

QT

Team Emertxe



Fundamentals of Qt Module

Fundamentals of Qt

- The Story of Qt
- Developing a Hello World Application
- Hello World using Qt Creator
- Practical Tips for Developers

Objectives

- About the history of Qt
- About Qt's ecosystem
- A high-level overview of Qt
- How to create first hello world program
- Build and run a program cross platform
- To use Qt Creator IDE
- Some practical tips for developing with Qt

Fundamentals of Qt

- The Story of Qt
- Developing a Hello World Application
- Hello World using Qt Creator
- Practical Tips for Developers

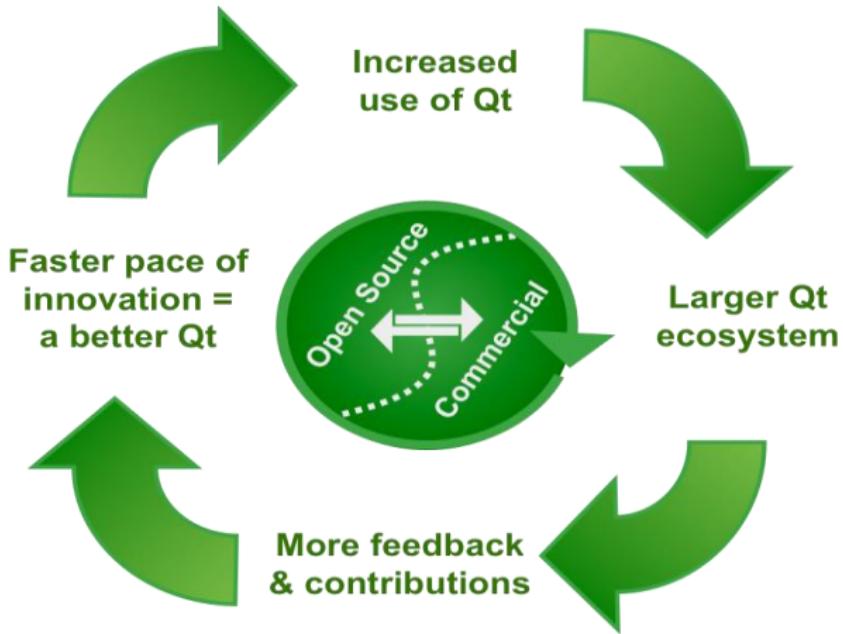
Some history

- **Qt Development Frameworks founded in 1994**
 - Trolltech acquired by Nokia in 2008
 - Qt Commercial business acquired by Digia in 2011
 - Qt business acquired by Digia from Nokia in 2012
 - Trusted by over 6,500 companies worldwide
- **Qt: a cross-platform application and UI framework**
 - For desktop, mobile and embedded development
 - Used by more than 350,000 commercial and open source developers
 - Backed by Qt consulting, support and training

Qt is everywhere



Qt virtuous cycle

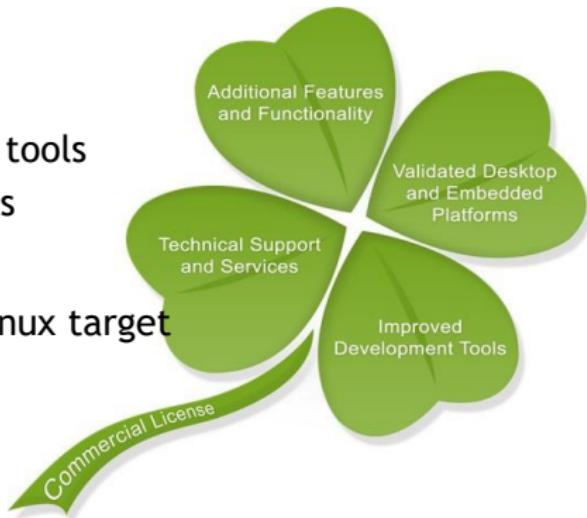


Why Qt

- Write code once to target multiple platforms
- Produce compact, high-performance applications
- Focus on innovation, not infrastructure coding
- Choose the license that fits you
 - Commercial, LGPL or GPL
- Count on professional services, support and training
- Take part in an active Qt ecosystem

Qt Commercial

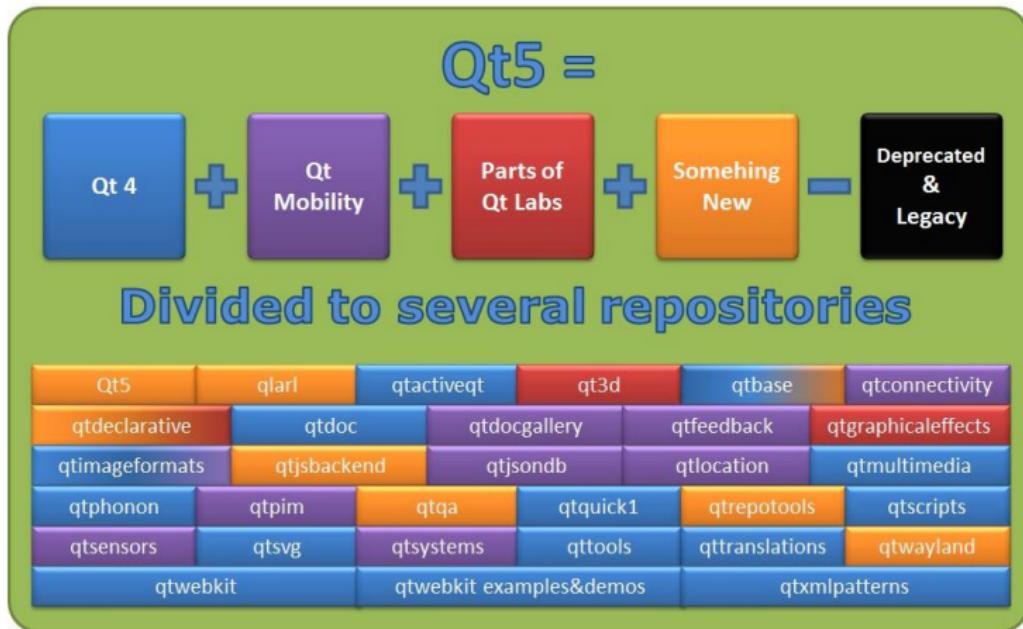
- Improved development tools for increased productivity and tangible cost savings
- Qt Commercial SDK
 - All Qt Commercial libraries and tools
 - Additional tools and components
- Qt Creator Embedded Target
 - Deploy directly to embedded Linux target
- RTOS toolchain integration
- Visual Studio Add-On



Qt5

- Awesome graphics capabilities
 - OpenGL as a standard feature of user interfaces
 - Shader-based graphics effects in QtQuick 2
- New modular structure
 - Qt Essential modules available on all platforms
 - Add-on modules provide additional or platform-specific functionality
- Developer productivity and flexibility
 - More web-like development with QtQuick 2, V8 JavaScript engine, and Qt JSON DB
- Cross-platform portability
 - Qt Platform Abstraction (QPA) replaces QWS and platform ports

Qt5 Modules



Qt Demo

- Let's have a look at the QtDemo Application
- Comes with every Qt installation

Technology	Demo
Painting	<i>Demonstrations/Path Stroking</i>
Widgets	<i>Demonstrations/Books</i>
Widgets	<i>Demonstrations/TextEdit</i>
Graphics View	<i>Demonstrations/40.000 Chips</i>
OpenGL	<i>Demonstrations/Boxes</i>
WebKit	<i>Demonstrations/Browser</i>



Fundamentals of Qt

- The Story of Qt
- **Developing a Hello World Application**
- Hello World using Qt Creator
- Practical Tips for Developers

“Hello world” in Qt

```
// main.cpp  
#include <QtWidgets>  
  
int main(int argc, char *argv[])  
{  
    QApplication app(argc, argv);  
    QPushButton button("Hello world");  
    button.show();  
    return app.exec();  
}
```



- Program consists of
 - main.cpp- application code
 - helloworld.pro- project file
- [Hello world demo](#)

Project File

helloworld.pro

- **helloworld.pro file**
 - lists source and header files
 - provides project configuration

```
# File: helloworld.pro
SOURCES = main.cpp

HEADERS += # No headers used
QT = core gui widgets
```

- Assignment to variables
 - Possible operators =, +=, -=
- [Qmake-tutorial](#)

Using qmake

- qmake tool
 - Creates cross-platform makefiles

- Build project using qmake

```
cd helloworld  
qmake helloworld.pro      # creates Makefile  
Make                      # compiles and links application  
.helloworld              # executes application
```

- Tip: qmake -project

- Creates default project file based on directory content

- [Qmake manual](#)

Qt Creator does it all for you

Fundamentals of Qt

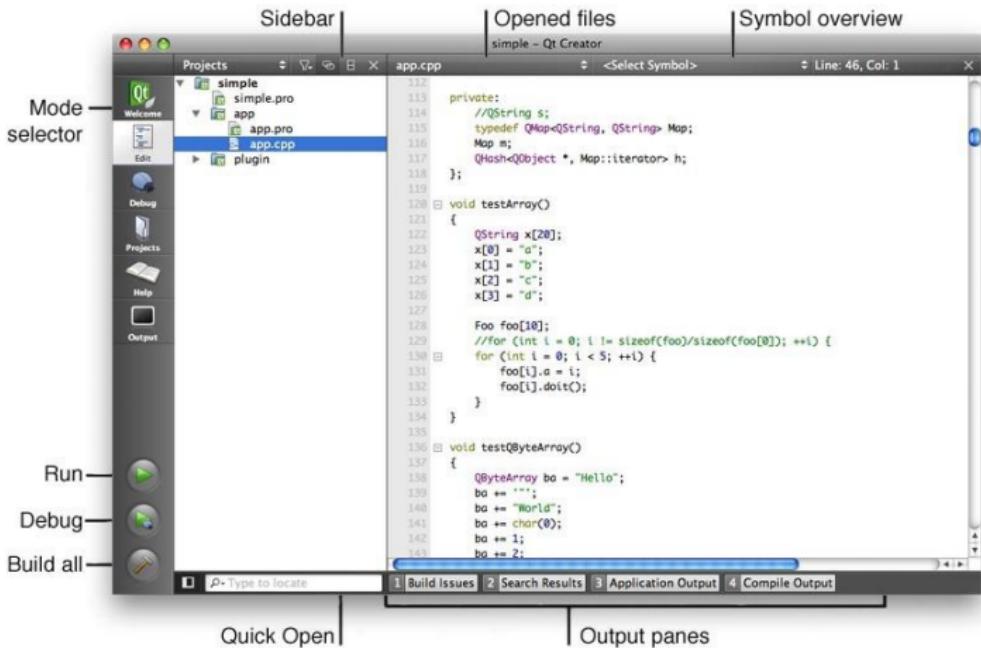
- The Story of Qt
- Developing a Hello World Application
- **Hello World using Qt Creator**
- Practical Tips for Developers

QtCreator IDE

- Advanced C++ code editor
- Integrated GUI layout and forms designer
- Project and build management tools
- Integrated, context-sensitive help system
- Visual debugger
- Rapid code navigation tools
- Supports multiple platforms



Qt Creator Components



Debugging

The screenshot shows the Qt Creator IDE interface during a debugging session. The main window displays the code for `main.cpp`:

```
#include <QtGui>
int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QPushButton *button = new QPushButton("Hello world");
    button->show();
    return app.exec();
}
```

The debugger status bar indicates "Stopped: 'signal-received'" with thread ID 1. The Locals and Watchers tab is selected, showing the state of variables:

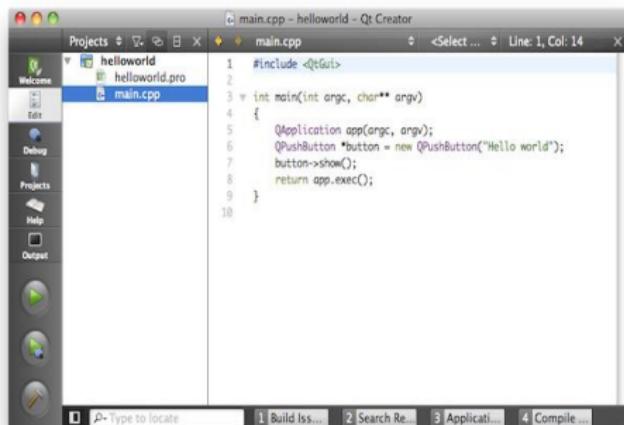
Level	Function	File	Name	Value	Type
0	mach_msg...		app		QApplication
1	mach_msg...		argc	1	int
2	__CFRunLoop...		argv	0xbffff9fc	char **
3	CFRunLoop...		button	0xe8fdb0	QPushButton *
4	CFRunLoop...		*button		QAbstractButton
5	QEEventDisp...		QWidget	...	QObject
6	QEEventLoop...		className	QPushButton	
7	QEEventLoop...		parent	0x0	QObject *
8	QCoreApplication...		properties	<72 items>	QObjectProperty...
9	main	main.cpp	signals	<8 items>	QObjectSignalList
			slots	<32 items>	QObjectSlotList

The bottom navigation bar includes tabs for Build, Search, Application, and Compilation.

Qt Creator Demo

"Hello World"

- Creation of an empty Qt Widget project
- Adding the main.cpp source file
- Writing of the Qt Hello World Code
- Running the application
- Debugging the application



The screenshot shows the Qt Creator IDE interface. On the left is the 'Projects' panel with a 'helloworld' project containing a 'helloworld.pro' file and a selected 'main.cpp' file. The main window is titled 'main.cpp - helloworld - Qt Creator'. It displays the following C++ code:

```
#include <QtGui>
int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QPushButton *button = new QPushButton("Hello world");
    button->show();
    return app.exec();
}
```

At the bottom of the interface are tabs for 'Build Issues...', 'Search Results...', 'Applications...', and 'Compile...'. There is also a search bar with the placeholder 'Type to locate'.

Fundamentals of Qt

- The Story of Qt
- Developing a Hello World Application
- Hello World using Qt Creator
- Practical Tips for Developers

C++ knowledge

- Objects and classes
 - Declaring a class, inheritance, calling member functions etc.
- Polymorphism
 - That is virtual methods
- Operator overloading

Qt Documentation

- Reference Documentation
 - All classes documented
 - Contains tons of examples
- Collection of Howto's and Overviews
- A set of Tutorials for Learners



Modules and Include files

- Qt Modules
 - QtCore, QtGui, QtWidgets, QDom, QSql, QtNetwork, QtTest .. [See documentation](#)
- Enable Qt Module in qmake .profile:
 - QT += network
- Default: qmake projects use QtCore and QtGui
 - Any Qt class has a header file.

```
#include <QLabel>
#include <QtWidgets/QLabel>
```
 - Any Qt Module has a header file.

```
#include <QtGui>
```

Includes and Compilation Time

Module includes

```
#include <QtGui>
```

- Precompiled header and the compiler
 - If **not** supported may add extra compile time
 - If supported may speed up compilation
 - Supported on: Windows, Mac OS X, Unix

Class includes

```
#include <QLabel>
```

- Reduce compilation time
 - Use class includes (#include <QLabel>)
 - Forward declarations (class QLabel;)

Place module includes before other includes.

Summary

- What is Qt
- Which code lines do you need for a minimal Qt application?
- What is a .pro file?
- What is qmake, and when is it a good idea to use it?
- What is a Qt module and how to enable it in your project?
- How can you include a QLabel from the QtGui module?
- Name places where you can find answers about Qt problems

Core classes



Core classes

- Qt's Object Model
 - QObject
 - QWidget
 - Variants
 - Properties
- String Handling
- Container Classes

Objectives

- What the Qt object model is
- The basics of the widget system in C++
- Which utility classes exists to help you in C++

Core classes

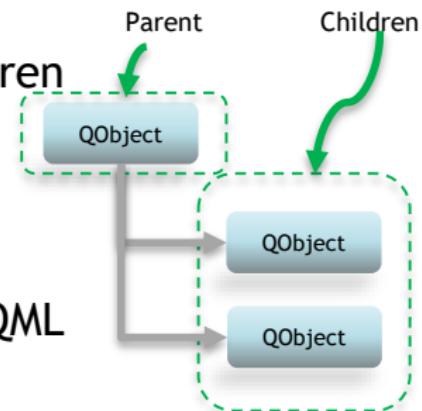
- Qt's Object Model
 - QObject
 - QWidget
 - Variants
 - Properties
- String Handling
- Container Classes

QObject

- QObject is the heart of Qt's object model
- Include these features:
 - Memory management
 - Object properties
 - Signals and slots
 - Event handling
- QObject has no visual representation

Object tree

- QObjects organize themselves in object trees
 - Based on parent-child relationship
- `QObject(QObject *parent = 0)`
 - Parent adds object to list of children
 - Parent owns children
- Widget Centric
 - Used intensively with `QWidget`
 - Less so when using `Qt/C++` from `QML`



Note: Parent-child relationship is NOT inheritance

Creating Objects

- **On Heap** - QObject with parent:

```
QTimer* timer = new QTimer(this);
```

- Parent takes ownership.
- Copy is disabled

- **On Stack** - value types

- QString, QStringList, QColor

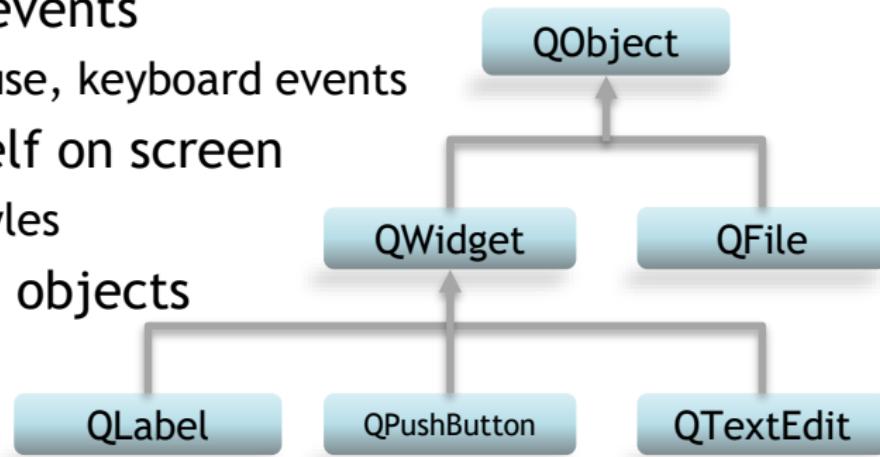
- **On Stack** - QObject without parent: Exceptions

- QFile, QApplication (Usually allocated on the stack)
- Top level QWidgets: QMainWindow

- **Stack or Heap** - QDialog - depending on lifetime

QWidget

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



QWidget

- new QWidget(0)
 - Widget with no parent="window"
- QWidget's children
 - Positioned in parent's coordinate system
 - Clipped by parent's boundaries
- QWidget parent
 - Propagates state changes
 - hides/shows children when it is hidden/shown itself
 - enables/disables children when it is enabled/disabled itself

Widgets contain other widgets

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

Container Widget

```
// container (window) widget creation
QWidget* container = new QWidget;
QLabel* label = new QLabel("Note:", container);
QTextEdit* edit = new QTextEdit(container);
QPushButton* clear = new QPushButton("Clear", container);
QPushButton* save = new QPushButton("Save", container);
// widget layout
QVBoxLayout* outer = new QVBoxLayout();
outer->addWidget(label);
outer->addWidget(edit);
QHBoxLayout* inner = new QHBoxLayout();
inner->addWidget(clear);
inner->addWidget(save);
container->setLayout(outer);
outer->addLayout(inner); // nesting layouts
```

[Demo](#)



QVariant

- QVariant
 - Union for common Qt "value types" (copyable, assignable)
 - Supports implicit sharing (fast copying)
 - Supports user types
- Use cases:

```
QVariant property(const char* name) const;  
void setProperty(const char* name, const  
QVariant &value);  
  
class QAbstractItemModel {  
  
virtual QVariant data( const QModelIndex&  
index, int role );  
}
```

- [Qvariant Documentation](#)

QVariant

- For QtCore types

```
QVariant variant(42);
int value = variant.toInt(); // read
// back
qDebug() << variant.typeName(); // int
```

- For non-core and custom types:

```
QVariant variant =
QVariant::fromValue(QColor(Qt::red));
QColor color = variant.value<QColor>();
// read back
qDebug() << variant.typeName();
// "QColor"
```

Properties

- Direct access (Broken, due to private headers)

```
QQuickRectangle* rectangle=
root->findChild<QQuickRectangle*>("myRect");
int height = rectangle->height();
```

- Generic property access:

```
QObject* rectangle = root-
>findChild<QObject*>("myRect");
int height = rectangle-
>property("height").value<int>();
```

Properties

- **Q_Property is a macro:**

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

- **Property access methods:**

```
QVariant property(const char* name) const;  
void setProperty(const char* name, const QVariant &value);
```

- If name is not declared as a **Q_PROPERTY**

- -> **dynamic property**
- Not accessible from Qt Quick.

Note:

- **Q_OBJECT** macro required for properties to work
- **QMetaObject** knows nothing about dynamic properties

Properties

- QMetaObject support property introspection

```
const QMetaObject *metaObject = object->metaObject();
const QString className = metaObject->className();
const int propertyCount = metaObject->propertyCount();
for ( int i=0; i<propertyCount; ++i ) {
    const QMetaProperty metaProperty = metaObject-
>property(i);
    const QString typeName = metaProperty.typeName()
    const QString propertyName = metaProperty.name();
    const QVariant value = object-
>property(metaProperty.name());
}
```

[Demo](#)

Core classes

- Qt's Object Model
 - QObject
 - QWidget
 - Variants
 - Properties
- **String Handling**
- Container Classes

QString

Strings can be created in a number of ways:

- Conversion constructor and assignment operators:

```
QString str("abc");  
str = "def";
```

- From a number using a static function:

```
QString n = QString::number(1234);
```

- From a char pointer using the static functions:

```
QString text = QString::fromLatin1("Hello Qt");  
QString text = QString::fromUtf8(inputText);  
QString text = QString::fromLocal8Bit(cmdLineInput);  
QString text = QStringLiteral("Literal String");  
(Assumed to be UTF-8)
```

- From char pointer with translations:

```
QString text = tr("Hello Qt");
```

QString

- operator+ and operator+=

```
QString str = str1 + str2;  
fileName += ".txt";
```

- simplified() // removes duplicate whitespace
- left(), mid(), right() // part of a string
- leftJustified(), rightJustified() // padded version

```
QString s = "apple";  
QString t = s.leftJustified(8, '.');  
// t == "apple..."
```

QString

Data can be extracted from strings.

- Numbers:

```
QString text = ...;  
int value = text.toInt();  
float value = text.toFloat();
```

- Strings:

```
QString text = ...;  
QByteArray bytes = text.toLatin1();  
QByteArray bytes = text.toUtf8();  
QByteArray bytes = text.toLocal8Bit();
```

QString::arg()

```
int i = ...;
int total = ...;
QString fileName = ...;
QString status = tr("Processing file %1 of %2:
%3")
    .arg(i).arg(total).arg(fileName);
double d = 12.34;
QString str = QString::fromLatin1("delta:
%1").arg(d, 0, 'E', 3);
// str == "delta: 1.234E+01"
```

- Safer: arg(QString,...,QString) (` `multi-arg()` `).
- But: only works with QString arguments.

QString

- Obtaining raw character data from a QByteArray:

```
char *str = bytes.data();
```

```
const char *str = bytes.constData();
```

WARNING:

- Character data is only valid for the lifetime of the byte array.
- Calling a non-const member of bytes also invalidates ptr.
- Either copy the character data or keep a copy of the byte array.

QString

- `length()`
- `endsWith()` and `startsWith()`
- `contains()`, `count()`
- `indexOf()` and `lastIndexOf()`

Expression can be characters, strings, or regular expressions

QString

- `QString::split()`, `QStringList::join()`
- `QStringList::replaceInStrings()`
- `QStringList::filter()`

QString

- QRegExp supports
 - Regular expression matching
 - Wildcard matching
- QString cap(int)
QStringList capturedTexts()

```
QRegExp rx("^\d\d?"); // match integers 0 to 99
rx.indexIn("123"); // returns -1 (no match)
rx.indexIn("-6"); // returns -1 (no match)
rx.indexIn("6"); // returns 0 (matched as position 0)
```
- [Regular expression documentation](#)

Core classes

- Qt's Object Model
 - QObject
 - QWidget
 - Variants
 - Properties
- String Handling
- Container Classes

Container Classes

General purpose template-based container classes

- **QList<QString>** - *Sequence Container*
 - Other: QLinkedList, QStack , QQueue ...
- **QMap<int, QString>** - *Associative Container*
 - Other: QHash, QSet, QMultiMap, QMultiHash

Qt's Container Classes compared to STL

- Lighter, safer, and easier to use than STL containers
- If you prefer STL, feel free to continue using it.
- Methods exist that convert between Qt and STL
 - e.g. you need to pass std::list to a Qt method

Container Classes

Using QList

```
QList<QString> list;
list << "one" << "two" << "three";
QString item1 = list[1]; // "two"
for(int i=0; i<list.count(); i++) {
const QString &item2 = list.at(i);
}
int index = list.indexOf("two"); // returns 1
```

Using QMap

```
QMap<QString, int> map;
map["Norway"] = 5; map["Italy"] = 48;
int value = map["France"]; // inserts key if not exists
if(map.contains("Norway")) {
int value2 = map.value("Norway"); // recommended lookup
}
```

Complexity

How fast is a function when number of items grow

Sequential Container

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

Complexity

Associative Container

	Lookup	Insert
Qmap	$O(\log(n))$	$O(\log(n))$
QHash	$O(1)$	$O(1)$

all complexities are amortized

Classes in Container

- Class must be an *assignable data type*
- Class is *assignable*, if:

```
class Contact {  
public:  
    Contact() {} // default constructor  
    Contact(const Contact &other); // copy  
    constructor  
    // assignment operator  
    Contact &operator=(const Contact &other);  
};
```

- If *copy constructor or assignment operator is not provided*
 - C++ will provide one (uses member copying)
- If *no constructors provided*
 - Empty default constructor provided by C++

Container Keys

- Type K as key for QMap:
 - `bool K::operator<(const K&)` or
`bool operator<(const K&, const K&)`
 - `bool Contact::operator<(const Contact& c);`
 - `bool operator<(const Contact& c1, const Contact& c2);`
 - [QMap Documentation](#)
- Type K as key for QHash or QSet:
 - `bool K::operator==(const K&)` or `bool operator==(const K&, const K&)`
 - `uint qHash(const K&)`
 - [QHash Documentation](#)

Iterators

- Allow reading a container's content sequentially
- **Java-style iterators:** simple and easy to use
 - `QListIterator<...>` for read
 - `QMutableListIterator<...>` for read-write
- **STL-style iterators** slightly more efficient
 - `QList::const_iterator` for read
 - `QList::iterator()` for read-write
- Same works for QSet, QMap, QHash, ...

Iterators

Java style

- Example QList iterator

```
QList<QString> list;  
list << "A" << "B" << "C" << "D";  
QListIterator<QString> it(list);
```

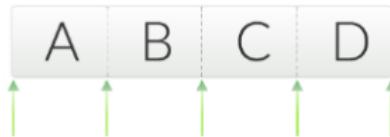
- Forward iteration

```
while(it.hasNext()) {  
    qDebug() << it.next(); // A B C D  
}
```

- Backward iteration

```
it.toBack(); // position after the last item  
while(it.hasPrevious()) {  
    qDebug() << it.previous(); // D C B A  
}
```

- [QListIterator Documentation](#)



Modifying During Iteration

- Use *mutable* versions of the iterators

- e.g. `QMutableListIterator`.

```
QList<int> list;
list << 1 << 2 << 3 << 4;
QMutableListIterator<int> i(list);
while (i.hasNext()) {
    if (i.next() % 2 != 0)
        i.remove();
}
// list now 2, 4
```

- `remove()` and `setValue()`

- Operate on items just jumped over using `next()`/`previous()`

- `insert()`

- Inserts item at current position in sequence
 - `previous()` reveals just inserted item

QMap and QHash

- `next()` and `previous()`
 - Return Item class with `key()` and `value()`
- Alternatively use `key()` and `value()` from iterator

```
QMap<QString, QString> map;
map["Paris"] = "France";
map["Guatemala City"] = "Guatemala";
map["Mexico City"] = "Mexico";
map["Moscow"] = "Russia";
QMutableMapIterator<QString, QString> i(map);
while (i.hasNext()) {
    if (i.next().key().endsWith("City"))
        i.remove();
}
// map now "Paris", "Moscow"
```

- [Demo](#)

Iterators

STL-style



- Example QList iterator

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

- Forward mutable iteration

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

- Backward mutable iteration

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

- QList<QString>::const_iterator for read-only



foreach

- It is a macro, feels like a keyword
foreach (variable, container) statement

```
foreach (const QString& str, list) {  
    if (str.isEmpty())  
        break;  
    qDebug() << str;  
}
```
- Break and continue as normal
- Modifying the container while iterating
 - results in container being copied
 - iteration continues in unmodified version
- Not possible to modify item
 - iterator variable is a const reference.

Qt algorithms

- **qSort(begin, end)** sort items in range
- **qFind(begin, end, value)** find value
- **qEqual(begin1, end1, begin2)** checks two ranges
- **qCopy(begin1, end1, begin2)** from one range to another
- **qCount(begin, end, value, n)** occurrences of value in range
- For parallel (ie. multi-threaded) algorithms
 - QtConcurrent()
- [QtAlgorithms Documentation](#)

Examples

- Counting 1's in list

```
QList<int> list;  
list << 1 << 2 << 3 << 1;  
int count = 0;  
qCount(list, 1, count); // count the 1's  
qDebug() << count; // 2 (means 2 times 1)
```

- Copy list to vector

```
QList<QString> list;  
list << "one" << "two" << "three";  
QVector<QString> vector(3);  
qCopy(list.begin(), list.end(),  
vector.begin());  
// vector: [ "one", "two", "three" ]
```

Examples

- Case insensitive sort

```
bool lessThan(const QString& s1, const QString&
s2) {
    return s1.toLower() < s2.toLower();
}
// ...
QList<QString> list;
list << "AlPhA" << "beTA" << "gamma" << "DELTA";
qSort(list.begin(), list.end(), lessThan);
// list: [ "AlPhA", "beTA", "DELTA", "gamma" ]
```

Implicit Sharing

If an object is copied, then its data is copied *only when the data of one of the objects is changed*

- Shared class has a pointer to shared data block
 - Shared data block = reference counter and actual data
- Assignment is a shallow copy
- Changing results into deep copy (detach)

```
QList<int> l1, l2; l1 << 1 << 2;  
l2 = l1; // shallow-copy: l2 shares data with l1  
l2 << 3; // deep-copy: change triggers detach from l1
```

Important to remember when inserting items into a container, or when returning a container.

Summary

- Explain how object ownership works in Qt?
- What are the key responsibilities of QObject?
- What does it mean when a QWidget has no parent?
- What is the purpose of the class QVariant?
- What do you need to do to support properties in a class?
- Name the different ways to go through the elements in a list, and
- Discuss advantages and disadvantages of each method.

Widgets



Widgets

- Common Widgets
- Layout Management
- Custom Widgets

Widgets

- **Common Widgets**
 - Text widgets
 - Value based widgets
 - Organizer widgets
 - Item based widgets
- **Layout Management**
 - Geometry management
 - Advantages of layout managers
 - Qt's layout managers
 - Size policies
- **Custom Widgets**
 - Rules for creating own widgets

Widgets

- Common Widgets
- Layout Management
- Custom Widgets

Text Widgets

- **QLabel**

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

- **QLineEdit**

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

- **QTextEdit**

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



Button widgets

- **QAbstractButton**

Abstract base class of buttons

- **QPushButton**

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

- **QRadioButton**

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

- **QCheckBox**

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



Value Widgets

- **QSlider**

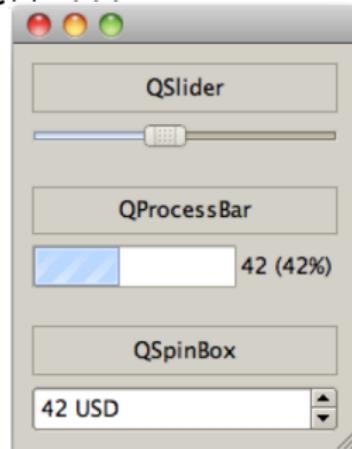
```
slider = new QSlider(Qt::Horizontal, parent);
slider->setRange(0, 99);
slider->setValue(42);
connect(slider, SIGNAL(valueChanged(int)), ...)
```

- **QProgressBar**

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

- **QSpinBox**

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



Organizer Widgets

- **QGroupBox**

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

- **QTabWidget**

```
tab = new QTabWidget(parent);
tab->addWidget(widget, icon, "Tab 1");
connect(tab, SIGNAL(currentChanged(int)) ...
```

- **setCurrentWidget(widget)**
 - Displays page associated by widget
- **setTabPosition(position)**
 - Defines where tabs are drawn
- **setTabsClosable(bool)**
 - Adds close buttons



Item Widgets

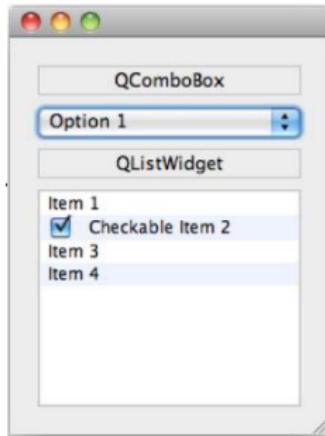
- **QComboBox**

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

- **QListWidget**

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

- Other Item Widgets: QTableWidget, QTreeWidget



Other Widgets

- **QToolBox**
Column of tabbed widget items
- **QDateEdit, QTimeEdit, QDateTimeEdit**
Widget for editing date and times
- **QCalendarWidget**
Monthly calendar widget
- **QToolButton**
Quick-access button to commands
- **QSplitter**
Implements a splitter widget
- **QStackedWidget**
Stack of widgets
Only one widget visible at a time
- [Widget Documentation](#)

Widgets

- Common Widgets
- Layout Management
- Custom Widgets

Layout widgets

Definition

Layout: 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.
- Disadvantage
 - Need to design your layout first.

Layout widgets

- Place and resize widgets (Without layouts)
 - move()
 - resize()
 - setGeometry()
- Example:

```
QWidget *parent = new QWidget(...);  
parent->resize(400, 400);  
QCheckBox *cb = new QCheckBox(parent);  
cb->move(10, 10);
```

Layout widgets

- On layout managed widgets never call `setGeometry()`, `resize()`, or `move()`
- Preferred
 - Override
 - `sizeHint()`
 - `minimumSizeHint()`
- Or call
 - `setFixedSize()`
 - `setMinimumSize()`
 - `setMaximumSize()`

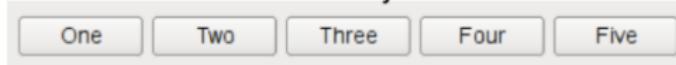
Layout classes

- **QHBoxLayout**
 - Lines up widgets horizontally
- **QVBoxLayout**
 - Lines up widgets vertically
- **QGridLayout**
 - Arranges the widgets in a grid
- **QFormLayout**
 - Lines up a (label, widget) pairs in two columns.
- **QStackedLayout**
 - Arranges widgets in a stack
 - only topmost is visible

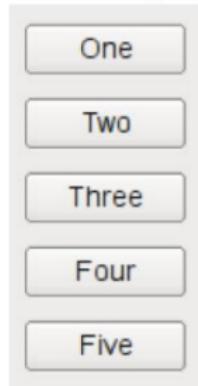
QHBoxLayout & QVBoxLayout

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

QHBoxLayout



VBoxLayout



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

QGridLayout

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

Demo

- Additional
 - `setColumnMinimumWidth()` (minimum width of column)
 - `setRowMinimumHeight()` (minimum height of row)



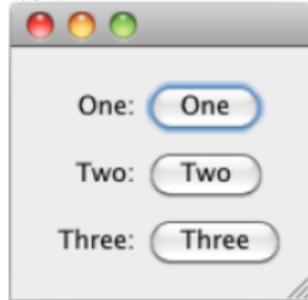
No need to specify rows and columns before adding children.

QFormLayout

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

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

- Form layout with clean looks and mac style



Hands-on

Lab 4 Contact form

- [Objectives](#)
- [Template code](#)

Layout Terms

- **Stretch**

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

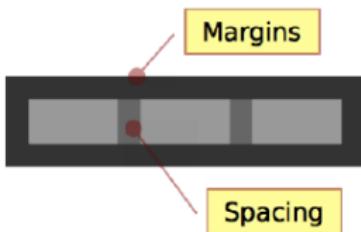


- **Contents Margins**

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

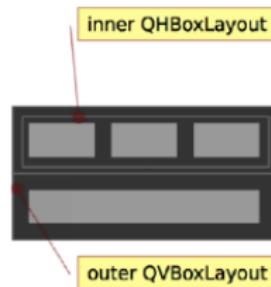
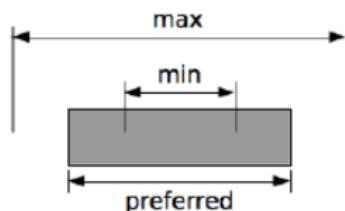
- **Spacing**

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



Layout Terms

- **Strut**
 - *Limits perpendicular box dimension*
 - e.g. height for QHBoxLayout
 - *Only for box layouts*
- **Min, max and fixed sizes**
 - QWidget::setMinimumSize(QSize)
 - QWidget::setMaximumSize(QSize)
 - QWidget::setFixedSize(QSize)
 - *Individual width and height constraints also available*
- **Nested Layouts**
 - *Allows flexible layouts*
 - QLayout::addLayout(...)



Widgets

Size Policies



- **QSizePolicy** describes interest of widget in resizing
`QSizePolicy policy = widget->sizePolicy();
policy.setHorizontalPolicy(QSizePolicy::Fixed);
widget->setSizePolicy(policy);`
- One policy per direction (horizontal and vertical)
- Button-like widgets set size policy to the following:
 - may stretch horizontally
 - are fixed vertically
 - Similar to QLineEdit, QProgressBar, ...
- Widgets which provide scroll bars (e.g. QTextEdit)
 - Can use additional space
 - Work with less than sizeHint()
- sizeHint() : recommended size for widget



Widgets

Size Policies

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

Hands-on

- Lab 4: Layout buttons
 - [Objective](#)
 - [Template code](#)

Summary

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

Widgets

- Common Widgets
- Layout Management
- Custom Widgets

Custom Widget Creation

- It's as easy as deriving from QWidget

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

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

Custom Widget

Base class and Event Handlers

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

Custom Widget

Drawing a Widget

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

Custom Widget

Internal States and Subclassing

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

Custom Widget

Widget Constructor

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

Hands-on

- Lab 5: File chooser
 - [Objective](#)
 - [Template code](#)
- Lab 6: Compass widget
 - [Objective](#)
 - [Template code](#)

Object communication



Object communication

- Signals & Slots
- Event Handling

Objectives

- How objects communication
- Details of signals & slots
- Which variations for signal/slot connections exist
- How to create custom signals & slots
- What the role of the Qt event loop is
- How Qt handles events

Object communication

- Between objects
Signals & Slots
- Between Qt and the application
Events
- Between Objects on threads
Signal & Slots + Events
- Between Applications
DBus, QSharedMemory

Callbacks

General Problem

How do you get from "the user clicks a button" to your logic?

- Possible solutions
 - **Callbacks**
 - Based on function pointers
 - Not type-safe
 - **Observer Pattern (Listener)**
 - Based on interface classes
 - Needs listener registration
 - Many interface classes
- Qt uses
 - Signals and slots for high-level (semantic) callbacks
 - Virtual methods for low-level (syntactic) events.

Object communication

- Signals & Slots
- Event Handling

Connecting signals and slots

```
void  
Qslider::mousePressEvent(...)  
{  
    emit valueChanged(value)  
}
```

```
void QProgressBar::setValue(int  
value)  
{  
    mvalue = value  
}
```



Signal emitted



Signal/slot connection



slot implemented



Demo

```
QObject::connect(slider,SIGNAL(  
valueChanged),progressbar,  
SLOT(setValue))
```

Connection variants

- Qt 4 style:

```
connect(slider, SIGNAL(valueChanged(int)),  
        progressBar, SLOT(setValue(int)));
```

- Using function pointers (Qt5):

```
connect( slider, &QSlider::valueChanged,  
        progressBar, &QSpinBox::setValue );
```

[Demo](#)

- Using non-member function:

```
static void printValue(int value) {...}  
connect( slider, &QSignal::valueChanged,  
        &printValue );
```

[Demo](#)

Custom slots

- File: **myclass.h**

```
class MyClass : public QObject
{
    Q_OBJECT // marker for moc
    // ...
public slots:
void setValue(int value); // a custom slot
};
```

- File: **myclass.cpp**

```
void MyClass::setValue(int value) {
    // slot implementation
}
```

[Demo](#)

Custom signals

- File: **myclass.h**

```
class MyClass : public QObject
{
    Q_OBJECT // marker for moc
    // ...
signals:
void valueChanged(int value); // a custom signal
};
```

- File: **myclass.cpp**

```
// No implementation for a signal
```

- Sending a signal

```
emit valueChanged(value);
```

- Demo

Q_OBJECT - flag

- **Q_OBJECT**
 - Enhances QObject with meta-object information
 - Required for signals
 - Required for slots when using the Qt4 way
- **moc** creates meta-object information
 - `moc -o moc_myclass.cpp myclass.h`
 - `c++ -c myclass.cpp; c++ -c moc_myclass.cpp`
 - `c++ -o myapp moc_myclass.o myclass.o`
- **qmake** takes care of mocing files for you

Variations of Signal/Slot

Signal(s)	Connect to	Slot(s)
One	✓	Many
Many	✓	One
One	✓	Another signal

- **Signal to Signal connection**

```
connect(bt, SIGNAL(clicked()), this,  
        SIGNAL(okSignal()));
```

- **Not allowed to name parameters**

```
connect( m_slider, SIGNAL( valueChanged(  
        int value ) )  
        this, SLOT( setValue( int newValue ) ) )
```

Rules for Signal/Slot

Can ignore arguments, but not create values from nothing. Eg:

Signal		Slot
rangeChanged(int,int)	✓ ✓ ✓	setRange(int,int) setValue(int) Update()
valueChanged(int)	✓ ✓ X	setValue(int) Update() setRange(int,int)
textChanged(QString)	X	setValue(int)

Hands-on

- Lab 1: Select colour
 - [Objective](#)
 - [Template code](#)
- Lab 2: Slider
 - [Objective](#)
 - [Template code](#)

Object communication

- Signals & Slots
- Event Handling

Event Processing

Qt is an event-driven UI toolkit

`QApplication::exec()` runs the *event loop*

1. Generate Events

- by input devices: keyboard, mouse, etc.
- by Qt itself (e.g. timers)

2. Queue Events

- by event loop

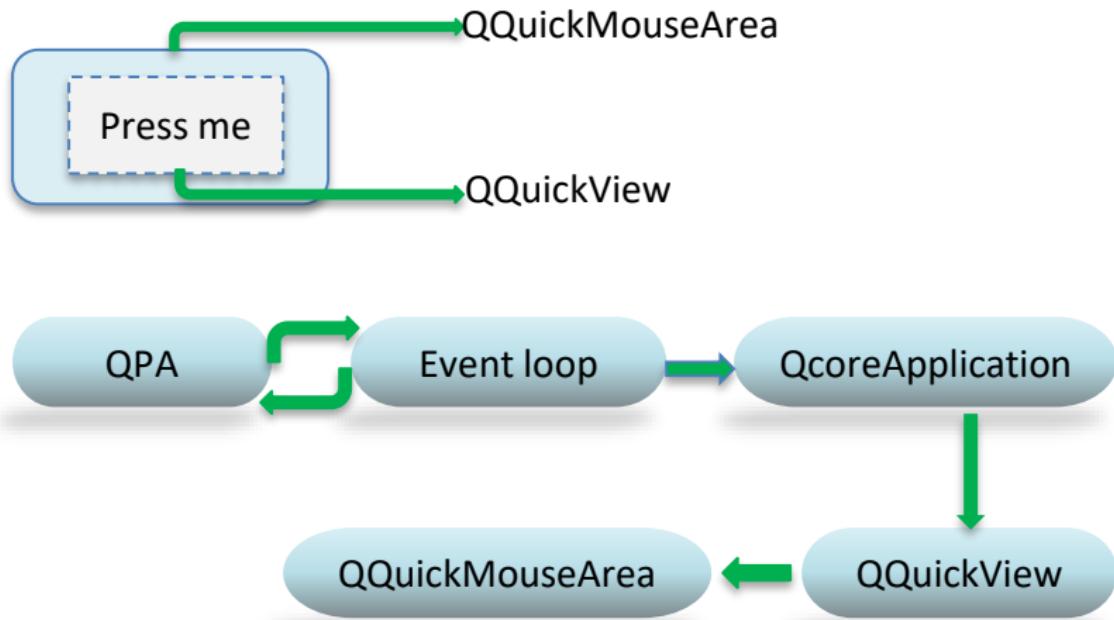
3. Dispatch Events

- by QApplication to receiver: QObject
 - *Key events sent to widget with focus*
 - *Mouse events sent to widget under cursor*

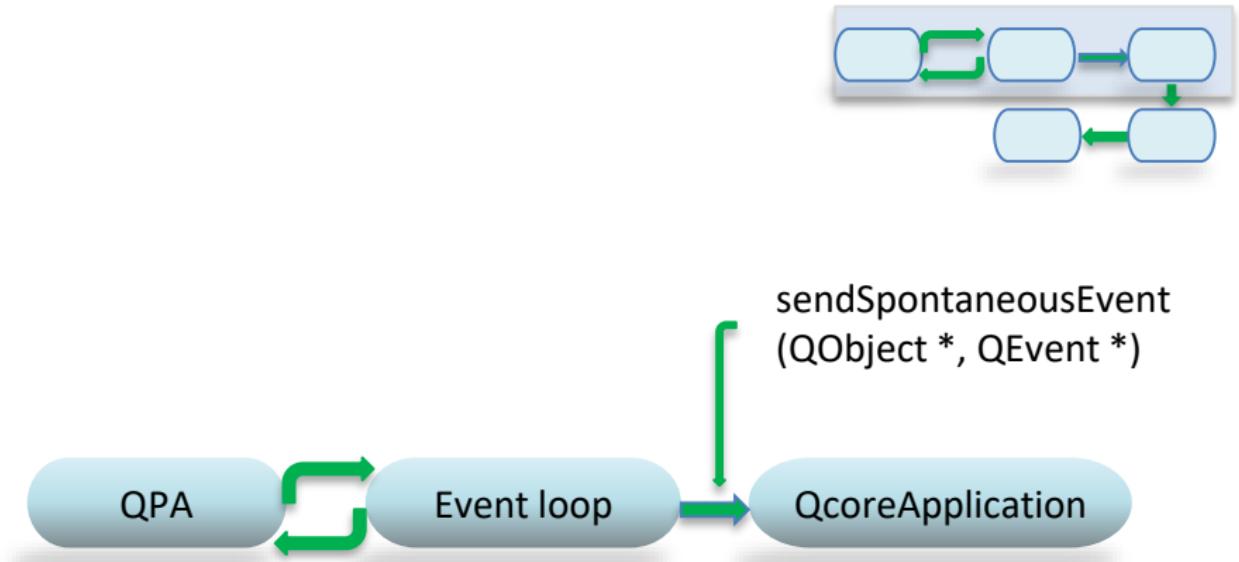
4. Handle Events

- by QObject event handler methods

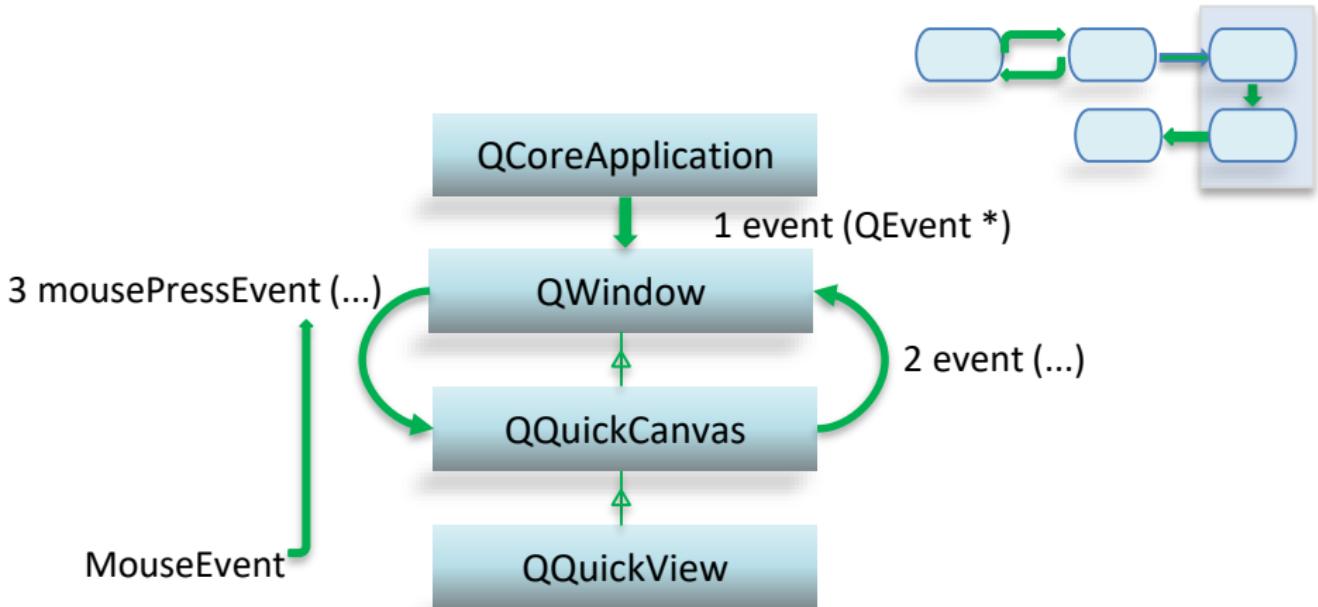
Event Processing



Event Processing

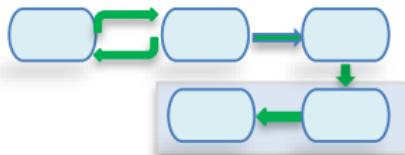
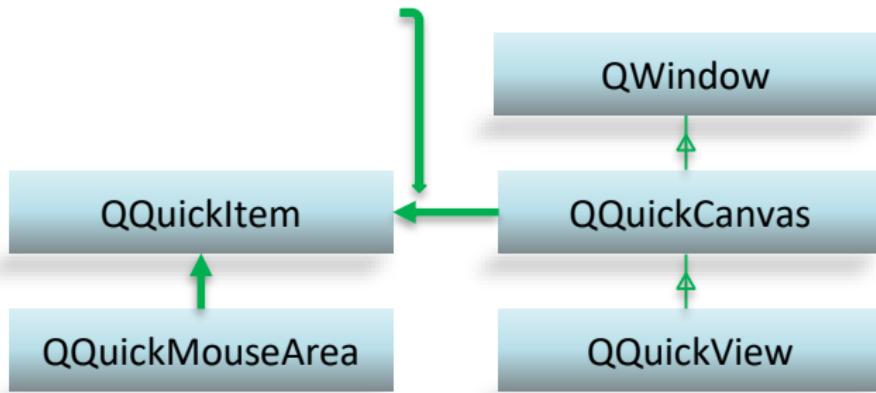


Event Processing



Event Processing

mousePressEvent (QMouseEvent *)



Event Handling

- `QObject::event(QEvent *event)`
 - Handles all events for this object
- Specialized event handlers for `QWidget` and `QQuickItem`:
 - `mousePressEvent()` for mouse clicks
 - `touchEvent()` for key presses
- Accepting an Event
 - `event->accept()` / `event->ignore()`
 - Accepts or ignores the event
 - Accepted is the default.
- Event propagation
 - Happens if event is ignored
 - Might be propagated to parent widget

[Demo](#)

Event Handling

- QCloseEvent delivered to top level widgets (windows)
- Accepting event allows window to close
- Ignoring event keeps window open

```
void MyWidget::closeEvent(QCloseEvent *event) {  
    if (maybeSave()) {  
        writeSettings();  
        event->accept(); // close window  
    } else {  
        event->ignore(); // keep window  
    }  
}
```

Demo

Summary

- How do you connect a signal to a slot?
- How would you implement a slot?
- How would you emit a signal?
- Can you return a value from a slot?
- When do you need to run qmake?
- Where do you place the `Q_OBJECT` macro and when do you need it?
- What is the purpose of the event loop
- How does an event make it from the device to an object in Qt?

Qt Multithreading



Multithreading

- Most GUI applications have a single thread of execution in which the event loop is running
- However, if the user invokes a time consuming operation the interface freezes. We can work around this in different ways:
 - Using the `QApplication::processEvent()` during long tasks to make sure events (key, window, etc.) are delivered and the UI stays responsive.
 - Using threads to perform the long running tasks. Qt has a number of options for this.

Multithreading Technologies

- QThread: Low-Level API with Optional Event Loops
- QThreadPool and QRunnable: Reusing Threads
- QtConcurrent: Using a High-level API
- WorkerScript: Threading in QML

QThread

- QThread is the central class in Qt to run code in a different thread
- It's a QObject subclass
 - Not copyable/moveable
 - Has signals to notify when the thread starts/finishes
- It is meant to manage a thread

QThread usage

- To create a new thread executing some code, subclass `QThread` and reimplement `run()`
- Then create an instance of the subclass and call `start()`
- Threads have priorities that you can specify as an optional parameter to `start()`, or change with `setPriority()`
- The thread will stop running when (some time after) returning from `run()`
- `QThread::isRunning()` and `QThread::isFinished()` provide information about the execution of the thread
- You can also connect to the `QThread::started()` and `QThread::finished()` signals
- A thread can stop its execution temporarily by calling one of the `QThread::sleep()` functions
 - Generally a *bad idea*, being event driven (or polling) is much much Better
- You can wait for a `QThread` to finish by calling `wait()` on it
 - Optionally passing a maximum number of milliseconds to wait

QThread caveats

From a non-main thread you cannot:

- Perform any GUI operation
 - Including, but not limited to: using any QWidget / Qt Quick / Qpixmap APIs
 - Using QImage, QPainter, etc. (i.e. "client side") is OK
 - Using OpenGL may be OK: check at runtime
`QOpenGLContext::supportsThreadedOpenGL()`
- Call Q(Core|Gui)Application::exec()
- Be sure to always destroy all the QObjects living in secondary threads before destroying the corresponding QThread object
- Do not *ever* block the GUI thread

QThread usage

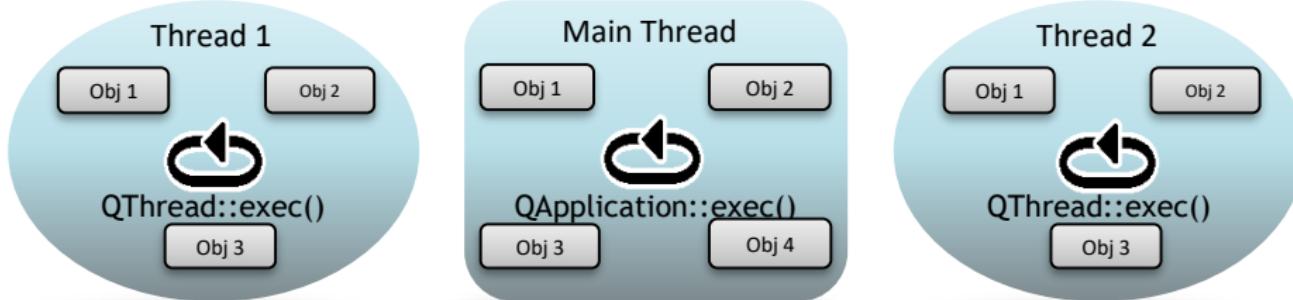
- There are two basic strategies of running code in a separate thread with QThread:
 - Without an event loop
 - With an event loop

QThread usage without an event loop

- Subclass QThread and override `QThread::run()`
- Create an instance and start the new thread via `QThread::start()`
- Demo

QThread usage with an event loop

- An event loop is necessary when dealing with timers, networking, queued connections, and so on.
- Qt supports per-thread event loops:



- Each thread-local event loop delivers events for the QObjects living in that thread.

QThread usage with an event loop

- We can start a thread-local event loop by calling QThread::exec() from within run():

```
1 class MyThread : public QThread {  
2 private:  
3     void run() override {  
4         auto socket = new QTcpSocket;  
5         socket->connectToHost(...);  
67         exec(); // run the event loop  
89         // cleanup  
10    }  
11}; Demo
```

- QThread::quit() or QThread::exit() will quit the event loop
- We can also use QEventLoop
 - Or manual calls to QApplication::processEvents()
- The default implementation of QThread::run() actually calls QThread::exec()
- This allows us to run code in other threads without sub classing QThread:

QtConcurrent

- QtConcurrent is a namespace that provides higher-level classes and algorithms for writing concurrent software.
- Using QtConcurrent's functional map/filter/reduce algorithms, which apply functions in parallel to each item in a container.
- You can write a program that automatically takes advantage of the system's multiple cores by distributing the processing across the threads managed by the thread pool.

QtConcurrent

- Qt Concurrent supports several STL-compatible container and iterator types, but works best with Qt containers that have random-access iterators, such as QList or QVector
- Demo

QThreadPool and QRunnable

- Creating and destroying threads frequently can be expensive.
- To avoid the cost of thread creation, a thread pool can be used.
- A thread pool is a place where threads can be parked and fetched.
- We derive a class from QRunnable. The code we want to run in another thread needs to be placed in the reimplemented QRunnable::run() method.
- Demo

Synchronization



Synchronization

- Any concurrent access to shared resources must not result in a data race
- Two conditions for this to happen:
 1. At least one of the accesses is a write
 2. The accesses are not atomic and no access happens before the other

Synchronization

Qt has a complete set of cross-platform, low-level APIs for dealing with synchronization:

- `QMutex` is a mutex class (recursive and non-recursive)
- `QSemaphore` is a semaphore
- `QWaitCondition` is a condition variable
- `QReadWriteLock` is a shared mutex
- `QAtomicInt` is an atomic int
- `QAtomicPointer<T>` is an atomic pointer to T
- Demo'

Thread safety in Qt

A function is:

- **Thread safe:** if it's safe for it to be invoked at the same time, from multiple threads, on the same data, without synchronization
- **Reentrant:** if it's safe for it to be invoked at the same time, from multiple threads, on different data; otherwise it requires external synchronization
- **Non-reentrant (thread unsafe):** if it cannot be invoked from more than one thread at all

For classes, the above definitions apply to non-static member functions when invoked on the same instance.

Examples

- Thread safe:
 - QMutex
 - QObject::connect()
 - QCoreApplication::postEvent()
- Reentrant:
 - QString
 - QVector
 - QImage
 - value classes in general
- Non-reentrant:
 - QWidget (including all of its subclasses)
 - QQuickItem
 - QPixmap
 - in general, GUI classes are usable only from the main thread

QtMultimedia



QtMultimedia

- Qt Multimedia is an essential module that provides a rich set of QML types and C++ classes to handle multimedia content.
- It also provides necessary APIs to access the camera and radio functionality.

Features

- Access raw audio devices for input and output
- Play low latency sound effects
- Play media files in playlists (such as compressed audio or video files)
- Record audio and compress it
- Tune and listen to radio stations
- Use a camera, including viewfinder, image capture, and movie recording
- Play 3D positional audio with Qt Audio Engine
- Decode audio media files into memory for processing
- Accessing video frames or audio buffers as they are played or recorded

Audio

- Qt Multimedia offers a range of audio classes, covering both low and high level approaches to audio input, output and processing.
- For playing media or audio files that are not simple, uncompressed audio, you can use the [QMediaPlayer](#) C++ class.
- The QMediaPlayer class and associated QML types are also capable of playing video, if required.
- The compressed audio formats supported does depend on the operating system environment, and also what media plugins the user may have installed.
- For recording audio to a file, the [QAudioRecorder](#) class allows you to compress audio data from an input device and record it.
- Demo

Video

- We can use the QMediaPlayer class to decode a video file, and display it using [QVideoWidget](#), [QGraphicsVideoItem](#), or a custom class.
- Demo

Painting & styling

Objectives

- Painting on Widgets
- Color Handling
- Painting Operations
- Style Sheets

Objectives

- Painting
 - You paint with a painter on a paint device during a paint event
 - Qt widgets know how to paint themselves
 - Often widgets look like we want
 - Painting allows device independent 2D visualization
 - Allows to draw pie charts, line charts and many more
- StyleSheets
 - Fine grained control over the look and feel
 - Easily applied using style sheets in CSS format

Module Objectives

Covers techniques for general 2D graphics and styling applications.

- **Painting**
 - Painting infrastructure
 - Painting on widget
- **Color Handling**
 - Define and use colors
 - Pens, Brushes, Palettes
- **Shapes**
 - Drawing shapes
- **Transformation**
 - 2D transformations of a coordinate system
- **Style Sheets**
 - How to make small customizations
 - How to apply a theme to a widget or application

Painting & Styling



- Painting on Widgets
- Color Handling
- Painting Operations
- Style Sheets



QPainter

- Paints on paint devices (QPaintDevice)
- QPaintDevice implemented by
 - On-Screen: QWidget
 - Off-Screen: QImage, QPixmap
 - And others ...
- Provides drawing functions
 - Lines, shapes, text or pixmaps
- Controls
 - Rendering quality
 - Clipping
 - Composition modes

Painting on Widgets

- Override `paintEvent(QPaintEvent*)`

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

- Schedule painting

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

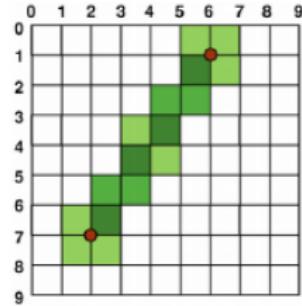
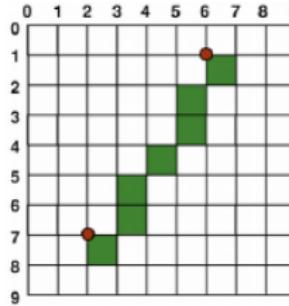
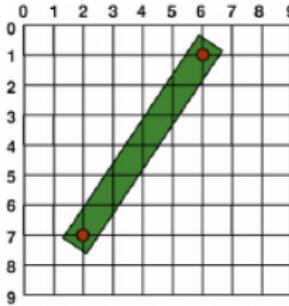
- Qt handles double-buffering

- To enable filling background:

- `QWidget::setAutoFillBackground (true)`

Coordinate System

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

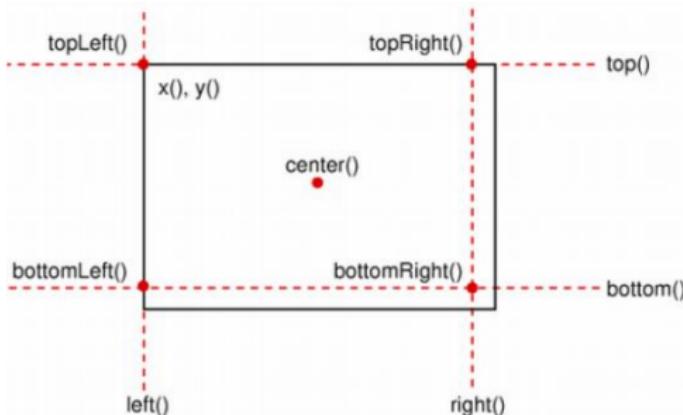


- Rendering quality switch
- QPainter::setRenderHint()

Geometry Classes

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

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



Painting & Styling

- Painting on Widgets
- Color Handling
- Painting Operations
- Style Sheets

Color Values

- Using different color models:

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

- Defining colors:

```
QColor(255,0,0); // red in RGB
QColor(255,0,0, 63); // red 25% opaque (75%
transparent)
```

```
QColor("#FF0000"); // red in web-notation
```

```
QColor("red"); // by svg-name
```

```
Qt::red; // predefined Qt global colors
```

- Many powerful helpers for manipulating colors

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

- QColor always refers to device color space

QPen

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

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

The Outline

Rule

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

- For a pen of width 1:

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

- *Due to integer rounding on a non-antialiased grid, the outline is shifted by 0.5 pixel towards the bottom right.*
- [Demo](#)

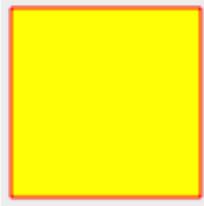
QBrush

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

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

- **Brush with solid red fill**

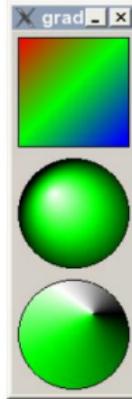
```
painter.setPen(Qt::red);
painter.setBrush(QBrush(Qt::yellow,
Qt::SolidPattern));
painter.drawRect(rect);
```



Gradient fills



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



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

- Demo

Brush on QPen

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



- [Demo](#)

Color Themes and Palettes



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



Painting & Styling

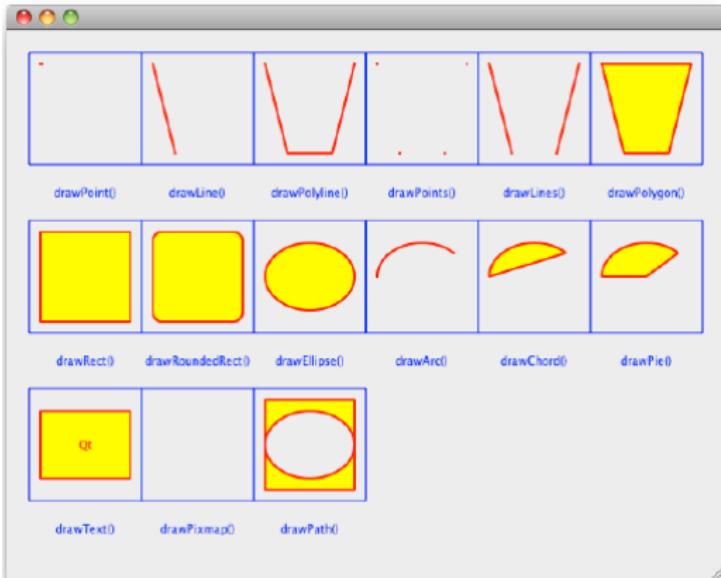


- Painting on Widgets
- Color Handling
- **Painting Operations**
- Style Sheets



Drawing Figures

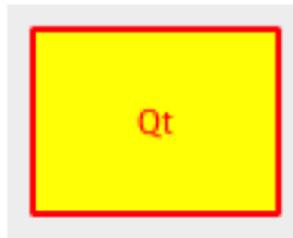
- Painter configuration
 - pen width: 2
 - pen color: red
 - font size: 10
 - brush color: yellow
 - brush style: solid
- Demo



Drawing Text

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

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



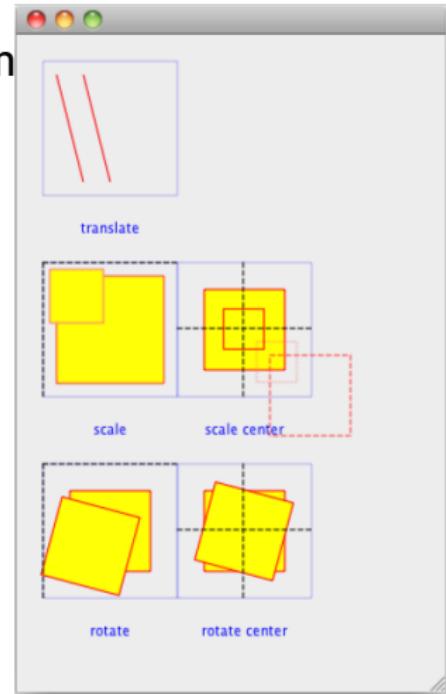
- **QFontMetrics**

- calculate size of strings

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

Transformation

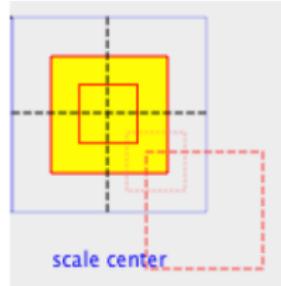
- Manipulating the coordinate system
 - $\text{translate}(x,y)$
 - $\text{scale}(sx,sy)$
 - $\text{rotate}(a)$
 - $\text{shear}(sh,$
- Demo



Transform and Center

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

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



QPainterPath

- Container for painting operations
- Enables reuse of shapes

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



- Path information

```
controlPointRect() - rect containing all points  
contains() - test if given shape is inside path  
intersects() - test given shape intersects path
```

- [Demo](#)

Hands-on

- Lab 7: Pie chart
 - [Objectives](#)
 - [Template code](#)

Painting & Styling

- Painting on Widgets
- Color Handling
- Painting Operations
- Style Sheets

Qt Style Sheets

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



CSS Rules

CSS Rule

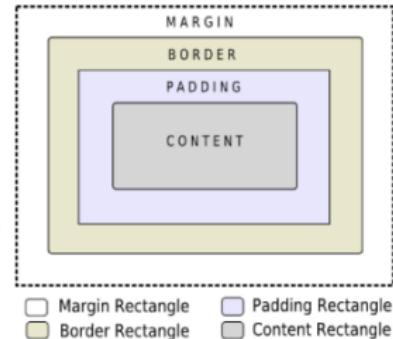
```
selector { property : value; property : value }
```

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

The Box Model

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

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



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

Selector Types

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

Selector Details

- Property Selector
 - If property changes it is required to re-set style sheet
- Combining Selectors
 - `QLineEdit, QComboBox, QPushButton { color: red }`
- Pseudo-States
 - Restrict selector based on widget's state
 - Example: `QPushButton:hover {color:red}`
- Demo
- Selecting Subcontrols
 - Access subcontrols of complex widgets
 - as `QComboBox, QSpinBox, ...`
 - `QComboBox::drop-down { image: url.dropdown.png }`
- Subcontrols positioned relative to other elements
 - Change using `subcontrol-origin` and `subcontrol-position`

Cascading

Effective style sheet obtained by merging

1. Widgets's ancestor (parent, grandparent, etc.)
 2. Application stylesheet
- On conflict: widget own style sheet preferred

```
qApp->setStyleSheet ("QPushButton { color:  
white }");  
  
button->setStyleSheet ("* { color: blue }");
```
 - Style on button forces button to have blue text
 - In spite of more specific application rule
 - [Demo](#)

Selector Specificity

- Conflict: When rules on same level specify same property
 - Specificity of selectors apply

```
QPushButton:hover { color: white }
QPushButton { color: red }
```
 - Selectors with pseudo-states are more specific
- Calculating selector's specificity
 - a Count number of ID attributes in selector
 - b Count number of property specifications
 - c Count number of class names
 - Concatenate numbers a-b-c. Highest score wins.
 - If rules scores equal, use last declared rule

```
QPushButton {} /* a=0 b=0 c=1 -> specificity = 1 */
QPushButton#ok {} /* a=1 b=0 c=1 -> specificity = 101 */
```
- Demo

Hands-on

- Try this [demo](#) code and
 - Investigate style sheet
 - Modify style sheet
 - Remove style sheet and implement your own

Application creation



Objectives

- Main Windows
- Settings
- Resources
- Deploying Qt Applications

Objectives

We will create an application to show fundamental concepts

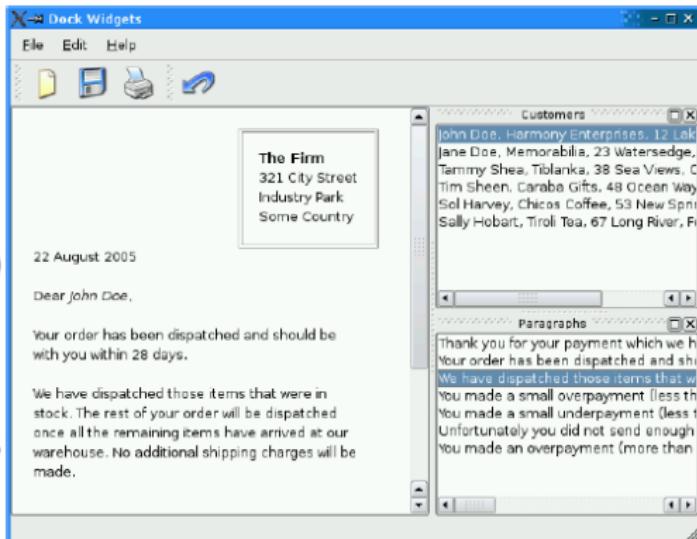
- **Main Window:** How a typical main window is structured
- **Settings:** Store/Restore application settings
- **Resources:** Adding icons and other files to your application
- **Deployment:** Distributing your application

Application creation

- Main Windows
- Settings
- Resources
- Deploying Qt Applications

Application Ingredients

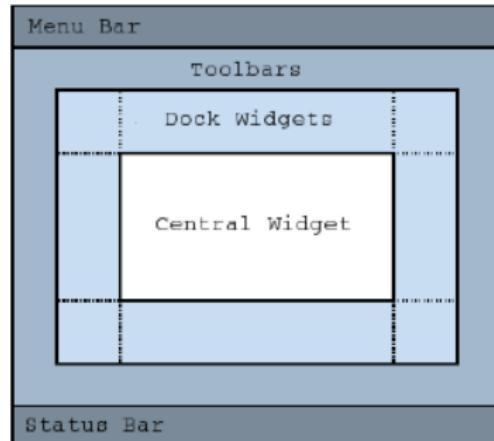
- Main window with
 - Menu bar
 - Tool bar, Status bar
 - Central widget
 - Often a dock window
- Settings (saving state)
- Resources (e.g icons)
- Translation
- Load/Save documents



Not a complete list

Main Window

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



QAction

Action is an abstract user interface command

- Emits signal triggered on execution
 - Connected slot performs action
- Added to menus, toolbar, key shortcuts
- Each performs same way
 - Regardless of user interface used

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

- [Qaction Documentation](#)

QAction capabilities

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

MenuBar

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

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



QToolBar

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



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

QToolButton

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

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

QStatusBar

Horizontal bar

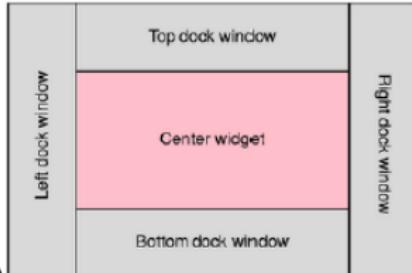
Suitable for presenting status information

- `showMessage(message, timeout)`
 - Displays temporary message for specified milli-seconds
- `clearMessage()`
 - Removes any temporary message
- `addWidget()` or `addPermanentWidget()`
 - Normal, permanent messages displayed by widget

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

QDockWidget

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



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

QMenu

and Context Menus

- Launch via event handler

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

- or signal `customContextMenuRequested()`
 - Connect to signal to show context menu
- Or via `QWidget::actions()` list
 - `QWidget::addAction(action)`
 - `setContextMenuPolicy(Qt::ActionsContextMenu)`
 - Displays `QWidget::actions()` as context menu

Hands-on

- Lab 8: Text editor
 - [Objectives](#)
 - [Template code](#)

Application creation

- Main Windows
- **Settings**
- Resources
- Deploying Qt Applications

QSettings

- Configure QSettings

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

- Typical usage

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

- Values are stored as QVariant

- Keys form hierarchies using '/'

- or use beginGroup(prefix) / endGroup()

- value() expects default value

- settings.value("group/value", 68).toInt()

- If value not found and default not specified

Invalid QVariant() returned

Restoring State

- Store geometry of application

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

- Restore geometry of application

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

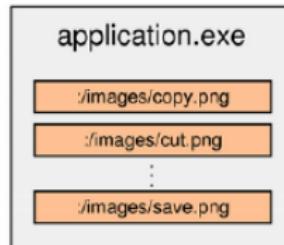
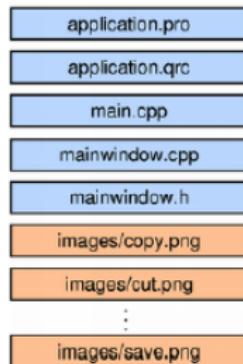
Application creation

- Main Windows
- Settings
- **Resources**
- Deploying Qt Applications

Resource System

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

- Docs



Using Resources

- Resources specified in .qrc file

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

- Can be created using QtCreator

- Resources are accessible with ':' prefix

- Example:(":/images/cut.png")
- Simply use resource path instead of file name
- QIcon(":/images/cut.png")

- To compile resource, edit .pro file

- RESOURCES += application.qrc
- qmake produces make rules to generate binary file

Hands-On

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

Application creation

- Main Windows
- Settings
- Resources
- Deploying Qt Applications

Ways of Deploying

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

Deployment

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

Dialogs module



Dialogs and Designer

- Dialogs
- Common Dialogs
- Qt Designer

Dialogs and Designer

- **Custom Dialogs**
 - Modality
 - Inheriting QDialog
 - Dialog buttons
- **Predefined Dialogs**
 - File, color, input and font dialogs
 - Message boxes
 - Progress dialogs
 - Wizard dialogs
- **Qt Designer**
 - Design UI Forms
 - Using forms in your code
 - Dynamic form loading

Dialogs and Designer

- Dialogs
- Common Dialogs
- Qt Designer

QDialog

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

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

Modeless Dialog

- Use `show()`

- Displays dialog
- Returns control to caller

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

Custom Dialogs

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

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

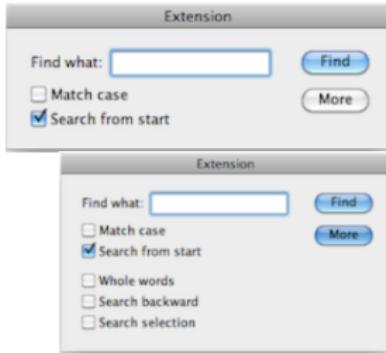
Deletion and Extension

- Deletion of dialogs
 - No need to keep dialogs around forever
 - Call `QObject::deleteLater()`
 - Or `setAttribute(Qt::WA_DeleteOnClose)`
 - Or override `closeEvent()`
- Dialogs with extensions:

- `QWidget::show()/hide()` used on extension

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

- Example



Dialogs and Designer

- Dialogs
- Common Dialogs
- Qt Designer

QFileDialog

- Allow users to select files or directories
- Asking for a file name

```
QString fileName =  
QFileDialog::getOpenFileName(this, tr("Open File"));  
if (!fileName.isNull()) {  
// do something useful  
}
```

- QFileDialog::getOpenFileNames()
 - Returns one or more selected existing files
- QFileDialog::getSaveFileName()
 - Returns a file name. File does not have to exist.
- QFileDialog::getExistingDirectory()
 - Returns an existing directory.
- setFilter("Image Files (*.png *.jpg *.bmp)")
 - Displays files matching the patterns

QMessageBox

- Provides a modal dialog for ...
 - informing the user
 - asking a question and receiving an answer
- Typical usage, questioning a user

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

- Very flexible in appearance
 - [Reference documentation](#)
- Other convenience methods
 - QMessageBox::information(...)
 - QMessageBox::warning(...)
 - QMessageBox::critical(...)
 - QMessageBox::about(...)

QProgressDialog

- Provides feedback on the progress of a slow operation

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

- Initialize with setValue(0)

- Otherwise estimation of duration will not work

- When operation progresses, check for cancel

- QProgressDialog::wasCanceled()
- Or connect to QProgressDialog::canceled()

- To stay reactive call QApplication::processEvents()

- See [Documentation](#)

QErrorMessage

- Similar to QMessageBox with checkbox
- Asks if message shall be displayed again

```
m_error = new QErrorMessage(this);  
m_error->showMessage(message, type);
```
- Messages will be queued
- QErrorMessage::qtHandler()
 - installs an error handler for debugging
 - Shows qDebug(), qWarning() and qFatal() messages in QErrorMessage box

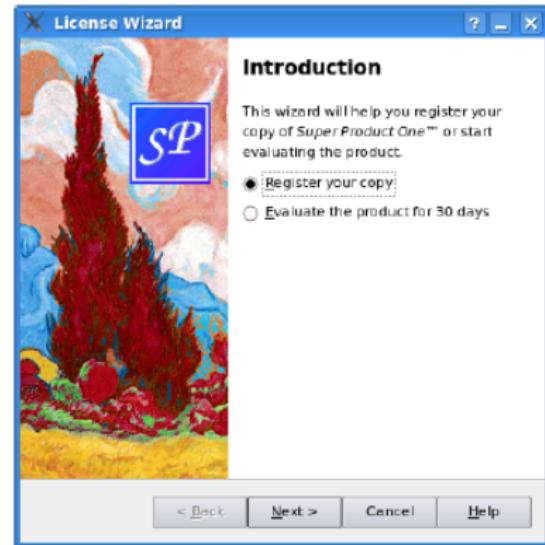
Other Common Dialogs

- Asking for Input - QInputDialog
 - `QInputDialog::getText(...)`
 - `QInputDialog::getInt(...)`
 - `QInputDialog::getDouble(...)`
 - `QInputDialog::getItem(...)`
- Selecting Color - QColorDialog
 - `QColorDialog::getColor(...)`
- Selecting Font - QFontDialog
 - `QFontDialog::getFont(...)`
- [Example](#)

Qwizard

Guiding the user

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



Hands-on

- Lab 9: Dialog
 - [Objectives](#)
 - [Template code](#)

Summary

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

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

Dialogs and Designer

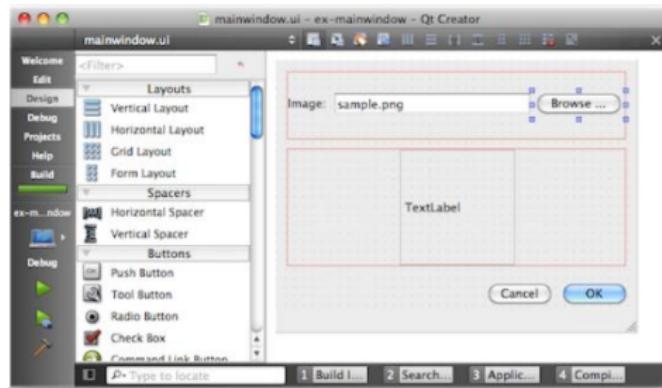


- Dialogs
- Common Dialogs
- Qt Designer



Qt Designer

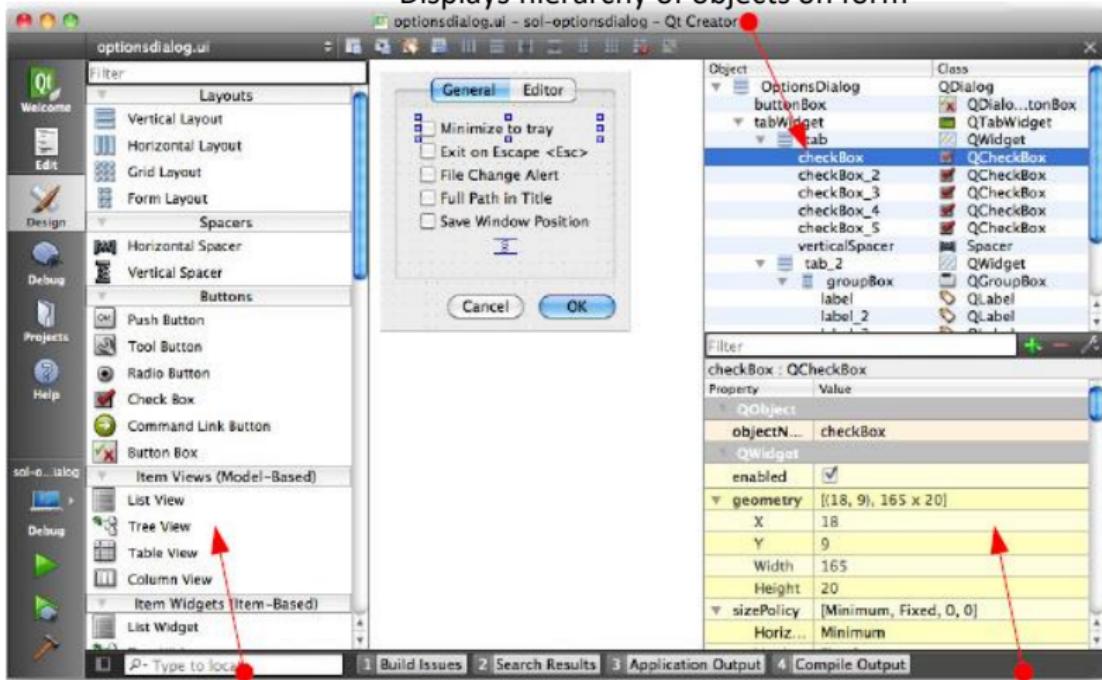
- Design UI forms visually
- Visual Editor for
 - Signal/slot connections
 - Actions
 - Tab handling
 - Buddy widgets
 - Widget properties
 - Integration of custom widgets
 - Resource files



Designer Views

Object Inspector

Displays hierarchy of objects on form



Widget Box

Provides selection of widgets, layouts

Property Editor

Displays properties of selected object

Editing Modes

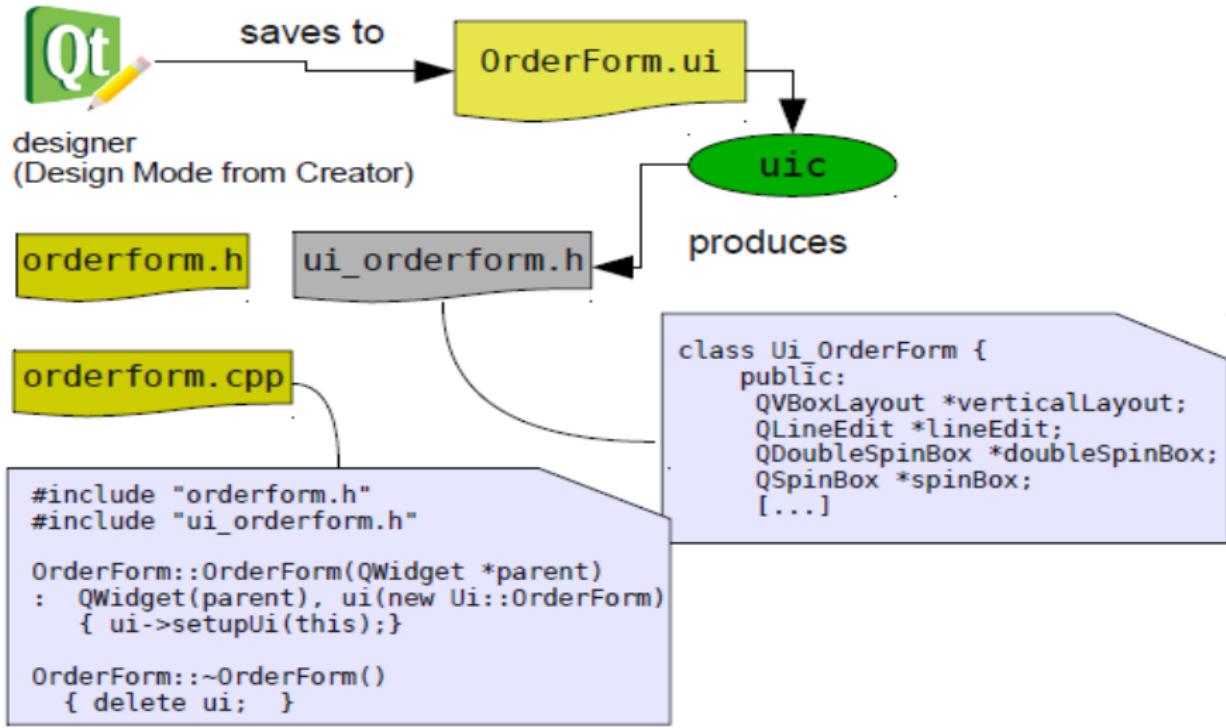
- Widget Editing
 - Change appearance of form
 - Add layouts
 - Edit properties of widgets
- Signal and Slots Editing
 - Connect widgets together with signals & slots
- Buddy Editing
 - Assign buddy widgets to label
 - *Buddy widgets help keyboard focus handling correctly*
- Tab Order Editing
 - Set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

UI Form Files

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

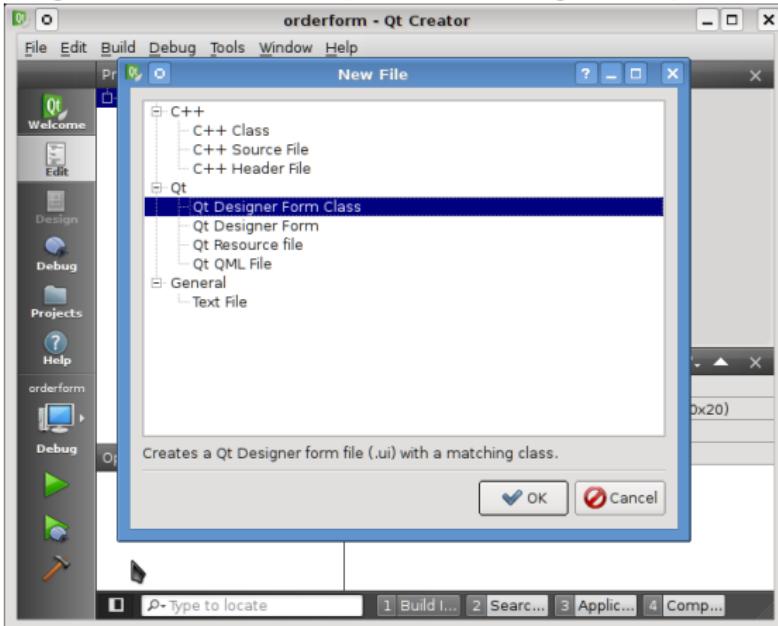
```
// ui_mainwindow.h
class Ui_MainWindow {
public:
    QLineEdit *fileName;
    ... // simplified code
    void setupUi(QWidget *) { /* setup widgets */ }
};
```
- Form ui file in project (.pro)
FORMS += mainwindow.ui

From .ui to C++



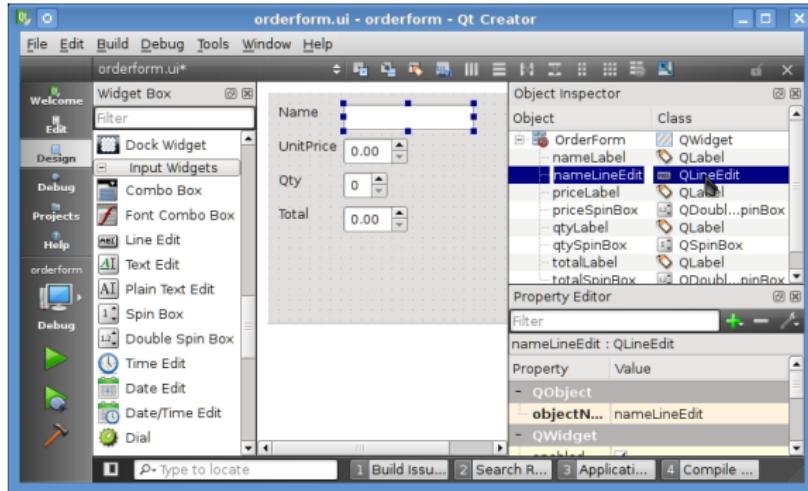
Form Wizards

- Add New... "Designer Form"
 - or "Designer Form Class" (for C++ integration)



Naming Widgets

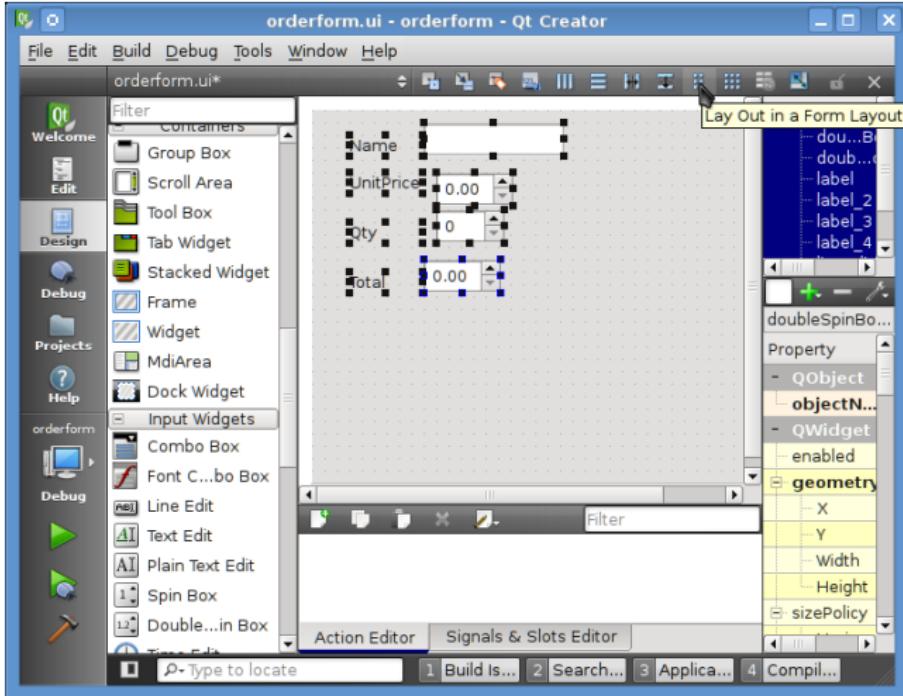
1. Place widgets on form
2. Edit objectName property



- *objectName defines member name in generated code*

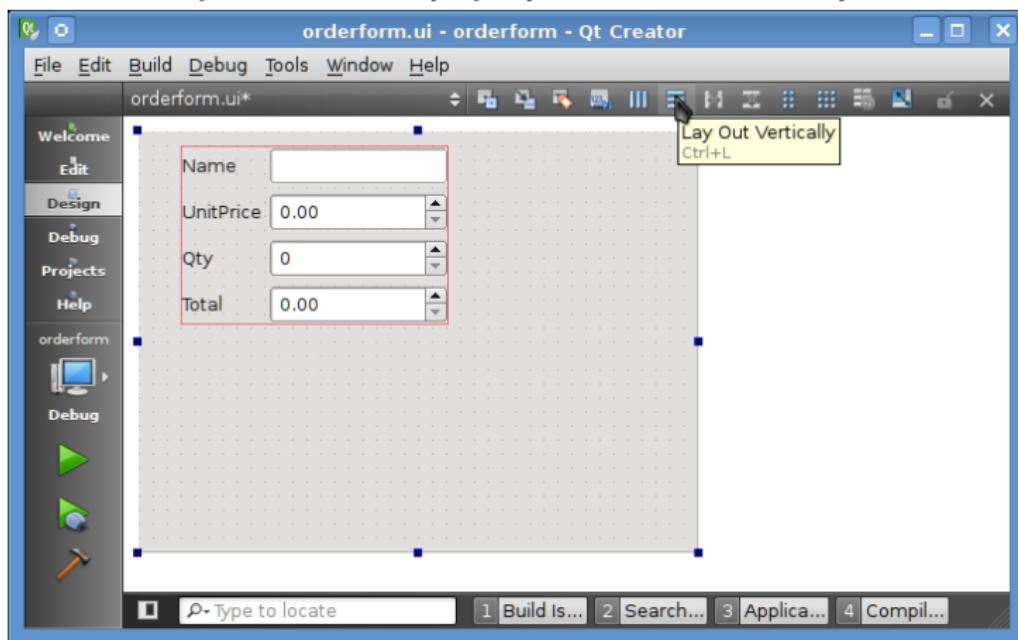
Form layout

QFormLayout: Suitable for most input forms



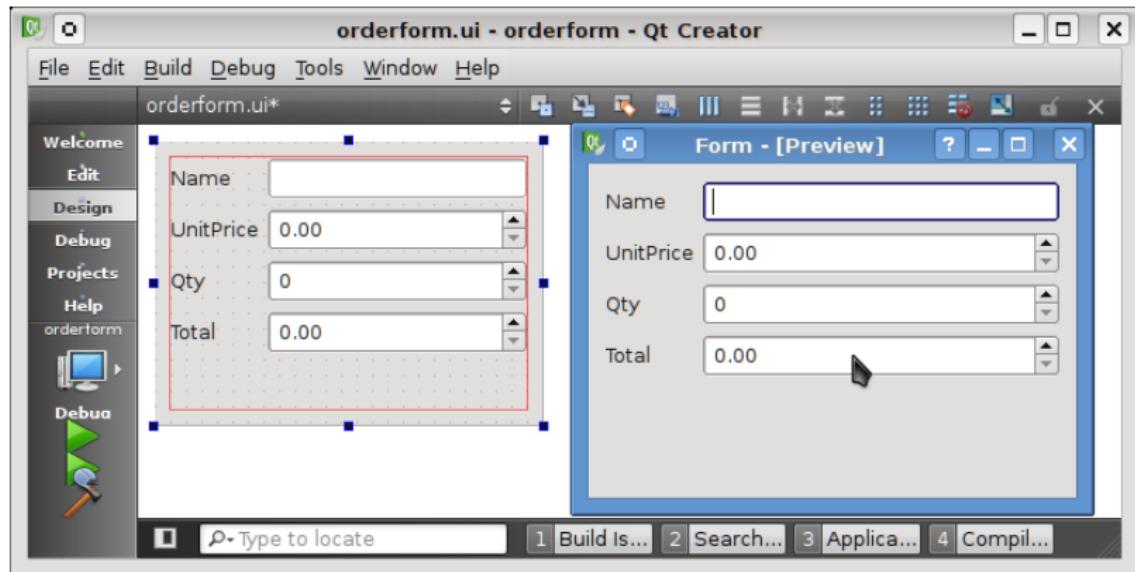
Top-Level Layout

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



Preview Mode

Check that widget is nicely resizable



Code Integration

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

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

Code Integration

```
// orderform.cpp
#include "ui_orderform.h"
OrderForm::OrderForm(QWidget *parent)
: QDialog(parent), ui(new Ui_OrderForm) {
    ui->setupUi(this);
}
OrderForm::~OrderForm() {
    delete ui; ui=0;
}
```

- *Default behavior in Qt Creator*

Signals and Slots

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

Loading .ui files

- Forms can be processed at runtime
 - Produces dynamically generated user interfaces
- Disadvantages
 - Slower, harder to maintain
 - Risk: .ui file not available at runtime
- Loading .ui file

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

- Locate objects in form

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

Hands-on

- Lab 10: Order form
 - [Objectives](#)
 - Template code

Model/View modules



Objectives

- Model/View Concept
- Custom Models
- Delegates
- Editing item data
- Data Widget Mapper
- Drag and Drop
- Custom Tree Model

Objectives

Using Model/View

- Introducing to the concepts of model-view
- Showing Data using standard item models

Custom Models

- Writing a simple read-only custom model.
- Editable Models
- Custom Delegates
- Using Data Widget Mapper
- Custom Proxy Models
- Drag and Drop

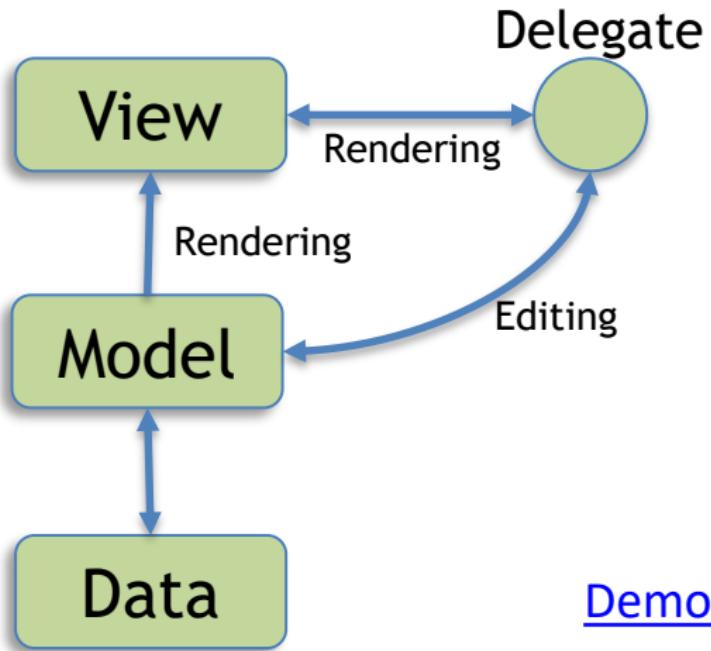
Model/View

- Model/View Concept
- Custom Models
- Delegates
- Editing item data
- Data Widget Mapper
- Drag and Drop
- Custom Tree Model

Why Model/View?

- **Isolated domain-logic**
 - From input and presentation
- **Makes Components Independent**
 - For Development
 - For Testing
 - For Maintenance
- **Foster Component Reuse**
 - Reuse of Presentation Logic
 - Reuse of Domain Model

Model/View Components



[Demo](#)

Model Structures

List Model

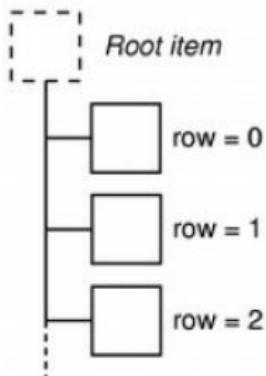
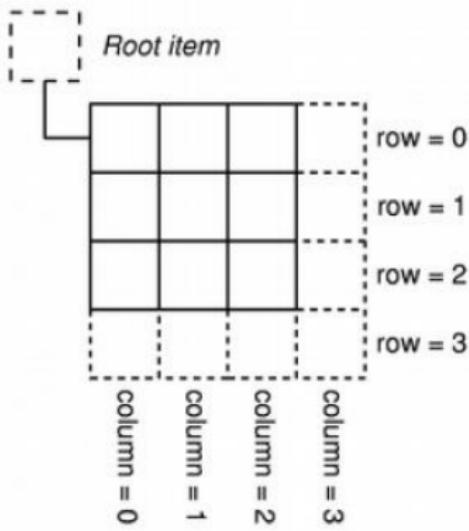
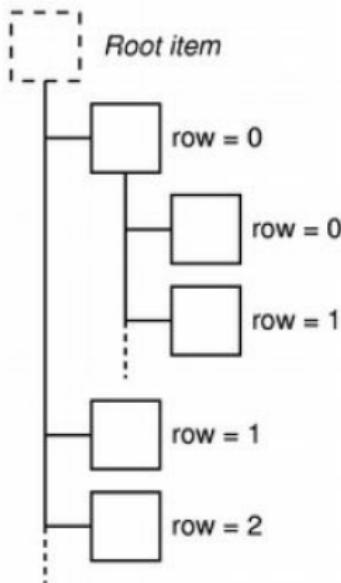


Table Model



Tree Model

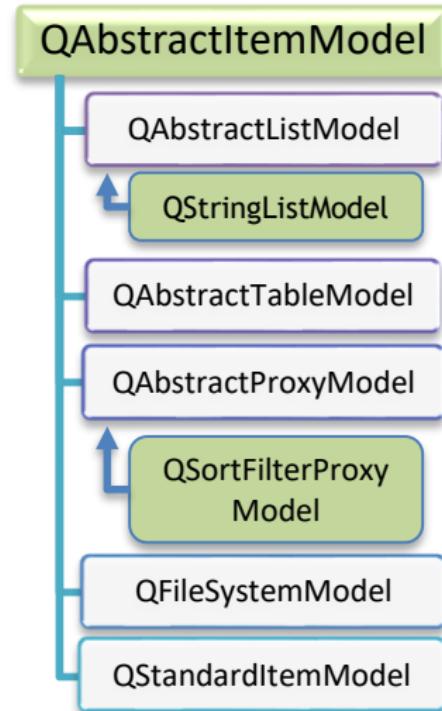


View Classes

- **QtQuick ItemView**
 - Abstract base class for scrollable views
- **QtQuick ListView**
 - Items of data in a list
- **QtQuick GridView**
 - Items of data in a grid
- **QtQuick PathView**
 - Items of data along a specified path

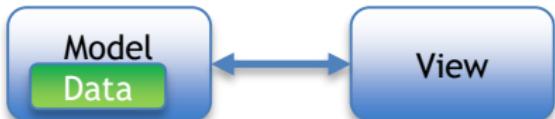
Model Classes

- **QAbstractItemModel**
 - Abstract interface of models
- **Abstract Item Models**
 - Implement to use
- **Ready-Made Models**
 - Convenient to use
- **Proxy Models**
 - Reorder/filter/sort your items
- [Model class documentation](#)



Data-Model-View Relationships

- **Standard Item Model**
 - Data+Model combined
 - View is separated
 - Model is your data
- **Custom Item Models**
 - Model is adapter to data
 - View is separated



QModelIndex

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

Table/Tree

- **Rows and columns**

- Item location in table model
 - Item has no parent (`parent.isValid() == false`)

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

- **Parents, rows, and columns**

- Item location in tree model

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

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

Item and Item Roles

- **Item performs various roles**
 - for other components (delegate, view, ...)
- **Supplies different data**
 - for different situations
- **Example:**
 - Qt::DisplayRole used displayed string in view
- **Asking for data**

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

Showing simple Data

QStandardItemModel - Convenient Model

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

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

Demo

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

Proxy Model

- **QSortFilterProxyModel**
 - Transforms structure of source model
 - Maps indexes to new indexes

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

Note: Need to load all data to sort or filter

Sorting/Filtering

- **Filter with Proxy Model**

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

- **Sorting with Proxy Model**

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

- **Filter via TextInput signal**

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

- [Demo](#)

Summary

- **Model Structures**
 - List, Table and Tree
- **Components**
 - Model - Adapter to Data
 - View - Displays Structure
 - Delegate - Paints Item
 - Index - Location in Model
- **Views**
 - ListView
 - GridView
 - PathView
- **Models**
 - QAbstractItemModel
 - Other Abstract Models
 - Ready-Made Models
 - Proxy Models
- **Index**
 - `row()`, `column()`, `parent()`
 - `data(role)`
 - `model()`
- **Item Role**
 - `Qt::DisplayRole`
 - Standard Roles in `Qt::ItemDataRoles`

Model/View

- Model/View Concept
- **Custom Models**
- Delegates
- Editing item data
- Data Widget Mapper
- Drag and Drop
- Custom Tree Model

Implementing a Model

- Variety of classes to choose from
 - **QAbstractListModel**
 - One dimensional list
 - **QAbstractTableModel**
 - Two-dimensional tables
 - **QAbstractItemModel**
 - Generic model class
 - **QStringListModel**
 - One-dimensional model
 - Works on string list
 - **QStandardItemModel**
 - Model that stores the data
- **Notice:** Need to subclass *abstract* models

Step 1:

Read Only List Model

```
class MyModel: public QAbstractListModel {  
public:  
    // return row count for given parent  
    int rowCount( const QModelIndex &parent) const;  
    // return data, based on current index and  
    // requested role  
    QVariant data( const QModelIndex &index,  
    int role = Qt::DisplayRole) const;  
};
```

[Demo](#)

Step 2:

Header Information

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

[Demo](#)

Step 3: Enabling Editing

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

[Demo](#)

Step 4: Row Manipulation

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

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

[Demo](#)

Hands-on

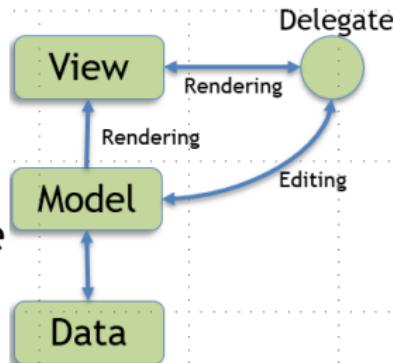
- Lab 11: City list model
 - Objectives
 - Template code

Model/View

- Model/View Concept
- Custom Models
- **Delegates**
- Editing item data
- Data Widget Mapper
- Drag and Drop
- Custom Tree Model

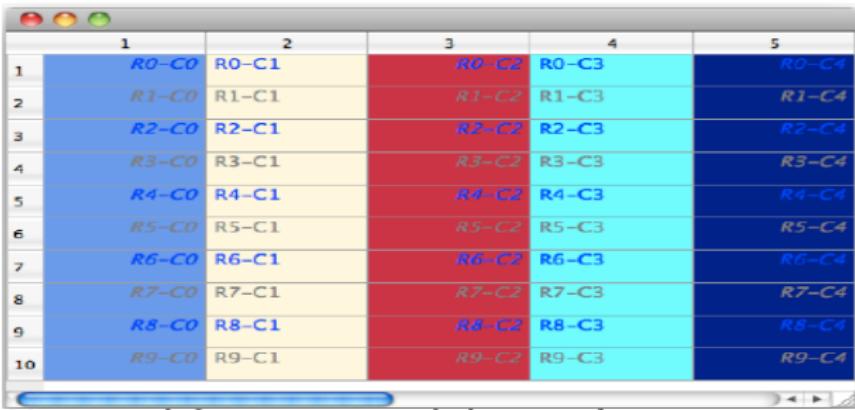
Item Delegates

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



Item Appearance

*Data table
shown has no custom delegate*



1	2	3	4	5
R0-C0	R0-C1	R0-C2	R0-C3	R0-C4
R1-C0	R1-C1	R1-C2	R1-C3	R1-C4
R2-C0	R2-C1	R2-C2	R2-C3	R2-C4
R3-C0	R3-C1	R3-C2	R3-C3	R3-C4
R4-C0	R4-C1	R4-C2	R4-C3	R4-C4
R5-C0	R5-C1	R5-C2	R5-C3	R5-C4
R6-C0	R6-C1	R6-C2	R6-C3	R6-C4
R7-C0	R7-C1	R7-C2	R7-C3	R7-C4
R8-C0	R8-C1	R8-C2	R8-C3	R8-C4
R9-C0	R9-C1	R9-C2	R9-C3	R9-C4

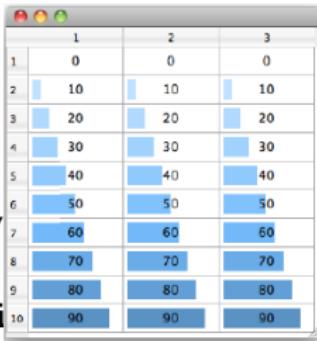
- No need for custom delegate!
- Use `Qt::ItemRole` to customize appearance

QAbstractItemDelegate

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

[Demo](#)

[Documentation](#)



Model/View

- Model/View Concept
- Custom Models
- Delegates
- **Editing item data**
- Data Widget Mapper
- Drag and Drop
- Custom Tree Model

Editor Delegate

- Provides QComboBox
 - for editing a series of values

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

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

Creating Editor

- Create editor by index

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

- Set data to editor

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

Data to the model

- When user finished editing

- view asks delegate to store data into model

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

- If editor has finished editing

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

Editor's geometry

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

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

- [Demo](#)
- Case of multi-index editor
 - Position editor in relation to indexes

Setting Delegates

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

Type Based Delegates

[Demo](#)

Model/View

- Model/View Concept
- Custom Models
- Delegates
- Editing item data
- **Data Widget Mapper**
- Drag and Drop
- Custom Tree Model

QDataWidgetMapper

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

	Name	Address	Age
1	Alice	123 Main Street Market Town	20
2	Bob	PO Box 32 Mail Handling S...	31
3	Carol	The Lighthouse Remote Isl...	32
4	Donald	4733B Park Avenue Big City	19
5	Emma	Research Station Base Cam...	26

Mapping

Simple Widget Mapper

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

QDataWidgetMapper

- **Mapping Setup**

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

- **Track Navigation**

```
connect(nextButton, SIGNAL(clicked()),
        mapper, SLOT(toNext()));
connect(previousButton, SIGNAL(clicked()),
        mapper, SLOT(toPrevious()));
```

[Demo](#)

Mapped Property

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

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

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

Demo

Model/View

- Model/View Concept
- Custom Models
- Delegates
- Editing item data
- Data Widget Mapper
- **Drag and Drop**
- Custom Tree Model

Drag and Drop for Views

- Enable the View

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

- Model has to provide support for drag and drop operations

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

- Model needs to support actions

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

QStandardItemModel

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

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

Demo

QAbstractItemModel

```
class MyModel : public QAbstractItemModel {  
public:  
    // actions supported by the data in this model  
    Qt::DropActions supportedDropActions() const;  
    // for supported index return Qt::ItemIs(Drag|Drop)Enabled  
    Qt::ItemFlags flags(const QModelIndex &index) const;  
    // returns list of MIME types that are supported  
    QStringList QAbstractItemModel::mimeTypes() const;  
    // returns object with serialized data in mime formats  
    QMimeData *mimeData(const QModelIndexList &indexes) const;  
    // true if data and action can be handled, otherwise false  
    bool dropMimeData(const QMimeData *data, Qt::DropAction  
                      action,  
                      int row, int column, const QModelIndex &parent);  
};
```

Demo

Model/View

- Model/View Concept
- Custom Models
- Delegates
- Editing item data
- Data Widget Mapper
- Drag and Drop
- Custom Tree Model

A Custom Tree Model in 5 Steps

1. Read-OnlyModel
2. EditableModel
3. Insert-RemoveModel
4. LazyModel
5. Drag and DropModel

A Node Structure

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

[Demo](#) (node.h)

Read-Only Model

```
class ReadOnlyModel : public QAbstractItemModel {  
public:  
    ...  
    QModelIndex index( row, column, parent ) const;  
    QModelIndex parent child ) const;  
    int rowCount( parent ) const;  
    int columnCount( parent ) const;  
    QVariant data( index, role) const;  
protected: // important helper methods  
    QModelIndex indexForNode(Node *node) const;  
    Node* nodeForIndex(const QModelIndex &index)  
        const;  
    int rowForNode(Node *node) const;  
};
```

Editable Model

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

Insert/Remove Model

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

Lazy Model

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

DnD Model

```
class DndModel : public InsertRemoveModel {  
public:  
    ...  
    Qt::ItemFlags flags( index ) const;  
    Qt::DropActions supportedDragActions() const;  
    Qt::DropActions supportedDropActions() const;  
    QStringList mimeTypes() const;  
    QMimeData *mimeData( indexes ) const;  
    bool dropMimeData(data, dropAction, row, column,  
                      parent);  
    bool removeRows(row, count, parent);  
    bool insertRows(row, count, parent);  
};
```

Graphics View



Objectives

- Using GraphicsView Classes
- Coordinate Systems and Transformations
- Widgets in a Scene
- Drag and Drop
- Effects
- Performance Tuning

Objectives

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

Graphics View

- Using GraphicsView Classes
- Coordinate Systems and Transformations
- Widgets in a Scene
- Drag and Drop
- Effects
- Performance Tuning

GraphicsView Framework

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

Hello World

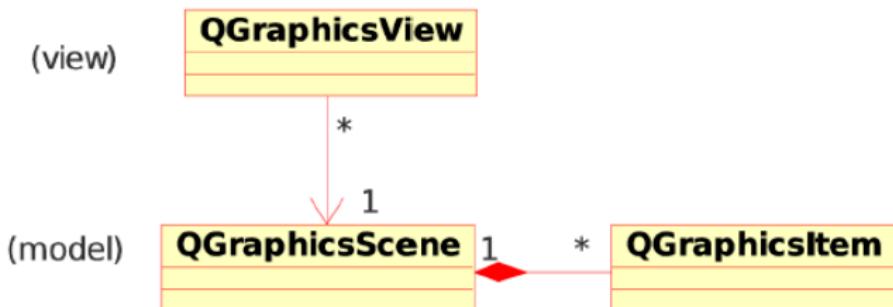
```
#include <QtWidgets>
int main(int argc, char **argv) {
    QApplication app(argc, argv);
    QGraphicsView view;
    QGraphicsScene *scene = new QGraphicsScene(&view);
    view.setScene(scene);
    QGraphicsRectItem *rect =
        new QGraphicsRectItem(-10, -10, 120, 50);
    scene->addItem(rect);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

Demo



UML relationship

- QGraphicsScene is:
 - a "model" for QGraphicsView
 - a "container" for QGraphicsItems



QGraphicsScene

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

QGraphicsScene

- `addItem()`
 - Add an item to the scene
 - (remove from previous scene if necessary)
 - Also `addEllipse()`, `addPolygon()`, `addText()`, etc
- `QGraphicsEllipseItem *ellipse =`
`scene->addEllipse(-10, -10, 120, 50);`
- `QGraphicsTextItem *text =`
`scene->addText("Hello World!");`
- `items()`
 - returns items intersecting a particular point or region
- `selectedItems()`
 - returns list of selected items
- `sceneRect()`
 - bounding rectangle for the entire scene

QGraphicsView

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

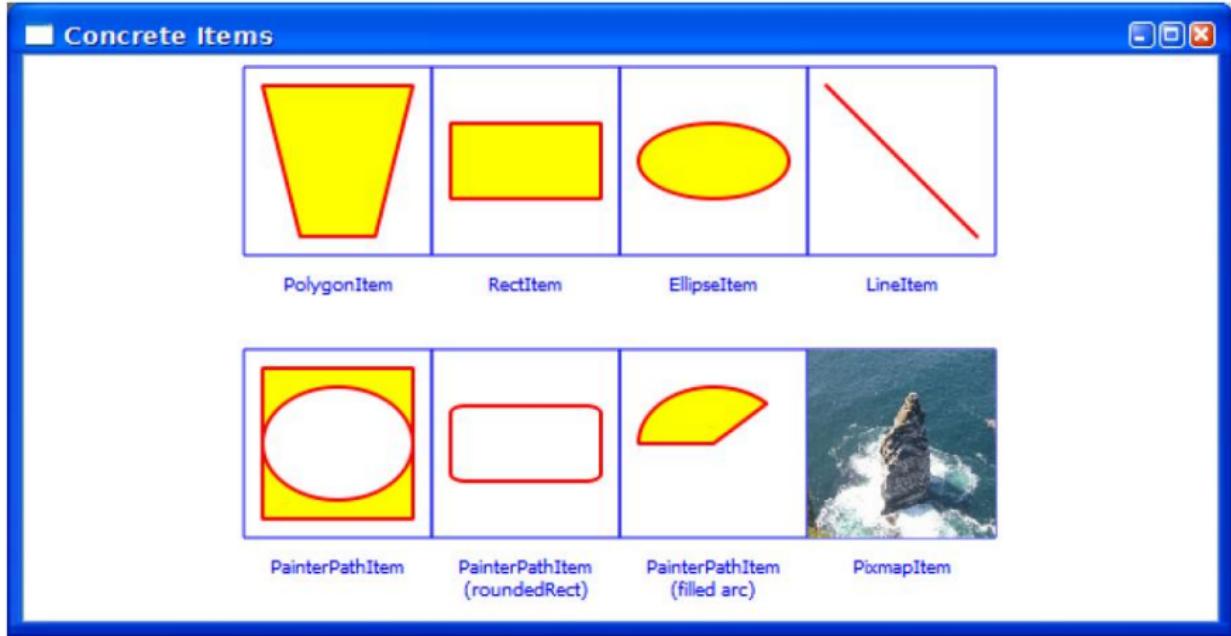
QGraphicsView

- `setScene()`
 - sets the QGraphicsScene to use
- `setRenderHints()`
 - antialiasing, smooth pixmap transformations, etc
- `centerOn()`
 - takes a QPoint or a QGraphicsItem as argument
 - ensures point/item is centered in View
- `mapFromScene()`, `mapToScene()`
 - map to/from scene coordinates
- `scale()`, `rotate()`, `translate()`, `matrix()`
 - transformations

QGraphicsItem

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

QGraphicsItem Types



Demo

QGraphicsItem methods

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

Select, Focus, Move

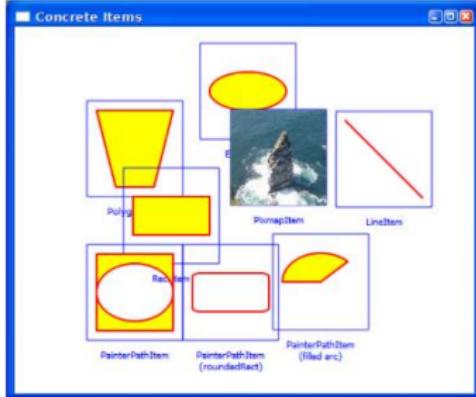
- `QGraphicsItem::setFlags()`
 - Determines which operations are supported on an item
- `QGraphicsItemFlags`
 - `QGraphicsItem::ItemIsMovable`
 - `QGraphicsItem::ItemIsSelectable`
 - `QGraphicsItem::ItemIsFocusable`

```
item->setFlags (
```

```
QGraphicsItem::ItemIsMovable | QGraphicsItem::ItemIsSelectable);
```

Groups of Items

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



Parents and Children

- Parent propagates values to child items:
 - `setEnabled()`
 - `setFlags()`
 - `setPos()`
 - `setOpacity()`
 - etc...
- Enables composition of items.

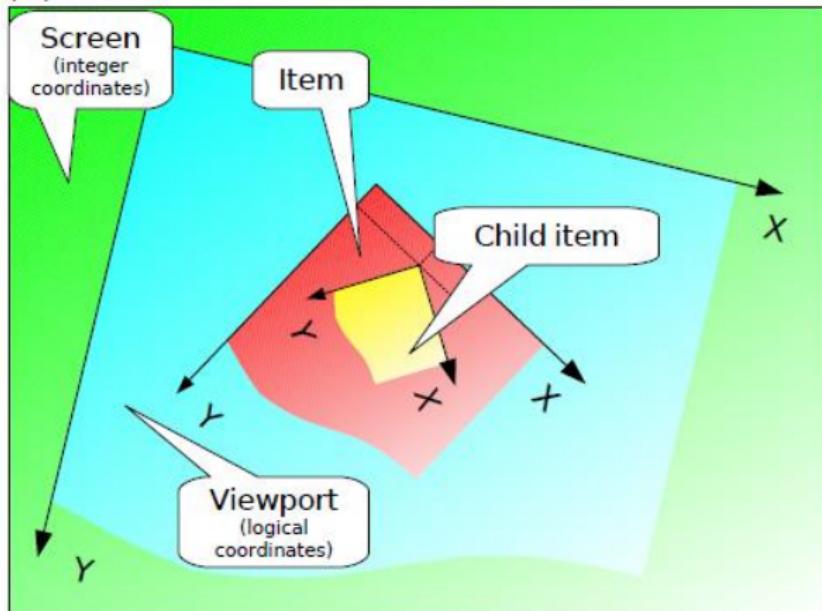
Graphics View

- Using GraphicsView Classes
- Coordinate Systems and Transformations
- Widgets in a Scene
- Drag and Drop
- Effects
- Performance Tuning

Coordinate Systems

Each View and Item has its own local coordinate system

(0,0)



Coordinates

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

QTransform

Coordinate systems can be transformed using QTransform

QTransform is a 3x3 matrix describing a linear transformation from (x,y) to (xt, yt)

m11	m12	m13
m21	m22	m23
m31	m32	m33

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

- m13 and m23
 - Control perspective transformations
- [Documentation](#)

Common Transformations

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

View transformations

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

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

Item Transformations

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

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

Zooming

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

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

Ignoring Transformations

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

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

Demo

Graphics View

- Using GraphicsView Classes
- Coordinate Systems and Transformations
- **Widgets in a Scene**
- Drag and Drop
- Effects
- Performance Tuning

Widgets in a Scene



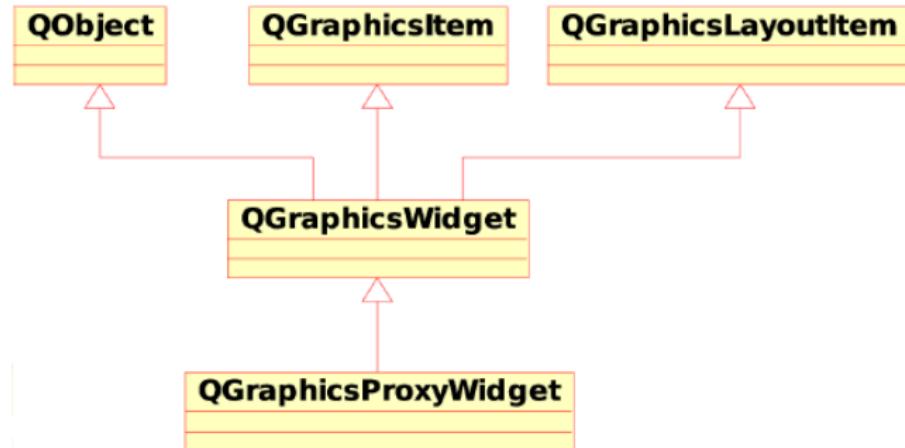
[Demo](#)

Items not widgets

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

QGraphicsWidget

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



QGraphicsProxyWidget

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

Embedded Widget

```
#include <QtWidgets>
int main(int argc, char **argv) {
    QApplication app(argc, argv);
    QCalendarWidget *calendar = new
    QCalendarWidget;
    QGraphicsScene scene;
    QGraphicsProxyWidget *proxy =
    scene.addWidget(calendar);
    QGraphicsView view(&scene);
    view.show();
    return app.exec();
}
```

QGraphicsLayout

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

Graphics View

- Using GraphicsView Classes
- Coordinate Systems and Transformations
- Widgets in a Scene
- **Drag and Drop**
- Effects
- Performance Tuning

Drag and Drop

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

Start Drag

Starting an item drag is similar to dragging from a QWidget.

- Override event handlers:
 - `mousePressEvent()`
 - `mouseMoveEvent()`
- In `mouseMoveEvent()`, decide if drag started? if so:
 - Create a `QDrag` instance
 - Attach a `QMimeType` to it
 - Call `QDrag::exec()`
 - Function returns when user drops
 - Does not block event loop
- [Demo](#)

Drop on a scene

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

Hands-on

Graphics View

- Using GraphicsView Classes
- Coordinate Systems and Transformations
- Widgets in a Scene
- Drag and Drop
- **Effects**
- Performance Tuning

Graphics Effects

Effects can be applied to graphics items:

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



Graphics Effects

- Applying a blur effect to a pixmap.

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



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

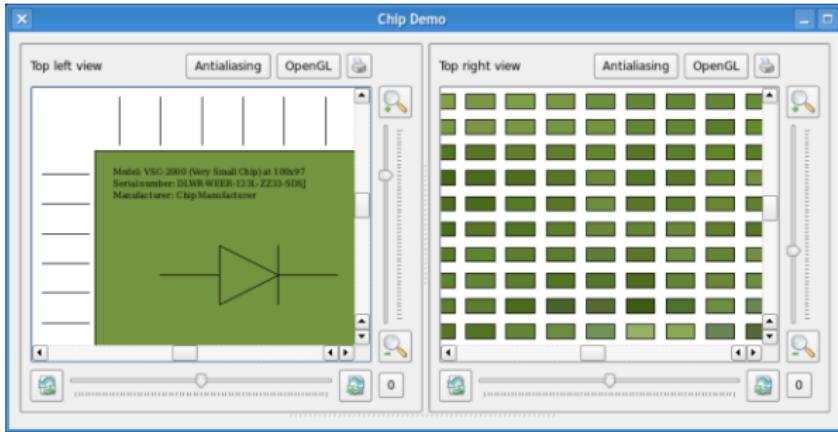
Graphics View

- Using GraphicsView Classes
- Coordinate Systems and Transformations
- Widgets in a Scene
- Drag and Drop
- Effects
- Performance Tuning

Level of Detail

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

Examples



Demo

Caching tips

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

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

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

setCacheMode()

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

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

Tweaking

The following methods allow you to tweak performance of view/scene/items:

- `QGraphicsView::setViewportUpdateMode()`
- `QGraphicsView::setOptimizationFlags()`
- `QGraphicsScene::setItemIndexMethod()`
- `QGraphicsScene::setBspTreeDepth()`
- `QGraphicsItem::setFlags()`
 - `ItemDoesntPropagateOpacityToChildren` and
`ItemIgnoresParentOpacity` especially recommended if your items are opaque!

Tips for better performance

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

Hand-on

Introduction to QML



What is QML

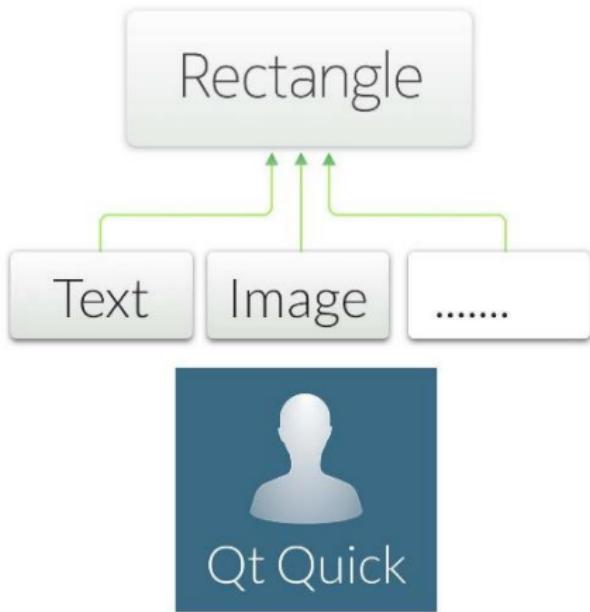
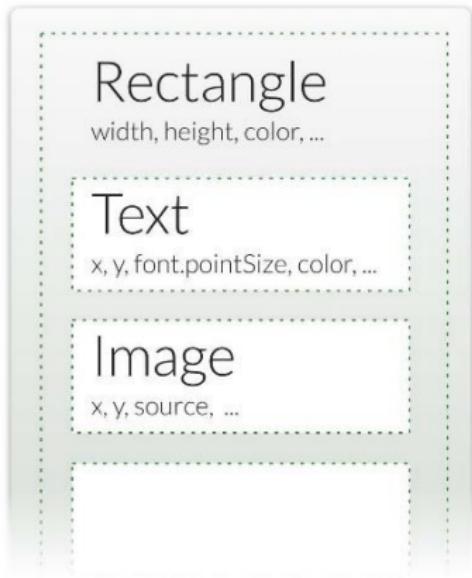
Declarative language for User Interface elements:

- Describes the user interface
 - What elements look like
 - How elements behave
- UI specified as tree of elements with properties

Elements

- Elements are structures in the markup language
 - Represent visual and non-visual parts
- Item is the base type of visual elements
 - Not visible itself
 - Has a position, dimensions
 - Usually used to group visual elements
 - Rectangle, Text, TextInput,...
- Non-visual elements:
 - States, transitions,...
 - Models, paths,...
 - Gradients, timers, etc.
- Elements contain properties
 - Can also be extended with custom properties
- [QML Elements](#)

Tree of elements



Properties

Elements are described by properties:

- Simple name-value definitions
 - width, height, color,...
 - With default values
 - Each has a well-defined type
 - Separated by semicolons or line breaks
- Used for
 - Identifying elements (id property)
 - Customizing their appearance
 - Changing their behavior
- Demo

THANK YOU