13th Summer School on SCIENTIFIC VISUALIZATION

Introduction to GUI development using Qt

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

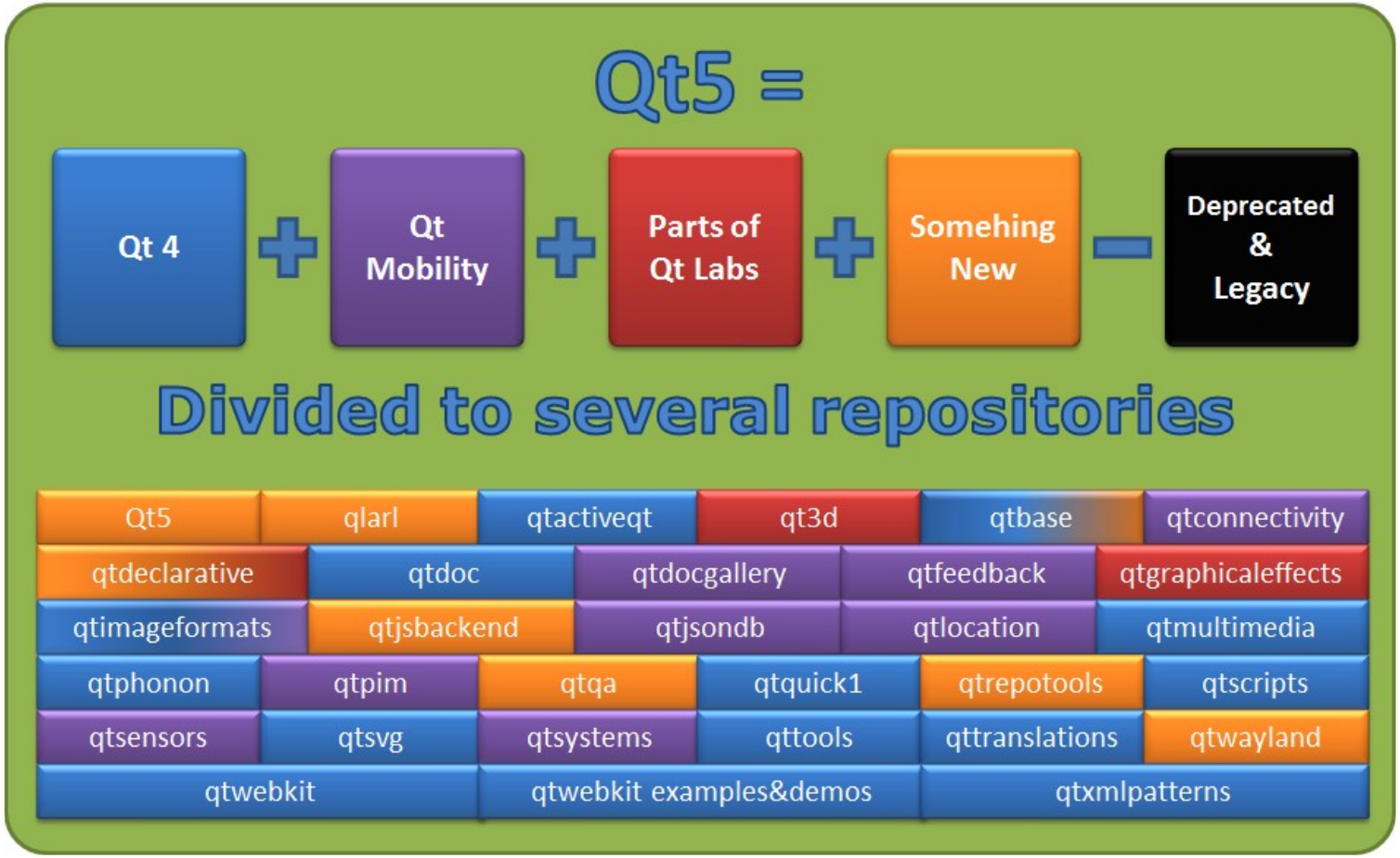
SuperComputing Applications and Innovation Department

CINECA

What is Qt

- Qt is a cross-platform development framework written in C++
- Can be used in several programming languages through bindings
  - Ruby
  - Java
  - Perl
  - Python → **PyQt**
- The Qt Toolkit is a collection of classes for various purposes
  - Database management
  - XML
  - WebKit
  - Multimedia
  - Networking
  - ...
- For **desktop**, **mobile** and **embedded** development
  - Used by more than 350,000 **commercial** and **open source** developers
  - Backed by Qt consulting, support and training
  - Trusted by over 6,500 companies worldwide

Qt 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

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

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

PyQt

- PyQt is a set of Python bindings for Qt framework
  - Bindings implemented as Python modules (620+ classes)
  - Almost the entire Qt library is available
- Take advantage of both languages key strength
  - Python: easy to learn, lot of extensions, no compilation required
  - Qt: abstraction of platform-specific details, GUI designer

“Hello world” in PyQt 1/2

```
from PyQt4.QtCore import *
from PyQt4.QtGui import *
import sys

app = QApplication(sys.argv)

PushButton = QPushButton("Hello World")
PushButton.show()

sys.exit(app.exec_())
```

“Hello world” in PyQt 2/2

* **`sys`** module needed to access command-line arguments

* **QtCore** and **QtGui** (from PyQt4 library) contains GUI widgets

* Every PyQt application must have a **QApplication** object

* Create a new instance of a **QPushButton**

* Call **`show()`** to schedule a “paint event”

* The call to **`app.exec_()`** starts the event loop

Core types

QObject

QObject is the heart of Qt's object model

Include these features:

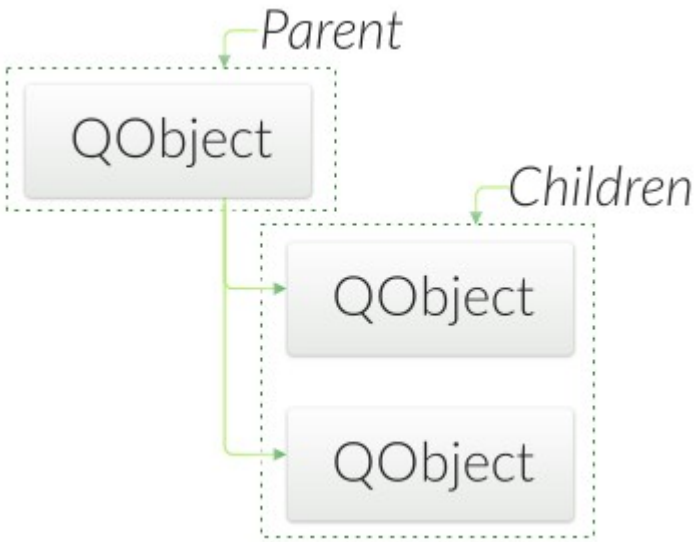
- Memory management
- Object properties
- Introspection
- Signals and slots
- Event handling

QObject has no visual representation

Object tree

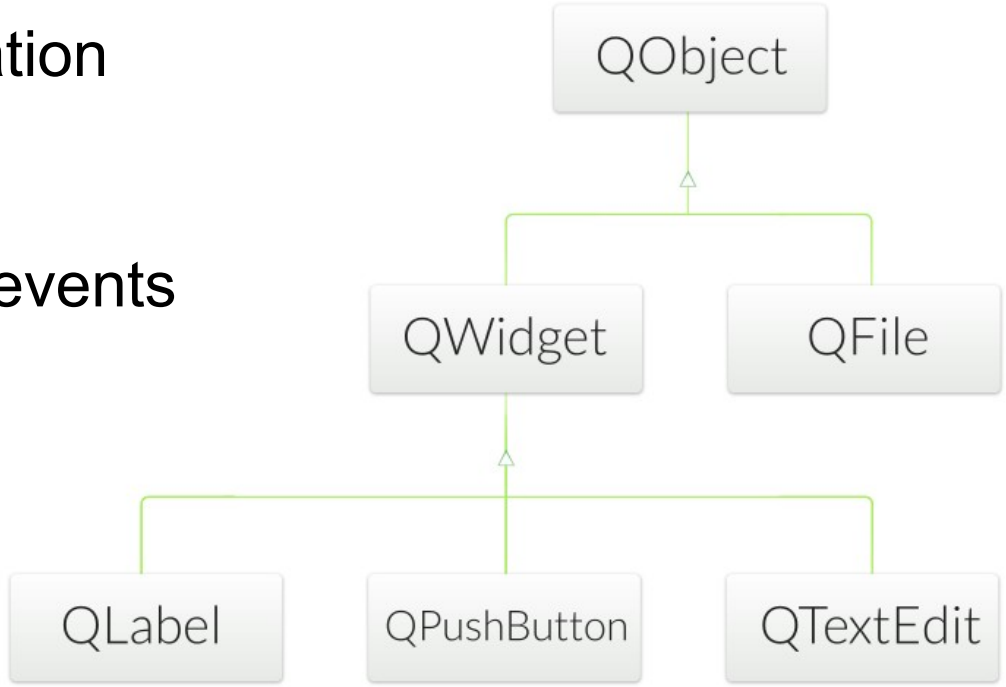
- QObjects organize themselves in object trees
  - Based on parent-child relationship
- QObject (QObject *parent = 0)
- Parent adds object to list of children
- Parent owns children
- Used intensively with QWidget

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

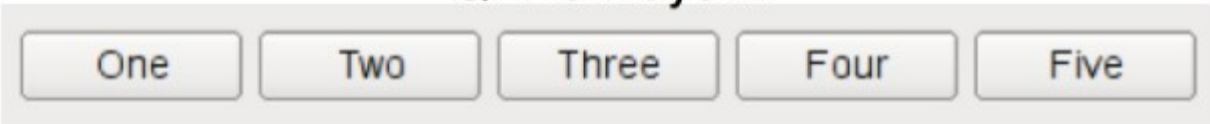
- `new QWidget(0)`
  - Widget with no parent = "window"
- QWidget children
  - Positioned in parent's coordinate system
  - Clipped by parent's boundaries
- QWidget parent
  - Propagates state changes
  - hides/shows children when it is hidden/shown itself
  - enables/disables children when it is enabled/disabled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

Widgets containing other widgets

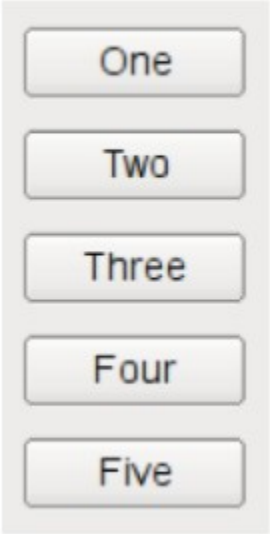
- **Container Widget**
  - Aggregates other child-widgets
- **Use layouts for aggregation**
  - QHBoxLayout, QVBoxLayout, QGridLayout
  - Note: Layouts are not widgets
- **Layout Process**
  - Add widgets to layout
  - Layouts may be nested
  - Set layout on container widget
  - Hint: use QtDesigner to apply layouts!

Layout: examples

QHBoxLayout



QVBoxLayout



QGridLayout

Object communication

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

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 and slots

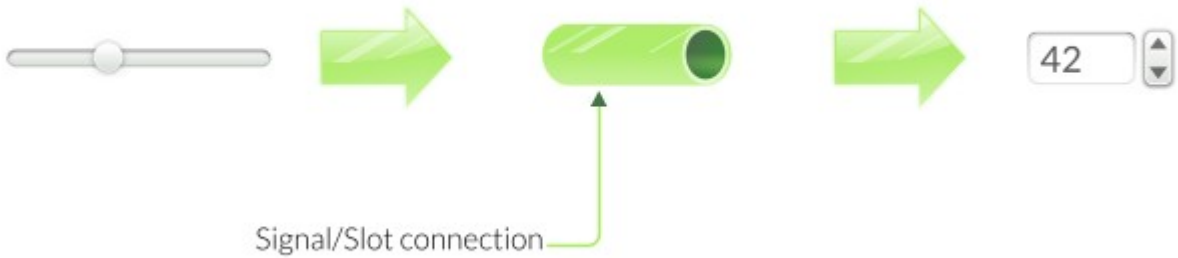
- Every PyQt object deriving from QObject supports S&S mechanism
- Widgets emit **signals**
- A signal announce state changes:
  - a button was clicked
  - a checkbox is checked/unchecked
  - editing in a text field finished
- Widgets react to a signal through **slots**
- **Connections** are used to link signals and slots

Signals & Slots 1/8

Signals & Slots 2/8

Signals & Slots 3/8

Signals & Slots 4/8



Signals & Slots 5/8


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

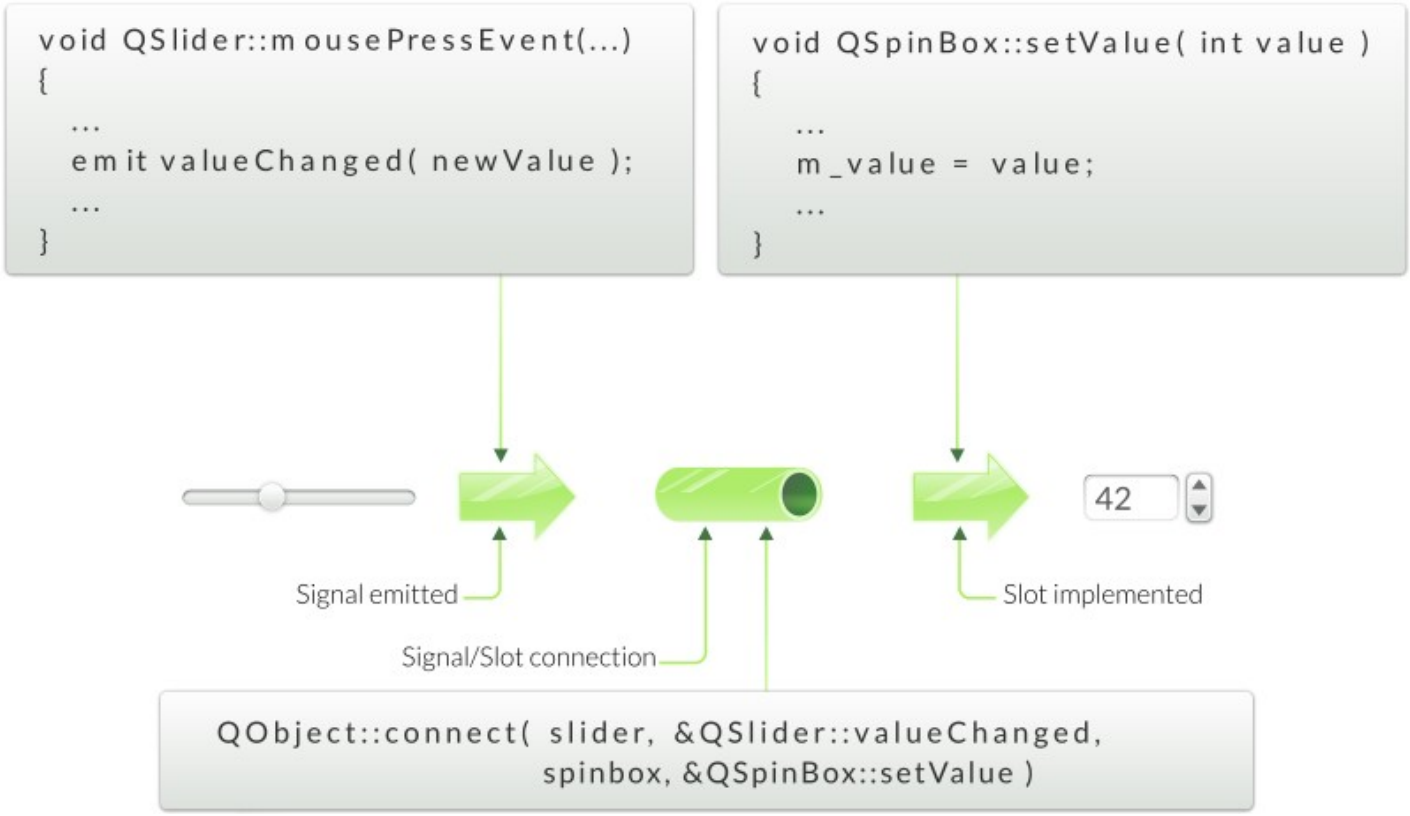
Signals & Slots 6/8

```
void QSlider::mousePressEvent(...)
{
  ...
  emit valueChanged( newValue );
  ...
}
```

Signals & Slots 7/8

```
void QSpinBox::setValue( int value )
{
   ...
   m_value = value;
   ...
}
```

Signals & Slots 8/8



About connections 1/4

Connection syntax (old school, the same as C++ Qt framework):

```
connect(w1, SIGNAL(signature), w2, SLOT(signature))
```

w1: source widget, sending a signal
SIGNAL(signature): signal to be connected
w2: destination widget, which react to the signal with a slot
SLOT(signature): method to be called when the signal is emitted

Example:

```
self.connect(aButton, SIGNAL('clicked()'), self, SLOT('close()'))
```

In this case, when the button aButton is clicked, the containing widget (self) will be closed

About connections 2/4

Rule for Signal/Slot Connection:

“Can ignore arguments, but not create values from nothing”

Signal		Slot
rangeChanged(int,int)	ok	setRange(int,int)
rangeChanged(int,int)	ok	setValue(int)
rangeChanged(int,int)	ok	update()
valueChanged(int)	ok	setValue(int)
valueChanged(int)	ok	update()
valueChanged(int)	ok	setRange(int,int)
valueChanged(int)	ko	setValue(float)*
textChanged(QString)	ko	setValue(int)

* Though not for Qt4 connection types

About connections 3/4

Signal(s)	Connect to	Slot(s)
one	OK	many
many	OK	one
one	OK	another signal

- Signal to Signal connection

```
connect(btn, SIGNAL('clicked()'),
        self, SIGNAL('emitOkSignal()'));
```

- **Not** allowed to name parameters

```
connect(mySlider,SIGNAL('valueChanged(int
value)')
    self, SLOT('setValue( int newValue )'))
```

About connections 4/4

Old connection syntax has a serious issue:

if you don't write the signal signature exactly, signal will not be fired, but no warning or exception will be thrown.
To avoid this behavior, there is another syntax for connections with PyQt:

```
sender.signalName.connect(receiver.slotName)
```

So the previous example:

```
self.connect(aButton, SIGNAL('clicked()'), self, SLOT('close()'))
```

Now become:

```
aButton.clicked.connect(self.close)
```

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

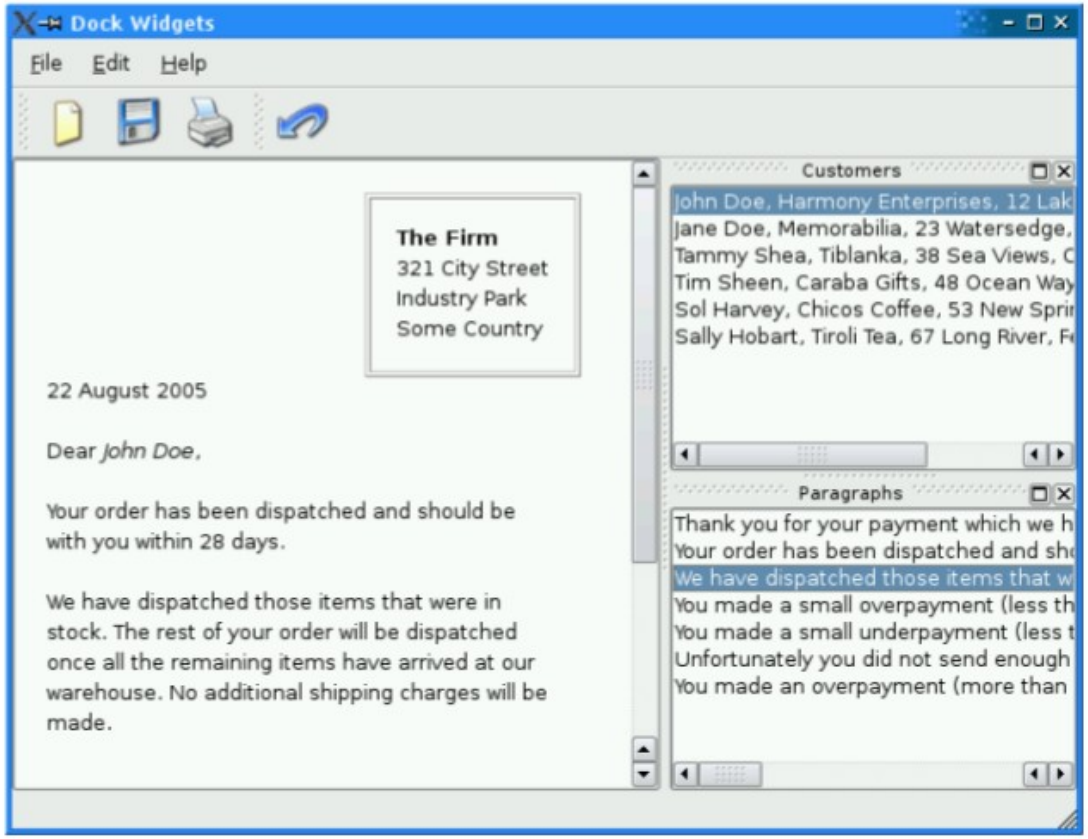
Event handling

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

Application creation

Main Window

- QMainWindow: main application window
  - Has own layout
  - Central Widget
  - QMenuBar
  - QToolBar
  - QDockWidget
  - QStatusBar



QAction 1/2

- 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 2/2

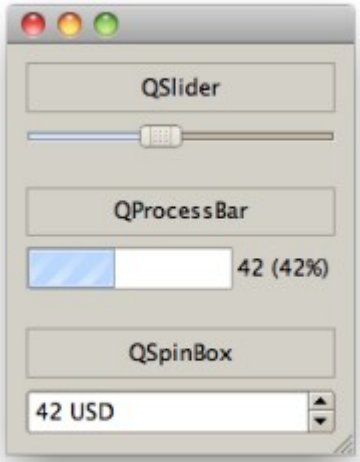
To create an action, you can:

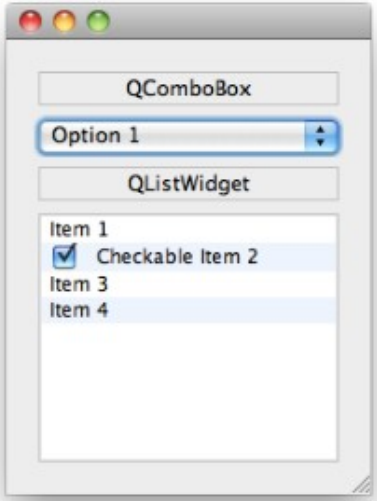
- Instantiate a QAction object directly
- Call addAction() on existing QMenu and QtoolBar objects
- Then you can share it with other objects

```
self.saveAction = QAction(QIcon(":/images/save.png"), "&Save...",
self)
self.saveAction.setShortcut("Ctrl+S")
self.saveAction.setStatusTip("Save the current form letter")
self.connect(self.saveAct, QtCore.SIGNAL("triggered()"), self.save)
...
self.fileMenu = self.menuBar().addMenu("&File")
self.fileMenu.addAction(self.saveAction)
...
self.fileToolBar = self.addToolBar("File")
self.fileToolBar.addAction(self.saveAct)
```

Widgets

Common widgets





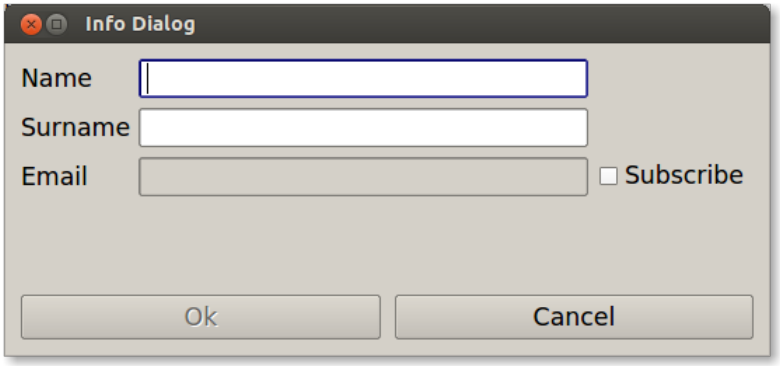
Common signals

Widget	Signals
QPushButton	clicked()
QLineEdit	editingFinished(), returnPressed(), textChanged(const QString&)
QComboBox	activated(int), currentIndexChanged(int)
QCheckBox	stateChanged(int)
QSpinBox	valueChanged(int)
QSlider	rangeChanged(int,int), valueChanged(int)

Dialogs

QDialog

- Base class of dialog window widgets
- General Dialogs can have 2 modes:

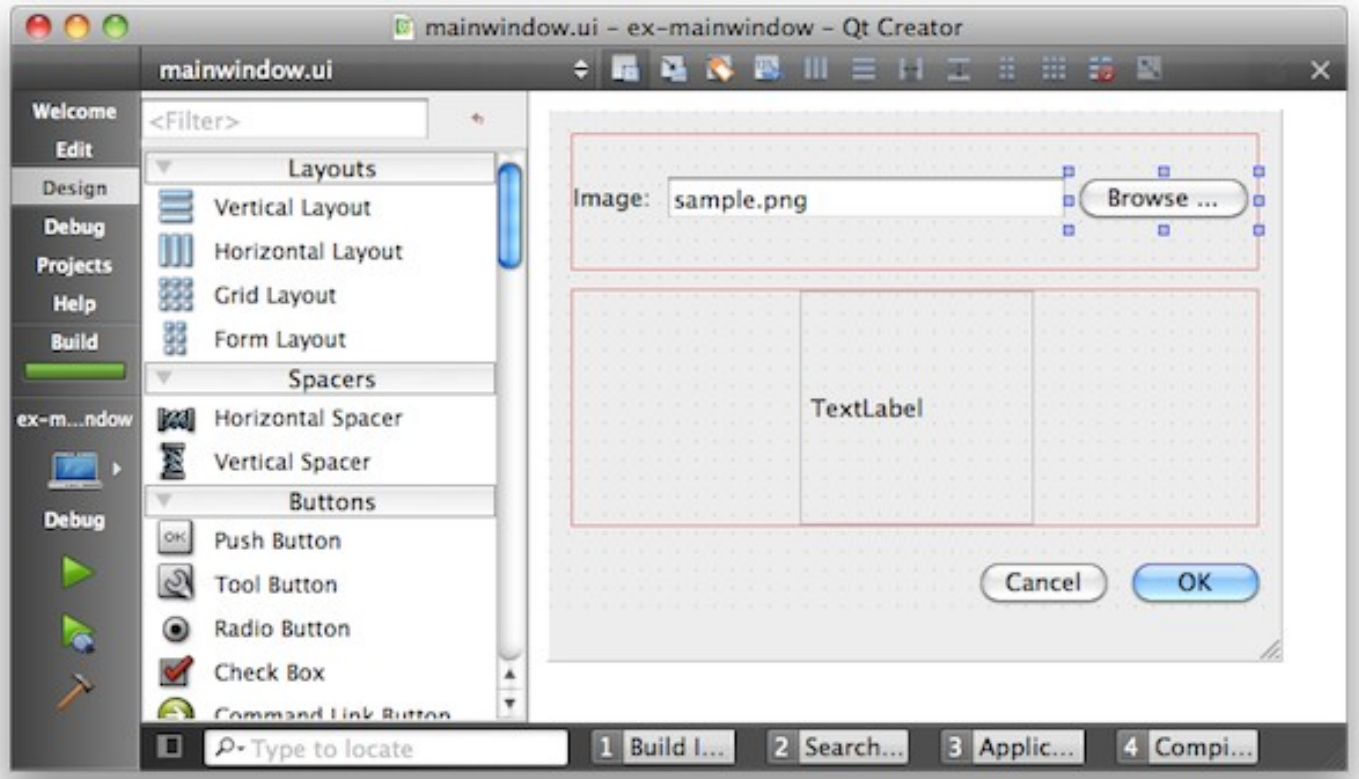


- **Modal dialog**
  - Remains in foreground, until closed
  - Blocks input to remaining application
  - Example: Configuration dialog

- **Non-Modal dialog**
  - Operates independently in application
  - Example: Find/Search dialog

Building User Interfaces

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...

Build GUI using QtDesigner 1/3

- Qt Designer uses **XML .ui files** to store designs and does not generate any code itself
- **`pyuic4`** takes a Qt4 user interface description file and compiles it to Python code
- The Python code is structured as a single class that is derived from the Python object type
- Class name is the name of the top level object set in Designer with **`Ui_`** prepended
- The class contains a method called **`setupUi()`**
  - This takes a single argument which is the widget in which the user interface is created

Build GUI using QtDesigner 2/3

1) create your GUI (or use MyDialog.ui from pyuicExample)

2) generate the .py file

```
pyuic4 -o MyDialog_auto.py MyDialog.ui
```

3) use ui interface

```
from MyDialog_auto import Ui_Dialog

app = QApplication(sys.argv)
Dialog = QDialog()  ### create new dialog
ui = Ui_Dialog()    ### create a new instance of your gui
ui.setupUi(Dialog)  ### apply the gui to the created dialog

Dialog.show()
sys.exit(app.exec_())
```

Build GUI using QtDesigner 3/3

> pyuic4 -h

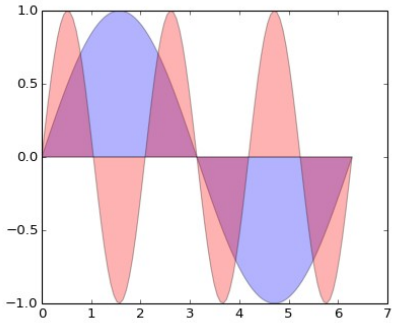
```
Usage: pyuic4 [options] <ui-file>

Options:
  --version             show program's version number and exit
  -h, --help            show this help message and exit
  -p, --preview         show a preview of the UI instead of generating code
  -o FILE, --output=FILE
                        write generated code to FILE instead of stdout
  -x, --execute         generate extra code to test and display the class
  -d, --debug           show debug output
  -i N, --indent=N      set indent width to N spaces, tab if N is 0 (default:
                        4)
  -w, --pyqt3-wrapper   generate a PyQt v3 style wrapper

  Code generation options:
    --from-imports      generate imports relative to '.'
```

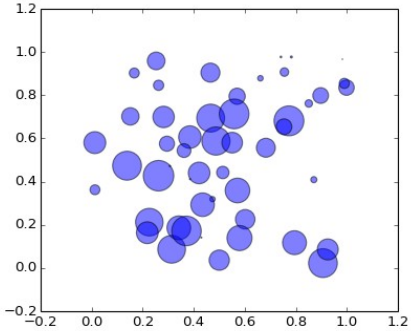
With `-x` option the generated Python class should be executed standalone to be displayed

Matplotlib and Qt 1/6

Matplotlib is a Python 2D interactive plotting library

http://matplotlib.org/



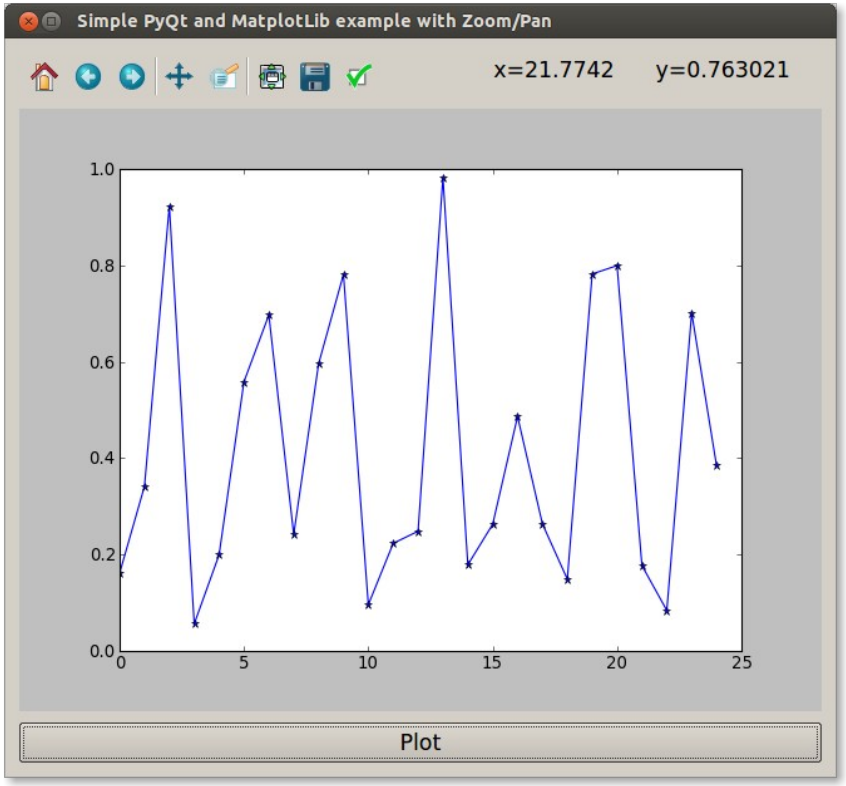


We will see how to:

- Embed a Matplotlib Figure into a Qt window
- Embed a Navigation Toolbar

Matplotlib and Qt 2/6

Open **`MatplotlibExample/matplotlibExample.py`**



App features:

- generate a set of 25 points and plot it pressing “Plot” button
- show navigation toolbar for zooming/panning

Matplotlib and Qt 3/6

```
#import modules from Matplotlib
from matplotlib.backends.backend_qt4agg import FigureCanvasQTAgg as FigureCanvas
from matplotlib.backends.backend_qt4agg import NavigationToolbar2QTAgg as NavigationToolbar
import matplotlib.pyplot as plt

#import random module to generate set
import random
```

Figure Matplotlib object: this is the backend-independent representation of our plot

Import from the **matplotlib.backends.backend_qt4agg** the module **FigureCanvasQTAgg** class, which is the backend-dependent figure canvas. It contains the backend-specific knowledge to render the Figure we've drawn.

Note that **FigureCanvasQTAgg**, other than being a Matplotlib class, is also a Qwidget, the base class of all user interface objects. So this means we can treat FigureCanvasQTAgg like a pure Qt widget Object. NavigationToolbar2QTAgg also inherits from QWidget, so it can be used as Qt objects in a Qapplication.

References:

http://matplotlib.org/api/backend_qt4agg_api.html
http://matplotlib.org/api/pyplot_api.html
http://matplotlib.org/api/figure_api.html#module-matplotlib.figure
https://docs.python.org/2/library/random.html

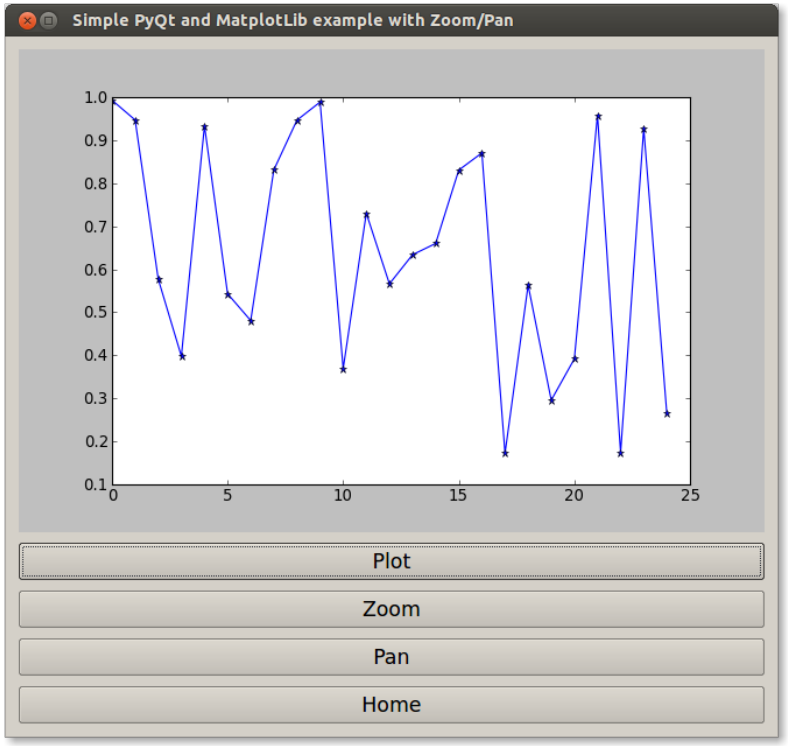
Matplotlib and Qt 4/6

```
class Window(QtGui.QDialog):
    def __init__(self, parent=None):
        super(Window, self).__init__(parent)

        #init figure and canvas
        self.figure = plt.figure()
        self.canvas = FigureCanvas(self.figure)

        #init nav toolbar
        self.toolbar = NavigationToolbar(self.canvas, self)

        # Add plot button
        self.button = QtGui.QPushButton('Plot')

        # connect button to custom slot (see later)
        self.button.clicked.connect(self.plot)

        # set the layout
        layout = QtGui.QVBoxLayout()
        layout.addWidget(self.toolbar)
        layout.addWidget(self.canvas)
        layout.addWidget(self.button)
        self.setLayout(layout)
```

Matplotlib and Qt 5/6

```
### our custom slot
def plot(self):
        # random data
        data = [random.random() for i in range(25)]

        # create an axis
        ax = self.figure.add_subplot(1,1,1)

        # discards the old graph
        ax.hold(False)

        # plot data
        ax.plot(data, '*-')

        # refresh canvas
        self.canvas.draw()
```

Matplotlib and Qt 6/6



Exercise

Modify the previous example adding custom buttons which will act as the navigation toolbar:

Plot > plot random dataset
Zoom > activate zoom on canvas
Pan > activate pan on canvas
Home > reset view

Hint #1: you will have to connect your buttons to navigation toolbar **`zoom()`**, **`pan()`** and **`home()`** methods

Hint #2: open **`MatplotlibExample/matplotlibExampleCustom.py`**

Resources

[PDF] PyQt whitepaper
http://www.riverbankcomputing.co.uk/static/Docs/PyQt4/pyqt-whitepaper-a4.pdf

[BOOK] Rapid GUI Programming with Python and Qt
http://qt-project.org/books/view/rapid_gui_programming_with_python_and_qt