Get started with Qt GUI Programming

By Suvish V.T.

1. Introduction to Qt

Qt is a framework to create cross-platform applications. Using Qt you can create amazing GUI applications quickly and fairly easily. You might have heard about Visual Studio or X code to create applications in Windows and Mac respectively...Qt is similar to these tools in that it helps you to design and code your application. But, the real advantage of Qt lies in the fact that your application can be made to run on a variety of operating systems without you having to change your code. Your application will run on Windows, Mac and Linux in pretty much the same way. Talk about killing three birds with one bullet!

To get started with using Qt, you have to download it from http://qt.nokia.com/downloads. You can go for the LGPL version or the commercial version (both of them can be used to create proprietary applications, if necessary) and choose a download for your OS.

Once you get the package installed, you get a number of Qt developer tools at your disposal. Lets examine each one of these:

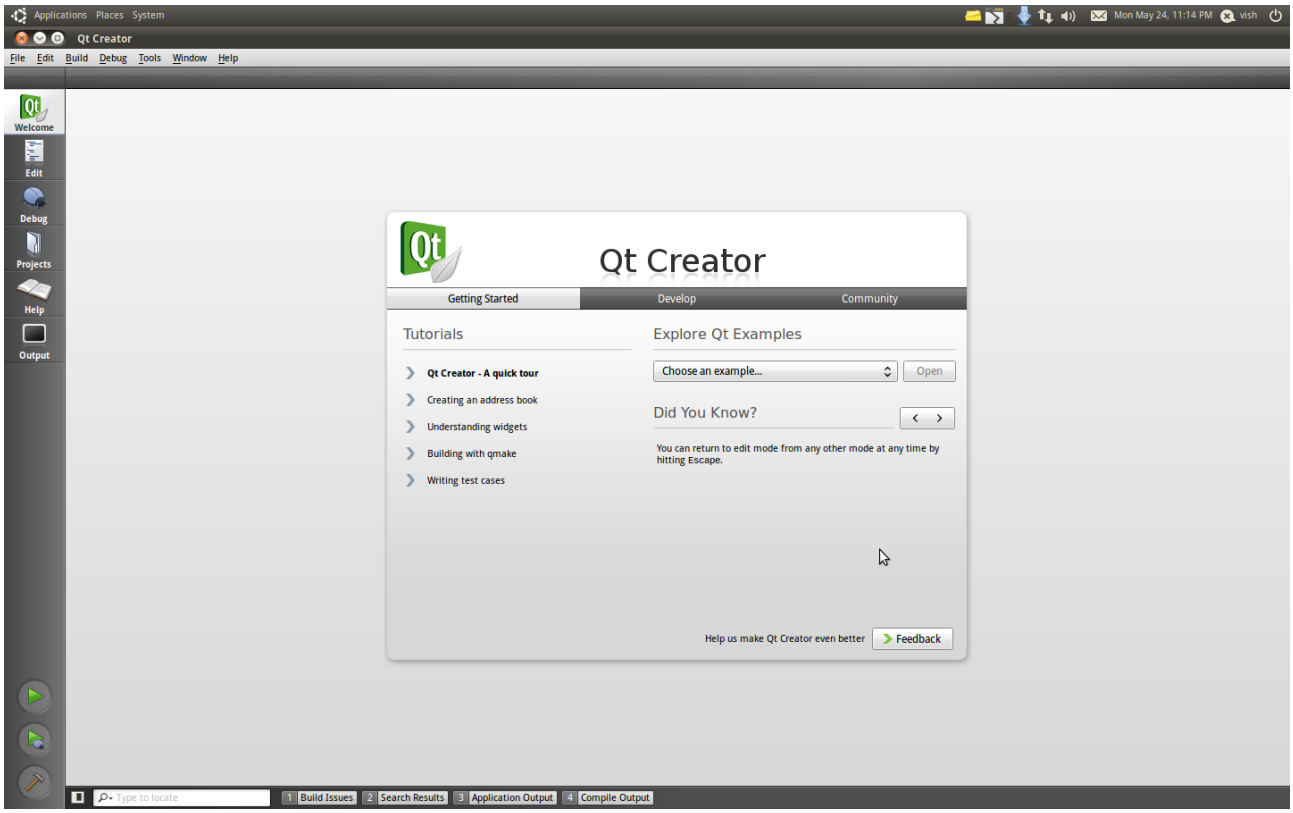
- **Qt Creator**: This is the main IDE (Integrated Development Environment) which we will be using for Qt development. It is similar to Visual Studio or X Code. It includes almost all tools for you to develop Qt applications.
- **Qt Assistant**: Assistant is used for viewing the help files and documentation related to Qt. You will be using it quite a lot for reference while learning Qt. Note that the Assistant is integrated into Qt Creator (the Help tab), so you don't need to launch it separately every time.
- **Qt Designer**: This is used to design the User Interfaces (Forms) of your application. Even the Designer is integrated into Qt Creator.
- **Qt Linguist**: Linguist is a tool for aiding the translation of your application into various languages.

Besides these standard applications, there is also the Qt Command Line in Windows, which is basically cmd.exe with all the PATH values set. If a Qt command is to be executed in Windows (like qmake, mingw32-

make, etc) you need to use this tool.

2. My first application!

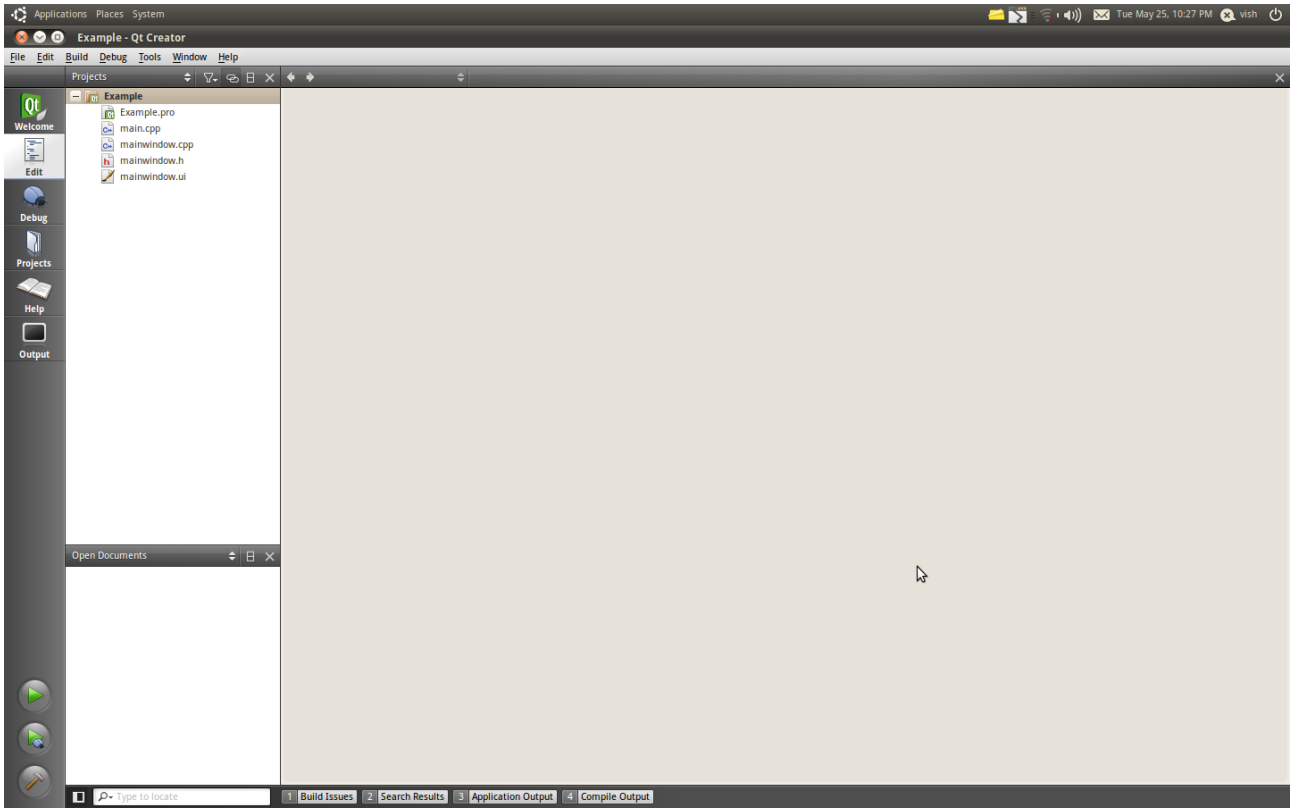
So, lets jump right into it. We'll go ahead and make our first Qt application. Fire up Qt Creator and take a look at its interface. You'll find something like this:



That is the Welcome Screen which gives you quick links to your recent projects or tutorials. We'll create a project by going to File>New file or project. In the dialog, we select “Qt4 GUI Project”. Next they ask you to name your project and the folder in which you want to store it in. I've named my project “Example” for demonstration purposes. Next you will be asked to select all the required modules for your project. The “QtCore” and “QtGui” modules are already selected, and for our project, that is more than enough. The other modules you can include later as and when you need it.

Then, you are asked to enter basic information about your source code files like, the base class name, form name etc. Its best to stick with

the defaults now, which is mostly a variation of “MainWindow”. Then Next, and then Finish, and so you have a basic Qt program set-up ready:

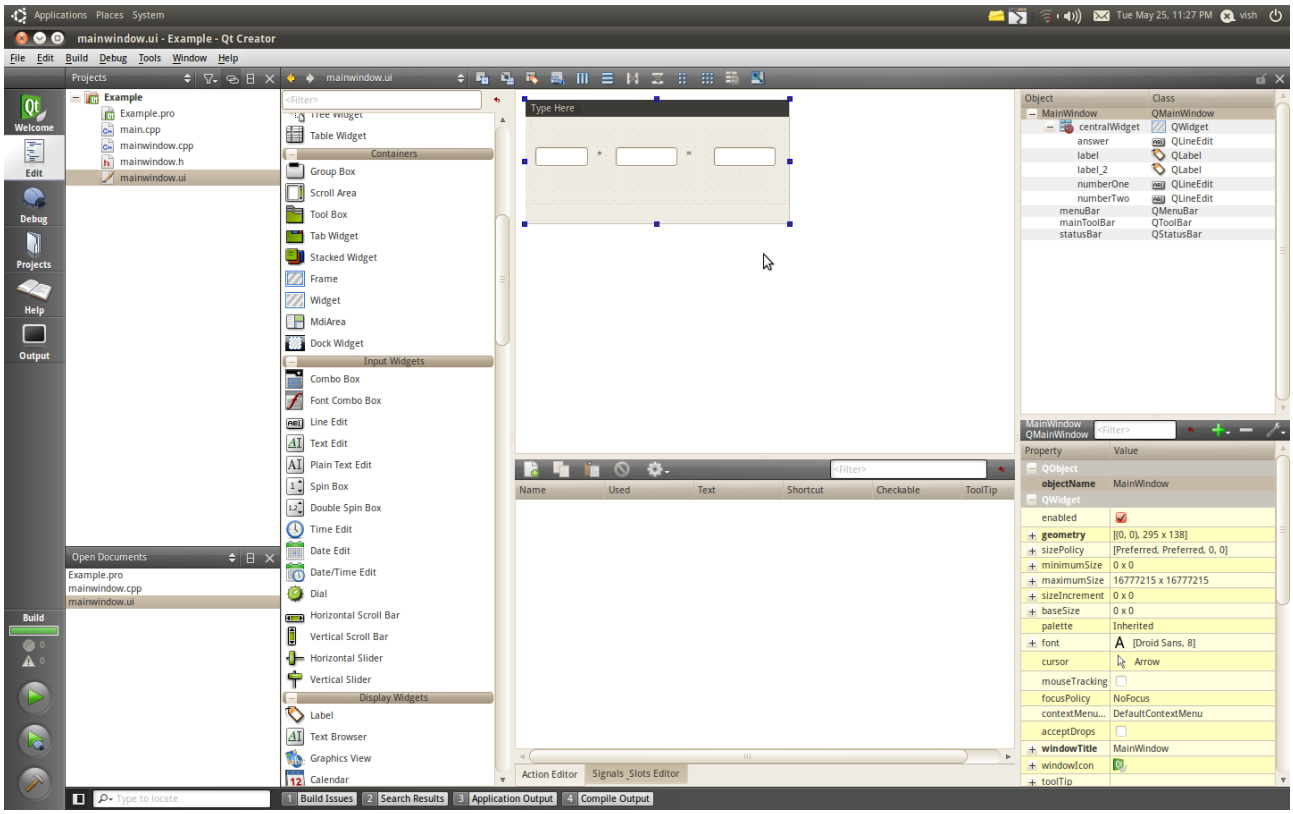


Now, if you click on the play button at the bottom left of the window, it will compile and run the program...voilà! A blank window appears, and that is your basic GUI application running!

A quick intro to the files used in the project. The first file is the .pro file which contains information on the entire project and the files in it. It is used by the program “qmake” to create a “makefile”. A makefile is a file which contains information on how to compile the program. It is interpreted by the make program which issues directions to the compiler for compiling and linking the project. There is no need to understand all that completely at this stage...we'll gradually get hang of it as we proceed. Now the next file is “main.cpp”. This file contains the Main function. As such, it is the entry point of the program. You can double click it to view its contents, we'll analyse the function later. Then, we have the “mainwindow.cpp” and “mainwindow.h” which is the header and definition of the main class used in the program, MainWindow. The last file is interesting, it is the “mainwindow.ui” file. This file is the user interface file (a form file, in Visual Basic language). When you double

click it, the Qt Designer opens and you can edit your GUI interactively.

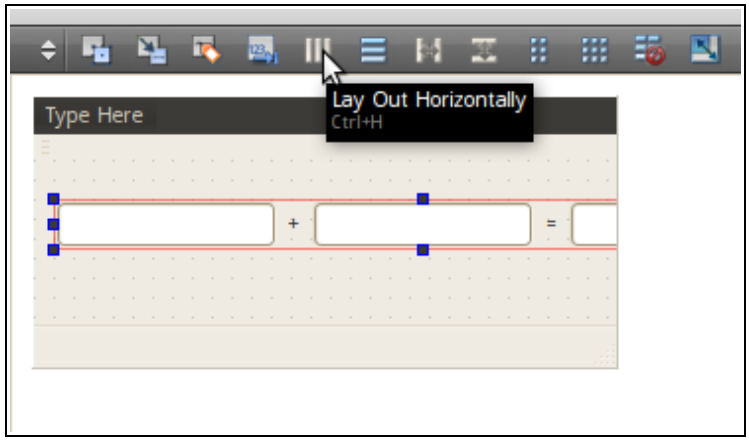
Well, lets do something with our new Application Skeleton. How about we make a simple program that adds two user input numbers and gives output? First, we will make the interface. Double click the .ui file. Play with the interface and try to make something like the below screen-shot:



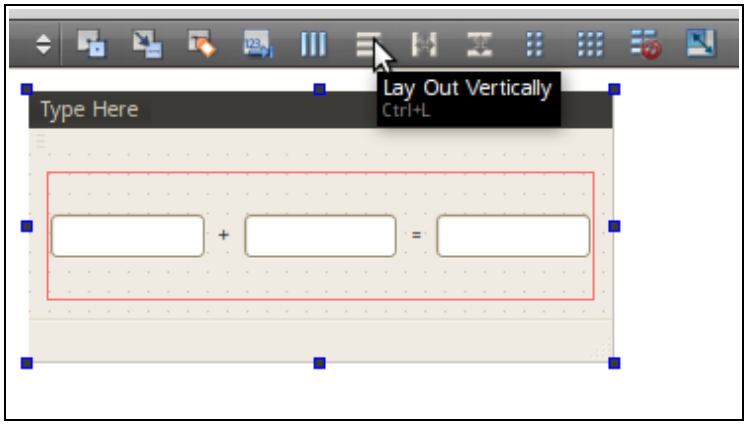
You can drag the Line Edit and Label widgets from the Tool Box to the Form to get something similar to the above screen-shot. Also, you can resize the form to something that fits. The first two LineEdits are used for input and the third one for output. Our goal is to update the third LineEdit with the addition of the first two LineEdits whenever they are changed. Now save it and view the result by pressing the “Play” button or Ctrl+R. The application will compile and run to give you what you designed on the interface. Of course, they do nothing now, but we'll add code for it later to run as intended.

Before we add the code, lets clean up a bit on the interface front. You may have noticed that the widgets in the interface are floating free, and if you resize the window, the widgets will remain in the same place. To correct this problem and make sure the widgets are lined up perfectly, we

need to understand the concept of **Layouts**. Layouts are a way to group widgets in the interface and to let them make good use of the available space. The Designer has four in-built layouts: Horizontal, Vertical, Form and Grid. First, we have to apply the Horizontal layout on all the widgets that we created. To do this, select all the widgets by Ctrl-clicking each one and selecting the “Lay Out Horizontally” button in the toolbar. See the screen-shot below for reference:



Notice that there appears a red border to the widgets as soon as the layout is applied. Also the widgets are lined up perfectly but runs outside the window border. This is because, even though the widgets **in** the interface is put in a layout, the interface itself isn't. The interface, as we have been calling it all this while, is nothing but a QWidget object. This means that it is also a container widget that holds other widgets. As such, even it needs to have a layout. To apply a layout to the container widget, click on the free space outside your current layout. This will select the whole container widget. Now, click on “Lay Out Vertically”. A screen-shot:



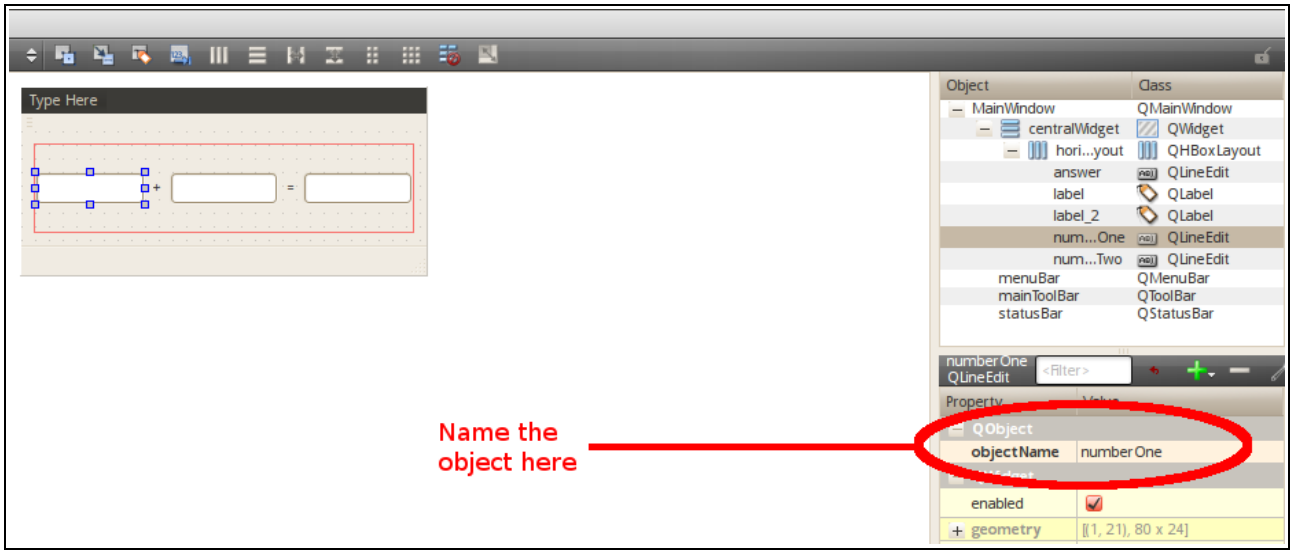
See how its all nice and perfect now? This is basically how an application

is laid out in Qt. A small note, here we have used the Vertical layout for the container widgets, but it will look the same even if you use another layout. This is because there is only one child layout in the container layout. So it does not matter if you use vertical or horizontal layout, it will stay in the same position. Of course, this will not apply in more complicated layouts.

A last thing before coding, we need to name our LineEdit widgets properly so that they are recognizable when we code. Doing this is simple, just click on each widget in turn and use the property editor to give names as follows:

- numberOne
- numberTwo
- answer

Screen-shot for reference:



Now that is out of the way, let's start coding!

3. Coding in Qt

Let's have a look at main.cpp:

```
#include <QtGui/QApplication>
#include "mainwindow.h"

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.show();
    return a.exec();
}
```

The headers: it includes QApplication from QtGui. QApplication is the class which defines your application. All Qt applications have one QApplication as the start point for the program. Even if you have many windows in your application, you only have one QApplication. After this, we include mainwindow.h. This is the header of your main class; we'll come back to that later. Main.cpp contains the main function which has the program's argument count and argument list as parameters. In it we make a QApplication object “a” with the program's argument variables as parameters. It is only after this that we make an object of MainWindow and call its show() method. The show() method makes the application window visible. In the end we return a.exec(). Now what does this mean? This means that we are passing control of the program from main() to QApplication. The exec() method finally starts the application.

We will now learn and modify our application's class. The application's class is MainWindow and it's header is in mainwindow.h. Open it up:

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>

namespace Ui {
    class MainWindow;
}

class MainWindow : public QMainWindow {
    Q_OBJECT
public:
    MainWindow(QWidget *parent = 0);
    ~MainWindow();

protected:
    void changeEvent(QEvent *e);

private:
    Ui::MainWindow *ui;
};

#endif // MAINWINDOW_H
```

The #ifndef and #define are preprocessor directives for the Qt build system, so we can ignore that for now. You see that it includes QMainWindow which is a pre-made Qt class for Application main windows. There is a namespace “Ui” declared in which our class is put. This shows that MainWindow is part of the user interface of the application. The declaration of MainWindow class inherits from QMainWindow, so it is like a reimplementation of QMainWindow. The constructor of MainWindow has a parameter *parent, which is set to 0. This means that our MainWindow has no parent class by default and it is the topmost window in the application. Inside the class, a Q_OBJECT macro appears. This “macro” tells the Qt system that the MainWindow class is also a QObject. Only one more thing to take note here, the MainWindow class is also having a pointer of its own type “ui”. It is using this pointer that we access the user interface elements in the application like LineEdit or buttons. We'll see how to do that shortly. Note: if you don't understand some things here, its all right, as its not needed right now for you to create your application. You'll get it all in time.

Onto the implementation then, mainwindow.cpp:

```
#include "mainwindow.h"
#include "ui_mainwindow.h"

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
}

MainWindow::~MainWindow()
{
    delete ui;
}

void MainWindow::changeEvent(QEvent *e)
{
    QMainWindow::changeEvent(e);
    switch (e->type()) {
    case QEvent::LanguageChange:
        ui->retranslateUi(this);
        break;
    default:
        break;
    }
}
```

So here is where we are going to do serious work. The implementation includes mainwindow.h the header, and a curious file called “ui_mainwindow.h”. Now if you look closely, this file is not part of our original project. What happens is, the mainwindow.ui file is converted to a header file which contains Qt code for all your user interface elements. So, Qt does all the grunt work of manually coding the interface while you create it using the Designer!

The first thing to notice is the constructor, which passes the “parent” parameter to both itself and it's base class constructor. In that, the previously discussed “ui” pointer is used. It is used to access the setupUi method, which sets up all the user interface elements in their correct place. There is a destructor which deletes the ui pointer. The last thing in the default setup is a method called “changeEvent”. This function is used in this context for translation purposes, so it can be ignored for now. So, this sets the stage for us to add our own coding, and get everything working. To do that, we have to learn about a little Qt speciality called “Signals and Slots”.

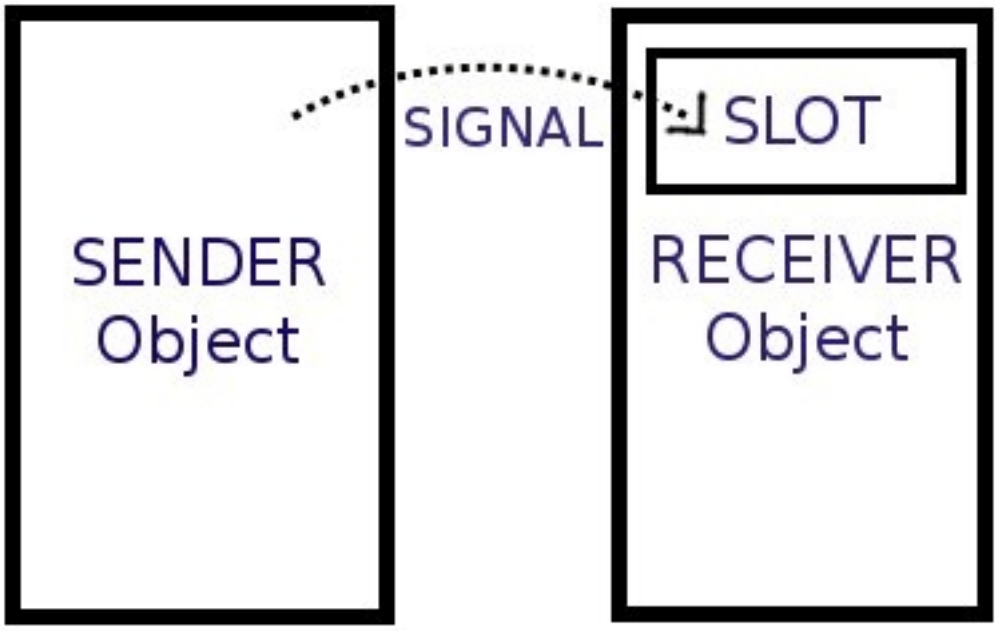
SIGNALS AND SLOTS

Signals and slots are the mechanism by which Qt achieves the unification of GUI and code. It is both similar and very dissimilar to Java's Event driven model. Each user interface component in Qt (Buttons, ListWidgets, LineEdits, etc) can emit a wide variety of Signals (like clicked(), triggered(), etc). These signals are then connected to Slots by the programmer, which are nothing but methods or functions in the program. So for example, we can connect the clicked() signal of a button to a Slot (method) that does something when the button is clicked.

In our program, we will connect the textChanged() Signal with a Slot called numberChanged(). To achieve this, let's check out the connect() function:

```
bool connect(SENDER,SIGNAL,RECEIVER,SLOT);
```

In the connect function, we specify four parameters to connect a signal to a slot. The SENDER is an object or UI element which emits the SIGNAL. The RECEIVER is an object which receives the SIGNAL into the SLOT, which is a method of the receiving object. The below diagram should make everything clear:



So, now we have the sender (numberOne, numberTwo), the Signal

(textChanged) and the receiver (MainWindow Object). But, we still have to create a Slot for our main logic. Let us code this Slot.

In Qt Creator, open mainwindow.h. At the end of the MainWindow class declaration, add this:

```
private slots:
    void numberChanged();
```

This will declare the numberChanged() function as a private slot. This kind of declaration is similar to how you declare members of a class as public or private.

Now open the mainwindow.cpp file and add the following at the end:

```
void MainWindow::numberChanged() {
    int numOne=ui->numberOne->text().toInt();
    int numTwo=ui->numberTwo->text().toInt();
    int addition=numOne+numTwo;
    ui->answer->setText(QString::number(addition));
}
```

In this method, the logic is as follows: the number from both the LineEdits (numberOne and numberTwo) are added and then the answer is put in the “answer” LineEdit. That's a simple enough logic, but there is a complication here. QLineEdits provide the text in them as QStrings, a string type provided by Qt, not integers. So, we need to convert them into integers before adding them, and into QStrings before putting the answer back into LineEdit. Fortunately, Qt provides us with methods to do just that.

Notice how the UI elements are accessed using the “ui” pointer. We retrieve the current text of the LineEdits using their text() method. The text is returned as QString. We use the toInt() method of QString to convert it into integers. We add them into the “addition” integer. Then we put the addition into the “answer” LineEdit, using a method called setText(). Again, since setText() only accepts QStrings as arguments, we convert the integer into QString using a static method of QString, number(). number() accepts the integer and returns a QString. Thus we now also have the Slot ready. There is only one thing to do before we connect our Signals and Slots.

LineEdits also accept alphabetical input in addition to numerical

input. Since alphabetical input will produce unexpected results in our application, we need to put a validation scheme on our LineEdits to accept only numbers. Add the below code to the MainWindow constructor right after the line ui->setupUi(this); :

```
    QIntValidator *validator=new QIntValidator(0,100,this);
    ui->numberOne->setValidator(validator);
    ui->numberTwo->setValidator(validator);
    ui->answer->setReadOnly(true);
    ui->answer->setText("0");
```

QIntValidator is a class which provides a way to ensure a QString contains an integer within a specified range. An object pointer of QIntValidator class is made and initialized with values 0 and 100. These are its minimum and maximum values. Also a third parameter is passed, “this”, which is the parent of the QIntValidator class, set to be our class itself. After this, we use the setValidator() method of our LineEdits and pass the QIntValidator object pointer to it as an argument. Validation is thus set up. We also set the answer widget as read-only, and insert an initial value “0” to it.

Now we connect the widgets. Add the below text at the end of the constructor i.e. after the ui->answer->setText(“0”); line:

```
    connect(ui->numberOne,SIGNAL(textChanged(QString)),this,SLOT(numberChanged()));
    connect(ui->numberTwo,SIGNAL(textChanged(QString)),this,SLOT(numberChanged()));
```

The connect() function above is taking 4 parameters, the first is the SENDER, numberOne or numberTwo, the second is the SIGNAL, textChanged(), the third is the receiver object, in this case our object itself, and the fourth is the Slot we made, numberChanged(). In this way, whenever the number in either of the LineEdits are changed, its addition is updated into the “answer” LineEdit. You can compile the application now to run the final output. Type numbers into it and watch the addition happen. Also test the validation.

Thus we now have a basic understanding of how Qt works and how you use it to create interfaces and code for them. You can now experiment further and use the Qt help system to create your own cool applications!

NB: You can get the source code of the above application ready made here.