LECTURE 18

GUI Programming Part 2

BASIC PYQT

Last lecture, we created a basic PyQt4 application which had a few buttons and a menu bar.


```
import sys
from PyQt4 import QtGui, QtCore

class Example(QtGui.QMainWindow):
    def __init__(self):
        super(Example, self).__init__()
        self.initUI()
    def initUI(self):
        ...
        exitAction = QtGui.QAction('Exit', self)
        exitAction.setShortcut('Ctrl+Q')
        exitAction.setStatusTip('Exit application')
        exitAction.triggered.connect(QtGui.qApp.quit)

        menubar = self.menuBar()
        fileMenu = menubar.addMenu('File')
        fileMenu.addAction(exitAction)
        ...
    def closeEvent(self, event):
        ...
app = QtGui.QApplication(sys.argv)
ex = Example()
app.exec_()
```

LAYOUT MANAGEMENT

Before we can add more components to our application, we need to talk about layout management in PyQt4.

The two options for managing the position of widgets are

- Absolute positioning
- Layout classes

Absolute positioning is as simple as calling the `move()` method on a widget with some arguments that specify a position (x,y). This can be impractical for a few reasons. First, applications might look differently on different platforms (or even using different fonts) as they won't scale. Secondly, changes in layout will be much more tedious.

LAYOUT MANAGEMENT

The other option is to use a layout class.

Layout classes automatically position and resize widgets to accommodate space changes so that the look and feel is consistent.

Every `QWidget` subclass may have a layout specified which gives a widget control over:

- Positioning of child widgets.
- Sensible default sizes for windows.
- Sensible minimum sizes for windows.
- Resize handling.
- Automatic updates when contents change (font size, text or other contents of child widgets, hiding or showing a child widget, removal of child widgets).

LAYOUT MANAGEMENT

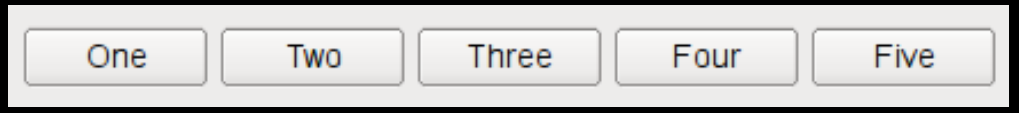
There are a large number of layout classes but the most common include:

- Box Layout (`QBoxLayout`): lays out widgets in a horizontal row (`QHBoxLayout`) or vertical column (`QVBoxLayout`) from left-to-right or top-to-bottom.
- Grid Layout (`QGridLayout`): lays out widgets in a 2-D grid where widgets can occupy multiple cells.
- Form Layout (`QFormLayout`): layout class for managing forms. Creates a two-column form with labels on the left and fields on the right.

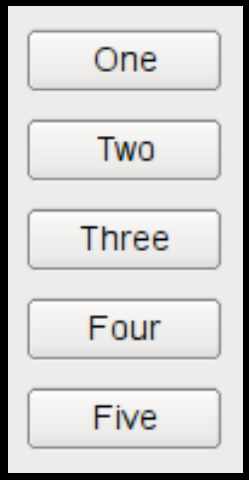
BOX LAYOUT

The `QBoxLayout` class lines up child widgets horizontally or vertically. `QBoxLayout` will take a given space and divide it up into boxes that contain widgets.

- `QHBoxLayout` makes horizontal rows of boxes.



- `QVBoxLayout` makes vertical columns of boxes.



BOX LAYOUT

Once you create a box layout and attach it to a widget, the following methods are used to add widgets and manage the space:

- `addWidget()` to add a widget to the `QBoxLayout` and set the widget's stretch factor.
- `addSpacing()` to create an empty box with a particular size.
- `addStretch()` to create an empty, stretchable box.
- `addLayout()` to add a box containing another `QLayout` to the row and set that layout's stretch factor.

Stretch factors indicate the relative amount of space that should be allocated to a block.

GRID LAYOUT

The Grid Layout class is the most universal and we will be using it for our application.

A grid is represented with multiple rows and columns and widgets can be attached to the grid by indicating the (row, column) space it should fill.

Create a grid layout with `QtGui.QGridLayout()`.

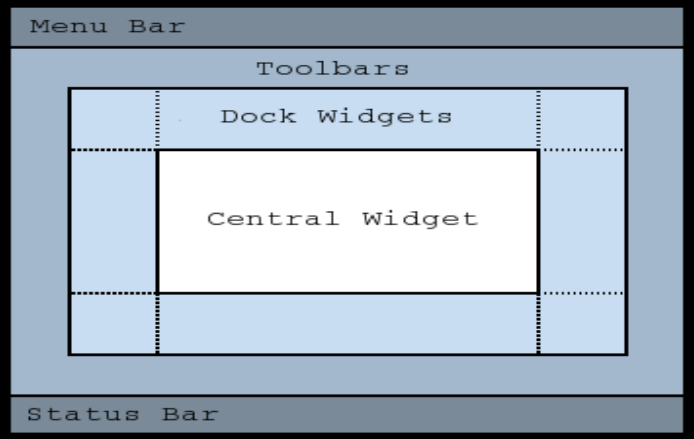
Attach widgets to the grid with `addWidget(QWidget, row, column)`.

You can also set the number of grid spaces that it should take up as well as stretch factors.

GRID LAYOUT

There's a lot to notice here. First of all, our main window has been renamed to `BlackjackApp`, which inherits from `QMainWindow`.

Recall that `QMainWindow` includes a default layout for traditional GUI components like a menu bar, status bar, tool bar, etc. The focal point of the application is stored in the Central Widget of the layout.




```
class CardTable(QtGui.QWidget):
    def __init__(self):
        super(CardTable, self).__init__()
        self.initUI()
    def initUI(self):
        hitButton = QtGui.QPushButton("Hit")
        stayButton = QtGui.QPushButton("Stay")
        grid = QtGui.QGridLayout()
        self.setLayout(grid)
        grid.addWidget(hitButton, 1, 0)
        grid.addWidget(stayButton, 0, 1)

class BlackjackApp(QtGui.QMainWindow):
    def __init__(self):
        ...
    def initUI(self):
        ...
        c = CardTable()
        self.setCentralWidget(c)
        ...
    def closeEvent(self, event):
        ...
```

GRID LAYOUT

Here, we create an instance of the `CardTable` class, which inherits from `QWidget`, and we make this the central widget of our main window.

```
class CardTable(QtGui.QWidget):
    def __init__(self):
        super(CardTable, self).__init__()
        self.initUI()
    def initUI(self):
        hitButton = QtGui.QPushButton("Hit")
        stayButton = QtGui.QPushButton("Stay")
        grid = QtGui.QGridLayout()
        self.setLayout(grid)
        grid.addWidget(hitButton, 1, 0)
        grid.addWidget(stayButton, 0, 1)

class BlackjackApp(QtGui.QMainWindow):
    def __init__(self):
        ...
    def initUI(self):
        ...
        c = CardTable()
        self.setCentralWidget(c)
        ...
    def closeEvent(self, event):
        ...
```

GRID LAYOUT

`BlackjackApp` has a built-in layout because it inherits from `QMainWindow`.

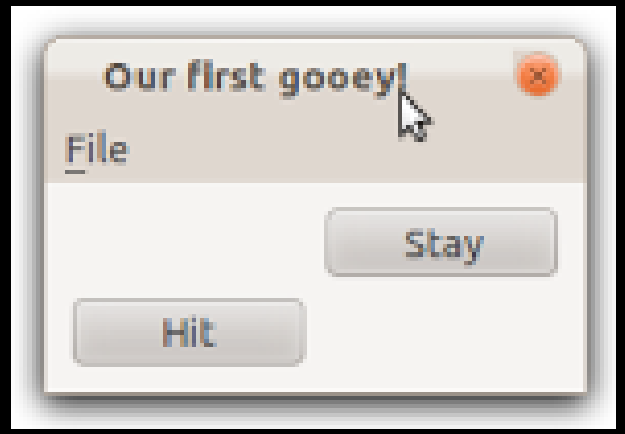
`CardTable`, however, is a plain widget. So we can associate a Grid layout with it.

On this grid layout, we'll attach two buttons at positions (1,0) and (0,1). These are basic buttons with no attached actions so they just look pretty.

```
class CardTable(QtGui.QWidget):
    def __init__(self):
        super(CardTable, self).__init__()
        self.initUI()
    def initUI(self):
        hitButton = QtGui.QPushButton("Hit")
        stayButton = QtGui.QPushButton("Stay")
        grid = QtGui.QGridLayout()
        self.setLayout(grid)
        grid.addWidget(hitButton, 1, 0)
        grid.addWidget(stayButton, 0, 1)

class BlackjackApp(QtGui.QMainWindow):
    def __init__(self):
        ...
    def initUI(self):
        ...
        c = CardTable()
        self.setCentralWidget(c)
        ...
    def closeEvent(self, event):
        ...
```

GRID LAYOUT




```
class CardTable(QtGui.QWidget):
    def __init__(self):
        super(CardTable, self).__init__()
        self.initUI()
    def initUI(self):
        hitButton = QtGui.QPushButton("Hit")
        stayButton = QtGui.QPushButton("Stay")
        grid = QtGui.QGridLayout()
        self.setLayout(grid)
        grid.addWidget(hitButton, 1, 0)
        grid.addWidget(stayButton, 0, 1)

class BlackjackApp(QtGui.QMainWindow):
    def __init__(self):
        ...
    def initUI(self):
        ...
        c = CardTable()
        self.setCentralWidget(c)
        ...
    def closeEvent(self, event):
        ...
```

QPALETTE

Now, we want to create a green card table which represents the focal point of our application.

To do this, we need to introduce the idea of a `QPalette`.

The `QPalette` class contains color groups for each widget state.

A palette consists of three color groups: *Active (keyboard focus)*, *Disabled (not in focus)*, and *Inactive (disabled)*. All widgets in Qt contain a palette and use their palette to draw themselves.

QPALETTE

Colors and brushes can be set for particular roles in any of a palette's color groups with `setColor()` for color and `setBrush()` for color, style, and texture.

Color roles include such roles as background, foreground, window text, button, button text, etc…

Calling, for example, `backgroundRole()` on a widget will return the current brush from the widget's palette that is being used to draw the role.

QPALETTE

Let's create a general widget to represent our card table.

Calling `palette()` on a `QWidget` object will return its currently associated `QPalette`. `QPalette` objects have a method `setColor()` which allows a `ColorRole` to be associated with a `QColor`.

`setAutoFillBackground` toggles the filling in of the background, which is transparent by default.

```
table = QtGui.QWidget()
p = table.palette()
p.setColor(table.backgroundRole(), Qt.Qcolor(34, 139, 34, 200))
table.setPalette(p)
table.setAutoFillBackground(True)
```

QPALETTE

```
class CardTable(QtGui.QWidget):
    def __init__(self):
        super(CardTable, self).__init__()
        self.initUI()
    def initUI(self):
        table = QtGui.QWidget()
        hitButton = QtGui.QPushButton("Hit")
        stayButton = QtGui.QPushButton("Stay")

        p = table.palette()
        p.setColor(table.backgroundRole(), Qt.QColor(34, 139, 34, 200))
        table.setPalette(p)
        table.setAutoFillBackground(True)

        grid = QtGui.QGridLayout()
        self.setLayout(grid)
        grid.addWidget(table, 0, 0, 2, 8)
        grid.addWidget(hitButton, 2, 6)
        grid.addWidget(stayButton, 2, 7)
```

We added a table widget as well as rearranged our grid, which is now 3x8.

Also, we've resized our main window to 400x600.

QPALETTE

Yay!

CARD TABLE

To demonstrate the use of our card table during the course of a game, let's place some sample cards. Each of the card images are stored locally as .png files.

Placing a card on the grid can be done with the following code:

```
dcard1 = QtGui.QPixmap("b2fv.png")
dlbl1 = QtGui.QLabel(table) # QLabel provides a text or image display.
dlbl1.setPixmap(dcard1)
grid.addWidget(dlbl1, 0, 1, 1, 1) # Place at (0,1) in grid, using 1 space
```

CARD TABLE

```
# Shrink outer columns to 5 pixels.
grid.setColumnMinimumWidth(0,5)
grid.setColumnMinimumWidth(7,5)
# Add 4 cards at locations (0,1),
# (0,2), (1,5) and (1,6).
grid.addWidget(dlbl1, 0, 1, 1, 1)
grid.addWidget(dlbl2, 0, 2, 1 ,1)
grid.addWidget(plbl1, 1, 5, 1, 1)
grid.addWidget(plbl2, 1, 6, 1, 1)

# Move buttons to the left one column
grid.addWidget(hitButton, 2, 5)
grid.addWidget(stayButton, 2, 6)
```

CARD TABLE

Not perfect, but we're getting there.

Now, let's clean it up, add a few more elements and then attach the game logic.

SO FAR

So far this lecture, we covered the following topics:

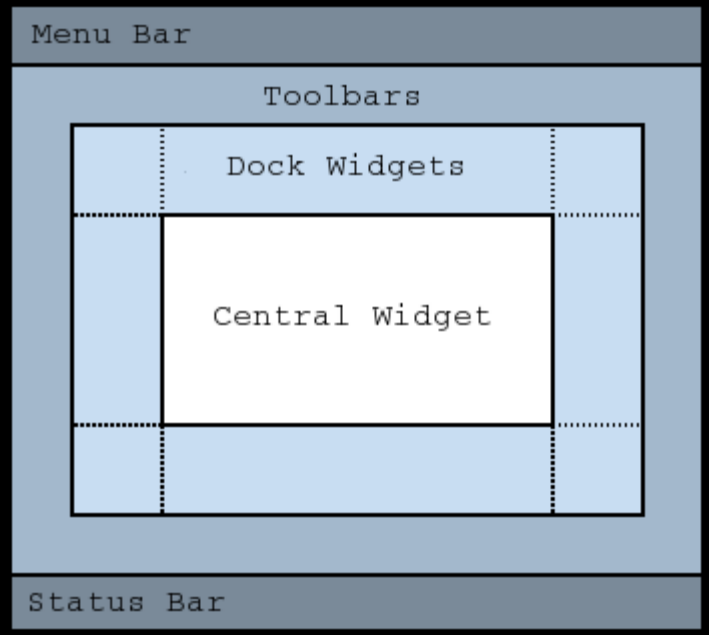
- Layout Management
  - Box
  - Grid
- Coloring widgets with `QPalette`.
- Creating text/image widgets with `QLabel`.

This gives us everything we need to basically create a static picture of what gameplay should look like. However, we want a GUI that is dynamic and changes with every turn of the game.

STATUS BAR

First thing we need to do is create a status bar for our main window and attach it to the layout.

We created a menu bar and an instance of our `CardTable` class (which inherits `QWidget`) was attached to the central widget location.



The only thing we need to do inside of our `BlackjackApp` class is the following:

```
statusBar = self.statusBar()
```

This will automatically create a basic status bar if there is none attached and return it.

GAME LABELS

The next thing we need to do is add `QLabel` widgets to our central widget which display to the user the number of games they have played and won.

Previously, we saw how we can create `QLabels` which contain images but this time we just want text.

Inside of `CardTable`, which has an attribute that tracks games won by the user:

```
games_won_lbl = QtGui.Qlabel("Games Won: " + str(self.gamesWon))
```

To change the text associated with the label, use the `setText()` method.

ADDING GAME LOGIC

The very last thing we need to do (although it's a pretty big step) is add the game logic so that our GUI is actually playable. Our application will consist of the following:

- card.py: `Card`, `Deck`, and `Hand` class definitions.
- blackjack.py: `BlackjackGame` class definition.
- pyqt_blackjack.py: our evolving GUI.

CARD.PY

The card.py version we'll be using is identical to the one we've used in previous examples except for one change: there is now a method for the Card class called `filename()` which constructs the .png filename for the card based on its suit and rank.

BLACKJACK.PY

The blackjack.py module represents a simple, class-based version of the blackjack game logic we've used in previous examples. It contains a definition for the `BlackjackGame` class with the following methods:

- `start_new_game()`
- `play_game()`: execute a hit phase.
- `stay_game()`: execute a stay phase to end the game.
- `check_end()`: check for automatic win conditions.
- `check_winner()`: find the winner based on the greatest hand value which doesn't exceed 21.

PYQT_BLACKJACK.PY

Our previous version of `CardTable`, our central widget, contained methods for initializing the UI but not much else. This time we have the following methods in addition:

- `start_new_game()`: clear cards from table, update stats, call `game.start_new_game()` and display new cards.
- `hitAction()`: Call `game.play_game()` and display new card.
- `stayAction()`: Reveal hidden dealer card and call `game.stay_game()`. Display all new dealer cards.
- `alert_game_end()`: Display a pop-up message to user telling them who won and asking if they'd like to play again.

PYQT_BLACKJACK.PY

While we have the methods down, we need to connect it all together and specifically connect it to user actions. Here's what we basically want:

User clicks 'hit' button → `hitAction()`

User clicks 'stay' button → `stayAction()`

User clicks 'yes' to play again → `start_new_game()`

User clicks 'no' to play again → `QApplication.quit()`

PYQT_BLACKJACK.PY

To create the first two associations, we can use the signal and slot mechanism.

Signals and slots are used for communication between objects. A *signal* is emitted when a particular event occurs. A *slot* can be any Python callable. A slot is called when a signal connected to it is emitted.

```
hitButton.clicked.connect(self.hitAction)
stayButton.clicked.connect(self.stayAction)
```

When each button is clicked, it is associated with a method to be executed.

PYQT_BLACKJACK.PY

For the other two associations, we can use the `QMessageBox` method we've seen before.

```
def alert_game_end(self, gameFlag):
    win_message = "YOU WIN! Play again?"
    lose_message = "Dealer wins. Play again?"
    if gameFlag:
        self.gamesWon = self.gamesWon + 1
        msg = win_message
    else:
        msg = lose_message
    self.gamesPlayed = self.gamesPlayed + 1
    reply = QtGui.QMessageBox.question(self,
                'Game Over', msg,
                QtGui.QMessageBox.Yes |
                QtGui.QMessageBox.No,
                QtGui.QMessageBox.Yes)
    if reply == QtGui.QMessageBox.Yes:
        self.start_new_game()
    else:
        app.quit()
```

GOOEY BLACKJACK

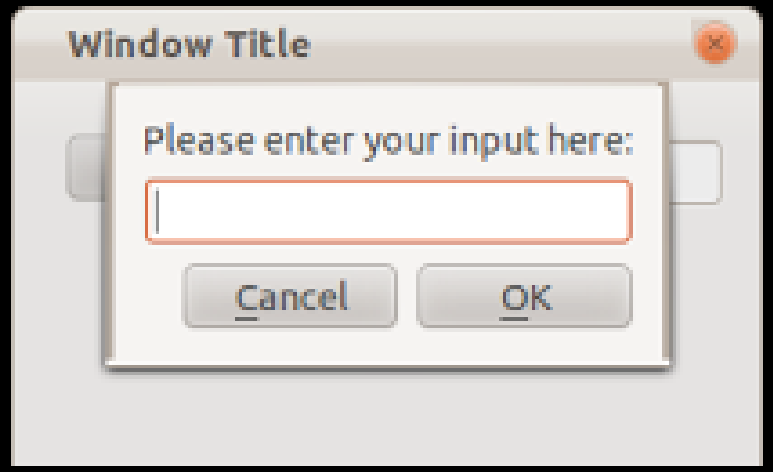
Let's check out a demo.

QINPUTDIALOG

So far, we've seen how the user can send signals to the application.

To allow a user to send text to the application, we can use a QInputDialog widget.

```
text, ok = QtGui.QInputDialog.getText(parent, 'Message Title',
'Please enter your input here: ')
```



Exercise idea: Modify Gooey Blackjack to include the winner's list. Users can enter their name and display the users who have won the most games in one sitting.

USER INPUT

Other more exotic kinds of user input are implemented by the following widgets:

- `QColorDialog`: provides a dialog widget for specifying colors.
- `QFontDialog`: allows user to select font.
- `QFileDialog`: allows users to select files or directories.
- `QPrintPreviewDialog`: previewing and configuring page layouts for printing.
- `QProgressDialog`: provides feedback on time-consuming operations.

QWIDGET DERIVATIONS

There are many, many kinds of widgets we haven't covered.

Browse the subclasses for `QWidget` here.

You can also create custom widgets using the `QtGui.QPainter` class.

`QPainter` provides methods for the drawing that GUI programs require. It can draw everything from simple lines to complex shapes. It can also draw aligned text and pixmaps.

`QPainter` can operate on any object that inherits the QPaintDevice class.

FURTHER READING

Check out this link for a little tutorial on custom widgets.

Check out this link for a small tetris game demo.

Very thorough PyQt source here.