Basic User Interaction

When the user clicks the mouse anywhere on the screen, or taps a touch screen with their finger, the operating system registers a user interface event (in this case, a click) and sends a notification to the front-most application, whose window bounds match the area where the click happened.

An application won't respond to these notifications immediately. Instead they are added to the end of a queue, and your application's event loop looks at the beginning of the queue whenever it is ready to process a new event. The event loop is the infinite while loop at the core of a *Qt* application that allows interactivity between user events and your code to happen. Events coming from the operating system aren't limited to user actions either, events might be triggered by the application itself.

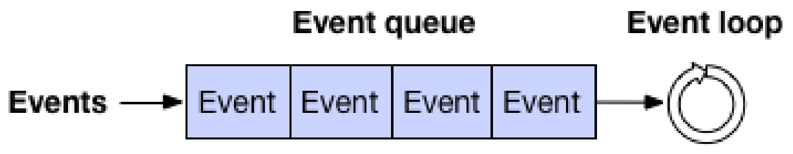


Fig1. from OS events, to the event queue, to your application

Once the event loop in your application is ready to handle more events, it starts reading them from the queue and bundles them into objects (a subclass of `QEvent`) that can be sent to receiver objects in your application: these could be `QActions` or `QWidgets`, or anything that is a subclass of `QObject`. Most often user interaction events get sent to the topmost widget of your application (usually a window). That widget then tries to respond to the event, if it has no programming to do so, it sends the event down to its child widgets in hopes that they can handle it. If none of the widgets are programmed to handle the event, then the event is ignored and the event loop looks at the next item in the queue.

Signals and Slots

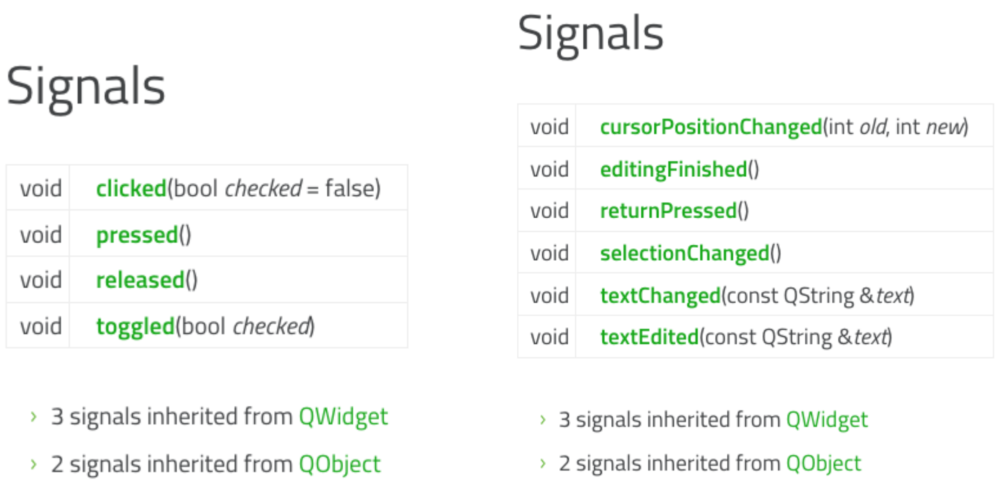
The event handling system in Qt is wrapped in an easier to use high-level formal abstraction of *signals* and *slots*. Using the abstraction of *signals* and *slots* it becomes intuitive to connect events triggered by one user action, or an event within your application, to functions that you write.

signals – are technically functions of a QObject subclass, but thanks to some Qt framework magic voodoo they have special properties which let you use them in a way quite unlike functions. One very un-function-like behavior is that all signals have the method `connect()` which allows you to connect them to *slots*. When certain events happen (e.g. a button push) a signal is *triggered* and all slots that are connected to that signal are run. For example `QPushButton` has signals for `clicked pressed released` and `toggled`

slots – are also functions, but are used in the regular way you're used to, without all the Qt framework voodoo. You pass a slot by its function name into the `connect()` method of a signal, and thereafter whenever that signal is triggered, your function gets run. Many widgets have special methods designated as slots, but it's good to remember that they're just regular methods like anything you would define in your own classes. For example, `QLineEdit` has "slots" for clearing its textfield or copying its contents to the clipboard.

USEFUL TO KNOW: *Because a single slot can be connected to multiple signals, it's sometimes useful to know which object sent the signal in the first place. You can call the* `QObject` *method* `sender()` *inside your custom slot functions to get the sending object.*

All the signals and slots of the various classes in Qt can be found in the Qt class reference. For example, if you look at the documentation for `QMediaPlayer` you'll find a number of signals that can allow your application to respond to all different events in the playback of a video or audio stream. On the other hand, if you look for the signals of `QPushButton` you won't find them, but if you scroll down to the section "Additional Inherited Members" you'll see that `QPushButton` does have signals, but they're all inherited from its parent classes – `QAbstractButton` `QWidget` and `QObject`. (admittedly, some of the signals of QWidget and QObject are pretty obscure!)



Signals

void	clicked(bool *checked* = false)
void	pressed()
void	released()
void	toggled(bool *checked*)

› 3 signals inherited from QWidget
› 2 signals inherited from QObject

Signals

void	cursorPositionChanged(int *old*, int *new*)
void	editingFinished()
void	returnPressed()
void	selectionChanged()
void	textChanged(const QString &*text*)
void	textEdited(const QString &*text*)

› 3 signals inherited from QWidget
› 2 signals inherited from QObject

Fig2. signals of `QAbstractButton` *(left) and* `QLineEdit` *(right)*

To summarize: you can connect any number of functions (slots) to a signal. So that when the signal happens, all the slots connected to that signal are triggered as well. The important method you need to know to make these connections is called `connect()` - all signals have it. The input to `connect()` is the slot (basically just a function) that should be called when the signal happens.

The program below creates a window with a text field and two big pushbuttons. When you push the "Print" button, the `clicked` signal of that button is triggered, and runs the `printText` method that was connected to it. When you push the "Clear" button, the `clicked` signal of that button is triggered, and the `clear` slot of the text field is run, erasing the text in the field.

Simple_Signals.py

```
from PyQt5.QtWidgets import QApplication, QWidget, QLineEdit, QPushButton
from PyQt5.QtGui import QFont
import sys

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

```
 9
10     def initUI(self):
11         self.setGeometry(0,0,300,150)
12         self.setWindowTitle('MyApp')
13         self.text = QLineEdit(self)
14         self.text.setGeometry(10,10,280,50)
15         self.text.setFont(QFont("Times", 30, QFont.Bold, True))
16         self.text.setText("Type Something")
17         self.but1 = QPushButton("Print", self)
18         self.but1.setGeometry(0,70,150,80)
19         self.but1.clicked.connect(self.printText)
20         self.but2 = QPushButton("Clear", self)
21         self.but2.setGeometry(150,70,150,80)
22         self.but2.clicked.connect(self.text.clear)
23         self.show()
24
25     def printText(self):
26         print self.text.text()
27
28 if __name__ == '__main__':
29     app = QApplication(sys.argv)
30     mywin = MyWindow()
31     sys.exit(app.exec_())
```

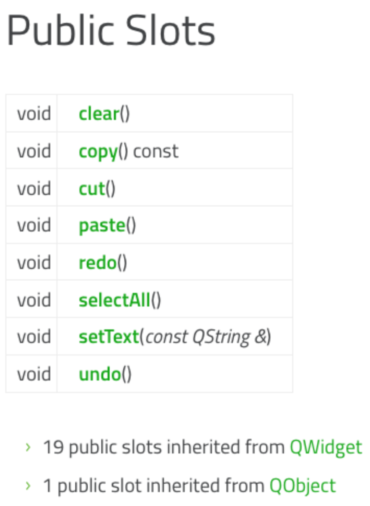
Example 1

What's New Here?

Line 19 is where the magic happens. The `clicked` signal of `self.but1` is connected to the method `self.printText –` - now whenever the user clicks on that button, Qt will run your `printText` method.

Line 22 follows the same logic, but instead of using our own custom function we use one of the predefined "slots" of `QLineEdit` (see fig 3). The `clicked` signal of `self.but2` is connected to the `clear` slot of self.text - whenever the user clicks on the "Clear" button, Qt will run the `clear` method on `self.text`

Line 26 gets the text contents of self.text using the text() method.



Public Slots

void	clear()
void	copy() const
void	cut()
void	paste()
void	redo()
void	selectAll()
void	setText(*const QString &*)
void	undo()

› 19 public slots inherited from QWidget

› 1 public slot inherited from QObject

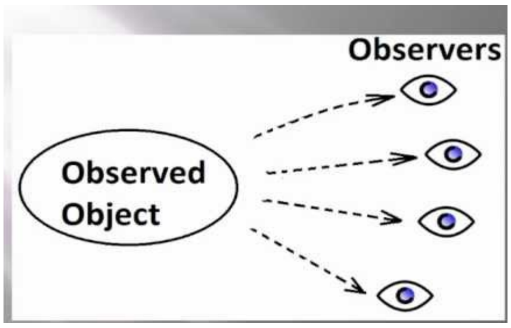
Fig3. slots of QLineEdit

The Observer Pattern

The *signals and slots* system of Qt is a specific example of a broader idea in software engineering called *The Observer Pattern*. *The Observer Pattern* is a general solution to a common problem: how can we create a one-to-many dependency between objects? In other words, how can we construct relationships between objects in our program such that when one object changes state, all its dependent objects change state in response? This should happen automatically.

This problem is extremely common in object oriented programming, and especially when making graphical user interfaces, where there are so many interconnections between all the different widgets and their user interactions.

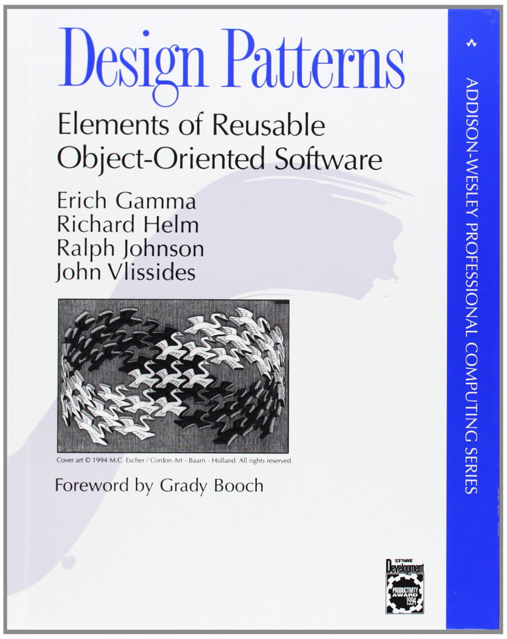
In *the Observer Pattern* you have one object, called *the subject*, that keeps a list of other objects which are dependent on it, these are called *the observers*. In Qt, *the subject* is the widget that announces a state change through one of its signals (i.e. a `QPushButton` announcing that the user has clicked on it). Whenever you use the `connect()` method, you are adding an *observer* to the subject.



Design Patterns

Design Patterns are a big deal in software engineering. When one gets to a professional level in object-oriented software design, you begin to realize that common problems come up again and again. Design patterns are proposed as recipe architectures for all sorts of common software problems. They're like a professional software engineer's playbook. They also give software engineers a common language to discuss various approaches to creating large-scale software architectures.

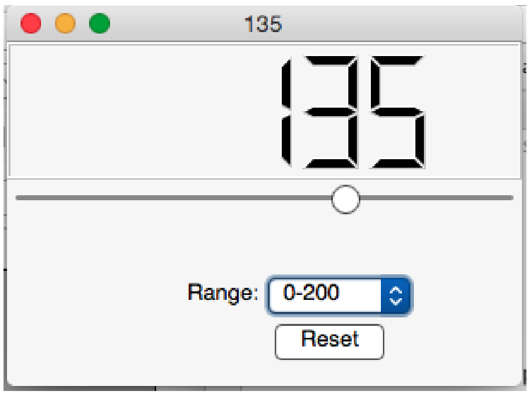
The list of Design Patterns also includes ***Anti-Patterns***, solutions that seem appealing at first but actually have significant drawbacks. *Design Patterns* are also a point of contention. Should programmers be shackled to standard best practice solutions, rather than innovate new approaches?


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

The seminal book on Design Patterns, by four authors known as "the Gang of Four", tried to formalize common object oriented programming templates when it was released in 1994. In the intervening time the ideas presented in this book have become a source of standardization of programming practices, for better and for worse.

Another Example of Interactive Widgets

Here's another program, a little more complex than the last but nothing you can't handle. This one creates a window with five widgets: a number display (`QLCDNumber`), a horizontal slider (`QSlider`), a drop-down menu (`QComboBox`), a text label (`QLabel`) and a button (`QPushButton`). The slider lets you change the x-position of the window (and the number displayed), the dropdown menu lets you choose the range of the slider, and the reset button resets the slider and range to its initial settings.



The interface of ***Example 2***

`Slippery_Window.py`

```python
from PyQt5.QtWidgets import *
from PyQt5.QtCore import Qt
from PyQt5.QtGui import QFont
import sys

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

    def initUI(self):
        self.setFixedWidth(300)
        self.setFixedHeight(200)
        self.move(20,20)
        self.setWindowTitle('Slippery Window')
        self.lcd = QLCDNumber(self)
        self.lcd.setGeometry(0, 0, 300, 80)
        label_pos = QLabel(self.lcd)
        label_pos.setText("X Position")
        label_pos.setFont(QFont("Mono", 16, QFont.Bold))
        label_pos.setGeometry(10, 10, 100, 25)

        self.slider = QSlider(Qt.Horizontal, self)
        self.slider.setRange(0, 50)
        self.slider.setFixedWidth(290)
        self.slider.move(5, 80)
        self.slider.valueChanged.connect(self.sliderChanged)
        label_range = QLabel(self)
        label_range.setText("Range: ")
        self.dropdown = QComboBox(self)
        self.dropdown.addItems(["0-50", "0-200", "0-800"])
        self.dropdown.currentIndexChanged.connect(self.rangeChanged)
        label_range.move(105, 140)
        self.dropdown.move(150, 135)
        self.button = QPushButton(self)
        self.button.setText("Reset")
        self.button.clicked.connect(self.reset)
        self.button.move(150, 160)
        self.show()

    def sliderChanged(self):
        newval = self.slider.value()
        self.lcd.display(newval)
        self.move(newval, 0)

    def rangeChanged(self):
        item = self.dropdown.currentText()
        if item == "0-50":
            self.slider.setRange(0, 50)
        elif item == "0-200":
            self.slider.setRange(0, 200)
        elif item == "0-800":
            self.slider.setRange(0, 800)

    def reset(self):
        self.slider.setRange(0, 15)
        self.slider.setValue(0)
        self.dropdown.setCurrentIndex(0)

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

Example 2

What's New Here?

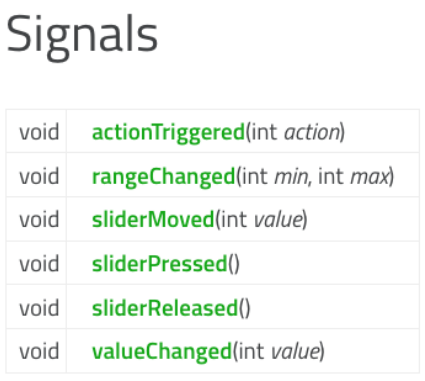
Line 11, 12 & 13 set up the dimensions and location of the window. Instead of using the `setGeometry()` method of `QWidget`, instead we use the `setFixedHeight()` and `setFixedWidth()` methods. Besides setting the height and width of the window, these methods also make the dimensions of the window ***fixed***. This means the window cannot be resized. The `move()` method just moves the window to screen coordinates (20, 20).

Line 15 & 16 create the QLCDNumber display and set its position, width, and height inside the window.

Line 17 creates the QSlider and configures it as a horizontal slider. You can also make vertically oriented sliders, depending on the first input argument you provide when creating the QSlider. The orientation options are stored in special variables inside one of the modules of PySide, you can find them at PySide.QtCore.Qt.Horizontal and PySide.QtCore.Qt.Vertical.

Line 18, 19 & 20 set the range of the slider, and sets a fixed width for it before positioning it within the window.

Line 21 connects the `valueChanged` signal of `QSlider` to our `sliderChanged()` method. Whenever the slider's value is changed, our method will be called and do its thing.



Signals

void	**actionTriggered**(int *action*)
void	**rangeChanged**(int *min*, int *max*)
void	**sliderMoved**(int *value*)
void	**sliderPressed**()
void	**sliderReleased**()
void	**valueChanged**(int *value*)

Signals of QAbstractSlider.

Line 24 creates the dropdown menu (`QComboBox`). And in ***Line 25*** three items are added to the menu using its `addItems()` method, which takes a list of strings as its input. ***Line 26*** connects the `currentIndexChanged` signal of `QComboBox` to our `rangeChanged()` method.

Try and figure out what the rest of the code does. Most of it is plain old Python and should be familiar to you. The rest are methods of specific Qt widgets that you can look up in the documentation.