QWidget

WEEK 3

Instructor: Ph.D. Oleg Tymchuk

Overview

Qt’s Widget Model

Object communication

Signal & Slots

Event Processing & handling

Qt’s Widget Model

The QWidget class is the base class of all user interface objects

The widget is the atom of the user interface:

- widget receives mouse, keyboard and other events from the window system
- widget paints a representation of itself on the screen
- widget is rectangular
- widgets are sorted in a Z-order
- widget is clipped by its parent and by the widgets in front of it

Qt’s Widget Model. Top-Level and Child Widgets

TOP-LEVEL WIDGET

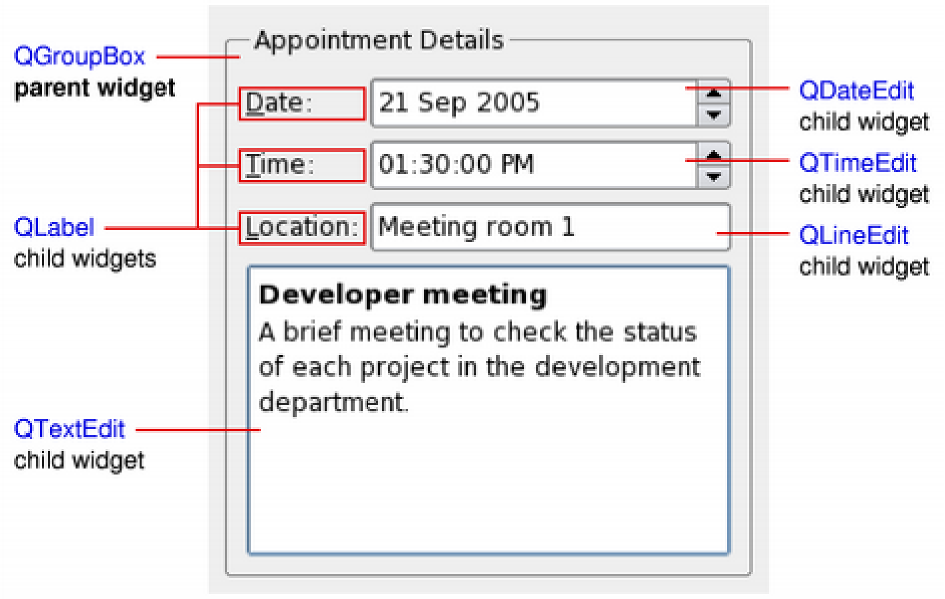
- widget without a parent widget is always an independent window
- usually, window has a frame and a title bar
- QMainWindow and the various subclasses of QDialog are the most common window types
- hides/shows children when it is hidden/shown itself
- enables/disables children when it is enabled/disabled itself

CHILD WIDGET

- тon-window widget
- displayed within its parent widget (positioned in parent's coordinate system)
- clipped by parent's boundaries

Qt's Widget Model. Composite Widgets

- Composite widget - widget that is used as a container to group a number of child widgets
- Composite widget can be created by constructing a widget with the required visual properties - a QFrame, for example - and adding child widgets to it, usually managed by a layout (e.g. QHBoxLayout, QVBoxLayout, QGridLayout)
- The Qt layout system provides a simple and powerful way of automatically arranging child widgets within a widget to ensure that they make good use of the available space



Developer meeting
A brief meeting to check the status of each project in the development department.

Hint: use QtDesigner to apply layouts!

Object communication

Between Qt and the application

- Events

Between objects

- Signals & Slots

Between Objects on threads

- Signal & Slots + Events

Object communication

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

A callback is a pointer to a function, so if you want a processing function to notify you about some event you pass a pointer to another function (the callback) to the processing function

Observer Pattern (Listener)

- based on interface classes
- needs listener registration
- many interface classes

The observer pattern is a software design pattern in which an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods

Qt uses

- signals and slots for high-level (semantic) callbacks
- virtual methods for low-level (syntactic) events

A signal is emitted when a particular event occurs. A slot is a function that is called in response to a particular signal

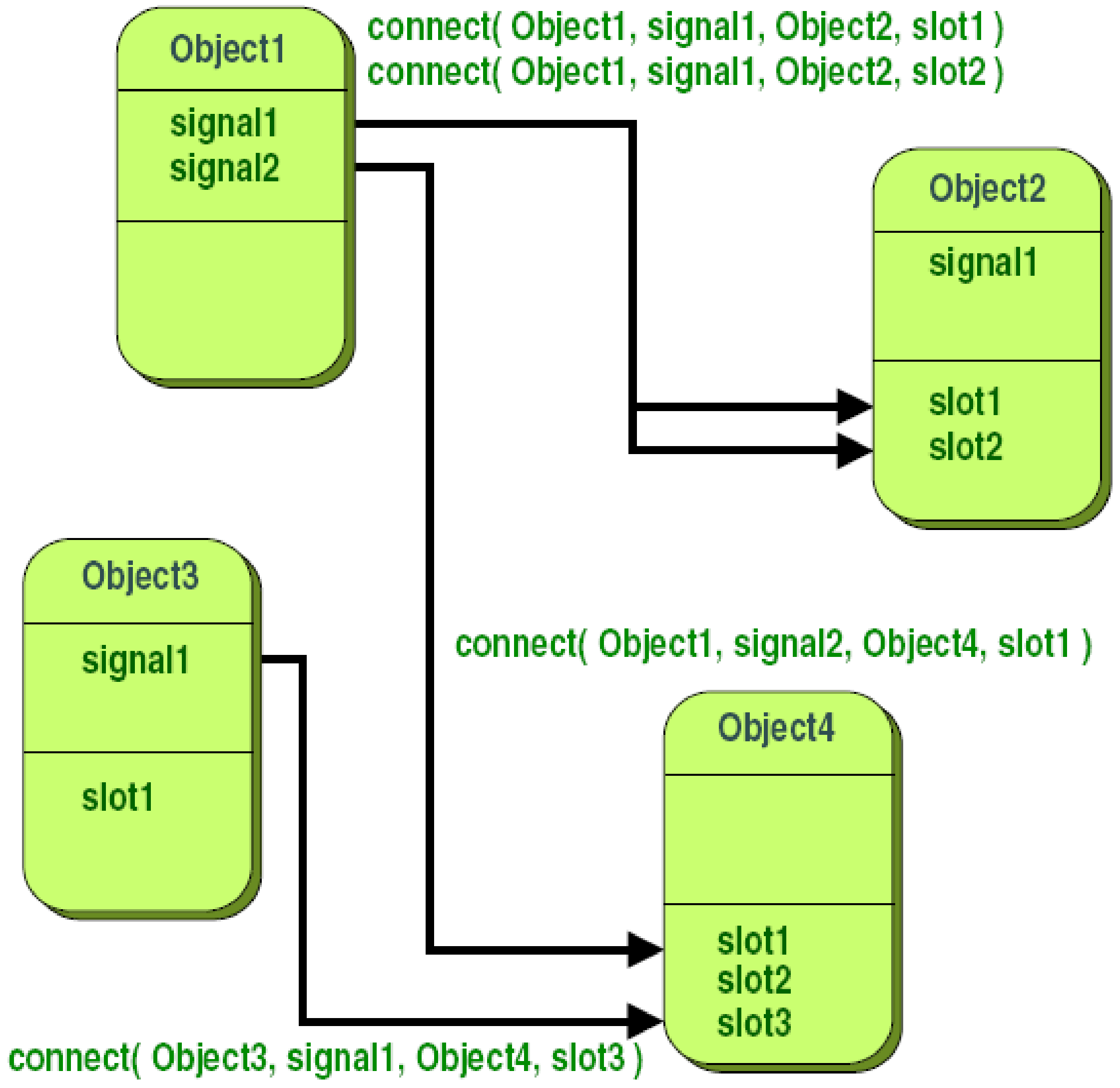
Signal & Slots

- In GUI programming, when we change one widget, we often want another widget to be notified
- More generally, we want objects of any kind to be able to communicate with one another

Example: if a user clicks a **Close** button, we probably want the window's close() function to be called

- All classes that inherit from QObject or one of its subclasses (e.g., QWidget) can contain signals and slots
- Signals are emitted by objects when they change their state in a way that may be interesting to other objects
- Slots can be used for receiving signals, but they are also normal member functions
- You can connect as many signals as you want to a single slot, and a signal can be connected to as many slots as you need
- Compared to callbacks, signals and slots are slightly slower

Signal & Slots


Object1
signal1
signal2
connect(Object1, signal1, Object2, slot1)
connect(Object1, signal1, Object2, slot2)
Object2
signal1
slot1
slot2
Object3
signal1
slot1
connect(Object1, signal2, Object4, slot1)
Object4
slot1
slot2
slot3
connect(Object3, signal1, Object4, slot3)

Signal & Slots. Connections

QObject::connect()

- Creates a connection of the given type from the signal in the sender object to the method in the receiver object
- Returns a handle to the connection that can be used to disconnect it later

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

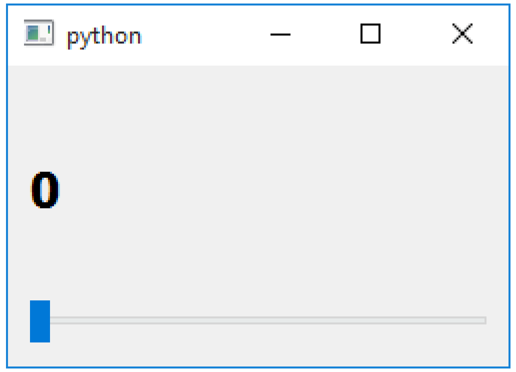
Example:

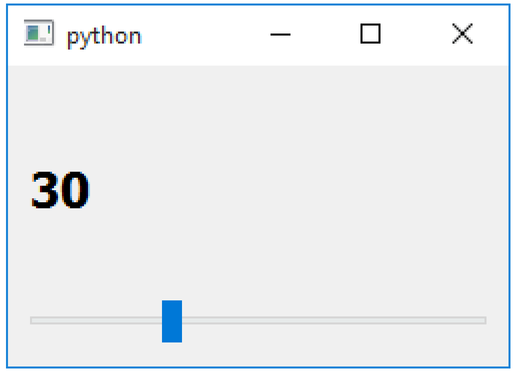
```
btn = QPushButton("Button", self)
btn.clicked.connect(self.close)
```

Signal & Slots. S&S Example

```python
import sys
from PyQt5 import QtCore, QtGui
from PyQt5.QtWidgets import
QWidget,QLabel,QSlider,QApplication,QVBoxLayout

class SignalSlot(QWidget):
    def __init__(self):
        super(SignalSlot, self).__init__()

        self.label = QLabel('0')
       f = QtGui.QFont('Times', 18, QtGui. QFont.Bold)
       self.label.setFont(f)

        self.sld = QSlider(QtCore.Qt.Horizontal, self)

        self.sld.valueChanged.connect(self.set_label_text)

        layout = QVBoxLayout(self)
        layout.addWidget(self.label)
        layout.addWidget(self.sld)
        self.setGeometry(300, 300, 250, 150)

    def set_label_text(self, value):
        self.label.setText(str(value))
```

```python
if __name__ == '__main__':
    app = QApplication(sys.argv)
    ex = SignalSlot()
    ex.show()
    sys.exit(app.exec())
```





Signal & Slots. Example of emitting signals

```python
import sys
from PyQt5 import QtCore
from PyQt5.QtWidgets import QWidget, QApplication

class Communicate(QtCore.QObject):
    closeApp = QtCore.pyqtSignal()

class Example(QWidget):
    def __init__(self):
        super(Example, self).__init__()
        self.c = Communicate()
        self.init_ui()

    def init_ui(self):
        self.c.closeApp.connect(self.close)
        self.setGeometry(300, 300, 290, 150)
        self.setWindowTitle('Emit signal')
        self.show()

    def mousePressEvent(self, event):
        self.c.closeApp.emit()
```

```python
if __name__ == '__main__':
    app = QApplication(sys.argv)
    ex = Example()
    sys.exit(app.exec())
```

Event Processing & handling

- Qt is an event-driven UI toolkit
- QApplication::exec() runs the event loop

Generate Events

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

Dispatch Events

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

Queue Events

- by event loop

Handle Events

- by QObject event handler methods

Event Processing & handling

QObject::event(QEvent *event)

- Handles all events for this object

Specialized event handlers for Qwidget

- mousePressEvent() for mouse clicks
- keyPressEvent() 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

Event Processing & handling. Event handling example

```
import sys
from PyQt5 import QtCore
from PyQt5.QtWidgets import QWidget, QApplication

class Example(QWidget):
    def __init__(self):
        super(Example, self).__init__()
        self.setGeometry(300, 300, 250, 150)
        self.setWindowTitle('Event handler')
        self.show()

    def keyPressEvent(self, e):
        if e.key() == QtCore.Qt.Key_Escape:
            self.close()

if __name__ == '__main__':
    app = QApplication(sys.argv)
    ex = Example()
    sys.exit(app.exec_())
```

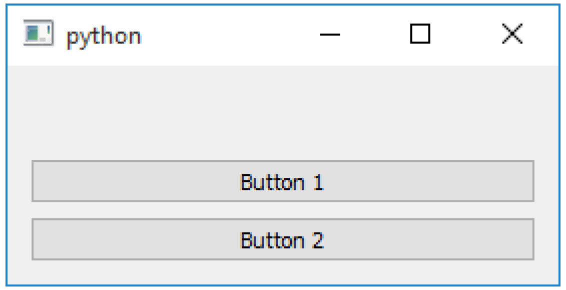
```
if __name__ == '__main__':
    app = QApplication(sys.argv)
    ex = Example()
    sys.exit(app.exec())
```

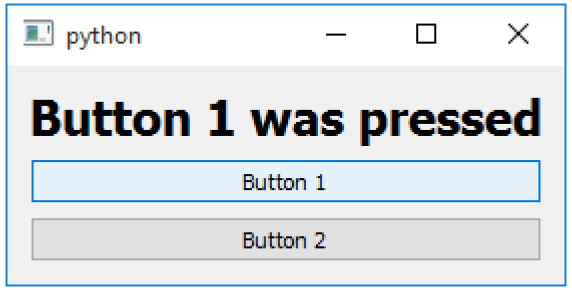
Event Processing & handling. Event sender example 1/2

```python
import sys
from PyQt5 import QtGui
from PyQt5.QtWidgets import QWidget, QApplication, QPushButton, QLabel, QVBoxLayout

class Example(QWidget):
    def __init__(self):
        super(Example, self).__init__()
        self.initUI()

    def initUI(self):
        self.label = QLabel()
        f = QtGui.QFont('Times',18, QtGui.QFont.Bold)
        self.label.setFont(f)
        btn1, btn2 = QPushButton("Button 1", self), QPushButton("Button 2", self)
        btn1.clicked.connect(self.buttonClicked)
        btn2.clicked.connect(self.buttonClicked)
        layout = QVBoxLayout(self)
        layout.addWidget(self.label)
        layout.addWidget(btn1)
        layout.addWidget(btn2)
        self.show()

    def buttonClicked(self):
        sender = self.sender()
        self.label.setText(sender.text() + ' was pressed')
```

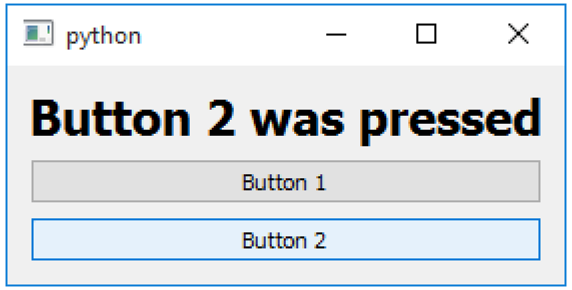
Event Processing & handling. Event sender example 2/2

```
if __name__ == '__main__':
    app = QApplication(sys.argv)
    ex = Example()
    sys.exit(app.exec_())
```





Button 1 was pressed



Button 2 was pressed