Qt Programming

Developed by Klarälvdalens Datakonsult AB for Trolltech ASA.

Overview

Overview cont'd.

Overview cont'd.

Overview cont'd.

Overview cont'd.

- The Qt Event System
  - Synthetic Events (page 218)
  - Delayed Invocation (page 221)
  - Event Filters (page 224)

Overview cont'd.

- Miscellaneous
  - Container Classes (page 229)
  - Qt Debugging Aids (page 254)
  - Some Thoughts About Portability (page 262)
  - QSignalMapper (page 270)
  - Find it in the Source (page 276)
  - Writing Your Own Widgets (page 279)

Overview - Optional Topics

- System Resources
  - QSound - sound support (page 285)
  - QImage (page 289)
  - Saving Settings using QSettings (page 341)
  - The System Clipboard (page 351)
  - Drag and Drop (page 353)
  - Network Programming (page 370)
  - External Processes with QProcess (page 388)

Overview - Optional Topics cont'd.

- Integration
  - ActiveQt (page 398)
  - Migrating Motif programs to Qt (page 429)

Overview - Optional Topics cont'd.

- Widgets
  - Emulating MDI with QWorkspace (page 474)
  - Q3Canvas (page 477)
  - QGraphicsView (page 488)
  - Model/View Programming (page 494)
  - QScrollArea (page 540)
  - Using OpenGL together with Qt (page 547)
  - QTextEdit (page 554)
  - Customized Drawing (page 572)
  - Widget Styles (page 609)

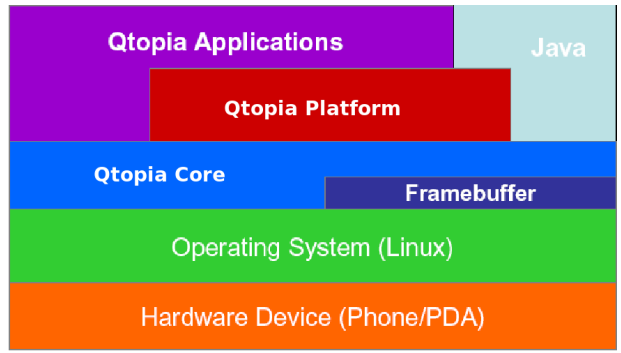
Overview - Optional Topics cont'd.

- Miscellaneous
  - QMake - automating Makefile creation (page 630)
  - Internationalization (page 653)
  - Using XML from Qt (page 669)
  - Multithreading (page 678)
  - SQL (page 701)
  - Plug-ins (page 727)
  - Development Tools for Linux (page 748)
  - Licensing (page 776)
  - Shipping Qt (page 782)
  - Unit Testing with QTestLib (page 789)

The Software Stack

- Qt - Cross-platform graphical user interface library
  - Check boxes, line edits, custom drawing, file access, socket access, ...
- Qtopia Core - Qt tailored for running on dedicated hardware
  - Interprocess communication, window decoration, access control to hardware, ...
- Qtopia - graphical desktop for embedded devices
  - Document standard, document manager, soft keyboard, ...

The Qtopia Software Stack, cont'd.



Qtopia Media Edition (upcoming)

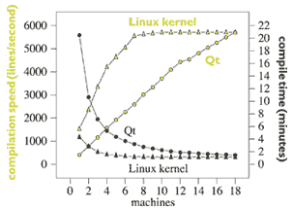
- Includes applications suitable for a PDA: Address book, calender, email, image viewers, multimedia players.

Qtopia Phone Edition

- Includes applications suitable for a Phone: Address book, calender, games.
- Qtopia Phone Edition also includes telephony features such as over-the-air configuration, SMS, MMS, GPRS, and a modem interface.

Other Products From Trolltech

- Qt Script for Applications (QSA) is a cross-platform application-scripting toolkit based on Qt.
- Teambuilder lets you build a compiler farm out of the unused CPU cycles of existing developer machines.



GUI Toolkits and Cross-Platform Libraries

- Why do you want GUI toolkits in the first place?
  - Native APIs (like Xlib and the Win32 API) are often difficult to use.
  - Native APIs provide a fine granularity that is often not needed or is outright unwanted.
  - Using native APIs contradicts portability.
  - The fine granularity makes it harder to avoid GUI bloopers.

GUI Toolkits and Cross-Platform Libraries cont'd.

- GUI toolkits vs. frameworks
  - GUI toolkits handle window creation, event handling, etc.
  - Frameworks also structure application design, event flow, etc.
  - Frameworks are almost always GUI toolkits
  - Qt and e.g. MFC are frameworks, Motif is not a framework.
- Why do you want portable software?
  - Larger market
  - More possible users
  - Portable code is often less buggy and more maintainable.

What are your options?

- MFC
  - Not portable.
  - Awkward programming, especially if you do not use Visual Studio.
  - Resulting programs huge and slow.
  - On the light side: Lots of literature and third-party software.
- wxWidgets
  - Uses layering approach (native widgets)
  - Programming model similar to MFC
  - Lacks signals and slots mechanism
  - Documentation and cross-platform uniformness are issues

What are your options? cont'd

- Java
  - Sloooow . . .
  - On the light side: Lots of literature and third-party software.
  - In the long run probably the major competitor to Qt when it comes to portability.
- .NET (not multiplatform beyond Windows)
  - Comprehensive framework, marketed very actively
  - Controlled by Microsoft and aimed to tie to Windows platform
- Defunct portable toolkits: StarView, Zinc, zApp, . . .

How can cross-platform libraries be implemented?

- **API layering**
  - API layer on top of the native API
  - Example: *java AWT* sits on top of Win32 API calls on Windows and on top of Motif/Xt API calls on Unix
  - Advantage: Styles are 100% platform-compatible
  - Disadvantage: Additional layer of complexity
  - Disadvantage: Often slower
  - Disadvantage: Toolkits often only provide lowest common denominator.
  - Disadvantage: Inheritance and specialization of widgets is difficult at best and impossible at worst.

How can cross-platform libraries be implemented? cont'd

- **API emulation**
  - Emulate one API on all non-native systems
  - Example: *MainWin* and *Wind/U* emulate the Win32 API on Unix systems.
  - Advantage: Native API can be used on one system.
  - Disadvantage: Emulated platforms are usually a lot slower than the native platform.

How can cross-platform libraries be implemented? cont'd

- **API emulation cont'd**
  - Disadvantage: Manufacturer has to follow changes on native system closely.
  - Disadvantage: Can lead to "API towering" (MFC on Win32 on Xlib...)
  - Disadvantage: Emulation only emulates documented features, but not undocumented features and bugs that are regularly exploited.

How can cross-platform libraries be implemented? cont'd

- **GUI emulation**
  - Emulates native styles by only using the most basic functions of the native APIs and drawing/handling the rest itself.
  - Example: *Qt* emulates Win32, Motif, Swing, ... styles on (so far) Win32, Unix, and MacOS X systems.
  - Advantage: Fast (sometimes even faster than pure-native programming).
  - Advantage: Styles are not tied to the platform.

How can cross-platform libraries be implemented? cont'd

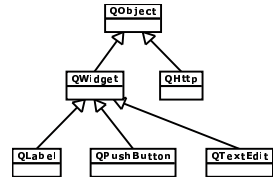
- **GUI emulation cont'd**
  - Advantage: It is relatively easy to inherit and specialize widgets, because the drawing is done within the toolkit.
  - Disadvantage: Manufacturer has to follow changes in styles of the various platforms closely and implement drawing of all widgets in all modes.
- Summary: GUI emulation is the most powerful technique.

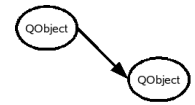
"Hello World" in Qt


```
#include <QtGui>

int main( int argc, char* argv[] )
{
  QApplication myapp( argc, argv );
  QLabel* mylabel = new QLabel( "Hello world", 0 );
  mylabel->show();
  return myapp.exec();
}
```


Parent/Child Relationships

- When a `QObject` (or inherited class) is created, it is given its parent as an argument. (Alternatively, the parent is assigned indirectly through layout managers.)
- The child informs its parent about its existence, upon which the parent adds it to its own list of children.
- If a widget's parent is 0, the widget is a window.
- Don't confuse parent/child relationship with inheritance.





Parent/Child Relationships cont'd.

- The parent does the following for children:
  - deletes them when it is deleted itself
  - hides/shows them when it is hidden/shown itself
  - enables/disables them when it is enabled/disabled itself
- For hide/show and enable/disable there is a *tristate* mechanism, which ensures that e.g. an explicitly hidden child is not shown when the parent is shown. (See Example *showhide*)

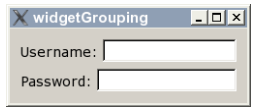
Creating Objects

- General Rule:
  - Objects inheriting from `QObject` are allocated on the heap using `new`, as the parent takes ownership of the object. Example:
    `QLabel* label = new QLabel( "Some Text", parent );`
  - Objects not inheriting `QObject` are allocated on the stack, not the heap. Examples:
    `QStringList list; QColor color;`
- Exceptions to the General Rule exist:
  - `QFile` and `QApplication` (inheriting `QObject`) are usually allocated on the stack.
  - Modal dialogs are often allocated on the stack, too.
- If in doubt, look at example code.

Geometry Management

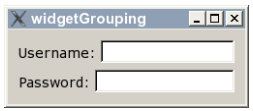
- On page 137 we will discuss geometry management in detail, but we will start with a simple example to get started.
- The two classes we will use are `QHBoxLayout` and `QVBoxLayout`. Notice that they are *not* widgets.
- Each widget should be added to a layout manager.
- Layout managers may be nested.

Example




```
QLabel* nameLabel = new QLabel( "Username: " );
QLineEdit* nameEdit = new QLineEdit;

QLabel* passwordLabel = new QLabel( "Password: " );
QLineEdit* passwordEdit = new QLineEdit;

QHBoxLayout* nameLayout = new QHBoxLayout;
nameLayout->addWidget( nameLabel );
nameLayout->addWidget( nameEdit );
```


Example cont'd

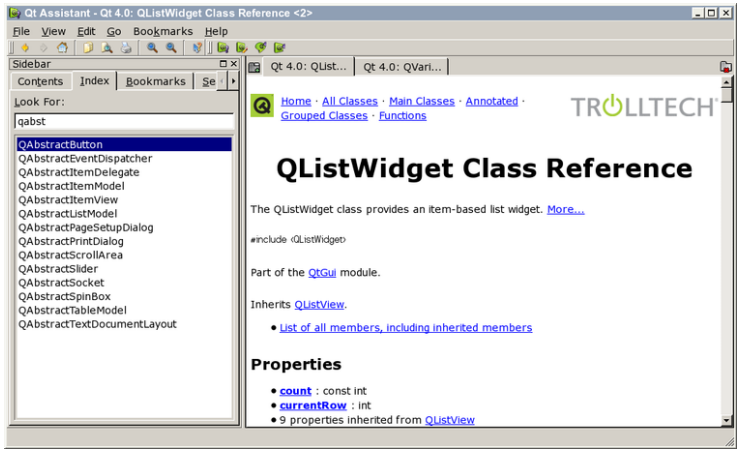



```
QHBoxLayout* passwordLayout = new QHBoxLayout;
passwordLayout->addWidget( passwordLabel );
passwordLayout->addWidget( passwordEdit );

QVBoxLayout* topLay = new QVBoxLayout;
topLay->addLayout( nameLayout );
topLay->addLayout( passwordLayout );

QWidget* top = new QWidget;
top->setLayout( topLay );
top->show();
```


How much C++ do you need to know?

- Objects and classes - Declaring a class, inheritance, calling member functions etc.
- Polymorphism - that is virtual methods.
- Operator overloading.
- Templates - for the container classes only.
- No namespaces, no RTTI, no sophisticated templates...

The Qt Reference Documentation - Viewed With Qt Assistant



Finding the Answers

- The reference documentation can be read using Qt Assistant
- There are lots of examples in `examples` to give you inspiration.
- Use the source! The source code is easy to read, and can answer questions the reference manual cannot answer!
- You can find an answer to your question in the mailing list archives: `http://lists.trolltech.com/`
- On irc.kde.org there is a channel called `#qt`.
- The complete multi-million line KDE source code is cross-referenced at `http://lxr.kde.org/`.

QMake Template

- QMake will help you create cross-platform make files easily.
- Simplest QMake file:

```
HEADERS = header1.h header2.h header3.h ...
SOURCES = source1.cpp source2.cpp source3.cpp ...
```

- On UNIX, create a Makefile: `qmake project.pro`
- On Windows, create a dsp file:
  `qmake -o project.dsp project.pro`
- A `.pro` file can initially be created using
  `qmake -project -o project.pro`.
- Typical usage:
  `cd projectdir; qmake -project; qmake; (n)make`
  (creates projectdir.pro and builds it).

Components and Include files

- Qt is split into a number of libraries: QtCore, QtGui, Qt3Support, QtXml, QtSvg, QtSql, QtNetwork, QtOpenGL, QtDesigner, QtUiTools, QtAssistant, QtTest, QAxContainer, QAxServer, and QtMotif.
- To use a given component, add a line similar to the following in your `.pro` file: `QT += network`
- By default, QMake projects are linked against QtCore and QtGui.
- Any class in Qt has a header file associated with it with a similar name. For example, the class `QLabel` is defined in `<QLabel>`
- For each module, there is also an include file for all files in the module, e.g. `<QtGui>`

Components and Include files cont'd.

- To make compilations as fast as possible, you should never use the module includes (`include <QtGui>`) in header files, but rather the specific includes (`include <QLabel>`), or, if possible, forward declarations like `class QLabel;`
- If your compiler does not support precompiled headers then it may add extra compile time to use the module includes.
- Alternatively, if your compiler supports precompiled headers, using the module headers consistently may speed up compilation.
- Be sure to always place `QtGui` (or `QtCore`, if `QtGui` is not used) as the first thing after any comments.

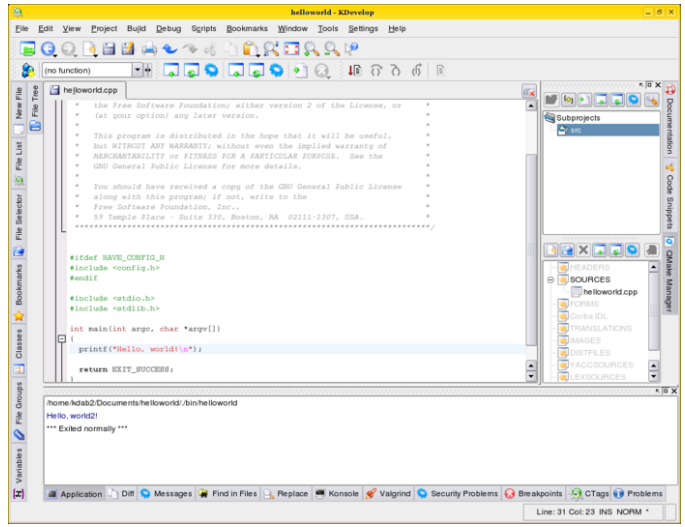
Questions And Answers

1. What does it mean when a QWidget has no parent?
2. What does it mean when a QObject (i.e., not a QWidget) has no parent?
3. What is QMake, and when is it a good idea to use it?
4. Name the places where you can find answers about Qt problems.

Overview

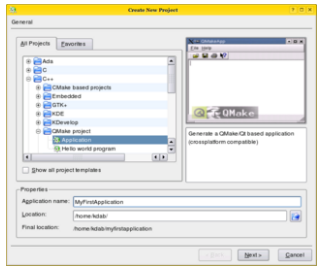
- KDevelop is an integrated development environment (IDE) similar to Borland C++ Builder or MS Visual Studio.
- It facilitates development thanks to file templates, class creation wizards, project management, documentation integration, integration with revision control systems and much more.
- Support for QMake, project templates for Qt-based libraries and applications and integration with Designer and Assistant make it well suited for Qt development.

Overview cont'd



Creating a new project

- Startup KDevelop and select `New Project` from the `Project` menu.



- Select `C++`, then `QMake` and finally `Hello world program` as the project type. Do not choose the `Application` type.

Creating a new project cont'd

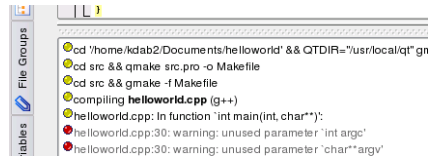
- Select a name for your project and a folder where the sources will be stored.
- Enter your name and email address and select a license, if you wish.
- Skip the next page, since we are not going to use a version control system in this course.
- Leave the file templates for implementation and header files as they are.

Compiling your program

- To compile your program, select `Build Project` from the `Build` menu, or press the `F8` key.
- The first time you do this, KDevelop will ask whether you want it to run QMake for you, to generate a Makefile. Answer yes to that question.
- To re-run QMake at a later time, right click on the project or subproject in the QMake Manager and select `Run qmake`.

Compiling your program cont'd

- The progress of the compilation and linking will be shown in a widget below the editor area. If there is an error during building, a click on the error line there will take you to the corresponding place in the code.

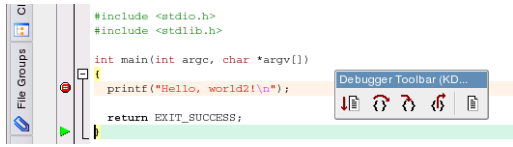


Debugging your program

- KDevelop integrates gdb, the GNU debugger, and allows you to set breakpoints, inspect the value of variables and for example inspect the contents of memory regions.
- To run your program under gdb, select `Start` from the `Debug` menu, or press `Ctrl-Shift-F9`. If the source code has changed since the last binary was built, the program will be rebuilt automatically before execution.
- Your program will be run and any output it produces will be displayed below the editor area.

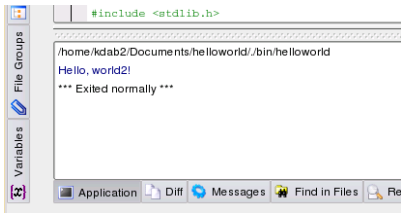
Debugging your program cont'd

- To set a breakpoint at a particular line in the code right click on it and select `Toggle Breakpoint`. The line will be marked with a small dot.
- Using the Debugger Toolbar you can single-step through your code or continue execution after a breakpoint has been hit.

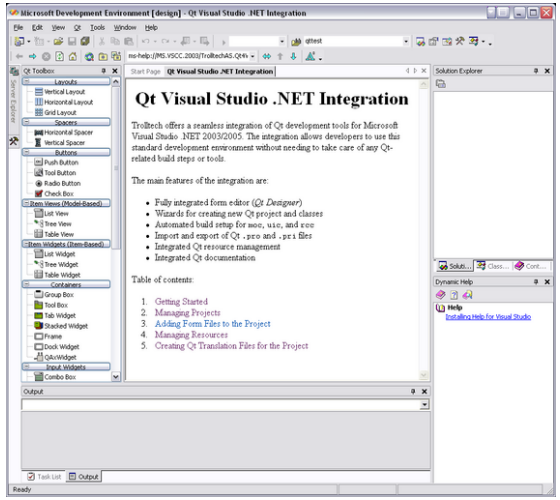


Exexuting your program

- To simply execute your program, select `Execute Main Program` from the `Build` menu, or press `Shift-F9`.
- Your program will be run and any output it produces will be displayed below the editor area.
- If the source code has changed, since the last successful build, KDevelop will automatically re-build it, before executing the program, when you press Shift-F9.



Introduction



Introduction

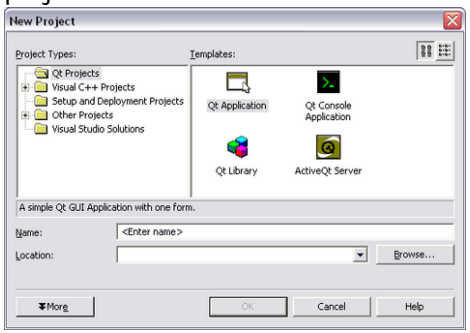
- Qt integrates with Microsoft Visual Studio in two ways:
  - For Visual Studio version 6 and 2002 a simple toolbar with buttons for running `moc`, starting Qt designer and opening `.pro` files is provided.
  - For Visual Studio 2003 and 2005 a more fully featured integration of designer, build system, class wizards etc. is offered. This kind of integration is the topic of this section.
  - We assume basic knowledge of Visual Studio, such as being able to build a project and navigate the GUI.

Installation

- Run the `qt-vsintegration-`*`version`*`.exe` installer. This requires no user intervention besides clicking "Next" and typing in the license key.
- Optionally, start VS.NET and go to the Tools→Options dialog, and check the setting in the `Qt` section.

Creating a Project

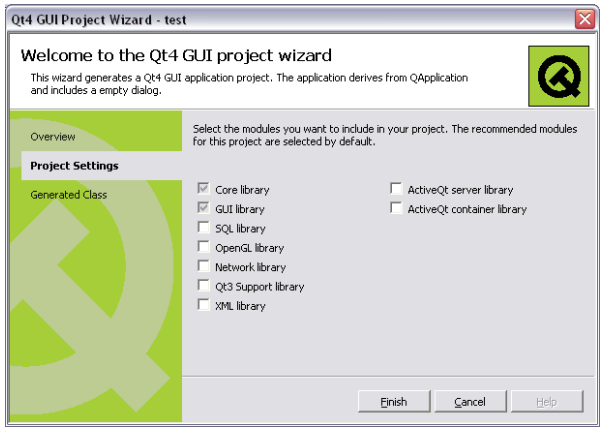
- Create a new project by selecting the File→New→Project... menuitem. The projects dialog includes a selection of Qt projects:



Creating a Project

- When creating a new project, the Qt project wizard shows up and allows you to adjust which libraries the project uses and the name of the generated class. The wizard for the Qt Application project template will always add a QWidget subclass header and `.cpp` file, a `.ui` file with a form that goes into the widget and a standard `main.cpp` file. See the screenshot of the project wizard on the following slide.

Creating a Project - Wizard

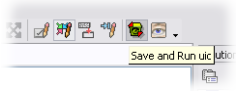


Creating a Project - Adding/Removing Files

- After the wizard completes, a new project is available with some files in it. The files can be opened by clicking them in the Solution Explorer dockwindow on the right.
- If you don't want or need the files created by the wizard, they can easily be removed from the project by rightclicking and selecting Remove.
- To add a non-gui class to the project, select `Project→Add Class...` and choose C++ Class.
- To add a Qt GUI class to the project, select `Project→Add Qt GUI Class...` and fill out the dialog. This will create a `.h`, `.cpp` and `.ui` file similar to the ones created by the project wizard.

Using Qt Designer in VS.NET

- Qt Designer is fully integrated and set to be the editor for `.ui` files. Just click on a `.ui` file in the Solution Explorer and designer will show up in the central area in VS.NET.
- Work with the designer as usual. You can at any time press the "Save and Run uic" toolbar button to make VS.NET pick up the generated code so it can be used for autocompletetion in the editor. (Can be automated from Tools|Options|Qt)



Using Qt Designer in VS.NET

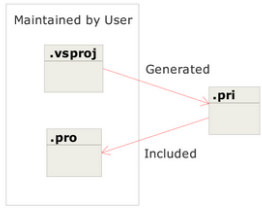
- Double-clicking on a widget in a form will make the editor jump to the `.cpp` file with widget encapsulating the form, adding a slot of the form

```
void MyWidget::on_widget_signal( ... )
{
}
```

where "widget" is the widget that was double-clicked, and "signal" the most common signal to connect to for that widget. In the corresponding `.h` file, the new slot will be listed as a private slot.

Build/Project Management

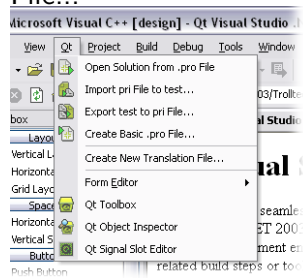
- With the project being fully handled by Visual Studio and not by qmake, we need a way of "exporting" the project to be able to build it from the windows command line or on other platforms.



- The Qt/VS.NET integration allows to export qmake files for this.
- The idea is to have a qmake .pro file that can be edited by the user and a qmake .pri file, listing all the files in the project, that is generated by VS.NET.

Build/Project Management

- A starting point for the `.pro` file can be created with Qt→Create Basic .pro File...
- The `.pri` file is updated with Qt→Export <projname> to pri File...



Project Task: Your first Qt Application

- Implement the application shown here.
- It is part of the exercise to find the correct widgets to use.
- Optionally: Insert data programmatically.
- Optionally: Make sure the labels for `Country` and `Comments` are centered.
- You should not use Qt Designer, QMainWindow, or QDialog subclasses.

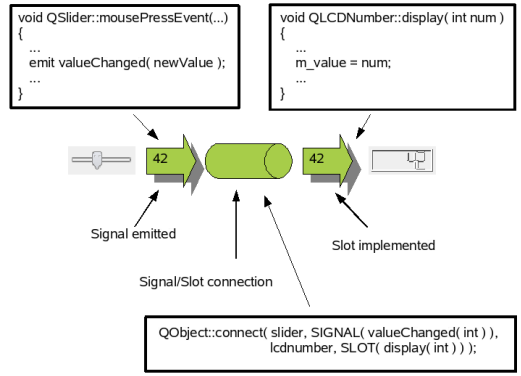
Alternative Implementations

- General problem: *How do you get from "the user presses a button" to your C++ code that implements the backend logic?*
- MFC uses message maps: difficult (if not impossible) to handle without Visual Studio or other tools, imposes a "new language" on top of C++
- Motif uses callbacks: not type-safe, easy to put fatal bugs in
- wxWidgets (and Java 1.0) use virtual methods: not very practical; for each widget, a new class is needed
- Qt uses signals and slots (and the Java 1.1+ event handling mechanism is similar) for high-level (semantic) events (listeners) and virtual methods for low-level (syntactic) events.

Signals and Slots

- Provides type-safe callbacks
- After getting used to it, they are easier to use than message maps, more secure than callbacks, more flexible than virtual methods
- Fosters component-based programming
- Needs a precompiler ("moc")

Linking UI to Backend Code



Signals and Slots cont'd

- Class declaration:

```
class MyClass : public QObject
{
    Q_OBJECT
    ...

signals:
    void mySignal( int );
    ...

public slots:
    void mySlot( int );
}
```


Signals and Slots cont'd

- You do not implement **any code** for signals, just specify them (and of course emit them).
- "Sending a signal": `emit valueChanged( 7 );`
- Slots are implemented as normal methods.

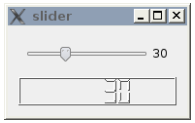
Signals and Slots cont'd

- Compilation:
  - Creating a moc file with the precompiler `moc`:
    `moc -o moc_myclass.cpp myclass.h`
  - Compiling this moc file and the class file itself:
    `c++ -c moc_myclass.cpp; c++ -c myclass.cpp`
  - Linking all files:
    `c++ -o myapp moc_myclass.o myclass.o`
- QMake takes care of moc files for you.

Questions And Answers

1. How do you connect a signal and a slot?
2. What code do you need to write in order to implement a slot?
3. What code do you need to write in order to emit a signal?
4. Do you need a class to implement a slot?
5. Can you return a value from a slot?
6. When do you need to run QMake?
7. Where do you place the `Q_OBJECT` macro and when do you need it?

Project Task: Slider with Value indicator



- Implement a slider class that is source level compatible with `QSlider`, and which in addition shows the current value of the slider.
- Use a `QSlider`, a `QLabel` and a layout manager to create your new slider.
- Use the example in *examples/connect* as a test case for your slider.
- Optional: Ensure the label has a fixed size that is wide enough to show the value without resizing as the value changes. (Hint: see `QFontMetrics`)
- Optional: Discuss advantages and disadvantages of inheriting from QSlider instead of using an instance in a layout.

Understanding moc-generated files

- See the definition of `Q_OBJECT`, `signals`, `slots`, and `emit` in `src/corelib/kernel/qobjectdefs.h`
- Try looking at the moc generated files in *examples/signalSlots*.

Back to the Original Problem

- Problem: *How do you react to a push button being pressed?*
- Solution: *Implement a slot in your dialog, and connect the button to it.*
- `connect( button, SIGNAL( clicked() ), this, SLOT( okClicked() ) );`

Project Task: Connecting Signals and Slots

- Create an application as shown here.
- When pressing the "Select Color" button, the label should be updated with the color's name.
- Use the static method `QColorDialog::getColor()` to fetch a color, `QColor::name()` to get the color name and `QLabel::setText()` to set it.
- Optional: In `QColorDialog`, honor the user clicking "cancel", and provide it with the current color to start from.
- Optional: Set the selected color as the label's background (Hint: see `QPalette`)

Libraries

Qt is split up into the following libraries:

- **QtCore**, included and linked by default
- **QtGui**, included and linked by default
- **QtNetwork**, included and linked optionally
- **QtOpenGL**, included and linked optionally
- **QtSql**, included and linked optionally
- **QtXml**, included and linked optionally
- **QtSvg**, included and linked optionally
- **Qt3Support**, included and linked optionally

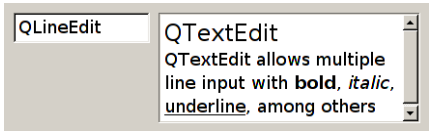
. . . plus a few rarely used ones.

Buttons

- **QAbstractButton** Abstract base class for all buttons
- **QCheckBox** Check boxes
- **QRadioButton** Radio buttons - if two radio buttons have the same parent, then they are exclusive.
- **QPushButton** Simple push buttons
- **QToolButton** Auto-raising push buttons, for use in toolbars
- **QButtonGroup** Manages groups of buttons (is *not* a widget!).

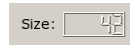
Widgets for Text Input

- **QLineEdit** Single-line text entry
- **QTextEdit** A sophisticated single-page rich text editor
- **QCompleter** Utility class offering completion in e.g a line edits.
- **QValidator** Utility class for limiting valid input to e.g. a line edit.



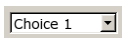
Widgets for Displaying Text

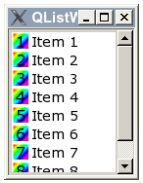
- **QLabel** Static text, pixmap or animation
- **QLCDNumber** Seven-segment-style display
- **QTextEdit** In read-only mode, this widget can be used as a rich text viewer without navigation features
- **QTextBrowser** Displays rich text and provides simple navigation features



Selection Widgets

- **QComboBox** Combo boxes
- **QFontComboBox** Specialized combo box for choosing font families.
- **QListView** Non-hierarchical list or icon view, uses model/view
- **QListWidget** Same, with classic item-style interface
- **QListWidgetItem** Item class for use with QListWidget





Selection Widgets

- **QTreeView** Multi-purpose hierarchical list display, uses model/view
- **QTreeWidget** Same, with classic item-style interface
- **QTreeWidgetItem** Item class for use with QTreeWidget

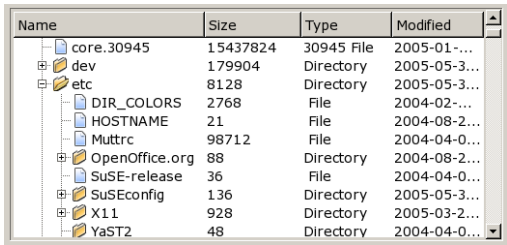
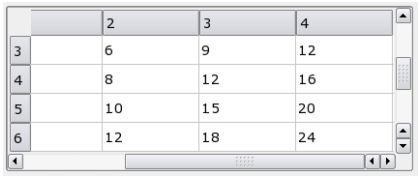


Table Widgets

- **QTableView** Table, uses model/view
- **QTableWidget** same, with classic item interface
- **QHeaderView** Header for tables
- **QTableWidgetItem** Items for `QTableWidget`

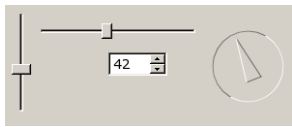


Model/View classes

- **QAbstractListModel**, **QAbstractTableModel,**
- **QDirModel**, **QAbstractProxyModel**,
- **QSortFilterProxyModel**, **QStandardItemModel**
  Model classes for use with model/view
- **QAbstractItemDelegate**, **QItemDelegate,**
- **QSqlRelationalDelegate**
  Delegate classes for user interaction in views

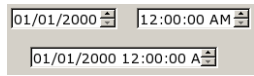
Widgets for Bounded Range Input

- **QAbstractSlider** Abstract base class for `QSlider`, **QDial**, and `QScrollBar`
- **QSlider** Linear number entry
- **QDial** Speedometer-like widget
- **QScrollBar** Classic scrollbar
- **QSpinBox**, **QDoubleSpinBox** Constrained number entry



Widgets for Bounded Range Input

- **QTimeEdit** Selection of a time (hour/minute/second).
- **QDateEdit** Selection of date (day/month/year)
- **QDateTimeEdit** Selection of date, and a time.
- **QCalenderWidget** Selection of date from a calendar.

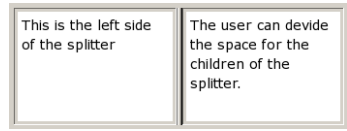


Widgets Arranging Others

- **QWidget** Base class for all widgets, but is often also used as an invisible container.
- **QFrame** Draws a frame around its children
- **QGroupBox** Groups its children with a frame, and a label at the top.

Widgets Arranging Others cont'd

- **QSplitter** Lets user distribute screen space among its children
- **QAbstractScrollArea** Abstract base class for everything that scrolls.
- **QScrollArea** A widget showing only parts of its children

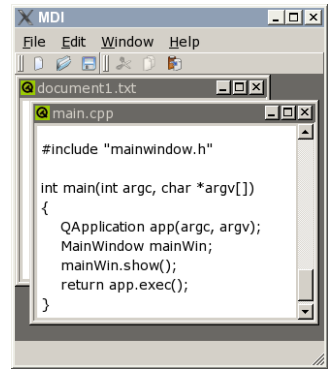


Widgets Arranging Others cont'd

- **QGraphicsView** A View for placing individually movable items. Includes features like collision detection.
- **QStackedWidget** Card deck-like stack of widgets, only top widget is visible

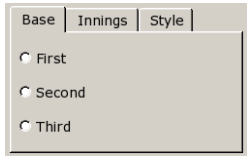
Widgets Arranging Others cont.

- **QWorkspace** Emulates MDI.
- **QDockWidget** A widget that can be docked into a QMainWindow or float around as a separate window



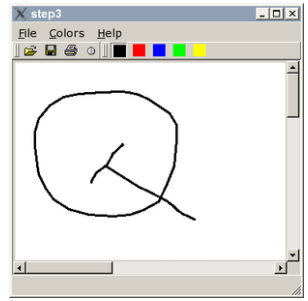
Miscellaneous Widgets

- **QToolBox** A vertical bar containing subpages
- **QTabWidget** Organizes a number of pages using tabs
- **QTabBar** Tab bar for tab dialogs
- **QSplashScreen** Shows splash screens



Widgets for the Office Look

- **QMenuBar**, **QToolBar** Menu/Tool bar
- **QMenu** Both for menus in menu bars and stand-alone context menus
- **QToolButton** Push button with typical auto-raise look
- **QStatusBar** Status bar
- **QMainWindow** Organizes menu/toolbar, status bar, and document area
- **QScrollBar** Scroll bars



Dialogs

- **QColorDialog** Lets user select a color (and the alpha channel)
- **QFileDialog** Lets user select one or more files or a directory
- **QFontDialog** Lets user select a font

Predefined Dialogs

- **QInputDialog** Lets user provide one-line input
- **QMessageBox** Displays message and icon to user
- **QProgressDialog** Displays progress of lengthy operations to user
- **QErrorMessage** A dialog offering "*Do not show this message again*"
- **QDialog** Base class for all dialogs

Geometry Management

- **QVBoxLayout** Manager for laying out items vertically
- **QHBoxLayout** Manager for laying out items horizontally
- **QGridLayout** Manager for laying out items both vertically and horizontally
- **QStackedLayout** Manager for laying out items such that only one is visible at a time

I/O and Network programming

- **QIODevice** Base class for everything that can consume and produce streamed data.
- **QFile** Platform-independent encapsulation of a file
- **QTemporaryFile** A temporary file that is removed from disk when the `QTemporaryFile` object is deleted.
- **QByteArray** Wrapper around a `const char*`
- **QBuffer** `QIODevice` that works on a `QByteArray` (i.e., in-memory)

I/O and Networking cont'd

- **QDataStream, QTextStream** Serializes binary/text data
- **QDir, QFileInfo** Platform-independent encapsulation of a directory and file information.
- **QResource** Accessing in-binary resources like images (can also be directly accessed via `QFile`, `QPixmap`, or `QImage`)
- **QTcpSocket**, **QTcpServer**, **QUdpSocket** Socket communication
- **QProcess** Asynchronous process operations

I/O and Networking cont'd

- **QHostInfo** Hostname lookups (using the native system lookup methods)
- **QNetworkInterface** Provides a listing of the host's IP addresses and network interfaces.
- **QHttp** Http network communication
- **QFtp** Ftp network communication

Graphics and Printing

- **QPaintDevice** Abstract class that represents objects which can be painted on. Inherited by `QWidget`, `QPicture`, `QPixmap`, `QImage` and `QPrinter`.
- **QPainter** Contains the actual painting functionality with which painting is done on `QPaintDevice` objects.
- **QPrinter** Represents a sheet of paper in a printer.
- **QPixmap** A raster-based, server/OS-side off-screen image
- **QImage** A raster-based, client/application-side off-screen image
- **QPicture** A vector-based, client/application-side off-screen image

Graphics and Printing cont'd.

- **QRectF**, **QPointF**, **QSizeF**, **QLineF**, **QPolygonF** Geometrical primitives
- **QPen, QBrush, QColor, QPalette** Appearance classes
- **QFont, QFontMetricsF, QFontInfo, QFontDatabase** Font classes
- **QPainterPath** General drawing primitive.

Graphics and Printing cont'd.

- **QGradient** Specifies gradient fills (by means of the subclasses `QConicalGradient`, `QLinearGradient`, and `QRadialGradient`)
- **QGLWidget**,**QGLContext**,**QGLFormat**,**QGLColormap** OpenGL classes
- **QMatrix** Used for defining non-trivial graphics transformations.

Database

- **QSqlQuery** Means of executing and manipulating SQL statements
- **QSqlRecord** Encapsulates a database record
- **QSqlQueryModel** High level model for accessing a database.
- **QSqlRelationModel** Stores foreign-key relations between tables

Multithreading

- **QThread** Platform-independent threads
- **QMutex**, **QSemaphore**, **QReadWriteLock** Access serialization between threads
- **QMutexLocker**, **QReadLocker**, **QWriteLocker** Management classes for locking and unlocking mutexes.
- **QWaitCondition** Allows waiting for/waking on conditions between threads
- **QThreadStorage** Per-thread data storage

Multimedia

- **QImageIOHandler** Abstract base class for supporting additional image formats
- **QImageReader**, **QImageWriter** Format-independent image decoding and encoding
- **QMovie** Incremental loading of animations or images
- **QSound** Access to the platform audio facilities

Desktop Integration

- **QDesktopWidget** Access to screen information on multi-head systems.
- **QSystemTrayIcon** Provides an icon for an application in the system tray.
- **QDesktopServices** Provides methods for accessing common desktop services (currently opening URLs).
- **QFileSystemWatcher** Provides an interface for monitoring files and directories for modifications.
- **QtDBus** Framework for interprocess communication on Unix.

Miscellaneous

- **Collection classes** Lists, arrays, maps, stacks, ...
- **String classes** QString, QStringList, QRegExp
- **XML** SAX and DOM classes for accessing XML documents
- **Drag and Drop** Classes for implementing drag and drop
- **QSettings** Saving user preferences between sessions
- **QDate**, **QTime**, **QDateTime** Classes for manipulating dates and times
- **Unicode text rendering** `QTextDocument` and related classes
- **Styling system** Classes for defining the look of user-interface elements, `QStyle` and related classes
- **Unit Testing** Classes for defining and executing unit tests.
- **Undo Framework** Implementation of the Command design pattern.

The Most Important Classes

- **QCoreApplication**, **QApplication** Event loop, integration into the system, command-line parameters. (May be accessed through the global variable `qApp`).
- **QObject** Base class for almost everything in Qt, implements the introspection needed for signal/slots.
- **QWidget** Base class for all UI elements, general methods like `show()`, `resize()`, `setGeometry()`; empty default implementations for event handlers (`mousePressEvent()`, etc.)

Lowlevel Events

- On page 67ff we discussed signal/slots.
- Notification about events without semantics outside the widget is not done via signals but via virtual methods.
- The class `QWidget` contains several empty implementations for low-level event handlers:
  - `void QWidget::mousePressEvent(QMouseEvent*)`
  - `void QWidget::keyPressEvent( QKeyEvent* )`
  - `void QWidget::focusInEvent( QFocusEvent* )`
  - `void QWidget::paintEvent( QPaintEvent* )`
  - ...

When are signals and slots used, and when are virtual methods used?

- Signals are used when the user interacts on a higher level ("semantic events"). Example: *A user chooses an element from a listbox.*
- Virtual methods are used when there are no (intrinsic) semantics in the interaction ("syntactic events"). Example: *The user clicks into a window.*

Some Differences between Signals and Events

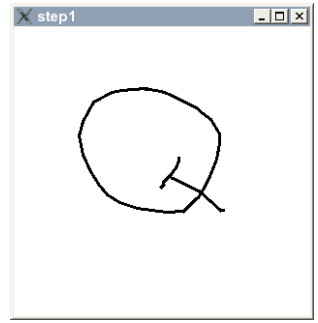
- Signals are just emitted without knowing the receiver. Events go to a specific widget.
- Signals can have any number of receivers. Events go to exactly one receiver.
- Events pass through event filters, signals do not.

Basic Drawing

- Drawing can be done on instances of `QWidget`, `QPrinter`, `QPixmap`, `QImage`, `QPicture`, and `QGLPixelBuffer`, in other words anything inheriting `QPaintDevice`.
- In order to draw, you need to create a `QPainter`, specifying the paint device as argument.
- In old code, you'll also see `QPainter::begin()` and `QPainter::end()` to start and end the painter.
- If you draw on a pixmap, it is important to de-activate the `QPainter` with `QPainter::end()` before using the pixmap further (e.g. copying or painting it).
- High-level rendering: SVG with `QSvgRenderer` or `QSvgWidget`, HTML with `QTextDocument`.

Basic Drawing cont'd.

- All painting on widgets must be done in the `paintEvent()` method, or functions called from there.
- You can schedule a paint event by calling `update()`.
- Qt uses double buffering by default (this can be switched off, though).
- *Basic paint program*: *paintProgram/step1*



Project Task: Adding Line Styles to the Paint Program - Part I

- Modify the `ScribbleArea` class, so it has a slot for changing the pen style.
- To test your code simply call the slot directory from main.cpp with canned data.
- Discuss why it doesn't seem to work. (Hint did you try moving your mouse really fast?)

QPixmap vs. QImage

- `QImage` is designed and optimized for I/O and for direct pixel access/manipulation
- `QPixmap` is designed and optimized for drawing.
- There are (slow) functions to convert between QImage and QPixmap.
- Think of it as *pixmaps being stored in graphics card memory, while images are stored in main memory.*

Main Windows

- Standard application windows are most easily implemented with `QMainWindow`.
- Menu bars and status bars are created on demand. Just call `QMainWindow::menuBar()` or `QMainWindow::statusBar()` and use the returned pointer.
- Tool bars must be created externally. Use `QMainWindow::addToolBar()` to add them to the window.
- The main area (usually: the "document window") must be set with `QMainWindow::setCentralWidget()`.

MainWindows cont'd.

- Each menu in the menu bar is an instance of the class `QMenu`.
- Insert a popup menu into a menu bar using `QMenuBar::addMenu()`.
- Items are inserted into a menu and onto a toolbar as `QActions`, using `QWidget::addAction()`, and `QWidget::addActions()`.
- Alternately, you may use `QToolBar::addAction()` and `QMenu::addAction()` which will build the actions from a title.

QAction

- An action bundles program functionality with its user-visible appearance. Actions are represented by the class `QAction`.
- Actions are either command actions or toggle actions. (The class `QActionGroup` adds radio behavior.)
- Assign a tool tip with `setToolTip()`, a status tip with `setStatusTip()`, an icon with `setIcon()`, a key binding with `setShortcut()`, etc.
- Actions provide the signals `triggered()` and `checked(bool)`.
- Enabling/disabling or toggling an action is immediately reflected in the user interface wherever the action is used.
- *paintProgram/step2* shows `QMainWindow` and `QAction` together.

Project Task: Adding Line Styles to the Paint Program - Part II

- Add Actions to the paint window for each of the pen styles.
- Use the icons for the line styles found in the icons subdirectory.
- By magic your changes to the previous project have made it into step 2!
- Hook things up so the newly created buttons actually works.

Working With Files

- Rule Number One for portable file access: Do not use the native functions like `open()` or `CreateFile()`, but Qt classes instead.
- Classes to use: `QFile` and either `QTextStream` or `QDataStream`.
- Optional: `QFileInfo` and `QDir`

Working With Files cont'd.

- Trivial file writing, text data:

```
QFile file( "myfile.txt" );
if( file.open( QIODevice::WriteOnly ) ) {
    QTextStream stream( &file );
    stream << "HelloWorld ";
    stream << 4711;
    file.close();
}
```


Working With Files cont'd.

- Trivial file reading, text data:

```
QFile file( "myfile.txt" );
if( file.open( QIODevice::ReadOnly ) ) {
    QTextStream stream( &file );
    QString mystring;
    stream >> mystring;
    int myint;
    stream >> myint;
    file.close();
}
```


Working With Files cont'd.

- When using `QTextStream`, you should be aware of encoding, and might need to call `QTextStream::setCodec()`.
- Alternative ways of reading a file includes:
  - `QByteArray QFile::readAll()`
  - `QByteArray QFile::readLine(qint64 maxlen = 0)`
  - `QString QTextStream::readAll()`
  - `QString QTextStream::readLine(qint64 maxlen = 0)`

Working With Files cont'd.

- An alternative to `QTextStream` is `QDataStream` which adds extra information about the data being written.
- Files written with `QDataStream` are portable between hardware architectures and operating systems.
- Files written with `QDataStream` are not human-readable.
- Lots of Qt classes can work together with `QDataStream`, but not `QTextStream`
- You can also use `QTextStream` and `QDataStream` to write into memory or even other destinations.

Working With Files cont'd.

- Even easier: Loading and saving images with `QPixmap::load()`, `QPixmap::save()`, `QPicture::load()`, and `QPicture::save()`.
- If we wanted more control (like loading a scaled version, or only part of an image) we might use the class `QImageReader`.
- Asking for filenames: `QFileDialog`.

Printing

- Printing in Qt is conceptually easy!
- Just let `QPainter` draw into a `QPrinter` object instead of a `QWidget` or `QPixmap` object.
- To get the user-selected printing options, use `QPrintDialog`.
- Paging has to be done in the drawing code, use `QPrinter::newPage()`.
- In order to get PDF output, use `QPrinter::setOutputFormat(PdfFormat)`.

Printing cont'd.

- Widgets can be *screen-shot* using the static method `QPixmap::grabWidget()`. See the example in *widgetPrinting*.
- The class `QTextDocument`, which represents rich text, has a method `print(QPrinter*)`.

Scrolled Areas

- Scrolled areas can easily be added with `QScrollArea` – no need to use individual scrollbars.
- Easiest way to use `QScrollArea`:
  - Create a `QScrollArea`.
  - Create a widget that is to be scrolled and call `QScrollArea::setWidget()` with the widget as argument.
  - set up a `sizeHint()` for the widget, or call `setFixedSize()`.
- *paint/step3* adds file operations, printing operations and a scrollable area to our paint program.

Project Task: Text Editor

- Create a text editor with *load*, *save*, *quit*, *about* and *About Qt*
- A `QTextEdit` serves for editing the text.
- Optional: Show whether the file is dirty in the status bar, ask the user whether (s)he wants to save if the file is dirty when (s)he quits the application. Make sure it also asks when the window is closed via the window manager.
- Optional: Add printing.
- Optional: Show the cursor position in the status bar.

Questions And Answers

1. What is the difference between signal/slots and events?
2. Which class would you use for drawing a line?
3. In which functions are you allowed to paint using a QPainter?
4. Name the components of a main window.
5. What is the difference between `QDataStream` and `QTextStream`?

Doing it Yourself

- You can place and resize widgets yourself using the three methods `move()`, `resize()`, and `setGeometry()`.
- Example:

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

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

Making Qt do the Work

- Definition: Specifying the relations of elements to each other instead of the absolute positions and sizes.
- Advantages:
  - Works with different languages.
  - Works with different dialog sizes.
  - Works with different font sizes.
  - Better to maintain.
  - Easier to program once you are used to it.
- Disadvantage: More difficult to program at first.

Specifying Sizes

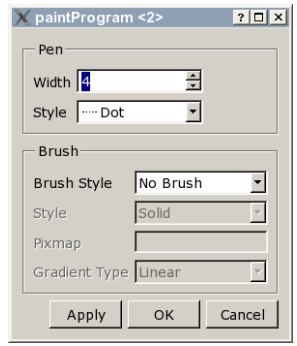
- Never call `setGeometry()`, `resize()`, or `move()` on the managed widgets.
- Override `sizeHint()`, `minimumSizeHint()` or call `setFixedSize()`, `setMinimumSize()`, `setMaximumSize()`

Classes Used

- **QHBoxLayout** Lines up widgets horizontally
- **QVBoxLayout** Lines up widgets vertically
- **QGridLayout** Arranges the widgets in a grid
- **QStackedLayout** Arranges all its managed widgets in a stack on top of each other, so only the topmost one is visible.

Project Task: Implementing a Configuration Dialog – part I

- Starting from *handout/config-dialog*, implement the dialog shown.
- First focus on getting the layout correct.
- Next populate the widgets.
- Optional: Get the enabling/disabling of the brush section working properly.
- Optional: Use the images for line types found in the icons subdirectory.



Some Terms

- **Stretch** Determines how a widget resizes relative to the other widgets when there is more space than the widgets need.
- **Margin** The space that a layout manager reserves around the managed widgets.
- **Spacing** The space that a layout manager reserves between the managed widgets.

Some Terms cont'd.

- **Strut** Gives a one-dimensional layout manager a minimum width or height.
- **Minimum and maximum sizes** Can be set with `QWidget::set{Minimum,Maximum,Fixed}{Width,Height,Size}()`
- **Chinese boxes** Nesting `QVBoxLayout` and `QHBoxLayout` into each other allows for very flexible layouts.

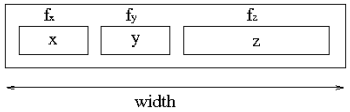
An Example

```
MyDialog::MyDialog( ... )
{
  QLabel* labelLA = new QLabel( "Some text" );
  QLineEdit* editED = new QLineEdit;
  ...
  QHBoxLayout* rowHBL = new QHBoxLayout;
  rowHBL->addWidget(labelLA);
  rowHBL->addWidget( editED, 2 );
  ...
}
```


Extra space

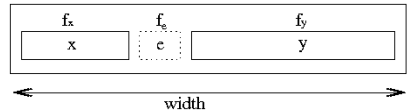
- To get some extra space in your layout, you can either:
  - Specify some spacing using `QLayout::setSpacing ()`and margin using `QLayout::setMargin()`
  - Invoke `QBoxLayout::addStretch()`, which acts like an invisible widget, which also claims some of the extra space.
  - Invoke `QBoxLayout::addSpacing()` which adds a non-stretchable space to your layout.

Stretching



- $\text{size}(x) = \text{maximum}(\ \textit{width} \times \frac{f_x}{f_x+f_y+f_z},\ \text{minimumSize}(x))$
  $\text{size}(y) = \text{maximum}(\ \textit{width} \times \frac{f_y}{f_x+f_y+f_z},\ \text{minimumSize}(y))$
  $\text{size}(z) = \text{maximum}(\ \textit{width} \times \frac{f_z}{f_x+f_y+f_z},\ \text{minimumSize}(z))$
- factors f_x, f_y, f_z is specified with QLayout::addWidget
- Example: `rowHBL->addWidget( editED, 2 )`

Adding Stretchable Space



- Empty space is like widgets with no visual appearance
- $\text{size}(x) = \text{maximum}\ (\textit{width} \times \frac{f_x}{f_x+f_y+f_e},\ \text{minimumSize}(x))$
  $\text{size}(e) = \text{maximum}\ (\textit{width} \times \frac{f_e}{f_x+f_y+f_e},\ 0))$
  $\text{size}(y) = \text{maximum}\ (\textit{width} \times \frac{f_y}{f_x+f_y+f_e},\ \text{minimumSize}(y))$
- The factor f_e is specified using QLayout::addStretch
- Example: `rowHBL->addStretch( 2 )`

Size Policy

- The size policy describes the interest of a widget in resizing.
- A size policy consists of one policy per direction (horizontal and vertical).
- Available options for the policy are: `Fixed`, `Minimum`, `Maximum`, `Preferred`, `Expanding`, `MinimumExpanding`, and `Ignored`.
- `QWidget::setSizePolicy()` and `QWidget::sizePolicy()` set and query the size policy of a widget.

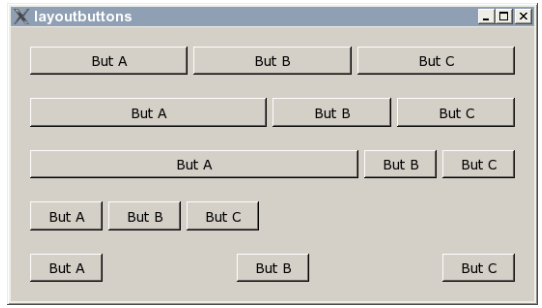
Available Size Policies

- **Fixed** - the `sizeHint()` is the only acceptable alternative.
- **Minimum** - the `sizeHint()` is minimal and sufficient. The widget can be expanded, but there is no advantage to it being larger.
- **Maximum** - the `sizeHint()` is a maximum. The widget can be shrunk by any amount without damage to its looks or functionality.

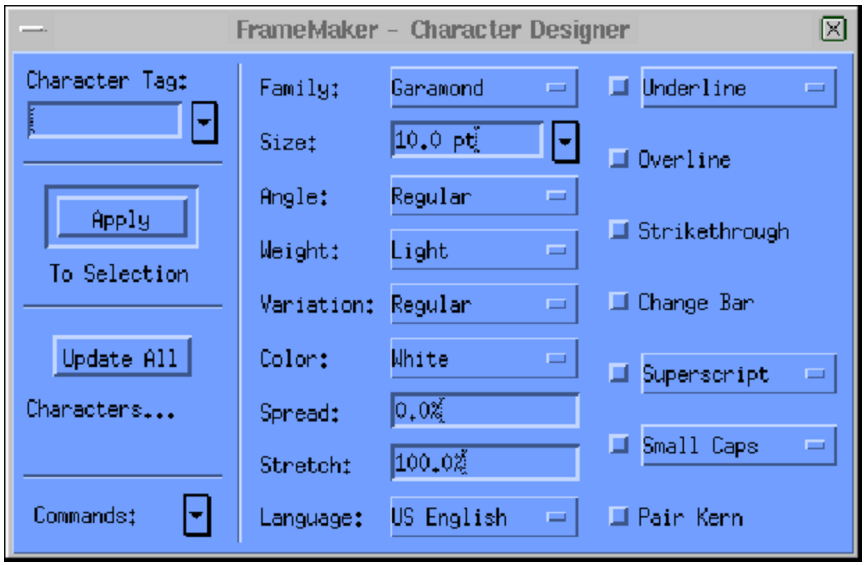
Available Size Policies cont'd.

- **Preferred** - the `sizeHint()` is best, but the widget can be shrunk below that and still be useful. The widget can be expanded, but there is no advantage to it being larger than `sizeHint()`.
- **MinimumExpanding** - the `sizeHint()` is a minimum. The widget can make use of extra space.
- **Expanding** - the `sizeHint()` is a sensible size, but the widget can be shrunk below that and still be useful.

Project Task: Develop the Following Layouts



Let's talk about GUI Bloopers



Questions And Answers

1. How do you change the minimum size a widget may have?
2. Name the available layout managers.
3. How do you specify stretch?
4. When are you allowed to call `resize` and `move` on a widget?

QFileDialog

- Lets the user select one or more files for loading or one filename for saving.
- Usually used via the static methods `QFileDialog::getOpenFileName()`, `QFileDialog::getOpenFileNames()`, `QFileDialog::getSaveFileName()`, and `QFileDialog::getExistingDirectory()`.
- Filename filters can be specified (with the static methods or `QFileDialog::setFilter()` resp. `QFileDialog::setFilters()`).

QColorDialog

- Lets the user select a color.
- Can only be used via the static methods `QColorDialog::getColor()` and `QColorDialog::getRgba()`.

QInputDialog

- One-line value entry.
- Only static methods: `getText()`, `getInteger()`, `getDouble()`, `getItem()`.

QFontDialog

- Can only be used via the static method `QFontDialog::getFont()`.

QMessageBox

- A general purpose message box.
- Icon, text and caption can be specified, as well as the number and labels of the buttons.
- Usually used via the static methods `QMessageBox::information()`, `QMessageBox::warning()`, `QMessageBox::critical()`, and `QMessageBox::question()`.

QProgressDialog

- A dialog for showing the progress of a lengthy operation.
- Set the amount of steps with `QProgressDialog::setMaximum()`.
- While the operation progresses, call `QProgressDialog::setValue()` to update the progress bar.
- While the operation progresses, check for cancel button clicks by either calling `QProgressDialog::wasCanceled()` or connecting to the signal `QProgressDialog::canceled()`.
- To react to the button, and to ensure repaints, you must call `QApplication::processEvents()` from time to time.
- NOTE: You must call `setValue(0)` before starting, otherwise the estimation of the duration will not work.

QErrorMessage

- QErrorMessage is basically a QMessageBox with a "*show this message again*" check box.
- You do not have to show or exec the dialog, it does that itself when needed.
- To show a message, call the slot `showMessage()`, which takes the string to show as an argument.
- Normally, the message is shown immediately, but if another message is currently shown, then the message is queued.
- For debugging purposes, call the static method `qtHandler()`, which will install an error handler showing `qDebug()`, `qWarning()` and `qFatal()` messages in a QErrorMessage box.

Custom Dialogs

- Inherit a class publically from `QDialog`
- Create the widgets as children, grandchildren, ... of the dialog object in the constructor of the class.
- Connect to the children's slots as necessary.
- Use the class `QDialogButtonBox` for the OK, Cancel, Apply, Help, ... buttons at the bottom of the dialog - this ensures the order is correct on different platforms.

An Example — Header File

```
#include <QDialog>
...
class MyDialog : public QDialog
{
  Q_OBJECT

public:
  explicit MyDialog( QWidget* parent = 0 );
  ...

private:
   QSomeWidget* _widget;
};
```


An Example — Implementation File

```
MyDialog::MyDialog( QWidget* parent ) : QDialog( parent )
{
  QLabel* label = new QLabel( "Some text" );
  QLineEdit* edit = new QLineEdit;
  ...
  QDialogButtonBox* buttons
    = new QDialogButtonBox( Ok | Cancel | ... );
  connect( buttons, SIGNAL(accepted()), this, SLOT(accept()));
  connect( buttons, SIGNAL(rejected()), this, SLOT(reject()));
  // put widgets into layouts...
}
```


OK/Cancel buttons

- Notice it is important you connect the OK and Cancel button to `accept()` and `reject()` rather than to say your own slotOK() and slotCancel().
- A consequence is of course that you must override `QDialog::accept()` and `QDialog::reject()`.
- The reason is that the window manager close button and the escape key are connected to `reject()`.
- See Example *dialog*

Modality

- `QDialogs` can be shown using `show()` or `exec()`.
- When you `exec()` a dialog, it will be modal.
- When you `show()` a dialog, it will normally be non-modal.
- It is possible to `show()` a modal dialog, by calling `QDialog::setModal(true)`.
- When *show*'d the function calling `show()` will continue right away, when *exec*'d the function calling `exec()` will not return until the dialog has been closed.
- Close the dialog using `QDialog::accept()` or `QDialog::reject()`.

Handling Deletion of Dialog

- How do you delete the dialog when you are done with it? Possible options are:
- Don't - create the dialog at startup and keep it around for ever.
- Don't - create the dialog when needed, and keep it around from then on.
- Call `QObject::deleteLater()`
- Use `setAttribute(Qt::WA_DeleteOnClose)`
- Override `closeEvent()`

Questions And Answers

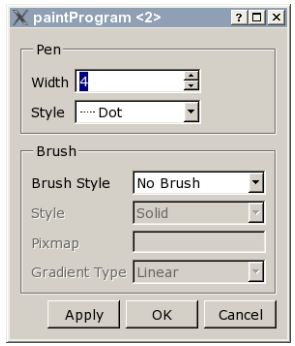
1. What is the problem with this code:

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

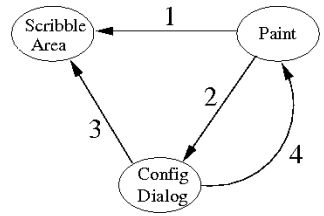
2. When would you use a modal dialog, and when would you use a non-modal dialog?
3. When should you call exec() and when should you call show()?
4. Can you bring up a modal dialog, when a modal dialog is already active?
5. When do you need to keep widgets as instance variables?

Project Task: Implementing a Configuration Dialog – part II

- Continue either from your solution on page 141 or from *solutions/config-dialog*
- Convert the widgets into a dialog, and hook it up to the paint program from *examples/paint-program/step3*
- Make sure that the values from the config dialog are sent to the scribble area and used there.



Signal/Slots in the Exercise



- The arrows show *who knows about whom*
- (1) and (2) are the only dependencies required when using signals/slots
- If ConfigDialog was given a ScribbleArea pointer in its constructor we would in addition have (3)
- If ConfigDialog called a method in Paint to make it read data and notify ScribbleArea we would have (4)

Qt Designer

- Qt Designer lets you design your GUI visually.
- Qt Designer saves files in an XML format.
- Another program, uic, converts these files into C++ source code.
- Forms are added to QMake files using the FORMS line:
  `FORMS = myform1.ui myform2.ui ...`

Using Generated Code

- `uic` generates a class in the namespace Ui, the name of the class is specified in Qt Designer's property editor.
- The class offers the method `setupUi(QWidget* widget)` which will add the Ui to the widget.
- There are three ways of using the generated class, *directly*, through *private inheritance*, or through *delegation*.

Direct Approach

- Advantage: Easy for simple exec'ed configuration dialogs.
- Disadvantage: Exposes implementation details.
- Disadvantage: No custom slots possible.
- Example:

```
QDialog dialog;
Ui_ConfigDialog conf;
conf.setupUi( &dialog );
if ( dialog.exec() ) {
    int min = conf.minimum->value();
}
```


Private Inheritance

- Advantage: Custom slots are no problem.
- Advantage: Simple and elegant, when concepts known.
- Disadvantage: Multiple and private inheritance *not* always well-known concepts.
- Disadvantage: Creates client dependency on Ui class.
- Example:

```
// --- configdialog.h ---
#include <QDialog>
#include "ui_configdialog.h"

class ConfigDialog :public QDialog,
                    private Ui_ConfigDialog {
...
```

Private Inheritance cont'd

```
// --- configdialog.cpp ---
ConfigDialog::ConfigDialog( QWidget* parent )
  : QDialog( parent )
{
  setupUi( this );
  minimum->setValue( 10 );
  ...
}
```


Delegation

- Advantage: Custom slots are no problem.
- Advantage: Removes client dependencies on Ui class.
- Disadvantage: Usage requires a bit more coding.
- Example:

```
// --- configdialog.h ---
#include <QDialog>
class Ui_ConfigDialog;

class ConfigDialog :public QDialog {
private:
  Ui_ConfigDialog * ui;
...
```


Delegation cont'd

```
// --- configdialog.cpp ---
#include "configdialog.h"
#include "ui_configdialog.h"
ConfigDialog::ConfigDialog( QWidget* parent )
  : QDialog( parent ), ui( new Ui_ConfigDialog )
{
  ui->setupUi( this );
  ui->minimum->setValue( 10 );
  ...
}
ConfigDialog::~ConfigDialog() {
  delete ui; ui = 0;
}
```


Connecting Signals/Slots

- All the widgets in the Ui are available as public members.
- In your code, you can of course set up signals/slots with these widgets.
- You can also connect signals/slots between widgets in Qt Designer.
- Finally, Qt will automatically connect signals from the Ui to slots in your code, based on the name of the widget, and the signal in question. The pattern is:
  `void on_`*`widgetName_signalName`*`(`*`signal-parameters`*`);`
- Ex: `on_ok_clicked()` for the `clicked()` signal from a button named `ok`.

Handling Your Own Classes

- Often, you want to use your own widgets in Qt Designer. In this case, you have two options:
  1. Choose the widget closest to yours that Qt Designer offers, and from its context menu choose *Promote to Custom Widget*. The code generated by Qt Designer will now refer to the widget with the class name you specify.
  2. Implement a plugin by inheriting the class `QDesignerCustomWidgetInterface`. See example `examples/designer/customwidgetplugin`

Properties for Widgets from Plugins

- If you specify some of the properties of your widget as being *Qt properties*, then you can edit these in the properties list view in Qt Designer
- A property is specified in the class declaration using the `Q_PROPERTY` macro
- Syntax:

```
Q_PROPERTY( type name READ getFunction
 [WRITE setFunction] [RESET resetFunction]
 [DESIGNABLE bool] [SCRIPTABLE bool] [STORED bool] )
```

- Example:

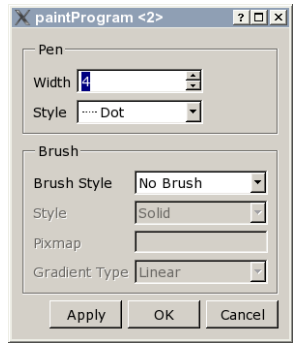
```
Q_PROPERTY( QString title READ title
                                    WRITE setTitle)
```


Properties for Widgets from Plugins cont'd

- The type specified in `Q_PROPERTY` can be anything that fits into a `QVariant`, or an enumeration defined inside the class.
- Enumerations must be registered using the `Q_ENUMS` macro.
- An alternative to enumerations are flags. A flag specifies a value, which can contain several enumerations or'ed together (e.g. READ|WRITE). Flags are specified using `Q_FLAGS`
-
```
class MyClass : public ... {
   Q_FLAGS( MODE )
   Q_PROPERTY( MODE mode READ mode WRITE setMode )
public:
   enum MODE { READ=1, WRITE=2 };
...
```


Project Task: Qt Designer

- Implement a config dialog for the paint program.
- Integrate the dialog with your paint program.
- Try out both methods of integration: delegation and inheritance.



Text Processing with `QString`

- Strings can be created in a number of ways:
  - Copy constructor and assignment operators:
    ```
    QString str("abc");
    str = "def";
    ```
  - From a number using the static method `number()`
  - From a char pointer using the static methods `fromLatin1()`, `fromUtf8()`, `fromLocal8Bit()`, ...

Text Processing with `QString` cont'd.

- Strings can be created from other strings:
  - Using `operator+` and `operator+=`:
    ```
    QString str = str1 + str2;
    fileName += ".txt";
    ```
  - A version with duplicate whitespace removed using `simplified()`
  - A part of a string using one of `left()`, `mid()`, and `right()`
  - Padded version using `leftJustified()` and `rightJustified()`:
    QString s = "apple";
    QString t = s.leftJustified(8, '.');// t == "apple..."

Text Processing with `QString` cont'd.

- Data can be extracted from strings using:
  - Numbers: `toInt()`, `toFloat()`, ...
  - String: `toLatin1()`, `toUtf8()`, `toLocal8Bit()`
- The later methods return a QByteArray, from which you can get a char* using `QByteArray::data()`

Text Processing with `QString` cont'd.

- Strings can be tested with:
  - `length()` returns the length of the string.
  - `endsWith()` and `startsWith()` test whether the string starts or ends with an other string.
  - `contains()` returns whether the string matches a given expression, `count()` tells you how many times.
  - `indexOf()` and `lastIndexOf()` search for the next matching expressions, and return its index.
- The expression can be a sequence of characters, strings, or regular expressions.

Text Processing with `QStringList`

- `QString::split()` and `QStringList::join()` are used to split a string or join several strings together using a specified substring.
- `replaceInStrings()` performs a search/replace on all strings in a list.
- `filter()` returns a list containing those items from a list which match a string or regular expression.

URLs

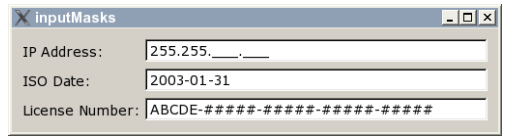
- URLs are handled by `QUrl`.
- QUrl include convenience methods for constructing URLs from its components (user, password, protocol, etc.), and methods for splitting a URL into its components.
- The class `QUrlInfo` provides info about the content the URL points to similar to what `QFile` does for local files.
- Using the methods in the `QDesktopServices` namespace, it is possible to *open* URLs, and configure how URLs are opened.

Working with Regular Expressions

- `QRegExp` supports both regular expression matching as well as shell globbing.
- Most regular expression features are supported.
- `QRegExp` objects can also be used for searching strings.
- `QRegExp` offers text capturing regular expressions, where a subexpression can be extracted using `QString cap(int)` and `QStringList capturedTexts()`.

QLineEdit's input masks

- Input masks provide a way of specifying what the input in a line edit should look like.
- This is useful for formatted text like IP numbers, license numbers etc, which must match a given pattern.
- Using masks, you can also specify a placeholder letter to be used in lieu of the space character.
- Masks are set using `QLineEdit::setInputMask()`.



Input Masks in QLineEdit cont'd.

- An input mask is a string with a token for each of the characters the user must type, interspersed with normal text (e.g. a period to separate octets in IP numbers)
- Tokens can be escaped with a backslash if they are meant as normal letters.
- The input mask can optionally end with a semicolon followed by a character. This character is then the character used in place of an empty space.

Input Masks in QLineEdit cont'd.

char set	require	optional
a-z,A-Z	A	a
a-z,A-Z,0-9	N	n
0-9	9	0
1-9	D	d
0-9,+,-		#
any	X	x

>	All following alphabetic characters are uppercased.
<	All following alphabetic characters are lowercased.
!	Switch off case conversion.

Examples: `000.000.000.000;_` `>AAAAA-AAAAA-AAAAA;#`
(see *examples/inputMasks*)

Validating Input

- It is possible to validate input for a `QLineEdit`, `QSpinBox` or a `QComboBox` with `QValidator` subclasses.
- This is especially useful for text, but can also be used for verifying correct number entry (hint: use `QSpinBox` for integer numbers).
- Set a validator with `QLineEdit::setValidator()`.
- Predefined validators: `QIntValidator`, `QDoubleValidator`, and `QRegExpValidator`.
- To define your own validators, override `QValidator::validate()` and optionally `QValidator::fixup()`

Project Task: Swedish Person Identification Numbers

- In 1947 Sweden introduced, as the first country in the world, a person identification number that every citizen have. The number consist of 10 digits of which the first six are the persons birthday (format DDMMYY) and the last is a checksum on the others. The number resembles to some degree the American social security number.
- Starting from the handout *social-security-number*, add first an input mask, and second a vaildator that ensure that day and month are valid, plus that the checksum computes.
- Optional: set font and size of the line edit, and implement fixup.
- For details on the Swedish personal identity number, see:

`http://en.wikipedia.org/wiki/Personal_identity_number_(Sweden)`

Resources

- When shipping an application it might be useful to compile any extra files needed into the application, to avoid that the user deletes those files from the hard disk.
- In this context, any such files are referred to as a resource.
- The resources can be anything that would otherwise be accessed from disk as an ordinary file.
- Examples include icons, images, audio files, translations, etc.

Specifying Resources

- Resources are listed in a *resource collection file*.
- Add a line similar to the following to your qmake file:
  `RESOURCES = application.qrc`
- `qmake -project` will add all `.qrc` files found.

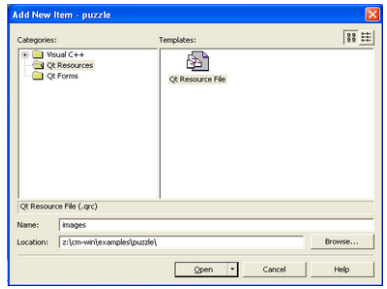
An Example Resource Collection File

- The following is an example resource collection file:

```
<!DOCTYPE RCC><RCC version="1.0">
<qresource>
    <file>images/bird.png</file>
    <file>images/dog.png</file>
    <file>sounds/bird.wav</file>
    <file>sounds/dog.wav</file>
    <file>resources_da.qm</file>
</qresource>
</RCC>
```


Resource Collection File in Visual Studio

- In Microsoft Visual Studio, you may add a resource collections from the solution explorer sidebar, by right clicking and choosing `Add->Add New Item`.
- This will create the *xxx*`.qrc` file for you. Open it, and right click to add a resource.



Accessing Resources

- Resources can be accessed from the application in a way similar to normal files. The only difference is that each file name is prefixed with `:/`
- In the above, the file which was located in `images/bird.png` on disk can be accessed as the file name `:/images/bird.png`
- That is really the only difference from normal file access.

Example of Resource Usage

- `QFile`:

```
QFile file( ":/sounds/bird.wav" );
```

- `QPixmap`:

```
_pixmap = QPixmap( ":/images/bird.png" );
```

- `QTranslator`:

```
QTranslator* translator = new QTranslator;
translator->load( ":/resources_da.qm" );
```


Aliases

- If you prefer to access your resources under a different name than the ones used on disk, you can use aliases:

```
<!DOCTYPE RCC><RCC version="1.0">
<qresource>
    <file alias="bird.png">images/bird.png</file>
    <file alias="dog.png">images/dog.png</file>
    <file>sounds/bird.wav</file>
    <file>sounds/dog.wav</file>
    <file>resources_da.qm</file>
</qresource>
</RCC>
```


Internationalization

- Different resources can be used for different languages.
- As an example, in the Western world, a cross is used to indicate death, in an Arabic culture this may at best not be understood, and at worst be offensive.
- When specifying resources, it is possible to specify a locale in the `qresource` element.
- Aliases are required to map the names on top of what they replace.

Internationalization Example

```
<!DOCTYPE RCC><RCC version="1.0">
<qresource>
    <file alias="bird.png">images/bird.png</file>
    <file alias="dog.png">images/dog.png</file>
    <file>sounds/bird.wav</file>
    <file>sounds/dog.wav</file>
    <file>resources_da.qm</file>
</qresource>
<qresource lang="da_DK">
    <file alias="sounds/bird.wav">sounds/bird-da.wav</file>
    <file alias="sounds/dog.wav">sounds/dog-da.wav</file>
</qresource>
</RCC>
```


Miscellaneous

- It is of course allowed to have multiple resource collection files.
- If you have the resources in a static library, you might need to force initialization by calling `Q_INIT_RESOURCE(`*xxx*`)`, where `xxx` is replaced with the basename of the resource collection file.
- The resource tree can be traversed using `QDir` pointing to `:/`, as a normal directory structure.

Project Task: Resources

- Try running the paint program where your current directory is different from the one containing the paint program.
- Modify the paint program to compile its icons into the binary.

Specifying Help

- Help can be made available in a number of different forms:
  - Tool tips. (Set using `QWidget::setToolTip()`.)
  - Status bar tips. (Set using `QWidget::setStatusTip()`.)
  - What's This help. (Set using `QWidget::setWhatsThis()`.)
  - System Tray Icon.
  - User manual. (May be viewed using Qt Assistant.)

System Tray Icon

- An entry may be added to the system tray using the class `QSystemTrayIcon`.
- Notice, this feature may not be available with certain window managers. Check with `isSystemTrayAvailable()` to check if it is.
- A context menu may be set using `setContextMenu()`.
- A message may be shown at any time using `showMessage()`. Not all window managers may support this (Mac OS X, and KDE doesn't e.g.)
- See example *system-tray*

Dynamic Help

- Sometimes, it is only possible to "calculate" the tool tip dynamically.
- This can be achieved by overriding `QWidget::event()`, and listening to events with the type `QEvent::ToolTip`, `QEvent::StatusTip`, or `QEvent::WhatsThis`.
- The events for these are instances of the class `QHelpEvent`
- To display the actual message use `QToolTip::showText()`, `QWhatsThis::showText()`, or `QStatusBar::showMessage()`.
- See the example *dynamic-help*.

QAssistantClient

- Using `QTextBrowser`, it is very easy to build a documentation browser as can be seen in the upcoming exercise.
- It is, however, also possible to use Qt Assistant as documentation browser.
- Your documentation needs to be augmented with a content file in this case.
- Once the above is in place you can start Qt Assistant using `QAssistantClient::openAssistant()`

Project Task: Documentation Browser

- Write an online help browser, starting from the text editor from previous projects.
- Selecting a word that is a Qt class name and pressing F1 should bring up a help browser (`QTextBrowser` window) that shows the Qt documentation for that word (as found in `doc/html/`).
- Hint: `QTextCursor` handles text selection in a QTextEdit.
- This exercise is about an in-process help window, do not use QAssistantClient.

Synthetic Events

- You can make a widget believe that it gets events from the underlying system by posting or sending events to it.
- Events posted to a widget are put into the event queue and handled during the next pass through the event loop.
- Events sent to a widget are handled right away.
- You post an event using `QCoreApplication::postEvent()`, and send one using `QCoreApplication::sendEvent()`.
- Both methods take the widget to send the event to as the first argument and the event as the second.

Synthetic Events cont'd.

- When calling `postEvent()`, the event must be allocated using `new`, and must **not** be deallocated (Qt takes ownership).
- When calling `sendEvent()`, you must take care of deleting the instance afterwards (or allocate it on the stack).
- You can create your own events by inheriting `QEvent`, and using an integer between `QEvent::User` (1000) and `QEvent::MaxUser` (65535).
- Qt does of course not understand user events. You must therefore handle user events in inherited classes implementing `QObject::customEvent()` to deal with your custom events.
- Alternatively, event filters can be used.
- See example *syntheticEvents*.

Delayed Invocation

- Sometimes, it is desirable to have some code invoked when the application is idle.
- One way to do this is to post custom QEvent and put your code in the `QObject::customEvent()` handler of the receiving object.
- Another way that is often easier and works when the code you want to invoke is available as a slot, `QMetaObject::invokeMethod()` with the connection type `QueuedConnection` can be used instead:

```
QMetaObject::invokeMethod( myobject,
                           "doDelayedStuff",
                           Qt::QueuedConnection);
```


Delayed Invocation cont'd

- A typical use of this is to make a delayed call from a constructor. This can be used to emit signals after the object has been created and connected.
- Another use is idle processing.
- An alternative way of handling idle processing is to call `QEventLoop::processEvents()` from your idle routine.
- See example *delayedInit*

Event Filters

- Sometimes you need to add some functionality to a number of different widgets (reacting on a certain event).
- The usual way to do this is to subclass each widget and implement the event. This is cumbersome if all you try to obtain is the possibility to e.g. add mouse movement handling to widgets.
- The alternative is to install an event filter for each instance.

Event Filters cont'd.

- Subclass from QObject (or any subclass of QObject), and reimplement the method:

  ```
  bool eventFilter(QObject* receiver, QEvent* event)
  ```

- The first parameter is the object for whom the event was intended, and the second argument is the event itself.
- If this method returns true, then the event is considered "handled", otherwise it will be sent on to the next event filter, or the object itself, if no more event filters are installed.
- Install the event filter for an object by invoking the method `installEventFilter()`. As the argument, pass an instance of the class you have created in the previous step.

Event Filters cont'd.

- Installing an event filter on the QApplication instance will install a global event filter.
- Event filters can be removed again by using the method `removeEventFilter(const QObject*)`
- When multiple event filters are installed, the order they are called in is the reverse of the order in which they are installed, i.e., the most recent installed filter is the first one invoked.
- See example *eventFilter*

Project Tasks: Rework the Event Filters to be installed on QApplication

- Rework the event filter from the previous example to install on `QApplication` instead of installing on each widget in turn.
- Hint: You need to add a method like `HelpFilter::registerHelp()`.
- Optional: Make sure that widget gets removed from the map when deleted.

Container Classes

- Introduction to container classes
- List of container classes
- Requirements on items in container classes
- Iterators
- Algorithms
- Implicitly shared classes

Introduction

- The Qt library provides a set of general purpose template-based container classes.
- These container classes are designed to be lighter, safer, and easier to use than the STL containers.
- You may want to use these classes if you are unfamiliar with the STL, or prefer to do things "the Qt way".
- If you are already familiar with the STL, feel free to continue using it.
- Methods exist that convert between Qt and STL containers, e.g. if you need to pass the contents of a `std::list` to a Qt method.

Sequence Containers

- **QLinkedList**<**T**> A doubly linked list. Optimized for inserting items in the middle (cf. `std::list`).
- **QVector**<**T**> A list optimized for random access (cf. `std::vector`).
- **QList**<**T**> An array list; good default container (cf .`std::deque`).
- **QStack**<**T**> This is a convenience subclass of `QVector` that provides "last in, first out" (LIFO) semantics (cf. `std::stack`).
- **QQueue**<**T**> This is a convenience subclass of `QList` that provides "first in, first out" (FIFO) semantics (cf. `std::queue`).

Associative Containers

- **QSet**<**T**> A single-valued mathematical set with fast lookups (cf. `std::set`).
- **QMap**<**K,T**> A dictionary (associative array) that maps keys of type `K` to values of type `T` (cf. `std::map`).
- **QMultiMap**<**K,T**> A variant of `QMap` that supports maps where one key can be associated with multiple values (cf. `std::multimap`).
- **QHash**<**K,T**> A variant of `QMap` using hash tables to perform lookup and insertion in constant time (cf. `std::tr1::unordered_map`).
- **QMultiHash**<**K,T**> A variant of `QHash` that supports hashes where one key can be associated with multiple values (cf. `std::tr1::unordered_multimap`).

Comparison of Containers

	Lookup	Insert	Append	Prepend
QLinkedList	O(n)	O(1)	O(1)	O(1)
QVector	O(1)	O(n)	O(1)	O(n)
QList	O(1)	O(n)	O(1)	O(1)

Complexity | Worst Case	Lookup		Insert	
Q(Multi)Map	O(log(n))		O(log(n))	
Q(Multi)Hash	O(1)	O(n)	O(1)	O(n)

(all complexities are amortized)

Requirements on Items in Container Classes

- For a class to be stored in one of the Qt containers it must be an *assignable data type*.
- A class is *assignable*, if it provides all of these:
  - a default constructor
  - a copy constructor
  - an assignment operator
- If a copy constructor or an assignment operator is not provided, C++ will provide a version that does member by member copying.
- If you do not provide *any* constructors, an empty default constructor will be provided by C++.

Requirements on Items in Container Classes cont'd.

- To use a type `K` as a key for `QHash`, the following additional functions must exist:
  - `bool K::operator==( const K& )` or `bool operator==( const K&, const K& )`
  - `uint qHash( const K& )`
- To use a type `K` as a key for `QMap` or with a `QSet`, you must instead implement:
  - `bool K::operator<( const K& )` or `bool operator<( const K&, const K& )`
- See the example *containers*.

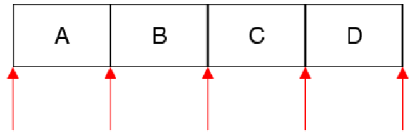
Iterators

- There are iterators for traversing the values of the containers.
- Qt has two kinds of iterators:
  - **Java-style iterators** simple and easy to use.
  - **STL-style iterators** slightly more efficient and can be used together with Qt's and STL's generic algorithms.
- In addition a `foreach` macro exists which allows you to iterate over all the values in a container.

Java-style Iterators

Containers	Read-only Iterator	Read-write Iterator
QList<T> QQueue<T>	QListIterator<T>	QMutableListIterator<T>
QLinkedList<T>	QLinkedListIterator<T>	QMutableLinkedListIterator<T>
QVector<T> QStack<T>	QVectorIterator<T>	QMutableVectorIterator<T>
QSet<T>	QSetIterator<T>	N/A
QMap<Key,T> QMultiMap<Key,T>	QMapIterator<Key,T>	QMutableMapIterator<Key,T>
QHash<Key,T> QMultiHash<Key,T>	QHashIterator<Key,T>	QMutableHashIterator<Key,T>

Java-style Iterators



Java style QList

```
QList<QString> list = ...;
QListIterator<QString> it(list);
```

Forward

```
while (it.hasNext())
  qDebug() << it.next();
```

Backward

```
it.toBack();
while (it.hasPrevious())
  qDebug() << it.previous();
```


QListIterator Methods

Function	Description
`toFront()` `toBack()`	Moves the iterator to the front of the list (before the first item) Moves the iterator to the back of the list (after the last item)

`hasNext()`
`next()`
`peekNext()`
	Returns true if the iterator is not at the back of the list
	Returns the next item and advances the iterator by one position
	Returns the next item without moving the iterator

`hasPrevious()`
`previous()`
`peekPrevious()`
	Returns true if the iterator isn't at the front of the list
	Returns the previous item and moves the iterator back by one position
	Returns the previous item without moving the iterator

`findNext(const T&)` `findPrevious(const T&)`	Searches forward for the specified element. Searches backwards for the specified element.

`QLinkedList`, `QVector`, and `QSet` have the same API.

Modifying During Iteration

- If you want to modify the items while you iterate, you need to use the *mutable* versions of the iterators, e.g. `QMutableListIterator`.
- Methods like `remove()` and `setValue()` operate on the items just jumped over using the previous `next()` or `previous()` call.
- `insert()` inserts an item at the current position in a sequence (remember that iterators point between items), and leaves the iterator *after* the item. Thus, a call to `previous()` will see the item that was just inserted.

Iterating Over QMap and QHash

- When iterating over `QMap` and `QHash`, the movement methods like `next()` and `previous()` will return an instance of an Item class which has the two methods `key()` and `value()` to get the info from the current item.
- Alternatively, the `key()` and `value()` methods from the iterator can be used. These methods return information about the item just passed by the iterator.

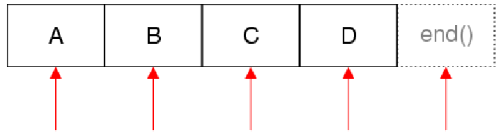
Iterating QMap example

```
QMap<QString, QString> map;
map.insert("Paris", "France");
map.insert("Guatemala City", "Guatemala");
map.insert("Mexico City", "Mexico");
map.insert("Moscow", "Russia");

QMutableMapIterator<QString, QString> i(map);
while (i.hasNext()) {
    if (i.next().key().endsWith("City"))
        i.remove();
}
```


STL-style iterators

Containers	Read-only Iterator	Read-write Iterator
QList<T> QQueue<T>	QList<T>::const_iterator	QList<T>::iterator
QLinkedList<T>	QLinkedList<T>::const_iterator	QLinkedList<T>::iterator
QVector<T> QStack<T>	QVector<T>::const_iterator	QVector<T>::iterator
QSet<T>	QSet<T>::const_iterator	N/A
QMap<Key,T> QMultiMap<Key,T>	QMap<Key,T>::const_iterator	QMap<Key,T>::iterator
QHash<Key,T> QMultiHash<Key,T>	QHash<Key,T>::const_iterator	QHash<Key,T>::iterator

STL-style Iterators




```
QList<QString> list;
list << "A" << "B" << "C" << "D";
QList<QString>::iterator i;
```

Forward

```
for (i = list.begin();
     i != list.end(); ++i)
    *i = (*i).toLower();
```

Backward

```
i = list.end();
while (i != list.begin()) {
    --i;
    *i = (*i).toLower();
}
```

The foreach Keyword

- Qt contains a `foreach` keyword (technically, it is of course a macro, but it looks and feels like a keyword).
- The syntax is:
  `foreach` (*variable*, *container*) *statement*
- The type of the variable can be specified along with the variable.
- `break` and `continue` can be used in the statement.
- Modifying the container while iterating results in the container being copied, and iteration continuing in the unmodified version.
- When iterating using foreach it is not possible to modify the items themself, as the iterator variable is a const reference.

foreach Example

```
QLinkedList<QString> list;
...
```

foreach

```
foreach (QString str, list) {
    if (str.isEmpty())
        break;
    qDebug() << str;
}
```

Java-style iterator

```
QLinkedListIterator<QString>
   it(list);
while (i.hasNext()) {
    QString str = it.next();
    if ( str.isEmpty() )
        break;
    qDebug() << str;
}
```


Algorithms

- One of the nice things about the STL-style iterators is that they are compatible with the STL algorithms, that are defined in the STL <algorithm> header.
- Qt comes with its own sets of algorithms in the include file <QtAlgorithms>
- If STL is available on all your supported platforms then you may choose to use to the STL algorithms as the collection is much larger than the one in Qt.

Algorithms cont'd.

- **qSort** sorts items in a range.
- **qStableSort** similar to sort, but guarantees that the order between two equal items stays unchanged.
- **qFind** searches for a value.
- **qEqual** tells if two ranges are the same.
- **qCopy** copies items from one range to another.
- **qCopyBackward** copies items backwards.
- **qCount** counts the number of items matching search criteria.

Algorithms Example

```
QList<int> list;
list << 3 << 3 << 2 << 2 << 7 << 2 << 8;

int countOf2;
qCount(list.begin(), list.end(), 2, countOf2);
// countOf2 == 3
```


Algorithms Example

```
QList<QString> list;
list << "one" << "two" << "three";
QVector<QString> vect1(3);

qCopy(list.begin(), list.end(), vect1.begin());
// vect: [ "one", "two", "three" ]
```


Algorithms Example

```
bool caseInsensitiveLessThan(const QString& s1,
                             const QString& s2)
{
    return s1.toLower() < s2.toLower();
}

int doSomething()
{
    QList<QString> list;
    list << "AlPha" << "beTA" << "gamma" << "DELTA";
    qSort(list.begin(), list.end(), caseInsensitiveLessThan);
   // list: [ "AlPha", "beTA", "DELTA", "gamma" ]
}
```


Implicitly Shared Classes

- The following classes are, among a number of others, implicitly shared: QByteArray, QHash, QIcon, QImage, QLinkedList, QList, QMap, QMultiHash, QMultiMap, QPalette, QPixmap, QPointArray, QQueue, QRegExp, QRegion, QString, QStringList,QVariant, QVector.
- *implicitly shared* means that if an object is copied, then its data is copied *only when* the data of one of the objects is changed.
- Implicit sharing is especially important when inserting items into a container, or when returning a container.

Qt Debugging Aids

- You can output debugging messages with `qDebug()`, `qWarning()`, `qCritical()`, and `qFatal()`. `printf`-style parameters and stream output can be used:

```
qDebug( "Method computed: %d", myIntVariable );
qDebug() << "Mouse was clicked at " << mouseEvent->pos();
```

- Setting the environment variable QT_FATAL_WARNINGS makes the application terminate on warnings.
- Connecting a signal and a slot incorrectly is *only* shown if Qt is compiled with debug information.
- Defining QT_NO_DEBUG_OUTPUT and/or QT_NO_WARNING_OUTPUT during compilation of your application, turns the messages in question off.

Qt Debugging Aids cont'd.

- `Q_ASSERT` is like the good old `assert`, except that you can compile it out by compiling Qt with the QT_NO_DEBUG flag.
- `Q_ASSERT_X()` is an assert that lets you specify a message to output on assertion failure.
- To show a QString in MSVC++ you need to insert the following code into the file `...\Program Files\Microsoft Visual Studio 8\Common7\Packages\Debugger\Autoexp.dat` (adjust for your Visual Studio installation):

```
; from Qt
QString=<d->data,su>
```


Printing QString in GDB

- Insert the following code into your `~/.gdbinit`, and you can do `qs str` (where str is a QString):

```
define qs
    set $i=0
    set $d = $arg0.d
    while $i < $d->size
        printf "%c", (char)($d->data[$i++] & 0xff)
    end
    printf "\n"
end
```

- `kdesdk/scripts/kde-devel-gdb` contains many more of these functions.

Qt Debugging Aids cont'd.

- You can pass a name to each Qt object with `setObjectName()`.
- This name can be retrieved with `QObject::objectName()`.
- As a lot of the debugging information is only really helpful if these names are set, it is good Qt programming style to do so.
- `QObject::dumpObjectInfo()` dumps information about object internals, like signals/slots.
- `QObject::dumpObjectTree()` dumps the parent/child relationships of all descendant widgets.

Shoot a Bug

- Installing a global event filter can help you get information about your widgets.
- Questions which can be answered include: "Which children does this widget have?" and "How far does this widget stretch?"
- `#include "shootabug.h"` and insert the following into your main:

```
qApp->installEventFilter( new ShootABug() );
```


Shoot a Bug

```
class ShootABug :public QObject
{
  Q_OBJECT
public:
  bool eventFilter( QObject* recv, QEvent* event )
  {
    if ( event->type() != QEvent::MouseButtonPress )
      return false; // pass it on
    QMouseEvent* mevent = static_cast<QMouseEvent*>(event);
    if ( (mevent->modifiers() & Qt::ControlModifier ) &&
         (mevent->button() & Qt::LeftButton) ) {  // Ctrl + left mouse click.
      qDebug("-----------------------------------------------------");
      qDebug("Widget name : %s", qPrintable( recv->objectName() ) );
      qDebug("Widget class: %s", recv->metaObject()->className() );
      qDebug("\nObject info:");
      recv->dumpObjectInfo();
      qDebug("\nObject tree:");
      recv->dumpObjectTree();
      qDebug("-----------------------------------------------------");
      return true; // Block
    }
    return false;
  }
};
```


Qt Debugging Aids cont'd.

- On X11 systems, the command-line switch `-nograb` prevents Qt from grabbing the mouse and keyboard and thus prevents the dreaded "debugger lock-ups".
- On Linux, Qt even detects if it is running from within gdb and automatically turns off grabbing. This can be overridden with `-dograb`. (Careful!)
- Qt also supports the command-line switch `-sync` which makes all Xlib calls synchronous.

Some Thoughts About Portability

- Portability can give you a larger market share.
- Even if you think you have an inherently unportable product (like a device driver), there might still be some portable parts in it (like a configuration screen).
- Portable programs tend to be less buggy because they can unveil boundary conditions on one platform that rarely arise on another (e.g., race conditions, uninitialized memory, etc.).

How to Make Your Programs Portable

- Avoid calling native API functions. The speed gain is usually minimal (or non-existent), and all the often used functionality is covered by Qt. In particular, avoid calling native graphical API functions. Calling OpenGL/Open Inventor graphics functions is OK, though, if you have an integration kit.
- When designing file formats, consider using XML or at least `QDataStream`. For portable images, use `QPixmap`, for portable vector graphics, use `QPicture`.
- If you cannot avoid using non-portable functions, isolate platform-dependent code in separate files. Try to keep the header files portable (at the cost of some #ifdef directives, if necessary).

Portability Pitfalls in Qt Itself

- Beware of bad compilers; check the platform-specific release notes at `http://www.trolltech.com/developer/compilers/`
- All methods that return internal, platform-specific data like `QCursor::handle()`, `QFont::handle()`, and `QWidget::winId()` are inherently dangerous. Calling the methods is in itself not unportable, but using the returned data most likely is.
- *Warning:* `Qt::HANDLE` is *not* a portable type.

Portability Pitfalls in Qt Itself cont'd.

- Some features are simply not available on all platforms. These include `QApplication::flushX()` (use `QCoreApplication::flush()` instead), some of the flags used for `QDir::setFilter()`, and `QWidget::setSizeIncrement()` (not supported on Windows).
- `QApplication::setColorSpec()` is portable in itself, but the acceptable values have different meanings on different platforms.
- The granularity of timers (`QObject::startTimer()` and classes `QTime` and `QTimer`) is 55 milliseconds on Windows 98/ME (one millisecond on Windows NT/2000/XP, Unix, MacOS).

Portability Pitfalls in Qt Itself cont'd.

- On Unix, some more command-line switches are filtered out by Qt. This includes `-geometry` which automatically sets the size of the main window and `-sync` which makes Xlib calls synchronous.
- `QDir::convertSeparators()` converts the Qt standard for directory separators (slashes) to backslashes on Windows (and does nothing on Unix). Native Windows functions on modern Windows versions can deal with slashes as well, however, so it is advisable to always use forward slashes.

Building Projects Portably

- *QMake* is your friend — even if you only build for one platform.
- If you cannot or do not want to use *QMake*, try to devise a build scheme, where you do not list the source and header files more than once.
- There are commercial *make* utilities available that support both Windows- and Unix-style makefiles. Besides, Cygnus has a free GNU make.
- Qt defines the following macros telling you which system you are currently compiling on: `Q_WS_X11`, `Q_WS_WIN`, `Q_WS_MACX`, and `Q_WS_QWS`. (Notice that technically, this is the window system, not the operating system. There are also `Q_OS_*` macros, but the window system macros are usually more relevant.

64 bit Issues

- Do not rely on specific data types having a given size, use one of the following instead: `qint8`, `quint8`, `qint16`, `quint16`, `qint32`, `quint32`, `qint64`, `quint64`.
- Do not cast pointers to numbers, it is unportable. (Assuming that a pointer fits into a `quint64` is probably safe, but still done at your own risk.)
- When streaming using `QDataStream`, always use `qint8`, `qint16`, ... rather than `short`, `int`, `long`, ...
- `QFile` is 64-bit clean and supports therefore "large files".
- `Q_INT_64_C()` and `Q_UINT_64_C()` allow you to specify 64-bit integer constants, e.g. `Q_INT_64_C(123456789123456)`.

QSignalMapper

- Beginners often try to do the following:

```
connect( button, SIGNAL( clicked() ),
         this, SLOT( action(7) ) );
```

- THIS IS WRONG! connect is only a tool for *connecting*, it is not possible to pass arguments to the slot when connecting.
- The above can however be obtained using the class `QSignalMapper`.

QSignalMapper cont'd.

```
QPushButton* but1 = new QPushButton("Button 1", this);
QPushButton* but2 = new QPushButton("Button 2", this);
QSignalMapper* mapper = new QSignalMapper( this );
mapper->setMapping( but1, 1 );
mapper->setMapping( but2, 2 );
connect(but1, SIGNAL(clicked()), mapper, SLOT(map()));
connect(but2, SIGNAL(clicked()), mapper, SLOT(map()));
connect(mapper, SIGNAL(mapped(int)), this, SLOT(action(int)));
```

QTimer

- `QTimer` is a class for executing functions at a later time.
- You create an instance of a `QTimer` and connect the signal `timeout()` to a slot containing the code you want to be executed.
- Call setSingleShot() for a single-shot timer.
- Finally, call `start(int msec)` on the timer to start it.
- For a one-time non-cancellable single-shot timer: `QTimer::singleShot(1000, this, SLOT(doit()))`

QTimer - Example (stopwatch)

```
StopWatch::StopWatch( QWidget* parent ) : QLabel( parent )
{
  QTimer* timer = new QTimer( this );
  timer->start( 1000 );
  connect( timer, SIGNAL( timeout() ),
           this, SLOT( shot() ) );
}

void StopWatch::shot()
{
  _secs += 1;
  QString str;
  str.sprintf("%d:%02d", _secs / 60, _secs % 60);
  setText( str );
}
```


Find it in the Source!

- Does QObject emit a signal or sends an event when user invokes `setObjectName()`
- `QAbstractItemView::EditTrigger` includes `QAbstractItemView::EditKeyPressed` with the description *Editing starts when an edit key has been pressed over an item.* Which key is that?
- Does `QMenu::addAction(QString)` take ownership so that `QMenu::clear()` will delete the action after use?

Find it in the Source!

- Is there any way you can get your string retranslated when the language changes? (Hint: look at `QCoreApplication::installTranslator()`, and trace code from there).
- Tracing the above code down to the widgets, can you tell which other changes you may react to?
- `QProgressDialog` has a facility for not showing the progress dialog if the action will finish within a certain amount of time (default 4 seconds). Imagine you use this dialog, and it shows the dialog, even if your progress finishes faster than the limit, what can you have done wrong (note: two things can go wrong).

Steps to Your Own Widget

- Check whether the widget you want to write does not exist yet. Most do.
- Decide on a base class. Often used base classes are `QWidget` and `QFrame`
- Reimplement the event handlers that are needed for your widget. This will almost always be `QWidget::mousePressEvent()` and `QWidget::mouseReleaseEvent()`, if your widget accepts keyboard input also `QWidget::keyPressEvent()` and if it changes appearance when the widget gets focus `QWidget::focusInEvent()` and `QWidget::focusOutEvent()` as well.

Steps to Your Own Widget cont'd.

- Reimplement `QWidget::paintEvent()` to draw your widget's visual appearance. This will most often be dependent on some internal state variables.
- Decide which signals you want to emit. Emitting signals will usually happen from within event handlers, especially `mousePressEvent()` or `mouseDoubleClickEvent()`.
- Decide carefully on the types of the signal parameters. The more general types you choose, the more likely your widget will be reusable. Good candidates are `bool`, `int` and `const QString&`.

Steps to Your Own Widget cont'd.

- Decide which internal state variables of the widget should be made publically accessible and implement accessor methods.
- Decide which of the setter methods should be slots. All methods with integral or common parameters are good candidates.
- Decide whether you want to make it possible to subclass your widget. If yes, decide which methods to make protected instead of private and which methods to make virtual.

Steps to Your Own Widget cont'd.

- Decide which parameters can be set at construction time and augment the constructor as necessary. Or implement more than one constructor. If a parameter is needed for the widget to work correctly, the user of the class should be forced to pass it in the constructor.
- Keep the Qt convention with:
  `explicit Constructor(..., QWidget* parent = 0 )`

Project Task: Implementing a Compass Widget

- Implement a "compass widget" that lets the user graphically select a direction (north, west, south, east, northwest, southwest, southeast, northeast, and optionally none).
- Make it possible to select the direction with the keyboard.
- Make it possible to change the direction programmatically from the classes using your widget.
- Make it possible for the classes using your widget to get informed whenever the user changes direction.

Optional Section

- QSound
- QImage
- QSettings
- Clipboard + Drag and Drop
- Network Programming
- QProcess
- ActiveQt
- Migrating Motif programs to Qt
- Qt Script for Applications
- QWorkspace
- QGraphicsView
- Model/View Programming
- QScrollArea
- OpenGL
- QTextEdit
- Customized Drawing
- Widget Styles
- QMake
- International-ization
- XML
- Multithreading
- SQL
- Plug-ins
- Development Tools for Linux
- Licensing
- Shipping Qt
- Unit Testing With QTestLib

QSound

- QSound provides cross-platform sound support.
- On Microsoft Windows, the underlying multimedia system is used; only the WAV format is supported for the sound files.
- On X11, the Network Audio System is used if available, otherwise all operations work silently. NAS supports WAV and AU files.
- On the Macintosh, all QuickTime-supported formats are supported.
- In Qtopia Core, a built-in mixing sound server is used, which accesses /dev/dsp directly. Only the WAV format is supported.

QSound cont'd.

- The easiest way to play sound is simply to use the static method `play( QString )`, taking a file name to play.
- You can also create an instance of `QSound` giving the file to play in the constructor.
- When using an instance, methods `play()` and `stop()` allows you to start and stop playing.
- `play()` returns immediately
- Depending on the platform audio facilities, other sounds may be stopped or mixed with the new sound.

QSound cont'd.

- To test if a sound is being played, use `isFinished()`.
- Using an instance of QSound, you can ask Qt to play the sound a number of times using `setLoops( int )`.
- When playing, test how many loops remain using `loopsRemaining()`
- On X11, QSound only works if NAS is installed. To test if it is installed, use `QSound::isAvailable()`

Limitations

- No signal is emitted when playing is done.
- It is not possible to access the mixer.

QImage

Introduction to the QImage Class

QImage Agenda

- Introduction to QImage
- QImage capabilities
- Using QImage
- Qt imaging stack
- Supporting custom images
- Custom image example

QImage Class

- Inherited from QPaintDevice
- Hardware independent
- Image storage class
- Direct access to pixel data
- Stored in main memory
- Uses implicit data sharing

QImage Capabilities

- Creating new images
- Reading and writing image files
- Managing image information
- Direct pixel manipulation
- Image transformations

Using QPainter

- QImage is a QPaintDevice
- QPainter can be used for:
  - Drawing basic shapes and lines
  - Rendering QImages and QPixmaps
  - Applying drawing transforms
  - Setting drawing coordinates
  - Enforcing clipping regions

Using QImage

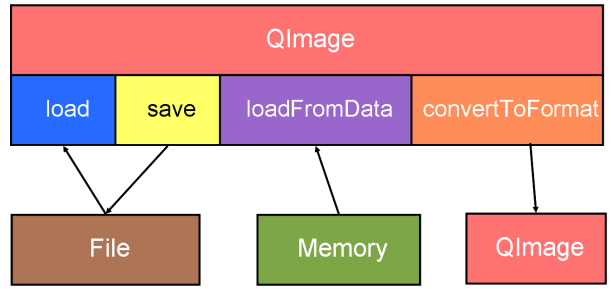
- Create a null image
- New blank image (size, format)
- From raw data (byte array, size, format)
- From a file (file name, format*)
- Copy an existing QImage (source QImage)

**Note: usually derived from the filename extension*

Reading and Writing Images

- Reading from a file: load
- Creating from raw data: loadFromData()
- Writing to a file: save
- Output other pixel formats: convertToFormat()
- Can support future formats using plug-ins, in addition to a number of standard formats

Reading and Writing Images

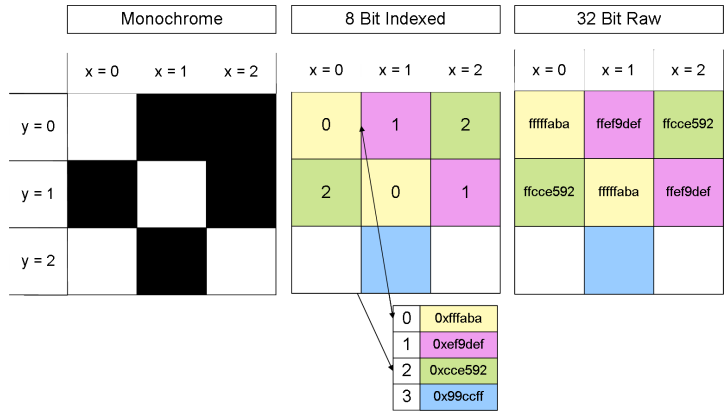


QImage Standard Formats

- Standard load formats:
  - BMP, GIF*, JPEG, PNG, PBM, PGM, PPM, XBM, XPM
- Standard save formats:
  - BMP, JPG, JPEG, PNG, PPM, XBM, XPM

**Note: Qt must be configured and compiled to support GIF*

QImage Pixel Representations



QImage Information

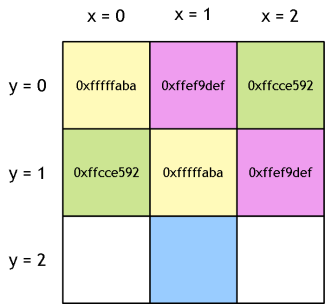
- Image
  - size, depth, format, bytes / lines, number of bytes, unique serial number
- Colors
  - pixels, color table, table size, transparency
- Text (such as JPEG 2000)
  - text format, keyed strings

Pixel Manipulation

- Set or read individual pixels
  - pixel(), setPixel(), and pixelIndex()
- Access to entire scan lines
  - scanLine() or bits()
- Add colors to the color table
  - color(), setColor(), colorTable(), setColorTable()

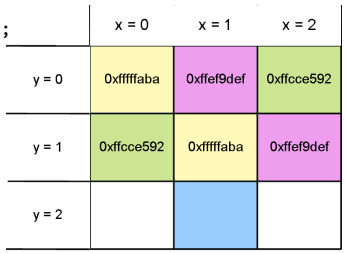
Direct Pixel Manipulation

```
QImage image(3, 3,
   QImage::Format_RGB32);
image.fill( qRgb(255,255,255) );
//white fill
QRgb value;
value = qRgb(0xff, 0xfa, 0xba);
image.setPixel(0, 0, value);
image.setPixel(1, 1, value);

value = qRgb(0xef, 0x9d, 0xef);
image.setPixel(1, 0, value);
image.setPixel(2, 1, value);

value = qRgb(0xcc, 0xe5, 0x92);
image.setPixel(2, 0, value);
image.setPixel(0, 1, value);
```



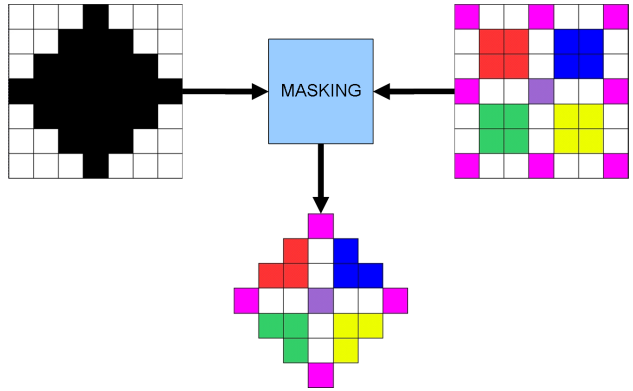
Using Indexed Colors

```
QImage image(3, 3, QImage::Format_Indexed8);
image.fill( qRgb(255,255,255)  );
//white fill

QRgb value;

value = qRgb(0xff, 0xfa, 0xba);
image.setColor(0, value);
value = qRgb(0xef, 0x9d, 0xef);
image.setColor(1, value);
value = qRgb(0xcc, 0xe5, 0x92);
image.setColor(2, value);

image.setPixel(0, 0, 0);
image.setPixel(1, 0, 1);
image.setPixel(2, 0, 2);
image.setPixel(0, 1, 2);
image.setPixel(1, 1, 0);
image.setPixel(2, 1, 1);
```



QImage Transformations

- Masking: image *appears* to be irregularly shaped
- Mirroring: a mirror of an original QImage
- Swapping: red and blue colors are exchanged
  - a RGB image becomes a "BGR" rendering of the original image
- Transparency: the QImage has translucent pixels
- Transforming: reformulates each pixel's x and y location based on a linear equation to: scale, rotate, and shear the image



Mirror Image

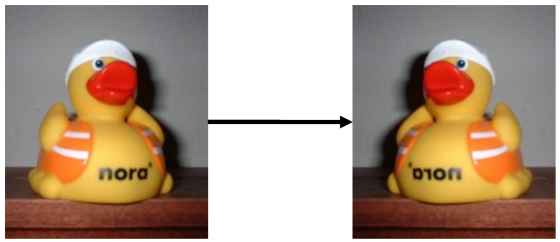
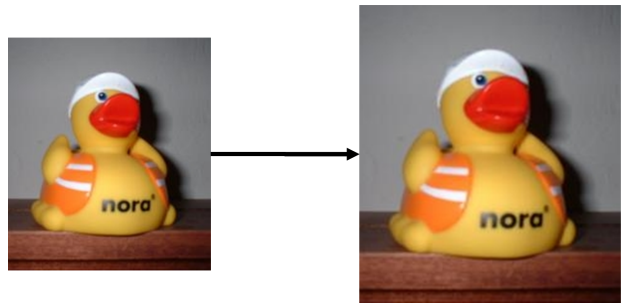
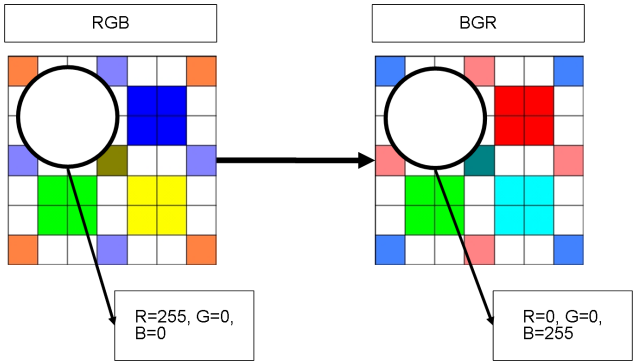


Image Scaling



Color Swapping



Transparency

Provides per color alpha channel

- 8 bit value (0 is transparent, 255 opaque)
- setAlphaChannel()
  - creates an alpha channel from a second QImage
- alphaChannel()
  - alpha channel is returned as a QImage
- hasAlphaChannel()
  - returns true if alpha channel exists

Transparency

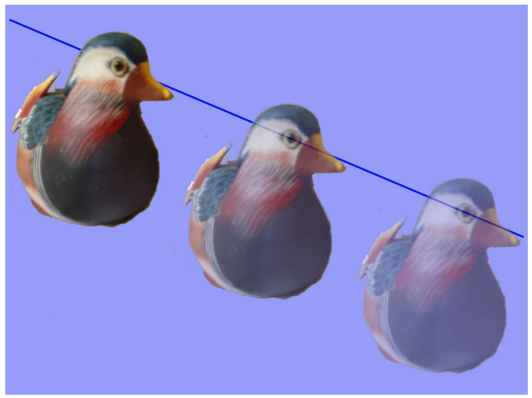


Image Transformations

Creating Masks

Image mask created from alpha channel

- createAlphaMask()
  - converts from alpha channel
  - argument *flags* control the conversion process
- createHeuristicMask()
  - generates a mask that clips away the "background color"

Custom Image Formats

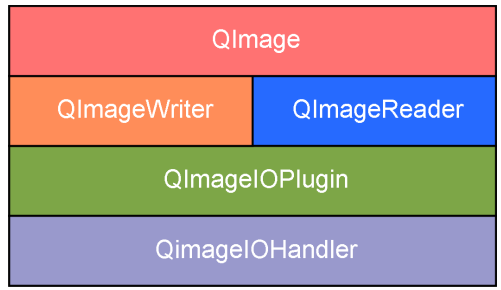
- Qt supports many image file formats by default:
  - BMP, JPEG, PBM, PNG, PPM, XBM, XPM
  - and GIF (if compiled in)
- QImage supports these color representations:
  - 8 bit index based image data (color map)
  - 16 and 32 bit raw RGB and ARGB data, pre-multiplied ARGB (fastest for painting)
  - monochrome (color table)

Goal of Custom Image Formats

Support arbitrary image file formats using the QImage *load()* method or its constructor:

```
QImage myImage("someImage.png");
myImage.load("someOtherImage.jpg");
myImage.load("someOtherImage.xyz");
```


Qt's Imaging Stack



The Qt Imaging I/O Stack

- QImage
  - format independent image storage
  - provides methods for loading, saving, and format conversion
- QImageReader and QImageWriter
  - format independent interface
  - for reading / writing images from files or other devices
  - uses both built-in and plug-in image handlers

The Qt Imaging I/O Stack, cont.

- QImageIOPlugin
  - extends the readable image file types
  - provides a well defined plug-in shared library interface
- QImageIOHandler
  - provides common I/O interface
  - defines new image formats
  - dynamically loads new formats via the plug-in
  - built on Qt plug-in classes as a shared library

New Image Type Handlers

In order to enable `QImage` support for new file formats, we need to:

- Create our own subclass of `QImageIOHandler`
- Make it available as a `QImageIOPlugin` so that previously created applications can also be extended without modification *or even rebuilding*

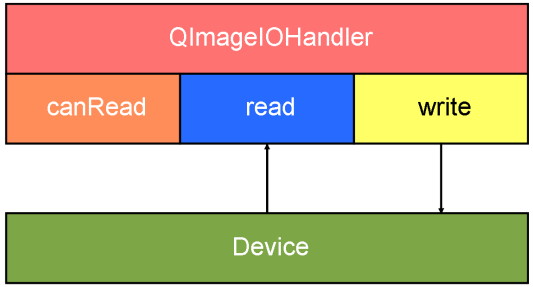
Introducing "SIF"

- For the purposes of our example, we have defined a "simple image format" (SIF) whose file layout looks something like:

```
SIF
3 3
000 000 000 255 000 000 000 255 000
255 255 255 255 255 255 255 255 255
128 128 128 128 128 128 128 128 128
```

- The ASCII text, white space delimited file format is as follows:
  - file extension of ".sif"
  - first token is the string "SIF" to identify it as a simple image file
  - second and third tokens are the image's width and height
  - succeeding tokens represent each pixel by specifying their RGB (red green blue) components as intensities ranging from 0 - 255

Basic QImageIOHandler



Subclassing QImageIOHandler

To create our own subclass of `QImageIOHandler`, we override at least these three methods:

```
class SimpleImageIOHandler : public QImageIOHandler
{
public:
     virtual bool canRead() const;
     virtual bool read(QImage *image);
     virtual bool write(const QImage &image);
};
```


canRead Method

bool **canRead()**

- Returns true if an image can be read from the device
- File format is supported
  - extension indicates the image format
  - header indicates a supported format
- Device is capable of performing reads

canRead() Example Code

```
bool SimpleImageIOHandler::canRead()
{
   return device ? device->peek(3) == "SIF" : false;
}
```

Write Method

bool **write** *(const QImage &image)*

- Writes the image to the assigned device
- Returns true on success
- Acts as input data generator to the *read()* method
- Iterates over all the pixels to be saved, and arranges for them to be encoded into the file's proper format

write() Example Code

```
bool SimpleImageIOHandler::write(const QImage & image) {
     QTextStream stream( device() );
     int width = image.width();
     int height = image.height();
     stream << "SIF" << endl << height << endl
        << width << endl;
     for (int y=0; y<height; y++) {
        for (int x=0; x<width; x++)  {
            QRgb rgb = image.pixel(x, y);
            stream << qRed(rgb) << " "  << qGreen(rgb)
               << " "  << qBlue(rgb) << " ";
            }
     stream << endl;
     }
     return true; }
```


Read Method

bool **read** *(QImage * image)*

- Reads an image from the device
- Stores it into the address pointed at by **image*
- Decodes the file's data
- converts image to QImage pixel representation

read() Example Code

```
bool SimpleImageIOHandler::read(QImage * image) {
     QString formatLine;
     int width, height, r, g, b;
     QTextStream stream( device() );
     stream >> formatLine >> height >> width;
     // demo code, so works for RGB32 only
     *image = QImage(width,height, QImage::Format_RGB32);
     for( int y=0; y<height; y++ ) {
          for( int x=0; x<width; x++ ) {
               stream >> r >> g >> b;
               image->setPixel(x,y, qRgb(r,g,b) );
          }
     }
     return true; }
```

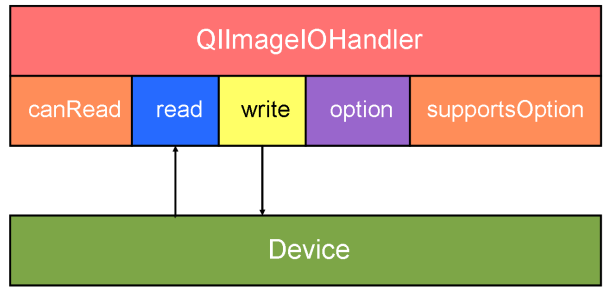
Advanced QImageIOHandler

Advanced methods for:

- Handling animation
- Reformatting the image during save/load calls, enabling compression, and toggling endianness
- Image formatting options to allow scaling and clipping

The new enhanced version of class definition

Advanced QImageIOHandler



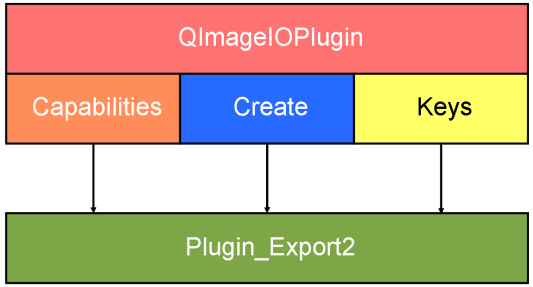
QImageIOHandler Declaration

```
class MyIOHandler : public QImageIOHandler {
public:
    MyIOHandler(QIODevice *device);
    virtual bool canRead() const;
    virtual bool supportsOption(ImageOption option) const;
    virtual QVariant option(ImageOption option) const;
    virtual bool read(QImage * image);
    virtual bool write(const QImage & image);

    // Convenience
    static bool canRead(QIODevice *device);
};
```


Advanced Methods

- supportsOption()
  - includes: compression, scaling, etc...
- option()
  - returns values for *every* supported option
- canRead()
  - static version of canRead()
  - *this will be very useful for the plugin*

QImageIOPlugin



QImageIOPlugin Interface

```
class MyImageIOPlugin : public QImageIOPlugin
{
public:
     virtual Capabilities capabilities(QIODevice * device,
        const QByteArray & format) const;

     virtual QImageIOHandler* create(QIODevice * device,
        const QByteArray & format = QByteArray()) const;

     virtual QStringList keys() const;
};
```


Create Method

QImageIOHandler ***create**(QIODevice * device,const QByteArray &format)

- Creates and returns an instance of a **QImageIOHandler** subclass, with device and format set
- The format must come from the list returned by keys()
- Format names are case sensitive

Keys Method

QStringList **keys***()*

- Returns the list of image keys this plugin supports
- These keys are usually the names of the image formats that are implemented in the plugin (e.g., "jpg" or "gif")
- Must return all supported formats
  - "BMP", "TIFF"..."MYFORMAT"

Capabilities Method

Capabilities **capabilities**(QIODevice * device,const QByteArray &format)

- Returns the types of functionality that this plugin can supply for a given format
  - read, write, incremental read or none
- Be careful to check both the device and the format
  - either could be invalid

Export the Plugin

```
Q_EXPORT_PLUGIN2( PluginName, ClassName )
```


Specify the Plugin to QMake

```
TEMPLATE = lib
TARGET += MyImageFormat
DESTDIR = $(QTDIR)\plugins\imageformats
CONFIG += plugin
HEADERS += SimpleImageIOHandler.h \
           SimpleImagePlugin.h
SOURCES += SimpleImageIOHandler.cpp \
           SimpleImagePlugin.cpp
```


Project Task: Creating an Image Handler

- Write a QImageIOHandler that will support the SimpleImage format
  - an easy format that is really only good for lab
  - RGB text based format
- Write a QImageIOPlugin that will wrap the above QImageIOPlugin
- Test
  - with supplied main.cpp and simpleImage.sif file

Simple Image Format I

Example of 3x3 Image

- First line is the format header
- Second line is the width and height

```
SIF
3 3
000 000 000 255 000 000 000 255 000
255 255 255 255 255 255 255 255 255
128 128 128 128 128 128 128 128 128
```


Simple Image Format II

- 3 text integers per pixel (RGB)
  - First line = Black, Red, Green
  - Second line = White, White, White
  - Third line = Grey, Grey, Grey
- One line = width pixels
- Data portion is height lines long

QSettings

- QSettings saves configurations of the type `KEY=VALUE`.
- On Windows, it uses the registry database by default. It can also use ini-style text files.
- On UNIX, it uses ini-style text files in a predefined directory.
- On MacOS X, it uses the Preferences API.
- On all platforms, custom formats can also be used.
- Write an entry using `setValue()`. All value types supported by `QVariant` can be used.
- Read back a value using `value()`. Use the type-conversion methods in `QVariant` or `qVariantValue<>`.
- You can specify a default value that is returned when the key is not found. If you do not specify a default value, and the key is not found, an invalid variant is returned.

Keys

- A key is a unicode string which consists of one or more subkeys.
- A subkey is a slash followed by a number of unicode characters.
- Examples:
  - `/background color`
  - `/foreground color`
  - `/geometry/x`
  - `/geometry/y`
  - `/geometry/width`
  - `/geometry/height`

Keys cont'd

- The first two parts of the key can be anything you like, but it has become a good practice (mandated by Qt 3) that you use your Internet domain and the name of the application.
- You specify these when creating the QSettings object: `QSettings mySettings( "KDAB", "MyApplication" );`
- You can set defaults for these with `QCoreApplication::setOrganizationName()` and `QCoreApplication::setApplicationDomain()`.
- MacOS X uses the domain name of an organization instead of the display name; you can either set it with `QCoreApplication::setOrganizationDomain()`, or have a fake one derived from the organization name.

Keys cont'd

- If you want to make several settings to the same path beyond the organization and application, you can enclose the commands with `beginGroup()` and `endGroup()`:

```
QSettings mySettings( "KDAB", "MyApplication" );
mySettings.beginGroup( "Renderer" );
// Sets /Renderer/HighSpeed
mySettings.setValue( "HighSpeed", true );
mySettings.endGroup();

// Sets /HighSpeed
mySettings.setValue( "HighSpeed", true );
```


Key Management

- Entries can be deleted using `remove()`.
- Groups at a certain path can be found using `childGroups()`.
- Leaf keys at a certain path can be found using `childKeys()`.
- In the example from two slides back, the following is returned from `childGroups("/")`: `geometry`
- In the example from two slides back, the following is returned from `childKeys("/")`: `background color`, `foreground color`
- `allKeys()` combines `childGroups()` and `childKeys()`. On MacOS X, this will even return other system-wide, read-only keys.

Search Paths

- `QSettings` will search for a certain key in the following locations in this order:
  1. a user-specific, application-specific location
  2. a user-specific, organization-wide location
  3. a system-wide, application-specific location
  4. a system-wide, organization-wide location
- You can turn this mechanism off with `setFallbacksEnabled(false)`.

Search Paths

- Keys are always written to the most specific location possible.
- You can further control the search algorithm by passing `QSettings::SystemScope` or `QSettings::UserScope` as the first, optional, parameter of the `QSettings` constructor.
- On Unix, the base directory for user-specific settings is taken from the environment variable `XDG_CONFIG_HOME` and defaults to *$HOME/.config*

Reading and Writing Non-Core Types

- Core types (`QString`, etc.) are supported by `QVariant` out-of-the-box with constructors for `QVariant` objects and conversion functions of the name `to`*`Type`*`()`.
- `QVariant`-supported non-core types (`QColor`, etc.) have conversion operators for creating the variants, but no conversion functions for converting back. Use `qVariantValue<>` in this case:

```
QFont textFont =
 qVariantValue<QFont>( settings.value( "TextFont" ) );
```


Multithreading and Multiprocessing

- `QSettings` objects are lightweight and can be cheaply created and discarded as needed. Therefore, it is rarely useful to create a `QSettings` object on the heap.
- Changes made to one `QSettings` object in one thread are visible to other `QSettings` objects in other threads immediately.
- `QSettings` merges entries between different processes. Changes from one process will not be visible, however, before either `sync()` has been called, or the `QSettings` object is deleted.

Limitations

- There are certain limitations on the Windows registry to be aware of.
- The length of a subkey must not exceed 255 characters.
- The size of a value must not exceed 16.300 characters
- All the values of a key may not exceed 65.535 characters.
- If these limitations are a problem for you, you should probably use XML for configuration files anyway.
- But you can force ini-style settings on Windows by specifying `IniFormat` in the `QSettings` constructor.

The System Clipboard

- Access to the system clipboard is possible via the class `QClipboard`.
- There is only one `QClipboard` object per application which can be accessed with `QApplication::clipboard()`.

Clipboard Dataformats

- The following table shows methods from the class `QClipboard` for reading/writing the types in the first column.

Type	read	write
QString	text()	setText()
QPixmap	pixmap()	setPixmap()
QImage	image()	setImage()
QMimeData*	mimeData()	setMimeData()

- In the following section, we will discuss `QMimeData` in detail.

Steps to Implementing Drag and Drop

- Detect that a drag is to be started.
- Start the drag.
- Detect a drop attempt and decide whether it can be accepted.
- React on the actual drop.
- Implementation of classes handling drag and drop data.

The Drag Side

- Drag-and-drop operations are usually initiated in response to certain mouse events, e.g., moving the mouse a specified number of pixels with mouse button pressed down.
- The class `QDrag` is the central part of starting the drag.
- Specify the data to drag using `QDrag::setMimeData()`.
- Start the actual drag using `Qt::DropAction QDrag::start(Qt::DropActions)`.
- Both the `QDrag` instance and the `QMimeData` instance must be allocated using `new` - Qt takes ownership of both.

The Drag Side cont'd.

```
/** Decide if a drag should be started **/
bool MyWidget::shouldStartDrag( QMouseEvent* event )
{
   return
     ( event->buttons() & Qt::LeftButton &&
        (_pressPos - event->pos()).manhattanLength() >
            QApplication::startDragDistance());
}
```


The Drag Side cont'd.

```
void MyWidget::mouseMoveEvent( QMouseEvent* event )
{
  if ( shouldStartDrag(event) ) {
      QDrag* drag = new QDrag( this );
      QMimeData *mimeData = new QMimeData;
      mimeData->setText( dataToDragOut() );
      drag->setMimeData( mimeData );
      Qt::DropAction dropAction = drag->start( ... );
      if ( dropAction == Qt::MoveAction ) {
          // remove the data from my site
      }
  }
  MyBaseClass::mouseMoveEvent( event );
}
```


The Drag Side cont'd.

- The following table shows the `QMimeData` methods for setting and reading data.
- You can set multiple data items, by calling multiple *set* functions.

Type	Qt Type	read	write	test
text	QString	text()	setText()	hasText()
HTML	QString	html()	setHtml()	hasHtml()
color	QColor	colorData()	setColorData()	hasColorData()
images	QVariant	imageData()	setImageData()	hasImage()
urls	QList <QUrl>	urls()	setUrls()	hasUrls()
data	QByteArray	data()	setData()	hasFormat()

The Drop Site

- Any widget that wants to receive drops must call `setAcceptDrops(true)`.
- In addition, the widget *must* at least reimplement `dragEnterEvent()` and `dropEvent()`.
- In addition the widget *may* reimplement `dragMoveEvent()` and `dragLeaveEvent()`.

The Drop Side cont'd.

- In the events, the widget can get at the `QMimeData`, through the event pointer using the method `mimeData()`.
- To check if the widget can decode the data in the event, it can call the *has* methods from page 357.
- In `dragEnterEvent()` and `dragMoveEvent()`, the widget must call `accept()` on the event pointer if it can accept the drag in question.

The Drop Side cont'd.

- In the `dropEvent()`, the widget should call `QDropEvent::acceptProposedAction()` if it can accept the drop as specified in `QDropEvent::proposedAction()`
- Proposed actions being one of `CopyAction`, `MoveAction`, and `LinkAction`.
- Alternatively, if it can handle one of the `QDropEvent::possibleActions()`, it should call `QDropEvent::setDropAction()`.

The Drop Side cont'd.

```
void MyWidget::dragEnterEvent( QDragEnterEvent* event ) {
    if ( event->mimeData()->hasText() )
        event->accept();
}

void MyWidget::dropEvent( QDropEvent* event ) {
    const QMimeData* data = event->mimeData();
    QString txt = data->text();
    event->acceptProposedAction();
}
```


Transferring Your Own Data

Transferring your own data

- `QMimeData` allows you to set custom data using `QMimeData::setData ( const QString& mimetype, const QByteArray& data )`
- The mimetype is a tag describing what the data represents. A lot of standard types exist, e.g. `text/plain`, `image/gif`, `video/mpeg`, and `audio/x-wav`
- The easiest way to get data into—and out of—a `QByteArray` is using `QDataStream` or `QTextStream`.
- Lots of Qt classes support `QDataStream`: `QString`, `QColor`, `QFont`, `QPixmap`, `QPoint`, ...

Example Implementation - Drag Site

```
void MyWidget::mouseMoveEvent( QMouseEvent* event )
{
  if ( shouldStartDrag(event) ) {
      ...
      QByteArray data;
      QDataStream stream( &data, QIODevice::WriteOnly );
      stream << _col1 << _col2 << (int) _orientation;

      QMimeData *mimeData = new QMimeData;
      mimeData->setData( "x-gizmo/x-drag", data );
      ...
  }
}
```


Example Implementation - Drop Site

```
void MyWidget::dragEnterEvent( QDragEnterEvent* e ) {
    if ( e->mimeData()->hasText() ||
         e->mimeData()->hasFormat( "x-gizmo/x-drag" ) ) {
        e->accept();
    }
}
```


Example Implementation - Drop Site

```
void MyWidget::dropEvent( QDropEvent* event ) {
    QMimeData* mimeData = event->mimeData();
    if ( mimeData->hasFormat( "x-gizmo/x-drag" ) ) {
        QByteArray data = mimeData->data( "x-gizmo/x-drag" );
        QDataStream stream( data );
        stream >> _col1 >> _col2 >> (int&) _orientation;
        event->acceptProposedAction();
    }
    else if ( mimeData->hasText() ) {
       QString txt = data->text();
       ...
       event->acceptProposedAction();
    }
}
```


Data on the Fly

- If your object offers data in many different formats, then it might be inefficient to generate all the formats at the time the drag starts.
- What you want instead is to only generate the type requested at the time the drop is performed.
- This can be done by inheriting `QMimeData`, and reimplementing
  - `QStringList formats()`
  - `QVariant retrieveData( const QString& mimetype, QVariant::Type type )`

Data on the Fly cont'd.

- If the drop is within the same application as the drag, the `QMimeData` instance given to the drop site is the one created during drag.
- This allows you to subclass `QMimeData`, and store your own data with the subclass.
- On the drop side, simply use `qobject_cast` to check if it actually is your subclass.

Project Task

- Implement drag and drop for the gizmo widget in *handout/drag-and-drop*. Following these steps will make your work easier:
  - Implement dragging text out of the gizmo object. (drop to the terminal window (on UNIX/MacOS) or to Wordpad (on Windows) to test it.) Simply drag the text "hello world".
  - Add drop functionality to the Gizmo object. Let it accept text, and just print out the dropped text.
  - Now add a *x-gizmo/x-drag* mime type encoded with `QByteArray`.

Project Task cont'd.

- Optional Tasks:
  - Add a drag pixmap showing the data being dragged (see `QDrag` class for details)
  - Inherit `QMimeData` to encode your data on the fly.
  - Implement cut-and-paste for the Gizmo object.

Abstraction Levels

- Qt supports network programming on two different levels.
- On the higher level, `QHttp` and `QFtp` support downloading and uploading of data to HTTP and FTP servers.
- On the lower level, Qt supports communicating with TCP and UDP sockets by means of `QTcpSocket`, `QTcpServer`, and `QUdpSocket`.
- In order to use the network support, add `QT += network` to your QMake files.

QFtp/QHttp

- `QHttp` and `QFtp` provide non-blocking client-side HTTP and FTP support.
- Both work with so-called *requests* (called *commands* in FTP context) that are queued first, and executed when the event loop is reached.
- Each request has a *command ID*. These IDs are passed in the signals that indicate starting and finishing requests.

Feedback About QFtp/QHttp Operations

- The simplest feedback is the end of the series of requests (one "download" or "upload" operation completed). This is indicated with the `done()` signal..
- For more detailed feedback, connect to the signals `QHttp::requestStarted()`, `QHttp::requestFinished()`, `QFtp::commandStarted()`, and `QFtp::commandFinished()`.
- Feedback about data upload/download progress is provided with the signals `QHttp::dataReadProgress()`, `QHttp::dataWriteProgress()`, `QFtp::dataTransferProgress()`.
- A general signal `stateChanged()` informs about meta state changes like closing connections.

Example: HTTP get

The simplest way to download a file via HTTP:

```
QIODevice* target = new ... // could be QFile, QBuffer, etc
QHttp* http = new QHttp();
http->get(url, target);
connect( http, SIGNAL( done() ),
         this, SLOT( slotDownloadFinished() ) );
/* do not delete target and http until
   slotDownloadFinished() is called */
```


Example cont'd

- If you have structured data, remember that you can use `QTextStream` and `QDataStream` on the target `QIODevice`.
- If you do not want to use a `QIODevice`, connect to the `readyRead()` or `done()` signal and read the data with `read()` or `readAll()`.
- You can do the same for FTP with `QFtp::get()`; an optional third parameter allows to specify binary (default) or text mode.
- Similar methods `QHttp::post()` and `QFtp::put()` are provided for uploading.

TCP Sockets

- TCP connections of protocols other than HTTP and FTP can easily be implemented using `QTcpSocket` (or `QTcpServer`), even for the HTTP and FTP.
- UDP connections are implemented using `QUdpSocket`.
- TCP and UDP connections can be written in a non-blocking and a blocking fashion.
- `QTcpSocket` and `QUdpSocket` are (indirect) subclasses of `QIODevice`. You can therefore read and write data using `QTextStream` and `QDataStream` (and the raw byte-oriented methods, of course).

Steps for Connecting to a TCP Server (Non-Blocking)

- Create an instance of `QTcpSocket`.
- Call `QAbstractSocket::connectToHost()`
- Create a stream from the socket, and write data to the stream.
- The signal `readyRead()` is emitted whenever data is ready to be read on the socket.
- See the example *sockets*.

Steps for Connecting to a TCP Server (Blocking)

- The socket classes also have a blocking mode. In this case, a local event loop is provided, you do not need a global one.
- Do not use the blocking mode from the GUI thread, otherwise your UI will freeze. Use it from a separate thread (see the *Multithreading* section on page 678 for more details), or in non-GUI applications (e.g., command-line tools like *wget*).
- For a blocking connect, call `QAbstractSocket::waitForConnected()` *after* the call to `connectToHost()`.
- Then call either `waitForReadyRead()` or `waitForBytesWritten()`, and read or write as usual.
- Call `disconnectFromHost()` and `waitForDisconnected()` at the end.

Steps in Setting up a TCP Server (Non-Blocking)

- Non-blocking TCP servers need an event loop (like non-blocking TCP clients).
- Create an object of class `QTcpServer`.
- Call `listen()` on that object. You can either specify the port to listen to or let `QTcpServer` pick a free one. `serverPort()` will tell you the one it is using.
- When a connection is made, the `newConnection()` signal is emitted. Upon this, call `nextPendingConnection()` to get a `QTcpSocket` that is already connected to the client, and that you can then use for communication.

Steps in Setting up a TCP Server (Blocking)

- Like blocking TCP clients, blocking TCP servers should not be run in the GUI thread, as that would freeze the UI. Use a separate thread or use this in command-line tools.
- Start like in the previous example, but call `waitForNewConnection()` after the `listen()` call. When this method returns, you can call `nextPendingConnection()` and proceed as with the non-blocking server.

Steps in Setting up a UDP Connection

- Create a `QUdpSocket` object and send data with `writeDatagram()`. You need to specify the destination host and port each time.
- You can use `connectToHost()` and then just use `read()` and `write()`, but this is still not connection-oriented, it just sets the values for host and port for the subsequent calls.
- If you want to read data as well, you need to call `bind()`, specifying the port and host address. Like `QTcpServer::listen()`, `QUdpSocket::bind()` can pick a free port for you.
- The signal `readyRead()` is emitted when data is available for reading.

Resolving Hostnames

- `QHostInfo` serves for resolving host names.
- `QHostInfo` uses the underlying `gethostbyname()` call and therefore supports all lookup schemes that the operating system supports (this could e.g. be NIS).
- In order to do a blocking lookup, call the static method `QHostInfo::fromName()` and use the `addresses()` method to get all addresses for the named host:

```
QHostInfo info = QHostInfo::fromName( "www.trolltech.com" );
QList<QHostAddress> addresses = info.addresses();
if( addresses.count() > 0 )
  qDebug() << "First address: "
           << addresses.first().toString();
```


Resolving Hostnames cont'd

- Do not do this in the GUI thread, or your UI will freeze.
- For a non-blocking lookup, call `QHostInfo::lookupHost()`, providing the receiver object and slot to be called when the lookup is finished (see the example on the next slide).
- You can abort a host lookup using `abortHostLookup()`.

Resolving Hostnames: Example

```
MyClass::startLookup()
{
    QHostInfo::lookupHost( "www.trolltech.com", this,
            SLOT( slotLookupDone( const QHostInfo& ) ) );
}

MyClass::slotLookupDone( const QHostInfo& info )
{
    QList<QHostAddress> addresses = info.addresses();
    if( addresses.count() > 0 )
        qDebug() << "First address: "
                 << addresses.first().toString();
}
```


Network Interfaces

- Using the class `QNetworkInterface`, it is possible to obtain information about the host's IP addresses and network interfaces.

Proxies

- Proxies can be used with the class `QNetworkProxy`.
- In order to use a proxy, create a `QNetworkProxy` object and populate it with hostname, port, etc.
- Then assign the proxy either globally with the static method `QNetworkProxy::setApplicationProxy()` or for one socket using `setProxy()`.
- Besides normal proxying, `QNetworkProxy` also supports SOCKS5.

Error Handling

- `QHttp` and `QFtp` indicate errors by means of a boolean parameter in the `done()`, `QHttp::requestFinished()`, and `QFtp::commandFinished()` signals.
- `QTcpSocket` and `QUdpSocket` indicate errors by means of the `error()` signal.
- `QTcpServer` and `QHostInfo` have no notification mechanism for errors.
- In any of these classes, you can then request an error code with `error()` (`serverError()` for `QTcpServer`).
- `errorString()` provides (for all classes) a human-readable error description.

Project Tasks

- Implement a TCP server that listens on port 4242.
- The server should have the user interface shown here.
- A new window should appear for each accepted connection.
- Use either blocking or non-blocking networking;
  optionally try the other alternative as well.

QProcess

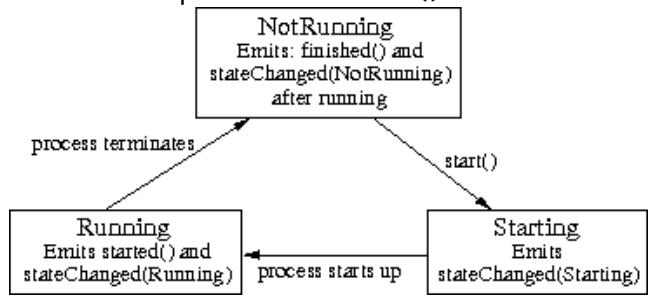
- QProcess executes external processes asynchronously (but you can force your Qt program to wait for the execution of the process, thus making control flow synchronous).
- Signals are emitted when data arrives on stdout, data arrives on stderr, and when the process finishes, and when it changes state.
- The process is started with `start()`; one `QProcess` object can be used to start the same process several times.

Setting up the Process Call

- The command to invoke as well as the arguments are specified in the `start()` call.
- Alternatively, you can specify the arguments (but not the command) using the << streaming operator.
- No escaping is required for the arguments.
- You can set environment variables with `setEnvironment()`. If you do not specify an environment, the environment of the calling process will be used.
- The current working directory for the command can be set using `setWorkingDirectory()`.

Process states

A process passes the following states during its lifetime; the current state can be queried with `state()`:



Process Input

- `QProcess` inherits `QIODevice`.
- `QTextStream` and `QDataStream` can therefore be used (`QDataStream` is really only useful if the child process uses `QDataStream` for its output as well.)
- You can also use `write()`, `read()`, `readLine()`, and `getChar()` directly.
- In order to provide *input* to the external process, just `write()` to it. (Writing *from* the current process means providing *input* to the child process.)

Process Output

- The *output* of the external process is the *input* to the current process and is therefore *read*, using `read()`, `readLine()`, or similar methods.
- Since the process output can be on either of the two channels *stdout* and *stderr*, you need to select the one you want using `setReadChannel()`.
- Alternatively, merge the output with `setReadChannelMode(QProcess::MergedChannels)` (careful that you are not mixing up information then!).

Process Output

- You can also read everything from one channel with `readAllStandardOutput()` and `readAllStandardError()`.
- The signals `readyReadStandardOutput()` and `readyReadStandardError()` signal availability of data to be read.

Process Output, Example

Example: Reading all of standard output as a string after process

- Example: Reading all of standard output as a string after process termination:

```
QProcess process;
process.start( "command"  );
process.waitForFinished();
QTextStream stream( &process );
QString output = stream.readAll();
```

- See the example findTool for an example that reads output as it becomes available.

Controlling the Process

- The signal `finished(int)` is emitted when the program terminates, passing the exit code. `exitCode()` also provides the exit code.
- You can ask the external process to terminate with `terminate()`, `kill()` does the same, but more forceful, without leaving the process a chance to resist.
- `error()` tells you which error occurred last (only useful if `state()` returns `QProcess::NotRunning`).

Synchronous Process Invocation

- `QProcess` has blocking methods to wait for a certain event in the lifecycle of a process:
  - **waitForStarted()** returns when the `started()` signal has been emitted, i.e., the process has started successfully (or has failed to start).
  - **waitForReadyRead()** returns when data is available for reading on the current channel (careful!).
  - **waitForBytesWritten()** returns when certain (unspecified) amount of data has been written.
  - **waitForFinished()** returns when the `finished()` signal has been emitted, i.e., when the process has finished its execution.
- Remember that calling any of these methods in the main thread will freeze your user interface; use multithreading.

Simplified Synchronous Process Invocation

- If you are not interested in processing the output of the external process, nor in supplying input to it, use the static convenience method `QProcess::execute()` that starts the process, waits for its termination, and returns:

```
QStringList arguments;
arguments << "Argument1" << "Argument2";
QProcess::execute("do_it_now", arguments);
// won't get here until do_it_now terminates
```

- Use `QProcess::startDetached()` instead if you want the child process to detach from the current one. This method will not wait for termination, and the child process will not be terminated when the current process terminates ("fire and forget").

Introduction

- ActiveQt is a framework for:
  - embedding Microsoft ActiveX controls into Qt applications (QAxContainer)
  - accessing Microsoft COM objects from Qt applications
  - turning Qt applications into COM servers (QAxServer)
- ActiveQt is for Microsoft Windows only.
- ActiveQt sources are located in `extensions/activeqt`, the examples are in `examples/activeqt`.

The QAxContainer library

- For embedding ActiveX controls in Qt applications
- Important classes: The abstract base class `QAxBase`, and its subclasses `QAxObject` and `QAxWidget`
- Usage in qmake:

  `CONFIG += qaxcontainer`

QAxContainer – Embedding a Control

- If the control is a widget, then the easiest way to find the UUID for the control is simply to create it in Qt designer by creating an instance of the QAxWidget container, right-click it and select control..
- Otherwise use the OLE/COM browser in MSVC.
- Create an instance of `QAxWidget`, passing the UUID to the constructor:

```
QAxWidget* myqax = new QAxWidget(
    "{8856F961-340A-11D0-A96B-00C04FD705A2}",
    this );
```


QAxContainer – Embedding a Control cont'd

- ActiveX properties are mapped to Qt properties.
- ActiveX methods are mapped to Qt slots.
- ActiveX events are mapped to Qt signals.
- The mappings work only if the control supports the `IDispatch` interface.
- You can get information about signal/slot for an object from `Tools | Containers | AxWidget Documentation`
- COM datatypes are mapped to Qt datatypes in a "reasonable" way. See the reference documentation for `QAxBase` class.

QAxContainer – Supported Data Types for Properties

Qt Property	COM Type	Qt Property	COM Type
bool	VARIANT_BOOL	QFont	IFontDisp*
QString	BSTR	QPixmap	IPictureDisp*
int	int	QVariant	VARIANT
uint	unsigned int	QList<QVariant>	SAFEARRAY(VARIANT)
double	double	QStringList	SAFEARRAY(BSTR)
Q_LLONG	CY	QByteArray	SAFEARRAY(BYTE)
Q_ULLONG	CY	QRect	User defined type
QColor	OLE_COLOR	QSize	User defined type
QDate	DATE	QPoint	User defined type
QDateTime	DATE		
QTime	DATE		

QAxContainer – Supported Data Types for Signals and Slots

Qt Type		COM Type	
in-parameter	out-parameter	in-parameter	out-parameter
bool	bool&	VARIANT_BOOL	VARIANT_BOOL*
const QString&	QString&	BSTR	BSTR*
int	int&	int	int*
uint	uint&	unsigned int	unsigned int*
double	double&	double	double*
const QColor&	QColor&	OLE_COLOR	OLE_COLOR*
const QDate&	QDate&	DATE	DATE*
const QDateTime&	QDateTime&	DATE	DATE*
const QFont&	QFont&	IFontDisp	IFontDisp*
const QPixmap&	QPixmap&	IPictureDisp	IPictureDisp*
const QList<QVariant>&	QList<QVariant>&	SAFEARRAY(VARIANT)	SAFEARRAY(VARIANT)*
const QStringList&	QStringList&	SAFEARRAY(BSTR)	SAFEARRAY(BSTR)*
const QByteArray&	QByteArray&	SAFEARRAY(BYTE)	SAFEARRAY(BYTE)*
QObject*		IDispatch*	
QRect&		struct QRect (user defined)	
QSize&		struct QSize (user defined)	
QPoint&		struct QPoint (user defined)	

QAxContainer – Embedding a Control cont'd

- Signal/Slot connection:

```
QAxWidget* myqax = new QAxWidget(
                       "{8856F961-340A-11D0-A96B-00C04FD705A2}",
                       this );
connect( this, SIGNAL(locationChanged(const QString&)),
         myqax, SLOT(Navigate(const QString&)));
connect( myqax, SIGNAL(StatusTextChange(const QString&)),
         statusBar(), SLOT(showMessage(const QString&)));
```


QAxContainer – Embedding a Control cont'd

- Use `QVariant QAxBase::dynamicCall(...)` to call methods directly:

```
myqax->dynamicCall( "Navigate( const QString& )",
                    "www.kdab.net" )
```

- See the examples `examples/activeqt/flash` and `examples/activeqt/webbrowser`

QAxContainer – Misc.

- Use `QAxObject` for objects that don't have a GUI, otherwise use `QAxWidget`
- If a signal cannot be mapped to Qt datatypes, connect to the signal

```
void QAxBase::signal( const QString& name,
                      int argc, void* argv )
```

instead, and filter out the information you need.

- Use the `QAxObject* QAxBase::querySubObject()` method to get references to subobjects of a control.

QAxContainer – Misc.

- Examine the `metaObject()` and `propertyBag()` of the object to see which signals, slots and properties are available.
- Readable documentation for the control can be obtained with the `QString QAxBase::generateDocumentation()` method.
- The ActiveX control of a `QAxWidget` can be set/changed afterwards with the `QAXBase::setControl(const QString&)` method

QAxContainer – dumpcpp

- The dumpcpp tool is used to create a "Qt-friendly" C++ API to COM objects that have a type library. Use the OLE/COM Object Viewer to discover available type libraries.
- Synopsis: `dumpcpp.exe <input> [-n <namespace>] [-o <filename>]`, where `<input>` is a DLL, a typelib or a UUID for a class or typelib. `<namespace>` is the namespace that the resulting code will live in, and `<filename>` is the output filename.
- dumpcpp will not be able to wrap all classes in a type library, see the reference documentation for details.
- See example *activeqt/speech*

QAxContainer – QAxScriptManager

- The classes `QAxScriptManager` and `QAxScript` provide access to the Windows Scripting Host. This allows you to embed interpreters into Qt applications using COM objects.
- Create a script manager: `QAxScriptManager* mgr = new QAxScriptManager();`.
- Expose a COM object to scripts: `mgr->addObject(static_cast<QAxBase*>(my_qax_obj))` (call this once for each object you want to expose).
- To expose "regular" QObjects, add qaxserver to the CONFIG variable in the .pro file and make sure the objects exposed have the required `Q_CLASSINFO` macros as described in the `QAxServer` section.

QAxContainer – `QAxScriptManager` cont'd

- Call one of the `load()` methods to load a script:

```
QAxScript* script =
    mgr->load("myscript.vbs","myscript");
```

and start calling functions in it

```
QVariant result =
    script->call("myfunction("some","args")");
```

- See example *examples/activeqt/flash-script*.

QAxServer – Introduction

- For converting a standard Qt binary into an ActiveX control server
- Can be used both for building out-of-process servers (`.exe` binaries) and in-process servers (`.dll` libraries)
- Important classes: `QAxBindable`, `QAxFactory`, and `QAxAggregated`.

QAxServer – QMake

- When building an out-of-process server:

```
TEMPLATE = app
CONFIG  += qaxserver
RC_FILE  = qaxserver.rc
```

- When building an in-process server:

```
TEMPLATE = lib
CONFIG  += qaxserver dll
DEF_FILE = qaxserver.def
RC_FILE  = qaxserver.rc
```


QAxServer – QMake cont'd

- The files qaxserver.def and qaxserver.rc are parts of ActiveQt and seldomly need to be modified. Either refer directly to the files in the ActiveQt installation or make a local copy.
- The `activeqt` `CONFIG` option will link the application against qaxserver.lib instead of the normal qtmain.lib and it will generate an interface definition and link the type library into the binary and register the server.

QAxServer – Implementing a Server

- Implement a normal QWidget as usual.
- The `Q_OBJECT` macro is required to provide the meta object information about the widget to the ActiveQt framework.
- Generate unique identifiers with `uuidgen` and add them with the `Q_CLASSINFO` macro to the declaration of the class:

```
class MyControl : public QWidget {
    Q_OBJECT
    Q_CLASSINFO("ClassID", "{8E62B253-C95E-4dec-99AE-31ECDADCEBE5}")
    Q_CLASSINFO("InterfaceID", "{966414E5-C430-46eb-8B89-3FE877714A02}")
    Q_CLASSINFO("EventsID", "{528716CD-18F5-4bdf-9EF9-C2924DE87CE7}")
...
};
```

`EventsID` can be omitted if the class does not have any signals.

QAxServer – Implementing a Server

- Mark attributes and methods with the `Q_PROPERTY` macro and decide on signals and slots.
- Remember that only signals, slots and properties with supported datatypes will be made available through ActiveQt.

QAxServer – Implementing a Server cont'd

- Implement a factory for the widget.
- The easiest way is to use the `QAXFACTORY_...` macros:

```
#include "mycontrol.h"
QAXFACTORY_BEGIN("{F1419A01-34EB-4987-9021-5D47A416B83C}", // typelib ID
                 "{61799A2A-EDE5-4714-8259-D60A1A0402E7}") // app ID
  QAXCLASS(MyControl)
  ... // Other Controls
QAXFACTORY_END()
```

- If you only serve one control, there is an even easier macro `QAXFACTORY_DEFAULT` that can be used instead.

QAxServer – Implementing a Server cont'd

- The unique identifiers (UUIDs) for the `QAXFACTORY_BEGIN` macro can be generated with the tools `uuidgen` (console) and `guidgen` (gui). You **must** create new UUIDs for every new application/factory.

QAxServer – Implementing a Server cont'd

- Out-of-process servers should of course also implement a `main()` function:

```
int main( int argc, char **argv ) {
    QApplication app( argc, argv );
    if ( !QAxFactory::isServer() ) {
        // create and show main window...
    }
    return app.exec();
}
```

QAxServer – Implementing a Server cont'd

- If you want to serve more than one control from a server, or if a control does not have the standard (`QWidget* parent`) constructor, then you need to implement a subclass of `QAxFactory` and implement the pure virtual methods

```
virtual QStringList featureList() const = 0
virtual QObject* createObject( const QString& key ) = 0
virtual const QMetaObject*
        QAxFactory::metaObject(const QString& key) const = 0
virtual QUuid classID( const QString& key ) const = 0
virtual QUuid interfaceID( const QString& key ) const = 0
virtual QUuid eventsID( const QString& key ) const = 0
```


QAxServer – Implementing a Server cont'd

- Example factory:

```
#include <qaxfactory.h>
class MyFactory : public QAxFactory {
  ...
}
```


QAxServer – Implementing a Server cont'd

```
QStringList MyFactory::featureList() {
   QStringList lst;
   lst << "AWidget" << "AnotherWidget";
   return lst;
}

QObject* MyFactory::createObject( const QString& key ) {
  if( key == "AWidget" ) { return new AWidget(); }
  if( key == "AnotherWidget" ) {
    return new AnotherWidget( "Hello world" );
  }
  return 0; // return 0 for unknown controls
}
```


QAxServer – Implementing a Server cont'd

```
const QMetaObject *MyFactory::metaObject(const QString &key) const {
  if (key == "AWidget")
    return &AWidget::staticMetaObject;
  if (key == "AnotherWidget")
    return &AnotherWidget::staticMetaObject;
  return 0;
}

QUuid MyFactory::classID( const QString &key ) const {
  if ( key == "AWidget" )
     return "{5BB2516E-A4C6-4c3c-9062-30271ED92ACB}";
  if ( key == "AnotherWidget" )
     return "{3FB74836-62EF-4b0e-BE47-F69F656D36C8}";
  return QUuid();
}
```

QAxServer – Implementing a Server cont'd

```
QUuid MyFactory::interfaceID( const QString &key ) const {
  if ( key == "AWidget" )
     return "{D69653B1-B98D-487e-919B-99C13E9B916B}";
  if ( key == "AnotherWidget" )
     return "{6FCFA76B-8B7D-4120-AF7B-6D693C9AB666}";
  return QUuid();
}

QUuid MyFactory::eventsID( const QString &key ) const {
  if ( key == "AWidget" )
     return "{C6256ED5-D497-4b76-87B0-4D65B49D58DD}";
  if ( key == "AnotherWidget" )
     return "{D3A3B1EE-D40B-481d-8D15-7C7A7025E2CF}";
  return QUuid();
}
```


QAxServer – Implementing a Server cont'd

- Finally, the new factory needs to be instantiated. Only one factory class can exist in each server. Use the `QAXFACTORY_EXPORT` macro:

```
QAXFACTORY_EXPORT( MyFactory,                  // factory class
     "{3E433C28-92E3-4090-894A-529A8EEE421B}", // type library ID
     "{0DEACA70-A214-4145-8855-08A7A33E3D6D}"  // application ID
    )
```

- See the example *examples/activeqt/helloserver*.

QAxServer – Advanced features

- If you need more control over the ActiveX interface of a `QWidget`, you can inherit multiply from both `QWidget` and `QAxBindable`
- This will make the control "bindable", with the following benefits:
  - A mechanism for controlling when properties can be changed (see `QAxBindable::requestPropertyChange()`) and the ability to notify the client when a property has changed.
  - Allows the control to implement additional COM interfaces and provide alternative implementations for COM interfaces. (see `QAxBindable::createAggregate()`)

QAxServer – Advanced Features cont'd

- The `createAggregate()` method should return a new object of type `QAxAggregated`
- The class should inherit multiply from `QAxAggregated` and any COM interfaces it implements, and it should implement the virtual function `QAxBase::queryInterface(const QUuid& iid, void** iface)`
  and return a pointer to the requested interface implementation in `iface`. See the example `opengl` in ActiveQt

QAxServer – Registering the Server

- Before an ActiveX server is available to other applications, it must be registered.
- The QMake-generated Makefile takes care of this at build time.
- If you distribute a binary version of an out-of-process server, run the server with the `-regserver` option to register it, and with `-unregserver` to unregister it. This could be done from the installer/uninstaller of the application.
- For in-process `.dll` servers, use the Windows tool `regsrv32` to register.
- Servers can be tested using *extensions/activeqt/tools/testcon*.

Project Task

- Convert the `ScribbleArea` widget into an ActiveX control and implement an out-of-process server for it.
- Add properties for the pen color and pen width to make them accessible from an ActiveX client.
- Add a slot to clear the scribble area
- Optionally: Create a webpage that embeds a scribble area and is able to control the color and pen width, as well as to clear the scribble area.
- Optional: Make the scripting-related security warnings go away (Hint: see the opengl example in ActiveQt)

Migrating Motif programs to Qt

- Using the Motif solution, it is possible to gradually migrate your programs from Motif to Qt.
- Basically, this means that you can have Motif widgets inside Qt widgets.
- It is not possible to have Qt widgets inside Motif widgets, however (unlike with the old Xt extension).

Requirements

- QtMotif is part of Qt Solutions.
- Ensure that you are using X11R6 and Motif 2.x.
- Ensure that the programs you wish to port are compile with a C++ compiler, or at least that this is the case for the parts which need migration. (C programs sometimes use variable names like `class` that are keywords in C++.)

Classes Involved

- The Motif solution consists of three classes:
- `QtMotif`: This class takes care of integration—among other things it implements an event loop.
- `QtMotifWidget`: This is a QWidget which wraps a Motif widget.
- `QtMotifDialog`: This is a QDialog wrapper for Motif dialogs. This is needed to get the modality right.

QtMotif

- To use the Motif migration tools, you need to create an instance of `QtMotif` before creating the instance of `QApplication`.
- `QtMotif` takes a number of arguments that it passes to `XtDisplayInitialize()`: applicationClass (name used in resource database), options and numOptions (options and count of options used for command line parsing).
- Normally adding a line like the following to the `main()` function should be enough: `QtMotif integrator( "integrator" );`

QtMotifWidget

- `QtMotifWidget` is a wrapper around a Motif widget.
- The constructor for `QtMotifWidget` takes the `Widget` class, the argument list, and the number of arguments.
- `QtMotifWidget::motifWidget()` returns the wrapped `Widget`.
- Old code:
  `XmCreatePushButton( parent, "name", args, n );`
- New code:
```
new QtMotifWidget( "name", xmPushButtonWidgetClass,
                   parent, args, n );
```
- See the example *examples/customwidget* in the Qt Motif solution package. in `$QTDIR`

QtMotifDialog

- The class `QtMotifDialog` can either be a Qt dialog with Motif children, or a Motif dialog with Qt children:
```
QtMotifDialog( Widget parent,
               Qt::WFlags flags = 0 )
QtMotifDialog( QWidget * parent,
               Qt::WFlags flags = 0 )
```
- In both situations, modality continues to work as expected.

QtMotif Dialog - Motif Parent

- This is the case where you have a Motif main window or dialog that you have not rewritten yet, but want to act as a parent for a new dialog written in Qt.
- Simply use the `QtMotifDialog` in place of `QDialog`:

```
class MyDialog :public QtMotifDialog {
    MyDialog( Widget parent, ... )
      : QtMotifDialog( parent, ... ) {
      // normal Qt code
    }
}
```


QtMotif Dialog - Qt Parent

- This is the case where you want a not-yet-ported Motif dialog to have a Qt dialog as parent.
- When used this way `QtMotifDialog` offers a method `shell()`, which returns a window `Widget`, to replace the dialog shell:

```
QtMotifDialog* dialog = new QtMotifDialog( qtParent );
Widget form = XmCreateForm( dialog->shell(),
              "custom Motif dialog", NULL, 0 );
```


QtMotif Dialog - Qt Parent

- Using `QtMotifDialog` with Motif children, the following two static methods of `QtMotifDialog` can be very useful:

```
void acceptCallback(Widget, XtPointer client_data,
                    XtPointer)
void rejectCallback(Widget, XtPointer client_data,
                    XtPointer)
```

- `client_data` must point to the dialog in question.

Suggested Migration plan

- Find out which part of your source code uses Motif code at all—if you are lucky (or have a clean design), you may not have to "port" all of it.
- Change the main window(s) to using QMainWindow and QActions, connect any action to slots which call the original Motif functions. Use `QtMotifWidget` to wrap the content of the main windows.
- Wrap any dialogs in `QtMotifDialog`.
- In your preferred order you can now rewrite the main window, and dialogs step-by-step in small iterations. This allows you to test your application after each step.

Caveats

- Always include qt headers before Motif,Xt, and X11 headers, as X11 has some conflicting `#define`'s
- QtMotifWidget can not be `reparented`, which among other things means that you can not call `QWidget::showFullScreen()` and `QWidget::showNormal()`.

Overview

- What is QSA.
- Object, Factories, and Wrappers.
- Projects, Interpreter and editors.
- ECMAScript language.

What is QSA

- QSA stands for **Q**t **S**cript for **A**pplications.
- QSA is an interpreted extension to Qt, using a scripting language derived from ECMAScript (AKA. Javascript)
- The developer decides how much of his application he will make accessible from scripting.
- QSA offers an editor for the end user to script the application.

QSA basics

- To link QSA into your application, simply add `load(qsa)` to your `.pro` file, which will add include path and libs to your setup.
- For objects to be available to scripting, you need to add them using `QSInterpreter::addTransientObject()` or if you have a `QSProject` using `QSProject::addObject()`.
- An interpreter can be obtained using `QSInterpreter::defaultInterpreter()` and `QSProject::interpreter()`.
- Scripts can be evaluated using `QSInterpreter::evaluate()`.

Objects

- Only `QObject` inherited objects can be made available for scripting. (This is slightly wrong - a QSWrapperFactory could help you).
- The object is available in the script as an object with the name returned from `QObject::name()`.
- Slots are available as functions (notice in QSA it makes sense for slots to return a value).
- Properties are available as variables on objects.
- See example *qsa/simpleApp*.

Creating C++ objects in QSA

- To be able to create C++ objects in scripts you need a factory class, which given a name creates an instance of the object in question.
- Create a class that inherit `QSObjectFactory`, and register it using `QSInterpreter::addObjectFactory()`.
- From the constructor of your factory, you must call `registerClass()` for each class name the factory is capable of creating.
- `registerClass()` takes as an optional last argument a pointer to an instance of the class in question. This instance will be used to access static content of the class.

Creating C++ objects in QSA cont'd.

- A single factory can be used to generate several different objects.
- Be aware of items 9.5 – 9.10 in the License!
- See example *qsa/file*.

Accessing non QObject's from QSA

- Sometimes, you may have a function in C++ which returns pointers to classes that do not inherit QObject, and which you do not or cannot make inherit QObject.
- Sometimes, you may want to use a class inheriting QObject from QSA, but it does not have the methods you need available as slots.
- Both problems can be solved by creating a wrapper factory.
- The wrapper factory is responsible for creating an instance of a class that wraps the troublesome class in question, and implements the necessary slots and properties.

Accessing non QObject's from QSA cont'd.

- Create a class inheriting from QSWrapperFactory, implementing a few virtual methods, and registering the factory using `QSInterpreter::addWrapperFactory()`.
- In the constructor of your wrapper, you must call `registerWrapper()` for each class name the wrapper wraps.
- See examples *qsa/wrapperNonQObject* and *qsa/wrapperQTable*

Enums

- Enums that are registered using `Q_ENUMS` can be accessed from QSA.
- Without doing anything extra you can access them by dotting into an object of the class in question.
- If however you want to be able to refer to the enum without an instance of the class, then you need to use `QSObjectFactory` to make the class available in QSA.
- `QSObjectFactory::registerClass()` takes as the last argument a pointer to an object representing the static part of the class.
- Sometimes you may wish to put the enums in a class for themselves.
- See example *qsa/enums*

Projects

- The class `QSProject` bundles a number of scripts into a project, and makes it convenient to load and save scripts.
- The project is loaded and saved using `load()` and `save()`, both taking a project file name as argument.
- Alternatively, you can use `loadFromData()` and `saveToData()` which uses a `QByteArray` instead.

Creating new scripts

- New scripts may be created using `QSScript* QSProject::createScript()`
- The class `QSScript` is a simple container for scripts.
- It is possible to associate a `QSScript` with a context object, meaning that the script will be evaluated in the context of that object.
  use **property=42;**
  rather than **Application.path1.path2.property=42**
- If a script and an object have the same name, they are associated.

Creating new scripts cont'd.

- The method `createScript` is overloaded to either take a context object or a name as its first argument. The second argument is the actual script code itself.
- It is possible to query the project for scripts using one of:
  `script(const QString& name)`
  `script(QObject* context)`
- `scriptNames()` returns a list of scripts in the project.

Objects

- Scriptable objects are added to a project using `QSProject::addObject()`.
- Several actions in the interpreter will trigger re-evaluation of the project; consequently, the interpreter will be cleared and the scripts re-evaluated. These actions include modifying scripts and removing objects.
- Objects added to the project become persistent, meaning that when the interpreter is cleared, they will still be scriptable, in contrast to objects added using `QSInterpreter::addTransientObject()`.

Editor and Workbench

- QSA offers a complete workbench for editing code, and a simple editor (actually the editor part of the workbench).
- The editor may be created for a script using `QSProject::createEditor(QSScript *)`, and a previously created editor can be fetched from the project using `QSProject::editor(QSScript *)`.
- The editor is an instance of `QSEditor`, and is enhanced for editing QSA code.
- Several editors may exist at the same time – the active editor can be obtained using `QSProject::activeEditor()`

Editor and Workbench cont'd.

- The signal `QSProject::editorTextChanged` will be emitted when the content of an editor is changed.
- To reevaluate the editor's content, i.e. (re)load it into the script, call `QSProject::commitEditorContents()`.
- To discard editor changes, call `revertEditorContents()`.
- See example *qsa/editor*

Editor and Workbench cont'd.

- A complete workbench can be obtained by creating an instance of the class `QSWorkbench`, giving it a project as argument to the constructor.
- The workbench takes care of much more for the programmer, for example reevaluating scripts.
- We'll see `QSWorkBench` in action at the end of the QSA section.

Interpreter

- In smaller setups you can do without the use of a project. In these situations you will simply work with a default interpreter, which you can obtain using `QSInterpreter::defaultInterpreter()`.
- In setups with a project you get to an interpreter using `QSProject::interpreter()`
- `QSInterpreter::evaluate()` evaluates an arbitrary expression.
- `QSInterpreter::call()` calls a function specified by name.

Interpreter cont'd.

- `QSInterpreter::functions()`, `QSInterpreter::classes()` and `QSInterpreter::variables()` queries the interpreter for functions, classes, and variables in existence.
- `QSInterpreter::presentObjects()` returns a list of objects available for scripting in QSA.

Error handling in the interpreter

- `QSInterpreter::checkSyntax()` tells you whether a script given as argument is syntactically correct.
- The signal `QSInterpreter::error()` is emitted when an error occurs in the interpreter (syntax or runtime error).
- Alternatively you can ask the interpreter if an error occurred during the previous evaluation.
- `QSInterpreter::errorMessage()` return the last reported error message.
- `QSInterpreter::stackTrace()` returns the last reported stack trace.

QSA Input Dialog Framework

- QSA's purpose is to script applications not to build applications.
- Still, simple dialogs may be needed from time to time, to query the user for input parameters to scripts.
- QSA Input Dialog Framework exists exactly for that purpose.
- The easiest way to get to the reference page is from the front page of the tutorial.
- QSA Input Dialog Framework can only create windows with content (thus it is impossible to add a widget to an existing application).

QSA Input Dialog Framework cont'd.

- Currently the following widgets are accessible within the dialog: `CheckBox`, `ComboBox`, `DateEdit`, `GroupBox`, `LineEdit`, `NumberEdit`, `RadioButton`, `SpinBox`, `TextEdit`, `TimeEdit`
- It is not possible to inherit from the widgets.
- In addition a number of dialogs exist on their own: `MessageBox`, `FileDialog`, and `Input`.
- In your application you must create an instance of `QSInputDialogFactory` to use QSA Input Dialog Framework, plus of course add it to your interpreter using `QSInterpreter::addObjectFactory()`.
- See example *qsa/dialog*

Variables

```
var a;      // undefined local variable
var b = 42; // local variable
c = 42;     // global variable
const d = 42;  // constant
```


Statements

- The following constructs with a syntax similar to C++ exist:
- if-then-else, break, continue
- while, do, for
- switch, case, break, default
- try, catch, finally, throw
- In addition a scope mechanism called `with` exists

Builtin Types

- QSA is capable of handling booleans, numbers and strings natively, so if for example a C++ function called foo takes a string as argument, then you can simply call it from QSA as: `foo( "a string" );`
- QSA has a number of built-in types that translate to Qt types. These includes `Date`, `Point`, `Rect`, `Size`, `Color`, `ByteArray`, `Font`, and `Pixmap`.
- Thus to invoke `foo( const QPoint& )`, use a command like this: `foo( new Point(10,20) );`
- The above types offer functionality similar to the Qt counterparts, for example `String.mid()`, `Rect.contains()` etc.

Arrays

- Arrays implement simple lists:

```
var mammals = ["human", "dolphin", "elephant", "monkey"
var a = new Array( 10 ); // 10 elements
var plants = new Array( "flower", "tree", "shrub" );
for ( i = 0; i < mammals.length; i++ ) {
      things[i] = new Array( 2 );
      things[i][0] = mammals[i];
      things[i][1] = plants[i];
}
```


Array functions

pop returns the rightmost element and removes it from the list.

push appends an element to the end of the list.

shift/unshift similar to pop/push but operates on the beginning of the list.

length this is not a function but a property: list.length (note no parenthesis).

concat Concatenates the array with a number of arrays in the given order, and returns a single array.

join Joins all the elements of an array together, separated by commas, or the specified separator.

Array functions cont'd

slice Copies a slice of the array.

splice inserts, removes or replace elements of a list into another.

reverse reverses the content of a list.

sort sorts the elements in a list. Takes an optional function which is used to sort the elements.

Dictionaries

- An array can also be used as a dictionary:

```
var names = [];
names["first"] = "John";
names["last"] = "Doe";
var firstName = names["first"];
var lastName = names["last"];
```

- Property syntax can be used for dictionaries:

```
names["address"] = "Somewhere street 2";
names.phoneNumber = "+0123456789";
var address = names.address;
var phoneNumber = names["phoneNumber"];
```


Classes and Methods

-
```
class Circle {
  var x; // instance variable

  // constructor
  function Circle( posx, posy, radius ) {
    this.x = posx;
    this.y = posy;
    this.r = radius;
  }
}
```


Classes and Methods cont'd.

-
```
class ColorCircle extends Circle {
  var rgb;
  function ColorCircle( posx, posy, radius, rgbcolor)
  {
    // Calling superclass constructor
    Circle( posx, posy, radius );
    this.rgb = rgbcolor;
  }
  // method
  function setRgb( rgbcolor ) { rgb = rgbcolor; }
}
```


connect

- connect exists as a global function in scripts, with two possible syntaxes:
-
```
connect( signallingObject, signal,
         receivingObject, slot )
```
-
```
connect( sender, signal, functionRef )
```
- example:

```
connect( Application.top.edit1,
         "textChanged( QString )", obj.method );
```

- See example *qsa/connect*.

Misc

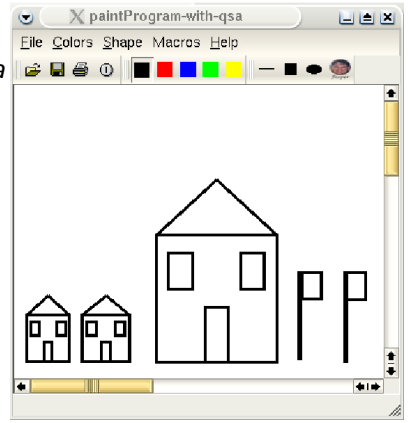
- Comments are C styles: Either /* ... */ or //
- Timers are available using the global functions `startTimer`, `killTimer`, and `killTimers`
- The class Math contains mathematical constants and methods.
- In scripts you always create pointers to instances. QSA will however ensure that the pointers are dereferenced for you if you call a function taking an instance rather than a pointer.

Misc cont'd.

- You can only use the return value from a function if it is a basic type (like int, char, QString, ...) or a pointer. Thus "`Data function()`" and "`Data& function()`" do not work.
- The function can still be invoked, though.
- Finally let see example *spreadsheet* from the QSA installation.

Project Task

- Starting from *handout/paintProgram-for-qsa* add scripting facilities to the paint program, so that it is possible to drag out shapes, implemented in QSA.
- Search for `TODO` , to find places to add implementation.
- See the script `houses.qs`, to see what the interface should be like.



Emulating MDI with QWorkspace

- Provides a "window manager within a window"
- Easy to use:
  - Start with a `QMainWindow` with menu bar, tool bars, etc., and `#include <QWorkspace>`.
  - Use a `QWorkspace` widget as the central widget in the main window.
  - Make all document (or other normally top-level) windows children of the `QWorkspace` and for each of them call:

```
QWorkspace::addWindow( QWidget* w,
                       Qt::WFlags flags = 0 )
```

Emulating MDI with QWorkspace cont'd.

- `QWorkspace::addWindow()` returns a `QWidget` pointer to the window.
- On this pointer you may use `show()`, `hide()`, `move()`, `resize()`, `showMaximized()`, `showMinimized()` and `showNormal()` as you would with windows.
- `QWorkspace` has a number of slots that can be used to manipulate a collection of subwindows:
  - `cascade()` and `tile()` to layout the windows
  - `activateNextWindow()` and `activatePreviousWindow()` to control which window is active
  - `closeActiveWindow()` and `closeActiveWindow()` to close windows.

Emulating MDI with QWorkspace cont'd.

- `QWorkspace::windowList(WindowOrder order = CreationOrder)` returns a list of windows. This is typically used to fill out a popup menu from which the user can select which window to activate. The menu should call the method `QWorkspace::setActiveWindow(QWidget*)` to activate the chosen window.
- The look is like the KDE window manager on X11 and like ordinary Windows MDI on Windows.
- See example *MDI*.

Overview

- `Q3Canvas`, `Q3CanvasView` etc. are Qt 3 compatibility classes. Later Qt versions are expected to provide a replacement module for these classes, based on Qt 4's powerful new 2D paint system.
- The canvas module provides a highly optimized 2D graphics area called `Q3Canvas`.
- `Q3Canvas` can contain an arbitrary number of Q3CanvasItems.
- The items can, with the help of `Q3Canvas`, move around on their own, you just specify a velocity in x and y direction.

Overview cont'd.

- `Q3Canvas` offers methods for collision detection.
- For each `Q3CanvasItem` you can specify a z-order. Items with a higher z-order will go on top of items with a lower z-order.
- `Q3Canvas` is viewed using a `Q3CanvasView`. A view can be transformed using a `QMatrix`.

The Disadvantages

- The major disadvantage of `Q3Canvas` (and the reason why it is so fast) is that the items on the canvas do **not** inherit `QWidget`.
- This has, among others, the following implications:
  - You can not have `Q3CanvasItems` within `Q3CanvasItems`.
  - You can not use a `QPainter` on the items (there exists, however, an item allowing a `QPixmap` to be used).
  - Events are not sent to each individual item (though events can be received by the view, and easily mapped to the item in question).
  - You cannot use layout managers on Q3Canvas.

Basics steps in using `Q3Canvas`

- Create an instance of `Q3Canvas`, and size it using `resize`.
- Create a view to the canvas using `Q3CanvasView`, and insert it like any other widget into your layout.
- Add `Q3CanvasItems` to the canvas. For each item, call show if you want it to be visible.
- Optionally, create custom items by inheriting one of the `Q3CanvasItem` subclasses. Most likely you will at least want to implement the method `advance`.

Existing Q3CanvasItems

- `Q3CanvasEllipse` – An ellipse or "pie segment".
- `Q3CanvasLine` – A line segment.
- `Q3CanvasPolygon` – A polygon.
- `Q3CanvasPolygonalItem` – A base class for items that have a non-rectangular shape. Most canvas items derive from this class.
- `Q3CanvasRectangle` – A rectangle.
- `Q3CanvasSpline` – A multi-bezier spline.
- `Q3CanvasSprite` – An animated pixmap.
- `Q3CanvasText` – A text string.

Moving Q3CanvasItems

- `Q3CanvasItems` can move around on their own, all you have to do is to place the items using `move`, set their z-order using `setZ`, and set their velocities using `setVelocity`.
- The velocities are specified as pixels per step.

- To get it all started, call `Q3Canvas::setAdvancePeriod`. This method takes as argument, milliseconds between each step.
- For more control, override `Q3CanvasItem::advance`.

Q3CanvasItem::advance(int phase)

- Override `advance` in order to either take full control over movement, or just change speed and direction on collisions.
- The method is called in two steps, specified using the integer argument `phase`.
- Phase 0 is the *calculation phase*, no items should move in this phase, instead they should calculate their move, optionally on the basis of placement of other items.
- Phase 1 is the *movement phase*. Here, items should move unconditionally on the basis of the calculation in phase 0.

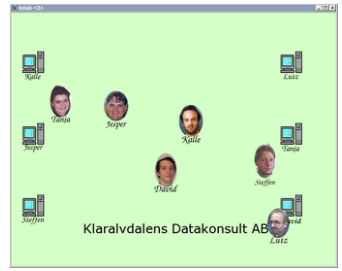
Collisions

- `Q3Canvas` does not handle collisions itself, that is the task of the program (often in the `advance` method).
- `Q3CanvasItem` offers a `collision` method which returns a list of items which will collide with the item after the next `advance`, if no velocity or position changes.
- `Q3Canvas` offers a number of overloaded `collision` methods which can check for intersection of `Q3CanvasItems` at a given point, within a rectangle or within a point array.

Miscellaneous

- `Q3CanvasItems` can be shown and hidden using `show` and `hide`. You must call show for each item.
- `Q3CanvasItem` provides `setEnabled`, `setActive` and `setSelected`. These functions set the relevant boolean values and cause a repaint, but the boolean values are not used in `Q3CanvasItem` itself.
- `Q3CanvasItems::rtti` makes it easy to do dynamic type checks.
- See example *kdab*.

Project Work: Make a Game



- Extend the example to a game where the player has to make the employees work at their desks.
- Step 1: Make it possible to move the employees with the mouse (implement: KDAB::contentsMousePressEvent KDAB::contentsMouseMoveEvent)
- Step 2: Add computers to the canvas, and make sure the employees don't run on top of them.

Project Work: Make a Game cont'd.

- Step 3: Make it possible to drop an employee at a computer. When an employee is dropped at his computer, `hide()` him, and make him `show()` up again after a number of seconds.
- Step 4 (Optional): Make sure your game works when the view is transformed with a matrix.
- Step 5 (Optional): Implement a high score, which you can add yourself to, when you have made all employees work at the same time.
- Step 6 (Optional): Fix the problems when employees can bump over each other or get stuck at the border.

Overview

- `QGraphicsView` is a view for fast structured 2D graphics that was introduced in Qt 4.2.
- Graphics items (`QGraphicsItem`) are placed in a `QGraphicsScene`. A `QGraphicsScene` can then be displayed on one or more `QGraphicsViews`.
- Graphics items can be...
  - lines, rectangles, polygons, ellipses, pixmaps, text (even editable text!), SVG drawings, collections of graphics items or completely custom items.
  - moved, scaled, rotated etc.

Overview cont'd.

- `QGraphicsItem` offers methods for collision detection.
- For each `QGraphicsItem` you can specify a z-order. Items with a higher z-order will go on top of items with a lower z-order.
- A `QGraphicsScene` is viewed using a `QGraphicsView`. A view can be transformed using a `QMatrix`.

Items are not widgets

- The major difference between `QGraphicsItem` and, say `QLabel` is that the former is **not** a `QWidget` subclass.
- This has, among others, the following implications:
  - You can't place `QWidgets` in a `QGraphicsScene`.
  - You can't place `QGraphicsItems` outside a `QGraphicsScene`.
  - You can't use layout managers with `QGraphicsItems`.
  - `QGraphicsScene`/`QGraphicsItem` has its own internal event system that is slightly different from the regular `QEvent` system.
  - `QGraphicsItems` can, unlike widgets, be arbitrarily transformed. This allows the creation of widgets that support high quality zooming etc.

Basics steps in using `QGraphicsView`

- Create an instance of `QGraphicsScene`.
- Create a view to the scene using `QGraphicsView`, and insert it like any other widget into your layout.
- Create `QGraphicsItems` to the scene. For each item, set its position and other properties to your liking. Add each item with `scene->addItem(item);`
- Optionally, create custom items by subclassing `QGraphicsItem` (or a subclass). As a minimum, you need to implement `boundingRect()` and `paint(QPainter*,const QStyleOptionGraphicsItem*,QWidget*)`

Existing QGraphisIsItems

- `QGraphicsLineItem` – A line segment.
- `QAbstractGraphicsShapeItem` – Common baseclass for "shape items", ie. ellipses, polygons etc.
- `QGraphicsEllipseItem` – An ellipse or "pie segment".
- `QGraphicsPolygonItem` – A polygon.
- `QGraphicsRectItem` – A rectangle.
- `QGraphicsPathItem` – A QPainterPath.
- `QGraphicsPixmapItem` – A pixmap.
- `QGraphicsTextItem` – A rich text string (can be editable).
- `QGraphicsSimpleTextItem` – A non-editable plain text string.
- `QGraphicsSvgItem` – An SVG element.

Miscellaneous

- `QGraphicsItems` can be shown and hidden using `show()` and `hide()`. Items are created in their visible state.
- `QGraphicsItem` provides
  - `setEnabled(bool)` to enable/disable an item. Disabled items can not take focus or receive events.
  - `setFocus(Qt::FocusReason)` to set input focus to an item. Invisible items can't have focus.
  - `setSelected(bool)` to select/deselect an item. This member function is often not called directly, but rather through `QGraphicsScene::setSelectionArea()`.
  - `type()` makes it easy to do dynamic type checks. Custom item classes should always override this with an implementation that returns a class-unique number larger than `QGraphicsItem::UserType`.
- See example *graphicsview*.

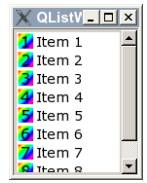
Model/View Programming

Introducing the Widgets

- Qt contains the following high-level widgets which we will talk about in this section:
  - `QListWidget` / `QListView`
  - `QTreeWidget` / `QTreeView`
  - `QTableWidget` / `QTableView`
- The *Widget* versions are widgets that you can use right away as regular item based widgets.
- The *View* versions are for model/view usage, and display the contents of a model.

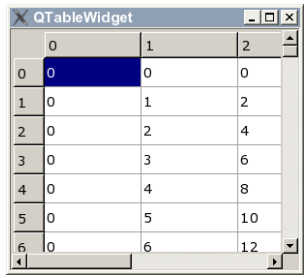
QListWidget

- `QListWidget` can display its data either as a single column of text plus an optional icon (`ListMode`), or as movable icons (`IconMode`), select the mode using `setViewMode()`
- Items are instances of `QListWidgetItem`.
- See the example *QListWidget*.



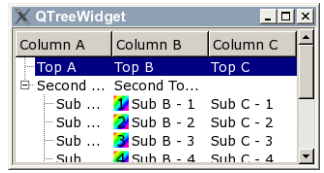
QTableWidget

- Displaying and editing tabular data, e.g. a spreadsheet.
- Very memory-efficient: Only cells with contents need memory.
- Cells in the table are represented using the class `QTableWidgetItem`.
- Set a cell using `QTableWidget::setItem()` specifying a coordinate and an item instance.
- See the example *QTableWidget*



QTreeWidget

- `QTreeWidget` is much like a `QListWidget`, but has a number of additional features.
  - `QTreeWidget` supports multiple columns.
  - `QTreeWidget` allows the user to sort and change the order of the columns.
  - Branches can be expanded or collapsed.
- See the example *QTreeWidget*



QTreeWidget cont'd.

- Each of the three widget classes have, to a large extent, the same API, so we will only talk about `QTreeWidget` in the following.
- If you are not too interested in the widget version, we may jump to page 506.
- The `QTreeWidget` class inherits from `QTreeView`, where many of the methods are defined.
- The items in the widget are subclasses of `QTreeWidgetItem`.
- When constructing an item, pass either the tree widget or an already inserted item as the parent to specify its location in the widget.
- Alternatively, you can call `addTopLevelItem()` or `addChild()`.

Items

- Use `QTreeWidgetItem::setText()` and `QTreeWidgetItem::setIcon()` to set text and icon per column.
- In addition, you can specify tool tips, status tips, What's This? messages, font, text alignment, text color and background color per item, per column.
- Example: `myItem->setIcon( 2, QIcon(...) );`

The Header

- You need to explicitly add columns to the widget to see them. Use `setColumnCount()` or specify the header labels individually.
- The header labels of the tree widget can be specified using `QTreeWidget::setHeaderLabels(const QStringList&)`
- Alternatively, use `QTreeWidget::setHeaderItem(QTreeWidgetItem*)` This allows you to specify an icon, tool tips. etc.

Selection and Keyboard Focus

- Keyboard focus is shown using a line around the item.
- Selection is shown by inverting the item.
- Only one item can have keyboard focus while several items can be selected (depending on the selection mode)
- Four different selection modes exist: `NoSelection()`, `Single()`, `Multi()`, and `Extended()`.

Selection and Keyboard Focus cont'd.

- `Multi()` and `Extended()` allow the user to select several items. In `Extended()` mode, selecting one item deselects other selected items unless Ctrl or Shift is held. This is not the case for `Multi()`-mode.
- Keyboard focus is retrieved and set using `QTreeWidget::currentItem()` and `QTreeWidget::setCurrentItem()` respectively.
- Selection is retrieved and set using `QTreeWidget::selectedItems()` and `QTreeWidget::setItemSelected(QTreeWidgetItem*, bool)` respectively.

Sorting

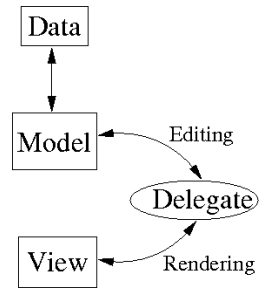
- The user can sort the items by clicking in the header.
- This can be enabled by calling `QTreeWidget::setSortingEnabled(true)`.
- The tree view shows an arrow in the header to indicate sorting direction. This can be disabled by calling `QHeaderView::setShowSortIndicator(false)`.
- You can sort the items programmatically using `QListView::sortItems(int column, Qt::SortOrder)`.
- You can change the sorting algorithm by overriding `QTreeWidgetItem::operator<()`.

Events

- `QTreeWidgetItem` does *not* inherit from `QWidget` or `QObject`, and therefore does not get any events nor emit any signals.
- Events can be handled by `QTreeWidget`, and can be mapped to the item in question using `QTreeWidget::itemAt(QPoint)`
- `QTreeWidget` does part of this job for you by emitting a large number of signals notifying you about changes to the items: `itemClicked()`, `itemEntered()`, `itemExpanded()`, ...
- When constructing a `QTreeWidgetItem`, you can specify a type, which is basically a number. This type can later be retrieved using the method `type()`. This can be used for RTTI.

Concept

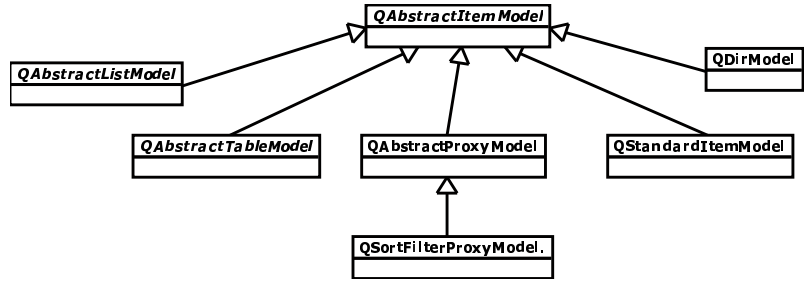
- **Model** Provides an interface to the real data.
- **View** Displays the data.
- **Delegate** Renders the items of data, and supports editing of data.



Concept cont'd.

- The views can almost always be used as-is, and it would be the exception for you to add a new view.
- Often you will subclass one of the model classes to supply your own models.
- Occasionally you might want more control over how the data is displayed or edited, and you will then create your own delegate.
- First, lets see how simple it can be: *model-view/simple*

Model Classes



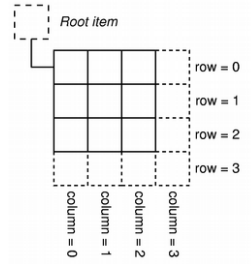
Model Indexes

- Each piece of information that can be obtained via a model is represented by a *model index*.
- Two kinds of model index exists:
  - Temporary indexes that should not be kept around (`QModelIndex`)
  - Persistent model indexes, which are updated by the model, when it internally reorganizes (`QPersistentModelIndex`)
    Warning: they might be expensive for Qt to maintain.
- An index consists of three parameters: *row*, *column*, and *parent index*.

Table Models

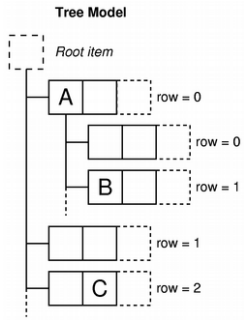
- In the simplest form a model can be represented as a table.
- In that case, the parent index is not used.
- Example:

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



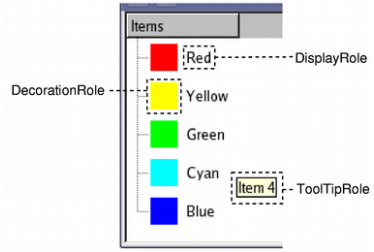
Tree Models

```
QModelIndex indexA =
  model->index(0, 0, QModelIndex());

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

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



Item Roles

- Each item in the model can contain different data (aka. performing different kind of *roles*).
- You get the data for a given role using `QAbstractItemModel::data()`, providing an index, and a `DisplayRole` enum.



Item Roles cont'd.

- The existing views and delegates understand a lot of roles:
  - **General purpose:** DisplayRole, DecorationRole, EditRole, ToolTipRole, StatusTipRole, WhatsThisRole
  - **Appearance and meta data:** FontRole, TextAlignmentRole, BackgroundColorMode, TextColorRole, CheckStateRole, SizeHintRole
  - **Accessibility:** AccessibleTextRole, AccessibleDescriptionRole
- You can also provide your own roles, though Qt's Views and Delegates might not make use of them.

Implementing Your Own Model

- When implementing our own models, we have a variety of classes to choose from, each with their advantages:
  - **QAbstractListModel** One dimensional list.
  - **QAbstractTableModel** Two-dimensional tables.
  - **QStandardItemModel** A simple model that stores the data.
  - **QAbstractItemModel** Generic model class.
  - **QStringListModel** Simple one-dimensional model that works on a string list.
- Notice that you need to subclass each of the above classes except `QStandardItemModel` and `QStringListModel`.

Implementing Your Own Model cont'd.

- Using `QStandardItemModel`:
  - Create an instance of the class.
  - Call `setData()` or `setItem()` on the instance, with each of your data items.
  - `setData()` takes the value as a QVariant, while `setItem()` uses an instance of the class `QStandardItem`.
  - To use inherited version of `QStandardItems`, you need to set up a factory using `QStandardItemModel::setItemPrototype(const QStandardItem*)`
  - The item given to `setItemPrototype()` will act as a factory using its `clone()` method.
- See example *model-view/QStandardItemModel*

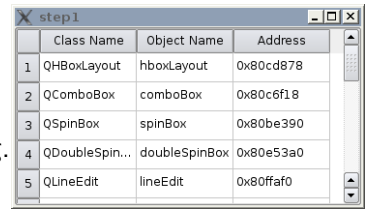
Implementing Your Own Model cont'd.

- Using `QAbstractListModel`:
  - Inherit from the class
  - Implement `int rowCount()` and `QVariant data()`
  - Optionally reimplement `QVariant headerData()` to make the model work better with `QTableView` and `QTreeView`.
  - If the model is editable, reimplement `Qt::ItemFlags flags()`, and `setData()`.
  - Optionally, to allow row insertion and removal, implement `insertRows()` and `removeRows()`
- See example *model-view/QAbstractListModel*

Implementing Your Own Model cont'd.

- Using `QAbstractTableModel`:
  - Everything from `QAbstractListModel` applies here, too.
  - In addition, you must implement `int columnCount()`
  - Optionally also implement `insertColumns()` and `removeColumns()`.

Project Task: Object Browser - step 1



- In the following exercises we will implement an object browser that allows us to see the parent/child hierarchy of a dialog.
- In the first step you must implement a table model.
- Your model will take as argument a widget which should be displayed.
- Start with the handout in *handout/object-browser/step1*
- Slow teams may get a kick start by using the implementation in the file *extra-help*.

Implementing Your Own Model cont'd.

- Using `QAbstractItemModel`:
  - Everything from `QAbstractTableModel` applies here, too.
  - Implement `QModelIndex index(row, column, parent)`, where *parent* is itself a model index.
  - Implement `QModelIndex parent(QModelIndex)`.
- Indexes can be created using the protected method `QAbstractItemModel::createIndex()`, which is overloaded to take either an integer index, or a void pointer (usually internal data pointer to your data structure).

Implementing Your Own Model cont'd.

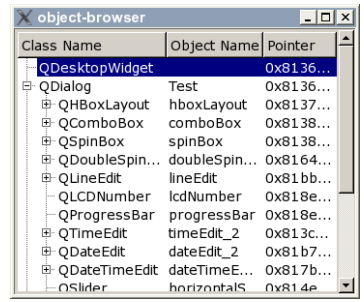
- With the void pointer, you can associate a model index with your internal data structures.
- You can ask an index for its id or pointer using `QModelIndex::internalId()` and `QModelIndex::internalPointer()`
- See example `$QTDIR/examples/itemviews/simpletreemodel/`

Summary of Methods Needed for Your own Model

Minimum methods to implement when inheriting QAbstractItemModel:

- int columnCount(QModelIndex parent)
- int rowCount(QModelIndex parent)
- QVariant data(QModelIndex index, int role)
- QModelIndex index(int row, int column, QModelIndex parent)
- QModelIndex parent(QModelIndex index)

Project Task: Object Browser - step 2



- Its now time to implement an object browser that uses a tree view.
- Start with *handout/object-browser/step2*. First step is to verify this code against your solution.
- This exercise is rather tricky as everything needs to work before anything works. If you get completely stuck, try taking the implementation of *index()* from the solution without looking too much at it.

Project Task: Object Browser - step 2 cont'd.

- Follow these steps carefully.
  1. Discuss internally in your group what data you are going to stick into the indexes.
  2. Without touching any other code implement the method `map()`. This maps from a model index to a widget. Verify with the instructor if in doubt if it is correct.
  3. Update the existing code using the `map()` method.
  4. Implement the `index()` method.
  5. Compile and hope for the best, if it doesn't work, go through the code line by line explaining it to each other.
  6. If it still doesn't work, check and double check if you return empty indexes the right places, and if you check for empty index everywhere where you get an index.
  7. Optional task: throw out the implementation for `parent()` and implement it yourself.

Proxy Models - Motivation

- How would sorting, filtering or reordering of model data for the view be implemented?
  - In the model? All attached views will be affected, which might be unwanted.
  - In the view? Then all model data may have to be queried repeatedly for sorting.
- Solution: Use a stack of models. Intermediate models are called Proxy Models.

Proxy Models - Introduction

- Proxy Models take a source model and transform it, usually by manipulating the model indexes.
- Common uses:
  - Customizable sorting by column.
  - Filtering of rows matching a regular expression or wildcard.
  - Reordering of cells by manipulating the indexes.
- See example: *The Model Flip*

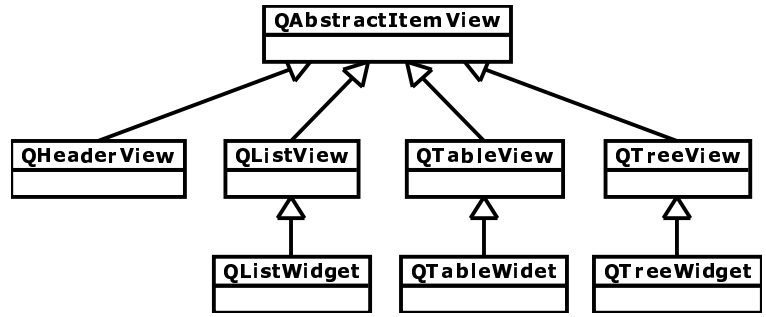
Proxy Model Classes

- `QAbstractProxyModel`: Abstract Proxy Model class that can be used to implement any index, row or column based model transformation. Has to be reimplemented, as it contains pure virtual methods.
- `QSortFilterProxyModel`: Generic implementation of a Proxy Model that allows sorting and filtering of the model data.
- Proxymodels may never give out indexes of their source models. In other words index() and parent() must always call createIndex().
- See the example `QSortFilterProxyModel`

QSortFilterProxyModel

- Can be used to display filtered, sorted model data.
- Model rows can be filtered using regular expressions or wildcards.
- Model rows can be sorted by different columnns.
- Sorting order can be customized by reimplementing the `bool lessThan()` function.
- needs to be reimplemented to overload `lessThan()`, `filterAcceptsColumn()`, `filterAcceptsRow()`

Views



Views cont'd.

- Using `QAbstractItemView::setEditTriggers()`, you specify when editing starts, the possible options are:
  - **NoEditTriggers** No editing possible.
  - **CurrentChanged** Start whenever current item changes.
  - **DoubleClicked** Start when an item is double clicked.
  - **SelectedClicked** Start when clicking on an already selected item.
  - **EditKeyPressed** Start when an edit key has been pressed over an item.
  - **AnyKeyPressed** Start when any key is pressed over an item.
  - **AllEditTriggers** Start for all above actions.

Delegates

- Rendering and editing of items in the views are handled by subclasses of `QAbstractItemDelegate`.
- As we have seen so far, a suitable delegate is already in place, which renders the items, and allows editing using an appropriate widget for the data type in question (`QItemEditorFactory` decides what is *appropriate*).
- Many editing properties are specified directly on the items in the model using `QAbstractItemModel::setData()` with appropriate roles, as specified on page 512.

Delegates cont'd.

- New input methods can be implemented by subclassing `QAbstractItemDelegate` or `QItemDelegate`
- Subclassing from `QAbstractItemDelegate` requires that you also implement rendering methods.
- To create a custom editor, the methods to reimplement are `createEditor()`, `setEditorData()`, `setModelData()`, and optionally `updateEditorGeometry()`.

Delegates cont'd.

- To improve usability it is possible to let the delegate emit the signal `closeEditor()` with a hint on where focus should go.
- This signal should be emitted when the editor is done editing, an event filter might be useful to figure out when that is.
- See example *model-view/delegate*

Selection

- Selection is handled by an instance of the class `QItemSelectionModel`.
- The selection model can be retrieved using `QAbstractItemView::selectionModel()`, and set using `QAbstractItemView::setSelectionModel()`.
- A selection model can be shared between several views.

Selection cont'd.

- The current selection can be retrieved using QModelIndexListQItemSelectionModel::selectedIndexes().
- You build selections using the class `QItemSelection`, and apply them using `QItemSelectionModel::select()`.
- The select method takes as a second parameter `SelectionFlags`, which describes how the selection is to be applied.

Selection cont'd.

- `QItemSelectionModel::SelectionFlag`:
  - **Select** All specified indexes will be selected.
  - **Deselect** All specified indexes will be deselected.
  - **Toggle** All specified indexes will be selected or deselected, depending on their current state.
  - **Rows** All indexes will be expanded to span rows.
  - **Columns** All indexes will be expanded to span columns.

Selection cont'd.

- `QItemSelectionModel::SelectionFlag`:
  - **Clear** The complete selection will be cleared.
  - **Current** The current selection will be updated.
  - **SelectCurrent** Select | Current
  - **ToggleCurrent** Toggle | Current
  - **ClearAndSelect** Clear | Select

Selection cont'd.

- The selection model emits the following signals:
  - currentChanged (const QModelIndex & current, const QModelIndex & previous)
  - selectionChanged (const QItemSelection & selected, const QItemSelection & deselected)
  - currentColumnChanged (const QModelIndex & current, const QModelIndex & previous)
  - currentRowChanged (const QModelIndex & current, const QModelIndex & previous)
- See example *model-view/selection*

Drag and Drop

- The model/view classes support drag and drop to a certain extent, so you do not have to implement it from scratch.
- To enable dragging:
  - Call `QAbstractItemView::setDragEnabled(true)` on each of the views.
  - Reimplement `QAbstactItemModel::mimeTypes()` to return a list of mime types you support
  - Reimplement `QAbstractItemModel::mimeData()` to encode data in your supported formats.
- You only need to implement `mimeTypes()` and `mimeData()` if you want to be able to drop outside of `QAbstractItemViews`.

Drag and Drop cont'd.

- To enable dropping of data:
  - Implement `QAbstractItemModel::dropMimeData()` to handle drops of the types specified with `QAbstactItemModel::mimeTypes()`.
  - Optionally, implement `supportedDropActions()` to specify which drop operations you support. (Default is copy)
- For drag and drop only inside the views, you do not need to implement `dropMimeData()`.

QScrollArea/QAbstractScrollArea

- `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 by 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, so 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 it's 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.
- See the example *qscrollarea/simple*.

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()` (see the example *qscrollarea/margins*).

Drawing the Contents Yourself

- Simplest road to custom drawing is to create a `QWidget` subclass with a custom `paintEvent()` handler and use that widget in a `QScrollArea`.
- To get complete control over painting and scrolling, create a subclass of `QAbstractScollArea` and override its `paintEvent()` handler.
- `QAbstractScollArea` provides a non-scrolling widget that fills the entire central part of the widget.
- A pointer to that widget can be obtained through the method `viewport()`.

Drawing the Contents Yourself cont'd

- Call `setRange()`, `setPageStep()`, etc. on the scrollbars to make them reflect the size of the area you want to draw.
- To paint on the viewport, override `QAbstractScollArea::paintEvent(QPaintEvent* ev)` and open a `QPainter` on the widget returned by `viewport()`.
- The x and y scroll offsets can be obtained through `horizontalScrollBar()->value()` and `verticalScrollBar()->value()`. It is the programmer's responsibility to calculate what needs to be drawn depending on where the scrollbars are.

Drawing the Contents Yourself cont'd

- Just scrolling existing parts of the viewport can be done much more efficiently than repainting the entire widget:
  - Override `QAbstractScollArea::scrollContentsBy(int dx, int dy)` and make it do something more intelligent than the default implementation which just calls `viewport()->update()`.
  - A fast way to scroll is `viewport()->scroll(dx,dy);`
- When the content area is large, you should of course be very careful to only paint the region specified in `ev->region()`.
- See the example `scrollview/drawing`.

Events

- As with all other widgets, `QAbstractScrollArea` allows you to implement a number of event methods. The only difference is that most of the event handlers are remapped to handle events for the viewport widget instead of the scrolled area itself. Those handlers are `paintEvent()`, `mousePressEvent()`, `mouseReleaseEvent()`, `mouseDoubleClickEvent()`, `mouseMoveEvent()`, `wheelEvent()`, `dragEnterEvent()`, `dragMoveEvent()`, `dragLeaveEvent()`, `dropEvent()`, `contextMenuEvent()` and `resizeEvent()`.
- The coordinates reported in the events are in the viewport's coordinate system.

OpenGL and Qt

- OpenGL is a platform-independent API for implementing 3D graphics.
- OpenGL does not provide any support for graphical user interfaces.
- The Qt OpenGL extension allows you to tie together OpenGL and Qt in the same program by providing a widget (`QGLWidget`) that opens an OpenGL display buffer.
- In order to use the OpenGL support, add `QT += opengl` to your QMake files.

Preparations

- The OpenGL module is only included in the Qt Desktop Edition (and the Open Source Edition, of course).
- On Windows (NT/2000/XP), OpenGL is always included and uses the native OpenGL implementations shipped with the OS.
- On Unix, the `configure` script searches for OpenGL header files and includes the OpenGL module if the header files were found.
- Tested OpenGL implementations are Mesa, MetroGL and Microsoft's. Others are likely to work as well.

Classes

The QtOpenGL module consists of five classes:

QGLWidget A widget that represents an OpenGL display buffer and nicely integrates into Qt programs.

QGLContext Represents an OpenGL rendering context. When constructing a `QGLWidget` you give it a context as a parameter.

QGLFormat Provides the display properties of the rendering context (alpha blending, double buffering etc). Set on `QGLContext` using `setFormat()` or globally with the static `QGLContext::setDefaultFormat()`.

Classes cont'd

- **QGLColormap** Represents the OpenGL colormap if the indexed color mode is used.
- **QGLPixelBuffer** For hardware-accelerated rendering into a pixmap, faster than rendering into a `QPixmap`. Also allows using textures if the OpenGL implementation supports that.

Normal Use

- Inherit QGLWidget, and implement the following functions:
  - **initializeGL()** Setting up the OpenGL subsystem. All initialization must be done here and not in the constructor.
  - **paintGL()** Similar to paintEvent, but here you use OpenGL code.
  - **resizeGL()** Alternative to `resizeEvent()`. Set up viewport and projection.
- It is only possible to write OpenGL code directly in the above methods. If you want to use OpenGL in other functions you must first call `makeCurrent()` to set the current context.
- See *examples/opengl/sierpinski*.

Textures

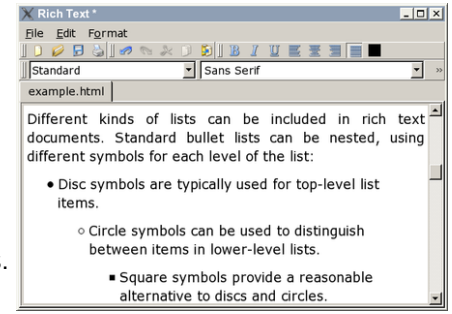
- `QGLWidget`/iClsQGLContext have a number of methods to load images into the OpenGL context for use as texturemaps:
  - **bindTexture(const QImage& image, GLenum target=GL_TEXTURE_2D, GLint format=GL_RGBA8)**: Loads a `QImage` as a texture.
  - **bindTexture(const QPixmap& pixmap, GLenum target=GL_TEXTURE_2D, GLint format=GL_RGBA8)**: Same as above, but using the data from a `QPixmap`.
  - **bindTexture(const QString& fileName)**: Loads DirectDrawSurface (DDS) compressed texture data from `fileName`.

Textures cont'd.

- All `bindTexture()` methods return a handle to the texture (a `GLuint`). This can be used in calls to `glBindTexture` later.
- Clean up with `QGLWidget::deleteTexture(GLUint)`.
- See *examples/opengl/texture*.
- Textures can also be dynamically created from `QGLPixelBuffer` objects using `QGLPixelBuffer::updateDynamicTexture()`.

QTextEdit

- `QTextEdit` is a powerful editor widget which supports a wide range of editing functionalities, including fonts, colors, tables, images, links, alignments, and bullet lists.



QTextEdit cont'd

- `QTextBrowser` is a read-only extension to `QTextEdit`, which provides browsing facilities.
- In addition, a number of widgets allow you to specify the text they display in a subset of HTML.
- Example:

```
QMessageBox::critical( this, "Unable to load file",
  QString( "Unable to load file <b>%1</b>")
     .arg( fileName ) );
```

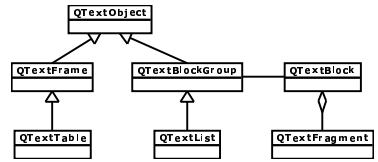

Classes Involved

- **QTextDocument** - Represents a document.
- **QTextCursor** - Represents a cursor on the document. Cursors are used to represent selection, handle movement, and support creating documents.
- **QText*Format** - classes that represent formatting of characters, blocks, tables, etc.
- Classes representing content.

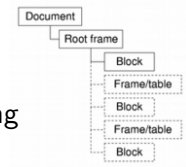
Setting and Reading Text with QTextEdit

- Set the content of a `QTextEdit` using
  - `setDocument(QTextDocument *)`
  - `setPlainText(const QString&)`
  - `setHtml(const QString&)`
- Read the content of a `QTextEdit` using
  - `QTextDocument* QTextEdit::document()`
  - `QString toPlainText()`
  - `QString toHtml()`

Content Classes



Iterating Over the Document



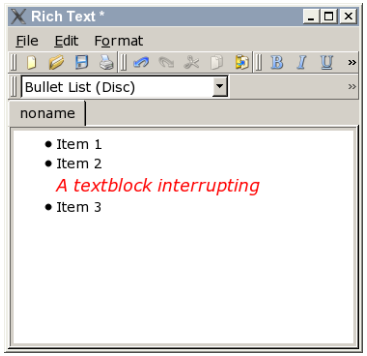
- It is possible to traverse a document node by node.
- You can get the root of the document tree using `QTextDocument::rootFrame()`
- Use `QTextFrame::begin()` to get an iterator.
- The iterator offers `currentBlock()` to get the node as a `QTextBlock`, and `currentFrame()` to get the node as a QTextFrame.

Iterating Over the Document cont'd.

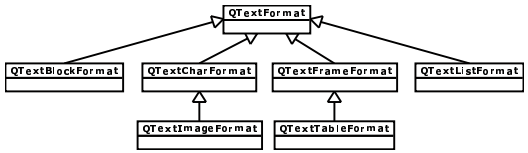
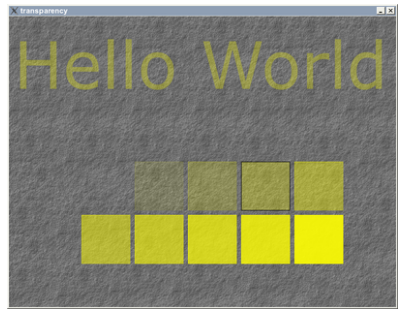
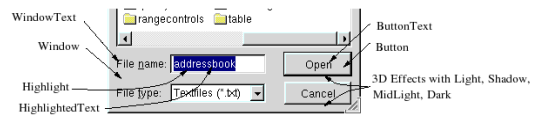
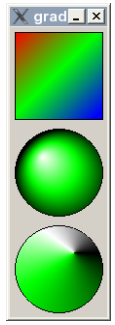
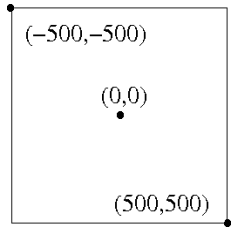
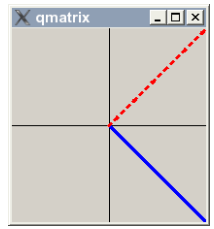
```
QTextDocument* document = ...
QTextFrame::Iterator it = document->rootFrame()->begin();
for ( ; !it.atEnd(); ++it ) {
    QTextFrame* childFrame = it.currentFrame();
    QTextBlock childBlock = it.currentBlock();

    if ( childFrame )
      ...
    else if ( childBlock.isValid() )
      ...
}
```


Iterating Over the Document cont'd.

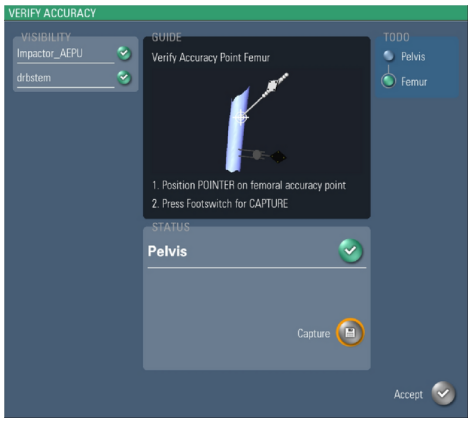
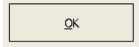
- If you combine the previous three slides, then you will see that the classes `QTextBlockGroup` and `QTextList` are not present when traversing the document.
- The reason is that the items of a `QTextList` may not be continuous.
- Use `QTextBlock::textList()` to get a list of items.



Iterating Over the Document cont'd.

- To iterate over only textual information use `QTextDocument::begin()` and `QTextBlock::next()`
- ```
  QTextBlock block = document->begin();
  while ( block.isValid() ) {
      ...
      block = block.next();
  }
  ```
  ```

Text Blocks vs. Text Fragments

- A text block is split into several fragments. An iterator to a block can be obtained from `QTextBlock::begin()`
- ```
  QTextBlock currentBlock = ...
  QTextBlock::Iterator it = currentBlock.begin();
  for ( ; !it.atEnd(); ++it ) {
      QTextFragment fragment = it.fragment();
      if ( fragment.isValid() ) { ... }
  }
  ```

The text can be written in different styles. Before editing: one fragment

The text **can be written** in different styles. After editing: three fragments

Images

- Images are represented by text fragments that refer to external images via the resource mechanism.
- Use `QTextFragment::charFormat()` to get a `QTextCharFormat` for the fragment, and convert it to a `QTextImageFormat` using `toImageFormat()`:
- ```
  QTextFragment fragment = ...
  QTextCharFormat charFormat = fragment.charFormat();
  QTextImageFormay imageFormat
      = charFormat.toImageFormat();
  if ( imageFormat.isValid() ) {
     ...
  }
  ```


Format classes



QTextCursor

- The class `QTextCursor` is used to modify a document, move the insertion cursor, operate on selections and more.
- You can create a cursor by giving the constructor either a `QTextDocument*`, a `QTextFrame*`, or a `QTextBlock`.
- You can also obtain a cursor from:
  - `QTextEdit::textCursor()` - current insertion cursor
  - `QTextEdit::cursorForPosition()` - cursor for a given position
  - `QTextDocument::find()` - cursor for next match.
- You make a cursor the insertion cursor using `QTextEdit::setTextCursor()`

Inserting using QTextCursor

- QTextCursor contains a number of methods for inserting elements:
  - **insertText(QString, QTextCharFormat)**
  - **insertImage(QString), insertImage(QTextImageFormat)**
  - **insertTable(int rows, int columns, QTextTableFormat)**
  - **insertFrame(QTextFrameFormat)**
  - **insertList(QTextListFormat)**

Selection

- A cursor can span several items.
- If the cursor is the current cursor for a `QTextEdit`, such a span will be displayed as a selection. You make the cursor current using `QTextEdit::setTextCursor()`.
- A span can be created programmatically using code similar to:

```
cursor.movePosition( QTextCursor::StartOfWord );
cursor.movePosition( QTextCursor::EndOfWord,
                     QTextCursor::KeepAnchor );
```


Syntax Highlighting

- In order to get syntax-highlighting in a `QTextEdit`, create a subclass of `QSyntaxHighlighter` and pass a pointer to the `QTextEdit` or `QTextDocument` to it in the constructor.
- `QTextEdit` will call the `highlightBlock()` method of your class for each piece of text.
- Put together a `QTextCharFormat` object with the requested font/color settings settings and pass this to `QSyntaxHighlighter::setFormat()`.
- Convenience methods exist for setting a font or a color directly.

Miscellaneous

- Using `QTextCursor::beginEditBlock()` and `QTextCursor::endEditBlock()` it is possible to implement operations which from a undo/redo point of view seem like atomic operations.

Project Task

- Starting from *handout/texteditor-advanced*, add the following features:
- Insert a table (Editor::slotInsertTable()).
- Set boldface, italic and underline (Editor::slotBoldface(), Editor::slotItalic(), and Editor::slotUnderline()).
- Make sure the tool buttons for the above actions are updated when the cursor moves around.
- Implement Search and Replace (class ReplaceProgressDialog).
- Optional: Inserting lists (Editor::slotInsertList()).
- Optional: Insert Frames (Editor::slotInsertTextBox()).

Creating Color Values

- **RGB**: `QColor(`*`red, green, blue, alpha`*`);`
- **HSV**: `QColor::fromHsv(`*`hue, saturation, value, alpha`*`)`
- **CMYK**: `QColor::fromCmyk(`*`cyan, magenta, yellow, black, alpha`*`)`
- **by name**: `QColor mycolor("MidnightBlue")` (file *rgb.txt*)
- **static colors in Qt namespace** black, white, darkGray, gray, lightGray, red, green, blue, cyan, magenta, yellow, darkRed, darkGreen, darkBlue, darkCyan, darkMagenta, darkYellow (access with e.g. Qt::green).
- **RGB**: `::qRgb(...)` and `::qRgba(...)`

QColor vs. QRgb

- `QColor` is a class with numerous methods for manipulation, e.g. `QColor::light()`.
- `QRgb` is equal to an unsigned int; `qRgb()` returns a `QRgb`.
- `qRgb` and `qRgba` are used for low level access in `QImage`, and `QGLColormap`.
- `QColor` has a number of static methods for accessing components in `qRgb` and `qRgba`: `qRed()`, `qGreen()`, `qBlue()`, `qAlpha()`, `qGray()`.

Transparency

- Transparency effects are obtained by drawing with a color that has an alpha channel.
- The alpha channel is specified with a number between 0 (completely transparent) and 255 (completely opaque)
- See example *transparency*



Shaped Windows

- *Talking about transparency...*
- It is possible to set a mask on a widget which specifies its area. Using this technique, it is possible to create non-rectangular widgets and windows.
- A mask is set using `QWidget::setMask(const QBitmap&)`
- A `QBitmap` is a subclass of `QPixmap` with 1 bit depth.
- You can draw on a `QBitmap` using a `QPainter`.
- The colors `color0` and `color1` are for drawing on bitmaps.

Shaped Windows cont'd.

- You can ask a `QImage` if it has an alpha channel using `QImage::hasAlphaChannel()`.
- Using `QImage::createHeuristicMask()` you can get a mask from an image which does not have an alpha channel.
- **Note:** Complex masks can be very slow!
- See the example *tux*.

Color Groups and Palettes

- Theming in Qt makes changing colors on widgets slightly more difficult than just ~~`setColor(blue)`~~.
- Colors are managed through *palettes* (class `QPalette`), which consist of three color groups: *active*, *inactive*, and *disabled*.
- You set a palette for a single widget using `QWidget::setPalette()`, and as a default for the whole application using `QApplication::setPalette()`

Color Groups and Palettes cont'd.

- A color group consists of matching colors for *window text*, *general button color*, *light shadows*, *midlight shadows*, *dark shadows*, *mid shadows and contrasts*, *text*, *bright text*, *button text*, *base*, *window*, *shadow*, *highlight*, and *highlighted text*.



Color Groups and Palettes cont'd.

- A given role can be fetched using either `QColor QPalette::color()` or `QBrush QPalette::brush()`.
- A color can be set in the palette using `QPalette::setColor()` and `QPalette::setBrush()`.
- Notice, however, that internally in QPalette it is always stored as a QBrush, so it is really just one setting.
- The above functions are overloaded to take either a color role, or a color group and a color role.

Pens

- A pen (`QPen`) consists of:
  - **a color or brush**
  - **a width**
  - **a style** (`NoPen`, `SolidLine`, `DashLine`, `DotLine`, `DashDotLine`, and `DashDotDotLine`).
  - **a cap style**, specifies how a line ends (`FlatCap`, `SquareCap`, `RoundCap`).
  - **a join style**, specifies how two connect lines (`MilterJoin`, `BevelJoin`, `RoundJoin`).
- Pens are activated with `QPainter::setPen()`.

Brushes

- Brushes are used for filling shapes.
- A brush can be one of the following:
  - A color plus a brush style
  - A pixmap
  - A gradient
- Brushes are activated with `QPainter::setBrush()`.

Brushes cont'd.

- Available brush styles are: `NoBrush`, `SolidPattern`, `Dense1Pattern`, `Dense2Pattern`, `Dense3Pattern`, `Dense4Pattern`, `Dense5Pattern`, `Dense6Pattern`, `Dense7Pattern`, `HorPattern`, `VerPattern`, `CrossPattern`, `BDiagPattern`, `FDiagPattern`, `DiagCrossPattern`, and `TexturePattern` (use `QBrush::setTexture()` for the latter).
- See `QBrush` class description for a detailed description.

Brushes cont'd.

- QBrush supports three different kind of gradients: `QLinearGradient`, `QRadialGradient`, and `QConicalGradient`

```
QLinearGradient gradient(0, 0, 100, 100);
gradient.setColorAt(0, Qt::red);
gradient.setColorAt(0.5, Qt::green);
gradient.setColorAt(1, Qt::blue);
painter.setBrush(gradient);
painter.drawRect(0, 0, 100, 100 );
```



- See example *gradients*.

Brushes cont'd

- It is possible to set a brush on a pen, which means that the strokes generated with the pen will be filled with the brush.
- See example *pen_with_brush*

The Outline

- The rule for drawing the outline is: *The outline will follow the shape with half the pen width on each side.*.
- For a pen of 1, the outline for a rectangle will start in (x-0.5, y-0.5) and extend to (x+w+0.5, y+h+0.5).
- Due to integer rounding on a non-antialiased grid, the outline is shifted half a pixel towards the bottom right, and thus covers (x,y) to (x+w+1,y+h+1).
- See example *rect-outline*

Auxiliary Drawing Classes

- **QPoint**/**QPointF** - represent a point (x,y).
- **QSize**/**QSizeF** - represent a size (width,height).
- **QRect**/**QRectF** - a point and a size (x,y,width,height).
- **QPolygon**/**QPolygonF** - A list of points.
- **QRegion** - a region (built from rectangles, ellipses, and polygons)
- **QPainterPath** - a container for painting operations, often a faster alternative to `QRegion`

Drawing Operations

- Available drawing operations in QPainter `drawArc`, `drawChord`, `drawConvexPolygon`, `drawEllipse`, `drawImage`, `drawLine`, `drawLines`, `drawPath`, `drawPicture`, `drawPie`, `drawPixmap`, `drawPoint`, `drawPoints`, `drawPolygon`, `drawPolyline`, `drawRect`, `drawRects`, `drawRoundRect`, `drawText`, `drawTiledPixmap`.

Double Buffering and Backing Store

- Each top-level widget in Qt has a double buffer, a *backing store*. (Exception: MacOS/X, which double-buffers natively.)
- All widgets are composited from parent to child into this buffer. This allows subwidget transparency.
- The backing store is kept in sync with the screen automatically.
- Operating system-level expose events will not result in `paintEvent()` calls.

Double Buffering and Backing Store cont'd.

- Since Qt 4.1, widgets do not paint their background any more by default, use `QWidget::setAutoFillBackground(true)` in order to get the old behavior.
- Double buffering can be turned off for a given widget on X11 and on embedded systems by setting the `Qt::WA_PaintOnScreen` attribute. Still painting to widgets is only allowed from within `paintEvent()`.

Clipping

- With clipping, you specify the area the painter will work on.
- You can ask a painter if clipping is enabled using `hasClipping()`
- Enable clipping using `setClipRect()`, `setClipRegion()`, and `setClipPath()`
- Each of the methods above takes an optional enum specifying if the clip should be *united*, *replaced*, or *intersected* with existing clipping.

Clipping

- Note that clipping can be slow. It is completely system-dependent, but as a rule of thumb, you can assume that drawing speed is inversely proportional to the number of rectangles in the clip region.
- Widgets are composed, therefore child areas are not clipped away by default. A child that is rectangular and fully opaque will still cover all of its parent.
- See example *puzzle*

Painter Paths

- The class `QPainterPath` is used to draw complicated objects.
- It has several advantages over normal QPainter usage:
  - It is possible to fill shapes that would require complicated clipping without painter path.
  - Unlike regions, painter paths can be transformed, and you can use the same painter path for filling, stroking, and clipping. (Regions can be transformed, too, but that is costly and loses precision.)

Painter Paths cont'd.

- You draw a painter path using `QPainter::drawPath()`, `QPainter::strokePath()`, or `QPainter::fillPath()`.
- The bounding rectangle for a path can be fetched using `QPainterPath::boundingRect()`
- `QPainterPath::controlPointRect()` returns a rectangle enclosing the control points. The size might be larger than the result from `boundingRect()`, but is faster to calculate.

Painter Paths cont'd.

- Closed or open painter paths can be built using low-level methods like `moveTo()`, `lineTo()`, `arcTo()`, `quadTo()`, `cubicTo()`, and `closeSubPath()`.
- Convenience functions for common cases: `addEllipse()`, `addRect()`, `addRegion()`, and `addText()`.
- It is possible to connect two paths using `connectPath()`. This method connects the last point of the current path with the first point of the other path.

Painter Paths cont'd.

- `QPainterPath` has two algorithms for filling, select with `setFillRule()`:
  - **OddEvenFill** Fill a point if it takes an odd number of lines to cross to get outside of the path.
  - **WindingFill** Fill everything inside a closed shape
- These descriptions are simplified, see the documentation of `Qt::FillRule` for the whole truth.
- See example *house-drawings*

Anti-Aliasing

- Anti-aliasing makes lines smoother, but is also slower to paint.
- On X11, anti-aliasing requires the X Render extension, when drawing to widgets or pixmap.
- Enable anti-aliasing for drawing using `QPainter::setRenderHint(QPainter::Antialiasing)`.
- See example *anti-aliased*

Rubberbanding

- In applications in which it is possible to select items, rubberbanding is an often used technique in which the mouse is dragged over the items to be selected.
- In applications where you do the drawing yourself, you can simply choose to draw the selection rectangle yourself.
- Alternatively, the class `QRubberBand` can be used.

Two-dimensional Transformations

- The coordinate systems of `QPainter` objects can be rotated, sheared, scaled, etc. This makes it easy to draw e.g. rotated text.
- Transformations: `QPainter::scale()`, `QPainter::shear()`, `QPainter::rotate()`, `QPainter::translate()`
- Utilities: `QPainter::save()`, `QPainter::restore()`, and `QPainter::resetMatrix()`.

Example

```
void drawRotatedText( QPainter *painter, float degrees,
                      int x, int y, const QString& text)
{
    // Precondition: painter->begin() has been called.
    painter->save();
    painter->translate( x, y );
    painter->rotate( degrees );
    painter->drawText( 0, 0, text );
    painter->restore();
}
```


Two-dimensional Transformations cont'd

- If the predefined transformations are not sufficient, you can also create a `QMatrix` object and then call `QPainter::setMatrix()`.
- To facilitate creation of `QMatrix` objects, this class supports the methods `rotate()`, `translate()`, `shear()` and `scale()` as well.
- A constructor for QMatrix exists, which sets the individual items in the transformation matrix.

  ```
  QMatrix ( double m11, double m12, double m21, double m22,
            double dx, double dy )
  ```

- The transformation formula is as follows:

  ```
  x' = m11*x + m21*y + dx    y' = m22*y + m12*x + dy
  ```


Example cont'd

```
void drawRotatedText( QPainter *painter, float degrees,
                      int x, int y, const QString& text)
{
    // Precondition: painter->begin() has been called.
    QMatrix oldMatrix = painter->worldMatrix();
    QMatrix matrix = oldMatrix;
    matrix.translate( x, y );
    matrix.rotate( degrees );
    painter->setMatrix( matrix );
    painter->drawText( 0, 0, text );
    painter->setMatrix( oldMatrix );
}
```


Window Transformations

- Window transformations are used to normalize the model coordinate system, so that the painting code can be used for different physical sizes.
- Window transformations are enabled by calling `QPainter::setWindow()`.
- Example:
  `setWindow(-500,-500,1000,1000)`



Viewport Transformations

- The part of the paint device that is active is called the viewport.
- Viewport transformation changes this area.
- Viewport transformations are enabled by calling `QPainter::setViewport()`.

Normal Usage of Transformations

- Normally, viewport and window transformations are used together. In that case the viewport coordinates tell which part of the physical surface is to be drawn to, and the window coordinates specify the model coordinates for this area.
- If viewport transformation is not specified then it defaults to the whole physical area.
- If window transformation is not specified then the viewport transformation scales the coordinates, compared to the original viewport.

Miscellaneous

- The world matrix positions the objects and QPainter::setWindow() positions the window, specifying what coordinates will be visible.
- Viewport transformations are always done in device coordinates.
- **Note:** `QPens` are scaled as expected, however, a pen with width 0 is a 1 pixel-wide pen that is not scaled.
- See the examples `Analog Clock` and `Transformations` from the Qt examples.

Customized QMatrix (examples/qmatrix)




```
QPainter painter(this);
painter.setWindow( -100,-100,200,200 );
painter.drawLine( -100,0,100,0 );
painter.drawLine( 0, -100, 0, 100 );

QPen pen;
pen.setWidth(3); pen.setColor( Qt::blue );
painter.setPen( pen );
painter.drawLine( 0,0,100,100 );

QMatrix matrix( 1, 0, 0, -1, 0, 0 );
painter.setMatrix( matrix );
pen.setColor( Qt::red ); pen.setStyle( Qt::DotLine );
painter.setPen( pen ); painter.drawLine( 0,0,100,100 );
```

Questions And Answers

1. Which classes would you use for drawing a filled rectangle?
2. How do you draw a circle with QPainter?
3. On which types of objects can you use a QPainter?
4. Name the available coordinate transformation methods.
5. When would you use a QMatrix?
6. What is the difference between window transformation and viewport transformation?

Project Task: Implementing Figure Drawing in the Paint Program

- Finish the *handout/paintProgramForShapeDrawing* program so that it can draw circles and rectangles. (Hint: You will need an extra temporary pixmap for the rubber banding)
- Implement brushes, and add anti-aliasing.
- Optional: When dragging out new shapes, use alpha blending to allow the user to see what he would be covering.
- Optional: Complete `ScribbleArea::drawArrow()` to add an arrow to lines dragged out (This requires that line drawing is changed from freehand drawing to dragging out lines)
- Optional: implement functions for inserting images.

Using Widget Styles



This screenshot is using a style developed by Klarälvdalens Datakonsult AB for Medivision. Everything you see are regular Qt widgets, styled so you will not even be able to tell what is what. More screenshots are available from http://www.kdab.net/medivision

Using Widget Styles

- Styles change the appearance of standard widgets: different drawing code, different metrics.
- How to see the existing styles:
  - "Preview" in Qt designer
  - Qt "widgets/styles" example
  - Passing -style motif, -style plastique ... on the command line
  - In code: `QApplication::setStyle()`
  - Default style: stored in `QSettings`. Configurable with qtconfig on Unix.

The Delegation Mechanism

- In Qt, styles are implemented as a separate class, `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
- The style does *not* change the real behavior of the widget, only its appearence and GUI details of the user interface.

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

- For testing:

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


The styleHint() method

- Styles can reimplement `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
  - `SH_LineEdit_PasswordCharacter` - the character to be used for passwords
  - ...

styleHint(): Example Implementation

```
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);
    }
}
```

The drawControl() Method

- `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
  - ...

Implementing 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 to
    // be visible inside the rectangle, see QPainter documentati
    p->drawRect( opt->rect.adjusted(0,0,-1,-1) );
    break;
  default:
    QWindowsStyle::drawControl( ce, opt, p, w );
  }
}
```

Windows: 

TestStyle: 

QStyleOption

- QStyleOption groups many parameters into a single struct:
  - **rect()** - the widget rectangle
  - **palette()** - the widget palette
  - **state()** - the widget state (enabled, has focus, mouse over, sunken, raised etc.)
- 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.

QStyleOption subclasses

- All drawing methods take a QStyleOption.
- To access widget-specific parameters, cast a QStyleOption to the appropriate subclass using `qstyleoption_cast<T>`
- `qstyleoption_cast<T>(opt)` returns 0 if opt is not of the correct type.
- Example:

```
case CE_PushButton:
  const QStyleOptionButton *btn =
    qstyleoption_cast<const QStyleOptionButton *>(opt);
```

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`.

The drawPrimitive method

- `drawPrimitive()` is 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

The drawComplexControl() method

- `drawComplexControl()` is called for drawing widgets with multiple sub-controls.
- ComplexControl is an enum with the following values:
  - `CC_SpinBox`
  - `CC_ComboBox`
  - `CC_ScrollBar`
  - `CC_Slider`
  - `CC_ToolButton`
  - `CC_TitleBar`

drawComplexControl() cont'd

- Each of those is divided into sub-controls, which the SubControl enum represents. For instance, a spinbox has `SC_SpinBoxUp`, `SC_SpinBoxDown`, `SC_SpinBoxFrame` and `SC_SpinBoxEditField`.
- `QStyleOptionComplex::subControls()` indicates which sub-controls should be painted, and `QStyleOptionComplex::activeSubControls()` indicates which ones are active (pressed), which one is under the mouse, etc.
- Styles can reimplement the virtual `subControlRect()` method to define where each subcontrol is; this is used by e.g. `hitTestComplexControl`.

The pixelMetric() method

- A pixel metric is a style-dependent size represented as a single pixel value.
- Styles reimplement 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)

polish(): Example

- Example: how to change the color of a `QPushButton` when the mouse is over the button.
- ```
  void TestStyle::polish( QWidget* w )
  {
      if ( qobject_cast<QPushButton *>( w ) ) {
          w->setAttribute( Qt::WA_Hover );
      }
      QWindowsStyle::polish( w );
  }
  ```
- Then the drawing code can use `opt->state & State_MouseOver` to know when the mouse is over the widget.
  ```

Colors

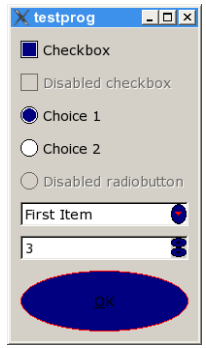
- Styles can provide a standardPalette() with style-specific colors.
- Avoid hard-coded colors elsewhere in the style code, use the palette instead.
- For application-specific styles, the palette should be set in `main()`.
- The user can configure a color theme (e.g. with qtconfig on Unix).

Finding Solutions

- The `QStyle` documentation—a starting point, but not enough to figure it all out
- The source code for `qstyle.cpp`, `qcommonstyle.cpp`, and actual styles (e.g. `qwindowsstyle.cpp`)
- The widget code—e.g. for finding out that QWindowsStyle::drawPrimitive() uses the ButtonText color to draw the arrow in a combobox.
- For pixmap backgrounds, see the "widgets/styles" Qt example

Project Task: Styling Radio Buttons with a Plain Circle

- Start from *handout/widgetStyle*.
- Launch it with `./testprog -style teststyle`
- Make the radio buttons as big as the check boxes
- Paint radio buttons as a plain circle - filled correctly
- Paint checked radio buttons with a circle
- Optional: anything else from the screen shot



QMake

- QMake is a makefile generator.
- Supports multiple platforms and multiple C++ compilers.
- Automatically maintains dependencies.
- Includes support for MOC, UI and QRC files.
- Is very terse—you do not have to write much.
- Supports compiling both applications and libraries.
- Supports Unix-style Makefiles, nmake Makefiles, Visual C++ and VS.NET, and MacOS Xcode, among others.

Installation

- Quick Start: type `qmake -project` to make QMake try to make a project file for you. The more source files you already have in place, the less you will have to type later.
- You should name your *.pro* files *directory.pro*, this will help you use features like automatic building in subdirectories.

Variables

- QMake consists of a number of *System Variables* which define what the generated Makefiles or dsp files should do.
- All System Variables are written in capital letters.
- You can also add your own variables.
- Variables can be referenced using `$$`:

```
PARSER = myparser.cpp
SOURCES= main.cpp $$PARSER
```

- Lines can be split using backslashes.
- # is the comment character; comments extend till the end of the line.

Most Common System Variables

You will almost always use the following system variables:

- **TEMPLATE** Defines the type of project. Must be one of:
  - **app, vcapp** Creates a binary.
  - **lib, vclib** Creates a library.
  - **subdirs** Invokes make recursively for subdirectories.
- **HEADERS** List of header files.
- **SOURCES** List of (C/C++) source files.
- **TARGET** Name of the executable (adds .exe on Windows). Defaults to the name of the .pro file/directory.

Most Common System Variables cont'd

- **QT** Specifies the Qt *features* to be included. Core and GUI are included by default. To make other feature available, use

  `QT += xml sql network`

  If you do not need the GUI classes (e.g., because you are building a server application), use

  `QT -= gui`

  When porting from Qt 3, you want to add `qt3support` here in order to link to the Qt3Support library and get access to the backwards compatibility functions. Try turning this off before release, though!

Other Useful System Variables

- **DEFINES** List of C preprocessor macros (-D option).
- **LIBS** List of additional libraries to be linked in.
- **INCLUDEPATH** Sets the include search path (-I option).
- **DEPENDPATH** Sets the dependency search path for QMake.
- **DESTDIR** Specifies where to put the target file.
- **MOC_DIR** Specifies directory to place moc files in.
- **OBJECT_DIR** Specifies directory to place object files in.

System Variables

- **LEXSOURCES** Specifies a list of lex files.
- **YACCSOURCES** Specifies a list of yacc sources.
- **SUBDIRS** Specifies a list of directories to run make recursively in.
- **VERSION** Specifies the version for the library template.
- **FORMS** Specifies interfaces (`.ui`) files from Qt Designer.
- **RESOURCES** Specifies resource collection (`.qrc`) files to include in build.

The `CONFIG` System Variable

- The system variable CONFIG is a list of system configurations. The following options exist:
  - **release** Compile with optimization enabled, ignored if `debug` is specified.
  - **debug** Compile with debug options enabled.
  - **warn_on** The compiler should emit more warnings than normally, ignored if `warn_off` is specified.
  - **warn_off** The compiler should emit no warnings or as few as possible.
  - **ordered** Order subdirectories so parallelized builds work.

The `CONFIG` System Variable

- **qt** The target is a Qt application/library and requires Qt header files/library.
- **opengl** The target requires the OpenGL (or Mesa) headers/libraries.
- **thread** The target is a multi threaded application or library.
- **x11** The target is an X11 application or library.
- **windows** The target is a Win32 window application.
- **console** The target is a Win32 console application.
- **shared** The target is a shared object/DLL.
- **static** The target is a static library.

The CONFIG System Variable

- **plugin** The target is a plugin (requires `TARGET=lib`, and implies `shared`).
- **exceptions** Turns on compiler exception support.
- **rtti** Turns on compiler RTTI support.
- **stl** Turns on Qt STL support.
- **flat** *only for TEMPLATE=vcapp*; puts all sources files into one group and all header files into another group, independent of the directory structure.

You can even assign any values of your own to the `CONFIG` variable and use these as scope qualifiers later (see below).

subdirs template

```
test.pro:
  TEMPLATE = subdirs
  SUBDIRS = dirA dirB dirC
```

That's it!

- If QMake is used in the subdirectories to generate the Makefiles, the project files in these directories should be called <dir-name>.pro.
- Example: *dirA/dirA.pro*. This makes it possible for qmake to execute qmake in the subdirectories.

Modifying Variables

- Variables can be modified in a number of ways:

```
A   = abc
X   = xyz
A  += abc def       # A = abc abc def
B   = $$A           # B = abc abc def
B  -= abc           # B = def
B  *= abc def       # B = abc def
B  ~= s/ab[xc]/xyz/ # B = xyz def
```

- This can be used to modify variables from the command line:

```
qmake -after "LIBS+=-pg" "CONFIG+=debug" "CONFIG-=release"
```


Scopes

- You can use scopes to conditionally assign to variables.
- Scopes can be specified based on the following things:
  - OS mode (`win32`, `unix`, `mac`, ...)
  - Patterns matching the platform (`QMAKESPEC` environment variable, or `-platform` command line option): `solaris-cc` or `linux-*` or `*-g++*` or `*-64` etc.
  - Options appearing in the `CONFIG` variable, including your own.
  - The result of a function (described later)

Scopes

- Scopes can be specified in "concatenation" syntax: $scope_1$:$scope_2$:...:$scope_n$:var = $value$
- Examples:

```
SOURCES = common.cpp
win32:SOURCES += win_hack.cpp # OS.
unix:profile:LIBS += -pg # profile is a user supplied
                         # option from the CONFIG
                         # variable.
```


Scopes

- Scopes can also be specified in groups using { ... }:

```
win32 {
  debug {
    SOURCES += debug.cpp    # win32 && debug
  }
  release {
    SOURCES += release.cpp # win32 && release
  }
}
```

- Be aware that with the { ... } syntax, the opening brace must be on the same line as the test, and the closing brace must be on a line by itself.

Scopes

- The two methods can be combined.

```
win32 {
  debug:SOURCES += debug.cpp
  release:SOURCES += release.cpp
}
```

- Scopes can be negated with an exclamation mark:

```
network:SOURCES += network.cpp
!network:SOURCES += no_network.cpp
```

- Alternatively you may use *else*.

Functions

- A number of predefined functions exists, they are split in two categories:
  - **test functions** used as part of tests.
  - **replacement functions** used to create strings (i.e., right-hand-side values).
- It is also possible to implement your own functions, which is beyond the scope of this course, please refer to the reference manual.

Test Functions

- **isEmpty(variablename)** This function will succeed if the variable *variablename* is empty (same as count(variable, 0)).
- **CONFIG(config)** Like a scope qualifier, but allows mutual exclusions to be specified. Rarely needed, the specifiers that are useful here should not go into the *.pro* file anyway, but rather be specified on the command line.
- **count(variablename, number)** This function will succeed if the variable *variablename* contains *number* elements.
- **contains(variablename, value)** This function will succeed if *variablename* contains the value *value*.

Test functions cont'd.

- **system(command)** This function will execute the command in a secondary shell and will succeed if the command exits with an exit status of 0.
- **error(txt)** This function will never return. It will display the given string to the user, and then exit qmake.
- **message(txt)** This function will always succeed, and will display the given string to the user. `warning()` is a synonym.
- **infile(filename, var, val)** Succeeds if *filename*, as parsed by QMake, contains a variable *var* set to *val*. If *val* is not specified, it tests for the mere declaration of *var*.

Test functions cont'd.

- **include(filename)** This function will include the contents of *filename* into the current project at the point it was included. The function succeeds if filename was included.
- **exists(filename)** Succeeds if *filename* exists. If you must have a certain file for proceeding further, you can use `error()` (see below) for terminating in case this file does not exist:

```
!exists( file.dat ):error( "file.dat missing" )
```


Replacement Functions

- **basename(file)** This function returns the basename of *file*, i.e., the filename stripped of the path and a possible extension. The path can be extracted with `dirname()`.
- **dirname(file)** Returns the path of the directory containing the *file*.
- **prompt(question)** Displays question, waits for input from standard input, and returns that input. **Warning:** This will break automated builds, unless standard input is redirected. It might be useful for very simple install scripts, though.

Replacement Functions cont'd.

- **system(command)** This function will execute the command in a secondary shell and will return stdout and sdterr concatenated.
- **find(variablename, substr)** Returns all values in *variablename* that match the regular expression *substr*. See the `QRegExp` documentation for the regular expression syntax.
- **for(iterator, list)** Iterates over *list*, using the loop variable *iterator*. The syntax *<num>..<num>* is supported as well.
- **join(varname, glue, before, after)** Joins the value of *variablename* according to what `QStringList::join()` does.

Replacement Functions cont'd.

- **member(varname, position)** Equivalent to `QList::at()`.
- **quote(string)** Turns *string* into a single entity and returns that.
- **unique(varname)** Removes duplicate values from *varname* and returns the new list.
- **sprintf(string, arguments...)** Like `QString::sprintf()`, but uses %1-%9 for the placeholders.

Internationalization

- *Internationalization (I18N)* is the task of making a program so general that it can be used with different natural languages, in different cultures, etc.
- *Localization (L10N)* is the task of adapting the program to one combination of language, culture, customs, etc. — a so-called *locale*.
- Qt provides tools to do both for on-screen texts, and number formats.

Internationalizing a Program

- Identify all user-visible strings, i.e., texts in dialogs, menus, tooltips, etc.
- When you pass user-visible texts in the program, use `QString` and `const QString&` instead of `char*` or `const char*` in order to prepare for Unicode.
- Surround each of these strings with `tr()`:

```
QPushButton* pb = new QPushButton(tr("Cancel"), this);
```

- `tr` can take a second argument, which will be shown as a hint to the translators and is also used to distinguish identical strings that may need to be translated differently.
- Outside of a `QObject` subclass, such as in a global function, use `QCoreApplication::translate()`.

Internationalizing a Program cont'd.

- If you are in a context where no function can be called (e.g., in a static initializer), use Q_TR_NOOP or Q_TRANSLATE_NOOP:

```
static const char* continents[] = {
    QT_TR_NOOP( "Africa" ),
    QT_TR_NOOP( "America" ),
    QT_TR_NOOP( "Antarctica" ),
    QT_TR_NOOP( "Asia" ),
    QT_TR_NOOP( "Australia" ),
    QT_TR_NOOP( "Europe" )
};
```

- These expand to just the string, but mark the string up for translation (see below).
- QT_TRANSLATE_NOOP lets you specify a context.

Internationalizing a Program cont'd.

- To make it possible to change accelerators, these must be translated too:

```
QAction* load =
  new QAction ( tr("Load from a file"),
       loadIcon, tr("&Load"),
       QKeySequence(tr("CTRL+L", "load action")))
```

- Avoid hard-coded positions and sizes, use geometry management instead.

Internationalizing a Program cont'd

- If you have variable parts in on-screen texts, use QString::arg() which allows the translator to move the variable parts around:

```
statuslabel->setText( tr( "Saved file %1" )
                      .arg( filename ) );
```

- QLocale contains useful methods for converting numbers to strings and back according to the current locale (taking into account cultural differences like decimal point/comma, etc.)

Useful Preprocessor Macros

- QT_NO_CAST_TO_ASCII prevents automatic casts from QString to char*.
- QT_NO_CAST_FROM_ASCII prevents automatic casts from char* to QString. It helps you remember all locations of strings which must be translated.
- In QMake, you can specify these with:

```
DEFINES += QT_NO_CAST_TO_ASCII
DEFINES += QT_NO_CAST_FROM_ASCII
```

Localizing a Program

- Add a line to your QMake project file listing all the files containing translations (one for each language):

  `TRANSLATIONS=editor_da.ts editor_fi.ts`
- Run `bin/lupdate` on your `.pro` file to extract strings.
- Run `bin/linguist` on each of the `.ts` files to translate.
- Run `bin/lrelease` on the `.pro` file to generate an *object* file.
- For subsequent updates of the translations rerun the steps above. Existing translations will be preserved.

Localizing a Program cont'd.

- In your program, add code like the following near the start:

```
QTranslator qt_translator;
QTranslator app_translator;
if( usersettings.language == DUTCH ) {
  // Qt's own translations
  qt_translator.load("qt_nl");
  qApp->installTranslator( &qt_translator );
  // application translation
  app_translator.load("myprogram_nl.qm");
  qApp->installTranslator( &app_translator );
}
```

- All texts will automatically appear in Dutch now.

Localizing a Program cont'd.

- Alternatively, use system settings:

```
QTranslator qt_translator;
QTranslator app_translator;

// Qt's own translations
qt_translator.load("qt_" + QLocale::system().name());
qApp->installTranslator( &qt_translator );

// application translations
QString file = "myprogram" + QLocale::system().name();
app_translator.load(file);
qApp->installTranslator( &app_translator );
```


Project Task: Translate the Paint Program

- Translate the paint program.
- Include the macros from page 658 to ensure that you translate all strings.
- The strings containing the text "`*.bmp *.png *.jpg`" is likely translated in your version, is that a good idea? Change it in the source file, and retranslate the application.
- Optionally: Translate the shortcuts.
- Optionally: Compile the translation into the binary as described on page 200ff.

Codecs

- Traditionally, characters have been represented using 8 bit, which is fine for most western languages such as English, German, and Swedish.
- Different languages would use different encodings of the 8 bits, as a sum of all different letters would amount to more than 255 characters.
- Disadvantage: Some languages, like Chinese and Japanese, need more than 8 bit.
- Disadvantage: Given a file, you need to know the encoding to be able to read it.

Codecs cont'd.

- The solution to the above problem is Unicode (ISO 10646), which is one encoding for most available characters in the world.
- A special case of encoding is UTF8, which is a variable-length encoding, UTF8 is interesting since it preserves Unicode information while looking like plain US-ASCII if the text is wholly US-ASCII.
- Unfortunately, Unicode is not supported by all available operating systems and hardware, and therefore an 8 bit encoding might sometimes be needed.

Codecs cont'd.

- To map between an 8-bit encoding and Unicode, a codec ("**co**der/**dec**oder") is needed.
- Codecs are represented by the subclasses of `QTextCodec`.
- Qt contains a number of codecs for often-used languages.
- If needed you can implement your own codecs (Ask Trolltech before doing so in order to avoid duplicated work).

Codecs cont'd.

- `QTextCodec::codecForName()` takes a codec name as a string, and returns a QTextCodec instance. See the QTextCodec reference page for the complete list.
- `QTextCodec::availableCodecs()` returns a `QList<QByteArray>` that you can use in order to traverse the available codecs.
- `QTextCodec::name()` returns the name of a codec - useful in a popup menu offering different codecs.

Codecs cont'd.

- You convert to and from Unicode by means of `QTextCodec::fromUnicode()` and `QTextCodec::toUnicode()`.
- If you read characters from a stream, you may need to use a QTextDecoder to avoid chopping a multi-byte character in the middle. Call `QTextCodec::makeDecoder()` to get one.
- QString offers four sets of convenience methods for transforming: toAscii/fromAscii, toUtf8/fromUtf8, toLatin1/fromLatin1, and toLocal8Bit/fromLocal8Bit (plus raw-UTF16 conversions).
- Windows and MacOS X already are Unicode-aware, a few Unix versions as well. We can expect not to need any codecs any longer in a few years (but have been saying this for a few years already...).

Default Codec

- At application startup, a codec is activated depending on system settings (for example the LANG variable - see the source code for `QTextCodec::codecForLocale()`.)
- You can also set the default codec using `QTextCodec::setCodecForTr()`;
- See example *codec*.

XML APIs in Qt

- Qt provides two different means of accessing XML data:
- SAX (version 2), which provides a sequential view on an XML document ("Builder" design pattern)
- DOM (level 1 and 2), which provides a tree view on an XML document ("Composite" design pattern)
- In order to use the XML support, add `QT += xml` to your QMake files.

Using SAX

- Create a `QXmlInputSource` object by passing a `QFile` or a `QTextStream`.
- Create an XML reader class that inherits `QXmlReader`. Normally, it's enough to use `QXmlSimpleReader`.
- Create a handler for the SAX events you want to handle.
- Register the handler with the reader.
- Call `parse()` on the reader, passing the `QXmlInputSource`.

SAX Handlers

- There are six abstract handler classes that define SAX events:
  - `QXmlContentHandler` for content-related events
  - `QXmlErrorHandler` for errors, fatal errors, and warnings
  - `QXmlEntityResolver` for external entities that need resolution
  - `QXmlDTDHandler` for DTD-related events
  - `QXmlDeclHandler` for rarely used DTD-related events
  - `QXmlLexicalHandler` for events related to the lexical structure. (start/end of DTD, comments,...)
- It is easiest to subclass your handler class from `QXmlDefaultHandler` which provides empty implementation for all callbacks.

SAX Handlers cont'd.

- When using QXmlDefaultHandler, you still need to register it for each role it has:

```
class MyHandler :public QXmlDefaultHandler { ... }
MyHandler handler;
QXmlSimpleReader reader;
reader.setContentHandler( &handler );
reader.setErrorHandler( &handler );
```

- see `examples/xml/saxbookmarks` for an example.

DOM - The Document Object Model

- DOM is a standard for accessing XML data in tree form.
- The entire XML document needs to be kept in memory in a DOM implementation.
- Everything in DOM is a *node* (in Qt: `QDomNode`).
- The DOM interface is specified by the W3C, not by Trolltech.
- The DOM interface is awkward in places, but has the advantage of being standardized.

Working with DOM

- Create a `QDomDocument`.
- Call `QDomDocument::setContent()`, passing a string, a file, or a byte array.
- Access the root element with `QDomDocument::documentElement()`.
- Iterate over the tree with `firstChild()`, `nextSibling()`, `firstChildElement()`, and `nextSiblingElement()`.
- Test the type of nodes using the `is*` methods, e.g. `isElement()`, `isText()`, `isComment()`.
- Convert a node to an element with `toElement()`.
- See example *qdom*.

Writing XML with DOM

- Create a node with `QDomDocument::createElement()`, `QDomDocument::createAttribute()`, `QDomDocument::createTextNode()`.
- Insert nodes into tree with `QDomNode::insertBefore()`, `insertAfter()`, `appendChild()`.
- Retrieve a textual representation of the whole tree with `QDomDocument::toString()`.
- See example *qdom-write*.

Project Task

- Extend the provided program (testtool) so that it is capable of displaying XML as shown on the next page.
- Optionally extend the program so that it can save the content as well.
- Optionally detect/fix shortcomings in the provided program.

XML example

```
<test-case name="resize-test">
  <objective>
     Verify that the application window can resize.
  </objective>
  <input>
     Using the window manager handles, resize the window
  </input>
  <output>
     The content of the window should use all space.
  </output>
</test-case>
...
```


Introduction

- Qt is made for multithreading from the ground up.
- Before you venture into multithreading, make sure you understand the concepts of synchronization and deadlocks.
- If all you want to do is to move a long-running computation into a separate thread and then pick up the results of that computation after the thread is finished, you can get away without a real understanding.

Introduction cont'd

- Two terms you will find in the documentation:
  - A *reentrant* function can be called simultaneously by multiple threads provided that each invocation of the function references unique data.
  - A *thread-safe* function can be called simultaneously by multiple threads when each invocation references shared data. All access to the shared data is serialized.
- A class is reentrant or thread-safe if all its non-static methods are reentrant or thread-safe.

Starting Threads And Waiting For Them To End

- Java model: To create a new thread, subclass from `QThread` and reimplement `QThread::run()`.
- Then call `start()` on your instance.
- Threads have priorities that you can specify as an optional parameter to `start()`, or change with `setPriority()`.
- To wait for a thread to finish, use `QThread::wait()`.
- You can also specify a maximum number of milliseconds to wait.

Starting Threads And Waiting For Them To End cont'd

- `QThread::isFinished()` and `QThread::isRunning()` provide information about the execution of the thread; you can also connect to the signals `started()`, `finished()`, and `terminated()`
- A thread can stop execution temporarily with `QThread::sleep()`, `QThread::msleep()`, `QThread::usleep()`.
- Threads can have per-thread event loops, started with `QThread::exec()` and `QThread::exit()`. This is useful for network access (for which there are also convenience functions `waitFor*()`), and for cross-thread signal/slot communication.

QObject Thread Affinity and Event Processing

- Every `QObject` (subclass) instance holds a reference to the thread in which it was created (the owning thread, `QObject::thread()`)
- Event handling for the instance is *always* performed in the context of the owning thread.
- You can move `QObject` instances from one thread to another using `QObject::moveToThread.()`

Mutexes

- Mutexes are implemented by the class `QMutex`.
- The two important methods are `QMutex::lock()` and `QMutex::unlock()`.
- You can try locking a mutex using `QMutex::tryLock()`. If the lock was obtained it will return true, otherwise it will return false right away, rather than waiting for the mutex.

Mutexes cont'd.

- When you lock a mutex you must, of course, unlock it again!
- This can be troublesome if you want to lock a mutex at the entrance of a function, and unlock it at exit—your function may return from many places (code like "`if (...) return false;`"). If you are using exceptions (or libraries that do), every statement can be an exit point from your function!
- `QMutexLocker` will help you here, simply put the following code right before you need the lock, and it will lock the mutex for the duration of the block:

```
QMutexLocker lock(&myMutex);
```


Semaphores

- Semaphores (`QSemaphore`) are generalized mutexes.
- While a mutex can only be locked once at a time, a semaphore can be locked *n* times before it will refuse further locks. This is useful when protecting many identical resources.
- The maximum number of locks ("resources") is specified in the constructor.
- You lock a semaphore with `QSemaphore::acquire()`, specifying the number of resources (locks) you want. If not enough resources are available, the call will block until all requested resources can be locked.

Semaphores cont'd

- `release()` frees the specified number of resources.
- `tryAcquire()` tries to get the specified number of resources; returns `false` if it does not succeed.
- `available()` returns the number of available resources (but remember that this can change at any time, so the result is not very useful).

Mutexes vs. Semaphores

	Mutex	Semaphore
Terminology	Locks	Resources
At any time	one	many
Locking	`lock()`	`acquire()`
Unlocking	`unlock()`	`release()`
Testing	`tryLock()`	`tryAcquire()`

Read/Write Locks

- A special case in thread synchronisation: Any number of threads can access a resource read-only, but only one thread at a time can access a resource for writing.
- Also, if one thread is writing, no thread may read.
- Example: A network thread ("writer") fetches data from a network resource into a data pool. Several processor threads ("readers") process the data without changing it (e.g., convert into PDF or HTML, format for on-screen display, etc.).

Read/Write Locks

- Qt class for this scenario: `QReadWriteLock`, with methods `lockForRead()` and `lockForWrite()`, as well as `unlock()`.
- `tryLockForRead()` and `tryLockForWrite()` exist as well.
- `QReadLocker` and `QWriteLocker` are resource acquisition classes much like `QMutexLocker`.

Waiting for events

- `QWaitCondition::wait()` lets a thread wait for a certain event.
- You can specify a maximum waiting time.
- You must pass a locked `QMutex` (no `QReadWriteLock`, though), to atomically go from locked state to wait state. The mutex will be automatically locked before the thread is woken.
- Wake one (random) thread waiting on a wait condition with `QWaitCondition::wakeOne()` and all waiting threads with `QWaitCondition::wakeAll()`.

Thread local data

- General rule: static variables are always global, local ("automatic") variables are always thread-local.
- Sometimes, you may need to have data that is local to a thread, but you can't make it function-local. For this purpose, you can use `QThreadStorage`
- `QThreadStorage` is a template class taking the data type to store as it's template argument, e.g. `QThreadStorage<int*>`.

Thread-local data cont'd

- Use `setLocalData()` to store data, `localData()` to fetch data, and `hasLocalData()` to test if any data has been set.
- Due to compiler limitations, `QThreadStorage` can currently only store pointers, and the data *must be created on the heap*, as `QThreadStorage` takes ownership of it and deletes it as necessary.
- **IMPORTANT:** see the Caveats section of the reference page.

Inter-Thread Communication

- Task: Pass data and notifications from one thread to another.
- `QWaitCondition` plus mutex-protected thread-global data goes a long way, but does not work when the main thread is involved, because the UI would freeze while the main thread is waiting for a condition.
- Two other solutions: Posting events with `QCoreApplication::postEvent()` and using cross-thread signal/slot communication.

Inter-Thread Communication cont'd

- Posting events:

```
QCoreApplication::postEvent( receiver,
                             new MyQEventSubclass( mydata ) );
```

- You can pick up the event in the other thread using an eventFilter, reimplementing `QCoreApplication::notify()` (only for the GUI thread!), or reimplementing `QObject::event()` in the receiving class. The former two are advanced techniques that are not recommended for beginners.
- This technique requires that the receiving thread (but not necessarily the sending one) has an event loop (the main thread has `QCoreApplication::exec()`, the others can get one with `QThread::exec()`).

Inter-Thread Communication cont'd

- Cross-thread signal/slots

```
connect( sender, SIGNAL( ... ), receiver, SLOT( ... ),
         Qt::QueuedConnection );
...
// emit the signal on the sender thread as usual
```

- Even this technique requires that the receiving thread (but not necessarily the sending one) has an event loop; see previous slide.
- If you do not specify the `Qt::QueuedConnection`, the default is `Qt::AutoConnection`, which will do the right thing if emitter and receiver are on different threads.

Inter-Thread Communication cont'd

- If you want to emit signals across threads that use custom (your own) data types, those data types need to be serializable.
- Declare the types with `Q_DECLARE_METATYPE`:

```
class MyType
{
...
};

Q_DECLARE_METATYPE( MyType )
```


Tips and tricks

- Do not attempt any GUI operations from another than the main thread.
- Never call GUI operations that circumvent Qt from another than the main thread (e.g. calling native X11 or Win32 functions).
- Never create anything inheriting from `QWidget`, or `QPixmap` from another than the main thread.
- When you create other objects from a secondary thread, make sure that you delete them before you delete the thread.
- Don't block in the main (GUI) thread, otherwise, the UI freezes.

Tips and tricks cont'd

- `Q(Core)Application::exec()`, `QDialog::exec()`, and `QMenu::exec()` *must* always be called from the main GUI thread.
- Don't call `QCoreApplication::sendEvent()` from another than the main GUI thread, but instead `QCoreApplication::postEvent()`.
- If a class or method is not explicitly marked as reentrant or thread-safe, it probably isn't. In particular, while copying e.g. QStrings between threads is safe, accessing the same QString object is not.

Tips and tricks cont'd

- If you need to copy larger amounts of data around and want to use implicit sharing, use `QSharedDataPointer` in order to do so in a thread-safe way.
- Unlock each `QMutex`, `QSemaphore`. and `QReadWriteLock` exactly the same amount of times you have locked it.

Tips and tricks cont'd

- In case Qt complains it doesn't know MyType, check that you have `Q_DECLARE_METATYPE(MyType)` in the classes header file.
- If that still doesn't solve the problem, you have to call `qRegisterMetaType<MyType>( "MyType" )` yourself.
- When emitting a signal, always be aware that it can be connected queued or directly, and that queued slots are executed a long time after the emit has returned. Adjust lifetimes of objects passed to emit accordingly.
- See the examples *fibonacci/events* and *fibonacci/signalslot*

Overview

- The Commercial Desktop Edition of Qt contains cross platform and database independent SQL functions. Supported drivers are:
  - IBM DB2, v7.1 and higher
  - Borland Interbase Driver
  - MySQL Driver
  - Oracle Call Interface Driver
  - ODBC Driver (includes Microsoft SQL Server)
  - PostgreSQL v6.x and v7.x Driver
  - SQLite version 2, 3 or above
  - Sybase Adaptive Server

Preliminaries

- Extra drivers can be developed as Qt plug-ins.
- In order to use the SQL support, add `QT += sql` to your QMake files.

Overview cont'd.

- In the following, we will describe the SQL functions, starting with the most basic ones, ending with specialized widgets:
  - Connecting to a database.
  - Invoking SQL queries using `QSqlQuery`.
  - Model/View classes.
- We will not cover how to develop a new database driver.

Connecting to a Database

- Choose driver(s) to use in the static method `QSqlDatabase::addDatabase(QString)`.

Driver Type	Description
QDB2	IBM DB2, v7.1 and higher
QIBASE	Borland Interbase Driver
QMYSQL	MySQL Driver
QOCI	Oracle Call Interface Driver
QODBC	ODBC Driver (includes Microsoft SQL Server)
QPSQL	PostgreSQL v6.x and v7.x Driver
QSQLITE	SQLite version 3 or above
QSQLITE2	SQLite version 2
QTDS	Sybase Adaptive Server

Connecting to a Database cont'd

- `addDatabase()` returns a database object; you will use this later when accessing the database.
- An alternative is to pass a second argument to `addDatabase()`, which is a `QString` naming the database. You can then get the database object using the static method `QSqlDatabase::database()`.

Connecting to a Database cont'd

- If you do not specify a name for one of the databases then this is the default database which is implicitly used in SQL queries later on.
- So far, we have only specified the driver to use. Next you must specify the database, host, user, and password using: `setDatabaseName()`, `setHostName()`, `setUserName()`, and `setPassword()`.
- Finally, it's time to connect to the database, which is done using `open()`.
- `open()` returns true if a connection could be established, and false if something went wrong.

Connecting to a Database cont'd

- Error messages and error codes can be obtained from the object returned by the method `QSqlDatabase::lastError()`.
- The error object contains, among others, the following methods: `driverText()`, `databaseText()`, `text()`, `type()` (driver error number), and `number()` (database error number).
- `text()` is a concatenation of `databaseText()` and `driverText()`
- Note that the text returned from `databaseText()` is most likely not localized.

Connecting to a Database cont'd

- Example:

```
QSqlDatabase db = QSqlDatabase::addDatabase("QMYSQL");
db.setHostName("bigblue");
db.setDatabaseName("flightdb");
db.setUserName("acarlson");
db.setPassword("1uTbSbAs");
bool ok = db.open();
if ( !ok )
  qFatal() << "Error opening database: "
           << db.lastError();
```


Invoking SQL Queries

- The class `QSqlQuery` can be used to run any SQL query. This includes select statements, insert statement, and even statements for creating new tables.
- To run a query, create an instance of `QSqlQuery`, and run the `exec()` method specifying the statement as a parameter:

```
QSqlQuery query;
if ( !query.exec( "SELECT name FROM author" ) )
  ...
```

- An optional argument can be given to the constructor specifying which database to use.

Invoking SQL Queries cont'd.

- `QSqlQuery::lastError()` can be used to query for error messages.
- In case of a select statement, the result can be iterated over using `QSqlQuery::next()`. This method returns true as long as there are more records.
- The value of the records are fetched using `QSqlQuery::value(int)`, which returns a `QVariant`.
- `QSqlQuery::first()`, `QSqlQuery::last()`, `QSqlQuery::prev()`, and `QSqlQuery::seek(int)` can be used for navigation.

Invoking SQL Queries cont'd.

- `QSqlQuery::size()` reports how many rows were matched by a select query . It returns -1 if the number of rows can not be determined.
- `QSqlQuery::numRowsAffected()` tells how many rows was affected by a non select query, say an update query.
- See example *sql/qsql-query*.

Prepared Queries

- `QSqlQuery` can be used for any kind of queries including inserting.
- If you are going to insert a large number of records after one another, using prepared queries can speed things up.
- If the database does not supports prepared queries Qt will translate the query into an ordinary query.
- Two kinds of prepared queries exists: *named bindings* and *positional bindings*

Prepared Queries cont'd.

- Named bindings:

```
QSqlQuery query;
query.prepare("INSERT INTO employee (id, name, salary)
              "VALUES (:id, :name, :salary)");
query.bindValue(":id", 1001);
query.bindValue(":name", "Thad Beaumont");
query.bindValue(":salary", 65000);
query.exec();
```


Prepared Query cont'd.

- Positional bindings:

```
QSqlQuery query;
query.prepare("INSERT INTO employee (id, name, salary)
              "VALUES (?, ?, ?)");
query.addBindValue(1001);
query.addBindValue("Thad Beaumont");
query.addBindValue(65000);
query.exec();
```

Prepared Queries cont'd.

- An advantage with prepared queries is that you do not need to worry about escaping.
- In the above examples, you do of course only need to execute `prepare()` once.

QSqlQueryModel

- The class `QSqlQueryModel` basically wraps a `QSqlQuery` in a `QAbstractItemModel`, so that the result set of the query can be used with the model/view frame work.
- The titles displayed in views are the column names from the database, this can be changed using `QSqlQueryModel::setHeaderData()`
- See the example *model-view/qsql-query-model*.

QSqlTableModel

- The class `QSqlTableModel` wraps a single table in a model, and does therefore allow editing the items.
- Normal usage:
  - Create an instance of `QSqlTableModel`, an call `setTable()` specifying the table to use.
  - Optionally call `setFilter()` specifying a *WHERE* part of a SQL query
  - Optionally call `setSort()` specifying column number and sort direction.
  - Call `select()` to execute the query.

QSqlTableModel cont'd.

- It is of course possible to access the table programmatically using the methods of `QAbstractItemModel`, but the `QSqlTableModel` add a few methods for convenience.
- `record()`, `setRecord()` and `insertRecord()` all works with instances of `QSqlRecord`, and refers to rows in the table rather than `QModelIndexes`.
- `QSqlRecord` is a simple container for records containing methods like `setValue(int index, QVariant value)`, `setValue( QString name, QVariant value )`, and similar `QVariant value(...)` methods.

QSqlTableModel cont'd.

- Example updating a table using `QSqlTableModel`:

```
for ( int row = 0; row < model->rowCount(); ++row ) {
    QSqlRecord record = model->record( row );
    double price = record.value( "price" ).toDouble();
    price *= 1.1;
    record.setValue( "price", price );
    model->setRecord( row, record );
}
model->submitAll();
```


QSqlTableModel cont'd.

- Using `setEditStrategy()` you specify when changes made in the GUI should be committed to the database. You have the following possibilities:
  - **OnFieldChange** Data will be saved as soon as you start editing a new cell.
  - **OnRowChange** Data will be saved when you start editing a new record. (changes can be discarded by calling `revert`)
  - **OnManualSubmit** Data will only be saved when you call `submit()`. (Changes can be discarded with `revertAll()`)

QSqlTableModel cont'd.

- Be careful with `OnFieldChange`:
  - performance can drop significantly compared to using the other editing strategies.
  - If you modify a primary key, the record might slip through your fingers while you are trying to fill it.

QSqlRelationalTableModel

- The class `QSqlRelationalTableModel` is similar to `QSqlTableModel`, with the addition to automatically replace a field with a field from a foreign relational table.
- Using the method `setRelation()` you specify a column in the current view to be replaced with a field in a foreign table
- See example *sql/qsql-relational-table-model*

Editable Queries Using QSqlQueryModel

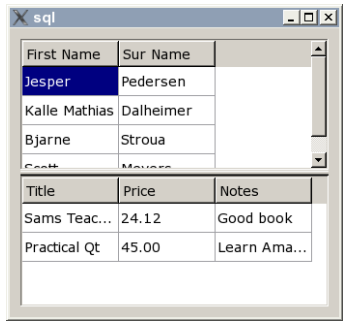
- Without modifications `QSqlQueryModel` is read only, while `QSqlTableModel` only works on a single table.
- To be able to edit the result of an arbitrary query, override `QAbstractItemModel::setData()` to update the date yourself, and `QAbstractItemModel::flags()` to specify that the table is editable.
- See the example *sql/editable-query*

Transactions

- You start a transaction using `QSqlDatabase::transaction()`, and ends it using `QSqlDatabase::commit()` or `QSqlDatabase::rollback()`.
- The above methods returns true if the action succeeded.
- Transaction requires support from the database, which you can check if it has using `QSqlDriver::hasFeature(QSqlDriver::Transactions)`

Project Task

- Implement an application that shows the author table in the top part, and shows the book table with only the books from the current author in the lower part.
- Follow these steps:
  1. Set up the author table, connecting a signal telling you about current item change to a slot of your choice.
  2. Set up the book table using a `QSqlQueryModel`, rerun the query in the above slot.



Project task

- Steps:
  3. Make it possible to edit the content of both tables
  4. Optionally, polish the views not to show the Id coulmns, nor the row headers.
  5. Optionally, make it possible to add and delete rows using a context menu.

Conceptual View

- A plugin is a dynamically loadable module (`.dll` on Microsoft Windows, `.so` on UNIX, and `.dylib` on Macintosh).
- A host application loads the plugins into its address space, and can control when they are supposed to do their job.

Overview

- It is possible to extend Qt in several ways using plugins, these include plugins for the image loader, styles, new SQL drivers, and plugins for Qt designer.
- It is also possible to develop a plugin mechanism for your own application using building blocks from Qt.

Extending Qt with Plugins

- Developing a plugin for Qt consists of three steps:
  1. Inherit an abstract base class and implement it.
  2. Add a factory macro to your implementation file.
  3. Set up an appropriate QMake file.

Extending Qt with Plugins - Step 1

- The following base classes exist: `QAccessibleBridgePlugin`, `QAccessiblePlugin`, `QDecorationPlugin`, `QGfxDriverPlugin`, `QIconEnginePlugin`, `QImageIOPlugin`, `QInputContextPlugin`, `QKbdDriverPlugin`, `QMouseDriverPlugin`, `QPictureFormatPlugin`, `QSqlDriverPlugin`, `QStylePlugin`, `QTextCodecPlugin`.
- To implement e.g. a new style, you must inherit `QStylePlugin`, and implement the two pure virtual methods `QStylePlugin::keys()` and `QStylePlugin::create()`.

Extending Qt with Plugins - Step 2

- In your implementation file, you must add the macro `Q_EXPORT_PLUGIN`, passing the class name as a parameter.
- This allows Qt to load the macro from the dynamic library, into which it will be compiled.
- Example: `Q_EXPORT_PLUGIN(MyStylePlugin)`

Extending Qt with Plugins - Step 3

- The QMake project file should contain the following two lines:

```
TEMPLATE = lib
CONFIG += plugin
```

- The QMake project file should place the dynamic library in a special directory, as specified on the next page.
- Do so using one of QMake's `DESTDIR` or `INSTALL` variables.

Searching for Plugins

- Qt will search for plugins in several different directory trees:
  - The directory of the application executable. (As returned from `QCoreApplication::applicationDirPath()`).
  - The directory returned by `QLibraryInfo::location(QLibraryInfo::PluginsPath)` - usually `plugins`.
  - Directories added by `QCoreApplication::addLibraryPath()` or set by `QCoreApplication::setLibraryPaths()`
- Each plugin type has its own subdirectory, style plugins are e.g. located in `styles` subdirectory: `plugins/styles/`

The Build Key

- To avoid that Qt loads a plugin not compatible with the application, a build key is compiled into the plugin.
- The following aspects are compared between the current Qt version and the plugin:
  - The Qt version.
  - The architecture, operating system, and compiler.
  - The compilation flags for Qt.
  - An optional string that can be specified when running Qt's configure script (`--buildkey` option).

Extending Your Own Applications Using Plugins

- Using the building blocks from Qt, it is easy to develop your own plugin mechanism, which allows your application to be extended using plugins.
- There are two sides to plugins:
  - Making the application ready for plugins.
  - Implementing actual plugins.

Offering and Loading Plugins

- Steps to making the application extensible with plugins:
  1. Define a set of interfaces (classes with pure virtual functions).
  2. Use the `Q_DECLARE_INTERFACE()` macro to tell Qt about the interface.
  3. Use `QPluginLoader` in the application to load the plugin.
  4. Use `qobject_cast()` to test whether a plugin implements a certain interface. (see page 745).

Interface Example

```
class FilterInterface
{
public:
    virtual QStringList filters() const = 0;
    virtual QImage filterImage(const QString &filter,
                               const QImage &image,
                               QWidget *parent) = 0;
};

Q_DECLARE_INTERFACE(FilterInterface,
    "com.trolltech.PlugAndPaint.FilterInterface/1.0")
```


Offering and Loading Plugins cont'd.

- The interface may only contain pure virtual classes, otherwise we would have to deal with exporting symbols to get it working on Windows and newer GCC's.
- The first argument to the `Q_DECLARE_INTERFACE()` is the class name implementing the interface.
- The second argument must be a unique string identifying the interface.
- The interface may not inherit `QObject`, as this, among other things, would make the class non-pure virtual.

Loading Plugins Example

```
foreach (QString fileName, pluginsDir.entryList(QDir::Files)) {
    QPluginLoader loader(pluginsDir.absoluteFilePath(fileName));
    QObject* plugin = loader.instance();
    if (plugin) {
        FilterInterface* iFilter =
              qobject_cast<FilterInterface*>(plugin);
        if (iFilter)
            ...
    }
}
```


Implementing Actual Plugins

- Steps to implement an actual plugin:
  1. Create a class inheriting from QObject and the required interface.
  2. Use `Q_INTERFACES()` to tell Qt about the implemented interface.
  3. Export the plugin using `Q_EXPORT_PLUGIN`
  4. Build the plugin using a suitable `.pro` file.

Example Plugin

```
class ExtraFiltersPlugin : public QObject,
                           public FilterInterface
{
    Q_OBJECT
    Q_INTERFACES(FilterInterface)

public:
    QStringList filters() const;
    QImage filterImage(const QString &filter,
                       const QImage &image,
                       QWidget *parent);
};
```


Implementing Actual Plugins cont'd.

- In case the plugin implemented several interfaces, simply list them as a space-separated list using `Q_INTERFACES`.
- The interface class may not inherit `QObject`, but the actual plugin must.
- In one of the implementation files for the plugin, the following code must be placed: `Q_EXPORT_PLUGIN(ExtraFiltersPlugin)`.

Signals/Slots

- Interfaces may not inherit `QObject`, so you can expect that it is not possible to use signal/slots with interfaces.
- The introspection of QObject works, however, with dynamic types rather than static types, so even if you only have a pointer to a QObject, you can actually connect to signals of the real type of the object.

Signals/Slots cont'd.

- ```
  class FilterInterface
  {
  public:
      ...
      // slots:
      virtual void someSlot() = 0;
      // signals:
      // void someSignal();
  };
  ```
- The above is only an informal contract that the users of the interface must fulfill.

Casting using qobject_cast et. al.

- C++ has four casting operators static_cast, dynamic_cast, const_cast, reinterpret_cast.
- dynamic_cast can check if what is being casted to actually is of the given type:

  ```
  X* a = dynamic_cast<X*>( b );
  if ( a )
    ...
  ```
- `dynamic_cast` does, however, not work across dynamic library boundaries on all compilers/platforms.

Casting using qobject_cast et. al. cont'd.

- Qt comes with qobject_cast, qvariant_cast, and qstyleoption_cast.
- These cast methods use internals from the respective classes to do the cast, and thus solve the dynamic library boundary problems.
- Example:

  ```
  QPushButton* button = qobject_cast<QPushButton *>(obj);
  if ( button )
    ...
  ```
- **Note:** the casting functions only work within their respective classes, thus you can only use `qobject_cast` with `QObject` subclasses.
  ```

Project Task: Add Plugins to the Paint Program

- In the handout directory *handout/paintProgramForPlugins*, you can find a slightly modified version of the, by now well-known, paint program.
- Add plugin capabilities to the application and implement circle drawing in the plugin.
- Optional: Develop other plugins, e.g. a house drawing plugin.

GDB – GNU Debugger

- Introduction
  - GDB is a free debugger for Linux and UNIX
  - GDB is a command-line tool, but GUI frontends exist (e.g., DDD and kdbg)
- Requirements
  - Build Qt and application with debug information.
  - Qmake projects:

    `CONFIG+=debug`
  - Rebuild project:

    `make clean && make && make install`

GDB – GNU Debugger

- Running the application in gdb:

```
$ gdb myapp
(gdb) run <command-line arguments>
```

- Stopping and continuing

```
# Press CTRL-c to stop the application
Program received signal SIGINT, Interrupt.
...
(gdb) cont # continue execution where we stopped
```


GDB – GNU Debugger

- Setting breakpoints: If the application is running, press CTRL-C to stop it. Then set a breakpoint:

```
(gdb) list main.cpp:10 # show contents of main.cpp
  around line 10
(gdb) break 17 # set breakpoint on line 17
(gdb) break 'QString::length() const' # set
  breakpoint at the start of some method
(gdb) cont # continue running
  (use run if not started yet)
```

GDB – GNU Debugger

- Debug crashes. If the application crashes while running in gdb, do this

```
Program received signal SIGSEGV, Segmentation fault.
...
(gdb) backtrace # will print out a trace of function
  calls leading to the crash
```

- Alternatively, if the application was run outside gdb and crashed, it produces a core dump which can be examined post mortem with gdb:

```
$ gdb myapp core
(gdb) backtrace
```


Strace

- strace – trace system calls and signals. Simplest usage:

```
$ strace myapp <command-line-argument>
```

strace will now print out a line for each system call the application performs

- Often only a few system calls are of interest, so to produce more manageable output, use e.g.

```
$ strace -e trace=open myapp <command-line-argument>
# only report calls to the open() system call
```

- Useful shortcuts for collections of system calls: `-e trace=file`, `-e trace=network`, `-e trace=signal`.

Valgrind

- Introduction:
  - Valgrind is a free high-quality debugger/profiler tool for Linux/x86 (and only that!)
  - Many "skins" available: `memcheck`, `addrcheck`, `cachegrind`, `massif` and `helgrind`
  - We will examine the `memcheck` "skin" of valgrind here (addrcheck is faster, but slightly less featureful)

Valgrind cont'd

- Valgrind is easy to use - no recompilation, no relinking, and no modification of source code.
- Valgrinds output is accurate and easy to understand.
- Valgrind is not a toy (some products valgrind has been executed on include most of KDE and Gnome, Mozilla, OpenOffice, and MySQL.)

Valgrind cont'd

- Simplest way to run myapp in valgrind:
  `valgrind --skin=memcheck myapp` <*command-line-argument*> This will print out information about access to unallocated memory, mismatching new/delete etc. on the console.

Valgrind – an example

- Consider the following piece of code

```
const char* str = "Hello World!";
char* copy = (char*)new char[strlen(str)];
strcpy(copy,str);
delete copy;
```

Do you see the problems? The array allocated for the copy of the string is one byte too short to contain the string plus the trailing null. This code may or may not crash when writing one byte past the end of the char array. In some cases, it does not crash, but the code is clearly wrong anyway. The other error is that memory allocated with `new[]` is deallocated with `delete` and not `delete[]`, as it should have been!

Valgrind – an example

- Valgrinding the code on the previous page yields:

```
==12294== Invalid write of size 1
==12294==    at 0x4002178E: strcpy (mac_replace_strmem.c:174)
==12294==    by 0x8048672: main (main.cpp:5)
...
==12294== Mismatched free() / delete / delete []
==12294==    at 0x40028ED7: __builtin_delete (vg_replace_malloc.c:244)
==12294==    by 0x40028EF5: operator delete(void*) (vg_replace_malloc.c:253)
==12294==    by 0x8048680: main (main.cpp:6)
...
==12294==    Address 0x43E9C43C is 0 bytes inside a block of size 12 alloc'd
==12294==    at 0x40028D70: __builtin_vec_new (vg_replace_malloc.c:203)
==12294==    by 0x40028DC7: operator new[](unsigned) (vg_replace_malloc.c:216)
==12294==    by 0x804865E: main (main.cpp:4)
```

So valgrind finds both of our problems. "Invalid write of size 1" means that we write one byte into unallocated memory (the trailing null of the string), "Mismatched free()/delete/delete[]" means that we are freeing memory with a function or operator that does not match the one that was used to allocate it—and it even tells us where that particular piece of memory was allocated and which operator was used!

Valgrind

- So this is very useful, but where is the catch?
  - Applications run 5-100 times slower when run in valgrind.
  - Increased memory usage, too.
  - Tied to one OS and one CPU architecture
  - No support for "fancy" CPU instructions (like 3DNow!) or non-POSIX signals.

Valgrind – Memcheck in Detail

What memcheck can do:

- Use of uninitialized memory
- Reading/writing memory after it has been free'd
- Reading/writing past the end of malloc'd blocks
- Reading/writing inappropriate areas on the stack
- Memory leaks—pointers to malloc'd blocks that are lost forever
- Passing of uninitialized and/or unaddressable memory to system calls
- Mismatched use of malloc/new/new [] vs free/delete/delete []
- Some misuses of the POSIX pthreads API

For details of error messages see the user manual available from http://valgrind.kde.org

Valgrind – Suppressing Errors

- Valgrind does not only check your application, but also the libraries you use, which include glibc, X11, Qt, KDE, ...
- Glibc and X11 have an alarming amount of errors, so it is useful to be able to filter them out.
- Valgrind is capable of suppressing error reports, and is shipped with a number of suppression rules for standard Linux libraries.
- You can also develop your own suppression rules, simply put them in a file, and start valgrind with the `--suppressions=<filename>`
- Run valgrind using `--gen-suppressions=yes`, to make it create suppression rules.

Running GDB from Valgrind

- When valgrind encounters an error, you have the option to start GDB to analyze your program.
- Simply start valgrind with the option

  `--gdb-attach=yes`
- When valgrind finds an error it will ask:

```
Attach to GDB ? --- [Return/N/n/Y/y/C/c]
Return/n/N - no
y/Y - yes
c/C - no, and do not ask again.
```

- *Do not(!)* continue execution from within GDB but instead call detach and then quit.

Valgrind – Querying Valgrind from C++

- Its possible to add certain macros to your C++ code that valgrind understands.
- In order to use these, you must include `memcheck.h`
- These macros are no-ops when the application is not executing under valgrind.
- A number of different macros are available, see the documentation for the full list.
- `VALGRIND_CHECK_DEFINED` is the most useful one - it checks whether a variable is defined (see the next slide on why this is useful).

Valgrind – Understanding the Output of Memcheck

- Memcheck only checks the *definedness* of variables in the following cases:
- A variable is used to generate a memory address (dereferencing a pointer).
- A control flow decision is made based on the value of the variable.
- The variable is passed to a system call.
- Therefore, valgrind will not complain about uninitialized variables that are just copied around.
- It will, of course, complain if an uninitialized variable is copied to another, and the other one is then used in one of the three situations above.

Valgrind – Understanding the Output of Memcheck

- ```
  struct S {
   int x;
      char v;
  };
  struct S s1, s2;
  s1.x = 42;
  s1.v = 'z';
  s2 = s1;
  ```
- The struct really takes up just 5 bytes, but most compilers would pad that to two words (8 bytes), and the copy thus copies 3 uninitialized bytes.

Valgrind – Understanding Memcheck

- Valgrind hooks into malloc/free (new/delete), and keeps track of each allocated byte (each allocated bit has a so-called "A" bit set). Additionally it stores information about where in the code the allocation was performed etc. Every access to memory is checked against the "A" bits to detect out-of-bounds read/writes on the heap.
- This is, however, not possible on the stack where all memory is statically allocated.
- Therefore, valgrind cannot do out-of-bounds checks on the stack.

Valgrind – Use of Released Memory

- Valgrind can check if memory is used after it has been free'd or deleted.
- To do so, valgrind will keep the memory from not being reallocated shortly after it has been released.
- Normally it will use 1Mb of memory for this purpose. You can increase the amount using `--freelist-vol=<number-of-bytes>`
  ```

Valgrind – Detecting Memory Leaks

- The memcheck skin can also be used to detect memory leaks, simply use `--leak-check=yes`.
- By default, memcheck will only print out those leaks that are definitely lost. Add `--show-reachable=yes` to see memory that is still addressable.
- Memcheck will by default group together traces where the most recent function call are the same: `main()->f1()->g()->h()` and `main()->f2()->g()->h()`.
- In the above, the leak in `h()` will be reported together for both traces.
- Use `--leak-resolution=lowmed—high—`. (low=2 entries, med=4 entries, high=all entries)

Valgrind – Miscellaneous

- Compile your application using `-g` and `-fno-inline` for best backtraces.
- Don't use `-O2` and above as it may confuse memcheck. (Using `-O` should be OK, though).
- It is possible to ask valgrind to also trace child processes by adding `--trace-children=yes`.
- Valgrind reads options from the environment variable `VALGRIND_OPTS`.
- Get more context information using `--num-callers=50`
- A number of GUI's exists for valgrind, see `http://valgrind.kde.org/` -> Related Projects.

Profiling with (k)cachegrind

- Finding the central parts of your code where the application spends most of its CPU resources can be difficult—do not rely on your intuition, it will most likely fail you!
- Use profiling tools instead to identify the few percent of your code that are performance critical.
- `callgrind` is such a tool, and `kcachegrind` is an application to examine the results.
- `kcachegrind` is available from `http://kcachegrind.sourceforge.net/`

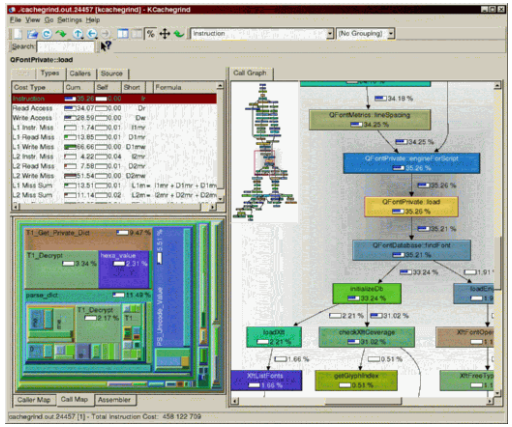
Recording profiling information

- To record information for profiling, simply execute valgrind like this: `callgrind <application-and-args>`
- Record realistically sized data!
- Be patient!
- `callgrind` has a lot of options, but most are very specialized, and rarely needed
- `callgrind` can be controlled using `callgrind_control`:

```
callgrind_control -z # zero's collection information
callgrind_control -d <str> # dumps information.
```

- Now start `kcachegrind`.

KCacheGrind



Examining profiling information

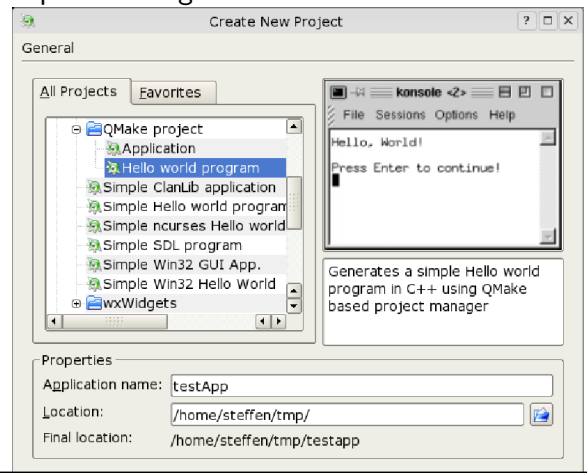
- When `kcachegrind` is started, it will look in the current directory for the file `callgrind.out.<PID>` that `callgrind` created.
- Browsing the calltree with `kcachegrind` can be useful to become familiar with existing code.
- Where is time spend? In the lower left list view, click the "Self" header to sort functions according to how much time was spend in them.
- Look at the "Called" column. If a function has not been called too many times but still a lot of time was spend in it, use the source view to find the lines that need optimization.

KDevelop – Introduction

- KDevelop
  - is an integrated development environment (IDE) for KDE.
  - supports many programming languages (currently 12 from Ada to Ruby) including C++.
  - supports several build systems, including GNU autoconf/automake and qmake
  - supports several version control systems, including CVS.
  - supports valgrind integration.
  - supports Qt designer integration.
- We will look at how to use KDevelop for pure Qt projects using QMake.

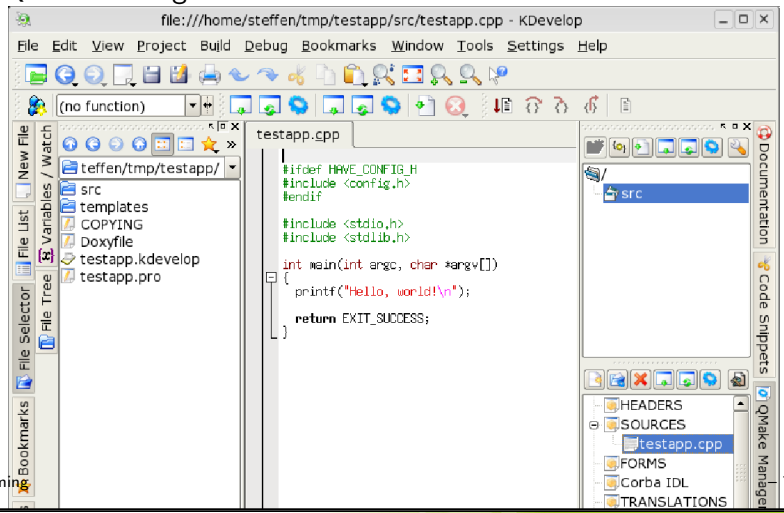
KDevelop – Creating a project

- To get started, go to the Projects menu and choose "New Project..."
- This will open a dialog like this:



KDevelop

- KDevelop main window showing the source editor, file list and QMake manager:



Basics

- Dual licensing: Qt, Qtopia Core, and Qtopia are available under both GPL and commercial licenses.
- The GPL License is appropriate for Open Source development.
- Commercial license: needed for development of non-GPL software,
  and when support is needed.
- Additional licenses: Evaluation, Educational (site), Academic, Product Review (suited for Journalists).

GPL license

- GPL license: offered to support true Open Source Development.
- Source code must always be available to users of binary.
- Anyone can re-use and re-distribute the source of your applications.
- No rights to compensation for reuse or redistribution.
- Resulting software must be licensed under GPL as well.

When are you allowed to use Qt under GPL

- When end product will be licensed under the GPL, and will remain so.
- You can't un-GPL your source, "take the GPL back".
- All code developed using GPL version of Qt must always remain under the GPL. Late commercialization is disallowed by Trolltech commercial license.
- GPL use is not allowed for pre-release closed source commercial. development

Commercial license

- Commercial license is designed for proprietary (closed source) development, commercial or otherwise..
- Support and upgrades are included.
- You can release your code under any license you want.

Who needs a commercial license?

- Programmers who use Qt for proprietary, closed source development need a license.
- Qt licenses are personal, per programmer, non-floating.
- Use definition: Editing source code containing Qt API calls.
- Use definition: Using Qt Designer.
- License not needed for compiling / linking code made by others in team.
- License not needed for using Assistant / Linguist.

Qtopia and Qtopia Core Licensing

- Developer licenses apply as for desktop versions (X11, Win, Mac).
- Runtime licenses apply for device manufacturers.
- No runtime license apply for creating Qtopia Core and Qtopia applications.

Static Linking

- Mechanics: Everything the application needs is pulled from the library into the binary.
- Advantage: No extra binary to ship.
- Advantage: No version conflicts.
- Disadvantage: Bugfixes in the library require recompilation, and *you* have to provide customers with updates.
- Disadvantage: Library code is not shared between different Qt applications.
- Disadvantage: Qt plugins (codecs, styles, image formats, etc.) do not work.

Dynamic Linking Against Private Library Copy

- Mechanics: Qt code is stored in a separate binary, which is combined with the application at run time to form the final executable.
- Mechanics: To avoid conflicts with other installed Qt libraries, use `-rpath`, or a wrapper script that sets `LD_LIBRARY_PATH` (Unix/Linux) or `PATH` (Windows) before calling the real application, e.g.:

```
#!/bin/sh
MYAPPDIR=/opt/myapp
LD_LIBRARY_PATH=$MYAPPDIR/lib:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH
exec $MYAPPDIR/bin/myapp
```


Dynamic Linking Against Private Library Copy

- Advantage: No version conflicts.
- Advantage: Bugfixes in the library do *not* require recompilation.
- Disadvantage: You still have to provide customers with updates.
- Disadvantage: Extra binary to ship.
- Disadvantage: Library code is not shared between different Qt applications.

Dynamic Linking Against System's Qt (Unix only)

- Mechanics: Same as before, but links against the system's Qt, not a private copy.
- Mechanics: When using the operating system's package manager, Qt will be automatically installed when your application is installed.
- Mechanics: When using a custom installer, you have to be careful to avoid erasing the installed Qt. Better use the previous method in this case.
- Advantage: No extra binary to ship.

Dynamic Linking Against System's Qt (Unix only)

- Advantage: Bugfixes in the library do *not* require recompilation.
- Advantage: Updates to Qt are shipped by the operating system vendor.
- Advantage: Library code is shared between different Qt applications (esp. useful when targeting the KDE desktop).
- Disadvantage: Choice of Qt version and configuration constrained by lowest common denominator of what the target distributors provide (Linux distributors are pretty uniform, though).
- Disadvantage: Some Unix vendors do not provide a system Qt.

Introduction

- Testing helps you gain confidence in your product, and makes paradigms like refactoring possible.
- Test-driven development is (part of) a whole development methodology based entirely on unit tests.
- At least two kind of test types exists: *unit testing* and *black box testing*.
- QTestLib is about unit testing, while products like KD Executor and Squish are more about black box testing.

Introduction cont'd.

- When writing unit tests, you write code that tests individual functions or methods.
- These tests are compiled into a test binary, which is compiled against the functions of your classes that you want to test.
- Since the tests are compiled into a separate binary, your final product will not contain any traces of test functions.
- As an added bonus, unit testing can well improve the design of your application.

QTestLib Facts

- **Lightweight** 6000 lines of code, 60 exported symbols.
- **Self-contained** Requires only a few symbols from Qt.
- **Rapid testing** No special test-runners; No special registration for tests.
- **Data-driven testing** A test can be executed multiple times with different test data.
- **Basic GUI testing** Offers functionality for mouse and keyboard simulation.
- **Type-safety** Extensive use of templates prevent errors introduced by implicit type casting.
- **Easily extendable** Custom types can easily be added to the test data and test output.

Writing a Simple Test

- Test cases are implemented as functions in a class which must inherit `QObject`.
- The macro `QTEST_MAIN` implements a main function and registers the test class at the same time.
- Add `CONFIG += qtestlib` to your qmake file to compile QTestLib into a test binary.
- A number of macros exist in order to verify your result, including `QVERIFY` and `QCOMPARE`
- See the example *unit-test/simple*

Data-Driven Testing

- When testing a certain function it might be useful to test it with different parameters.
- We could do this with a number of calls to e.g. QCOMPARE.
- The alternative is to use QTestLib's capability for data-driven testing.
- Here, one function is the actual test case, while another, with the postfix _data, contains the dataset. The test function is invoked for each data item in the data set.

Data Driven Testing cont'd.

- In the *_data* function, you set up your data using the method `QTest::addColumn()`, and create test data using `QTest::newRow()`:

```
QTest::addColumn<QString>( "input" );
QTest::addColumn<QString>( "result" );

QTest::newRow( "all lower" ) << "hello" << "hello";
QTest::newRow( "mixed" ) << "HellO" << "hello";
```

- In the test case, you fetch the data using the `QFETCH` macro:

```
QFETCH( QString, input );
QFETCH( QString, result );
QCOMPARE( input.toLower(), result );
```


Data Driven Testing cont'd.

- It is possible to use your own data types as long as they have been registered using `Q_DECLARE_METATYPE`.
- The macro `QEXPECT_FAIL` may be set up to expect failure from a given data in the set:

```
QFETCH( QString, data );
QFETCH( int, result );
QEXPECT_FAIL( "expected error",
              "To be fixed", Abort );
QCOMPARE( data.count(), result );
```

- The first parameter is the name of the data value, the second a message to display, and the third tells whether the test case should continue. after an error.
- See example *unit-test/data-driven*

GUI Testing

- Using QTestLib, you can simulate mouse and keyboard events.
- Keyboard events: `keyClick()`, `keyClicks()`, `keyEvent()`
- Mouse events: `mouseClick()`, `mouseDClick()`, `mousePress()`, `mouseRelease()`, `mouseMove()`
- Examples:

```
QTest::keyClicks( lineEdit, "Hello World" );
QTest::keyClick( lineEdit, Qt::Key_Backspace );
QTest::keyEvent( QTest::Press, lineEdit, 'a',
                 Qt::ControlModifier );
QTest::mouseClick( lineEdit, Qt::LeftButton );
```

GUI Testing cont'd.

- During testing, it can be useful to wait a few milliseconds, e.g. when waiting for a remote server to respond.
- This can be done using `QTest::qSleep()` or `QTest::qWait()`.
- `qSleep()` is blocking, i.e. no event processing is done, while `qWait()` keeps the event loop running while waiting.
- The methods `keyPress()`, `keyClick()`, ... all take an optional argument specifying the amount of milliseconds to wait before continuing.
- In addition, the command-line arguments `-keydelay`, `-mousedelay`, and `-eventdelay` can be used to insert a delay after each key and/or mouse event.

Data-Driven GUI Testing

- To support data driven GUI testing, the class `QTestEventList` can be used for storing key and mouse events.
- The class has methods similar to those of QTest, namely `addKeyPress()`, `addKeyClick()`, `addMouseClick()`, ...
- To playback the events, use `QTestEventList::simulate()`, which takes a widget on which the events are invoked.
- See the example *unit-test/gui-testing*.

QSignalSpy

- Unit testing is about stimulating an object, and verifying response.
- QObjects may emit signals as the response.
- The class `QSignalSpy` is designed to verify emitted signals.
- The class inherits `QList< QList<QVariant> >`, the outer list items each represent a signal emission, while the inner list items represent its argument.
- See example *unit-test/signals*

Project Task: Test the Compass Widget

- Create unit tests for the compass widgets you implemented, alternatively use the one from `solutions/compasswidget`.
- Discuss what you can do during development of your application, in order to better support unit testing.
- Optional: Modify the compass widget according to the suggestion above.

References - Home Pages

- Trolltech's home page: http://www.trolltech.com
- KDAB's home page: http://www.klaralvdalens-datakonsult.se
- Interface hall of fame/shame: http://homepage.mac.com/bradster/iarchitect/

References - Books

- The C++ programming Language, Bjarne Stroustrup, ISBN 9-780201-889543
- Effective C++, Scott Meyers, ISBN 9-780201-924886
- More Effective C++, Scott Meyers, ISBN: 9-780201-633719
- GUI Bloopers, Jeff Johnson, ISBN 1-55860-582-7
- Refactoring, Martin Fowler, ISBN 0-201-48567-2
- Head First, Design Patterns, ISBN 0-596-00712-4

Parent/Child-relationships (page 42)

1. This means that the widget will be a window, plus the answer to the next question
2. This means that it is not being owned by any other widgets, and you must therefore take care of deleting it when needed.
3. QMake is a makefile generator, that generates Makefiles on unix, and project files on windows. Unless you only develop applications using Microsoft Visual Studio, then it is a very good idea to use QMake. QMake makes your work easier when maintaining your project for multiple platforms.

Parent/Child-relationships cont'd.

4. Yes, if either it has been explicitly hidden at some point, or if you add it after its parent has been shown.
5. Anything allocated in the constructor needs to be deallocated in the destructor (unless the allocated instance is inheriting QObject).
6. See page 38

Signal/Slots (page 73)

1. Using QObject::connect()
2. See page 70 and 71
3. See page 70 and 71
4. Yes, slots are implemented as methods of a class, so you need a *host*-class.
5. Yes it can. This value will of course not make it to the sender of the signal, but may be used when calling the slot as a normal function.
6. Whenever you add `Q_OBJECT` to a class.
7. Anywhere in the class, but it is advised to place it at the top. It is needed whenever the class has a signal or a slot.

Paint Program (page 135)

1. Simplified, you could say: A signal is sent out without knowing the receiver, an event is sent by Qt to notify an object about system changes.
2. QPainter
3. In any function! You are, however, only allowed to paint to a widget in the `paintEvent()` method, but you may of course paint to e.g. a pixmap in any function.
4. QMainWindow, QMenuBar, QMenu, QToolBar, QStatusBar.
5. Create a QPainter on an instance of QPrinter, and paint onto it. Use QPrintDialog to configure the printer.
6. `QTextStream` produces readable text, while `QDataStream` encodes extra informations like length of string written to the stream.

Dialogs (page 169)

1. Technically, there is nothing wrong, but you should rather inherit the dialog, and set it up in the constructor.
2. A modal dialog is created when you want the user to complete the dialog before continuing to work with the rest of the application. By contrast, a non-modal dialog is used when you want the user to be capable of working both with the dialog and with the rest of the application at the same time.
3. Call `exec()` when you want to get an answer from the dialog before your code continues execution.

Dialogs (page 169)

4. You should use make the dialog into a class on its own, and create the checkbox instance in the constructor.
5. Rule of thumb use a non-modal dialog unless it really has to be modal.
6. Use exec() when you want the invocation of your code to halt until the dialog has been closed.
7. Yes, in that case the original dialog will be inaccessible until you close the new one. After closing the new dialog, the original will regain its modal status. You can, however, not `show()` a dialog when a modal dialog is active.
8. You only need to keep widgets as instance variables if you want to access them later, for example to enable/disable or read their content. You do especially not need to keep widgets for being able to delete them at destruction time, as the parent child relationship handles that.

Layout Managers (page 153)

1. Using `QWidget::setMinimumSize()`, or override `QWidget::minimumSizeHint()`.
2. `QHBoxLayout`, `QVBoxLayout`, `QGridLayout`.
3. For a widget, you specify stretch using the second argument to `QLayout::addWidget()`, for empty space you call `QLayout::addStretch()`
4. Either when the widget is a window, or when the parent does not have a layout manager of its own.

Advanced Drawing (page 607)

1. `QPainter` for the actual drawing, `QPen` for specifying the border of the rectangle, and `QBrush` for specifying the fill.
2. Use `QPainter::drawEllipse()`.
3. Classes inheriting QPaintDevice, which is `QWidget`, `QPixmap`, `QImage`, `QPicture`, `QPrinter`, and `QGLPixelBuffer`.
4. Translate, Scale, Sheer, Rotate.
5. When doing the same set of transformation often, or when needing a special setup, which can not be obtained using the four transformations.
6. Window transformations are for specifying the coordinate system, viewport transformations are for specifying what part of the widget to draw on.

Motif vs. Qt Widgets

Motif Widget	Qt Widget
VendorShell	(any top-level widget)
XmArrowButton	(QPushButton with arrow pixmap)
XmBulletinBoard	(QWidget)
XmCSText	QLabel
XmCascadeButton	(QPushButton)
XmClipWindow	(QWidget, setClipArea(...))
XmComboBox	QComboBox
XmCommand	(QAction)
XmContainer	(QWidget)
XmDialogShell	QDialog
XmDrawingArea	(QWidget)
XmDrawnButton	(QButton)
XmFileSelectionBox	QFileDialog
XmForm	(QWidget)
XmFrame	QFrame
XmGrabShell	(QWidget, grabMouse())
XmIconGadget	QLabel (with setPixmap())
XmLabel	QLabel
XmList	QListBox
XmMainWindow	(QMainWindow)
XmManager	(QWidget)

Motif vs. Qt Widgets cont'd.

Motif Widget	Qt Widget
XmMenuShell	(QMenu)
XmMessageBox	QMessageBox
XmNotebook	(QTabDialog)
XmPanedWindow	QVBoxWidget
XmPrimitive	QWidget
XmPrintShell	(QPrinter)
XmPushButton	QPushButton
XmRowColumn	QVBoxLayout, QHBoxLayout, QGridLayout
XmSash	QSplitter
XmScale	QSlider
XmScrollBar	QScrollBar (QScrollView)
XmScrolledWindow	QScrollView
XmSelectionBox	(QInputDialog)
XmSeparator	QFrame
XmSimpleSpinBox	QSpinBox
XmSpinBox	QSpinBox
XmText	QLineEdit
XmTextField	QTextEdit
XmToggleButton	QPushButton (with setToggleButton(true))

Motif Callbacks vs. Qt Signals/Slots

Motif Callback	Qt Signal
XmNactivateCallback	QButton::clicked(), QLineEdit::returnPressed(), QTextEdit::returnPressed()
XmNapplyCallback XmNarmCallback XmNbrowseSelectionCallback	(QPushButton::clicked()) QButton::pressed() QListBox::highlighted()
XmNcancelCallback XmNcascadingCallback XmNcommandChangedCallback	(QPushButton::clicked()) (QButton::clicked(),QMenuData::highlighted()) –
XmNcommandEnteredCallback XmNconvertCallback XmNdecrementCallback	QAction::triggered() – QScrollBar::prevLine()
XmNdefaultActionCallback XmNdestinationCallback XmNdisarmCallback	QListBox::selected() (drag-and-drop classes) QButton::released()
XmNdragCallback XmNexposeCallback	QScrollBar::sliderMoved(), QSlider::sliderMoved() QWidget::paintEvent() (QFrame::drawContents)
XmNextendedSelectionCallback XmNfocusCallback XmNgainPrimaryCallback	QListBox::selected() QWidget::focusInEvent() QClipboard::selectionChanged()
XmNhelpCallback XmNincrementCallback XmNinputCallback	(can use QAccel) QScrollBar::nextLine() QWidget::mousePressEvent, QWidget::keyPressEvent

Motif Callbacks vs. Qt Signals/Slots cont'd.

Motif Callback	Qt Signal
XmNlosePrimaryCallback XmNlosingFocusCallback XmNmapCallback	QClipboard::selectionChanged() QWidget::focusOutEvent() QWidget::showEvent(), QWidget::polish()
XmNmodifyVerifyCallback XmNmultipleSelectionCallback XmNnoMatchCallback	QLineEdit::textChanged(), QTextEdit::textChanged() QListBox::selected() –
XmNokCallback XmNpageChangedCallback XmNpageDecrementCallback	(QPushButton::clicked()) QTabWidget::currentChanged() QScrollBar::prevPage()
XmNpageIncrementCallback XmNpopdownCallback XmNpopupCallback	QScrollBar::nextPage() QWidget::closeEvent() (QWidget::show())
XmNpopupHandlerCallback XmNresizeCallback XmNselectionCallback	(QWidget::mousePressEvent or various rightClicked() signals in e.g. QListView) QWidget::resizeEvent() QComboBox::selected()
XmNsingleSelectionCallback XmNtoBottomCallback XmNtoTopCallback	QListBox::selected() – –
XmNunmapCallback XmNvalueChangedCallback XmNvalueChangedCallback	QWidget::hideEvent() QScrollBar::valueChanged(), QSlider::valueChanged(), QSpinBox::valueChanged() QWidget::toggled(bool), QLineEdit::textChanged(), QTextEdit::textChanged()

Concepts

Name Spaces, Classes, and Functions

Examples

Projects