPainter p(this);

    int extent;
    if (width()>height())
        extent = height()-20;
    else
        extent = width()-20;

    p.translate((width()-extent)/2, (height()-extent)/2);

    p.setPen(Qt::white);
    p.setBrush(Qt::black);

    p.drawEllipse(0, 0, extent, extent);

    ...
```

Centering the gauge in the available area

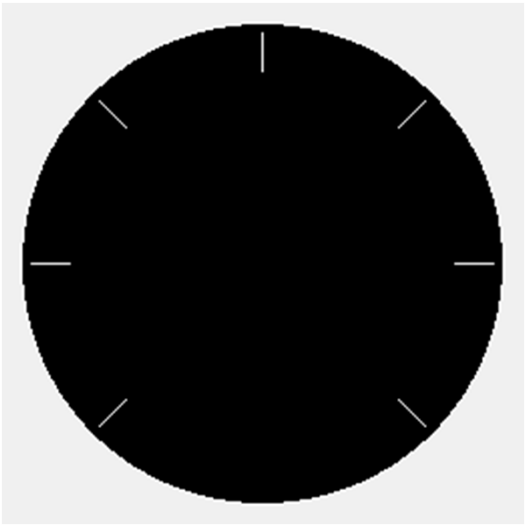


Drawing the background circle

A Gauge

- Painting the scale around the edge of the gauge

```
void CircularGauge::paintEvent(QPaintEvent *ev)
{
    ...

    p.translate(extent/2, extent/2);
    for(int angle=0; angle<=270; angle+=45)
    {
        p.save();
        p.rotate(angle+135);
        p.drawLine(extent*0.4, 0, extent*0.48, 0);
        p.restore();
    }

   ...
```



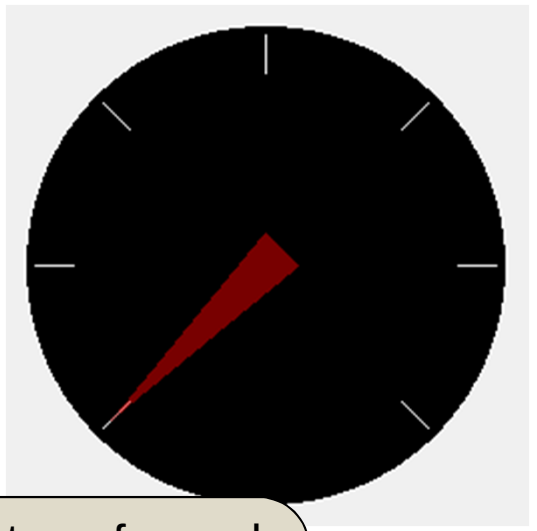
Notice the `save` and `restore` pair inside the loop.

Simply calling `rotate(45)` accumulates a potential rounding error.

A Gauge

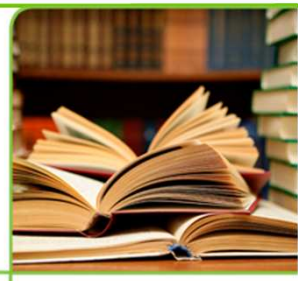
- Painting the needle

```
void CircularGauge::paintEvent(QPaintEvent *ev)
{
    ...

    p.rotate(m_value+135);
    QPolygon polygon;
    polygon << QPoint(-extent*0.05, extent*0.05)
            << QPoint(-extent*0.05, -extent*0.05)
            << QPoint(extent*0.46, 0);
    p.setPen(Qt::NoPen);
    p.setBrush(QColor(255,0,0,120));
    p.drawPolygon(polygon);
}
```



The arrow is, untransformed, pointing left and positioned around the origin

Acting on Events



- There are more events than the paint event
  - Keyboard events
  - Mouse events
  - window events, touch events, gesture events, timer events, change events, accessibility events, clipboard events, layout events, drag events, etc.

Reacting to Keys

- Re-implement the protected `keyPressEvent`
- Act on the key being pressed
- Pass non-used keys to the base class

```
void CircularGauge::keyPressEvent(QKeyEvent *ev)
{
    switch(ev->key())
    {
    case Qt::Key_Up:
    case Qt::Key_Right:
        setValue(value()+1);
        break;
    case Qt::Key_Down:
    case Qt::Key_Left:
        setValue(value()-1);
        break;
    case Qt::Key_PageUp:
        setValue(value()+10);
        break;
    case Qt::Key_PageDown:
        setValue(value()-10);
        break;
    default:
        QWidget::keyPressEvent(ev);
    }
}
```

Reacting to the Mouse

- Mouse events are handled through overriding the following protected methods
  - `mousePressEvent` and `mouseReleaseEvent`
  - `mouseMoveEvent` – only called while a button is pressed unless `mouseTracking` is enabled
- `setValueFromPos` is a private method for converting a point into an angle

```
void CircularGauge::mousePressEvent(QMouseEvent *ev)
{
    setValueFromPos(ev->pos());
}

void CircularGauge::mouseReleaseEvent(QMouseEvent *ev)
{
    setValueFromPos(ev->pos());
}

void CircularGauge::mouseMoveEvent(QMouseEvent *ev)
{
    setValueFromPos(ev->pos());
}
```

Drawing less is quicker

- The `paintEvent` method takes a `QPaintEvent` as argument
- The `QPaintEvent` has two methods
  - `QRect rect` – returns the rectangle needing repainting
  - `QRegion region` – returns the region needing repainting
- A region is more complex than a rectangle
- When re-painting, try to avoid drawing complex shapes outside the rectangle / region

QTimer

- `QTimer` is used to let the clock generate events

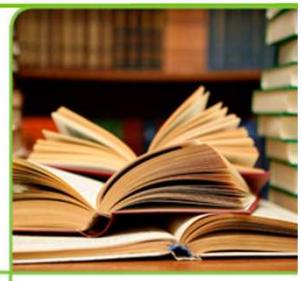
```
MyClass(QObject *parent) : QObject(parent)
{
    QTimer *timer = new QTimer(this);
    timer->setInterval(5000);
    connect(timer, SIGNAL(timeout()), this, SLOT(doSomething()));
    timer->start();
}
```

Every 5000ms
i.e. every 5s

- Or to delay an action
  - passes through the event loop, can be used to queue slot calls

```
QTimer::singleShot(1500, dest, SLOT(doSomething());
```

The Event Mechanism



- All events are posted on the Qt event queue
- In the event queue, they can be processed
  - Only the last `mouseMoveEvent` will be delivered
  - Multiple `paintEvent` requests can be merged
- When a `QObject` receives an event, the `event` method is activated
  - The `event` method can either `accept` or `ignore` the event
  - Ignored events are propagated through the object hierarchy

Filtering Events

- It is possible to install event filters on a `QObject`
- The filter itself is a `QObject` that implements the `eventFilter` method

- An event filter receives the watched object's events and can let them through or stop them
  - Can be used to add functionality to an object without sub-classing
  - Can be used to prevent a given event from reaching its target

Filtering Events

- Implementing a filter for the gauges
  - Adds a function: Pressing 0 (zero) zeroes the value

```
class KeyboardFilter : public QObject ...
```

```
bool KeyboardFilter::eventFilter(QObject *o, QEvent *ev)
{
    if (ev->type() == QEvent::KeyPress)
        if (QKeyEvent *ke = static_cast<QKeyEvent*>(ev))
            if (ke->key() == Qt::Key_0)
                if (o->metaObject()->indexOfProperty("value") != -1 )
                {
                    o->setProperty("value", 0);
                    return true;
                }

    return false;
}
```

`true` uses the event, i.e. it is not passed on to the watched object

Installing the filter

- Activating the filter is as easy as calling `installEventFilter`

```
ComposedGauge compg;
CircularGauge circg;

KeyboardFilter filter;

compg.installEventFilter(&filter);
circg.installEventFilter(&filter);
```

- As the filter works on the property, not a particular class, it can be used with `QSlider`, `QDial`, `QSpinBox`, etc.

- The brave can install an event filter on the `QApplication`

Break

digia

Style Aware Widgets

- Qt paints widgets using different styles on different platforms
  - Control the style used from the command line

```
./myapplication -style name-of-style
```

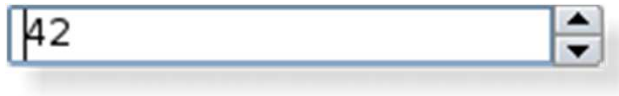
- For widgets to properly integrate across platforms they must be made style aware
  - Build from standard elements
  - Use platform specific elements when painting
  - Asking the platform style for sizes

Avoiding Style Awareness

- Style awareness means that you have to make your widget fit in across all platforms
- Consider using existing widgets
  - Directly or in composed widgets
- Use QFrame as base
  - If the content is independent of style

The Structure of Styles

- Complex controls
  - Sub-controls
- Primitive elements
- Control elements
- Metrics
- Standard pixmaps



Style Options

- When painting elements using QStyle, a QStyleOption instance is used to convey information such as
  - Font
  - Palette
  - Rectangle on screen
  - State (active, has focus, is selected, etc)
  - Element specific settings (e.g. icon and text)

Painting using styles

```
void Widget::paintEvent(QPaintEvent *event)
{
    QPainter painter(this);

    QStyleOptionFocusRect option;
    option.initFrom(this);
    option.backgroundColor = palette().color(QPalette::Background);

    style()->drawPrimitive(QStyle::PE_FrameFocusRect,
        &option, &painter, this);
}
```

QStylePainter

- The QStylePainter class encapsulates the QPainter and QStyle

```
void MyWidget::paintEvent(QPaintEvent *event)
{
    QStylePainter painter(this);

    QStyleOptionFocusRect option;
    option.initFrom(this);
    option.backgroundColor = palette().color(QPalette::Background);

    painter.drawPrimitive(QStyle::PE_FrameFocusRect, option);
}
```

Widgets in Designer

- Having created custom widgets, you can include them in Designer
  - Write a plugin based on implementing `QDesignerCustomWidgetInterface`
  - Read more at http://doc.trolltech.com/designer-creating-custom-widgets.html and http://doc.trolltech.com/4.6/designer-customwidgetplugin.html

Classes around Painter

- When using QPainter, you encounter a number of surrounding classes
  - QColor – represents a color, including transparency
  - QPen – represents a pen used for stroking outlines
  - QBrush – represents a brush for filling interiors

QColor

- The constructor of QColor takes three colors and an alpha channel
  - The alpha channel controls how transparent or opaque the color is

```
QColor( int r, int g, int b, int a )
```

- Qt provides a range of predefined colors

digia

Color Spaces

- The RGB colorspace is commonly used for computers, but there are more colorspaces
  - CMYK – commonly used in printing
  - HSV / HSL – used in color pickers, etc
- QColor can be set from any of these colorspaces using static functions
  - QColor::fromCmyk
  - QColor::fromHsl
  - QColor::fromHsv

Color Spaces cont'd

- The values of the individual components of RGB, CMYK, HSL and HSV colors can be read using
  - getRgb, getCmyk, getHsl, getHsv

```
getRgb(int *r, int *g, int *b)
```

- The can also be read individually using
  - red, green, blue, cyan and magenta, yellow, black
  - hslHue, hslSaturation, lightness
  - hsvHue, hsvSaturation, value
  - alpha

Tuning Colors

- The QColor class lets you create lighter and darker colors
  - QColor::lighter(int factor)
  - QColor::darker(int factor)

QRgb

- The QColor class is great for representing colors, but when storing colors, a more compact alternative is needed

- QRgb is a 32-bit color triplet with alpha (RGB+A)

QRgb

- Create new QRgb values using qRgb and qRgba

```
QRgb orange = qRgb(255, 127, 0);
QRgb overlay = qRgb(255, 0, 0, 100);
```

- Read components using qRed, qGreen, qBlue, qAlpha

```
int red = qRed(orange);
```

- Convert to gray scale using qGray
  - Not the average value – weighted by luminance

```
int gray = qGray(orange);
```

Pens

- When stroking outlines of shapes, a QPen is used.
- A pen defines properties such as color, width and line style
- Pens can be cosmetic, i.e. not affected by transformations
  - Set using setCosmetic(bool)
  - Can greatly improve performance

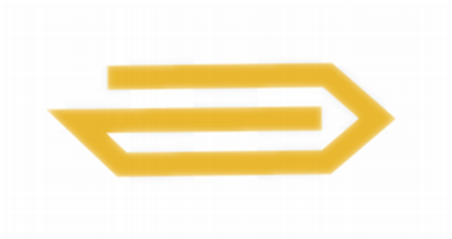
Line Styles

- The line style is set :Style method
  - Qt::SolidLine
  - Qt::DashLine
  - Qt::DotLine
  - Qt::DashDotLine
  - Qt::DashDotDotLine
  - Qt::CustomDashLine – controlled by dashPattern

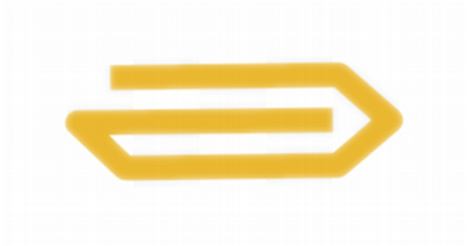
Joining and Ending Lines

- joinStyle
  - Qt::BevelJoin (default)

  - Qt::MiterJoin



  - Qt::RoundJoin



capStyle

Qt::SquareCap (default)

Qt::FlatCap

Square covers the end point flat does not cover the end

Qt::RoundCap

Brushes

- Brushes are used for filling the interior of shapes
- There are several types of brushes, all available through the QBrush class
- They can be divided into the following groups
  - Solid
  - Patterned
  - Textured
  - Gradients

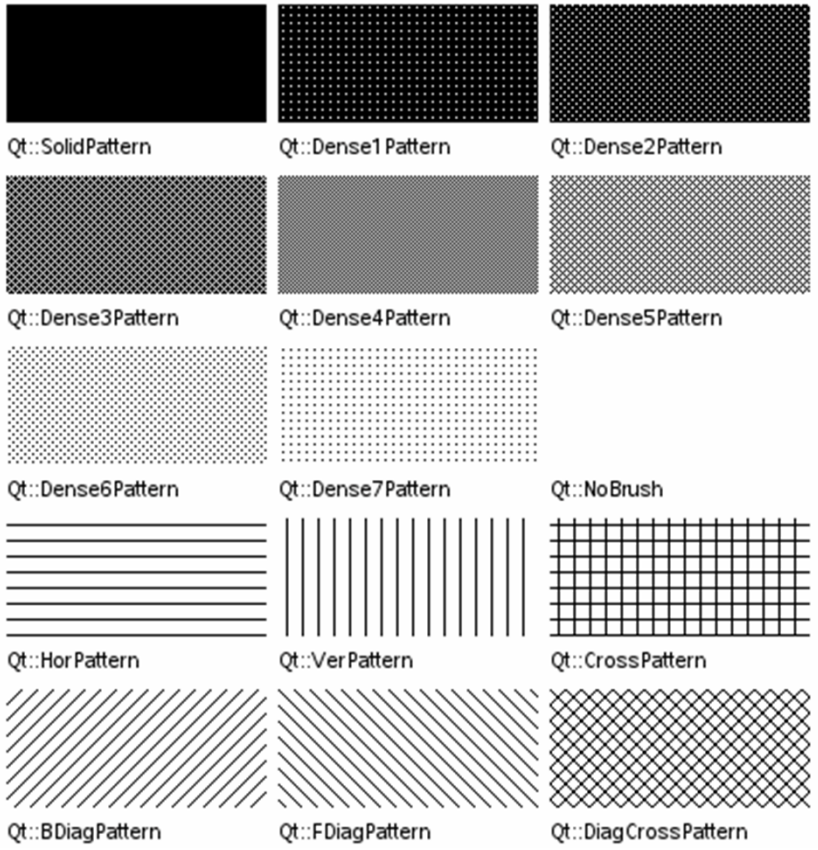
Solid Brushes

- Solid, single color, brushes are created by giving the QBrush constructor a color as argument

```
QBrush red(Qt::red);

QBrush odd(QColor(55, 128, 97));
```

Patterned Brushes

- A solid brush is really an instance of a patterned brush, but with a different brushStyle



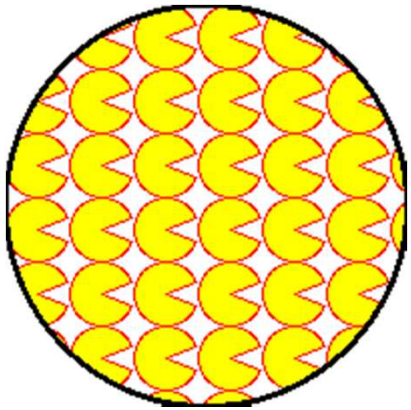

```
QBrush( const QColor &color, Qt::BrushStyle style )
```

Textured Brushes

- A textured brush uses a QPixmap as texture

```
QBrush( const QPixmap &pixmap )
```

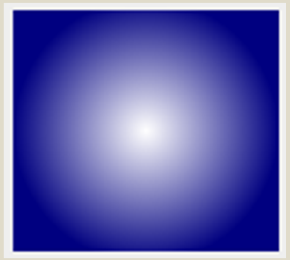
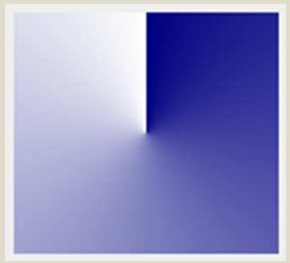
```
QPixmap pacPixmap("pacman.png");

painter.setPen(QPen(Qt::black, 3));
painter.setBrush(pacPixmap);
painter.drawEllipse(rect());
```



If the texture is monochrome,
the color of the brush is used.

Otherwise the pixmap's colors are used.

Gradients

QLinearGradient	QRadialGradient	QConicalGradient
		

- Create a QBrush by passing a QGradient object to it, e.g.

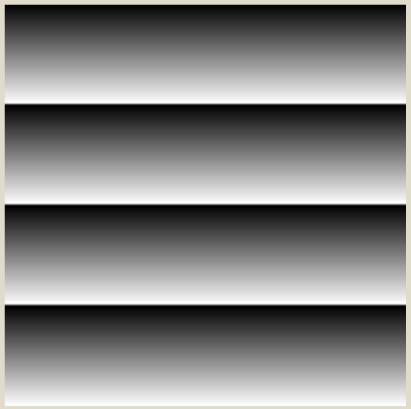
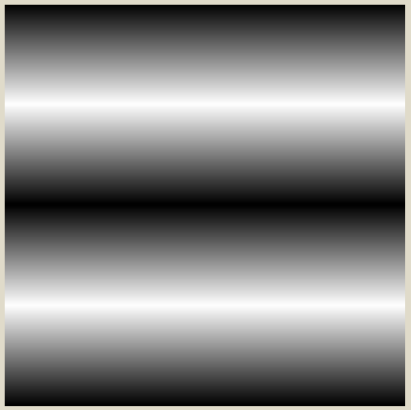
```
QBrush b = QBrush( QRadialGradient( ... ) );
```

A Generic Gradient

- Divides the distance from a start point to a end point in the 0.0 to 1.0 range

```
QGradient::setColorAt( qreal pos, QColor );
```

- Spread the colors outside the interval

PadSpread (default)	RepeatSpread	ReflectSpread
		

digia

Example: a linear gradient

```
setColorAt(0.0,
    QColor(0, 0, 0))
```

```
setColorAt(0.7,
    QColor(255, 0, 0))
```

```
setColorAt(1.0,
    QColor(255, 255, 0))
```

Using pens and brushes

- To avoid filling or stroking, clear the pen or brush

```
QPainter p;
p.setPen(Qt::NoPen);
p.setBrush(Qt::NoBrush);
```

- It can be costly to change pen and brush
  - Plan you painting to gain performance.

Text

- Painting text can be a complex task
  - Font sizes
  - Alignment
  - Tabs
  - Wrapping
  - Flowing around images
  - Left-to-right and right-to-left

QPainter and Text

- Basic painting of text

```
drawText( QPoint, QString )
```

- Painting of text with options

```
drawText( QRect, QString, QTextOptions )
```

- Painting of text with feedback

```
drawText( QRect, flags, QString, QRect* )
```

digia

Fonts

- The QFont class represents a font
  - Font family
  - Size
  - Bold / Italic / Underline / Strikeout / etc

Font Family

- Create new QFont instances by specifying the font name to the c'tor

```
QFont font("Helvetica");

font.setFamily("Times");
```

- Use QFontDatabase::families to get a list of available fonts.

```
QFontDatabase database;
QStringList families = database.families();
```

digia

Font Size

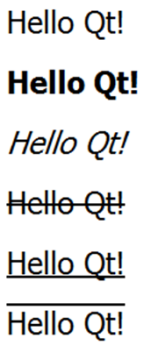
- Fonts can either be sized using pixel size or point size

```
QFont font("Helvetica");

font.setPointSize(14); // 12 points high
                       // depending on the paint device's dpi

font.setPixelSize(10); // 10 pixels high
```

- Notice that the pixelSize == -1 if the size was set using setPointSize and vise versa

Font Effects

- Font effects can be enabled or disabled



Normal, bold, italic, strike out, underline, overline

- QWidget::font and QPainter::font returns a const QFont reference, i.e. you must modify a copy

```
QFont tempFont = w->font();
tempFont.setBold( true );
w->setFont( tempFont );
```

digia

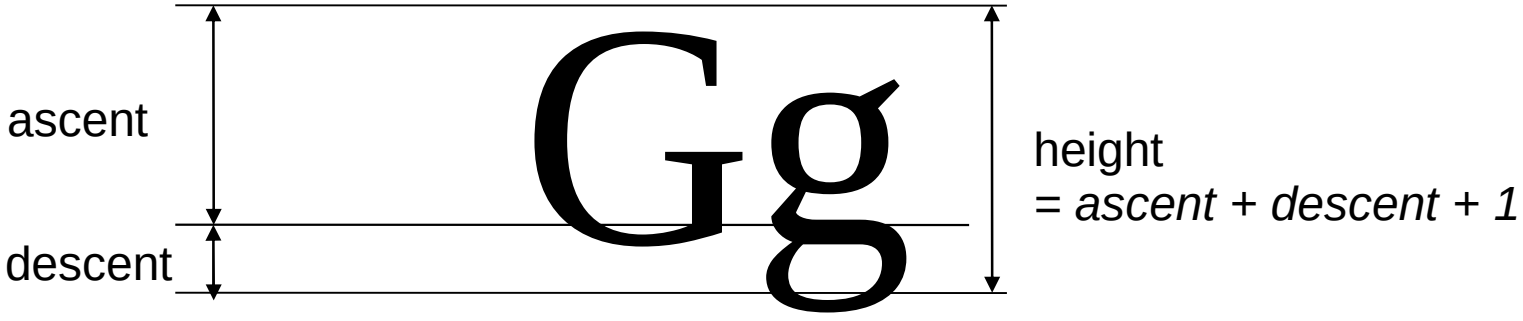
Measuring Text

- It is interesting to know how large a text will be before painting it
  - QFontMetrics is used to measure text and fonts
  - The boundingRect function makes it easy to measure the size of a text block

```
QImage image(200, 200, QImage::Format_ARGB32);
QPainter painter(&image);
QFontMetrics fm(painter.font(), &image);

qDebug("width: %d", fm.width("Hello Qt!"));
qDebug("height: %d", fm.boundingRect(0, 0, 200, 0,
    Qt::AlignLeft | Qt::TextWordWrap, loremIpsum).height());
```

Measuring Text

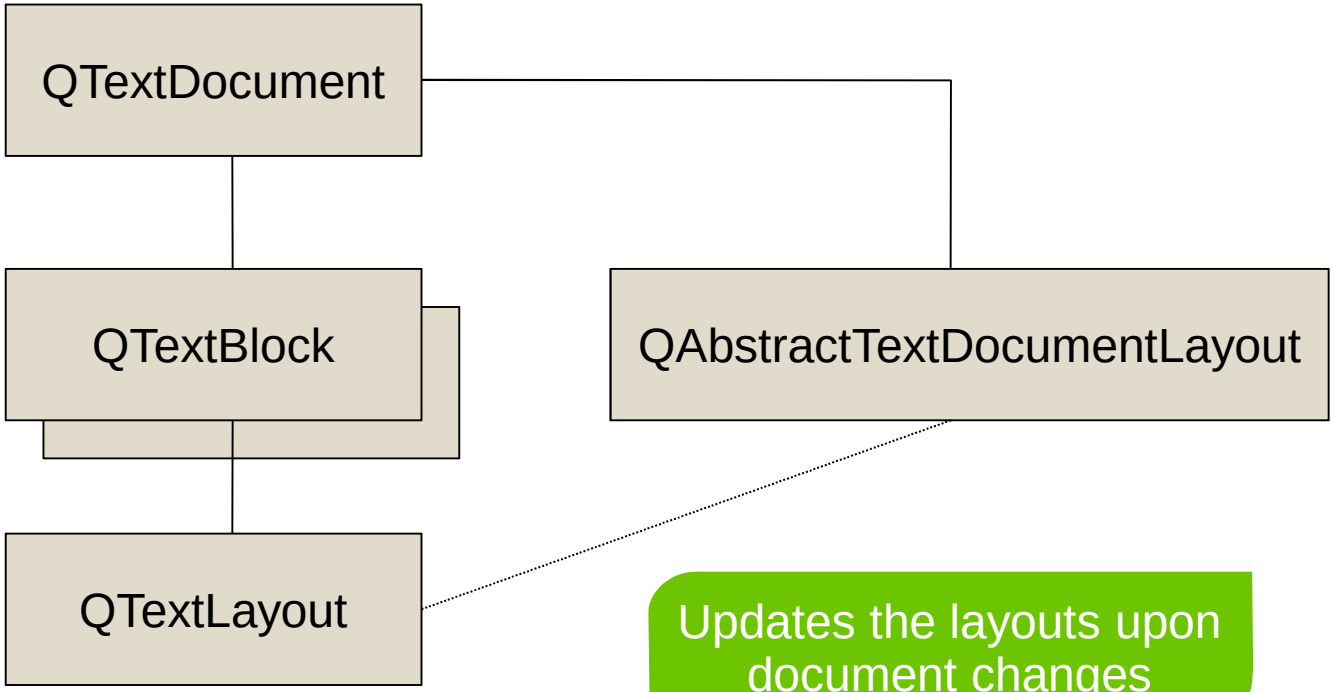
- These measurements are useful when aligning text with other graphics



Working with Documents

- The QTextDocument class is used to handle rich text documents
  - Consists of blocks of text, QTextBlock
  - Lays out the text using a QAbstractTextDocumentLayout layout engine
  - Using the standard layout engine, it is possible to render rich text to any QPainter

The Document Classes



Used for traversing and modifying documents

QTextCursor

Painting with Text Documents

- Use the textWidth property to control the width
  - Read the resulting height using size
- Use the pageSize property to control pagination
  - pageCount holds the resulting number of pages

Working with Text Documents

- Use drawContents to draw the contents of a document using a QPainter

```
QPainter painter;

QTextDocument doc;
doc.setTextWidth(width());
doc.drawContents(&p, rect());
```

Lorem Ipsum

Lorem ipsum dolor sit amet, consectetur adipiscing elit. **Aliquam** nec purus egestas odio dictum molestie. Nulla quis urna libero, non venenatis risus. Nunc nisl lacus, vehicula vitae consectetur in, euismod vel elit. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Nam pulvinar tempor sapien in placerat. Fusce sagittis arcu eu metus pharetra pellentesque. Praesent lacinia mollis augue vel dignissim. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Nam interdum neque metus, at mollis leo. Morbi a lectus nulla. Curabitur nec sem eros. Suspendisse sit amet nunc nunc, consectetur venenatis felis. Etiam vulputate auctor tempor. Maecenas viverra vestibulum nisi ac fermentum.

- Use setTextWidth to limit the width of the text

Images

QPixmap	QImage
Optimized for showing images on-screen	*Optimized for manipulation*

- If you plan on painting a QImage to the screen even twice, it is better to convert it to a QPixmap first

Converting

- Conversion between the QImage and QPixmap is handed in QPixmap

```
QImage QPixmap::toImage();

QPixmap QPixmap::fromImage( const QImage& );
```

Loading and Saving

```
QPixmap pixmap( "image.png" );
pixmap.save( "image.jpeg" );
```

```
QImage image( "image.png" );
image.save( "image.jpeg" );
```

This code uses the QImageReader and QImageWriter classes. These classes determine the image file format from extension when saving.

Painting to a QImage

- The QImage is a QPaintDevice, so a QPainter can paint on it

```
QImage image( 100, 100, QImage::Format_ARGB32 );
QPainter painter(&image);

painter.setBrush(Qt::red);

painter.fillRect( image.rect(), Qt::white );
painter.drawRect(
    image.rect().adjusted( 20, 20, -20, -20 ) );
```

Painting a QPixmap

- QPixmap is optimized for being painted onto the screen

```
void MyWidget::imageChanged( const QImage &image )
{
    pixmap = QPixmap::fromImage( image );
    update();
}

void MyWidget::paintEvent( QPaintEvent* )
{
    QPainter painter( this );
    painter.drawPixmap( 10, 20, pixmap );
}
```

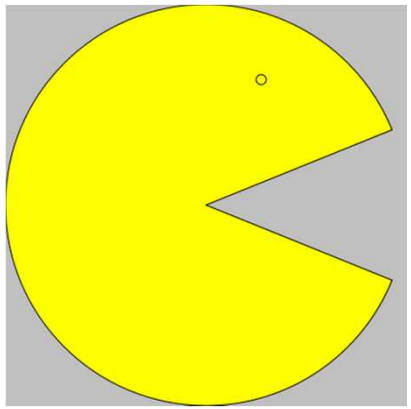
Scalable Vector Graphics

- The SVG file format is a W3C standard for describing vector graphics using XML
- Qt supports both generating and reading SVG files
- All SVG related classes reside in the QtSvg module

Rendering SVG

- Using the QSvgRenderer and the load and render methods, it is possible to render an SVG file to a QPainter

```
QPainter painter;

QSvgRenderer renderer;
renderer.load(QString("svg-test.svg"));

renderer.render(&painter);
```



- Use the defaultSize and viewBox methods to determine the size of the rendered graphics

Generating SVG

- To generate SVG files, use a QSvgGenerator as the QPaintDevice and open a QPainter to it

```
QSvgGenerator generator;
generator.setFileName("svg-test.svg");
generator.setSize(QSize(200, 200));
generator.setViewBox(QRect(0, 0, 200, 200));

QPainter p;
p.begin(&generator);

p.setPen(Qt::black);
p.setBrush(Qt::yellow);
p.drawPie(0, 0, 200, 200, 22*16, 316*16);
p.drawEllipse(125, 35, 5, 5);

p.end();
```