Contents

• “Traditional” desktop widgets
  • QtCreator UI designer introduction
  • Signals and slots
  • Layouts

• MVC pattern
  • Model and view framework
  • View delegates

• Dialogs
Contents

• Graphics view
  • Graphics items
  • Graphics object model
QtCreator UI designer
Building “traditional” desktop UI’s
UI designer introduction

• This is interactive part...
  • Widgets walkthrough
  • How to add signal handlers
  • How to use layouts
  • Widget properties
Event handling
Events

- Any object that inherits from QObject can be target of events
  - QCoreApplication::postEvent, QObject::Event
- In GUI programs events are propagated to widget, which has the focus
  - Focus can be switched with mouse or *tab* key
  - If widget doesn’t process the event, it is forwarded to parent widget etc.
Events

• In order to process GUI events, you’ll need to
  • Create your own widget class, which inherits from one of the regular widgets or
  • Use event filters

• Event filter can be added to any object
  • `QObject::installEventFilter(anotherQObject);`
  • Events go to `anotherQObject::eventFilter`
Event filters

• **Why filtering?**

  • You can suppress unwanted functionality
    • Although the result might not be something that users are accustomed to
  • You can install new functionality
    • Replace the original function
    • Do something in addition to original function

• **See** *helloevents* from examples directory
Notes about events

- In general, the required functionality is available via signals & slots
  - Events are needed when implementing custom widgets

- Need for event filters is quite rare
Models and views
Displaying data in GUI
MVC pattern

- Model-view-controller
  - Model takes care of storing the data
  - View displays the data to user
  - Controller takes care of user interaction

- Why?
  - Model can be tested separately
  - Different views can be applied to same data
    - Portability of data
MVC in Qt - model

• Model framework in core module
  • Generic model interface (QAbstractItemModel)
  • More specialized models
    • List model (QAbstractListModel)
    • Table model (QAbstractTableModel)
• Ready-made models
  • QStandardItemModel
  • QStringListModel
MVC in Qt - view

- Views in GUI module
  - Generic view interface (QAbstractItemView)
    - Usually there’s no need to implement a view
  - Ready-made views
    - QListView, QTableView, QTreeView
    - Can work with any type of model
  - View delegates
    - Way to extend the ready-made views
MVC in Qt - controller

- Application works as controller
  - No generic frameworks
Model framework

- Use ready-made `QStandardItemModel`
  - Usually simpler to use and works with any kind of model (tree, list, column)

- Create your own model
  - Usually more efficient, as data can be directly mapped from your data storage into the view
  - Helpers for list and column models, tree a bit more complicated
Model basics

- Each element in model is identified by model index

- Each model index has row and column identifiers and parent index
Standard item model

• List, table or tree of \texttt{QStandardItem} objects
  • Each item contains a string, which is shown in the view
  • Also other data may be associated with the item

• Can be used without subclassing
  • Mapping between data model and the standard item model is needed
Standard item model - lists

- Standard item model root is accessed with `invisibleRootItem()`
- When used as list, all items are added under the root item
Standard item model - table

- In table model each row consists of multiple items, but all rows are still added under the `invisibleRootItem()`
Standard item model - tree

- In tree model each item may contain child items in addition to having rows and columns.

```cpp
QStringList nodeList;
nodeList << "Node1" << "Node2" << "Node3";
QStringList rowList;
rowList << "Row1" << "Row2" << "Row3";
QStringList columnList;
columnList << "Col1" << "Col2" << "Col3";

QStandardItemModel *treeModel = new QStandardItemModel(this);
treeModel->setColumnCount(3); // Doesn't resize automatically
QStandardItem *root = treeModel->invisibleRootItem();
int nodeIndex = 0;
foreach (QString node, nodeList) {
    QStandardItem *nodeItem = new QStandardItem(node);
    root->setChild(nodeIndex++, nodeItem);
    foreach (QString row, rowList) {
        QStandardItem *rowData = new QStandardItem(row);
        foreach (QString column, columnList) {
            rowData << new QStandardItem(row + " \" + column);
        }
        nodeItem->appendRow(rowData);
    }
}
u->treeView->setModel(treeModel);
```
Creating own models

• If your data is already somehow structured, it’s usually a good idea to implement the model directly on top of that
  • Standard item model has a separate item tree, which can be avoided
  • With standard items any changes need to be delegated between data and model

• Check “model subclassing reference” from QtCreator integrated help
Using the view

- To use a view, add one to your UI form with Qt designer
- Associate model to view from source code

```cpp
ui->tableView->setModel(tableModel);
```
• Model elements may contain any data
  • Data is identified with \textit{role}
    • Data that is shown in view has \textit{Qt::DisplayRole}
    • User-defined data can be added with \textit{Qt::UserRole}
  • To store data into a standard item model, use \textit{setData(something, Qt::UserRole)} when setting up the model
    • Data can later be accessed via a model index
Interaction with model

- Model data is stored as `QVariant` type
  - A container for other data types
  - Type-safe way to pass data around Qt
    - For example, an object pointer cannot be used as string
  - Variants should be used with helper functions
    - `qVariantFromValue`, `qVariantValue`

```cpp
// Map QOBJECT-based object to a QVariant
MyObject *obj = new MyObject();
QVariant var = qVariantFromValue<MyObject *>(obj);

// Map a variant back to QOBJECT
MyObject *obj = static_cast<MyObject *>(qVariantValue<MyObject *>(var));
```
Interaction with model

- Adding data to standard item –based model

```cpp
QStandardItem *item = new QStandardItem(str);
MyObject *obj = new MyObject();
item->setVariantData<qVariantFromValue<QObject *>>(obj);
root->appendRow(item);
```

- From that point on, obj can be accessed via the model API
Interaction with model

• Each view has a selection model, which tracks the selected items from that particular view
  • Is same model is displayed in multiple views, each view still has separate selection model

• Selection changes are notified as signals
  • However, selection model is not a GUI widget, so signals are not visible in designer

```cpp
connect(ui->treeView->selectionModel(), SIGNAL(selectionChanged(QItemSelection,QItemSelection)), SLOT(treeSelectionChanged(QItemSelection,QItemSelection)));
```
Interaction with model

• The data that was stored into the model can now be accessed from the selection change slot

```cpp
void MainWindow::listSelectionChanged(const QItemSelection &selected, const QItemSelection &deselected)
{
    QModelIndexList list = selected.indexes();
    if (!list.isEmpty()) {
        QModelIndex first = list.at(0);
        MyObject *data = static_cast<MyObject*>(qVariantValue<QObject*>(first.data(Qt::UserRole)));
        // Do something with data
    }
}
```
Updating model elements

• Standard item model elements can be changed with model `setData` function

  • `Qt::DisplayRole` changes the content that’s shown on the views

  • When model changes, all views are automatically updated

```cpp
void MainWindow::listSelectionChanged(const QItemSelection &selected, const QItemSelection &deselected)
{
    QModelIndexList list = selected.indexes();
    if (!list.isEmpty()) {
        QModelIndex first = list.at(0);
        ui->listView->model()->setData(first, "Selected!!!", Qt::DisplayRole);
    }
}
```
GUI dialogs
Basic concepts and functionality
Dialogs overview

• A dialog is a separate window, which performs some user interaction
  • Informs users about events
  • Asks user for some kind of input

• Modality
  • A modal dialog blocks program interaction until the dialog has been dismissed
  • Modeless dialog can be left open into background
Dialogs in Qt

- Qt provides built-in dialogs for simple tasks
  - Displaying messages and alerts
  - Asking for some input

```cpp
QMessageBox::StandardButton result = QMessageBox::question( 
    this, "Question", "Would you like to quit?", 
    QMessageBox::Yes | QMessageBox::No); 
if (result == QMessageBox::Yes) { 
    QMessageBox::warning(this, "Quit", "That's not working...", 
    QMessageBox::Close); 
} else { 
    QMessageBox::information(this, "Quit", "Good choice"); 
}
```
Dialogs in Qt

- Input dialog can be used to ask for different kinds of data

```cpp
QString msg = QInputDialog::getText(this, "Query", "Enter message");
int value = QInputDialog::getInt(this, "Query", "Enter value", 0, -100, 100);
QMessageBox::information(this, "Message", "You said: " + msg + " and chose " + QString::number(value));
```
Dialogs in Qt

- If built-in dialogs are not suitable, new dialogs can be created with help of QtCreator form editor
Creating a new dialog

- Select the "Class" version, not plain form
Creating a new dialog
Creating a new dialog

Choose a class name

Class name: Dialog
Header file: dialog.h
Source file: dialog.cpp
Form file: dialog.ui
Path: /home/tilli/qtprojects/hellodialogs

Configure...
Creating a new dialog

- UI resource, header and source are created similarly as when creating a new Qt project
  - Inherits from QDialog

```cpp
class Dialog : public QDialog {
    Q_OBJECT
public:
    Dialog(QWidget *parent = 0);
    ~Dialog();

protected:
    void changeEvent(QEvent *e);

private:
    Ui::Dialog *ui;
};
```
QDialog functionality

• A dialog must always be explicitly shown

  • Modal dialogs may use exec, which returns whether dialog was accepted or rejected

  • Modeless dialogs use show and need signal-slot connection to see whether accepted or rejected
Modal dialog

- Modal dialogs can be allocated from stack
  - Exception to normal QObject rules
  - The `exec` function, which displays the dialog starts a new *nested event loop*, which quits when the dialog is dismissed

```cpp
Dialog d;
int result = d.exec();
if (result == QDialog::Accepted) {
    QMessageBox::information(this, "Message", "Got OK");
} else {
    QMessageBox::information(this, "Message", "Got cancel");
}
```
Modeless dialog

- Modeless dialog needs to be allocated from heap (or as class member variable)
  - Needs to be alive until dialog is closed

```cpp
Dialog *d = new Dialog(this);
d->setModal(false);
d->show();
d->raise();
d->activateWindow();
connect(d, SIGNAL(accepted()), SLOT(dialogAccepted()));
connect(d, SIGNAL(rejected()), SLOT(dialogRejected()));
```

```cpp
void MainWindow::dialogAccepted()
{
    QMessageBox::information(this, "Message", "Got OK");
    delete sender();
}
void MainWindow::dialogRejected()
{
    QMessageBox::information(this, "Message", "Got cancel");
    delete sender();
}
```
Short exercise

• Get the *hellodialogs* example
  
  • Add two text input widgets into it using Qt form editor
  
  • If user selects *Ok*, display the content of the two text input widgets in a message box after the input dialog has been closed
  
• Try to find an alternative way to access and delete the modeless dialog from the slots
  
  • Instead of using QObject::sender()
Settings

Storing the GUI state
Settings

• In general, it’s nice if the application opens into the same state it was left last time
  • QSettings helps with that

• Principle is quite simple, load data in constructor, save in destructor
  • Some UI components seem to cause problems
    • View column width doesn’t change until view is visible
    • Also possible to load data in the main window showEvent
Settings

- Settings object takes two parameters when constructed
  - Company name and program name

- Each value is a QVariant
  - To save something, call `setValue`
  - To load, call `value` and provide a default

```cpp
QSettings settings("Symbio", "MusicLibraryGUI");
settings.setValue("Combo", ui->comboBox->currentIndex());
```

```cpp
QSettings settings("Symbio", "MusicLibraryGUI");
ui->comboBox->setCurrentIndex(settings.value("Combo", 0).toInt());
```
Settings

• Settings are stored into a text file
  • ~/.config/<company>/<program>.conf
Graphics view
Creating “non-traditional” UI’s
Introduction

• Why graphics view?
  • Traditional widgets are not designed for hardware-accelerated animated UI’s that users nowadays expect from mobile devices

• What is graphics view?
  • Higher-level abstraction over QPainter used by “traditional” widgets
    • Based on graphics items, which are cheap to paint compared to painting a QWidget
Introduction

- Can also be integrated with QWidgets

- Uses Model-View-Controller pattern
  - Multiple views can observe the same model (called graphics scene)

- Used by
  - KDE plasma desktop
  - Nokia mobile UI frameworks (Orbit, DUI)
Introduction

• Problems
  
  • No designer support
  
  • QtCreator can add a Graphics View widget to the form, but that’s about it
Back to Qt core

- Today’s GUI’s usually need pictures, animations and other fancy stuff

  - Qt has animations framework, which works with QObject properties

  - Pictures are usually stored in resources, which are compiled into the binary during build
Object properties

- All QObject-based classes support properties
  - A property is QVariant type, which is stored in a dictionary that uses C-style zero-terminated character arrays as keys
  - Properties can be dynamic or static
    - Dynamic properties are assigned at run-time
    - Static properties are defined at compile time and processed by the meta-object compiler
Object properties

• Static properties are declared into class header using Q\_PROPERTY macro

```cpp
class AnimatedPixmap : public QObject, public QGraphicsPixmapItem
{
    Q_OBJECT
    Q_PROPERTY(qreal rotation READ rotation WRITE setRotation)
}
```

• The above statement defines a property
  • Type is qreal, which is a floating-point number
  • Name is rotation
  • When read, rotation function is called
  • When changed, setRotation function is called
Object properties

• Static properties are used for example by QtCreator GUI designer
Why properties?

- When C++ objects are exported and used from *QtScript* or *QML* languages, the properties are automatically available
  - Signals & slots also

- Easiest way to use the animation framework is by modifying object properties
Property animations

- A property animation is created with help of QPropertyAnimation class
  - Specify which object and which property to update
  - Specify duration and start and end points
  - Start the animation

- Optional things
  - Specify number of loops (or infinite)
  - Specify an easing curve
Property animations

• When a property animation is started, the specified property is updated at regular intervals
  • Starting from start value, ending at end value and lasting for the duration specified

• After animation is finished, it either stops or runs a new loop
Property animations

- Property animation provides some signals that can be used to monitor the state
  - *finished* is particularly interesting
    - If animation is repeated, there might be a need to reverse the animation to avoid a jump
    - Just change direction and start again

```cpp
animation = new QPropertyAnimation(this, "scale", this);
animation->setStartValue(0.9);
animation->setEndValue(1.1);
animation->setDuration(10000);
connect(animation, SIGNAL(finished()), SLOT(animationFinished()));
animation->start();

if (animation->direction() == QPropertyAnimation::Backward) {
    animation->setDirection(QPropertyAnimation::Forward);
} else {
    animation->setDirection(QPropertyAnimation::Backward);
}
animation->start();
```
Animation notes

- Although presented here, animations framework is in no way related to graphics framework
  - Anything that requires timer-based events can be scheduled with the animations
Resource files

- Resource file specifies a collection of data that should be bundled into the binary file
  - For example pictures and localization data

- QtCreator can help add resources to project
  - Qt project file has RESOURCES statement, which contains a list of .qrc files
  - .qrc file is a text file, which is parsed by resource compiler during project build
Resource files
Resource files
Resource files

• After resource file has been created, add a *prefix* into it and a file under the *prefix*

• Resource is identified with a path, quite similarly as a file

  • :/<prefix>/resource-name>
Back to graphics

- Graphics framework is based on *graphics items* and *graphics layouts*
  
  - Quite similar functionality as with widgets and layouts, but not based on widgets

- Basic items
  
  - Shapes (like rectangle, ellipse, polygon), picture, text

- Graphics widgets
  
  - Similar to QWidget, but in context of graphics scene

- Proxy widget
  
  - Embeds “traditional” widgets into graphics scene
Graphics item vs. widget

• Graphics items can be target of events, similarly as a widget

• Unlike widgets a graphics item can be transformed
  • Move, rotate, shear, scale, project, etc.
  • Multiple transformations can be queued
Graphics items

• Graphics items are arranged into a tree hierarchy, similarly as widgets

  • Items may have a parent and a number of children
    • Item position is relative to parent item coordinates
    • If item doesn’t have parent, position is relative to scene coordinates

  • When a transformation is applied to an item, it is also applied to the child items
Graphics item events

- Graphics items get event notifications when user interacts with the scene
  - Similarly as with widgets, the graphics item needs to be subclassed and the required event handler functions implemented
Graphics scene

- A graphics scene is a container for items
  - The surface where the items are drawn
    - Each item has a z-order, higher z-order is drawn on top of lower one
  - Propagates events (for example mouse) to correct graphics items for processing
    - Regardless of transformations
  - Scene can be displayed in multiple views
Graphics view

• A graphics view is a \textit{QWidget}
  
  • Displays the contents of a graphics scene or a part of it

• A transform can be applied to the view
  
  • Transforms all items within the viewed part of the scene
Graphics example

• Illustrates some aspects of the graphics view and property animations
  • Adding items to scene and other items
  • Moving scene elements around
  • Animating transforms
  • Event handling
Programming exercise
Add GUI for music library
Programming exercise

- Get *musiclibrary-day4-pre* from the web page and open the root project file into QtCreator
  - Add *make install* build step
Programming exercise

• Add GUI run configuration, which runs `musiclibrarygui` from the `bin` directory
Programming exercise

• Several models are created in the constructor

• Add some views to visualize the model contents

```cpp
// Build a music library
MusicLibrary *library = MusicLibraryBuilder::build(this);

// Create a library model object
MusicLibraryModel *model = new MusicLibraryModel(library);

// Tree model of all objects
QAbstractItemModel *treeModel = model->tree();

// List of artists, records and songs
QAbstractItemModel *artistListModel = model->artists();
QAbstractItemModel *recordListModel = model->records();
QAbstractItemModel *songListModel = model->songs();
```
Programming exercise

• Each *Record* in the music library has a cover image
  
  • Add a graphics view, which displays the cover images