8th Advanced School on SCIENTIFIC VISUALIZATION

Advanced GUI development using Qt

Paolo Quadrani - p.quadrani@cineca.it
Andrea Negri - a.negri@cineca.it

SuperComputing Applications and Innovation Department

CINECA

Agenda

- The Story of Qt
- Developing a Hello World Application
- Qt SDK
- Hands-On & Examples

Qt brief timeline

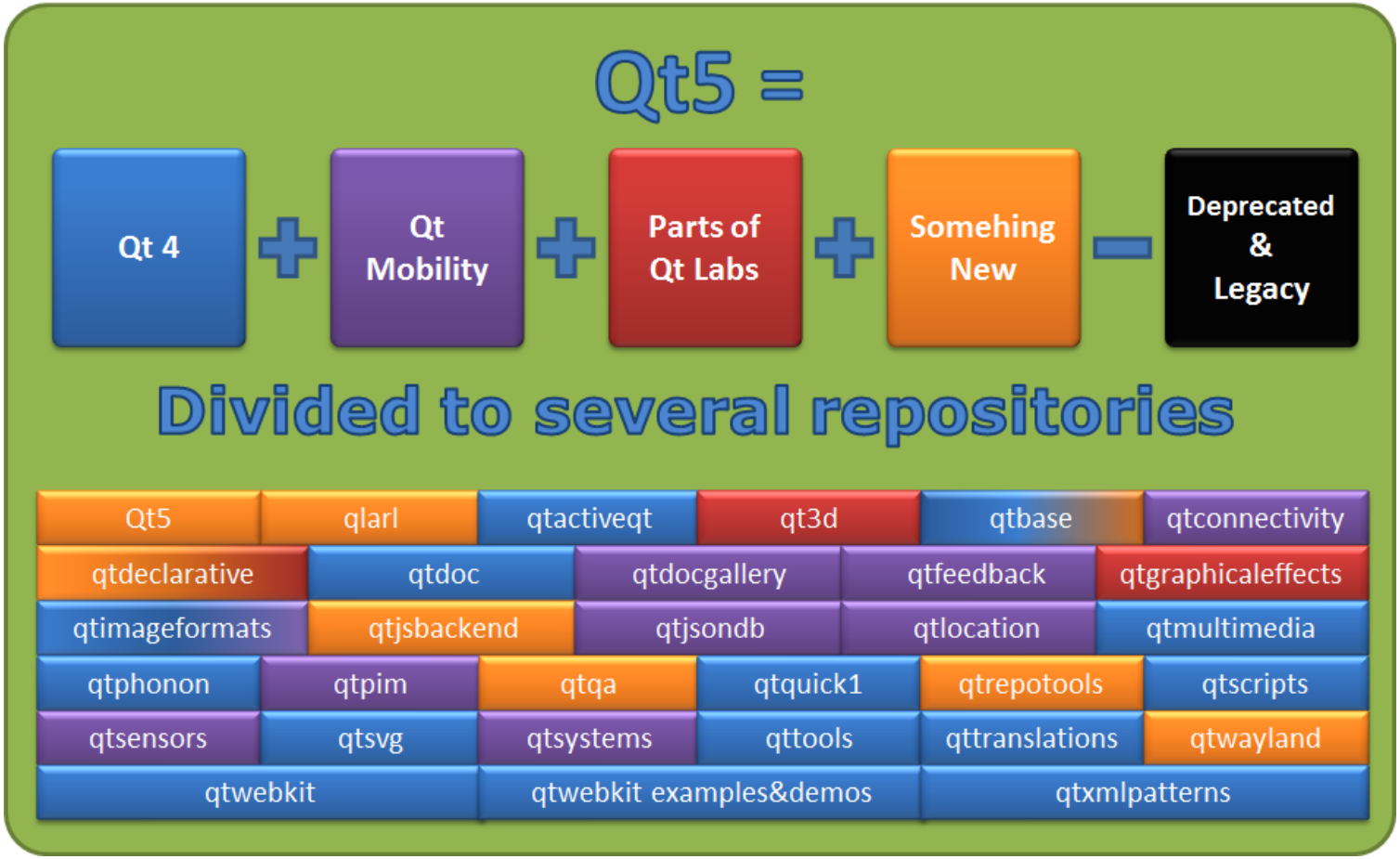
- Qt Development Frameworks founded in 1994
- Trolltech acquired by Nokia in 2008
- Qt Commercial business acquired by Digia in 2011
- Qt business acquired by Digia from Nokia in 2012
- Trusted by over 6,500 companies worldwide

- 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

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

Qt5 modules


Qt5 =
Qt 4
+
Qt Mobility
+
Parts of Qt Labs
+
Somehing New
−
Deprecated & Legacy
Divided to several repositories
Qt5
qlarl
qtactiveqt
qt3d
qtbase
qtconnectivity
qtdeclarative
qtdoc
qtdocgallery
qtfeedback
qtgraphicaleffects
qtimageformats
qtjsbackend
qtjsondb
qtlocation
qtmultimedia
qtphonon
qtpim
qtqa
qtquick1
qtrepotools
qtscripts
qtsensors
qtsvg
qtsystems
qttools
qttranslations
qtwayland
qtwebkit
qtwebkit examples&demos
qtxmlpatterns

“Hello world” in Qt /1 - Project files

Program consists of

- *main.cpp* - application code
- *helloworld.pro* - project file
  - lists source and header files
  - provides project configuration

“Hello world” in Qt /2

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

“Hello world” in Qt /3

```
# File: helloworld.pro
SOURCES = main.cpp
HEADERS += # No headers used
QT = core gui widgets
```

“Hello world” in Qt /4

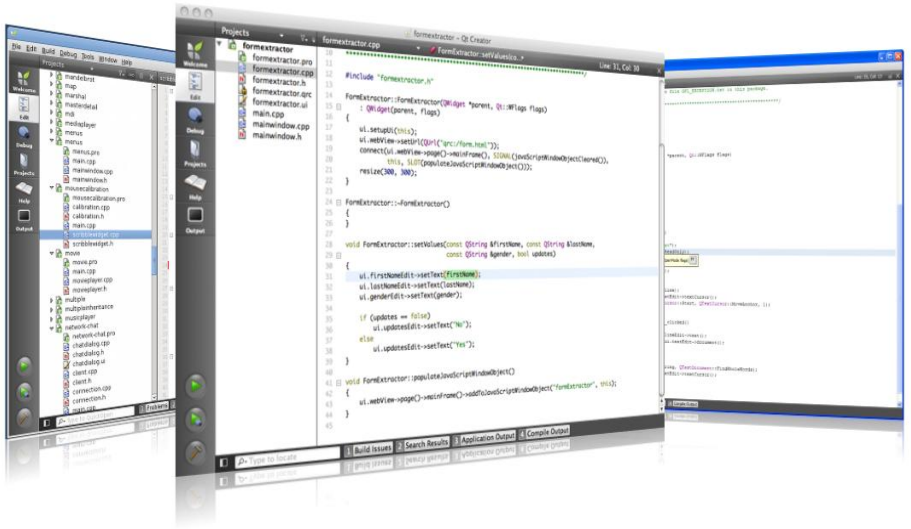
- qmake tool
- Creates cross-platform make-files

- Build project using qmake

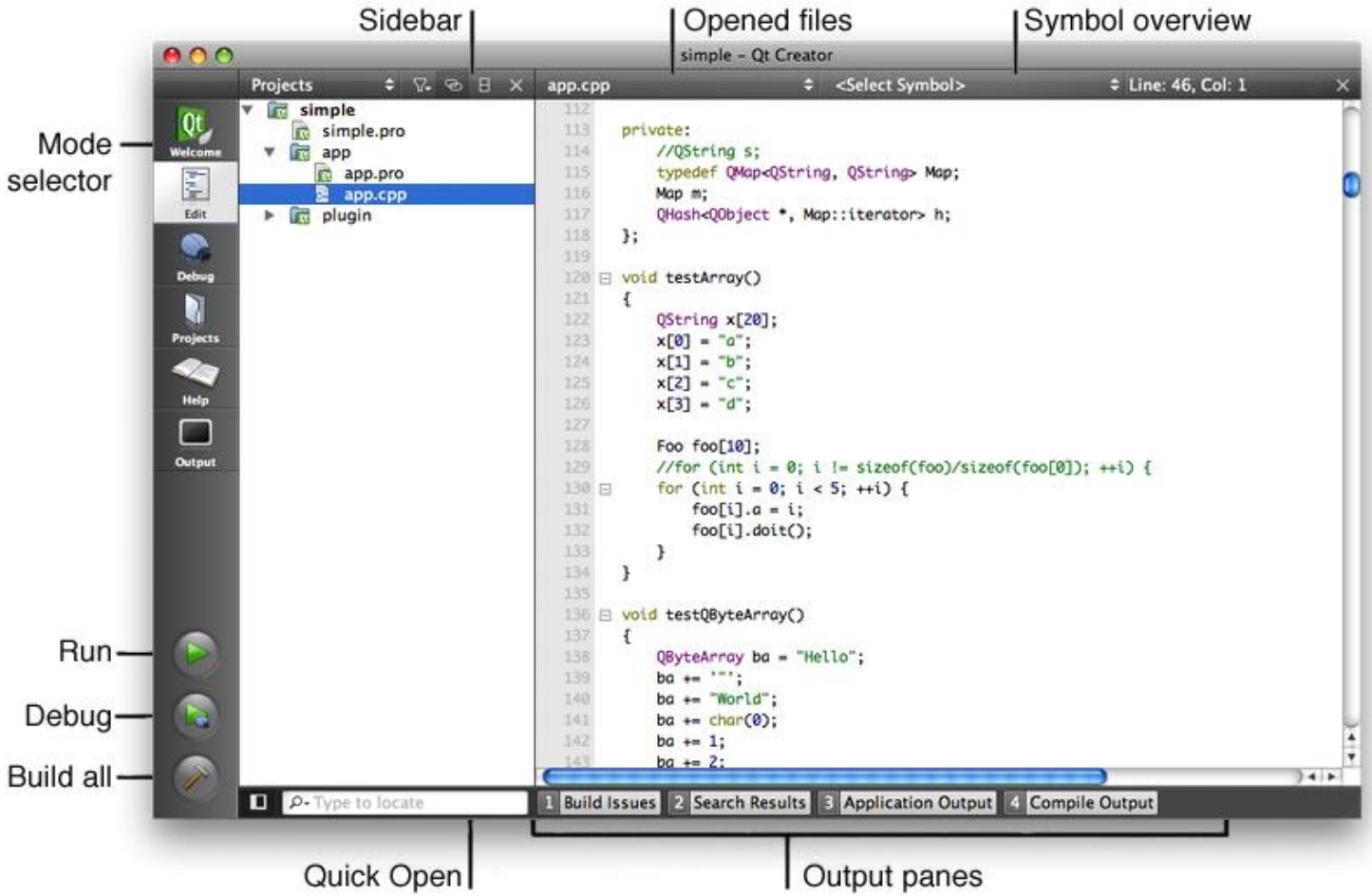
```
> cd helloworld
> qmake helloworld.pro # creates Makefile
> make # compiles and links application

> ./helloworld # executes application
```

QtCreator

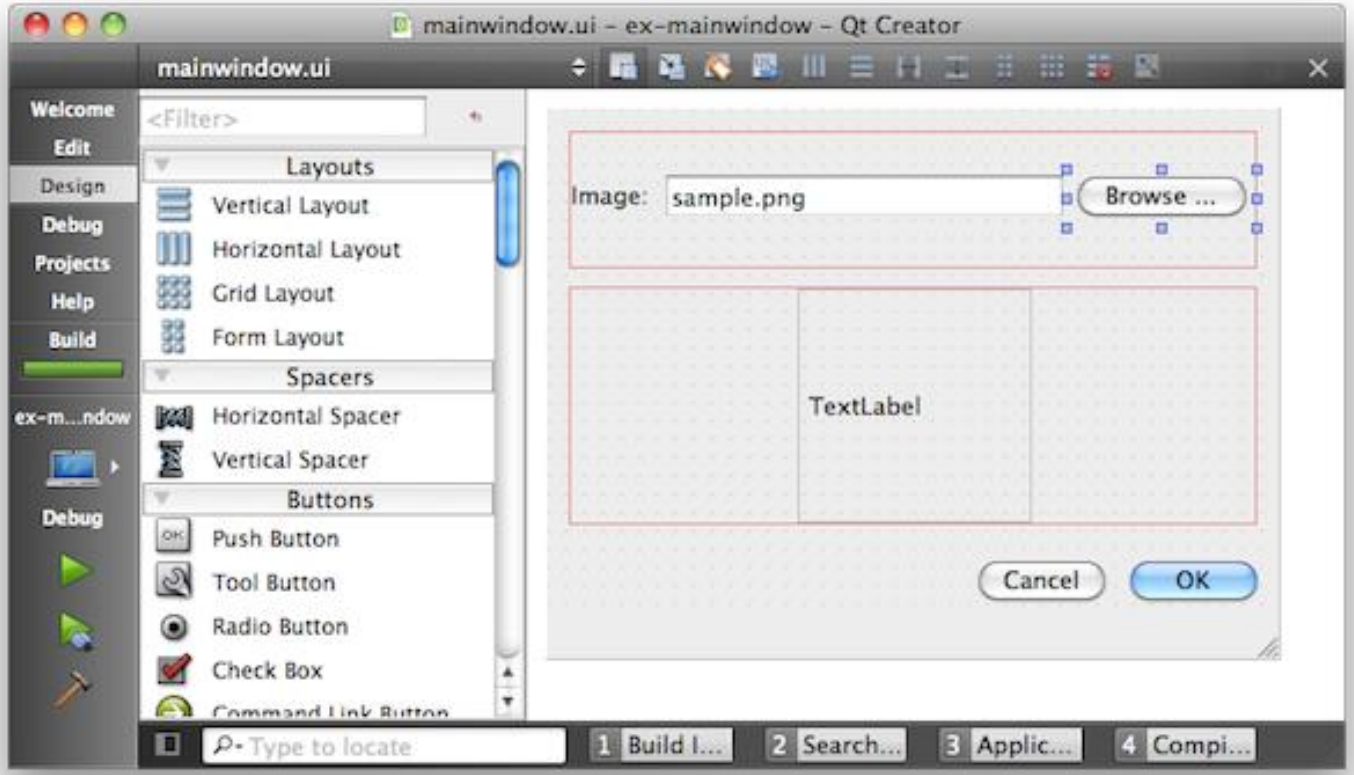
- Advanced C++ code editor
- Integrated GUI layout and forms designer
- Project and build management tools
- Integrated, context-sensitive help system
- Visual debugger
- Rapid code navigation tools
- Supports multiple platforms



QtCreator


Sidebar
Opened files
Symbol overview
Mode selector
Run
Debug
Build all
Quick Open
Output panes
simple - Qt Creator
Projects
app.cpp
<Select Symbol>
Line: 46, Col: 1
Welcome
Edit
Debug
Projects
Help
Output
simple
simple.pro
app
app.pro
app.cpp
plugin
Type to locate
1 Build Issues
2 Search Results
3 Application Output
4 Compile Output

Qt Designer


mainwindow.ui – ex-mainwindow – Qt Creator
mainwindow.ui
Welcome
Edit
Design
Debug
Projects
Help
Build
ex-m...ndow
Debug
<Filter>
Layouts
Vertical Layout
Horizontal Layout
Grid Layout
Form Layout
Spacers
Horizontal Spacer
Vertical Spacer
Buttons
Push Button
Tool Button
Radio Button
Check Box
Image:
sample.png
Browse ...
TextLabel
Cancel
OK
Type to locate
1 Build I...
2 Search...
3 Applic...
4 Compi...

Qt Object Communication

Object communication

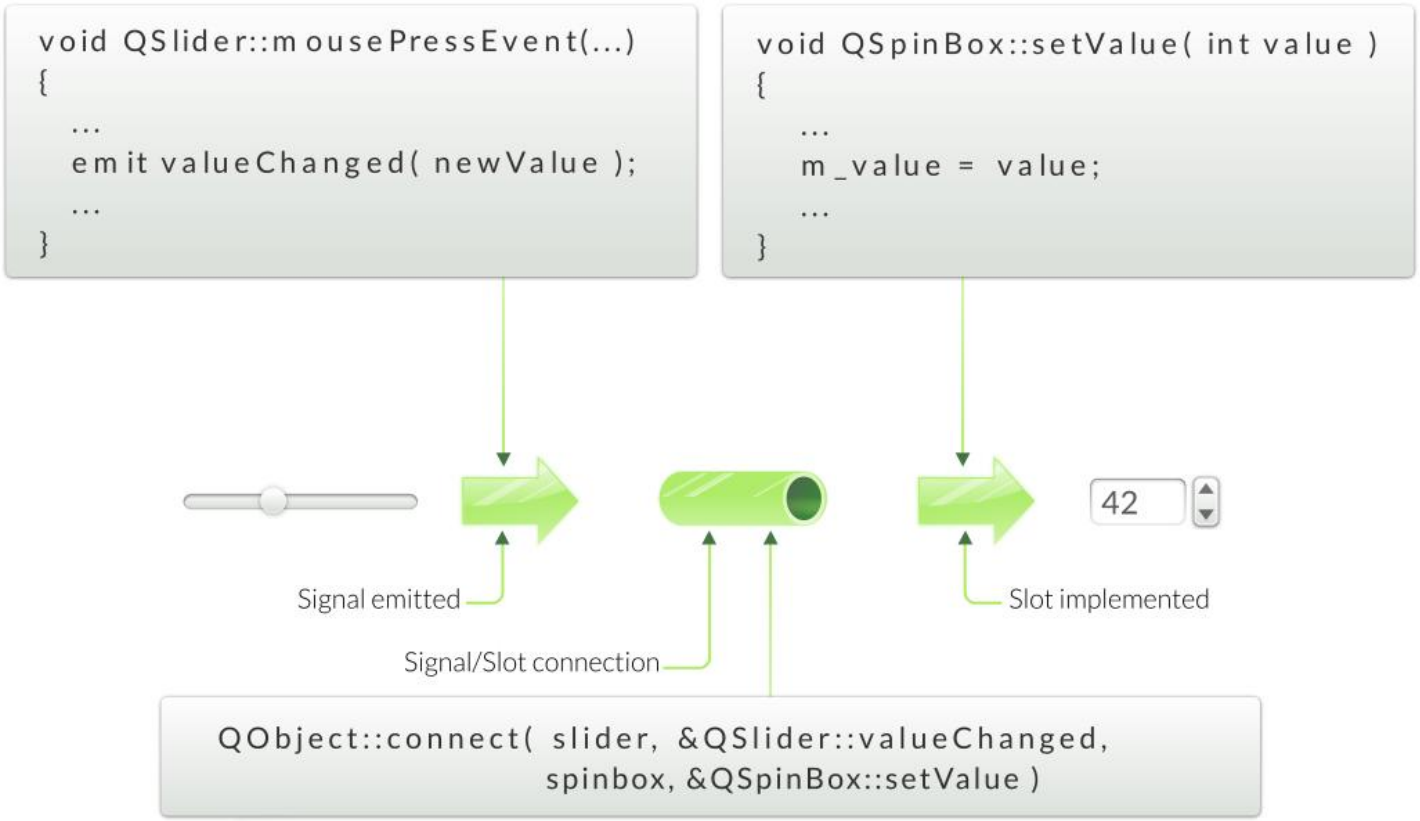
- Between objects
  Signals & Slots
- Between Qt and the application
  Events
- Between Objects on threads
  Signal & Slots + Events
- Between Applications
  DBus, QSharedMemory

Callbacks

General Problem
How do you get from "the user clicks a button" to your business 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.

Signals & Slots



Connection variants

- Qt 4 style:

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

- Qt 5, using function pointers:

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

- Using non-member function:

  static void **printValue**(int value) {...}

```
connect( slider, &QSignal::valueChanged, &printValue );
```

Custom slots

- File: **myclass.h**

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

- File: **myclass.cpp**

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

Custom signals

- File: **myclass.h**

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

- File: **myclass.cpp**

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

- Sending a signal

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

About connections

Rule for Signal/Slot Connection

Can ignore arguments, but not create values from nothing

Signals		**Slot**
rangeChanged(int,int)	V	setRange(int,int)
	V	setValue(int)
	V	update()
valueChanged(int)	V	setValue(int)
	V	update()
	X	setRange(int,int)
	V	setValue(float)*
textChanged(QString)	X	setValue(int)

* Though not for Qt4 connection types

Variations of Signal/Slot Connections

Signal(s)	Connect to	Slot(s)
one	OK	many
many	OK	one
one	OK	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 )))
```

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 handling

- `QObject::event(QEvent *event)`
  - Handles all events for this object
- Specialized event handlers for QWidget:
  - `mousePressEvent()` for mouse clicks
- 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

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
    }
}
```

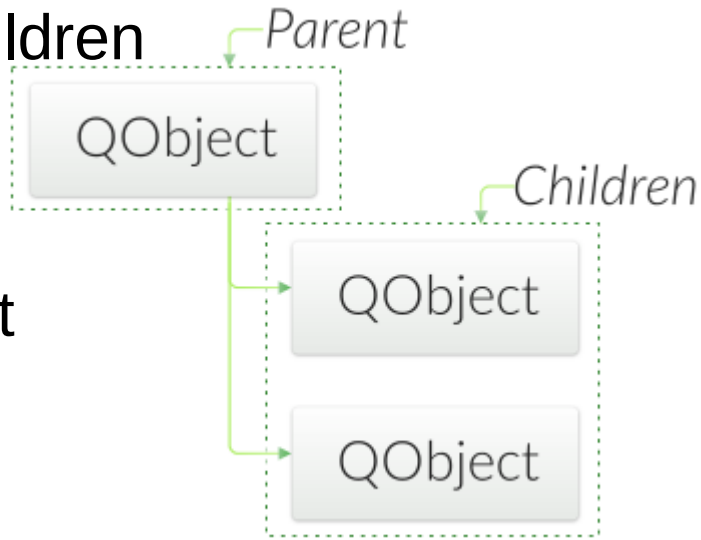
Core types

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

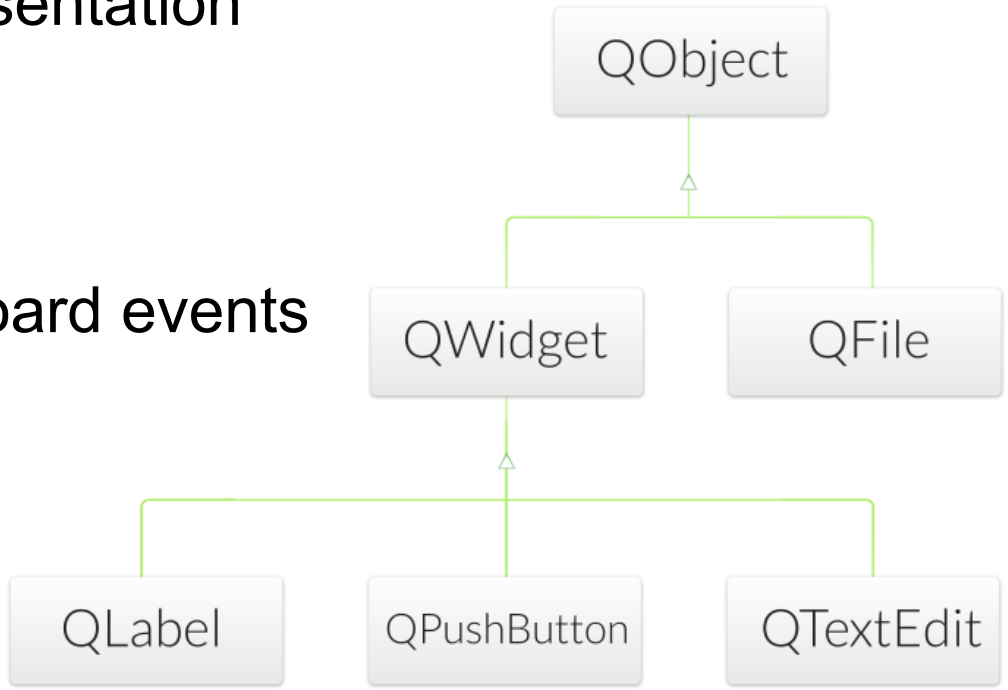
- QObjects organize themselves in object trees
  - Based on parent-child relationship
- `QObject(QObject *parent = 0)`
  - Parent adds object to list of children
  - Parent owns children
- Widget Centric
  - Used intensively with QtWidget



Note: Parent-child relationship is NOT inheritance

Qt's Widget Model - QWidget

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



Object Tree and QWidget

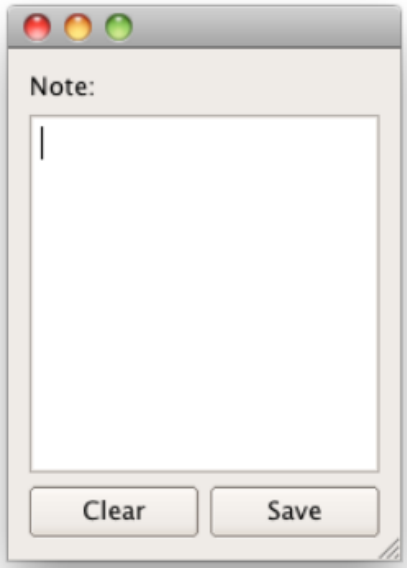
- `new QWidget(0)`
  - Widget with no parent = "window"
- QWidget's children
  - Positioned in parent's coordinate system
  - Clipped by parent's boundaries
- QWidget parent
  - Propagates state changes
  - hides/shows children when it is hidden/shown itself
  - enables/disables children when it is enabled/disabled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

Widgets that contain other widgets

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

Example Container Widget

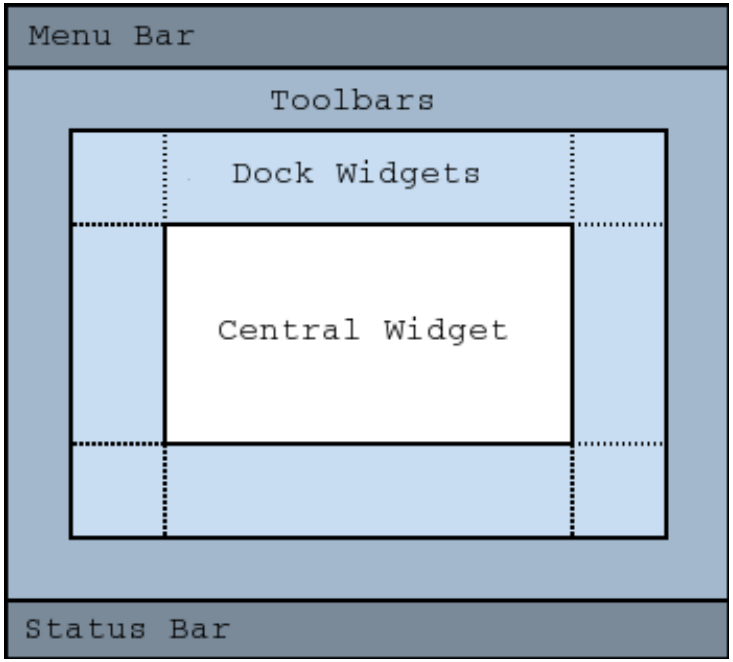
```
// container (window) widget creation
QWidget* container = new QWidget;
QLabel* label = new QLabel("Note:", container);
QTextEdit* edit = new QTextEdit(container);
QPushButton* clear = new QPushButton("Clear", container);
QPushButton* save = new QPushButton("Save", container);


// 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
```



Application creation

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

QAction /2

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

    connect(action, SIGNAL(triggered()), SLOT(onOpen()));

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

Dialogs

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

- 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()),
                              SLOT(onFindNext()));
    }
    m_findDialog->show(); // returns immediately
    m_findDialog->raise(); // on top of other windows
    m_findDialog->activateWindow(); // keyboard focus
}
```

Deploy your application

- 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