

Qt Training – Widget Edition

December 2016
Based on Qt 5.8

The Qt Company, 2017

Contents

| | |
|-------------------------------|--|
| Widgets | Common Widgets Layout Management Guidelines for Custom Widgets |
| State Machines and Animations | Animation Framework Animation Groups State Machines State-Based User Interfaces |
| Painting and Styling | Painting on Widgets Color Handling Painting Operations Styles and Style Sheets |
| Application Creation | Main Windows Settings Resources Translation for Developers Deploying Qt Applications |

Contents

| | |
|----------------------|--|
| Dialogs | Dialogs Common Dialogs Qt Designer |
| Model/View Framework | Model/View Concept Showing Simple Data Proxy Models Custom Models Delegates Editing item data Data Widget Mapper Drag and Drop Custom Tree Model Data Widget Mapper |

Contents

| | |
|--------------|---|
| GraphicsView | Using GraphicsView Classes Coordinate Systems and Transformations Creating Custom Items Widgets in a Scene Drag and Drop Effects Performance Tuning |
| Charts | Chart Types Using Qt Model Data in Charts Dynamic Charts User Interactions Themes and Customization |

Qt GUI Options

| UI | Typical Use Cases | Features |
|---------------|--|---|
| Widgets | Desktop-like UIs with menus, toolbars, status bars, dialogs | No GPU needed Memory efficient Custom styling |
| Graphics View | Mostly prior Qt5 for lightweight, rich UIs | Memory efficient Can handle thousands of UI items No UI controls |
| 3D | 3D UI, games, visualizations | 3D engine, 3D data visualization |
| Qt Quick | Rich UIs with fancy graphical effects, desktop, embedded, mobile | Fluid user experience easy to achieve Declarative programming Rendering optimizations Memory intensive Multicore CPU and GPU beneficial |
| Web Engine | Web-based UI, hybrid apps | No need to learn C++ or QML Existing web-based UIs easy to use with Qt Easy to expose Qt objects to JavaScript |

Contents

- › Common Widgets
- › Layout Management
- › Guidelines for Custom Widgets

Objectives

› **Common Widgets**

- › Text widgets
- › Value based widgets
- › Organizer widgets
- › Item-based widgets

› **Layout Management**

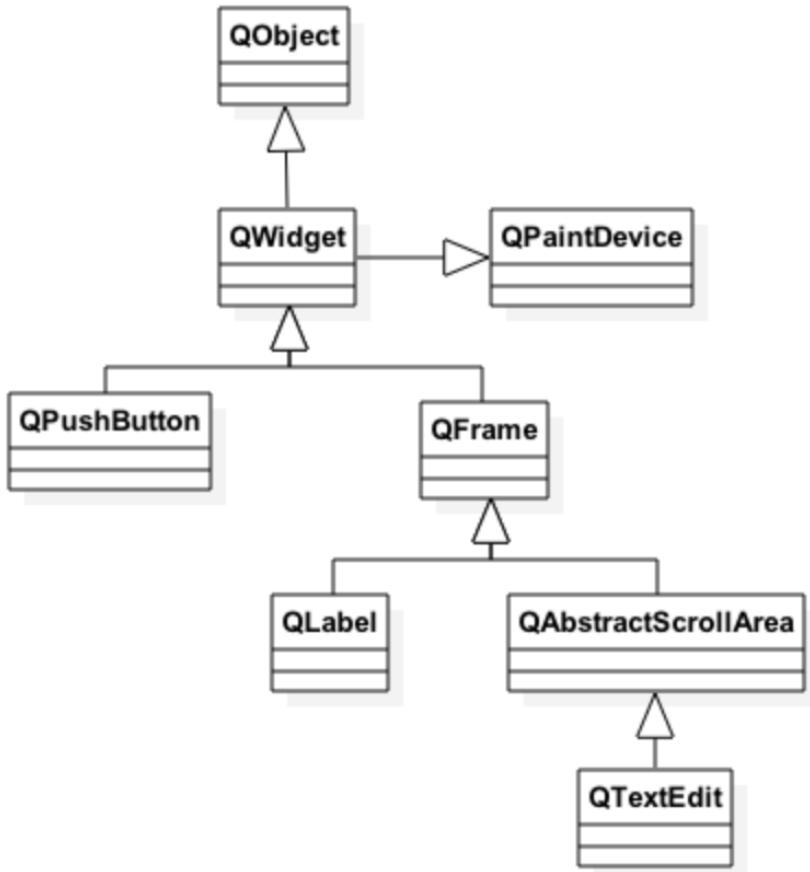
- › Geometry management
- › Advantages of layout managers
- › Qt's layout managers
- › Size policies

› **Custom Widgets**

- › Rules for creating own widgets

Qt's Widget Model - QWidget

- › **Derived from** QObject
 - › Adds visual representation
- › **Base of user interface objects**
- › **Receives events**
 - › e.g. mouse, keyboard events
- › **Paints itself on screen**
 - › Using styles



Object Tree and QWidget

- › new QWidget(0)
 - › Widget with no parent = "window" – QWindow created in the platform abstraction
 - › AKA top-level widget
- › QWidget's children
 - › Positioned in parent's coordinate system
 - › Clipped by parent's boundaries
- › QWidget parent
 - › Propagates state changes
 - › Hides/shows children when it is hidden/shown itself
 - › Enables/disables children when it is enabled/disabled itself
- › ***Tristate mechanism***
 - › For hide/show and enable/disable, ensures that e.g. an explicitly hidden child is not shown when the parent is shown.

Text Widgets

› QLabel

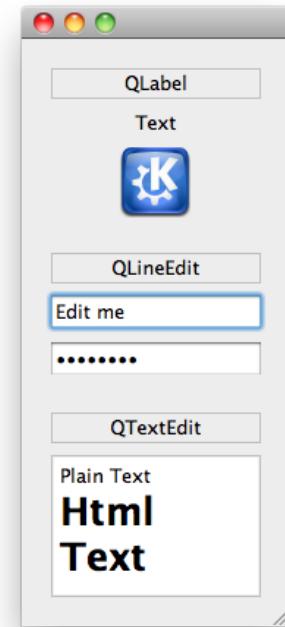
```
label = new QLabel("Text", parent);  
    > setPixmap( pixmap ) - as content
```

› QLineEdit

```
line = new QLineEdit(parent);  
line->setText("Edit me");  
line->setEchoMode(QLineEdit::Password);  
connect(line, SIGNAL(textChanged(QString)) ...  
connect(line, SIGNAL(editingFinished()) ...  
    > setInputMask( mask )  
    > setValidator( validator )
```

› QTextEdit

```
edit = new QTextEdit(parent);  
edit->setPlainText("Plain Text");  
edit->append("<h1>Html Text</h1>");  
connect(edit, SIGNAL(textChanged(QString)) ...
```



Button Widgets

> **QAbstractButton**

> Abstract base class of buttons

> **QPushButton**

```
button = new QPushButton("Push Me", parent);
button->setIcon(QIcon("images/icon.png"));
connect(button, SIGNAL(clicked())) ...
> setCheckable(bool)-togglebutton
```

> **QRadioButton**

```
radio = new QRadioButton("Option 1", parent);
```

> **QCheckBox**

```
check = new QCheckBox("Choice 1", parent);
```

> **QButtonGroup** - non-visual button manager

```
group = new QButtonGroup(parent);
group-> addButton(button); // add more buttons
group-> setExclusive(true);
connect(group, SIGNAL(buttonClicked(QAbstractButton*)) )
```



Value Widgets

> **QSlider**

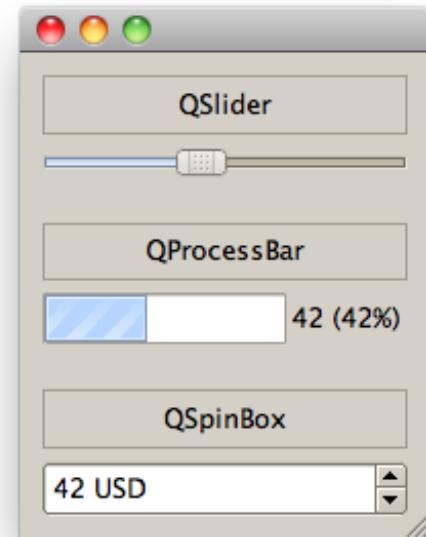
```
> slider = new QSlider(Qt::Horizontal, parent);  
> slider->setRange(0, 99);  
> slider->setValue(42);  
> connect(slider, SIGNAL(valueChanged(int)) ...
```

> **QProgressBar**

```
> progress = new QProgressBar(parent);  
> progress->setRange(0, 99);  
> progress->setValue(42);  
  // format: %v for value; %p for percentage  
> progress->setFormat("%v (%p%)");
```

> **QSpinBox**

```
> spin = new QSpinBox(parent);  
> spin->setRange(0, 99);  
> spin->setValue(42);  
> spin->setSuffix(" USD");  
> connect(spin, SIGNAL(valueChanged(int)) )
```



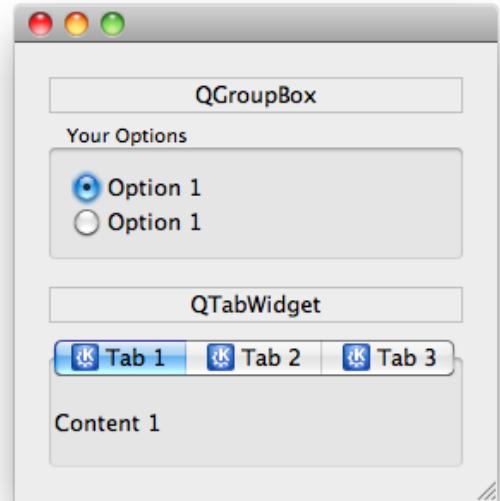
Organizer Widgets

> **QGroupBox**

```
box = new QGroupBox("Your Options", parent);  
// ... set layout and add widgets  
> setCheckable( bool ) - checkbox in title
```

> **QTabWidget**

```
> tab = new QTabWidget(parent);  
> tab->addWidget(widget, icon, "Tab 1");  
> connect(tab, SIGNAL(currentChanged(int)) ...  
> setCurrentWidget( widget )  
    > Displays page associated by widget  
> setTabPosition( position )  
    > Defines where tabs are drawn  
> setTabsClosable( bool )  
    > Adds close buttons
```



Item Widgets

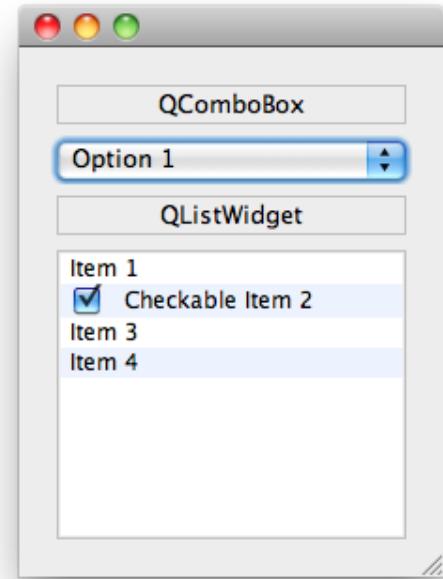
› **QComboBox**

```
combo = new QComboBox(parent);
combo->addItem("Option 1", data);
connect(combo, SIGNAL(activated(int)) ...
QVariant data = combo->itemData(index);
    > setCurrentIndex( index )
```

› **QListWidget**

```
list = new QListWidget(parent);
list->addItem("Item 1");
// ownership of items with list
item = new QListWidgetItem("Item 2", list); item->setCheckState(Qt::Checked);
connect(list, SIGNAL(itemActivated(QListWidgetItem*)) ...
    > QListWidgetItem::setData(Qt::UserRole, data)
```

› *Other Item Widgets:* QTableWidgetItem, QTreeWidget



Other Widgets

- › **QToolBox**

- › Column of tabbed widget items

- › **QDateEdit, QTimeEdit, QDateTimeEdit**

- › Widget for editing date and times

- › **QCalendarWidget**

- › Monthly calendar widget

- › **QToolButton**

- › Quick-access button to commands

- › **QSplitter**

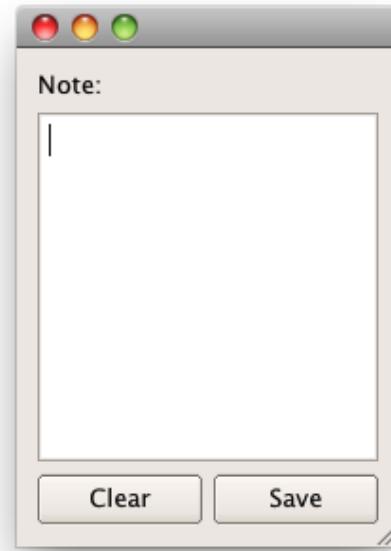
- › Implements a splitter widget

- › **QStackedWidget**

- › Stack of widgets
 - › Only one widget visible at a time

Widgets that Contain Other Widgets

- › Container Widget
 - › Aggregates other child-widgets
- › Use layouts for aggregation
 - › In this example: QHBoxLayout and QVBoxLayout
 - › Note: Layouts are *not* widgets
- › Layout Process
 - › Add widgets to layout
 - › Layouts may be nested
 - › Set layout on container widget



Container Widget: QScrollArea

- › QScrollArea provides on-demand scrollbars for a single child widget
- › QScrollArea works by scrolling a single (large) child widget provided by the programmer
- › QAbstractScollArea is the base class of QScrollArea and can be used when full control over the scrollbars and scrolling process is required
- › Most window systems limit the size of a widget in both directions to be smaller than 2^{16} , but Qt deals with this limitation in a way that is transparent to the programmer
 - › It is no problem to have a huge widget (up to $2^{31}-1$ pixels in each direction) inside a QScrollArea

QScrollArea

- › Create an instance of `QScrollArea` and the widget that should be inside the scrolled area
- › Call `QScrollArea::setWidget()` with a pointer to the widget. Note that the scroll area takes ownership of the widget.
- › If the widget is a plain `QWidget`, you'll have to set its minimum size to avoid that it shrinks beyond intention
- › Add a layout manager to the widget, and add subwidgets at your leisure
- › Optionally call `QScrollArea::setWidgetResizable(true)`
 - › This will make this widget resize to the size of the viewport in case it is smaller than the viewport

Organization

- › Scrollbars will by default be shown when needed. This behavior can, however, be configured using `setVerticalScrollBarPolicy()` and `setHorizontalScrollBarPolicy()`
- › The value passed to the two methods is one of
 - › `Qt::ScrollBarAsNeeded`
 - › `Qt::ScrollBarAlwaysOff`
 - › `Qt::ScrollBarAlwaysOn`
- › Space can be reserved outside the viewport, but still in between the scroll bars, using `setViewportMargins()`

Container Widget: QMdiArea

- › QMdiArea – Window manager for sub-windows
 - › Supports tiling and cascading sub-windows
 - › Provides slots to add, remove, and activate sub-windows
 - › QAbstractScrollArea subclass (by default scroll bars off)
 - › Subclass and use the viewport to define the visible area size
 - › Two view modes: subwindow with frames and tabbed mode
- › QMdiSubWindow
 - › Document (QWidget) container
 - › Manage your windows with “standard” window methods (`show()`, `showMaximized()`, `showFullScreen()`, `hide()`, `move()`, etc.)
 - › Provides a sub-window list (in the creation, stacking or activation history order)

Layout Management – Doing it Yourself

- › Place and resize widgets

- › move()
- › resize()
- › setGeometry()

- › Example:

```
QWidget *parent = new QWidget(...);
parent->resize(400, 400);

QCheckBox *cb = new QCheckBox(parent);
cb->move(10, 10);
```

Layout Management – Let Qt Handle the Layout

- › Define the position and size for top-level widgets only
 - › `setGeometry()`, `resize()` or `move()`
- › Manage the layout of child widgets with layout classes
 - › Makes your UI scalable
 - › Takes LTR and RTL into account
 - › Takes font sizes into account
- › All layout classes subclass `QLayout`
 - › Not a widget itself
 - › `QHBoxLayout`, `QVBoxLayout`, `QStackedLayout`, `QGridLayout`, `QFormLayout`
 - › Some widgets use directly the layouts: `QTabWidget` and `QStackedWidget` use `QStackedLayout`

Managed Widgets and Sizes

- › Managed widget = a widget geometry is managed by a layout
- › The layout is based on widget and layout properties
 - › `virtual QSize QWidget::sizeHint()` – recommended size for the widget in the layout
 - › `virtual QSize QWidget::minimumSizeHint()` – recommended minimum size for the widget in the layout
 - › Stretch factor, strut etc.
- › Instead of re-implementing the virtual functions, it is possible to call
 - › `QWidget::setFixedSize()` – minimum and maximum sizes set to the same value
 - › `QWidget::setMinimumSize()`
 - › `QWidget::setMaximumSize()` – The default maximum is `QWIDGETSIZE_MAX` ($2^{24} - 1$)

Layout Management Classes

- › **QHBoxLayout**

- › Lines up widgets horizontally

- › **QVBoxLayout**

- › Lines up widgets vertically

- › **QGridLayout**

- › Arranges the widgets in a grid

- › **QFormLayout**

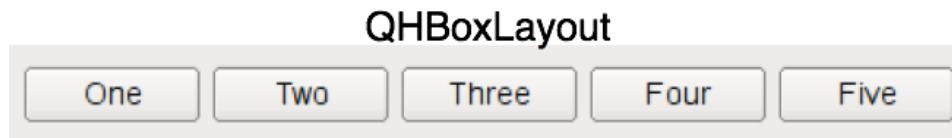
- › Lines up a (label, widget) pairs in two columns.

- › **QStackedLayout**

- › Arranges widgets in a stack
 - › Only top most is visible

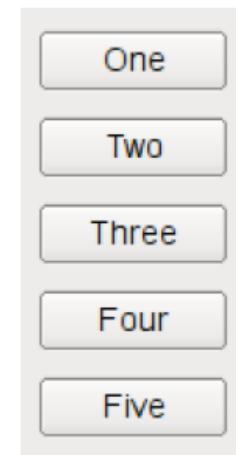
QHBoxLayout and QVBoxLayout

- › Lines up widgets horizontally or vertically
- › Divides space into boxes
- › Each managed widgets fills in one box



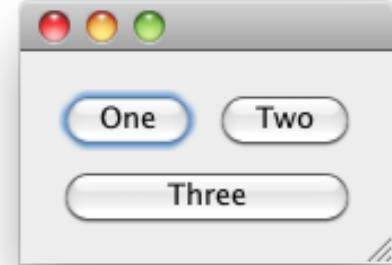
```
QWidget* window = new QWidget;
QPushButton* one = new QPushButton("One");
...
QHBoxLayout* layout = new QHBoxLayout;
layout->addWidget(one);
...
window->setLayout(layout);
```

VBoxLayout



Widgets in a Grid - QGridLayout

```
QWidget* window= new QWidget;
QPushButton* one = new QPushButton("One");
QGridLayout* layout = new QGridLayout;
layout->addWidget(one, 0, 0); // row:0, col:0
layout->addWidget(two, 0, 1); // row:0, col:1
// row:1, col:0, rowspan:1, colspan:2
layout->addWidget(three, 1, 0, 1, 2);
window->setLayout(layout)
```



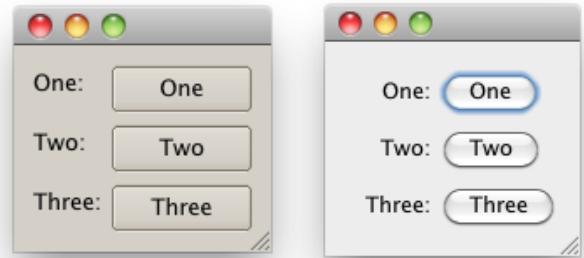
- › Additional
 - › setColumnMinimumWidth () (*minimum width of column*)
 - › setRowMinimumHeight () (*minimum height of row*)
- › *No need to specify rows and columns before adding children*

QFormLayout

- › A two-column layout
 - › Column 1 a label (as annotation)
 - › Column 2 a widget (as field)
- › Respects style guide of individual platforms.

```
QWidget* window = new QWidget();  
QPushButton* one = new QPushButton("One"); ...  
QFormLayout* layout = new QFormLayout();  
layout->addRow("One", one);  
...  
window->setLayout(layout)
```

- › Form layout with clean looks and mac style



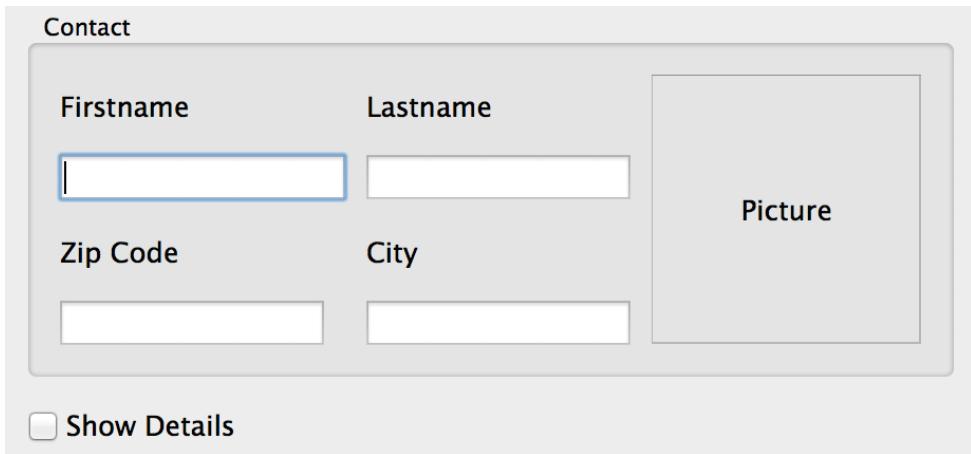
Lab – Contact Form

- › Specified by graphic designer
- › Your task: implement it
 - › Focus on correct layout
 - › Details disabled by default
 - › 'ShowDetails' enables details
- › Optional:
 - › Click on Picture
 - › Lets user choose image
 - › See lab description
- › Validate Zip-Code as integers

Contact

| | | |
|----------------------|----------------------|----------------------|
| Firstname | Lastname | Picture |
| <input type="text"/> | <input type="text"/> | |
| Zip Code | City | <input type="text"/> |
| <input type="text"/> | <input type="text"/> | |

Show Details



Some Layout Terms

› Stretch

- › *Relative size factor*
- › `QBoxLayout::addWidget(widget, stretch)`
- › `QBoxLayout::addStretch(stretch)`
- › `QGridLayout::setRowStretch(row, stretch)`
- › `QGridLayout::setColumnStretch(col, stretch)`

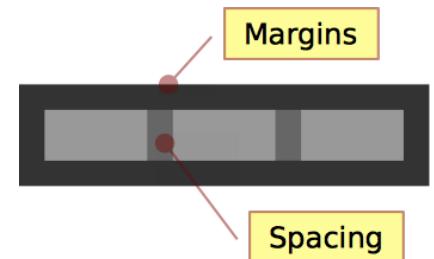


› Contents Margins

- › *Space reserved around the managed widgets.*
- › `QLayout::setContentsMargins(l,t,r,b)`

› Spacing

- › *Space reserved between widgets*
- › `QBoxLayout::addSpacing(size)`



More Layout Terms

> Strut

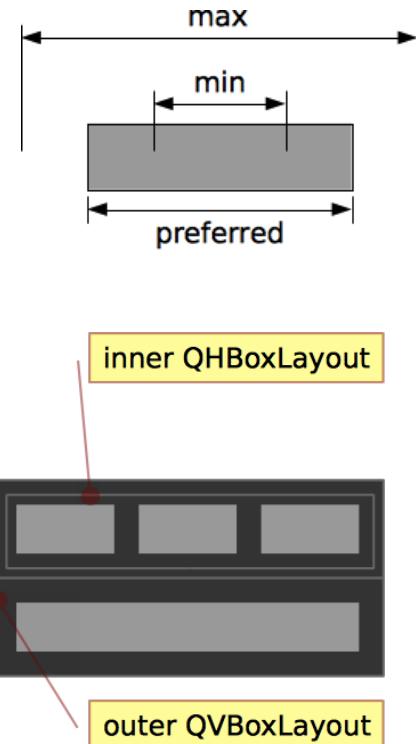
- > *Limits perpendicular box dimension*
- > e.g. height for QBoxLayout
- > *Only for box layouts*

> Min, max and fixed sizes

- > QWidget::setMinimumSize(QSize)
- > QWidget::setMaximumSize(QSize)
- > QWidget::setFixedSize(QSize)
- > *Individual width and height constraints also available*

> Nested Layouts

- > *Allows flexible layouts*
- > QLayout::addLayout(...)



Widgets Size Policies

- › `QSizePolicy` describes interest of widget in resizing

```
QSizePolicy policy = widget->sizePolicy();
policy.setHorizontalPolicy(QSizePolicy::Fixed);
widget->setSizePolicy(policy);
```

- › One policy per direction (horizontal and vertical)
- › Button-like widgets set size policy to the following:
 - › May stretch horizontally
 - › Are fixed vertically
 - › Similar to `QLineEdit`, `QProgressBar`, ...
- › Widgets which provide scroll bars (e.g. `QTextEdit`)
 - › Can use additional space
 - › Work with less than `sizeHint()`
- › `sizeHint()`: recommended size for widget

Available Size Policies

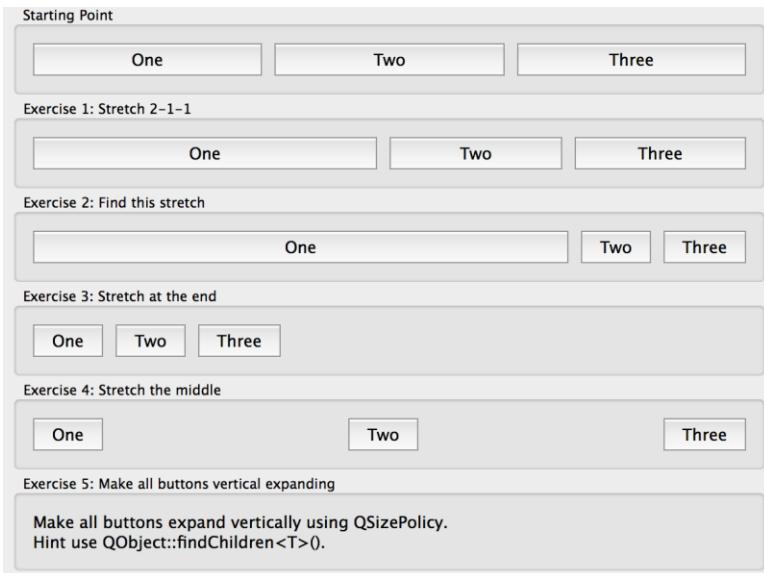
| Policy | sizeHint() | Widget |
|-------------------|---------------------|--|
| Fixed | authoritative | cannot grow or shrink |
| Minimum | minimal, sufficient | can expand, no advantage of being larger |
| Maximum | is maximum | can shrink |
| Preferred | is best | can shrink, no advantage of being larger |
| Minimum Expanding | is minimum | can use extra space |
| Expanding | sensible size | can grow and shrink |

Questions And Answers

- › How do you change the minimum size of a widget
- › Name the available layout managers.
- › How do you specify stretch?
When are you allowed to call `resize` and `move` on a widget?

Lab – Layout of buttons

- › Develop the following layouts
- › Adjust the layouts as shown below.
- › Optionally:
 - › Make buttons resize vertically when making the window higher.



Guidelines: Creating a Custom Widget

- › It's as easy as deriving from QWidget

```
class CustomWidget : public QWidget
{
public:
    explicit CustomWidget(QWidget* parent=0,
                          Qt::WindowFlags flags = Qt::WindowFlags());
}
```

- › If you need custom Signal Slots
 - › Add `Q_OBJECT`
- › Use layouts to arrange widgets inside, or paint the widget yourself

Window Flags

- › Window-system properties for a window
- › `Qt::FramelessWindowHint` – Produces a borderless window
- › `Qt::WindowStaysOnTopHint` – Window stays on top
- › `Qt::WindowStaysOnBottomHint`
- › `Qt::WindowTransparentForInput` – Window is used only for output
- › `Qt::WindowDoesNotAcceptFocus`

Guidelines: Base Class and Event Handlers

- › **Do not reinvent the wheel**
- › **Decide on a base class**
 - › Often QWidget or QFrame
- › **Overload needed event handlers**
 - › Often:
 - › QWidget::mousePressEvent(),
 - › QWidget::mouseReleaseEvent()
 - › If widget accepts keyboard input
 - › QWidget::keyPressEvent()
 - › If widget changes appearance on focus
 - › QWidget::focusInEvent(),
 - › QWidget::focusOutEvent()

Guidelines: Drawing a Widget

- › **Decide on composite or draw approach**
 - › *If composite*: Use layouts to arrange other widgets
 - › *If draw*: implement paint event
- › Re-implement `QWidget::paintEvent()` for drawing
 - › To draw widget's visual appearance
 - › Drawing often depends on internal states
- › **Decide which signals to emit**
 - › Usually from within event handlers
 - › Especially `mousePressEvent()` or `mouseDoubleClickEvent()`
- › **Decide carefully on types of signal parameters**
 - › General types increase reusability
 - › Candidates are `bool`, `int` and `const QString&`

Guidelines: Internal States and Sub-classing

- > **Decide on publishing internal states**
 - > Which internal states should be made publically accessible?
 - > Implement accessor methods
- > **Decide which setter methods should be slots**
 - > Candidates are methods with integral or common parameters
- > **Decide on allowing sub-classing**
 - > If yes
 - > Decide which methods to make protected instead of private
 - > Which methods to make virtual

Guidelines: Widget Constructor

› Decide on parameters at construction time

- › Enrich the constructor as necessary
- › Or implement more than one constructor
- › If a parameter is needed for widget to work correctly
 - › User should be forced to pass it in the constructor

› Keep the Qt convention with:

```
explicit Constructor(..., QWidget *parent = Q_NULLPTR,  
                     Qt::WindowFlags flags)
```

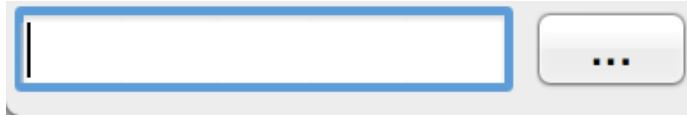
Summary

- › Widgets provide one option to implement user interfaces in Qt programs
- › Qt has a large selection of ready-made widgets
- › Ready-made widgets may be aggregated together to have more complicated custom widgets
- › Layout management takes care of positioning and resizing child widgets
 - › Qt has layout widgets and layout manager classes
 - › Horizontal and vertical size policies define, how child widgets size changes in the layout
- › Custom widgets can be done by re-implementing the `paintEvent()` method as well

Lab – *File Chooser*

- › Create a reusable file chooser component
- › 2 Modes
 - › Choose File
 - › Choose Directory
- › *Think about the Custom Widget Guidelines!*
- › *Create a reusable API for a FileChooser?*

- › *After lab discuss your API*



Lab – *Compass Widget*

- › Implement a “compass widget” and let user...

- › Select a direction
 - › north, west, south, east
 - › And optionally none

- › Provide API to...

- › Change direction programmatically
- › Get informed when direction changes

- › Optional

- › Add direction None
- › Select direction with the keyboard



Contents

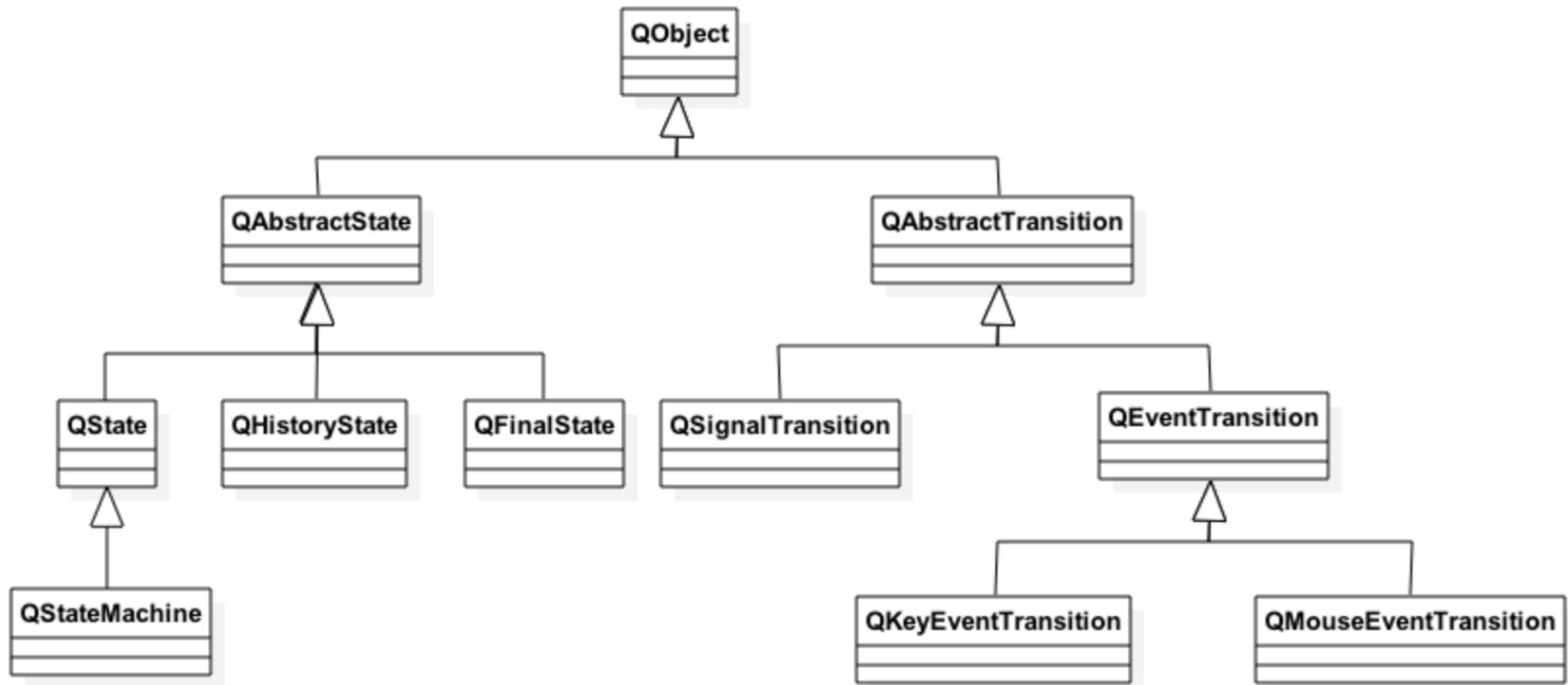
- › Animation Framework
- › Animation Groups
- › State Machines
- › State-Based User Interfaces

Objectives

Learn...

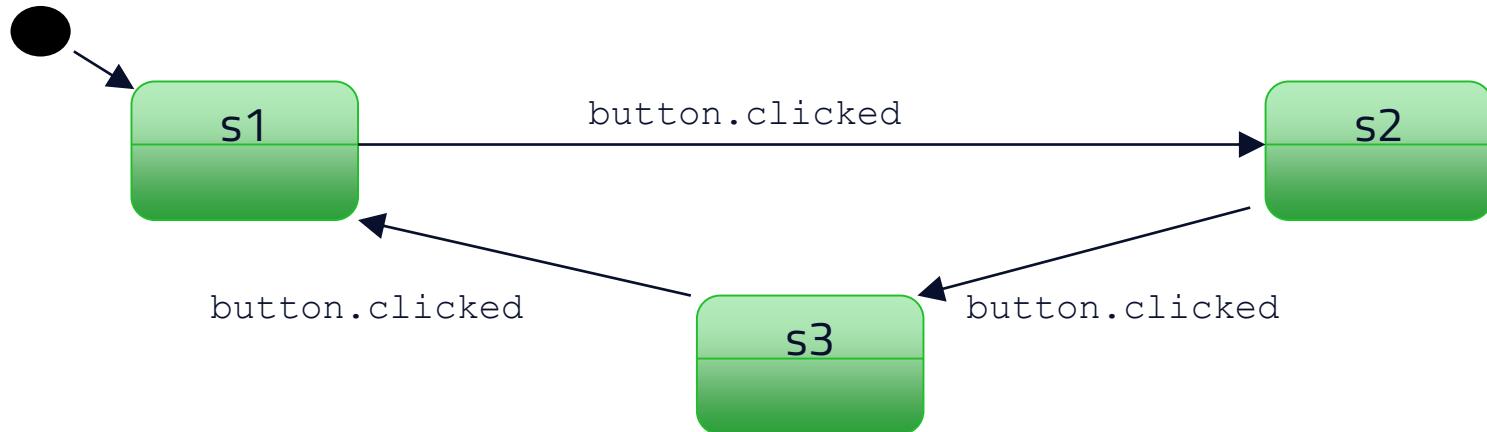
- › ...how to implement state-based user interfaces
- › ...how to animate widgets
- › ...how to use states with animations

State Machine Framework



First State Machine

- › Conceptually our state machine could be presented as shown below
 - › State 1 is the initial state
 - › In the next few slides we will fill in the source code and run the application as a simple demo
- › Useful base class signals:
 - › `QAbstractState::entered()` and
 - › `QAbstractState::exited()`



First State Machine

- › The initial application could be written as follows

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QWidget window;
    window.setFixedSize(200, 200);
    QPushButton* button = new QPushButton("Switch to state 2", &window);
    window.show();
    return a.exec();
}
```



First State Machine

```
QStateMachine* machine = new QStateMachine(&window);

QState *s1 = new QState();
QState *s2 = new QState();
QState *s3 = new QState();

s1->addTransition(button, &QPushButton::clicked, s2);
s2->addTransition(button, &QPushButton::clicked, s3);
s3->addTransition(button, &QPushButton::clicked, s1);

machine->addState(s1);
machine->addState(s2);
machine->addState(s3);

machine->setInitialState(s1);
machine->start();
```

First State Machine

- › So far our state machine does not perform any meaningful work
- › Typically such work means changing the properties of various QObjects in the application
 - › For example, the button's text:

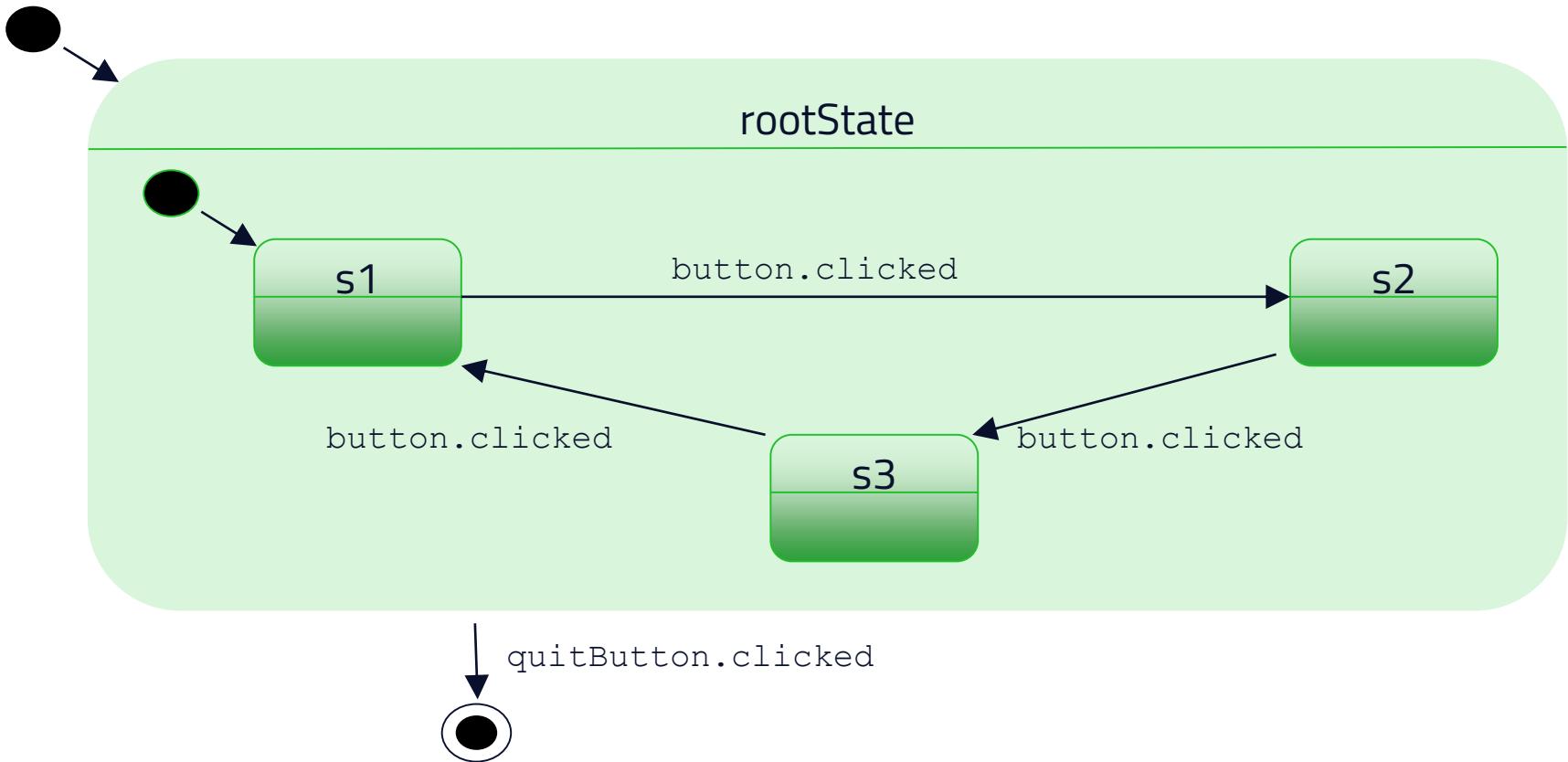
```
s1->assignProperty(button, "text", "Switch to state 2");
s2->assignProperty(button, "text", "Switch to state 3");
s3->assignProperty(button, "text", "Switch to state 1");
```

- › How would you make the button move to a different position each time the state changes?
- › How about changing the window title to display the current state?

Grouping States

- › Consider a situation where a common transition needs to be made regardless of the current state
 - › E.g. switch to a final state when user clicks a quit button
- › The first solution: add a transition to the final state from each of the existing states (1-3 in our example)
 - › Not very scalable – need to remember to add the same transition to each new state as well
- › The proper solution: group the states as children of a single top-level state
 - › The parent state gets the transition to the final state
 - › The transition is automatically inherited by the child states – also by new ones added later!
 - › A child state can also override an inherited transition

Grouping States



Grouping States

```
QState *rootState = new QState();
QState *s1 = new QState(rootState); // Notice that rootState is the parent state
QState *s2 = new QState(rootState);
QState *s3 = new QState(rootState);
QFinalState *finalState = new QFinalState(); // Final state is top-level

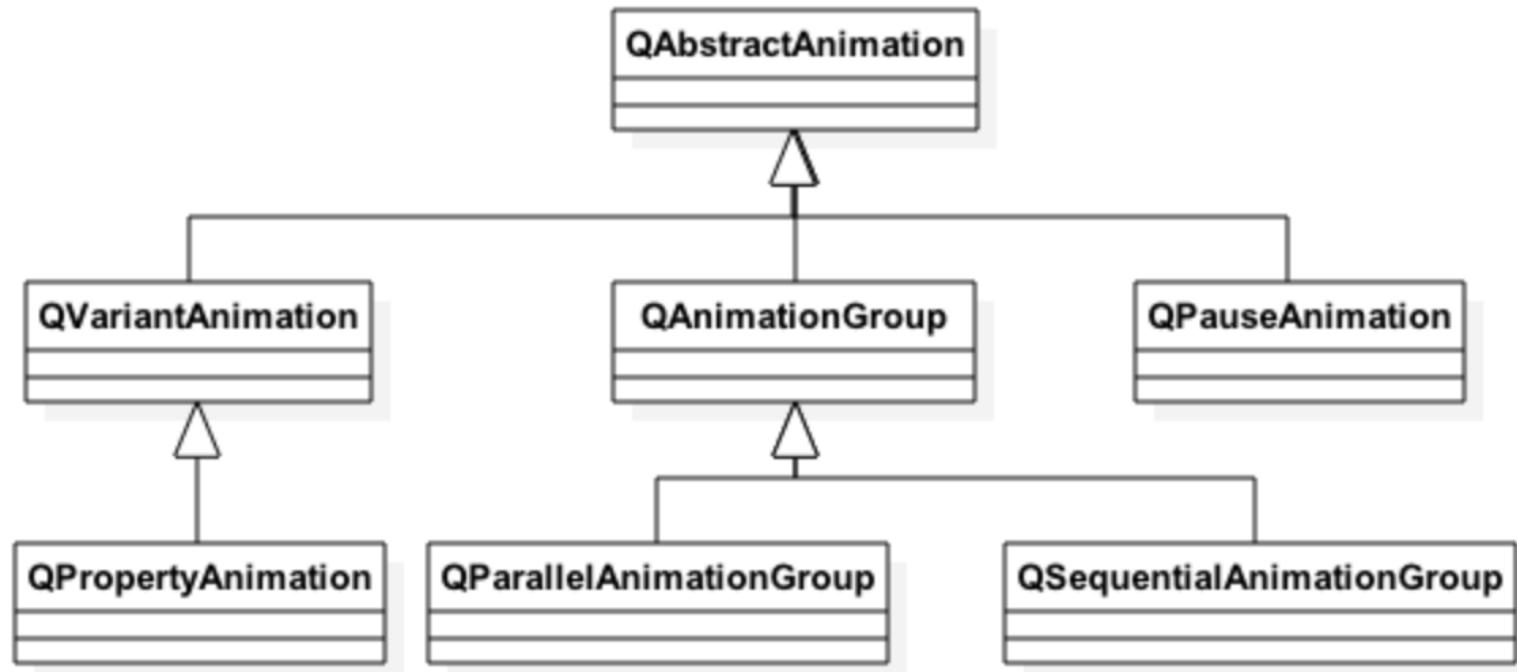
// Root state is entered as the machine starts...
machine->setInitialState(rootState);
// ...and state 1 is entered immediately after that
rootState->setInitialState(s1);
// Only top-level states are added to the state machine
machine->addState(rootState);
machine->addState(finalState);
// New transition, other transitions stay as they were
rootState->addTransition(quitButton, &QPushButton::clicked, finalState);
QObject::connect(machine, &QStateMachine::finished, qApp, &QApplication::quit);

machine->start();
```

Animation Framework

- › The idea is to provide an easy and scalable way of creating animated and smooth GUIs
 - › Hides the complexity of handling timers
 - › Provides a set of pre-defined easing curves, custom curves can be provided by the developer as needed
- › *Works by modifying the **properties** of the **QObjects** being animated*
- › Easy to use together with the State Machine Framework to provide animated transitions between states
- › Normally the animations have a finite duration
 - › QTimeLine used as a timer behind the scenes
 - › Can be made to loop and run backwards
- › If duration is set to -1, the animation will run until explicitly stopped

Animation Classes



Example – Animating a Button

```
QPushButton button("Animated Button");
button.show();

// This is a main() function. That's why we can allocate the animation in the stack
QPropertyAnimation animation(&button, "geometry");

animation.setDuration(10000); animation.setStartValue(QRect(0, 0, 100, 30));
animation.setKeyValueAt(0.5, QRect(250, 250, 100, 30));
animation.setEndValue(QRect(0, 0, 100, 30));
animation.start();
```

Animation Groups

- › Classes `QSequentialAnimationGroup` and `QParallelAnimationGroup` can be used to run multiple animations in sequence or in parallel
 - › An animation group can be used wherever a "normal" animation could – e.g. a group within a group
 - › Enables building very complex animation sequences

```
QPropertyAnimation* animation1 = new QPropertyAnimation(...);
QPropertyAnimation* animation2 = new QPropertyAnimation(...);
QPropertyAnimation* animation3 = new QPropertyAnimation(...);
QPropertyAnimation* animation4 = new QPropertyAnimation(...);
QParallelAnimationGroup *parallel = new QParallelAnimationGroup();

parallel->addAnimation(animation2);
parallel->addAnimation(animation3);

QSequentialAnimationGroup* seq = new QSequentialAnimationGroup();
seq->addAnimation(animation1);
seq->addAnimation(parallel);
seq->addAnimation(animation4);
seq->start();
```

Easing Curves

- › The Animation FW provides multiple pre-made easing curves for your convenience
 - › Represented by the class `QEasingCurve`
 - › Custom curves can made by the developer; see the class documentation for details
- › Typically used to control e.g. how the animation starts and/or finishes
 - › `QEasingCurve::OutBounce`
 - › `QEasingCurve::InOutQuad`
 - › ...

```
QPropertyAnimation animation(&button, "geometry");
animation.setEasingCurve(QEasingCurve::OutBounce);
```

Animations with a State Machine

- › As mentioned before, the Animation FW can easily be made to work with the State Machine FW
- › Simply a matter of assigning one or more animations for each wanted state transition
 - › Works the same way for both `QSignalTransitions` and `QEventTransitions`
- › In this case a property animation itself does not need a start or an end value
 - › These come from the states in the state machine
 - › Intermediate values can be set as before using the base class `QVariantAnimation::setKeyValueAt()` function

Animations with a State Machine

```
QStateMachine *machine = new QStateMachine();
QState *s1 = new QState();
machine->addState(s1);
QState *s2 = new QState();
machine->addState(s2);

s1->assignProperty(button, "geometry", QRect(0, 0, 100, 30));
s2->assignProperty(button, "geometry", QRect(250, 250, 100, 30));

QSignalTransition *t1 = s1->addTransition(button, SIGNAL(clicked()), s2);
t1->addAnimation(new QPropertyAnimation(button, "geometry"));

QSignalTransition *t2 = s2->addTransition(button, SIGNAL(clicked()), s1);
t2->addAnimation(new QPropertyAnimation(button, "geometry"));

machine->setInitialState(s1);
machine->start();
```

Default Animations

- › If a specific animation is wanted regardless of the state transition, a default animation can be set
 - › E.g. always animate "pos" property if its value is different in the start and target states
- › Animations explicitly set on transitions will take precedence over any default animation for the given property

```
// Created earlier: QState* s1, s2; QStateMachine* machine;  
s2->assignProperty(object, "fooBar", 2.0);  
s1->addTransition(s2);  
machine->setInitialState(s1);  
machine->addDefaultAnimation(new QPropertyAnimation(object, "fooBar"));
```

Questions and Answers

- › How to trigger a transition in a Qt state machine?
- › What means state grouping?
- › Why state grouping can be beneficial?
- › How to group animations?
- › Why is it better to use a sequential animation rather than two animations, executed one after the other?

Summary

- › Qt provides state machine and animation frameworks
- › State machines support all basic state machine concepts
 - › States (initial, final, history), transitions, nested states
- › Animations can be implemented using a timer
 - › Qt animation framework hides the timer management
- › Animations and state machines may be combined
 - › Each state has different property values
 - › Whenever a transition is triggered, changed property values are animated

Contents

- › Painting on Widgets
- › Color Handling
- › Painting Operations
- › Styles and Style Sheets

Objectives

- › Painting

- › You paint with a painter on a paint device during a paint event
- › Qt widgets know how to paint themselves
- › Often widgets look like we want
- › Painting allows device independent 2D visualization
- › Allows to draw pie charts, line charts and many more

- › Style Sheets

- › Fine grained control over the look and feel
- › Easily applied using style sheets in CSS format

Objectives

Covers techniques for general 2D graphics and styling applications.

> **Painting**

- > Painting infrastructure
- > Painting on widget

> **Color Handling**

- > Define and use colors
- > Pens, Brushes, Palettes

> **Shapes**

- > Drawing shapes

> **Transformation**

- > 2D transformations of a coordinate system

> **Style Sheets**

- > How to make small customizations
- > How to apply a theme to a widget or application



Low-level Painting with QPainter

- › Supports painting on any paint device
 - › QWidget, QPagedPaintDevice, QPixmap, QImage, QPicture, QOpenGLWidget, QOpenGLPaintDevice, QPaintDeviceWindow
- › Provides drawing functions
 - › Lines, shapes, text or pixmaps
- › Controls
 - › Rendering quality
 - › Clipping
 - › Composition modes

Paint Engines and Paint Devices

- › Paint engine converts `QPainter` primitives to device-specific functions and rasterizes pixels using graphics context functions or graphics pipeline
 - › X11, Windows, OpenGL, Picture, SVG, Raster, Direct2D, Direct3D, Pdf, OpenVG, User, MaxUser
- › `QBackingStore` provides a drawing area for raster painting
 - › Automatically created for `QWidget`, manually created for `QWindow`
 - › Drawing area is typically `QImage`, often mapped to video memory
 - › OpenGL painting is done using `QOpenGLContext`

Painting on Widgets

- › Override `paintEvent (QPaintEvent*)`

```
void CustomWidget::paintEvent (QPaintEvent * ) {  
    QPainter painter (this);  
    painter.drawRect (0, 0, 100, 200); // x, y, w, h  
}
```

- › Schedule painting

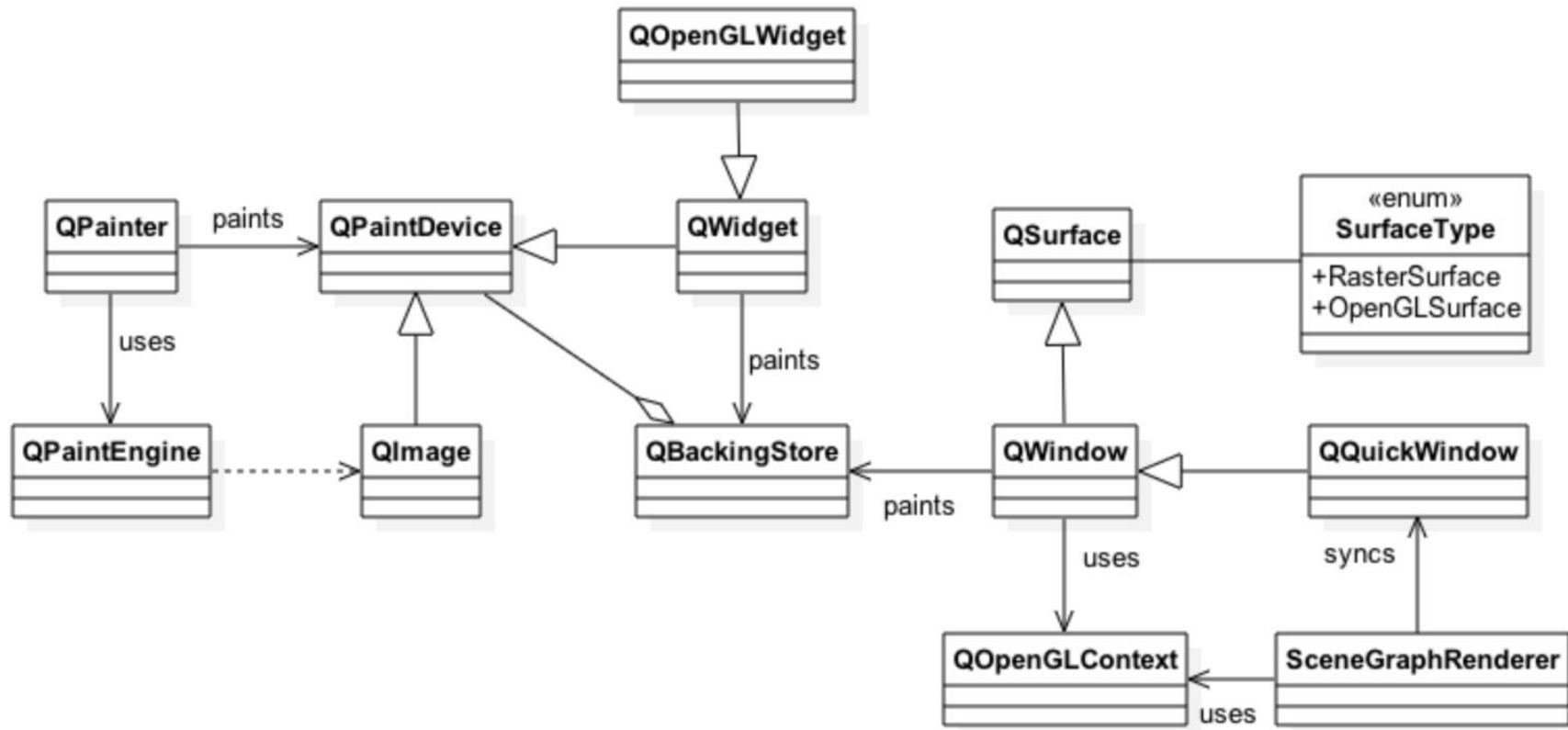
- › `update ()`: schedules paint event
 - › `repaint ()`: repaints directly

- › Qt handles double-buffering

- › To enable filling background:

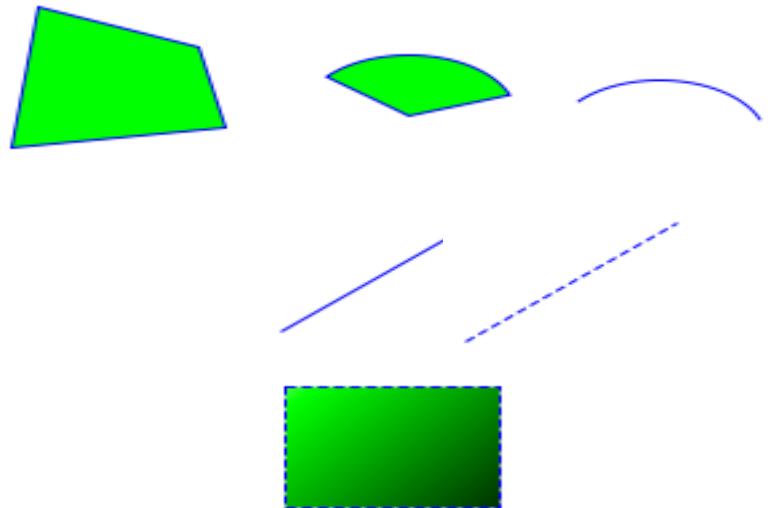
- › `QWidget::setAutoFillBackground (true)`

Painting on Widgets and Windows



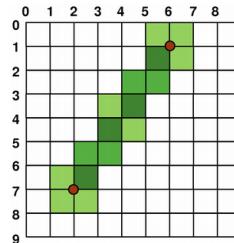
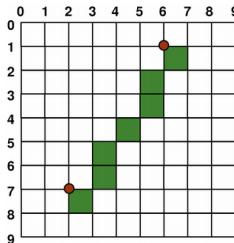
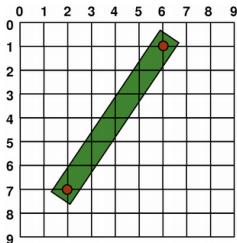
QPainter Paint Toolkit

- › Draw different shapes
 - › Polygon, rectangle, ellipse, pie, line, arc, text
- › Specify pen width and style
 - › Solid, dash (slow), dot, join style
- › Enable/disable antialiasing
 - › Antialiasing is slow
- › Set brush
 - › No brush, gradient color, texture
- › Use transformations
 - › Translate, rotate ($n \times 90$ degrees is fast), scale, shear
- › Save and restore drawing context
 - › QPainter settings



Coordinate System - Surface to Render

- › Controlled by QPainter
- › Origin: Top-Left
 - Rendering
 - › Logical-mathematical
 - › Aliased-right and below
 - › Anti-aliased-smoothing



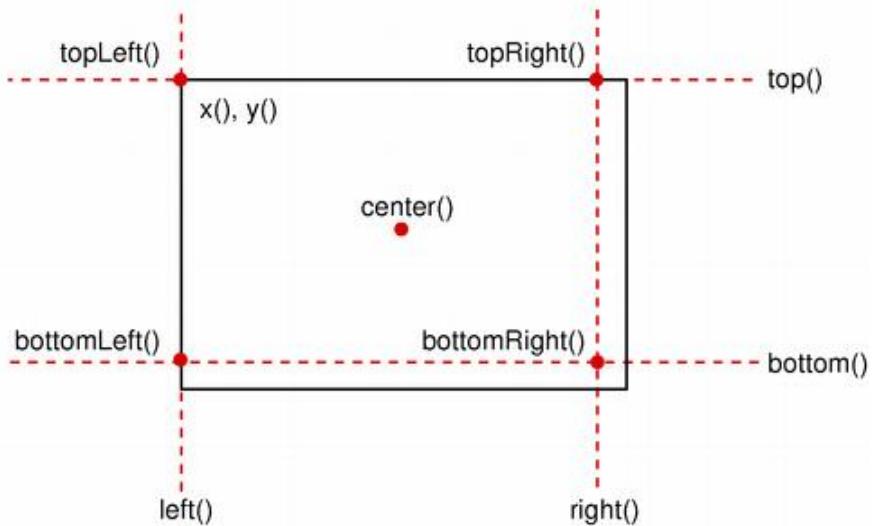
- › Rendering quality switch

- › QPainter::setRenderHint ()

Geometry Helper Classes

- > **QSize**(w, h)
 - > scale, transpose
- > **QPoint**(x, y)
- > **QLine**(point1, point2)
 - > translate, dx, dy
- > **QRect**(point, size)
 - > adjust, move
 - > translate, scale, center

```
QSize size(100,100);
QPoint point(0,0);
QRect rect(point, size);
rect.adjust(10,10,-10,-10);
QPoint center = rect.center();
```



Creating Color Values

- › Using different color models:

- > `QColor(255,0,0) // RGB`
 - > `QColor::fromHsv(h,s,v) // HSV`
 - > `QColor::fromCmyk(c,m,y,k) // CMYK`

- › Defining colors:

- > `QColor(255,0,0); // red in RGB`
 - > `QColor(255,0,0, 63); // red 25% opaque (75% transparent)`
 - > `QColor("#FF0000"); // red in web-notation`
 - > `QColor("red"); // by svg-name`
 - > `Qt::red; // predefined Qt global colors`

- › Many powerful helpers for manipulating colors

- > `QColor("black").lighter(150); // a shade of gray`

- › `QColor` always refers to device color space

Drawing Lines and Outlines with QPen

- › A pen (`QPen`) consists of:
 - › **A color or brush**
 - › **A width**
 - › **A style** (e.g. `NoPen` or `SolidLine`)
 - › **A cap style** (i.e. line endings)
 - › **A join style** (connection of lines)
- › Activate with `QPainter::setPen()`

```
QPainter painter(this);  
QPen pen = painter.pen();  
pen.setBrush(Qt::red);  
pen.setWidth(3);  
painter.setPen(pen);  
// draw a rectangle with 3 pixel width red outline  
painter.drawRect(0,0,100,100);
```

The Outline

› Rule: The outline equals the size plus half the pen width on each side.

› For a pen of width 1:

```
QPen pen(Qt::red, 1); // width = 1
float hpw = pen.widthF()/2; // half-pen width
QRectF rect(x, y, width, height);
QRectF outline = rect.adjusted(-hpw, -hpw, hpw, hpw);
```

› Due to integer rounding on a non-antialiased grid, the outline is shifted by 0.5 pixel towards the bottom right.

Filling Shapes with QBrush

- › QBrush defines fill pattern of shapes
- › Brush configuration

- › `setColor(color)`
- › `setStyle(Qt::BrushStyle)`
 - › NoBrush, SolidPattern,...
- › `QBrush(gradient) // QGradient's`
- › `setTexture(pixmap)`

- › Brush with solid red fill

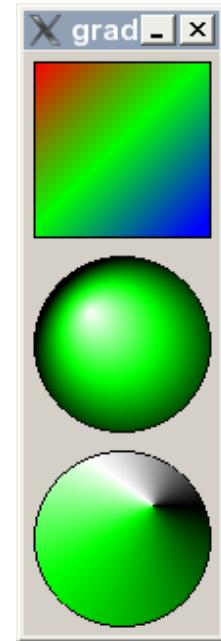
```
painter.setPen(Qt::red);  
painter.setBrush(QBrush(Qt::yellow, Qt::SolidPattern));  
painter.drawRect(rect);
```



Drawing Gradient Fills

- › Gradients used with QBrush
- › Gradient types
 - › QLinearGradient
 - › QConicalGradient
 - › QRadialGradient
- › Gradient from P1(0,0) to P2(100,100)

```
QLinearGradient gradient(0, 0, 100, 100);
// position, color: position from 0..1
gradient.setColorAt(0, Qt::red);
gradient.setColorAt(0.5, Qt::green);
gradient.setColorAt(1, Qt::blue);
painter.setBrush(gradient);
// draws rectangle, filled with brush
painter.drawRect(0, 0, 100, 100 );
```



Brush on QPen

- › Possible to set a brush on a pen
- › Strokes generated will be filled with the brush



Creating Color Values

- › Supported color spaces

- › `QColor(255,0,0) // RGB`
 - › `QColor::fromHsv(h,s,v) // HSV`
 - › `QColor::fromCmyk(c,m,y,k) // CMYK`

- › Defining Colors

- › `QColor(255,0,0); // red in RGB`
 - › `QColor(255,0,0, 127); // red 50% transparent`
 - › `QColor("#FF0000"); // red in web-notation`
 - › `QColor("red"); // by svg-name`
 - › `Qt::red; // predefined Qt global colors`

- › QColor powerful for manipulating colors

- › `QColor("red")::lighter(150); // defines 50% brighter red`

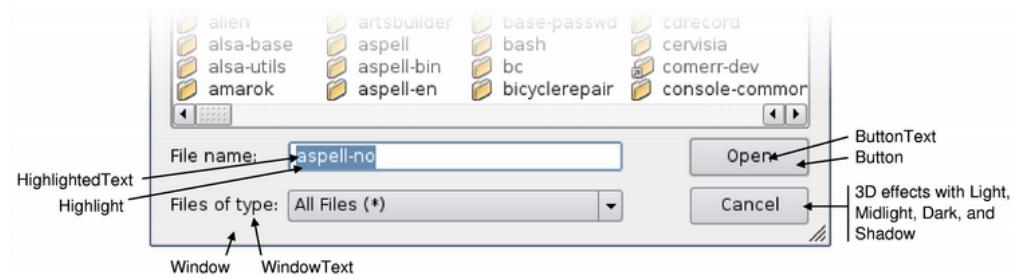
Color Themes and Palettes

- › To support widgets color theming
 - › `setColor (blue)` not recommended
 - › Colors need to be managed
- › QPalette manages colors
- › Consist of color groups
 - › `enum QPalette::ColorGroup`
 - › Resemble widget states
 - › `QPalette::Active`
 - › Used for window with keyboard focus
 - › `QPalette::Inactive`
 - › Used for other windows
 - › `QPalette::Disabled`
 - › Used for disabled widgets

Color Groups and Roles

- > Color group consists of color roles

- > enum QPalette::ColorRole
- > Defines symbolic color roles used in UI

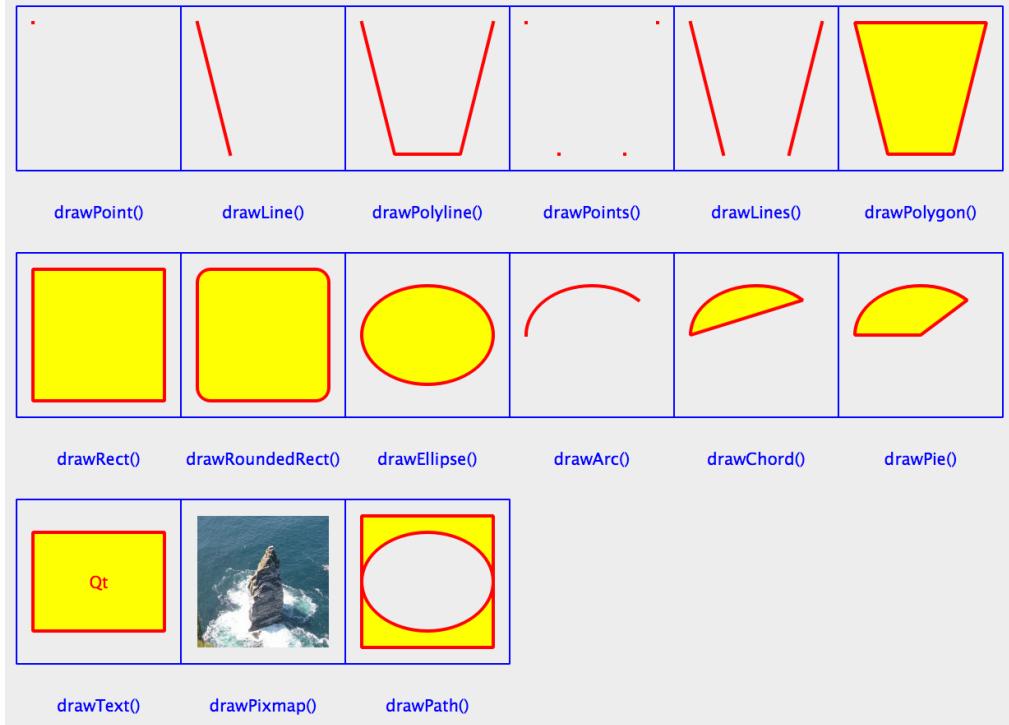


```
QPalette pal = widget->palette();
QColor color(Qt::red);
pal.setColor(QPalette::Active, QPalette::Window, color);
// for all group
pal.setBrush(QPalette::Window, QBrush(Qt::red));
widget->setPalette(pal);
```

- > QApplication::setPalette()
- > Sets application wide default palette

Drawing Figures

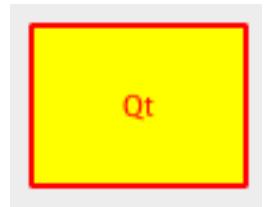
- > Painter configuration
 - > Pen width: 2
 - > Pen color: red
 - > Font size: 10
 - > Brush color: yellow
 - > Brush style: solid



Drawing Text

```
> QPainter::drawText(rect, flags, text)

QPainter painter(this);
painter.drawText(rect, Qt::AlignCenter, tr("Qt"));
painter.drawRect(rect);
painter.drawStaticText(left, top, staticText);
```



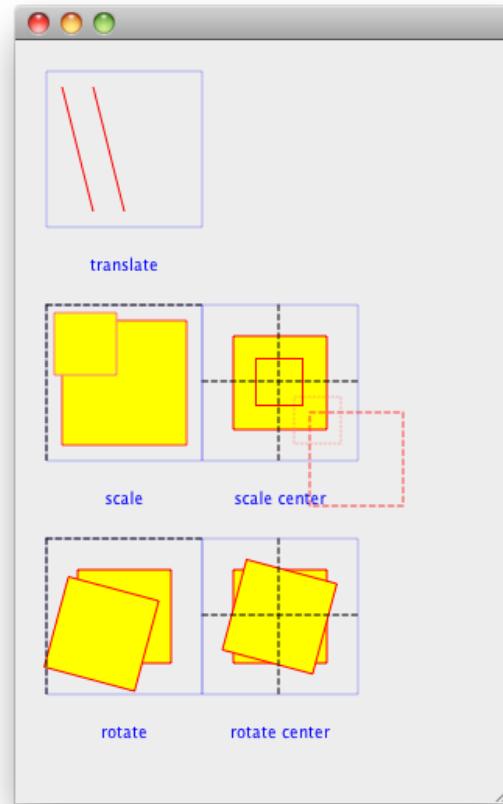
```
> QFontMetrics
```

```
> Calculate size of strings
```

```
QFont font("times", 24);
QFontMetrics fm(font);
int pixelsWide = fm.width("Width of this text?");
int pixelsHeight = fm.height();
```

Transformation

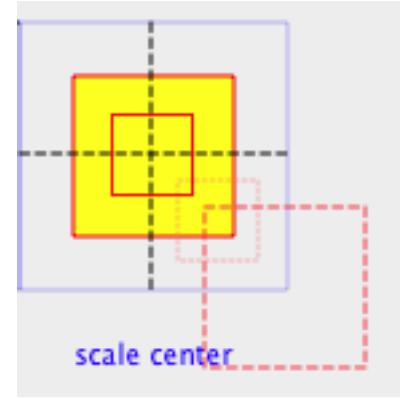
- › Manipulating the coordinate system
 - › `translate(x, y)`
 - › `scale(sx, sy)`
 - › `rotate(a)`
 - › `shear(sh, sv)`
 - › `reset()`



Transform and Center

- > `scale(sx, sy)`
 - > Scales around QPoint(0, 0)
- > Same applies to all transform operations
- > Scale around center?

```
painter.drawRect(r);
painter.translate(r.center());
painter.scale(sx,sy);
painter.translate(-r.center());
// draw center-scaled rect
painter.drawRect(r);
```



Painter Path - QPainterPath

- › Container for painting operations
- › Enables reuse of shapes

```
QPainterPath path;  
path.addRect(20, 20, 60, 60);  
path.moveTo(0, 0);  
path.cubicTo(99, 0, 50, 50, 99, 99);  
path.cubicTo(0, 99, 50, 50, 0, 0);  
painter.drawPath(path);
```

- › Path information
 - › controlPointRect() – rect containing all points
 - › contains() – test if given shape is inside path
 - › intersects() – test given shape intersects path

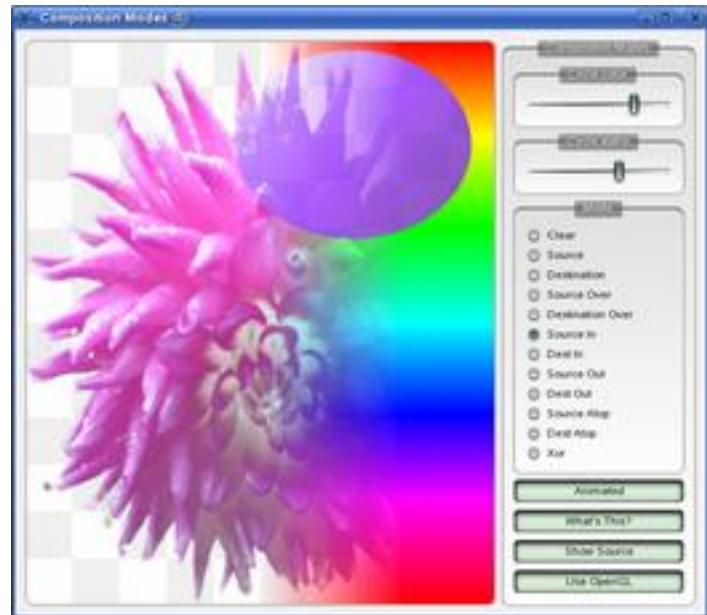


Painting and Widget Attributes

- › Control the widget behavior – in some cases affecting the performance as well
 - › Especially in embedded platforms
- › `Qt::WA_NoSystemBackground` – may be useful in embedded platforms for the top-level widget, which is considered to be opaque
- › `Qt::WA_OpaquePaintEvent` – no need to clear widget's background
- › `Qt::WA_TranslucentBackground` – non-opaque regions of a widget are translucent
 - › Easy way to paint arbitrary shaped widgets
 - › This requires also a window flag `Qt::FramelessWindowHint`
- › `Qt::WA_DeleteOnClose` – deletes a widget after the widget has accepted `closeEvent()`
 - › Easy way to delete dynamically created dialogs

Other Painter Concepts

- › Clipping
 - › Clip drawing operation to shape
- › Composition modes:
 - › Rules for digital image compositing
 - › Combining pixels from source to destination
- › Rubber Bands – QRubberBand
 - › Rectangle or line that indicate selection or boundary



Paint Operations and Performance

- › Slow CPU-intensive operations
 - › Rotations (not multiple of 90 degrees), non-linear gradients with three or more colors, dashed lines, anti-aliasing, text layout calculation, image composition (except source and source over modes), non-rectangular clip with complex transformations, painter path fill calculations
- › CPU-intensive operations typically result that the application uses most of the time in `qtbase/src/gui/painting/qdrawhelper.cpp`
 - › If this is not the case, performance problem is likely somewhere else
- › Good way to improve paint performance is to paint to `QPixmaps` and render the pixmaps
 - › Pixmap blending may still drop the performance to a few frames even in desktop CPUs

QImage vs. QPixmap

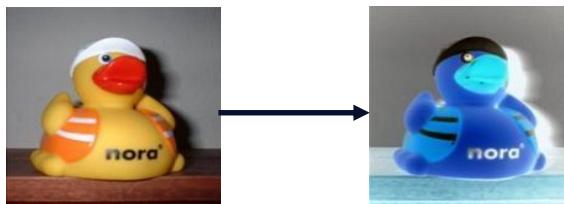
- › Designed and optimized for I/O
 - › Direct pixel access and manipulation
 - › Hardware-independent (unlike QPixmap)
- › Supports multiple image formats
 - › E.g. RGB32, ARGB32, ARGB32_Premultiplied, and so on
- › Can be used for drawing in a worker thread
 - › Draw your picture gallery application's images onto QImage in another thread, and
 - › Send finished images to the GUI thread (signal) to be drawn as QPixmap
- › Designed and optimized for showing images on the screen
 - › In your custom widget you can use QPainter::drawPixmap()
 - › Implementation might also depend on the underlying hardware/graphics subsystem
- › Can be created using static convenience functions (in addition to constructors)
 - › QPixmap::grabWidget()
 - › QPixmap::grabWindow()

Image Transformations

- › QImage provides multiple utility functions for image manipulation
 - › Some of these are available for QPixmap as well, although they operate less efficiently
 - › For image transparency, use QPainter composition modes or the QGraphicsOpacityEffect class
 - › Using composition modes is the most efficient choice

› Inverting pixels

- › void QImage::invertPixels()



› Mirroring

- › QImage QImage::mirrored(bool horizontal = false, bool vertical = true) const

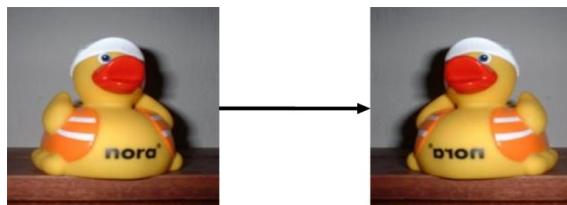
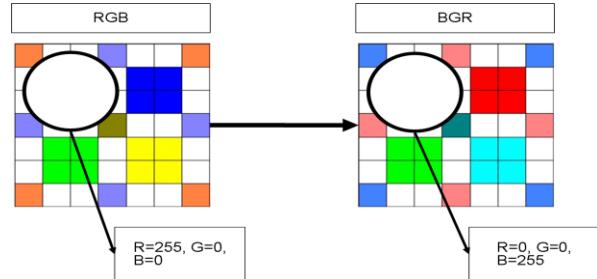


Image Transformations

> Color swapping

- > Red and blue colors are exchanged
- > An RGB image becomes a "BGR" rendering of the original
- > `QImage QImage::rgbSwapped()`



> Masking

- > Image appears to be irregularly shaped (i.e. non-rectangular)
- > `QImage QImage::createAlphaMask()`
- > `QImage QImage::createHeuristicMask()`
- > `QImage QImage::createMaskFromColor()`

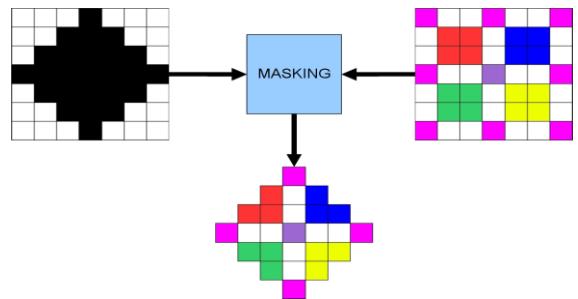


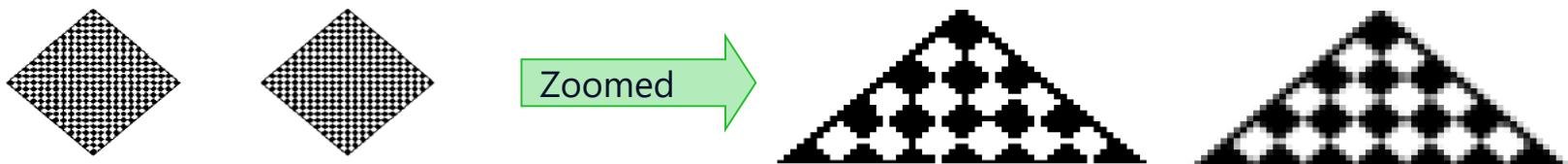
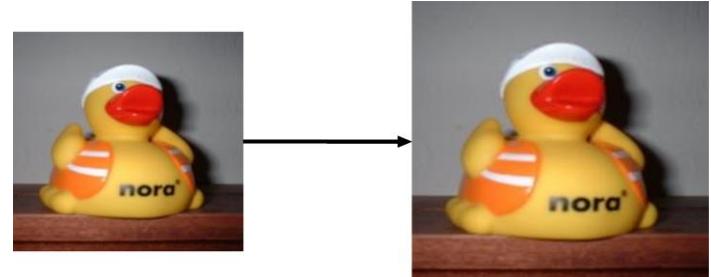
Image Transformations

- › Scaling

- › `QImage QImage::scaled()`
- › `QImage QImage::scaledToWidth()`
- › `QImage QImage::scaledToHeight()`

- › Two transformation modes available for scaling

- › `Qt::FastTransformation` (example on the left)
- › `Qt::SmoothTransformation` (example on the right)
- › Can you tell the difference? How about when zoomed?



OpenGL in Qt

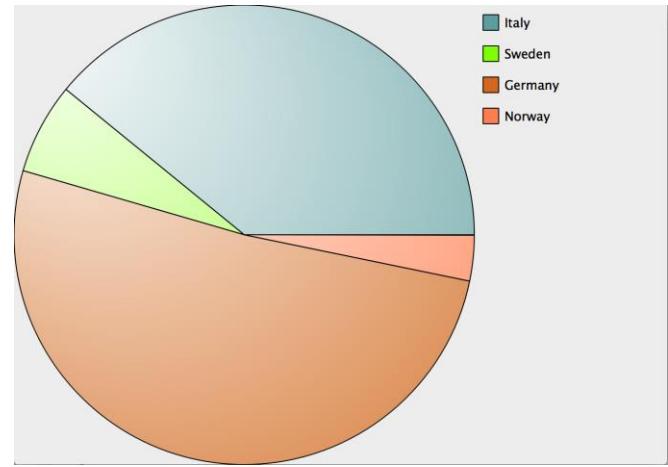
- › In addition to `QPainter`, it is possible to use OpenGL in Qt
- › Use `QOpenGLWidget` or `QOpenGLWindow`
 - › The latter does not have dependencies to widgets
- › Actually, plenty of other ways to use OpenGL
 - › Qt Quick
 - › Qt 3D
 - › Qt Data Visualization

QOpenGLWidget vs. QOpenGLWindow

- › Similar APIs in both classes
 - › `initializeGL()` – initialize your resources
 - › `resizeGL()` – set the viewport or projection, when window size changes
 - › `paintGL()` – render
- › Both render to FBO
- › Both supports partial rendering with `QPainter`
- › Function `update()` will request for a repaint – `paintGL()` function called
- › OpenGL context created automatically and made current

Lab – Pie Chart Widget

- › Task to implement a pie chart
- › Draw pies with painters based on data.
- › Data Example: Population of 4 countries
 - › Sweden
 - › Germany
 - › Norway
 - › Italy
- › Guess the population in millions of citizens ;-)
- › **Legend is optional**
- › See lab description for details



Styling

- › Widget style can be customized by
 - › changing the palette – `QPalette`
 - › changing style options
 - › subclassing platform style (`QStyle` subclass) and re-implementing one or more member functions
 - › using style sheets
- › Style sheets are easy to use and can be changed without re-compilation
 - › They also override `QStyle` settings
- › Using `QStyle` and style options provide better performance

QStyle

- › The widgets call `QStyle` methods for:
 - › drawing
 - › querying metrics (sizes and positions inside the widget)
 - › general look and feel “style hints”
 - › preparing a widget to be styled
- › `QStylePainter` provides a `QStyle` API
 - › Uses shorter argument list as `QStyle`, no need to pass `QPainter` and `QWidget` in every function call
 - › `drawComplexControl (QStyle::ComplexControl, QStyleOptionComplex)`
 - › `drawControl (QStyle::ControlElement, QStyleOption)`
 - › `drawPrimitive (QStyle::PrimitiveElement, QStyleOption)`

Using Styles

- › Styles may be used in custom widgets to provide native look-and-feel

```
void CoolCustomWidget::paintEvent(QPaintEvent *event)
{
    QStyleOptionFocusRect option;
    option.initFrom(this);
    option.backgroundColor = palette().color(QPalette::Background);
    style()->drawPrimitive(QStyle::PE_FrameFocusRect, &option, &p, this);

    // QStylePainter styledPainter(this);
    // styledPainter.drawPrimitive(QStyle::PE_FrameFocusRect, &option);
}
```

- › Or you may customize the style, used by existing widgets
 - › To have a company brand, for example

QStyleOption

- › QStyleOption groups/stores many parameters, used by QStyle functions, into a single struct:
 - › **rect** - the widget rectangle
 - › **palette** - the widget palette
 - › **state** - the widget state (enabled, has focus, mouse over, sunken, raised etc.)
 - › **direction** – layout direction
 - › **fontMetrics**
- › QStyleOption subclasses provide other widget-specific parameters:
 - › QStyleOptionButton - pushbutton features (flat, default etc.), text and icon
 - › QStyleOptionComboBox - whether the combobox is editable, the rectangle of the popup
- › You may sub-class QStyleOption and extend the enumeration after
`QStyleOption::SO_CustomBase`
- › You may obviously re-implement `drawPrimitive()` method and ignore/change all style data there

Writing Your Own QStyle

- › `QStyle` is only the API (virtual methods and helpers), `QCommonStyle` is a useful base class for many styles
- › For a new style, inherit `QCommonStyle` or an existing style:

```
#include <qwindowsstyle.h>

class TestStyle : public QWindowsStyle
{
public:
    virtual int styleHint(StyleHint sh, const QStyleOption *opt=0,
                          const QWidget *w=0, QStyleHintReturn* ret=0) const;
};
```

- › For testing:
 - › `QApplication::instance()->setStyle(new TestStyle);`

styleHint()

› Styles can re-implement `styleHint()` to set general hints such as:

- › `SH_UnderlineShortcut` - whether shortcuts are underlined
- › `SH_Button_FocusPolicy` - the default focus policy for buttons
- › `SH_TabBar_Alignment` - alignment for tabs in a `QTabWidget`
- › `SHLineEdit_PasswordCharacter` - the character to be used for passwords

```
int TestStyle::styleHint(StyleHint sh, const QStyleOption *opt,
    const QWidget *w, QStyleHintReturn *hret) const
{
    switch (sh) {
        case SH_GroupBox_TextLabelVerticalAlignment:
            return Qt::AlignVCenter;
        default:
            return QWindowsStyle::styleHint(sh, opt, w, hret);
    }
}
```

Multiple Levels of Method Calls

- › High-level drawing methods delegate the work to lower-level ones
- › For instance `QCommonStyle::drawControl()`, for a `CE_PushButton`, calls
 - › `drawControl(CE_PushButtonBevel)` for the border (bevel),
 - › `drawControl(CE_PushButtonLabel)` for the text label and icon,
 - › `drawPrimitive(PE_FrameFocusRect)` for the focus rectangle
- › Similarly, `CE_ProgressBar` calls `CE_ProgressBarGroove`, `CE_ProgressBarContents` and `CE_ProgressBarLabel`

drawControl()

- › Draws a “control element” (part of a widget that performs some action or displays information to the user)
- › ControlElement is an enum, for example:
 - › CE_PushButton - push button bevel and label
 - › CE_PushButtonBevel - the border of a push button
 - › CE_CheckBox - checkbox
 - › CE_Splitter - splitter handle
 - › CE_TabBarTab - tab shape and label within a QTabBar
 - › CE_TabBarTabShape - tab shape within a QTabBar
 - › CE_TabBarTabLabel - label within a tab

drawControl()

```
void TestStyle::drawControl(ControlElement ce,
    const QStyleOption *opt, QPainter *p, const QWidget *w) const
{
    switch( ce ) {
    case CE_PushButtonBevel:
        // Need to adjust the rect for the bottom and right lines
        // be visible inside the rectangle, see QPainter documentation
        p->drawRect( opt->rect.adjusted(0,0,-1,-1) );
        break;
    default:
        QWindowsStyle::drawControl( ce, opt, p, w );
    }
}
```



drawPrimitive()

- › Called for drawing a common GUI element, possibly used by several controls
 - › PrimitiveElement is an enum with many PE_* values:
 - › PE_FrameFocusRect - Generic focus indicator
 - › PE_IndicatorArrowUp - Generic Up arrow
 - › PE_IndicatorCheckBox - On/off indicator, e.g. QCheckBox
 - › PE_IndicatorBranch - Branches of a tree in a tree view
 - › PE_IndicatorMenuCheckMark - Check mark used in a menu
 - › PE_FrameLineEdit - Panel frame for line edits

Drawing Functions

- › Drawing utilities defined in qdrawutil.h, for use from style code
 - › `qDrawShadeLine()` - horizontal or vertical shaded line
 - › `qDrawShadeRect()` - shaded rectangle
 - › `qDrawShadePanel()` - shaded panel, either raised or sunken
 - › `qDrawWinButton()` - Windows-style (2 pixels line width) button, raised or sunken
 - › `qDrawWinPanel()` - Windows-style (2 pixels line width) panel, raised or sunken (for line edits, check boxes, etc.)
 - › `qDrawPlainRect()` - plain rectangle, with line width and fill

pixelMetric()

- › A pixel metric is a style-dependent size represented as a single pixel value
- › Styles re-implement the virtual method `int QStyle::pixelMetric(PixelMetric metric, const QStyleOption *option = 0, const QWidget *widget = 0)`
 - › `PM_ButtonMargin` - margin inside push button
 - › `PM_IndicatorWidth/Height` - width/height of check box
 - › `PM_Menu*` - metrics used for menus
 - › `PM_Slider*` - metrics used for sliders
 - › `PM_ScrollBar*` - metrics used for scrollbars
 - › `PM_TabBar*` - metrics used for tab bars

polish()

- › `QStyle::polish(QApplication*)`, called once on startup
 - › Fixing the palette and font for the purpose of the style
 - › Installing global event filters
- › `QStyle::polish(QWidget*)`, called for every widget
 - › `setBackgroundMode()` for pixmap backgrounds
 - › `setAttribute()` (e.g. `WA_Hover` so that the widget is repainted when the mouse enters or leaves the widget)
 - › Installing event filters on a widget (e.g. for setting a mask for non-rectangular widgets when resized)

```
void TestStyle::polish( QWidget* w )
{
    if ( qobject_cast<QPushButton*>( w ) ) {
        w->setAttribute( Qt::WA_Hover );
    }
    QWindowsStyle::polish( w );
}
```

Qt Style Sheets

- › Mechanism to customize appearance of widgets
 - › Additional to subclassing `QStyle`
- › Inspired by HTML CSS
- › Textual specifications of styles
- › Applying Style Sheets
 - › `QApplication::setStyleSheet(sheet)`
 - › On whole application
 - › `QWidget::setStyleSheet(sheet)`
 - › On a specific widget (incl. child widgets)



CSS Rules

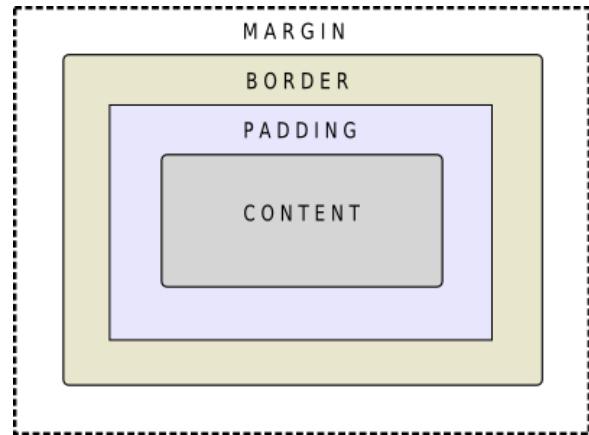
- › selector { property : value; property : value }
- › Selector: specifies the widgets
- › Property/value pairs: specify properties to change.
 - › `QPushButton { color: red; background-color: white }`
- › Examples of stylable elements
 - › Colors, fonts, pen style, alignment.
 - › Background images.
 - › Position and size of sub controls.
 - › Border and padding of the widget itself.

The Box Model

- > Every widget treated as box
- > Four concentric rectangles
 - > Margin, Border, Padding, Content
- > Customizing QPushButton

```
QPushButton {  
    border-width: 2px;  
    border-radius: 10px;  
    padding: 6px;  
    // ...  
}
```

- > By default, margin, border-width, and padding are 0



Margin Rectangle Padding Rectangle
 Border Rectangle Content Rectangle



Selector Types

- > * { } // Universal selector
 - > All widgets
- > QPushButton { } // Type Selector
 - > All instances of class
- > .QPushButton { } // Class Selector
 - > All instances of class, but not subclasses
- > QPushButton#objectName // ID Selector
 - > All Instances of class with objectName
- > QDialog QPushButton { } // Descendant Selector
 - > All instances of QPushButton which are children of QDialog
- > QDialog > QPushButton { } // Direct Child Selector
 - > All instances of QPushButton which are direct children of QDialog
- > QPushButton [enabled="true"] // Property Selector
 - > All instances of class which match property

Selector Details

- › Property Selector

- › If property changes it is required to re-set style sheet

- › Combining Selectors

- › `QLineEdit, QComboBox, QPushButton { color: red }`

- › Pseudo-States

- › Restrict selector based on widget's state
 - › Example: `QPushButton:hover {color:red}`

- › Selecting Subcontrols

- › Access subcontrols of complex widgets as `QComboBox, QSpinBox, ...`
 - › `QComboBox::drop-down { image: url.dropdown.png; }`

- › Subcontrols positioned relative to other elements

- › Change using `subcontrol-origin` and `subcontrol-position`

Conflict Resolution - Cascading

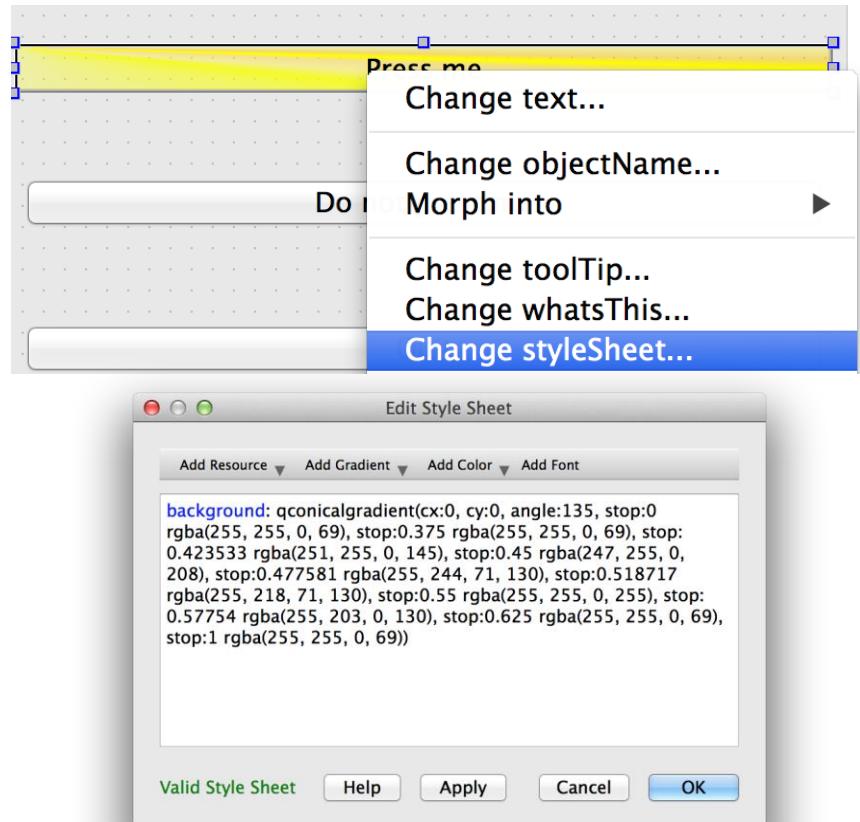
- › Effective style sheet obtained by merging
 - › Widget's ancestor (parent, grandparent, etc.)
 - › Application stylesheet
- › On conflict: widget own style sheet preferred
 - › `qApp->setStyleSheet("QPushButton { color: white }");`
 - › `button->setStyleSheet("* { color: blue }");`
- › Style on button forces button to have blue text
 - › In spite of more specific application rule

Conflict Resolution - Selector Specificity

- › Conflict: When rules on same level specify same property
 - › Specificity of selectors apply
 - › `QPushButton:hover { color: white }`
 - › `QPushButton { color: red }`
 - › Selectors with pseudo-states are more specific
- › Calculating selector's specificity
 - › **a** Count number of ID attributes in selector
 - › **b** Count number of property specifications
 - › **c** Count number of class names
 - › Concatenate numbers **a-b-c**. Highest score wins
 - › If rules scores equal, use last declared rule
 - › `QPushButton {} /* a=0 b=0 c=1 -> specificity = 1 */`
 - › `QPushButton#ok {} /* a=1 b=0 c=1 -> specificity = 101 */`
- Demo: painting/ex-qssconflict

Qt Designer Integration

- › Excellent tool to preview style sheets
- › Right-click on any widget
- › Select *Change styleSheet..*
- › Includes syntax highlighter and validator



Questions and Answers

- › Describe the Qt paint path at the high level.
- › What is the role of the paint device?
- › What kind of paint device classes does Qt have?
- › What is `QWindow`? How is it related to `QWidget`?
- › What kind of paint function `QPainter` has?
- › You want to use GPU to accelerate `QPainter` drawing? Is it possible? If it is, how to accelerate the painting?
- › How widgets can be styled?
- › How widgets are styled with style sheets?

Summary

- › `QPainter` provides a 2D paint toolkit supporting
 - › Pen and brush settings
 - › Transformations
 - › Gradient colors
 - › Image composition
- › `QPainter` can be used to paint to any paint device (widget, image, opengl frame buffer object)
- › Paint devices use different paint engines
 - › Paint engine converts all or some `QPainter` paint primitives to device-specific primitives
 - › `QPainter` => OpenGL

Lab – Project Task

- › Tasks (in QtCreator Welcome mode, look for 'Style Sheet Example')
 - › Investigate style sheet
 - › Modify style sheet
 - › Remove style sheet
 - › and implement your own
- › Example does not save changes
 - › Use designer for this.
- › Edit style sheet using
 - › File -> Edit StyleSheet



Contents

- › Main Windows
- › Settings
- › Resources
- › Translation for Developers
- › Deploying Qt Applications

Objectives

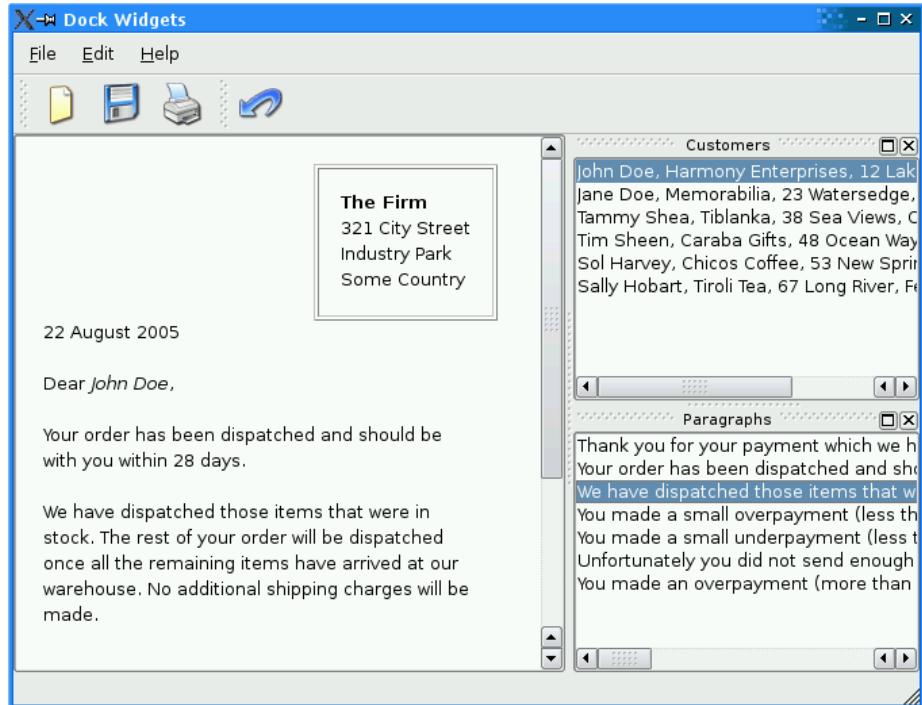
We will create an application to show fundamental concepts

- › **Main Window:** How a typical main window is structured
- › **Settings:** Store/Restore application settings
- › **Resources:** Adding icons and other files to your application
- › **Translation:** Short overview of internationalization
- › **Deployment:** Distributing your application

Typical Application Ingredients

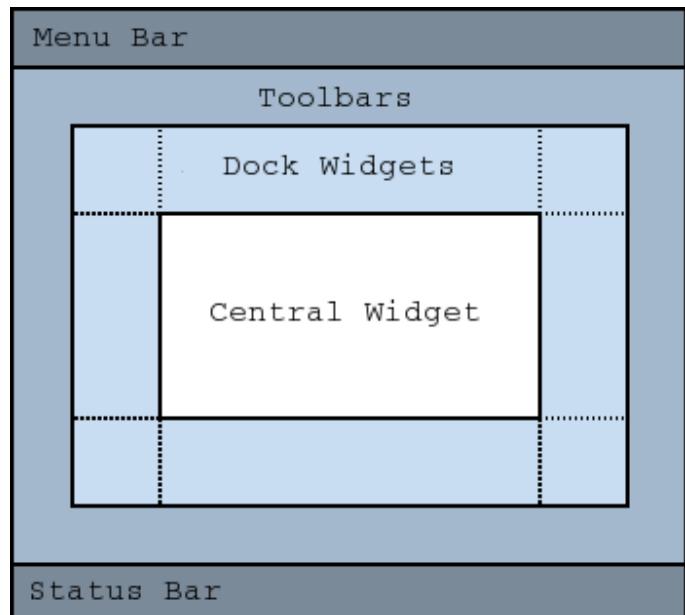
- > Main window with
 - > Menu bar
 - > Tool bar, Status bar
 - > Central widget
 - > Often a dock window
- > Settings (saving state)
- > Resources (e.g icons)
- > Translation
- > Load/Save documents

Not a complete list



Main Window

- › QMainWindow: main application window
- › Has own layout
 - › Central Widget
 - › QMenuBar
 - › QToolBar
 - › QDockWidget
 - › QStatusBar
- › Central Widget
 - › **QMainWindow::setCentralWidget(widget)**
 - › Just any widget object



Creating Actions - QAction

- › *Action is an abstract user interface command*
- › Emits signal triggered on execution
 - › Connected slot performs action
- › Added to menus, toolbar, key shortcuts
- › Each performs same way
 - › Regardless of user interface used

```
void MainWindow::setupActions() {
    QAction* action = new QAction(tr("Open ..."), this);
    action->setIcon(QIcon(":/images/open.png"));
    action->setShortcut(QKeySequence::Open);
    action->setStatusTip(tr("Open file"));
    connect(action, SIGNAL(triggered()), this, SLOT(onOpen()));

    menu->addAction(action);
    toolbar->addAction(action);
```

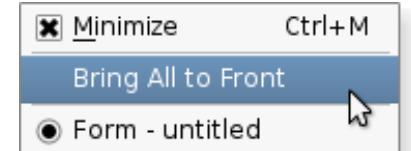
QAction Capabilities

- › `setEnabled(bool)`
 - › Enables/disables actions
 - › In menus and toolbars, etc...
- › `setCheckable(bool)`
 - › Switches checkable state (on/off)
 - › `setChecked(bool)` toggles checked state
- › `setData(QVariant)`
 - › Stores data with the action

Create Menu Bar

- > **QMenuBar**: a horizontal menu bar
- > **QMenu**: represents a menu
 - > Indicates action state
- > **QAction**: menu items added to **QMenu**

File Edit Options Help



```
void MainWindow::setupMenuBar() {  
    QMenuBar* bar = menuBar();  
    QMenu* menu = bar->addMenu(tr("&File"));  
    menu->addAction(action);  
    menu->addSeparator();  
  
    QMenu* subMenu = menu->addMenu(tr("Sub Menu"));  
    ...  
}
```

Creating Toolbars - QToolBar

- › Movable panel...
 - › Contains set of controls
 - › Can be horizontal or vertical
- › `QMainWindow::addToolbar(toolbar)`
 - › Adds toolbar to main window
- › `QMainWindow::addToolBarBreak()`
 - › Adds section splitter
- › `QToolBar::addAction(action)`
 - › Adds action to toolbar
- › `QToolBar::addWidget(widget)`
 - › Adds widget to toolbar



```
void MainWindow::setupToolBar() {
    QToolBar* bar = addToolBar(tr("File"));
    bar->addAction(action);
    bar->addSeparator();
    bar->addWidget(new QLineEdit(tr("Find ...")));
}
```

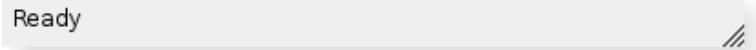
QToolButton

- › Quick-access button to commands or options
- › Used when adding action to QToolBar
- › Can be used instead QPushButton
 - › Different visual appearance!
- › Advantage: allows to attach action

```
QToolButton* button = new QToolButton(this);
button->setDefaultAction(action);
// Can have a menu
button->setMenu(menu);
// Shows menu indicator on button
button->setPopupMode(QToolButton::MenuButtonPopup);
// Control over text + icon placements
button->setToolButtonStyle(Qt::ToolButtonTextUnderIcon);
```

The Status Bar - QStatusBar

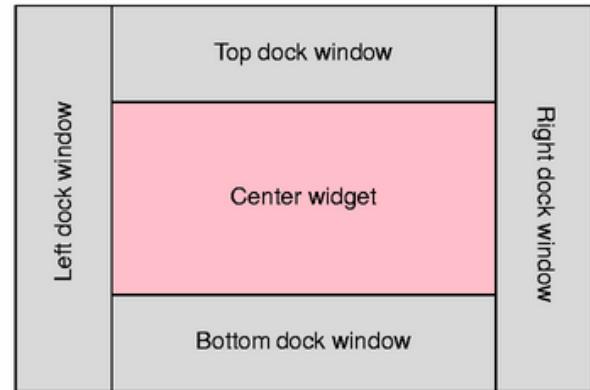
- › Horizontal bar
 - › Suitable for presenting status information
- › showMessage(message, timeout)
 - › Displays temporary message for specified milli-seconds
- › clearMessage ()
 - › Removes any temporary message
- › addWidget () or addPermanentWidget ()
 - › Normal, permanent messages displayed by widget



```
void MainWindow::createStatusBar() {
    QStatusBar* bar = statusBar();
    bar->showMessage(tr("Ready"));
    bar->addWidget(new QLabel(tr("Label on StatusBar")));
}
```

Creating Dock Windows - QDockWidget

- > Window docked into main window
- > Qt::DockWidgetArea enum
 - > Left, Right, Top, Bottom dock areas
- > QMainWindow::setCorner (corner, area)
 - > Sets area to occupy specified corner
- > QMainWindow::setDockOptions (options)
 - > Specifies docking behavior (animated, nested, tabbed,...)



```
void MainWindow::createDockWidget() {
    QDockWidget *dock = new QDockWidget(tr("Title"), this);
    dock->setAllowedAreas(Qt::LeftDockWidgetArea);
    QListWidget *widget = new QListWidget(dock);
    dock->setWidget(widget);
    addDockWidget(Qt::LeftDockWidgetArea, dock);
```

QMenu and Context Menus

- › Launch via event handler

```
void MyWidget::contextMenuEvent(QContextMenuEvent *event) {  
    m_contextMenu->exec(event->globalPos());  
}
```



- › Or signal `customContextMenuRequested()`

- › Connect to signal to show context menu

- › Or via `QWidget::actions()` list

- › `QWidget::addAction(action)`
 - › `setContextMenuPolicy(Qt::ActionsContextMenu)`
 - › Displays `QWidget::actions()` as context menu

Typical APIs

- > QWidget
 - > setWindowModified(...)
 - > setWindowTitle(...)
 - > addAction(...)
 - > contextMenuEvent(...)
- > QMainWindow
 - > setCentralWidget(...)
 - > menuBar()
 - > statusBar()
 - > addToolBar(...)
 - > addToolBarBreak()
 - > addDockWidget(...)
 - > setCorner(...)
 - > setDockOptions(...)
- > QAction
 - > setShortcuts(...)
 - > setStatusTip(...)
 - > signaltriggered()
- > QMenuBar
 - > addMenu(...)
- > QToolBar
 - > addAction(...)
- > QStatusBar
 - > showMessage(...)
 - > clearMessage()
 - > addWidget(...)

Lab – *Text Editor*

- › Create a text editor with
 - › *load, save, quit*
 - › *About* and *AboutQt*
- › A `QPlainTextEdit` serves for editing the text.
- › Optional:
 - › Show whether the file is dirty
 - › Ask the user whether to save if file is dirty when application quits
 - › Make sure also to ask when window is closed via window manager
 - › Show the cursor position in the status bar
 - › Position is determined by cursors block and column count

Persistent Settings - QSettings

› Configure QSettings

```
QCoreApplication::setOrganizationName("MyCompany");  
QCoreApplication::setOrganizationDomain("mycompany.com");  
QCoreApplication::setApplicationName("My Application");
```

› Typical usage

```
QSettings settings;  
settings.setValue("group/value", 68);  
int value = settings.value("group/value").toInt();
```

Settings Values

- › Values are stored as QVariant
- › Keys form hierarchies using ' / '
 - › Or use beginGroup(prefix) / endGroup()
- › value() excepts default value
 - › settings.value("group/value", 68).toInt()
- › If value not found and default not specified
 - › Invalid QVariant() returned

Restoring State of an Application

- › Store geometry of application

```
void MainWindow::writeSettings() {  
    QSettings settings;  
    settings.setValue("MainWindow/size", size());  
    settings.setValue("MainWindow/pos", pos());  
}
```

- › Restore geometry of application

```
void MainWindow::readSettings() {  
    QSettings settings;  
    settings.beginGroup("MainWindow");  
    resize(settings.value("size", QSize(400, 400)).toSize());  
    move(settings.value("pos", QPoint(200, 200)).toPoint());  
    settings.endGroup();  
}
```

Settings - Behind the Scenes

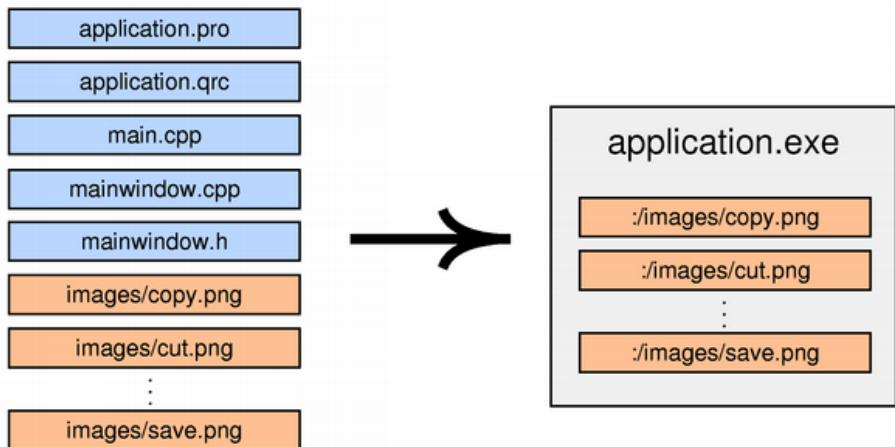
- › `QSettings` is re-entrant
- › Same file can be accessed from any number of threads (processes)
 - › Each thread has its own instance
 - › If `QSettings` in one thread set new data values, `QSettings` objects in other threads see immediately new values
 - › Uses `Q_GLOBAL_STATIC` macro to create static objects (e.g. `QConfFile` cache) in a thread-safe way
- › Settings may be accessed from multiple processes as well
 - › When `sync()` is called, new values written by other processes are read and new values committed to the file
 - › There is no inter-process communication between processes
 - › Called automatically in `QSettings` destructor and by the event loop in regular intervals

Settings - Behind the Scenes

- › Stored in platform specific format
 - › Unix :INI files
 - › Windows: System registry
 - › MacOS: CFPrefences API
- › *Value lookup will search several locations*
 - › User-specific location
 - › for application
 - › for applications by organization
 - › System-wide location
 - › for application
 - › for applications by organization
- › QSettings creation is cheap! Use on stack

Resource System

- › Platform-independent mechanism for storing binary files
 - › Not limited to images
- › Resource files stored in application's executable
- › Useful if application requires files
 - › E.g. icons, translation files, sounds
 - › Don't risk of losing files, easier deployment



Using Resources

- › Resources specified in .qrc file

```
<!DOCTYPE RCC><RCC version="1.0">
<qresource>
    <file>images/copy.png</file>
    <file>images/cut.png</file>
    ...
</qresource>
</RCC>
```

- › Resources are accessible with ':' prefix
 - › Example: ":/images/cut.png"
 - › Simply use resource path instead of file name
 - › QIcon(":/images/cut.png")
- › To compile resource, edit .pro file
 - › RESOURCES += application.qrc
 - › qmake produces make rules to generate binary file

Resource Specifics

- › Path Prefix
 - › <qresource prefix="/myresources">
 - › File accessible via ":/myresources/..."

- › Aliases
 - › <file alias="cut.png">images/scissors.png</file>
 - › File accessible via ":/cut.png"

- › Static Libraries and Resources
 - › Need to force initialization
 - › Q_INIT_RESOURCE (basename) ;

- › Loading resources at runtime
 - › Use rcc to create binary and register resource
 - › rcc -binary data.qrc -o data.rcc
 - › QResource::registerResource("data.rcc")

- › Traverse resource tree using QDir(":/")

Lab – *Upgrade Editor to Use Resources*

- › Use your previous editor, to use Qt resource system for icons
- › Tip: You can use QtCreator to create QRC files

Internationalization (i18n)

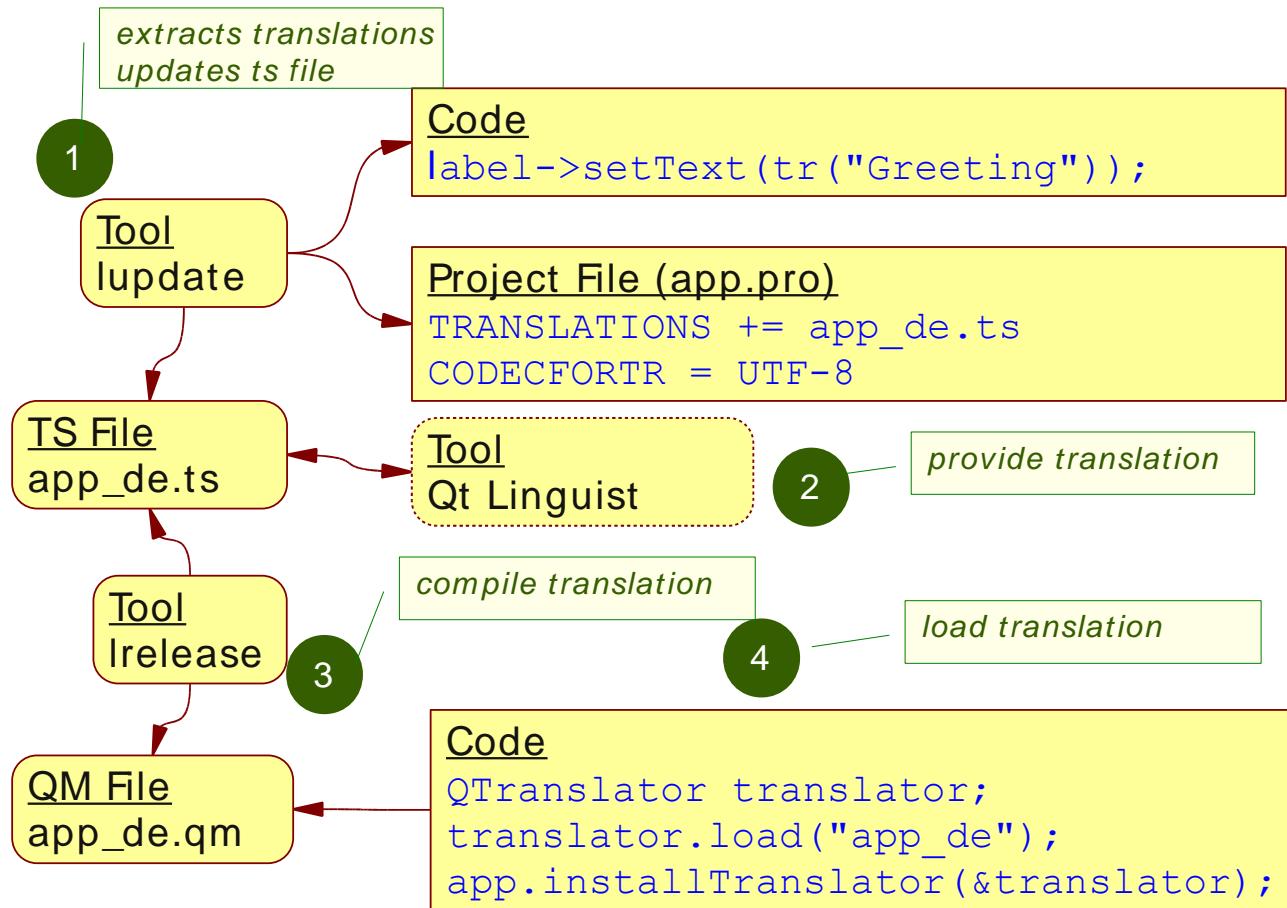
- > **This is by no means a complete guide!**
- > Internationalization(i18n)
 - > Designing applications to be adaptable to languages and regions without engineering changes.
- > Localization(l10n)
 - > Adapting applications for region by adding components and translations
- > Qt supports the whole process:
 - > QString supports unicode
 - > On-screen texts (`QObject::tr()`)
 - > Number and date formats (`QLocale`)
 - > Icons loading (Resource System)
 - > Translation tool (Qt Linguist)
 - > LTR and RTL text, layout and widgets (e.g. Arabic)
 - > Plural handling (1 file vs. 2 files)

Text Translation

- › **lupdate** - scan C++ and .ui files for strings. Create/update .ts file
- › **linguist** - edit .ts file for adding translations
- › **lrelease** - read .ts and creates .qm file for release
- › **QObject::tr(const char *srcTxt, const char *disambiguation = 0, int n = -1)**
 - › Mark translatable strings in C++ code
 - › combine with QString::arg() for dynamic text

```
setWindowTitle(tr("File: %1 Line: %2").arg(f).arg(l));  
  
//: This comment is seen by translation staff  
label->setText(tr("Name: %1 Date: %2").arg(name, d.toString()));
```

Translation Process



Local-Aware Applications

- › `QLocale (Locale Qt.locale())` makes your applications locale-aware
 - › Supports converting numbers to strings with different locale formatting
 - › Possible to get the system locale and set the default locale
 - › `var fin = Qt.locale("fi_FI");`
- › Numbers
 - › Use `%L1` to format a parameter according to the current locale
 - › `tr("%L1").arg(someNumber);`
- › Dates and Times
 - › `tr("Today is %1").arg(Date().toLocaleString(Qt.locale()));`
- › Currencies
 - › `var hugeAmountOfMoney = 12.97;`
 - › `var moneyString = hugeAmountOfMoney.toLocaleCurrencyString(fin, Locale.currencySymbol(Locale.CurrencySymbol)); // CurrencyIsoCode, CurrencyDisplayName`

Lab – *Translate Editor to Your Locale Language*

- › We use our existing editor
- › In the handout you will find a list of translation words
- › Use two character language code, e.g. de for German
- › Tip: You can use Qt Linguist to edit translations

Ways of Deploying

- › Static Linking
 - › Results in stand-alone executable
 - › + Only few files to deploy
 - › – Executables are large
 - › – No flexibility
 - › – You cannot deploy plugins
- › Shared Libraries
 - › + Can deploy plugins
 - › + Qt libs shared between applications
 - › + Smaller, more flexible executables
 - › – More files to deploy
- › Qt is by default compiled as shared library
- › If Qt is pre-installed on system
 - › Use shared libraries approach

Deployment

- › Static Linkage Version
- › Build Qt statically
 - › `$QTDIR/configure -static <your other options>`
 - › Specify required options (e.g. sqldrivers)
- › Link application against Qt
- › Check that application runs stand-alone
 - › Copy application to machine without Qt and run it
- › Shared Library Version
 - › If Qt is not a system library
 - › Need to redistribute Qt libs with application
 - › Minimal deployment
 - › Libraries used by application
 - › Plugins used by Qt
 - › Ensure Qt libraries use correct path to find Qt plugins

Deployment

- › The target platform has Qt libraries
 - › Use `INSTALLS` variable in the `.pro` file to install any files
 - › Use
`QCoreApplication::addLibraryPath() / setLibraryPaths()` to add search path for plugins
 - › `target.files = someFile *.qml qml.dir`
 - › `installDestination = $$[QT_INSTALL_QML]/MyModule/SubName`
 - › `target.path = $$installDestination`
 - › `INSTALLS += target`
- › How to detect Qt version installed on the target?
 - › `qmake -v`
 - › Qt include folder
 - › Platform dependent tools
 - › Linux: `ldd`
 - › OSX: `otool`
 - › Windows: Dependency Walker

Deployment – The Target Does not Have Qt Libs

- › Create a static build
 - › configure -static -platform
- › Create a bundle manually
 - › Copy the relevant Qt libs/plugins to your bundle
 - › Write a script which sets relevant environment variables and launches your application

```
#!/bin/sh
export LD_LIBRARY_PATH=`pwd`/qt_libs
export QML2_IMPORT_PATH=`pwd`/qt_libs/qml
export QT_QPA_PLATFORM_PLUGIN_PATH=`pwd`/qt_libs/plugins/platforms
./MyCoolApplication
```

Deployment – The Target Does not Have Qt Libs

- › Use platform-dependent (OSX, Windows) deployment tools in QTDIR/bin
 - › macdeployqt, windeployqt
 - › Some options
 - › `-no-plugins` (by default all release plugins will be added, if the corresponding Qt module used)
 - › `-dmg` create a disk image in OSX
 - › Third party libraries must be still manually copied to the bundle/added to the installation package
 - › You may need to handle different architectures 32/64 bit, Intel/PowerPC etc.
- › Create a custom binary installer
 1. Create a *package directory structure*
 2. Create a *configuration file*
 3. Create a *package information file*
 4. Create installer content and *copy* it to the package directory
 5. Use the `binarycreator` tool to create the *installer*. The installer pages are created by using the information you provide in the configuration and package information file

Questions and Answers

- › What are the benefits of using QSettings?
- › Would you store any application data into the settings?
- › What can be stored into the settings
- › What are the differences between built-in and external resources?
- › Can resources be localized? If yes, how localized resources are loaded?
- › How mutual exclusion is handled in settings?
- › How strings are localized?
- › What support Qt has to develop locale-aware applications?
- › What tools exist for deploying Qt programs?

Summary

- › Persistent application settings may be stored and loaded with `QSettings`
 - › Provides synchronization between threads
- › Resources may be deployed externally or internally using resource files
 - › Resource file is an XML file, defining the content of resources
 - › Actually resources are files or `const char array`, depending whether they are external or internal
- › Qt applications (and widgets) support
 - › String localization
 - › Resource localization
 - › RTL and LTR text
 - › Number, date, and currency formats based on the locale

Contents

- › Dialogs
- › Common Dialogs
- › Qt Designer

Objectives

› **Custom Dialogs**

- › Modality
- › Inheriting QDialog
- › Dialog buttons

› **Predefined Dialogs**

- › File, color, input and font dialogs
- › Message boxes
- › Progress dialogs
- › Wizard dialogs

› **Qt Designer**

- › Design UI Forms
- › Using forms in your code
- › Dynamic form loading

Dialog Windows - QDialog

- › Base class of dialog window widgets
- › General Dialogs can have 2 modes
- › Modal dialog
 - › Remains in foreground, until closed
 - › Blocks input to remaining application
 - › Example: Configuration dialog
- › Modeless dialog
 - › Operates independently in application
 - › Example: Find/Search dialog

```
MyDialog dialog(this);
dialog.setMyInput(text);
if(dialog.exec() == Dialog::Accepted) {
    // exec blocks until user closes dialog
```

Modeless Dialog

- › Use `show()`
 - › Displays dialog
 - › Returns control to caller

```
void EditorWindow::find() {  
    if (!m_findDialog) {  
        m_findDialog = new FindDialog(this);  
        connect(m_findDialog, SIGNAL(findNext()), this,  
                SLOT(onFindNext()));  
    }  
    m_findDialog->show(); // returns immediately  
    m_findDialog->raise(); // on top of other windows  
    m_findDialog->activateWindow(); // keyboard focus  
}
```

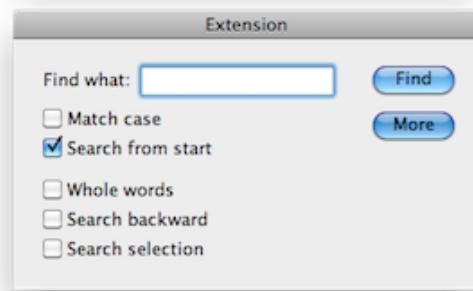
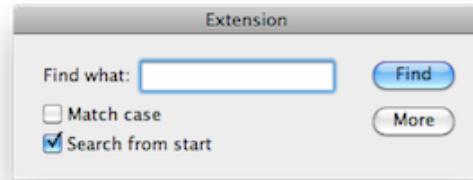
Custom Dialogs

- › Inherit from `QDialog`
- › Create and layout widgets
- › Use `QDialogButtonBox` for dialog buttons
 - › Connect buttons to `accept()`/`reject()`
- › Override `accept()`/`reject()`

```
MyDialog::MyDialog(QWidget *parent) : QDialog(parent) {  
    m_label = new QLabel(tr("Input Text"), this);  
    m_edit = new QLineEdit(this);  
    m_box = new QDialogButtonBox( QDialogButtonBox::Ok |  
        QDialogButtonBox::Cancel, this);  
    connect(m_box, SIGNAL(accepted()), this, SLOT(accept()));  
    connect(m_box, SIGNAL(rejected()), this, SLOT(reject()));  
    ... // layout widgets  
}  
void MyDialog::accept() { // customize close behaviour  
    if(isDataValid()) { QDialog::accept() }  
}
```

Other Issues

- > Deletion of dialogs
 - > No need to keep dialogs around forever
 - > Call `QObject::deleteLater()`
 - > Or `setAttribute(Qt::WA_DeleteOnClose)`
 - > Or override `closeEvent()`
- > Dialogs with extensions:
 - > `QWidget::show() / hide()` used on extension



```
m_more = new QPushButton(tr("&More"));
m_more->setCheckable(true);
m_extension = new QWidget(this);
// add your widgets to extension
m_extension->hide();
connect(m_more, SIGNAL(toggled(bool)), m_extension,
        SLOT(setVisible(bool)));
```

Asking for Files - QFileDialog

- › Allow users to select files or directories

- › Asking for a file name

```
QString fileName = QFileDialog::getOpenFileName(this, tr("Open File"));

if (!fileName.isNull()) {
    // do something useful
}
```

- › QFileDialog::getOpenFileNames()

- › Returns one or more selected existing files

- › QFileDialog::getSaveFileName()

- › Returns a file name. File does not have to exist.

- › QFileDialog::getExistingDirectory()

- › Returns an existing directory.

- › setFilter("Image Files (*.png *.jpg *.bmp)")

- › Displays files matching the patterns

Showing Messages - QMessageBox

- › Provides a modal dialog for...
 - › Informing the user
 - › Asking a question and receiving an answer
- › Typical usage, questioning a user

```
QMessageBox::StandardButton ret = QMessageBox::question(parent, title, text);  
if(ret == QMessageBox::Ok) {  
    // do something useful  
}
```

- › Very flexible in appearance
- › Other convenience methods

- › QMessageBox::information(...)
- › QMessageBox::warning(...)
- › QMessageBox::critical(...)
- › QMessageBox::about(...)

Feedback on Progress - QProgressDialog

- › Provides feedback on the progress of a slow operation

```
QProgressDialog dialog("Copy", "Abort", 0, count, this);
dialog.setWindowModality(Qt::WindowModal);
for (int i = 0; i < count; i++) {
    dialog.setValue(i);
    if (dialog.wasCanceled()) { break; }
    //... copy one file
}
dialog.setValue(count); // ensure set to maximum
```

- › Initialize with `setValue(0)`
 - › Otherwise estimation of duration will not work
- › When operation progresses, check for cancel
 - › `QProgressDialog::wasCanceled()`
 - › Or connect to `QProgressDialog::canceled()`
- › To stay reactive call `QApplication::processEvents()`

Providing Error Messages - QErrorMessage

- › Similar to QMessageBox with checkbox

- › Asks if message shall be displayed again

```
m_error = new QErrorMessage(this);  
m_error->showMessage(message, type);
```

- › Messages will be queued

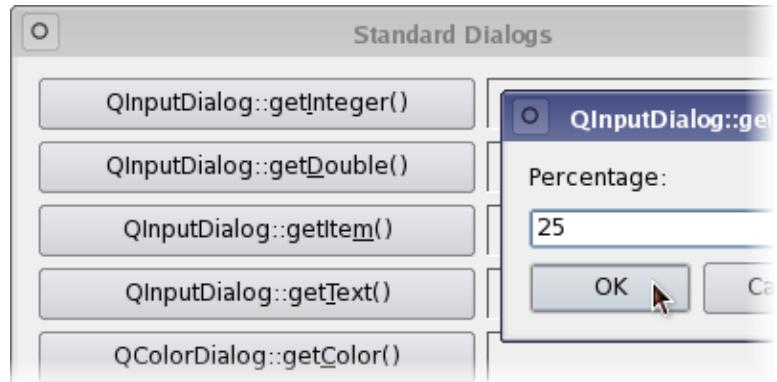
- › QErrorMessage::qtHandler()

- › Installs an error handler for debugging
 - › Shows qDebug(), qWarning() and qFatal() messages in QErrorMessage box



Other Common Dialogs

- › Asking for Input – QInputDialog
 - › `QInputDialog::getText(...)`
 - › `QInputDialog::getInt(...)`
 - › `QInputDialog::getDouble(...)`
 - › `QInputDialog::getItem(...)`
- › Selecting Color – QColorDialog
 - › `QColorDialog::getColor(...)`
- › Selecting Font – QFontDialog
 - › `QFontDialog::getFont(...)`



Guiding the User - QWizard

- › Input dialog
 - › Consisting of sequence of pages
- › Purpose: Guide user through process
 - › Page by page
- › Supports
 - › Linear and non-linear wizards
 - › Registering and using fields
 - › Access to pages by ID
 - › Page initialization and cleanup
 - › Title, sub-title
 - › Logo, banner, watermark, background
- › Each page is a QWizardPage
 - › `QWizard::addPage()` adds page to wizard



Simple Wizard Example

```
QWizardPage *createIntroPage() {  
    QWizardPage *page = new QWizardPage;  
    page->setTitle("Introduction");  
    // create widgets and layout them return page;  
    return page;  
}  
  
QWizardPage *createRegistrationPage() { ... }  
  
int main(int argc, char *argv[]) {  
    QApplication app(argc, argv);  
    QWizard wizard;  
    wizard.setWindowTitle("License Wizard");  
    wizard.addPage(createIntroPage());  
    wizard.addPage(createRegistrationPage());  
    wizard.show();  
    return app.exec();  
}
```

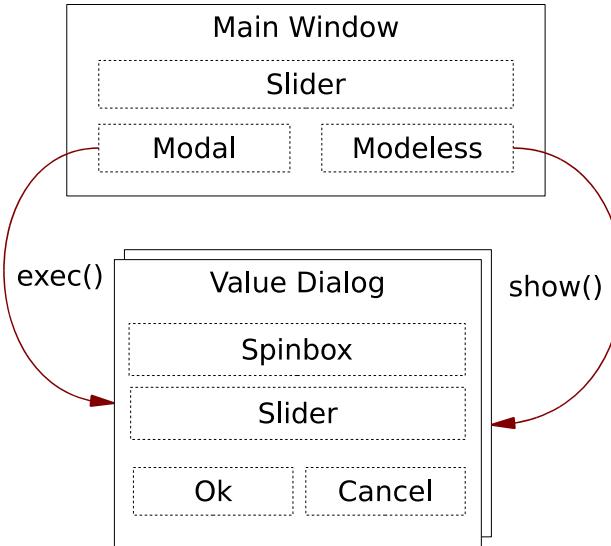
Questions And Answers

- › When would you use a modal dialog, and when would you use a non-modal dialog?
- › When should you call `exec()` and when should you call `show()`?
- › Can you bring up a modal dialog, when a modal dialog is already active?
- › When do you need to keep widgets as instance variables?
- › What is the problem with this code:

```
QDialog *dialog = new QDialog(parent);  
QCheckBox *box = new QCheckBox(dialog);
```

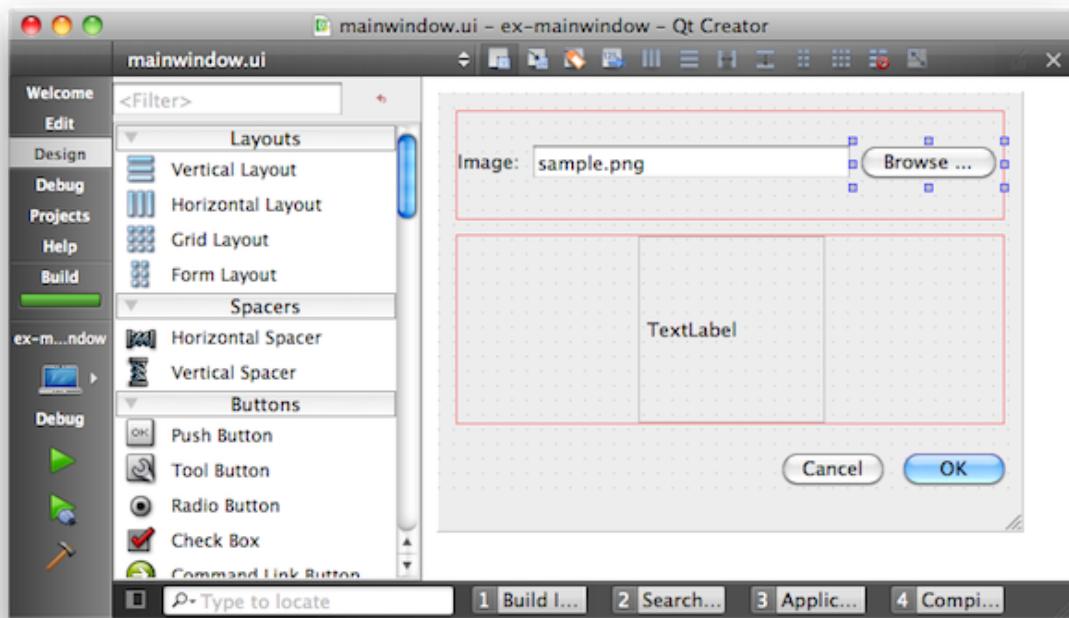
Lab – Custom Dialog

- › We create a simple value dialog
 - › Shows `int` value
 - › As slider
 - › As spin box
 - › Value must be < 50 to be accepted
- › A main window will show result
 - › Has a slider, connected to dialog
 - › Two buttons to launch dialog in modal and modeless mode

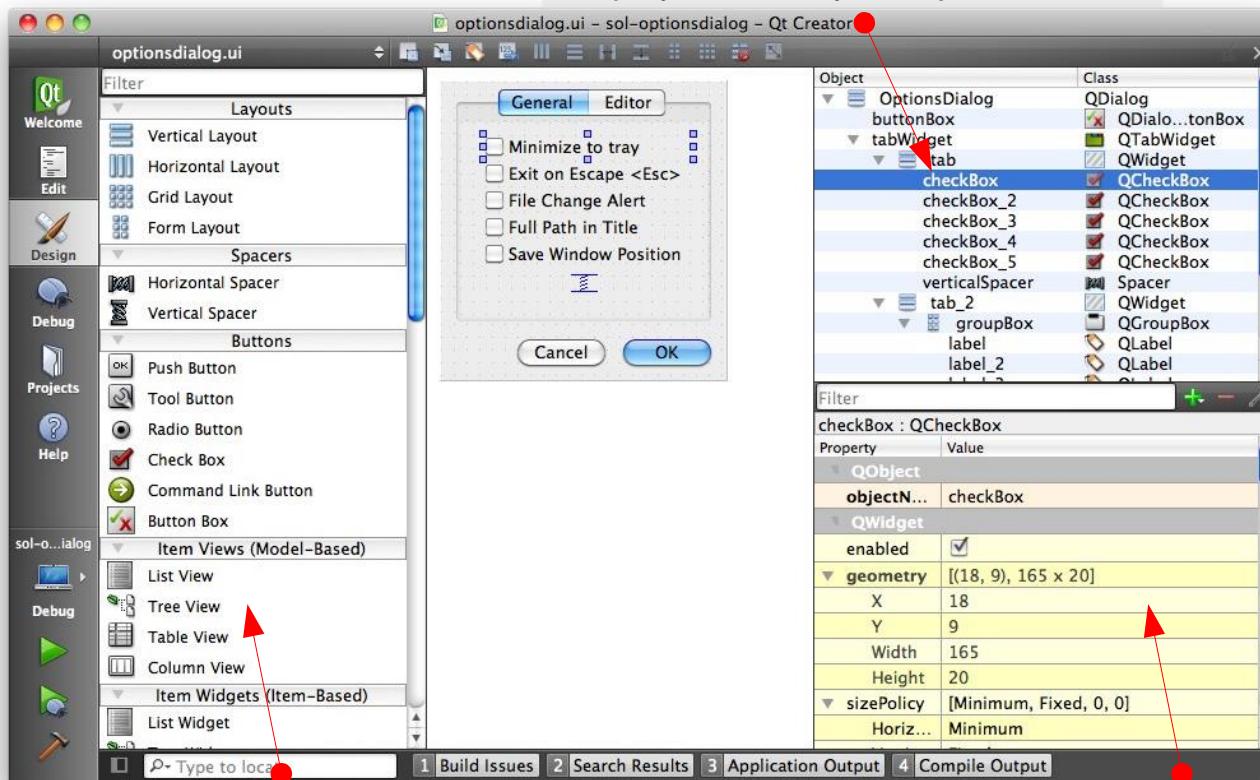


Qt Designer

- > Design UI forms visually
- > Visual Editor for
 - > Signal/slot connections
 - > Actions
 - > Tab handling
 - > Buddy widgets
 - > Widget properties
 - > Integration of custom widgets
 - > Resource files



Designer Views



Widget Box
Provides selection of widgets, layouts

Property Editor
Displays properties of selected object

Designer's Editing Modes

- >  Widget Editing
 - > Change appearance of form
 - > Add layouts
 - > Edit properties of widgets
- >  Signal and Slots Editing
 - > Connect widgets together with signals & slots
- >  Buddy Editing
 - > Assign buddy widgets to label
 - > *Buddy widgets help keyboard focus handling correctly*
- >  Tab Order Editing
 - > Set order for widgets to receive the keyboard focus

Designer UI Form Files

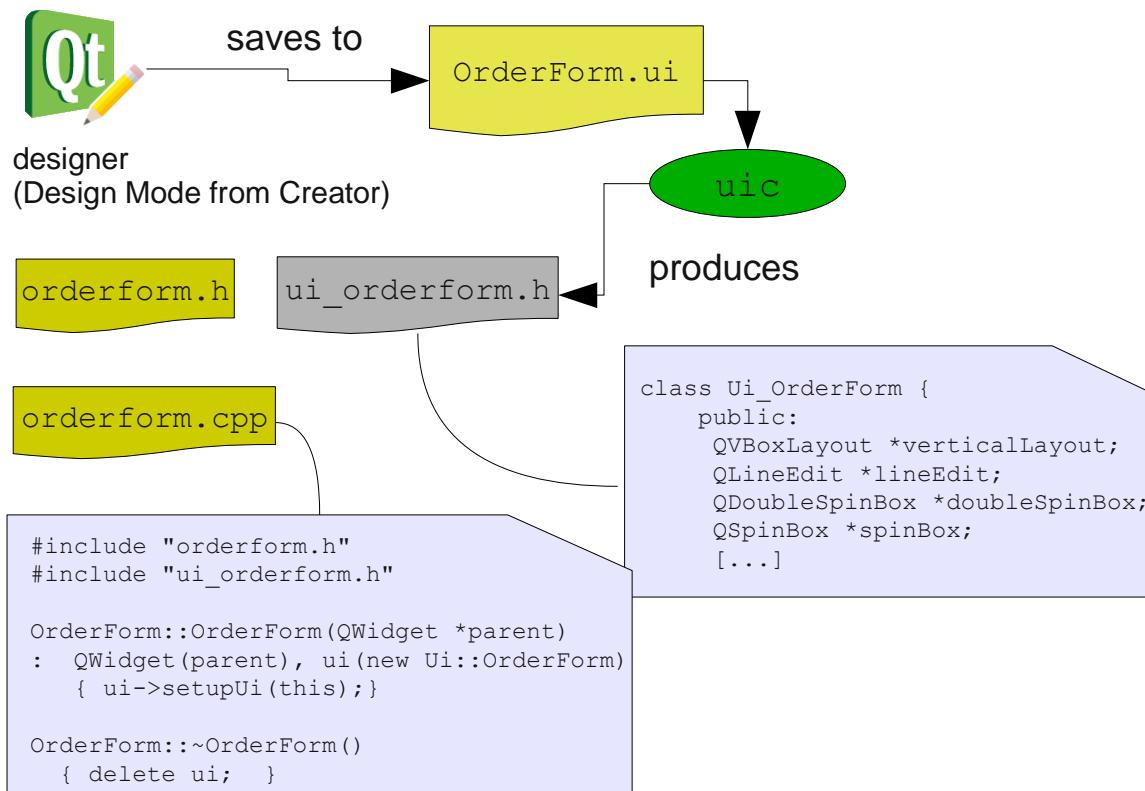
- > Form stored in .ui file
 - > Format is XML
- > uic tool generates code
 - > From myform.ui to ui_myform.h

```
// ui_mainwindow.h
class Ui_MainWindow {
public:
    QLineEdit *fileName;
    ... // simplified code
    void setupUi(QWidget *) { /* setup widgets */ }
};
```

```
<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
<class>MainWindow</class>
<widget class="QMainWindow" name="MainWindow">
    <widget class="QLineEdit" name="fileName">
        <property name="text">
            <string>sample.png</string>
        </property>
    </widget>
</ui>
```

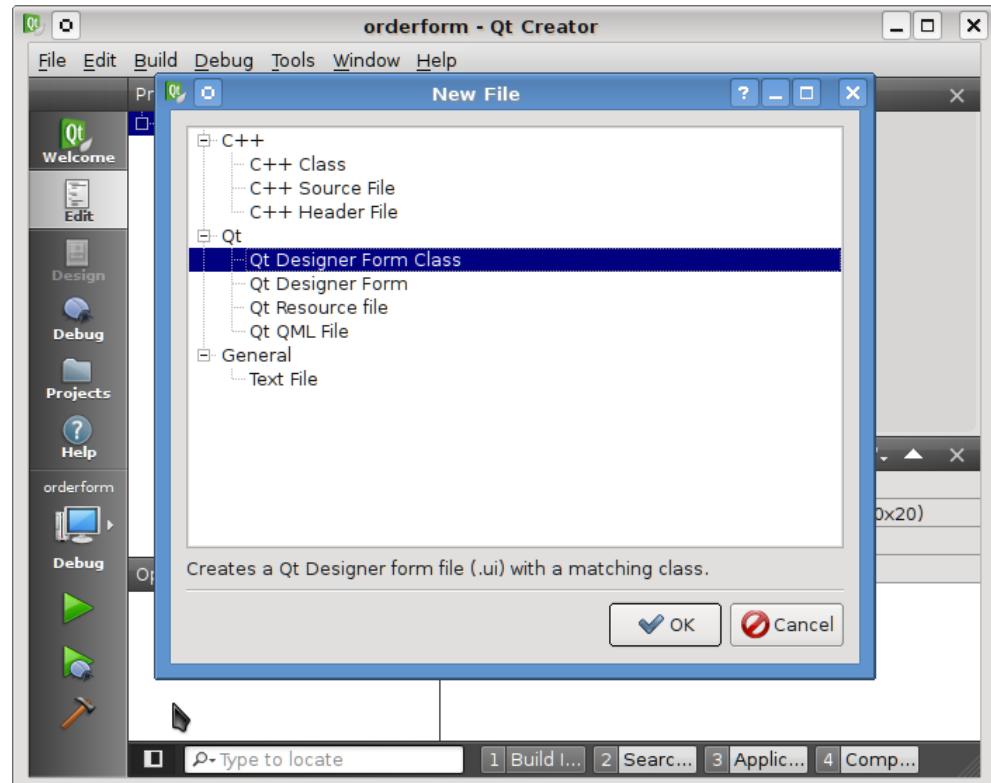
- > Form ui file in project (.pro)
 - > FORMS += mainwindow.ui

From .ui to C+



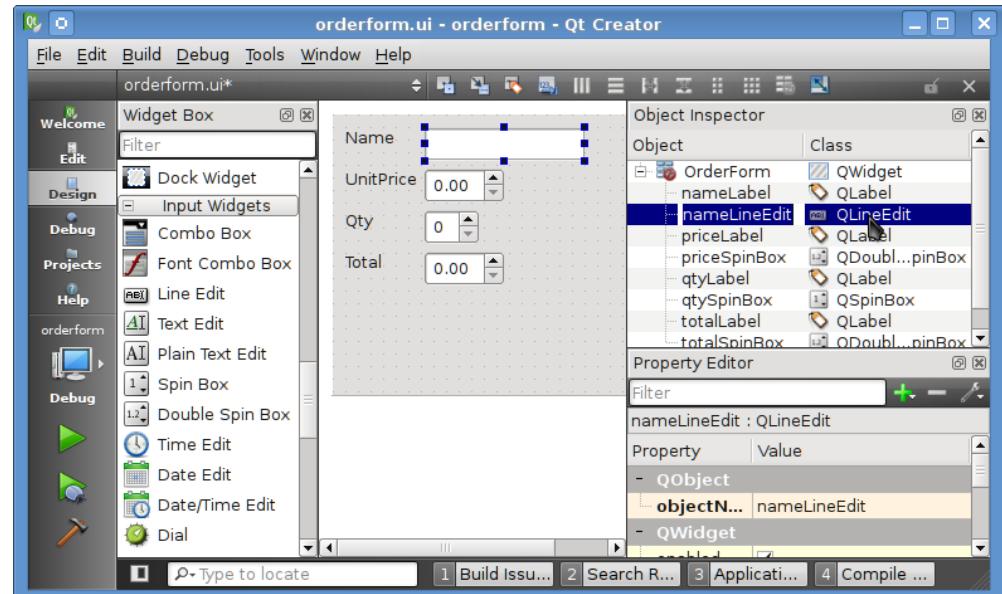
Qt Creator - Form Wizards

- > **Add New..."Designer Form"**
 - > Or "Designer Form Class" (for C++ integration)



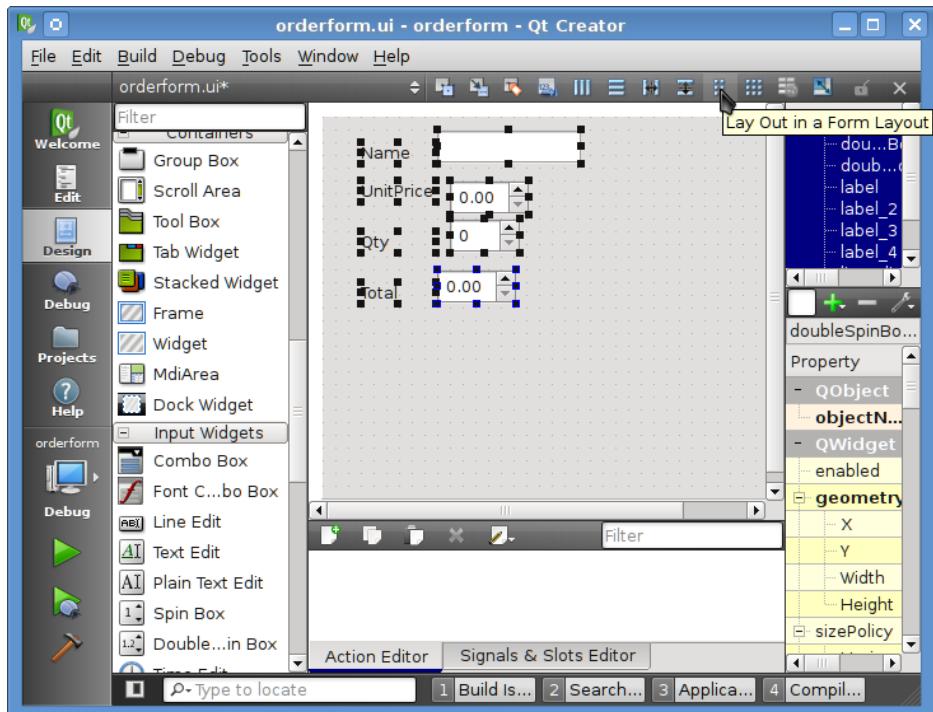
Naming Widgets

- › Place widgets on form
- › Edit `objectName` property
 - › *objectName defines member name in generated code*



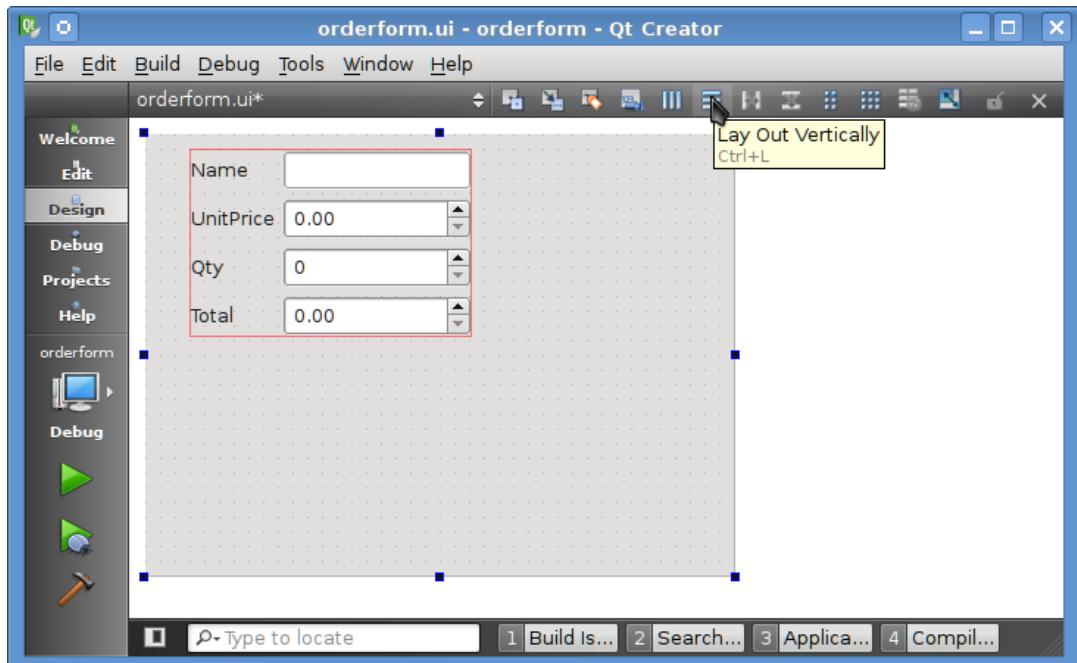
Form Layout in Designer

- › QFormLayout: Suitable for most input forms



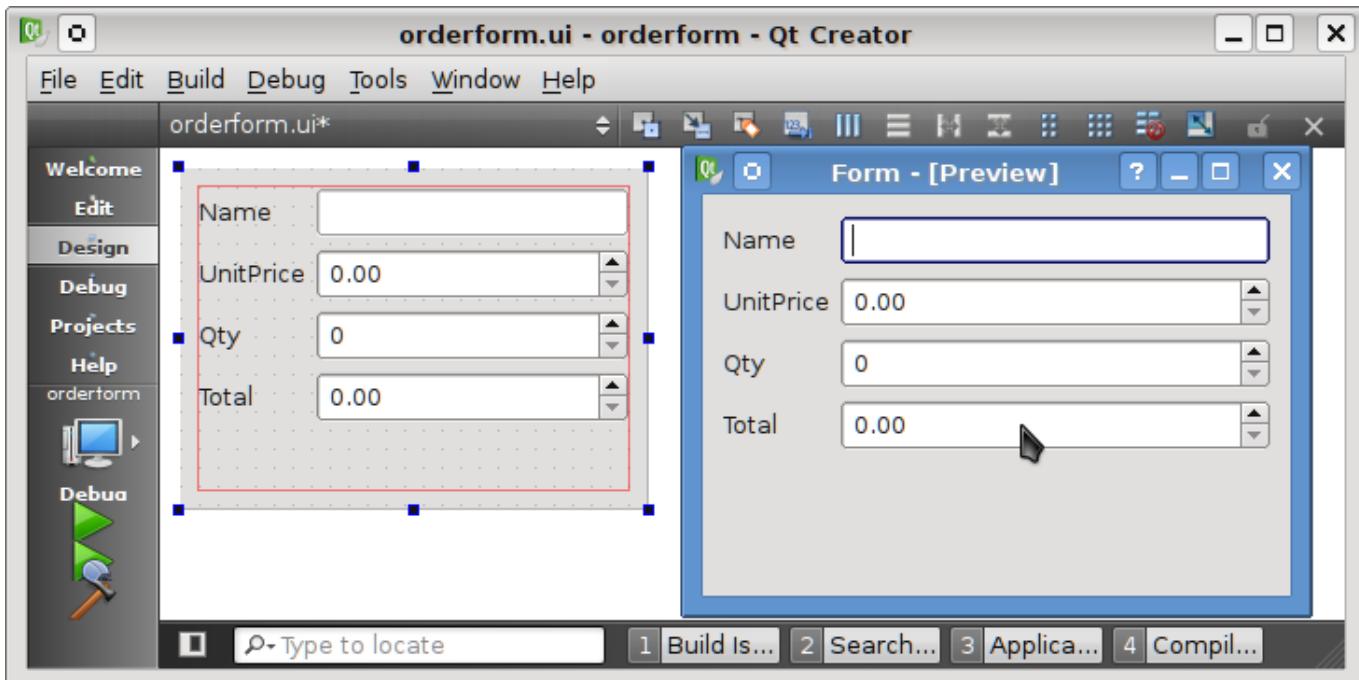
Top-Level Layout

- › First layout child widgets
- › Finally select empty space and set top-level layout



Preview Widget in Preview Mode

- › Check that widget is nicely resizable



Code Integration - Header File

- › "Your Widget" derives from appropriate base class
- › ***ui** member encapsulate UI class
 - › Makes header independent of designer generated code

```
// orderform.h
class Ui_OrderForm;
class OrderForm : public QDialog {
    private: Ui_OrderForm *ui; // pointer to UI object
};
```

Code Integration - Implementation File

- › *Default behavior in Qt Creator*

```
// orderform.cpp
#include "ui_orderform.h"

OrderForm::OrderForm(QWidget *parent) :
    QDialog(parent),
    ui(new Ui_OrderForm)
{
    ui->setupUi(this);
}

OrderForm::~OrderForm()
{
    delete ui;
    ui=0;
}
```

Signals and Slots in Designer

- › Widgets are available as public members
 - › `ui->fileName->setText ("image.png")`
 - › *Name based on widgets object name*
- › You can set up signals & slots traditionally...
 - › `connect(ui->okButton, SIGNAL(clicked()), ...)`
- › Auto-connection facility for custom slots
 - › Automatically connect signals to slots in your code
 - › Based on object name and signal
 - › `void on_objectName_signal(parameters);`
 - › Example: `on_okButton_clicked()` slot
- › Qt Creator: right-click on widget and "Go To Slot"
 - › Generates a slot using auto-connected name

Using Custom Widgets in Designer

- › Forms can be processed at runtime

- › Produces dynamically generated user interfaces

- › Disadvantages

- › Slower, harder to maintain

- › Risk: .ui file not available at runtime

- › Loading .ui file

```
QUiLoader loader;  
QFile file("forms/textfinder.ui");  
file.open(QFile::ReadOnly);  
QWidget *formWidget = loader.load(&file, this);
```

- › Locate objects in form

```
ui_okButton = qFindChild<QPushButton*>(this, "okButton");
```

Summary

- › Dialogs in Qt are specialized widgets
- › Both modal and modeless dialogs are supported
 - › Modal dialogs start a nested event loop, preventing other windows in the application to handle events
- › Dialogs, like any forms, can be designed with Qt Designer

Lab – Designer Order Form

- › Create an order form dialog
 - › With fields for price, quantity and total.
 - › Total field updates itself to reflect quantity and price entered

A screenshot of a Mac OS X style order form dialog. The dialog has a light gray background and a title bar at the top. Inside, there are four text fields with accompanying up/down arrows for adjusting values:

- Name: My Item
- Quantity: 8
- Price: 2,00
- Total: 16,00

At the bottom right are two buttons: "Cancel" and "OK". The "OK" button is highlighted with a blue glow, indicating it is the primary action button.

Contents

- › Model/View Concept
- › Showing Simple Data
- › Proxy Models
- › Custom Models

Objectives

Using Model/View

- › Introducing to the concepts of model-view
- › Showing Data using standard item models
- › Understand the limitations of standard item models
- › How to interface your model with a data backend
- › Understand what are proxy models and how to use them

Custom Models

- › Writing a simple read-only custom model

Why Model/View?

- › **Isolated domain-logic**

- › From input and presentation

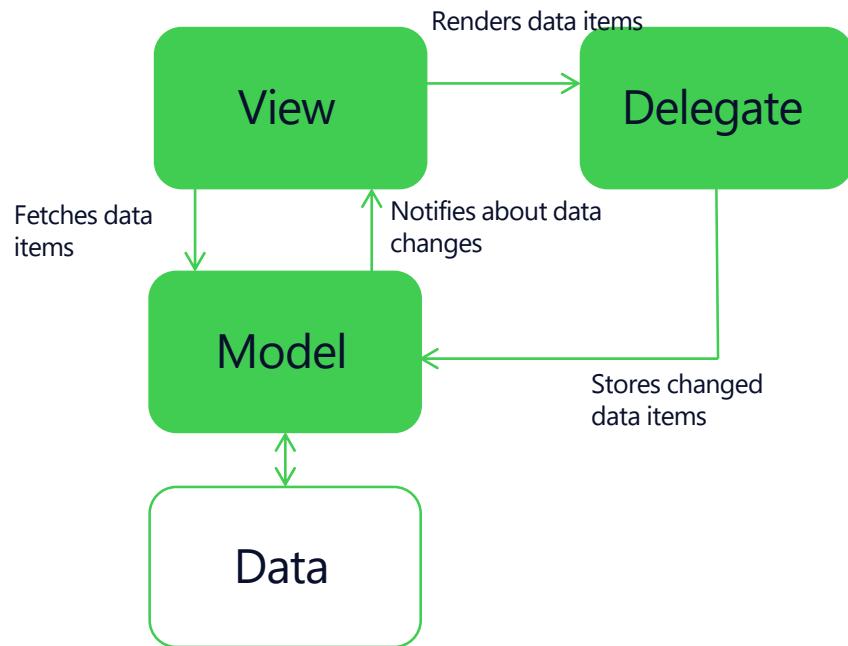
- › **Makes Components Independent**

- › For Development
 - › For Testing
 - › For Maintenance

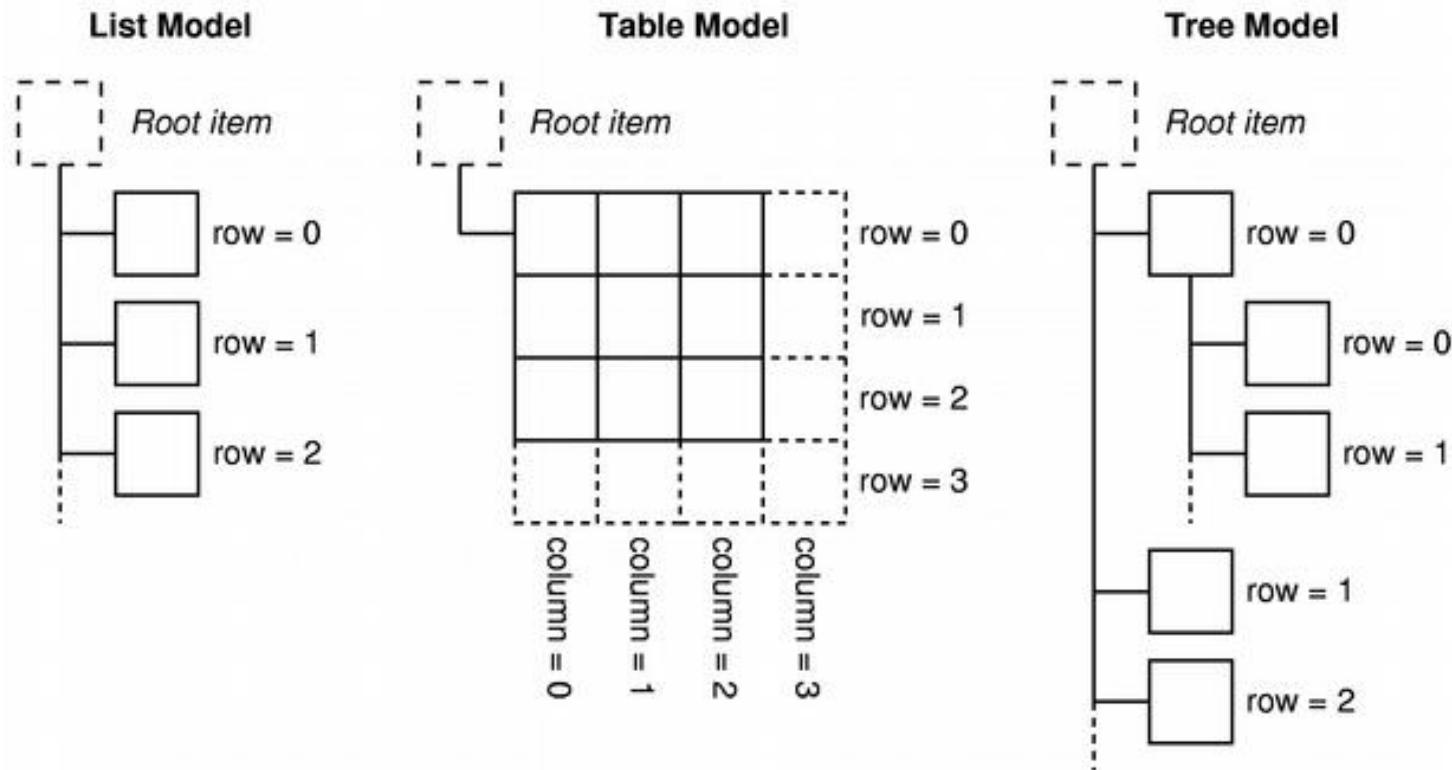
- › **Foster Component Reuse**

- › Reuse of Presentation Logic
 - › Reuse of Domain Model

Model/View Components



Model Structures



Display the Structure - View Classes

> **QtQuick ItemView**

- > Abstract base class for scrollable views

> **QtQuick ListView**

- > Items of data in a list

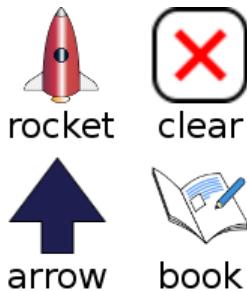
> **QtQuick GridView**

- > Items of data in a grid

> **QtQuick PathView**

- > Items of data along a specified path

Alice
Bob
Jane
Victor
Wendy



Adapts the Data - Model Classes

> **QAbstractItemModel**

- > Abstract interface of models

> **Abstract Item Models**

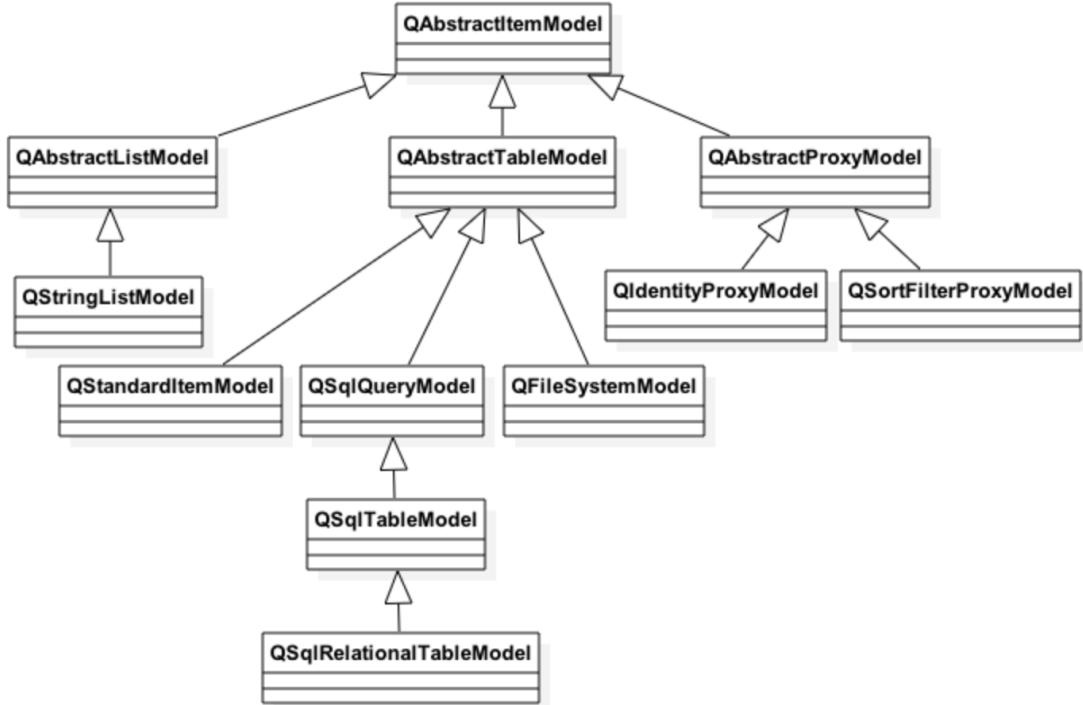
- > Implement to use

> **Ready-Made Models**

- > Convenient to use

> **Proxy Models**

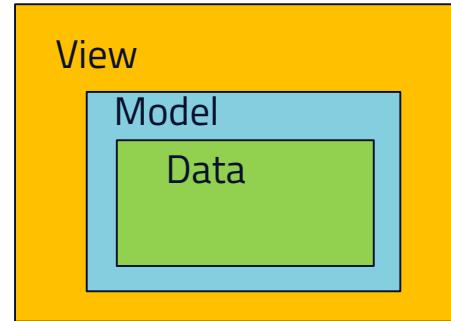
- > Reorder/filter/sort your items



Data - Model - View Relationships

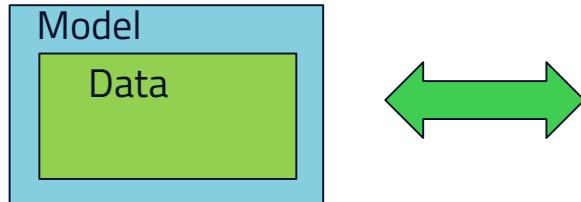
> Standard Item Model

- > Data+Model combined
- > View is separated
- > Model is your data



> Custom Item Models

- > Model is adapter to data
- > View is separated



Addressing Data - QModelIndex

- › Refers to item in model
- › Contains all information to specify location
- › Located in given row and column
 - › May have a parent index
- › **QModelIndex API**
 - › `row()` – row index refers to
 - › `column()` – column index refers to
 - › `parent()` – parent of index
 - › Or `QModelIndex()` if no parent
 - › `isValid()`
 - › Valid index belongs to a model
 - › Valid index has non-negative row and column numbers
 - › `model()` – the model index refers to
 - › `Data(role)` - data for given role

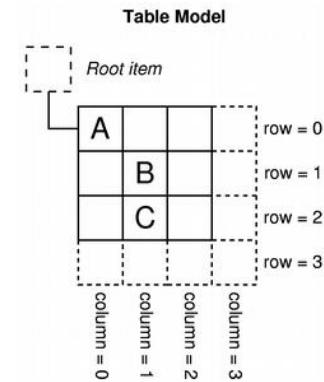
QModelIndex in Table/Tree Structures

> Rows and columns

> Item location in table model

> Item has no parent (`parent.isValid() == false`)

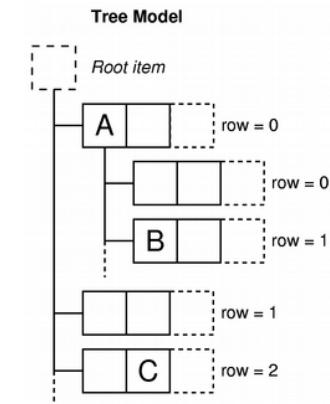
```
indexA = model->index(0, 0, QModelIndex());  
indexB = model->index(1, 1, QModelIndex());  
indexC = model->index(2, 1, QModelIndex());
```



> Parents, rows, and columns

> Item location in tree model

```
indexA = model->index(0, 0, QModelIndex());  
indexC = model->index(2, 1, QModelIndex());  
// asking for index with given row, column and parent  
indexB = model->index(1, 0, indexA);
```



Item and Item Roles

- > **Item performs various roles**
 - > For other components (delegate, view,...)

- > **Supplies different data**

- > For different situations

- > **Example:**

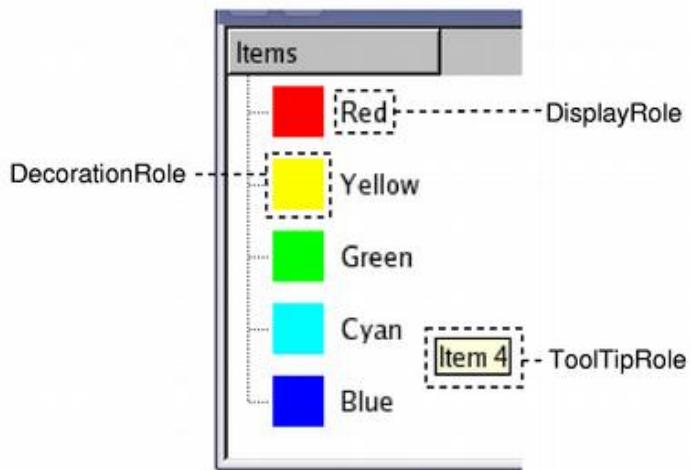
- > `Qt::DisplayRole` used display string in view

- > **Asking for data**

```
QVariant value = model->data(index, role);  
// Asking for display text  
QString text = model->data(index, Qt::DisplayRole).toString()
```

- > **Standard roles**

- > Defined by `Qt::ItemDataRole`



Mapping Item Roles from/to QML

- › Item Roles in C++

```
// Asking for display text
QString text = model->data(index, Qt::DisplayRole).toString()
```

- › Item properties in QML

```
onCurrentIndexChanged: {
    var text = model.get(index).display
}
```

- › Default mappings

- › Qt::DisplayRole in C++ is display in QML
- › Qt::DecorationRole in C++ is decoration in QML

- › Add additional mappings by re-implementing

- › QAbstractItemModel::roleNames()

Exporting Models to QML

- › Export model instance

- › Create model instance in C++
- › Set as a context property on the view's engine

```
CustomModel *model = new CustomModel;  
QQuickView view;  
view.engine()->rootContext("_model", model);
```

- › Use in QML by id

```
ListView { model: _model }
```

- › Export model type

- › Register custom model class with QML type system

```
qmlRegisterType<CustomModel>("Models", 1, 0, "CustomModel");
```

- › Use in QML like any other QML element

```
import Models 1.0  
ListView {  
    model: CustomModel {}  
}
```

Recap of Model/View Concept

> **Model Structures**

- > List, Table and Tree

> **Components**

- > Model - Adapter to Data
- > View - Displays Structure
- > Delegate – Paints Item
- > Index – Location in Model

> **Views**

- > ListView
- > GridView
- > PathView

> **Models**

- > QAbstractItemModel
- > Other Abstract Models
- > Ready-Made Models
- > Proxy Models

> **Index**

- > row(), column(), parent()
- > data(role)
- > model()

> **Item Role**

- > Qt::DisplayRole
- > Standard Roles in
 - > Qt::ItemDataRoles

QStandardItemModel - Convenient Model

- › QStandardItemModel

- › Classic item-based approach
 - › Only practical for small sets of data

```
model = new QStandardItemModel(parent);
item = new QStandardItem("A (0,0)");
model->appendRow(item);
model->setItem(0, 1, new QStandardItem("B (0,1)"));
item->appendRow(new QStandardItem("C (0,0)"));
```

- › 'B(0,1)' and 'C(0,0)' – Not visible. (list view is only 1-dimensional)

Meet the City Engine

- > Our Demo Model
 - > 62 most populous cities of the world
 - > Data in CSV file
- > Data Columns
 - > *City | Country | Population | Area | Flag*
- > Implemented as data backend
 - > Internal implementation is hidden
 - > Code in `CityEngine` class

```
City;Country;Population;Area  
Shanghai;China;13831900;1928  
Mumbai;India;13830884;603;22  
Karachi;Pakistan;12991000;35  
Delhi;India;12565901;431.09;  
Istanbul;Turkey;11372613;183  
São Paulo;Brazil;11037593;15  
Moscow;Russia;10508971;1081;  
Seoul;South Korea;10464051;6  
Beijing;China;10123000;1368.  
Mexico City;Mexico;8841916;1  
Tokyo;Japan;8795000;617;22px  
Kinshasa;Democratic Republic  
Jakarta;Indonesia;8489910;66  
New York City;United States;
```

Our Backend CityEngine API

```
public CityEngine : public QObject {
    // returns all city names
    QStringList cities() const;

    // returns country by given city name
    QString country(const QString &cityName) const;

    // returns population by given city name
    int population(const QString &cityName) const;

    // returns city area by given city name
    qreal area(const QString &cityName) const;

    // returns country flag by given country name
    QIcon flag(const QString &countryName) const;

    // returns all countries
    QStringList countries() const;

    // returns city names filtered by country
    QStringList citiesByCountry(const QString& countryName) const;
};
```

Lab: Standard Item Model for CityEngine

- › Implement `setupModel()` in `citymodel.cpp`
- › Display cities grouped by countries



Proxy Model - QSortFilterProxyModel

- › QSortFilterProxyModel

- › Transforms structure of source model
 - › Maps indexes to new indexes

```
view = new QQuickView(parent);  
// insert proxy model between model and view  
proxy = new QSortFilterProxyModel(parent);  
proxy->setSourceModel(model);  
view->engine()->rootContext()->setContextProperty("_proxy", proxy);
```

- › *Note:* Need to load all data to sort or filter

Sorting/Filtering - QSortFilterProxyModel

› Filter with Proxy Model

```
// filter column 1 by "India"  
proxy->setFilterWildcard("India");  
proxy->setFilterKeyColumn(1);
```

› Sorting with Proxy Model

```
// sort column 0 ascending  
proxy->sort(0, Qt::AscendingOrder);
```

› Filter via TextInput's signal

```
TextInput {  
    onTextChanged: _proxy.setFilterWildcard(text)  
}
```

Implementing a Model

Variety of classes to choose from

- > **QAbstractListModel**

- > One dimensional list

- > **QAbstractTableModel**

- > Two-dimensional tables

- > **QAbstractItemModel**

- > Generic model class

- > **QStringListModel**

- > One-dimensional model
 - > Works on string list

- > **QStandardItemModel**

- > Model that stores the data

Step 1: Read Only List Model

```
class MyModel: public QAbstractListModel
{
public:
    // return row count for given parent
    int rowCount( const QModelIndex &parent) const;

    // return data, based on current index and requested role
    QVariant data( const QModelIndex &index,
                   int role = Qt::DisplayRole) const;
};
```

Step 2: Supplying Header Information

```
QVariant MyModel::headerData(int section,
                             Qt::Orientation orientation,
                             int role) const
{
    // return column or row header based on orientation
}
```

Step 3: Enabling Editing

```
// should contain Qt::ItemIsEditable
Qt::ItemFlags MyModel::flags(const QModelIndex &index) const
{
    return QAbstractListModel::flags() | Qt::ItemIsEditable;
}

// set role data for item at index to value
bool MyModel::setData( const QModelIndex & index,
                      const QVariant & value,
                      int role = Qt::EditRole)
{
    ... = value; // set data to your backend
    emit dataChanged(topLeft, bottomRight); // if successful
}
```

Step 4: Row Manipulation

```
// insert count rows into model before row
bool MyModel::insertRows(int row, int count, parent)
{
    beginInsertRows(parent, first, last);
    // insert data into your backend
    endInsertRows();
}

// removes count rows from parent starting with row
bool MyModel::removeRows(int row, int count, parent)
{
    beginRemoveRows(parent, first, last);
    // remove data from your backend
    endRemoveRows();
}
```

Lab – *City List Model*

- › Please implement a City List Model
- › Given:
 - › Start with solution of modelview/lab-cities-standarditem
- › Your Task:
 - › Rebase `CityModel` to `QAbstractListModel`
- › **Optional**
 - › Make the model editable
 - › Enable adding / removing cities

Contents

- › Delegates
- › Editing item data
- › Data Widget Mapper
- › Drag and Drop
- › Custom Tree Model

Objectives

Custom Model/View

- › Editable Models
- › Custom Delegates
- › Using Data Widget Mapper
- › Custom Proxy Models
- › Drag and Drop

Item Delegates

- › `QAbstractItemDelegate` subclasses
 - › Control appearance of items in views
 - › Provide edit and display mechanisms
- › `QItemDelegate`, `QStyledItemDelegate`
 - › Default delegates
 - › Suitable in most cases
 - › Model needs to provide appropriate data
- › When to go for Custom Delegates?
 - › More control over appearance of items

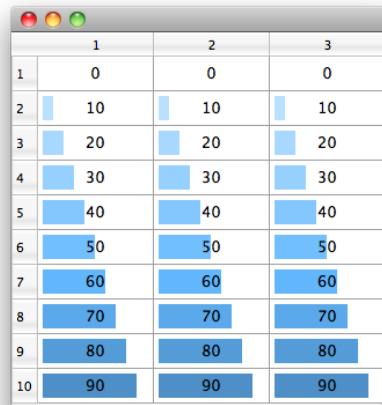
Item Appearance

- > Data table shown has no custom delegate
- > No need for custom delegate!
- > Use `Qt::ItemRole` to customize appearance

| | 1 | 2 | 3 | 4 | 5 |
|----|-------|-------|-------|-------|-------|
| 1 | R0-C0 | R0-C1 | R0-C2 | R0-C3 | R0-C4 |
| 2 | R1-C0 | R1-C1 | R1-C2 | R1-C3 | R1-C4 |
| 3 | R2-C0 | R2-C1 | R2-C2 | R2-C3 | R2-C4 |
| 4 | R3-C0 | R3-C1 | R3-C2 | R3-C3 | R3-C4 |
| 5 | R4-C0 | R4-C1 | R4-C2 | R4-C3 | R4-C4 |
| 6 | R5-C0 | R5-C1 | R5-C2 | R5-C3 | R5-C4 |
| 7 | R6-C0 | R6-C1 | R6-C2 | R6-C3 | R6-C4 |
| 8 | R7-C0 | R7-C1 | R7-C2 | R7-C3 | R7-C4 |
| 9 | R8-C0 | R8-C1 | R8-C2 | R8-C3 | R8-C4 |
| 10 | R9-C0 | R9-C1 | R9-C2 | R9-C3 | R9-C4 |

Delegate from QAbstractItemDelegate

```
class BarGraphDelegate : public QAbstractItemDelegate {
public:
    void paint(QPainter *painter,
               const QStyleOptionViewItem &option,
               const QModelIndex &index) const;
    QSize sizeHint(const QStyleOptionViewItem &option,
                  const QModelIndex &index) const;
};
```



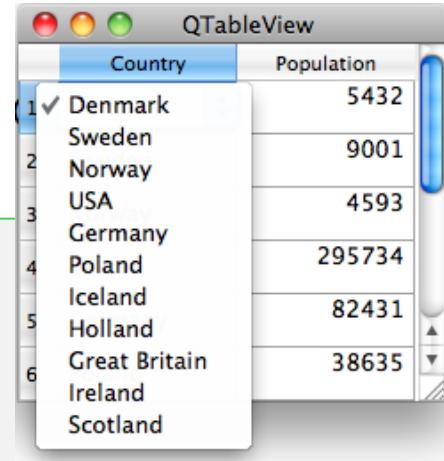
Integer Value - Bar Graph Delegate

```
void BarGraphDelegate::paint(painter, option, index) const
{
    if(index.data(Qt::EditRole).userType() == QVariant::Int) {
        int value = index.data(Qt::EditRole).toInt();
        // prepare rect with a width proportional to value
        QRect rect(option.rect.adjusted(4,4,-4,-4));
        rect.setWidth(rect.width()*value/MAX_VALUE);
        // draw the value bar
        painter->fillRect(rect, QColor("steelblue").lighter(value));
        painter->drawText(option.rect, index.data().toString());
    }
}
QSize BarGraphDelegate::sizeHint(option, index) const
{
    Q_UNUSED(index)
    return QSize(MIN_BAR_WIDTH, option.fontMetrics.height());
}
```

An Editor Delegate

- › Provides QComboBox
 - › For editing a series of values

```
class CountryDelegate : public QItemDelegate
{
public:
    // returns editor for editing data
    QWidget *createEditor( parent, option, index ) const;
    // sets data from model to editor void
    setEditorData( editor, index ) const;
    // sets data from editor to model
    void setModelData( editor, model, index ) const;
    // updates geometry of editor for index
    void updateEditorGeometry( editor, option, index ) const;
};
```



A screenshot of a Qt application window titled "QTableView". The window contains a table with two columns: "Country" and "Population". The data is as follows:

| | Country | Population |
|---|---------------|------------|
| 1 | Denmark | 5432 |
| 2 | Sweden | 9001 |
| 3 | Norway | 4593 |
| 4 | USA | 295734 |
| 5 | Germany | 82431 |
| 6 | Poland | 38635 |
| | Iceland | |
| | Holland | |
| | Great Britain | |
| | Ireland | |
| | Scotland | |

Providing an Editor

- › Create editor by index

```
QWidget *CountryDelegate::createEditor( ... ) const {
    QComboBox *editor = new QComboBox(parent);
    editor->addItems( m_countries );
    return editor;
}
```

- › Set data to editor

```
void CountryDelegate::setEditorData( ... ) const {
    QComboBox* combo = static_cast<QComboBox*>( editor );
    QString country = index.data().toString();
    int idx = m_countries.indexOf( country );
    combo->setCurrentIndex( idx );
}
```

Submitting Data to the Model

- › When user finished editing
 - › View asks delegate to store data into model

```
void CountryDelegate::setModelData(editor, model, index) const {  
    QComboBox* combo = static_cast<QComboBox*>( editor );  
    model->setData( index, combo->currentText() );  
}
```

- › If editor has finished editing

```
// copy editors data to model  
emit commitData( editor );  
// close/destroy editor  
emit closeEditor( editor, hint );  
// hint: indicates action performed next to editing
```

Updating the Editor's Geometry

- › Delegate manages editor's geometry
- › View provides geometry information
 - › `QStyleOptionViewItem`

```
void CountryDelegate::updateEditorGeometry( ... ) const {
    // don't allow to get smaller than editors sizeHint()
    QSize size = option.rect.size().expandedTo(editor->sizeHint());
    QRect rect(QPoint(0,0), size);
    rect.moveCenter(option.rect.center());
    editor->setGeometry( rect );
}
```

- › Case of multi-index editor
 - › Position editor in relation to indexes

Setting Delegates on Views

- › `view->setItemDelegate(...)`
- › `view->setItemDelegateForColumn(...)`
- › `view->setItemDelegateForRow(...)`

Type-Based Delegates

- › Our color editor widget



```
class ColorListEditor : public QComboBox {  
    ...  
};
```

- › Registering editor for type QVariant::Color

```
QItemEditorFactory *factory = new QItemEditorFactory;  
QItemEditorCreatorBase *creator = new  
    QStandardItemEditorCreator<ColorListEditor>();  
// registers item editor creator given type of data  
factory->registerEditor(QVariant::Color, creator);  
// new and existing delegates will use new factory  
QItemEditorFactory::setDefaultFactory(factory);
```

Data Widget Mapper- QDataWidgetMapper

- > Maps model sections to widgets
- > Widgets updated, when current index changes
- > Orientation
 - > Horizontal => Data Columns
 - > Vertical => Data Rows

| | Name | Address | Age |
|---|--------|----------------------------------|-----|
| 1 | Alice | 123 Main Street Market Town | 20 |
| 2 | Bob | PO Box 32 Mail Handling S... | 31 |
| 3 | Carol | The Lighthouse Remote Isl... | 32 |
| 4 | Donald | 47338 Park Avenue Big City | 19 |
| 5 | Emma | Research Station Base Cam... | 26 |

mapping

Simple Widget Mapper

| | | |
|-----------------|---|---|
| Name: | <input type="text" value="Carol"/> | <input type="button" value="Previous"/> |
| Address: | <input type="text" value="The Lighthouse Remote Island"/> | |
| Age (in years): | <input type="text" value="32"/> | <input type="button" value="Next"/> |

Using QDataWidgetMapper

› Mapping Setup

```
mapper = new QDataWidgetMapper(this);
mapper->setOrientation(Qt::Horizontal);
mapper->setModel(model);
// mapper->addMapping( widget, model-section)
mapper->addMapping(nameEdit, 0);
mapper->addMapping(addressEdit, 1);
mapper->addMapping(ageSpinBox, 2);
// populate widgets with 1st row
mapper->toFirst();
```

› Track Navigation

```
connect(nextButton, &QPushButton::clicked, mapper, &QSignalMapper::toNext);
connect(previousButton, &QPushButton::clicked, mapper,
&QSignalMapper::toPrevious);
```

Mapped Property - The USER Property

- › USER indicates property is user-editable property
- › Only one USER property per class
- › Used to transfer data between the model and the widget

```
addMapping(lineEdit, 0); // uses "text" user property  
addMapping(lineEdit, 0, "inputMask"); // uses named property
```

```
class QLineEdit : public QWidget  
{  
    ...  
    Q_PROPERTY(QString text  
              READ text WRITE setText NOTIFY textChanged  
              USER true) // USER property  
    ...  
};
```

Drag and Drop for Views

- › Enable the View

```
// enable item dragging
view->setDragEnabled(true);
// allow to drop internal or external items
view->setAcceptDrops(true);
// show where dragged item will be dropped
view->setDropIndicatorShown(true);
```

- › Model has to provide support for drag and drop operations

```
Qt::DropActions MyModel::supportedDropActions() const
{
    return Qt::CopyAction | Qt::MoveAction;
}
```

- › Model needs to support actions

- › For example `Qt::MoveAction`
- › Implement `MyModel::removeRows(...)`

Drag and Drop with QStandardItemModel

› Setup of Model

- › Model is ready by default
- › model->mimeTypes()
 - › "application/x-qabstractitemmodeldatalist"
 - › "application/x-qstandarditemmodeldatalist"
- › model->supportedDragActions()
 - › QDropEvent::Copy | QDropEvent::Move
- › model->supportedDropActions()
 - › QDropEvent::Copy | QDropEvent::Move

```
item = new QStandardItem("Drag and Droppable Item");
// drag by default copies item
item->setDragEnabled(true);
// drop mean adding dragged item as child
item->setDropEnabled(true);
```

Drag and Drop on QAbstractItemModel

```
class MyModel : public QAbstractItemModel {
public:
    // actions supported by the data in this model
    Qt::DropActions supportedDropActions() const;

    // for supported index return Qt::ItemIs(Drag|Drop)Enabled
    Qt::ItemFlags flags(const QModelIndex &index) const;

    // returns list of MIME types that are supported
    QStringList QAbstractItemModel::mimeTypes() const;

    // returns object with serialized data in mime formats
    QMimeData *mimeData(const QModelIndexList &indexes) const;

    // true if data and action can be handled, otherwise false
    bool dropMimeData(const QMimeData *data, Qt::DropAction action,
                      int row, int column, const QModelIndex &parent);
};
```

A Custom Tree Model in 5 Steps

- 1.Read-Only Model
- 2.Editable Model
- 3.Insert-Remove Model
- 4.Lazy Model
- 5.Drag and Drop Model

A (Simple) Node Structure

```
class Node {  
public:  
    Node(const QString& aText="No Data", Node *aParent=0);  
    ~Node();  
    QVariant data() const;  
  
public:  
    QString text;  
    Node *parent;  
    QList<Node*> children;  
};
```

Read-Only Model

```
class ReadOnlyModel : public QAbstractItemModel
{
public:
    ...
    QModelIndex index( row, column, parent ) const;
    QModelIndex parent( child ) const;

    int rowCount( parent ) const;
    int columnCount( parent ) const;
    QVariant data( index, role) const;

protected: // important helper methods
    QModelIndex indexForNode( Node *node) const;
    Node* nodeForIndex( const QModelIndex &index) const;
    int rowForNode( Node *node) const;
};
```

Editable Model

```
class EditableModel : public ReadOnlyModel {  
public:  
    ...  
    bool setData( index, value, role );  
    Qt::ItemFlags flags( index ) const;  
};
```

Insert Remove Model

```
class InsertRemoveModel : public EditableModel {  
public:  
    ...  
    void insertNode(Node *parentNode, int pos, Node *node);  
    void removeNode(Node *node);  
    void removeAllNodes();  
};
```

Lazy Model

```
class LazyModel : public ReadOnlyModel {
public:
    ...
    bool hasChildren( parent ) const;
    bool canFetchMore( parent ) const;
    void fetchMore( parent );
};
```

DnD Model

```
class DndModel : public InsertRemoveModel {
public:
    ...
    Qt::ItemFlags flags( index ) const;
    Qt::DropActions supportedDragActions() const;
    Qt::DropActions supportedDropActions() const;
    QStringList mimeTypes() const;
    QMimeData *mimeData( indexes ) const;

    bool dropMimeData(data, dropAction, row, column, parent);
    bool removeRows(row, count, parent);
    bool insertRows(row, count, parent);
};
```

Questions and Answers

- › In Qt model/view framework, what are view, model, delegate, role, item, index?
- › What kind of item model classes exist?
- › When would you implement a custom delegate?
- › How delegate data is written to the model?
- › How model data is sorted?
- › Who owns the model?
- › Can models be shared between views?
- › Is it possible to use the same model in widgets and QML?

Summary

- › Qt model/view framework allows the separation of model and views
- › Model classes derive from `QAbstractItemModel`
 - › Custom models can be implemented for memory and performance optimizations
- › Model data is referenced with index
 - › A data item may have several roles
 - › Roles will be used by the delegate to render the item
- › Model subclasses exist for sorting and filtering model data
- › Models may support
 - › Lazy loading
 - › Drag and drop

Contents

- › Using GraphicsView Classes
- › Coordinate Systems and Transformations
- › Creating Custom Items

Objectives

- › Using `QGraphicsView`-related classes
- › Coordinate Schemes, Transformations
- › Extending items
 - › Event handling
 - › Painting
 - › Boundaries

GraphicsView Framework

- › Provides:
 - › A surface for managing interactive 2D graphical items
 - › A view widget for visualizing the items
- › Uses MVC paradigm
- › Resolution Independent
- › Animation Support
- › Fast item discovery, hit tests, collision detection
 - › Using Binary Space Partitioning (BSP) tree indexes
- › Can manage large numbers of items (tens of thousands)
- › Supports zooming, printing and rendering

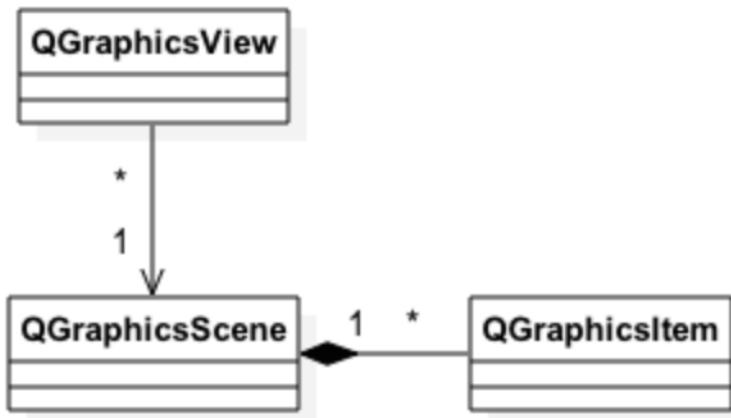
Hello World

```
#include <QtWidgets>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QGraphicsView view;
    QGraphicsScene *scene = new QGraphicsScene(&view);
    view.setScene(scene);
    QGraphicsRectItem *rect = new QGraphicsRectItem(-10, -10, 120, 50);
    scene->addItem(rect);
    QGraphicsTextItem *text = scene->addText("Hello World!");
    view.show();
    return app.exec();
}
```

UML Relationship

- › `QGraphicsScene` is:
 - › A "model" for `QGraphicsView`
 - › A "container" for `QGraphicsItems`



QGraphicsScene

- › Container for Graphic Items
 - Items can exist in only one scene at a time
- › Propagates events to items
 - › Manages Collision Detection
 - › Supports fast item indexing
 - › Manages item selection and focus
- › Renders scene onto view
 - › z-order determines which items show up in front of others

QGraphicsScene Important Methods

- › `addItem()`
 - › Add an item to the scene
 - › (remove from previous scene if necessary)
 - › Also `addEllipse()`, `addPolygon()`, `addText()`, etc.

```
QGraphicsEllipseItem *ellipse = scene->addEllipse(-10, -10, 120, 50);
QGraphicsTextItem *text = scene->addText("Hello World!");
```

- › `items()`
 - › Returns items intersecting a particular point or region
- › `selectedItems()`
 - › Returns list of selected items
- › `sceneRect()`
 - › Bounding rectangle for the entire scene

QGraphicsView

- › Scrollable widget view port onto the scene
 - › Zooming, rotation, and other transformations
 - › Translates input events (from the View) into `QGraphicsSceneEvents`
 - › Maps coordinates between scene and viewport
 - › Provides "level of detail" information to items
 - › Supports OpenGL

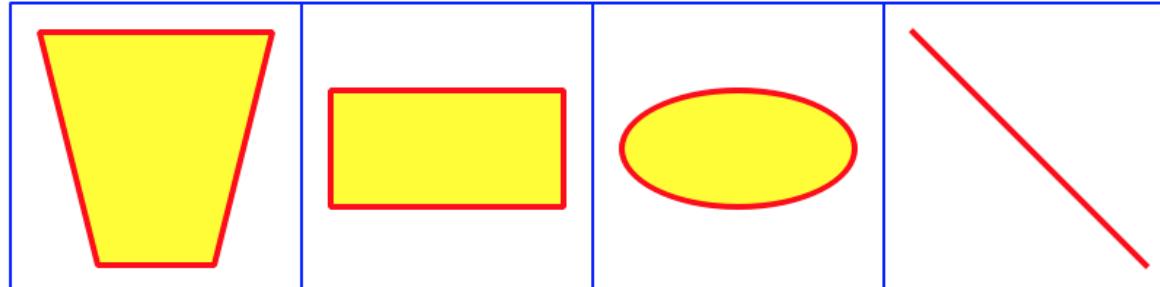
QGraphicsView Important Methods

- › `setScene()`
 - › Sets the QGraphicsScene to use
- › `setRenderHints()`
 - › antialiasing, smooth pixmap transformations, etc.
- › `centerOn()`
 - › Takes a QPoint or a QGraphicsItem as argument
 - › Ensures point/item is centered in View
- › `mapFromScene()`, `mapToScene()`
 - › Map to/from scene coordinates
- › `scale()`, `rotate()`, `translate()`, `matrix()`
 - › transformations

QGraphicsItem

- › Abstract base class: basic canvas element
 - › Supports parent/child hierarchy
- › Easy to extend or customize concrete items:
 - › QGraphicsRectItem, QGraphicsPolygonItem, QGraphicsPixmapItem, QGraphicsTextItem, etc.
 - › SVG Drawings, other widgets
- › Items can be transformed:
 - › move, scale, rotate
 - › Using local coordinate systems
- › Supports Drag and Drop similar to QWidget

Concrete QGraphicsItem Types

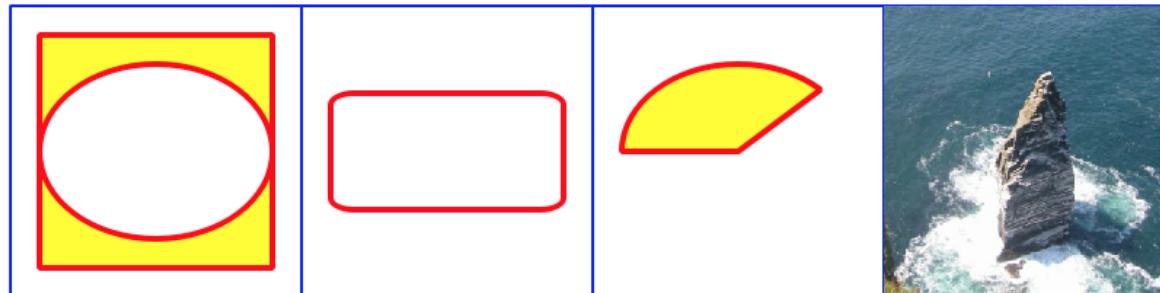


PolygonItem

RectItem

EllipseItem

LineItem



PainterPathItem

PainterPathItem
(roundedRect)

PainterPathItem
(filled arc)

PixmapItem

QGraphicsItem Important Methods

- › `pos()`
 - › Get the item's position in scene
- › `moveBy()`
 - › Moves an item relative to its own position.
- › `zValue()`
 - › Get a Z order for item in scene
- › `show(), hide() – set visibility`
- › `setEnabled(bool)` - disabled items can not take focus or receive events
- › `setFocus(Qt::FocusReason)` - sets input focus.
- › `setSelected(bool)`
 - › select/deselect an item
 - › Typically called from `QGraphicsScene::setSelectionArea()`

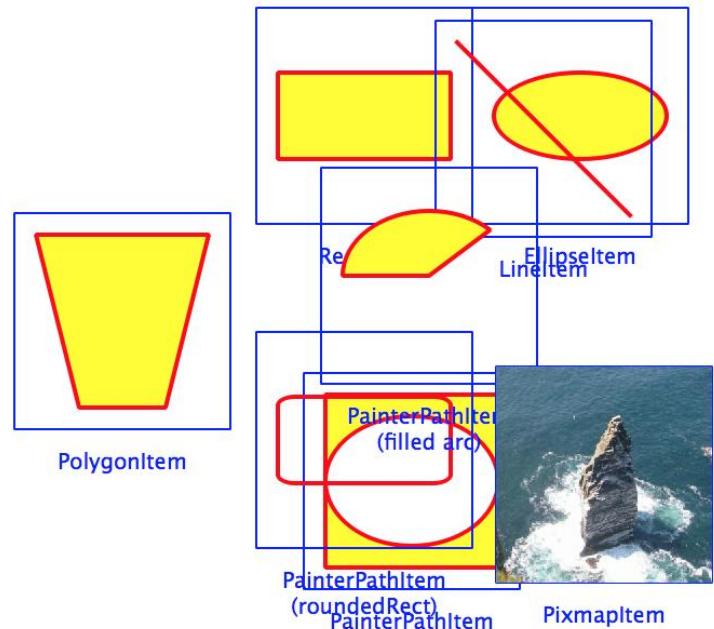
Select, Focus, Move

- › `QGraphicsItem::setFlags()`
 - › Determines which operations are supported on an item
- › `QGraphicsItemFlags`
 - › `QGraphicsItem::ItemIsMovable`
 - › `QGraphicsItem::ItemIsSelectable`
 - › `QGraphicsItem::ItemIsFocusable`

```
item->setFlags(QGraphicsItem::ItemIsMovable |
QGraphicsItem::ItemIsSelectable);
```

Groups of Items

- › Any `QGraphicsItem` can have children
- › `QGraphicsItemGroup` is an invisible item for grouping child items
- › To group child items in a box with an outline (for example), use a `QGraphicsRectItem`
- › Try dragging boxes in demo:



Parents and Children

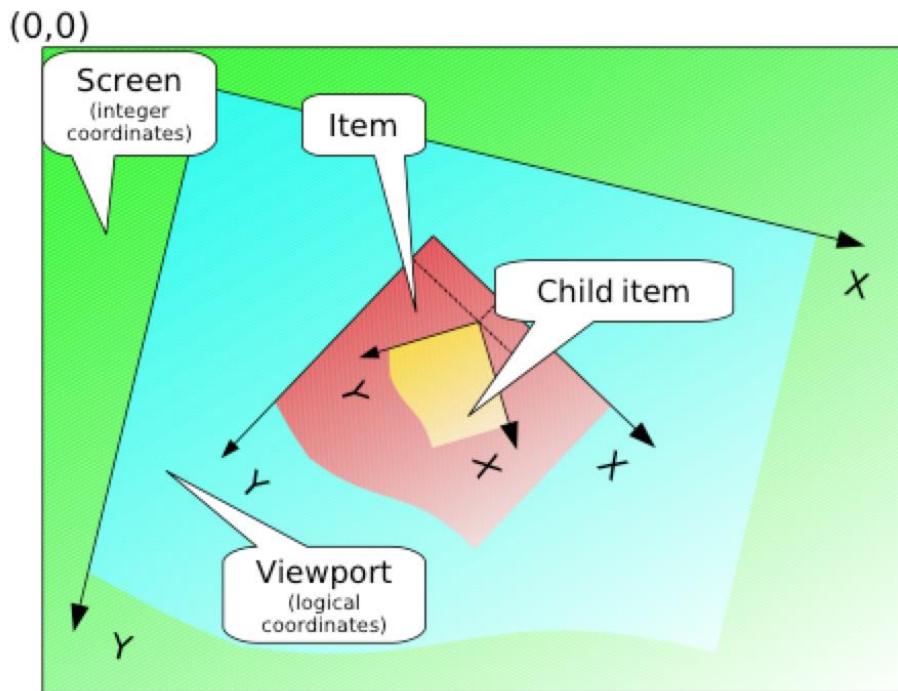
- › Parent propagates values to child items:

- › `setEnabled()`
 - › `setFlags()`
 - › `setPos()`
 - › `setOpacity()`
 - › etc...

- › Enables composition of items.

Coordinate Systems

- Each View and Item has its own local coordinate system



Coordinates

- › Coordinates are local to an item
 - › Logical coordinates, not pixels
 - › Floating point, not integer
 - › Without transformations, 1 logical coordinate = 1 pixel.
- › Items inherit position and transform from parent
- › zValue is relative to parent
- › Item transformation does not affect its local coordinate system
- › Items are painted recursively
 - › From parent to children
 - › In increasing zValue order

QTransform

- › Coordinate systems can be transformed using `Qtransform`
- › `QTransform` is a 3×3 matrix describing a linear transformation from (x, y) to (xt, yt)

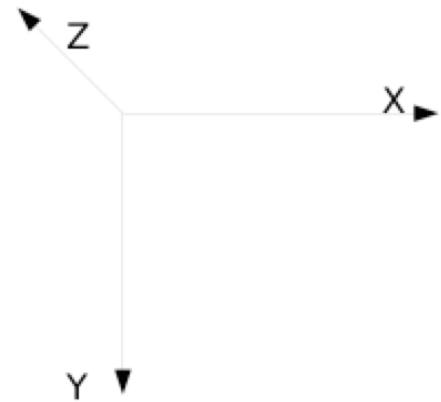
| col1 | col2 | col3 |
|----------|----------|----------|
| m_{11} | m_{12} | m_{13} |
| m_{21} | m_{22} | m_{23} |
| m_{31} | m_{32} | m_{33} |

```
xt = m11*x + m21*y + m31  
yt = m22*y + m12*x + m32  
  
if projected:  
    wt = m13*x + m23*y + m33  
    xt /= wt  
    yt /= wt
```

- › m_{13} and m_{23}
 - › Control perspective transformations

Common Transformations

- › Commonly-used convenience functions:
 - › `scale()`
 - › `rotate()`
 - › `shear()`
 - › `translate()`
- › Saves you the trouble of defining transformation matrices
- › `rotate()` takes optional 2nd argument: axis of rotation.
 - › Z axis is "simple 2D rotation"
 - › Non-Z axis rotations are "perspective" projections.



View Transformations

- › `setTransformationAnchor()`
 - › An **anchor** is a point that remains fixed before/after the transform.
 - › `AnchorViewCenter`: (Default) The *center point* remains the same
 - › `AnchorUnderMouse`: The *point under the mouse* remains the same
 - › `NoAnchor`: Scrollbars remain unchanged.

```
t = QTransform(); // identity matrix
t.rotate(45, Qt::ZAxis); // simple rotate
t.scale(1.5, 1.5) // scale by 150%
view->setTransform(t); // apply transform to entire view
```

Item Transformations

- › `QGraphicsItem` supports same transform operations:
 - › `setTransform()`, `transform()`
 - › `rotate()`, `scale()`, `shear()`, `translate()`
- › **An item's effective transformation:**
 - › The product of its own and all its ancestors' transformations

TIP: When managing the transformation of items, store the desired rotation, scaling etc. in member variables and build a `QTransform` from the identity transformation when they change. Don't try to deduce values from the current transformation and/or try to use it as the base for further changes.

Zooming

- › Zooming is done with `view->scale()`

```
void MyView::zoom(double factor)
{
    double width = matrix().mapRect(QRectF(0, 0, 1, 1)).width();
    width *= factor;
    if ((width < 0.05) || (width > 10))
        return;
    scale(factor, factor);
}
```

Mapping between Coordinate Systems

- › Mapping methods are overloaded for `QPolygonF`, `QPainterPath` etc.

- › `mapFromScene(const QPointF&)`:

- › Maps a point from scene coordinates to item coordinates. Inverse: `mapToScene(const QPointF&)`

- › `mapFromItem(const QGraphicsItem*, const QPointF&)`

- › Maps a point from another item's coordinate system to this item's .Inverse: `mapToItem(const QGraphicsItem*, const QPointF&)` .

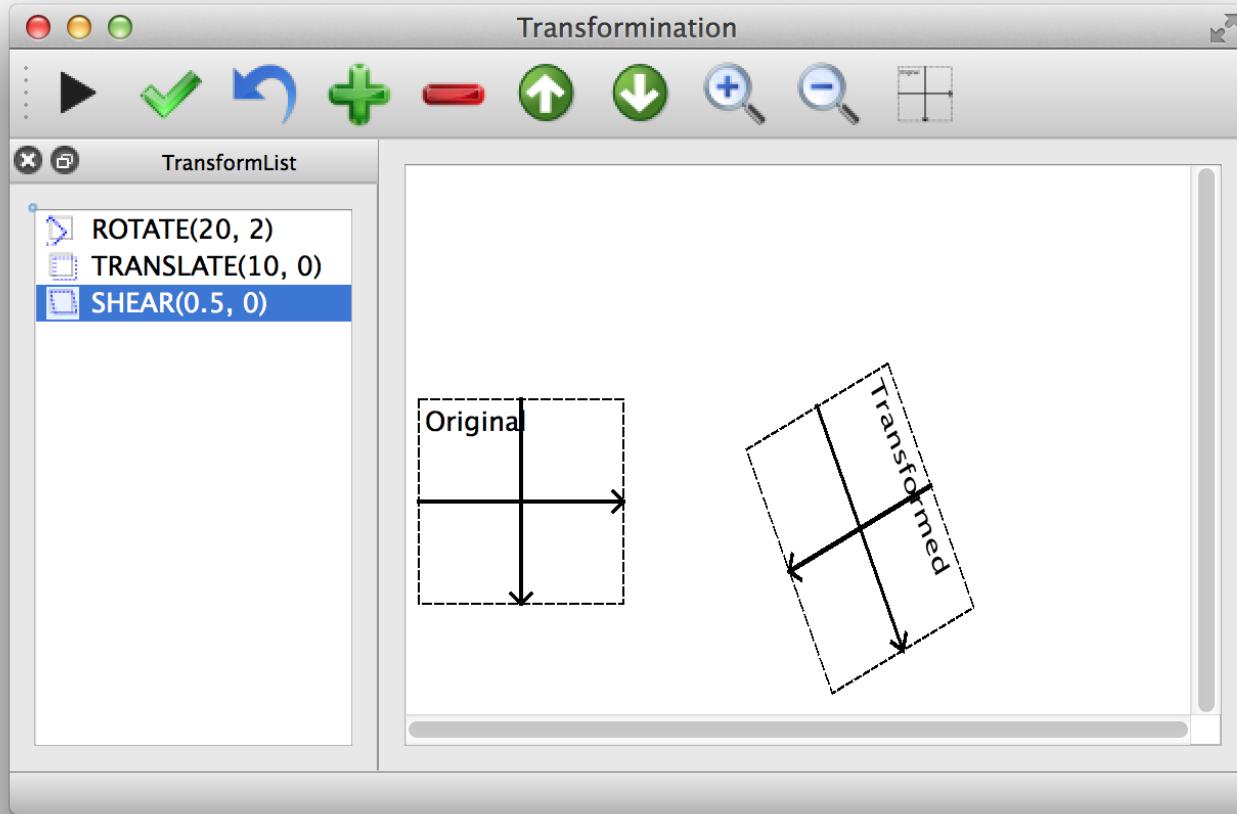
- › Special case: `mapFromParent(const QPointF&)` .

Ignoring Transformations

- › Sometimes we don't want particular items to be transformed before display.
- › View transformation can be disabled for individual items.
- › Used for text labels in a graph that should not change size when the graph is zoomed.

```
item->setFlag( QGraphicsItem::ItemIgnoresTransformations );
```

Transforms Demo

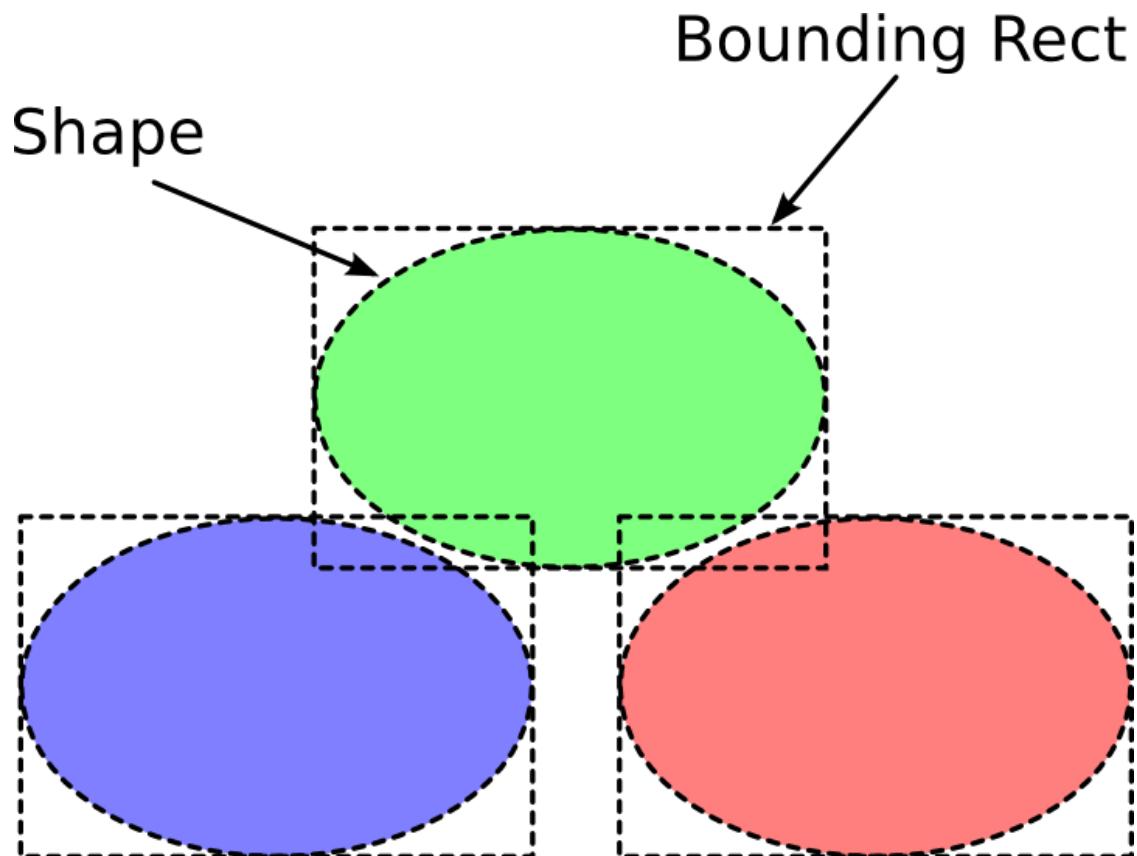


Extending QGraphicsItem

`QGraphicsItem` pure virtual methods (required overrides):

- › `void paint()`
 - › Paints contents of item in local coordinates
- › `QRectF boundingRect()`
 - › Returns outer bounds of item as a rectangle
 - › Called by `QGraphicsView` to determine what regions need to be re-drawn
- › `QPainterPath shape() – shape of item`
 - › Used by `contains()` and `collidesWithPath()` for collision detection
 - › Defaults to `boundingRect()` if not implemented

Boundaries

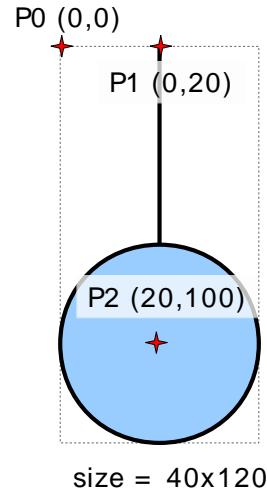
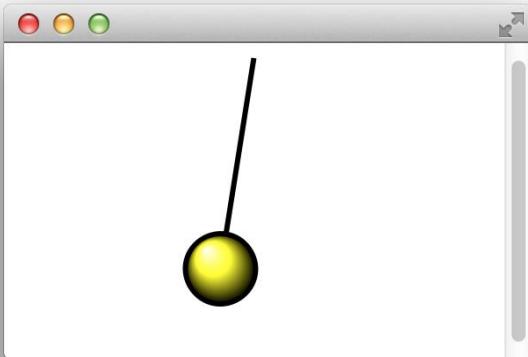


Painting Items

- › Item is in complete control of drawing itself
- › Use standard `QPainter` drawing methods
 - › `QPen`, `QBrush`, pixmaps, gradients, text, etc.
- › No background to draw
- › Dynamic boundary and arbitrary shape
 - › Polygon, curved, non-contiguous, etc.

Custom Item example

```
class PendulumItem : public QGraphicsItem {  
public:  
    QRectF boundingRect() const;  
    void paint(QPainter* painter, const QStyleOptionGraphicsItem* option,  
              QWidget* widget);  
};
```



paint() and boundingRect()

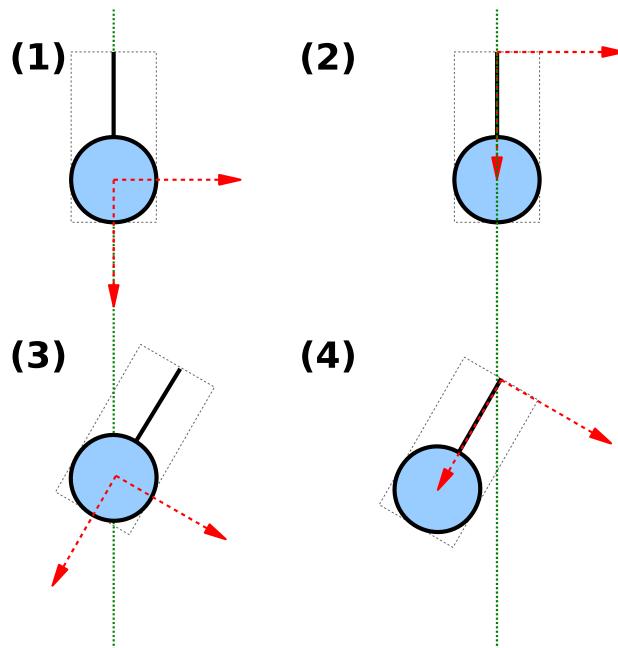
- › boundingRect() must take the pen width into consideration

```
QRectF PendulumItem::boundingRect() const {
    return QRectF(-20.0 - PENWIDTH/2.0, -PENWIDTH/2.0,
                  40.0 + PENWIDTH, 140.0 + PENWIDTH );
}

void PendulumItem::paint( QPainter* painter,
    const QStyleOptionGraphicsItem*, QWidget*) {
    painter->setPen( QPen( Qt::black, PENWIDTH ) );
    painter->drawLine(0,0,0,100);
    QRadialGradient g( 0, 120, 20, -10, 110 );
    g.setColorAt( 0.0, Qt::white );
    g.setColorAt( 0.5, Qt::yellow );
    g.setColorAt( 1.0, Qt::black );
    painter->setBrush(g);
    painter->drawEllipse(-20, 100, 40, 40);
}
```

Choosing a boundingRect()

- › `boundingRect()`
 - › Influences drawing code
 - › Influences "origin" of item transforms
- › i.e. for Pendulum that swings:
 - › Good origin is non-weighted end of line
 - › Can rotate around(0,0) without translation



QGraphicsItemGroup

- › Easier approach to making a Pendulum:
 - › Extend QGraphicsItemGroup
 - › Use other concrete items as elements, add as children
 - › No need to override `paint()` or `shape()`

```
PendulumItem::PendulumItem(QGraphicsItem* parent) :  
    QGraphicsItemGroup(parent) {  
    m_line = new QGraphicsLineItem( 0,0,0,100, this);  
    m_line->setPen( QPen( Qt::black, 3 ) );  
    m_circle = new QGraphicsEllipseItem( -20, 100, 40, 40, this );  
    m_circle->setPen( QPen(Qt::black, 3 ));  
    QRadialGradient g( 0, 120, 20, -10, 110 );  
    g.setColorAt( 0.0, Qt::white );  
    g.setColorAt( 0.5, Qt::yellow );  
    g.setColorAt( 1.0, Qt::black );  
    m_circle->setBrush(g);  
}
```

Event Handling

- › `QGraphicsItem::sceneEvent (QEvent*)`
 - › Receives all events for an item
 - › Similar to `QWidget::event ()`
- › **Specific typed event handlers:**
 - › `keyPressEvent (QKeyEvent*)`
 - › `mouseMoveEvent (QGraphicsSceneMouseEvent*)`
 - › `wheelEvent (QGraphicsSceneWheelEvent*)`
 - › `mousePressEvent (QGraphicsSceneMouseEvent*)`
 - › `contextMenuEvent (QGraphicsSceneContextMenuEvent*)`
 - › `dragEnterEvent (QGraphicsSceneDragDropEvent*)`
 - › `focusInEvent (QFocusEvent*)`
 - › `hoverEnterEvent (QGraphicsSceneHoverEvent*)`
- › **When overriding mouse event handlers:**
 - › Make sure to call base-class versions, too. Without this, the item select, focus, move behavior will not work as expected.

Event Handler Examples

```
void MyView::wheelEvent(QWheelEvent *event)
{
    double factor = 1.0 + (0.2 * qAbs(event->delta()) / 120.0);
    if (event->delta() > 0) zoom(factor);
    else zoom(1.0/factor);
}

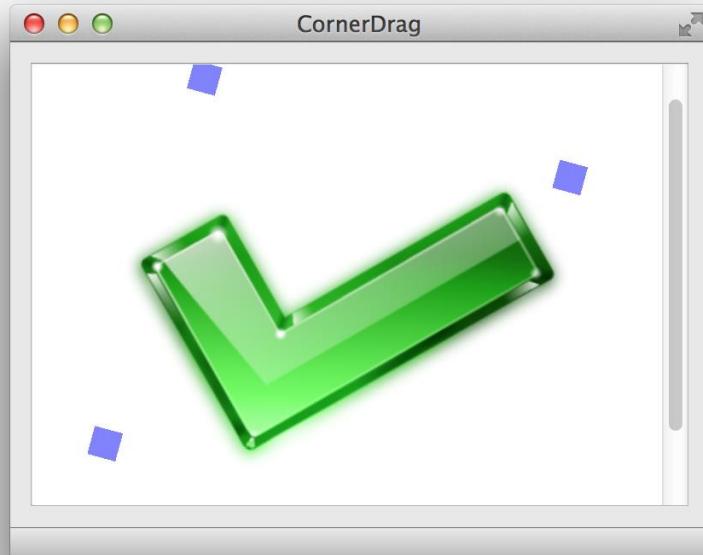
void MyView::keyPressEvent(QKeyEvent *event)
{
    switch (event->key()) {
        case Qt::Key_Plus:
            zoom(1.2);
            break;
        case Qt::Key_Minus:
            zoom(1.0/1.2);
            break;
        default:
            QGraphicsView::keyPressEvent(event);
    }
}
```

Collision Detection

- › Determines when items' shapes intersect
- › Two methods for collision detection:
 - › `collidesWithItem(QGraphicsItem* other)`
 - › `collidingItems(Qt::SelectionMode)`
- › `shape()`
 - › Returns `QPainterPath` used for collision detection
 - › Must be overridden properly
- › `items()`
 - › Overloaded forms take `QRectF`, `QPolygonF`, `QPainterPath`
 - › Return items found in rect/polygon/shape

Lab – *Corner Drag Button*

- › Define a `QGraphicsItem` which can display an image, and has at least 1 child item, that is a "corner drag" button, permitting the user to click and drag the button, to resize or rotate the image.
- › Start with the handout, provided in `lab-corner-drag`
- › Further details are in the `readme.txt` in the same directory



Widgets in a Scene

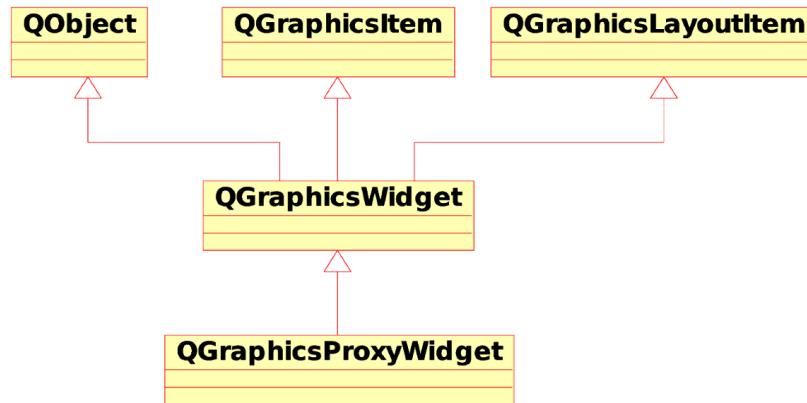


Items Are not Widgets

- › `QGraphicsItem`:
 - › Lightweight compared to `QWidget`
 - › No signals/slots/properties
 - › Scenes can easily contain thousands of Items
 - › Uses different `QEvent` sub-hierarchy (derived from `QGraphicsSceneEvent`)
 - › Supports transformations directly
- › `QWidget`:
 - › Derived from `QObject` (less light-weight)
 - › Supports signals, slots, properties, etc.
 - › Can be embedded in a `QGraphicsScene` with a `QGraphicsProxyWidget`

QGraphicsWidget

- › Advanced functionality graphics item
- › Provides signals/slots, layouts, geometry, palette, etc.
- › *Not a QWidget*
- › Base class for QGraphicsProxyWidget



QGraphicsProxyWidget

- › `QGraphicsItem` that can embed a `QWidget` in a `QGraphicsScene`
- › Handles complex widgets like `QFileDialog`
- › Takes *ownership* of related widget
 - › Synchronizes states/properties:
 - › visible, enabled, geometry, style, palette, font, cursor, sizeHint, windowTitle, etc.
 - › Proxies events between Widget and GraphicsView
 - › If either (widget or proxy) is deleted, the other is also!
- › Widget must not already have a parent
 - › Only top-level widgets can be added to a scene

Embedded Widget Example

```
#include <QtWidgets>

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

    QCalendarWidget *calendar = new QCalendarWidget;

    QGraphicsScene scene;
    QGraphicsProxyWidget *proxy = scene.addWidget(calendar);

    QGraphicsView view(&scene);
    view.show();

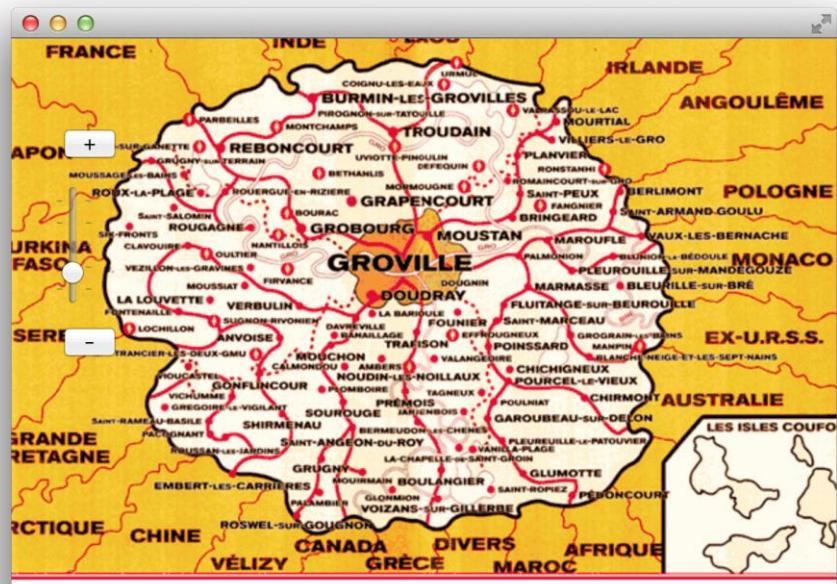
    return app.exec();
}
```

QGraphicsLayout

- › For layout of `QGraphicsLayoutItem` (+derived) classes in `QGraphicsView`
- › Concrete classes:
 - › `QGraphicsLinearLayout`: equivalent to `QBoxLayout`, arranges items horizontally or vertically
 - › `QGraphicsGridLayout`: equivalent to `QGridLayout`, arranges items in a grid
- › `QGraphicsWidget::setLayout()` - set layout for child items of this `QGraphicsWidget`

Lab: Widgets in a Scene

- › Starting with the lab-map-viewer handout, add zooming controls.
- › Suggested widgets:
 - › QPushButton for +/-
 - › QSlider for selecting zoom level directly.
- › Use QGraphicsLayout to layout the widgets
- › Make the mouse work like a "hand-grab" tool on drag, so we can see different zoomed areas.



Drag and Drop

- › Items can be:
 - › Dragged
 - › Dropped onto other items
 - › Dropped onto scenes
 - › For handling empty drop areas

Start Drag

- › Starting an item drag is similar to dragging from a QWidget.
- › Override event handlers:
 - › mousePressEvent ()
 - › mouseMoveEvent ()
- › In mouseMoveEvent () , decide if drag started? If so:
 - › Create a QDrag instance
 - › Attach a QMimeData to it
 - › See section on Drag and Drop for QMimeData info
 - › Call QDrag::exec ()
 - › Function returns when user drops
 - › Does not block event loop

Drop on a Scene

- › Override `QGraphicsScene::dropEvent()`
 - › To accept drop:
 - › `acceptProposedAction()`
 - › `setDropAction(Qt::DropAction); accept();`
- › Override `QGraphicsScene::dragMoveEvent()`
- › Optional overrides:
 - › `dragEnterEvent()`, `dragLeaveEvent()`

startDrag() Example

```
void startDrag( Qt::DropActions supportedActions ) {
    if ( selectedItems().size() > 0 ) {
        QListWidgetItem *item = selectedItems()[0];
        QDrag* drag = new QDrag( this );
        QMimeData *mimeData = new QMimeData; [...]
        QGraphicsItem* gitem = DiagramItem::createItem(
            item->toolType() );
        mimeData->setData( "application/x-qgraphicsitem-ptr",
                            QByteArray::number( ( qulonglong )gitem ) );
        drag->setMimeData( mimeData );
        QPixmap pix = item->icon().pixmap( 111, 111 );
        drag->setPixmap( pix );
        drag->setHotSpot( pix.rect().center() );
        if ( drag->exec(supportedActions) == Qt::IgnoreAction ) {
            delete gitem; // drag cancelled, must delete item
    }
}
```

dropEvent() on a Scene

```
void DiagramScene::dropEvent( QGraphicsSceneDragDropEvent* event) {  
    if (event->mimeData()->hasFormat("application/x-qgraphicsitem-ptr")) {  
        QGraphicsItem* item = reinterpret_cast<QGraphicsItem*>(  
            event->mimeData()->data(   
                "application/x-qgraphicsitem-ptr").toULongLong() );  
        if ( item ) {  
            addItem( item );  
            item->setFlag( QGraphicsItem::ItemIsMovable );  
            item->setFlag( QGraphicsItem::ItemIsSelectable );  
            item->setFlag( QGraphicsItem::ItemIsFocusable );  
            item->setPos( event->scenePos() );  
            event->acceptProposedAction();  
        }  
    }  
    else  
        /* Call baseclass to allow per-item dropEvent */  
        QGraphicsScene::dropEvent( event );  
}
```

Drop on an Item

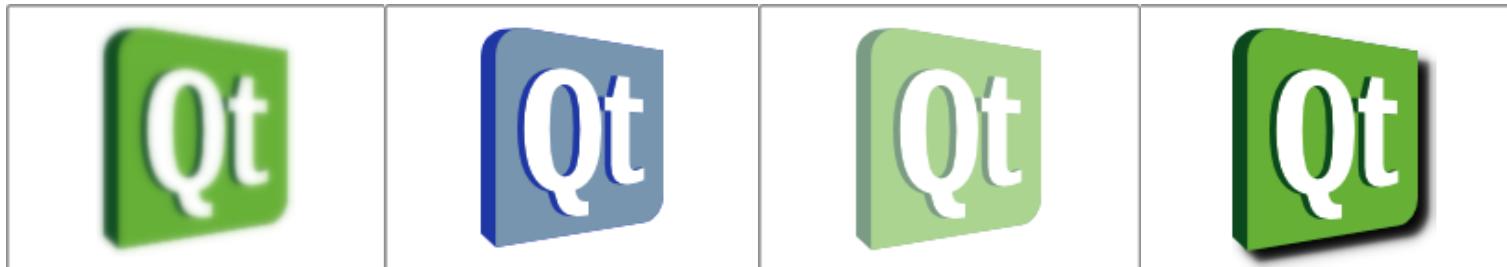
- › To drop into an item:
 - › Override `dragEnterEvent()`
 - › Optional override: `dragMoveEvent()` (if the item can only accept drops in some parts of its area)

```
void DiagramItem::dragEnterEvent(QGraphicsSceneDragDropEvent* e) {  
    if (e->mimeData()->hasColor())  
        e->acceptProposedAction();  
}  
  
void DiagramScene::dragEnterEvent(QGraphicsSceneDragDropEvent* e) {  
    if (e->mimeData()->hasFormat(  
        "application/x-qgraphicsitem-ptr"))  
        e->acceptProposedAction();  
    else  
        QGraphicsScene::dragEnterEvent(e);  
}
```

Graphics Effects

Effects can be applied to graphics items:

- › Base class for effects is `QGraphicsEffect`
- › Standard effects include blur, colorize, opacity and drop shadow.
- › Effects are set on items.
 - › `QGraphicsItem::setGraphicsEffect()`
- › Effects cannot be shared or layered.
- › Custom effects can be written.



Using a Graphics Effect

- › Applying a blur effect to a pixmap.

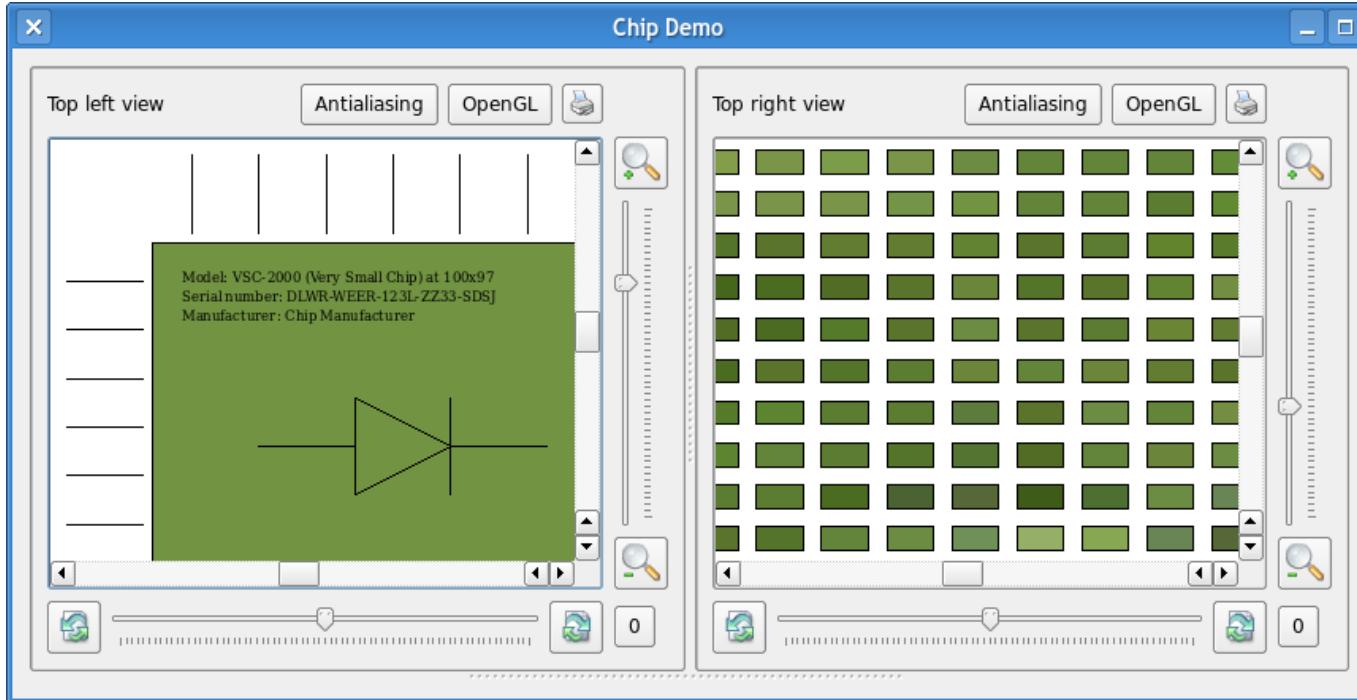
```
QPixmap pixmap(":/images/qt-banner.png");
QGraphicsItem *blurItem = scene->addPixmap(pixmap);
QGraphicsBlurEffect *blurEffect = new QGraphicsBlurEffect();
blurItem->setGraphicsEffect(blurEffect);
blurEffect->setBlurRadius(5);
```

- › An effect is owned by the item that uses it.
- › Updating an effect causes the item to be updated.

Level of Detail

- › Don't draw what you can't see
- › `QStyleOptionGraphicsItem` passed to `paint()`
 - › Contains `palette`, `state`, `matrix` members
 - › `qreal levelOfDetailFromTransform(QTransform T)` method
- › "levelOfDetail" is max width/height of the unity rectangle needed to draw this shape onto a `QPainter` with a `QTransform` of `T`.
- › Use `worldTransform()` of `painter` for current transform.
 - › Zoomed out: `levelOfDetail < 1.0`
 - › Zoomed in: `levelOfDetail > 1.0`

Level of Detail: Chip Demo



Level of Detail: Chip Demo 2

```
void Chip::paint(QPainter *painter,
                  const QStyleOptionGraphicsItem *option, QWidget *)
{
    const qreal lod = option->levelOfDetailFromTransform(
        painter->worldTransform());
    [...]
    if (lod >= 2) {
        QFont font("Times", 10);
        font.setStyleStrategy(QFont::ForceOutline);
        painter->save();
        painter->setFont(font);
        painter->scale(0.1, 0.1);
        painter->drawText(170, 180, QString("Model: VSC-2000 ..."));
        painter->drawText(170, 220, QString("Manufacturer: ..."));
        painter->restore();
    }
}
```

Caching Tips

- › Cache item painting into a pixmap
- › So `paint()` runs faster
- › Cache `boundingRect()` and `shape()`
 - › Avoid re-computing expensive operations that stay the same
 - › Be sure to invalidate manually cached items after zooming and other transforms

```
QRectF MyItem::boundingRect() const {
    if (m_rect.isNull())
        calculateBoundingRect();
    return m_rect;
}

QPainterPath MyItem::shape() const {
    if (m_shape.isEmpty())
        calculateShape();
    return m_shape;
}
```

setCacheMode()

- › Property of `QGraphicsView` and `QGraphicsItem`
- › Allows caching of pre-rendered content in a `QPixmap`
 - › Drawn on the viewport
 - › Especially useful for gradient shape backgrounds
 - › Invalidated whenever view is transformed.

```
QGraphicsView view;  
view.setBackgroundBrush(QImage(":/images/backgroundtile.png"));  
view.setCacheMode(QGraphicsView::CacheBackground);
```

Tweaking

- › The following methods allow you to tweak performance of view/scene/items:
 - › `QGraphicsView::setViewportUpdateMode()`
 - › `QGraphicsView::setOptimizationFlags()`
 - › `QGraphicsScene::setItemIndexMethod()`
 - › `QGraphicsScene::setBspTreeDepth()`
 - › `QGraphicsItem::setFlags()`
 - › `ItemDoesntPropagateOpacityToChildren` and `ItemIgnoresParentOpacity` especially recommended if your items are opaque!

Tips for Better Performance

- › `boundingRect()` and `shape()` are called frequently so they should run fast!
 - › `boundingRect()` should be as small as possible
 - › `shape()` should return simplest reasonable path
- › Try to avoid drawing gradients on the painter
 - › Consider using pre-rendered backgrounds from images instead.
- › It is costly to dynamically insert/remove items from the scene
 - › Consider hiding and reusing items instead.
- › Embedded widgets in a scene is costly.
- › Try using a different paint engine (OpenGL, Direct3D, etc.)
 - › `setViewport (new QGLWidget);`
- › Avoid curved and dashed lines
- › Alpha blending and antialiasing are expensive

Questions and Answers

- › How many scenes can be attached to a view? How many views can render a single scene?
- › How events are propagated to graphic items?
- › How do you use property animations with graphic items?"
- › How transformations are applied to a graphics view or a graphics item?
- › When is it beneficial to use binary space partitioning for accessing scene index?
- › What is level of detail and how is it used?
- › Which classes does `QGraphicsViewItem` derive from?
- › What should be taken, when embedding widgets into a scene?
- › How to use OpenGL for painting graphic items, painted with `QPainter`?
- › How graphics view performance can be adjusted?

Summary

- › Graphics view provides a light-weight framework for rendering possibly a large number of graphical items
- › Items are managed by a scene, which can be shared by several graphics view widgets
- › The scene forward events from the view widget to very light-weight graphic items
- › In addition to event handling, items can be transformed and grouped to make their management easier
- › Standard widgets may be embedded into the graphics view, but has a performance penalty
- › Several options exist to adjust graphics performance
 - › BSP index use, widget background caching, item caching, level of detail information

Contents

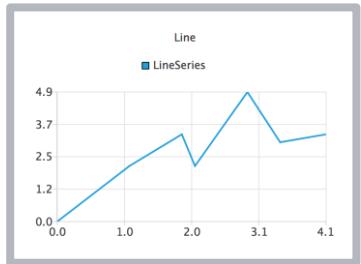
- › Chart Types
- › Using Qt Model Data in Charts
- › Dynamic Charts
- › User Interactions
- › Themes and Customization

Objectives

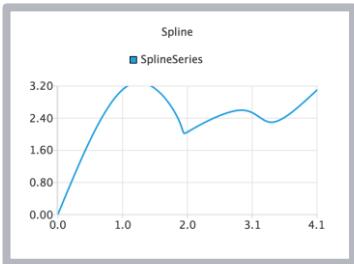
Learn...

- › ...Qt chart types
- › ...how to add and modify chart data
- › ...how to use item models for providing chart data
- › ...how to customize charts

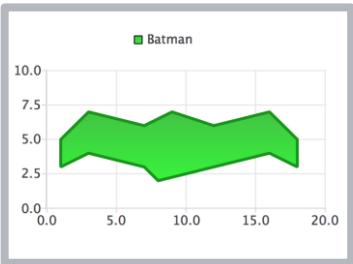
Qt Charts – Chart Types



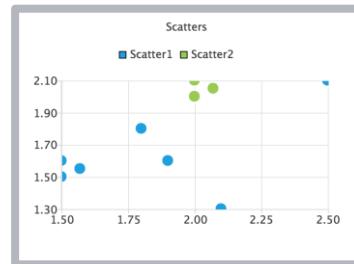
Line chart



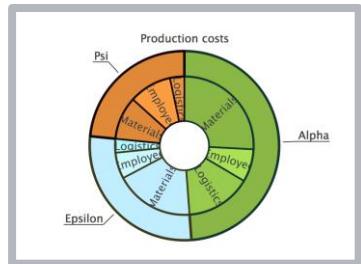
Spline chart



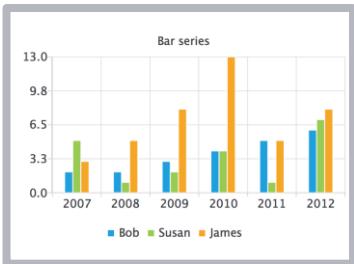
Area chart



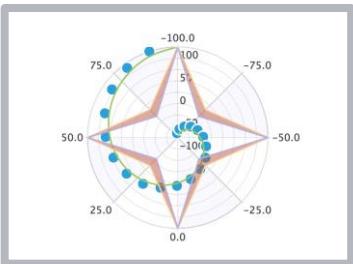
Scatter chart



Pie chart
(pie, donut)



Bar chart
(bar, stacked,
percent)



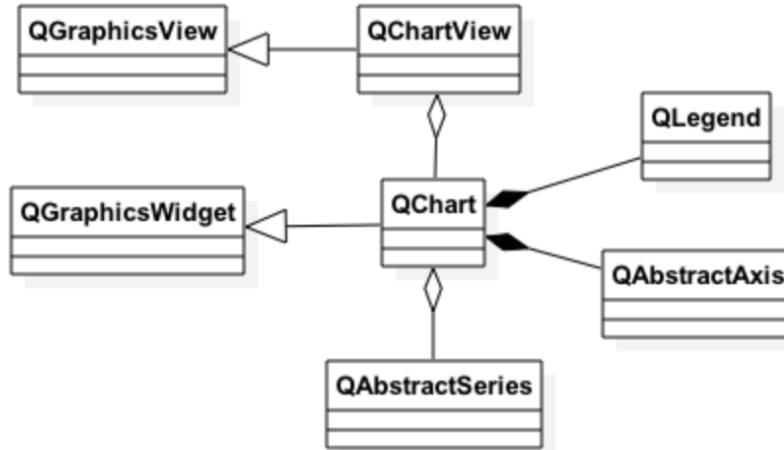
Polar chart



Candlestick
chart

Charts Architecture

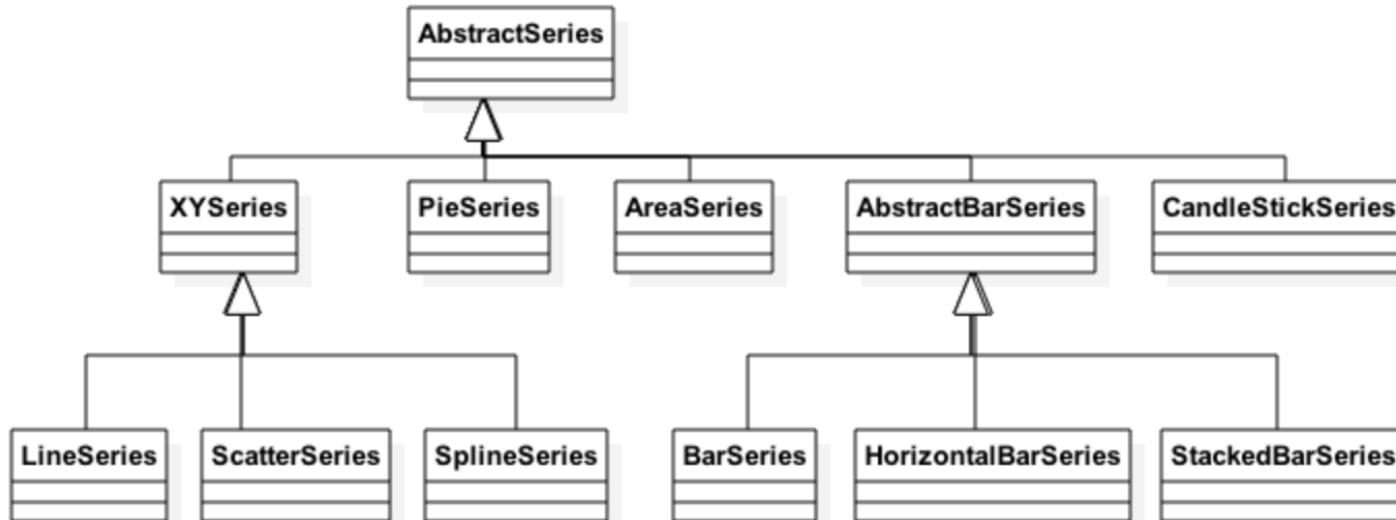
- › Charts API is based on the Qt Graphics View framework
- › Charts can be displayed as graphics widgets using `QChart`
 - › Provides API for zooming and scrolling
- › `QChartView` allows using the chart as a widget
- › Each chart type is represented by `QAbstractSeries` derived class



Series

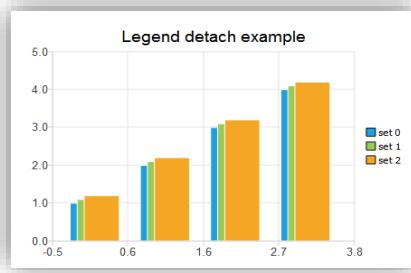
- › Determines the chart type
- › To use a certain chart type, create an instance of a series class and add it to the `QChart` instance
- › Series can be dynamically removed and added

```
QLineSeries* series = new QLineSeries();  
series->add(0, 6);  
series->add(2, 4);  
// ...  
chartView->chart()->addSeries(series);  
chartView->chart()->createDefaultAxes();
```

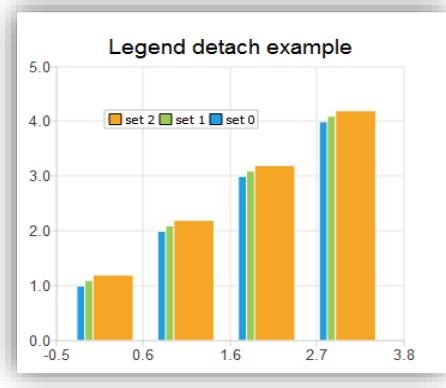


Legend

- > Shows what the chart data represents
 - > Fully automatic
- > Default legend placement options:
 - > Top, Bottom, Left or Right
- > Simple API to change
 - > Legend background pen and brush
 - > Label brush
- > Or draw yourself
 - > Set new parent to legend
 - > Chart updates legend state with signals, but won't draw it.



Legend aligned to right



Legend Detached and drawn on top of chart

Axis

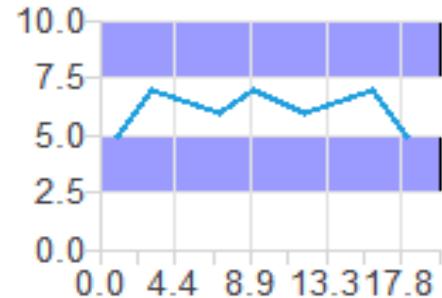
- › Controls the presentation of series
- › Only one x-axis may be visible at the time
 - › If there are several y-axes, they share the common x-axes
- › Several axis type exist (enumeration in `QAbstractAxis::AxisType`)
 - › No axis, value axis, category and bar category axis, date and time axis

| Series type | X-axis | Y-axis |
|-------------|------------------|------------|
| QXYSeries | QValueAxes | QValueAxis |
| QBarSeries | QBarCategoryAxis | QValueAxis |
| QPieSeries | None | None |

Axis

- › Can be setup to show tick marks, grid lines, and shades
- › Default axes can be created
 - › `QChart::createDefaultAxes()`
 - › Must be called after the series has been added to the chart

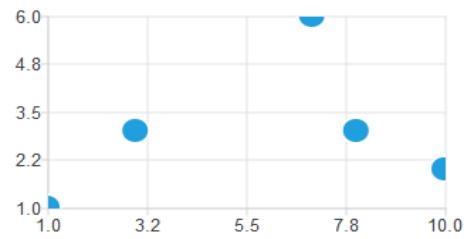
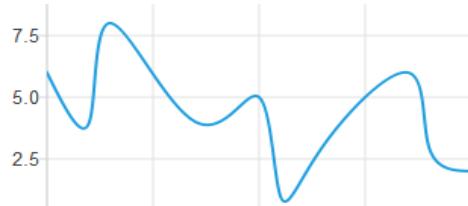
```
chart.axisX()->setRange(0, 20);
chart.axisY()->setRange(0, 10);
chart.axisY()->setShadesVisible(true);
chart.axisY()->setShadesColor(QColor(0, 0, 255, 100));
static_cast<QValueAxis *>(chart.axisX())->setTickCount(10);
```



Line Charts

- › Spline is exactly similar to line chart except an spline is drawn between the points in QSplineSeries
- › Scatter chart is created similarly, but QScatterSeries is used
 - › Additionally, marker properties can be set

```
QScatterSeries *series1 = new QScatterSeries();
series1->setName("scatter2");
series1->setMarkerShape(QScatterSeries::MarkerShapeCircle);
series1->setMarkerSize(20.0);
*series1 << QPointF(1, 1) << QPointF(3, 3) << QPointF(7, 6)
    << QPointF(8, 3) << QPointF(10, 2);
```



- › Two line series can be added to an area series
 - › QAreaSeries(lineSeries1, lineSeries2);
 - › Set the pen and brush to QAreaSeries to paint the area



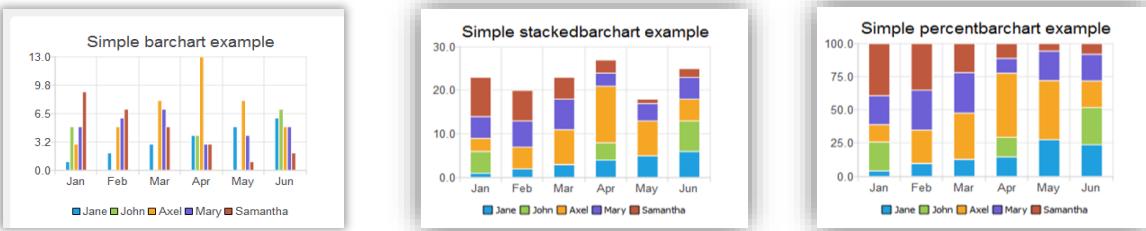
Bar Charts

> Bar chart types

- > **Bar** draws bars as groups, where bars in same category are grouped next to each other.
- > **Stacked bar** bar draws the bars in same category stacked on top of each other
- > **Percent Bar** draws bars as stacks, where each bar is shown as percentage of all bars in that category

> To create horizontal bar charts

- > Use `QHorizontal*BarSeries` classes



```
// Create a Bar Set
QBarSet *set0 = new QBarSet("Jane");
// Insert data
*set0 << 1 << 2 << 3 << 4 << 5 << 6;

// Create Bar Series and add Bar Set
QBarSeries* series = new QBarSeries();
// Create Chart and add Series
QChart* chart = new QChart();
chart->addSeries(series);
chart->setTitle("Simple grouped barchart example");
```

Bar Charts – Categories

- › To display categories on axis
 - › Create QBarCategoryAxis
 - › Append one or more categories as QString or QStringList
 - › Set the category to an axis
- › If you use QChart::createDefaultAxes()
 - › Set the axis to the chart after calling this function
 - › Otherwise, the default axis will override the category axis

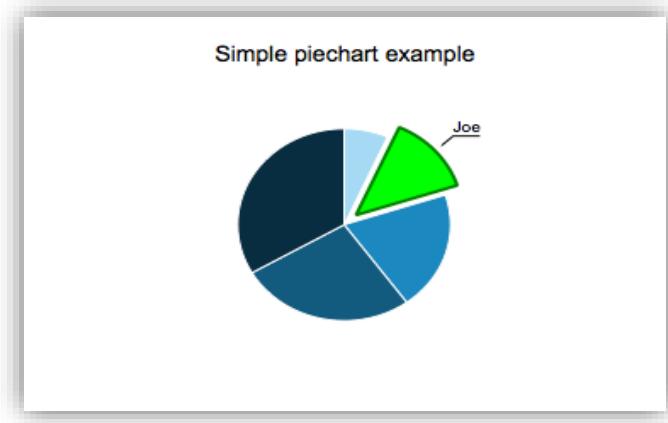
```
QStringList categories;
categories << "Jan" << "Feb" << "Mar" << "Apr" << "May" << "Jun";

QBarCategoryAxis* axis = new QBarCategoryAxis();
axis->append(categories);

chart->createDefaultAxes();
chart->setAxisX(axis, series);
```

Pie Charts

- › A simple API with two main classes
 - › QPieSeries
 - › QPieSlice
- › Appearance can be fully customized
 - › Pie size and position
 - › Donut like
 - › Slice pen, brush, label font, etc.
- › Signals for user interaction
 - › Hover, clicked
- › Basic animations support
 - › Adding, removing and exploding slices



Pie Chart Example

```
QPieSeries *series = new QPieSeries();
series->append("Jane", 1);
series->append("Joe", 2);
series->append("Andy", 3);
series->append("Barbara", 4);
series->append("Axel", 5);

QPieSlice *slice = series->slices().at(1);
slice->setExploded();
slice->setLabelVisible();
slice->setPen(QPen(Qt::darkGreen, 2));
slice->setBrush(Qt::green);
```

Pie Charts – Drilldown

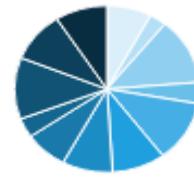
- › Implement, e.g., with `QPieSlice` signals
- › When the slice is clicked
 - › Change the series to show another pie chart
- › Note that `QChart::removeSeries()` deletes the series object
 - › Store it in some other object, unless you want to re-create the series data

Sales by year - All



| |
|---------------------|
| Jane \$7712, 21.0% |
| John \$5042, 13.7% |
| Axel \$7800, 21.2% |
| Mary \$5361, 14.6% |
| Susan \$5018, 13.6% |
| Bob \$5830, 15.9% |

Sales by month - Jane



| |
|------------------|
| Jan \$603, 7.8% |
| Feb \$226, 2.9% |
| Mar \$999, 13.0% |
| Apr \$317, 4.1% |
| May \$919, 11.9% |
| Jun \$719, 9.3% |
| Jul \$680, 8.8% |
| Aug \$581, 7.5% |
| Sep \$290, 3.8% |
| Oct \$921, 11.9% |
| Nov \$755, 9.8% |
| Dec \$702, 9.1% |

Pie Charts – Drilldown

```
QPieSeries* yearSeries = new QPieSeries(&window);
yearSeries->setName("Sales by year - All");

QList<QString> months; months << "Jan" << "Feb" << "Mar" << "Apr"
    << "May" << "Jun" << "Jul" << "Aug" << "Sep" << "Oct" << "Nov" << "Dec";
QList<QString> names; names << "Jane" << "John" << "Axel" << "Mary" << "Susan" << "Bob";

for (const QString &name, names) {
    QPieSeries *series = new QPieSeries(&window);
    series->setName("Sales by month - " + name);

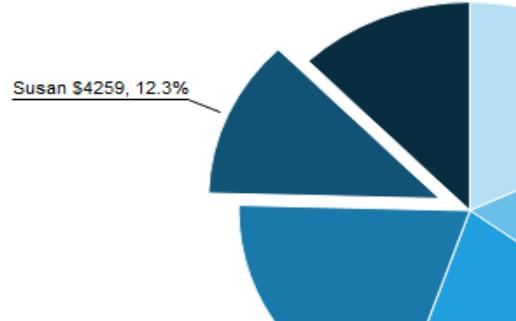
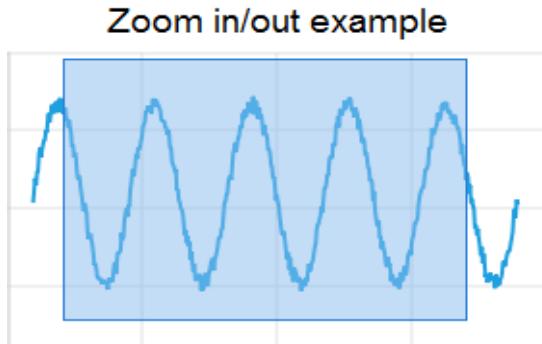
    for (const QString &month, months)
        // DrilldownSlice is QPieSlice sub-class, containing the series and
        // functionality to update the percentage label and expand the slice
        *series << new DrilldownSlice(qrand() % 1000, month, yearSeries);

    QObject::connect(series, &QPieSeries::clicked, chart, &Chart::handleSliceClicked);
    *yearSeries << new DrilldownSlice(series->sum(), name, series);
}

QObject::connect(yearSeries, &QPieSeries::clicked, chart, &Chart::handleSliceClicked);
chart->changeSeries(yearSeries);
```

User Interaction

- › Hover and clicked signals provide a good start for user interaction
 - › Drill-down for instance
- › Zooming in/out and scrolling also supported

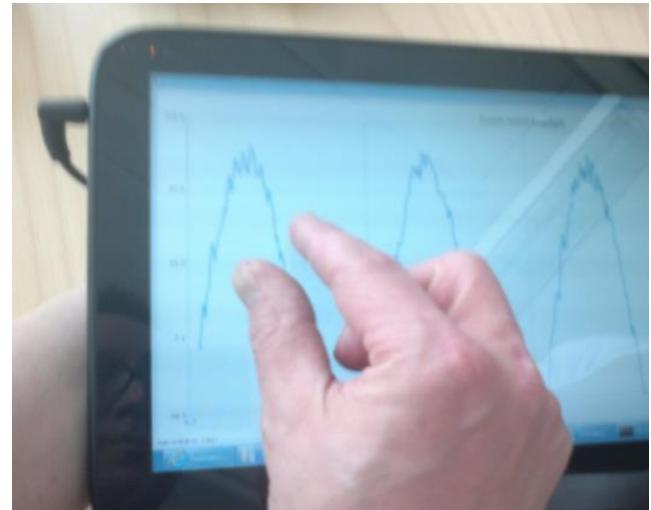


Zooming and Scrolling

- › `QChart::zoom(qreal factor)`
- › `QChart::zoomIn()`
- › `QChart::zoomIn(const QRectF& rect)`

- › `void QChart::scroll (qreal dx, qreal dy)`

- › Easy to integrate to Qt Gesture FW



Gesture Example (Zoom Line SDK Example)

```
bool Chart::gestureEvent(QGestureEvent *event)
{
    if (QGesture *gesture = event->gesture(Qt::PanGesture)) {
        QPanGesture *pan = static_cast<QPanGesture *>(gesture);
        QChart::scroll(-(pan->delta().x()), pan->delta().y());
    }
    if (QGesture *gesture = event->gesture(Qt::PinchGesture)) {
        QPinchGesture *pinch = static_cast<QPinchGesture *>(gesture);
        if (pinch->changeFlags() & QPinchGesture::ScaleFactorChanged)
            QChart::zoom(pinch->scaleFactor());
    }
    return true;
}
```

Using Model Data in Charts

- › `QAbstractItemModel` can be used as a data source for the series:
 - › Vertical and horizontal orientation
 - › Data area (i.e. how data is presented in a chart) is done with series specific model mapper classes
 - › Charts series are updated when changes are made to the model (data updated, rows or columns inserted or removed)
 - › Obviously, the model must support data insert and removal as well
- › This is especially handy when
 - › integrating user interfaces with table and chart presentation
 - › creating interactive charts with data entry forms

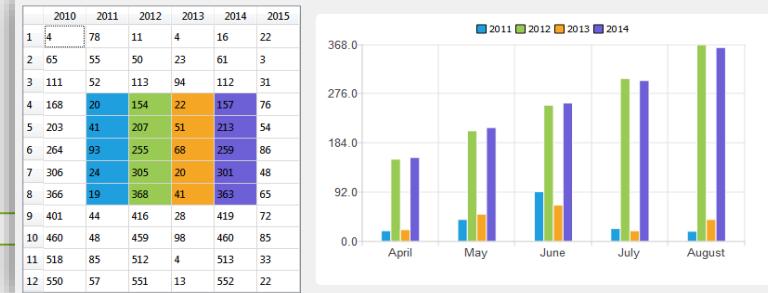
Model Mappers

- › Keep your series and the actual model in sync
- › Both APIs may be used at the same time:
 - › Series model API or `QAbstractItemModel` API
- › No common base class for model mappers
- › `QHXYModelMapper`/`QVXYModelMapper`
 - › Define which model rows/columns mapped (and kept sync) to the points in the XY series
 - › Define the first column/row and the number of columns/rows, if only part of the model data used
- › `QHBarModelMapper`/`QVBarModelMapper`
 - › Similarly to XY model mappers, define which model rows/columns mapped to bar sets
- › `QHPieModelMapper`/`QVPieModelMapper`
 - › Define a row/column, which contains values and another row/column, which contains labels

Model Mappers

```
// The model is a table with two columns
QVXYModelMapper *mapper = new QVXYModelMapper(this);
mapper->setXColumn(0);
mapper->setYColumn(1);
mapper->setSeries(series);
mapper->setModel(model);
chart->addSeries(series);
```

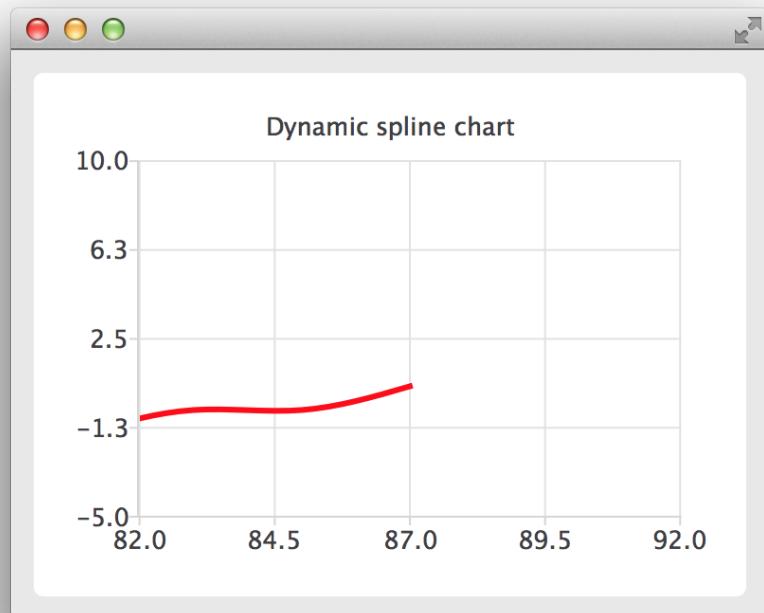
| | 2010 | 2011 | 2012 | 2013 | 2014 | 2015 |
|----|------|------|------|------|------|------|
| 1 | 4 | 78 | 11 | 4 | 16 | 22 |
| 2 | 65 | 55 | 50 | 23 | 61 | 3 |
| 3 | 111 | 52 | 113 | 94 | 112 | 31 |
| 4 | 168 | 20 | 154 | 22 | 157 | 76 |
| 5 | 203 | 41 | 207 | 51 | 213 | 54 |
| 6 | 264 | 93 | 255 | 68 | 259 | 86 |
| 7 | 306 | 24 | 305 | 20 | 301 | 48 |
| 8 | 366 | 19 | 368 | 41 | 363 | 65 |
| 9 | 401 | 44 | 416 | 28 | 419 | 72 |
| 10 | 460 | 48 | 459 | 98 | 460 | 85 |
| 11 | 518 | 85 | 512 | 4 | 513 | 33 |
| 12 | 550 | 57 | 551 | 13 | 552 | 22 |



```
// The model is a table with six columns
int first = 3; int count = 5;
QVBarModelMapper *mapper = new QVBarModelMapper(this);
mapper->setFirstBarSetColumn(1);
mapper->setLastBarSetColumn(4);
mapper->setFirstRow(first); // First 3 rows skipped
mapper->setRowCount(count); // Only 5 rows mapped
mapper->setSeries(series);
mapper->setModel(model);
```

Dynamic Charts

- › Chart series can be dynamically altered
- › Dynamically adding/changing values in a series
 - › The chart reacts automatically, nothing special here
- › Changing the range of axes (scrolling)



Scrolling Approaches

- › `QAbstractAxis::scroll(qreal dx, qreal dy)`
 - › Automated animation
- › `QAbstractAxis::setRange(x, y)`
 - › Over timer or by new incoming values
 - › Will progress as `setRange` is called
- › `QValueAxis` properties `min` and `max` with a `QPropertyAnimation`
 - › Have Qt Animation FW do the iteration

Time-Bound Scrolling

- › Use a timer (QTimer or timer events) to refresh “painting” aka. to set the range
- › Bind the new range values to real elapsed time using QTime

```
class MainWindow : public QMainWindow
{
...
private:
    QTime m_startTime; ...
};

// mainwindow.cpp
void MainWindow::startScrolling()
{
    m_startTime.start();
}
void MainWindow::scroll()
{
    double secsGone = m_startTime.elapsed() / 1000.0;
    if (secsGone >= 10.0) m_chart->axisX()->setRange(secsGone-10,secsGone);
}
```

Theming

- › Easy theming
- › Several different “one-line” themes available

```
myChart->setTheme (QChart::ChartThemeLight);
```

- › All colors and attributes modifiable through the API
 - › Start from the scratch or
 - › Modify an existing theme

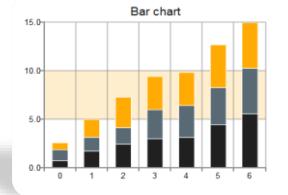
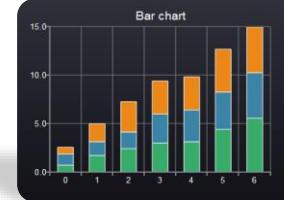
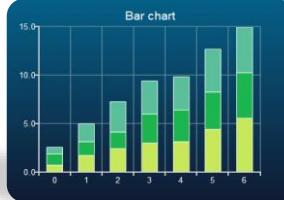
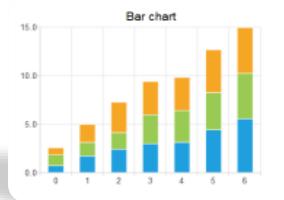
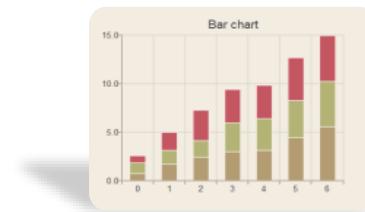
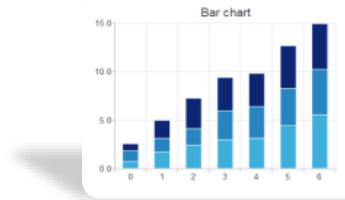
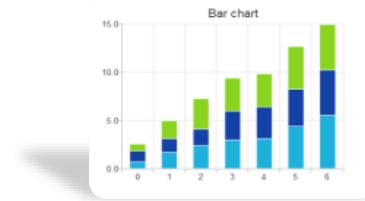


Chart Customization

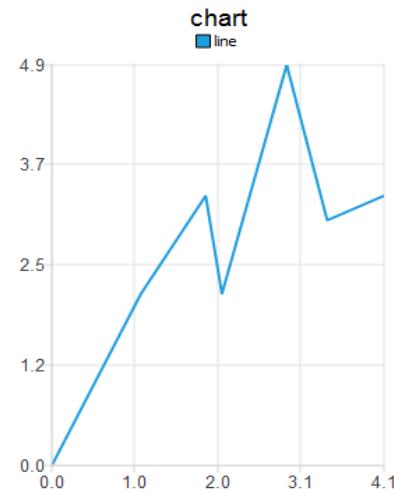
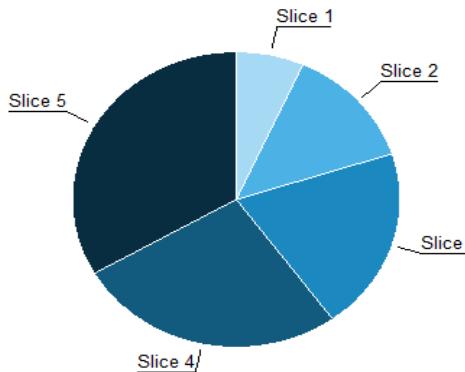
- › Lots of properties for customizing the look'n'feel manually
- › Trying out the examples gives you the best overview on customizable features

Series

| | |
|---------------------|--------|
| Horizontal position | 0,50 |
| Vertical position | 0,50 |
| Size factor | 0,70 |
| Start angle | 0,00 |
| End angle | 360,00 |

Slice

| | |
|---------------|--------------------------|
| Selected | <click a slice> |
| Value | 0,00 |
| Pen | <input type="button"/> |
| Brush | <input type="button"/> |
| Label visible | <input type="checkbox"/> |



- theme +
- theme -
- legend top
- legend bottom
- legend left
- legend right
- legend visible
- animation opt +
- animation opt -
- title color
- background color

Questions and Answers

- › Mention some Qt chart types
- › How chart type is defined?
- › Is it possible to add data into the series dynamically?
- › Does the chart automatically scale to show all data in the series?
- › How axis can be customized?
- › Can user interact with line points or bars in charts by clicking on them?
- › How item models are mapped to charts? Is it possible to pick up a subset of the model?

Summary

- › Qt Charts provides a comprehensive selection of different chart types
 - › Line charts, pie charts, bar charts, candle stick charts
- › Charts are based on the graphics view framework
 - › Chart itself derives from `QGraphicsWidget`
- › Charts type depends on series
 - › Series provide the data for the chart
 - › Items may be dynamically added and removed from the series
 - › Aixs may be scrolled, when new items are added or removed
- › Custom user interactions may be implemented by re-implementing event handlers

Thank You!

www.qt.io