
COMP2542 Lab Exercises

Release 0.2

Nick Efford

13 January 2016

1	Getting Started With Qt	1
1.1	Configuring SoC Linux PCs For Qt & C++	1
1.2	Configuring SoC Linux PCs For PyQt	2
1.3	“Hello World!” in Qt via C++	2
1.4	“Hello World!” in Qt via Python	3
2	Creating & Configuring Widgets	5
2.1	Simple Buttons	5
2.2	Spin Boxes	6
2.3	Single-Line Text Input	7
2.4	Multi-Line Text Input	7
3	Programming Widget Layout	11
3.1	Experimenting with QHBoxLayout	11
3.2	Developing a Nested Layout	12
3.3	Using QFormLayout	14
3.4	Using QGridLayout	15
4	Applications as Classes	17
4.1	Subclassing QWidget in Python	17
4.2	Subclassing QWidget in C++	19
5	Handling Interaction With Signals & Slots	21
5.1	Using Built-in Signals & Slots in C++	21
5.2	Using Built-in Signals & Slots in Python	22
5.3	Custom Slots in C++	22
5.4	Custom Slots in Python	24
5.5	Other Things to Try	25
6	Creating Custom Widgets in Qt	27
6.1	A Custom Widget For Text Input	27
6.2	Composite Widgets	27
6.3	Adding Custom Signals & Slots	28
7	Creating Dialogs & Application Windows	31
7.1	Creating Dialogs by Subclassing QDialog	31
7.2	Creating Application Windows by Subclassing QMainWindow	34
8	Using Qt’s Interface Designer	39
8.1	Getting Started	39
8.2	Simple Layout Using QWidget	40

8.3	Using Designer-Generated Forms	41
8.4	A More Complex Layout	44
9	Getting Started With Android Development	47
9.1	Preparation on SoC Linux Machines	47
9.2	Further Configuration	48
9.3	Creating an Android Project	48
9.4	Running in an Emulator	51
9.5	Running on a Real Device	51
9.6	Further Reading	52
10	Developing a Simple Android App	53
10.1	Widget Creation & Layout	53
10.2	Adding an Event Handler	54
10.3	Specifying Strings Properly	55
11	Linking Activities Using Intents	57
11.1	Modifying The Existing App	57
11.2	Creating The Intent	58
11.3	Creating The Secondary Activity	58
11.4	Testing The App	58
11.5	Other Things To Try	59

GETTING STARTED WITH QT

Author Nick Efford

Contact N.D.Efford@leeds.ac.uk

Status Final

Revised 2015-10-05

The goal of these exercises is to introduce you to the basic workflow for programming Qt GUIs. We will take you through the steps needed to run a minimal “Hello World” application using Qt, using both C++ and Python.

If you wish to set up your own PC to do these exercises, you will need to install Qt 5 and PyQt 5. You can obtain these from the following locations:

Qt 5	http://www.qt.io/download/
PyQt 5	http://www.riverbankcomputing.co.uk/software/pyqt/download5

If you are a Linux user, first check your distribution’s package manager to see if binary distributions of the above are available, as this will generally be the easiest way of installing the software.

If you are a Mac user, using a package manager such as [Homebrew](#) or [Fink](#) will be an easier way of getting a working installation than compiling the libraries from scratch yourself.

If you are a Windows user, note that the Windows release of PyQt 5 from the link above already includes Qt 5, so a separate download is not necessary in that case.

1.1 Configuring SoC Linux PCs For Qt & C++

SoC Linux PCs provide multiple versions of the Qt framework. Some configuration is therefore necessary to ensure that the correct version is used for COMP2542.

Note: The instructions below should be carried out before your first use of Qt on SoC Linux PCs. You will only need to perform these steps once and you will not need to do them on your own PC.

1. In a terminal window, enable the appropriate version of Qt by entering the following command:

```
module add qt/5.3.1
```

Then enter `module list` and check that the list of ‘currently loaded modulefiles’ includes `qt/5.3.1`.

2. Start the text editor of your choice (e.g., *gedit*) and edit the file `.bashrc` in your home directory. (Note the ‘dot’ at the start of the filename!) Add the aforementioned `module add` command to the bottom of the file, exactly as shown above, taking care not to interfere with any existing contents, then save the file.
3. **Start a new terminal window.** In this new window, enter the following:

```
qmake -v
```

Seek assistance if you don't see `QMake version 3.0` and `Using Qt version 5.3.1` displayed.

1.2 Configuring SoC Linux PCs For PyQt

The `module add` command is also needed if you wish to use Qt via its Python bindings on SoC Linux machines.

1. Start up a text editor and edit `.bashrc` again. Create an **alias** like the following:

```
alias pyqt='module add python/3.4.3 sip/4.16.9'
```

Make sure you've got the version numbers correct here. Then save the file.

2. **Start a new terminal window.** When you wish to use PyQt, all you need to do now is enter `pyqt`. Do this now, then run the Python interpreter by entering `python3`. In the interpreter, enter the following command:

```
>>> import PyQt5.QtCore
```

If this works without any errors, you should be ready to use PyQt!

Press `Ctrl+D` to exit the Python interpreter and use this terminal window for the remaining exercises.

Warning: Don't actually put the `module add python/3.4.3` command in your `.bashrc` file! It appears that this can create problems with the login process on our Linux machines.

1.3 “Hello World!” in Qt via C++

Qt is a C++ framework, so the most direct and efficient way of using it is to write your application in C++.

1. In your terminal window, use the `mkdir` and `cd` commands to create a directory for this exercise and then move into it. Use these commands a second time to create and move into a subdirectory called `hello`.
2. Start up a text editor of your choice (e.g., *gedit*) in this `hello` directory and create a file called `hello.cpp`. Add the following C++ code to this file:

```
#include <QtWidgets>

int main(int argc, char* argv[])
{
    QApplication app(argc, argv);

    QLabel* widget = new QLabel("<h1>Hello World!</h1>");
    widget->show();

    return app.exec();
}
```

Be sure to type the code exactly as shown. Save the file when you are done.

Warning: Do not copy and paste code from this web page into your editor, as this can lead to compilation errors due to issues with character encoding. Typing the code in yourself will, in any case, help you to become more familiar with Qt.

If there is anything here that you don't understand, consult the `line-by-line` explanation of the code.

3. Return to your terminal window. Check that you are in the `hello` directory and that the `hello.cpp` file is there, then create a **Qt project file** using the following command:

```
qmake -project
```

This should create a new file called `hello.pro`. Edit the file so that it looks like this:

```
TEMPLATE = app
TARGET = hello
INCLUDEPATH += .
QT += widgets

# Input
Sources += hello.cpp
```

The line you need to add is highlighted. Qt 5 is organised in a modular fashion and this line ensures that the widgets module will be accessible when compiling and linking your code.

4. Now create platform-specific build instructions for the project by entering the following command:

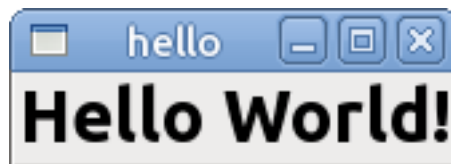
```
qmake hello.pro
```

Use `ls` to check the directory contents again. You should see a new (and fairly large) file called `Makefile`. This is a standard `makefile` defining the commands needed to compile the program and link it with the Qt libraries. Unlike the `.pro` file, this is platform-specific and will need to be regenerated if you copy your project files to another machine that has a different installation of Qt.

5. To build the program, enter `make` (not `qmake`). On Linux machines, this will create an executable file called `hello`. Run this now with the following command in the terminal window:

```
./hello
```

You should see a small window appear, looking something like the example shown below.



Note: If your code fails to compile, look in `hello.pro` to check whether you added the `QT += widgets` line or not.

1.4 “Hello World!” in Qt via Python

Using the Qt libraries via their Python bindings is slightly less efficient, but you usually won't need to write quite as much code. (This benefit won't be apparent here, but you will notice it for more complex GUIs.)

1. In the terminal window, use `cd ..` to move up one level from the `hello` subdirectory, back into the directory that you originally created for this exercise.
2. In this directory, use your preferred text editor to create a new file called `hello.py`, containing the following code:

```
import sys
from PyQt5.QtWidgets import QApplication, QLabel

app = QApplication(sys.argv)

widget = QLabel("<h1>Hello World!</h1>")
widget.show()

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

Compare this with the C++ code from the previous task. Barring obvious language syntax differences, the two versions are essentially identical - but note the use of `exec_` rather than `exec` in the Python version.

3. Run your program with the following command in the terminal window:

```
python3 hello.py
```


CREATING & CONFIGURING WIDGETS

Author Nick Efford

Contact N.D.Efford@leeds.ac.uk

Status Draft

Revised 2015-07-08

The goal of this exercise is to familiarise you with how GUI widgets are created and configuring in Qt.

You may find it useful to have the [Qt Widgets documentation](#) open in a separate browser tab before you begin. An alternative to viewing the class reference material in a web browser is to run the *Qt Assistant* application from a terminal window, like so:

```
assistant &
```

2.1 Simple Buttons

1. Open a terminal window. Use the `mkdir` and `cd` commands to create a directory for this exercise and then move into it. Use these commands a second time to create and move into a subdirectory called `button`.
2. In this subdirectory, create a file called `button.cpp`. Base this file on `hello.cpp` from the [previous exercise](#) but modify it so that it creates a button containing the text `Click Here` instead of a label. (You can simply replace all occurrences of `QLabel` with `QPushButton` and change the text string.)
3. Set up your Qt project:

```
qmake -project
```

Edit the resulting file `button.pro` and add the `QT += widgets` line, as you did in the previous exercise. Then create a makefile and build the program using the following pair of commands:

```
qmake button.pro  
make
```

Lastly, run the program with `./button`. A very small window containing the button should appear. Note that clicking on the button will do nothing (for now). Dismiss the window in the usual way.

4. Try adding an icon to the button.

Start by downloading the file `icon.png` to the same directory as `button.cpp`. Then add a line like the following to the main program, after the line creating the `QPushButton` object and before the line calling its `show` method:

```
widget->setIcon(QIcon("icon.png"));
```

Recompile using the `make` command, then rerun using `./button`.

5. Try changing the font used for button text by adding a line like the one below to the main program.

```
widget->setFont(QFont("Times", 18, QFont::Bold));
```

Recompile and rerun as before.

6. Add a line that calls `setWindowTitle` on `widget`, setting the title to “QPushButton”. Then add another line that calls `setMinimumSize`, specifying 240 and 60 as the width and height arguments. Recompile and rerun the program, and you should see something like this:



Fig. 2.1: QPushButton labelled with text and an icon

7. Try disabling the button. You will need to call the `setEnabled` method on the widget, with an argument of `false`.
8. Remove or comment out the code that disables the button, then add the following to the program, just before the call to the button’s `show` method:

```
QObject::connect(widget, SIGNAL(clicked()), &app, SLOT(quit()));
```

Recompile and rerun. Click the button and the program should quit.

Note: This is an example of the ‘signals and slots’ mechanism that Qt uses to handle user interaction, explored in more detail in a [later exercise](#).

2.2 Spin Boxes

1. Create a subdirectory called `spin`, alongside (not inside) the `button` directory created for the previous task. In this directory, create a file called `spin.cpp`. In this file, write a program that creates a `QDoubleSpinBox` widget. As in the previous task, this widget should act as the application’s window.

Follow the same steps as before to create the project file and makefile. Run `make` to build the application. Fix any compilation errors before continuing.

2. Configure your `QDoubleSpinBox` widget in the following ways:
 - Use `setRange` to set lower and upper limits of 0.0 and 1.0, respectively.
 - Use `setSingleStep` to specify a step size of 0.05.
 - Use `setFont` to specify 16-point DejaVu Sans as the font.
 - Use `setMinimumSize` to specify a minimum width and height of 250 pixels and 40 pixels, respectively.
 - Use `setWindowTitle` to specify the window title as “QDoubleSpinBox”.

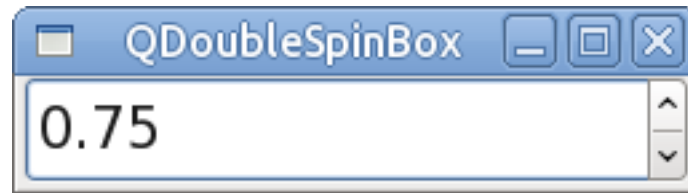


Fig. 2.2: Simple example of a `QDoubleSpinBox` widget

Your application now should look something like this when run:

Try entering values into the spin box. Try altering the value using the small buttons at the right of the spin box.

2.3 Single-Line Text Input

This part of the exercise uses PyQt rather than using the Qt libraries directly from C++.

1. In your terminal window, use `cd ..` to move up one level, out of the `button` subdirectory. Then create a new file called `text.py`. Base this file on `hello.py` from the *previous exercise* but modify it so that it creates a `QLineEdit` object instead of a `QLabel` object.

Run the program and you should see a small window containing an input field suitable for entering a single line of text.

2. Try changing text alignment. You will need to add an import statement like this:

```
from PyQt5.QtCore import Qt
```

Then you will need to add a call to the `setAlignment` method at the appropriate point in the code:

```
widget.setAlignment(Qt.AlignCenter)
```

(This assumes that `widget` is the name of your `QLineEdit` variable.)

3. Try disabling text input by calling `setEnabled` on the object, with an argument of `False`. Rerun the program and try entering some text.
4. Remove the call to `setEnabled`. Try restricting input to three-digit integers, using an **input mask**.

You can do this by calling `setInputMask` on the object and using `"999"` as the mask. The [QLineEdit documentation](#) has details of what this means and how masks are specified. (Click on `inputMask` in the list of properties to see this information.)

Run the program again after adding this code and try entering different kinds of input.

5. Remove or comment out the call to `setInputMask`. Set a **validator** that limits input to an integer in the range 1-20, like so:

```
widget.setValidator(QIntValidator(1, 20, None))
```

(You will need to modify the `import` statements at the top of the program so that `QIntValidator` is imported from `PyQt5.QtGui`.)

2.4 Multi-Line Text Input

1. Study the [QTextEdit documentation](#), then write a small program in either C++ or Python that creates and displays a `QTextEdit` widget, in a window by itself. Use the widget's `setWindowTitle` method to give the window

a title of “QTextEdit Example”. The final result should look something like this:

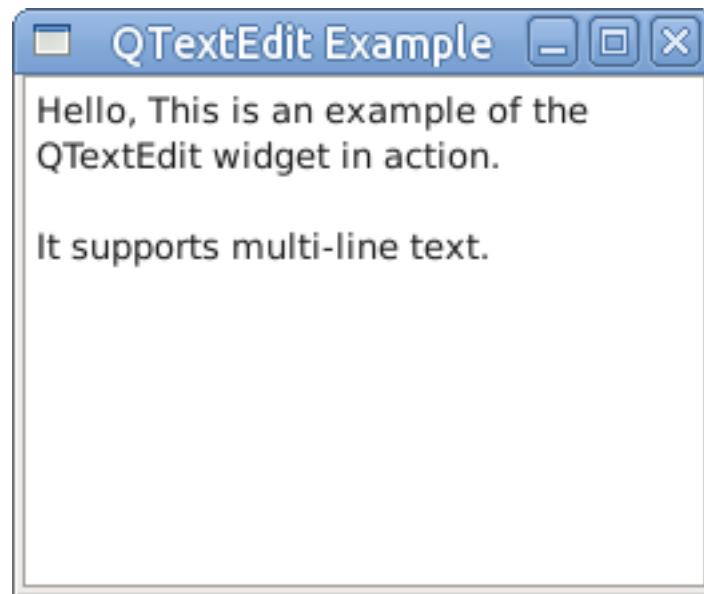


Fig. 2.3: Initial program with a QTextEdit widget

2. Try resizing the window, both horizontally and vertically. Notice how the text rewraps in response to changing dimensions, and how scroll bars appear when necessary.
3. Call methods on your widget to change the font to something different (e.g., 14-point DejaVu Sans) and change the text colour to a shade of red. Add another method call to disable word wrapping. You should end up with something like this:

Note: You will need to study the documentation for [QColor](#), [QFont](#) and [QTextOption](#) to determine the method calls that are required. If you are using PyQt, note that these classes must be imported from `PyQt5.QtGui`, not from `PyQt5.QtWidgets`.

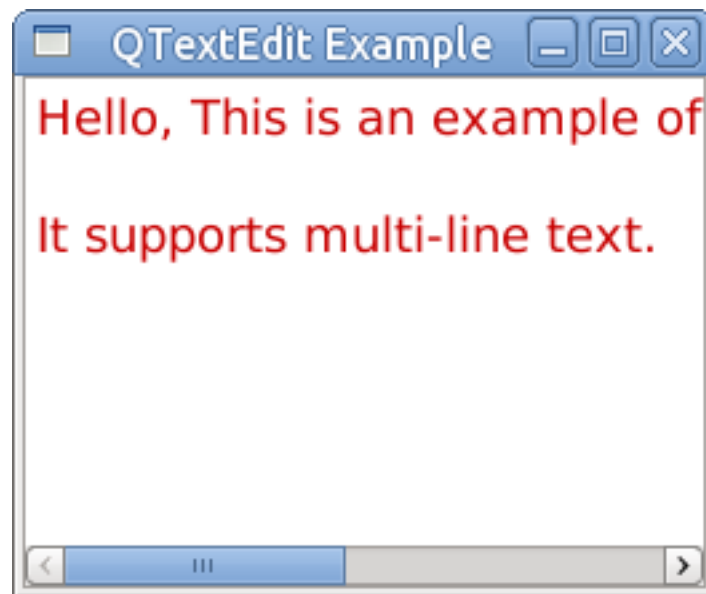


Fig. 2.4: Final version of program, with a customised QTextEdit widget

PROGRAMMING WIDGET LAYOUT

Author Nick Efford

Contact N.D.Efford@leeds.ac.uk

Status Draft

Revised 2015-07-10

As in the previous exercise, some of this work uses C++ and some of it uses Python.

You may find it useful to have the [Qt Widgets documentation](#) open in a separate browser tab before you begin. An alternative to viewing the class reference material in a web browser is to run the *Qt Assistant* application from a terminal window, like so:

```
assistant &
```

3.1 Experimenting with QHBoxLayout

1. Create a directory for this exercise, then create a subdirectory within it called `hbox`. Inside that directory, create a file called `hbox.cpp` containing the following:

```
#include <QtWidgets>

int main(int argc, char* argv[])
{
    QApplication app(argc, argv);

    QWidget* window = new QWidget();
    window->setWindowTitle("HBoxLayout Test");

    window->show();

    return app.exec();
}
```

Generate the project file using `qmake -project`, add the line `QT += widgets` to `hbox.pro` and then generate a makefile using `qmake`.

Note: Notice the use of `QWidget` to represent the application's window. Qt does have a 'proper' window class called `QMainWindow` that supports the use of menu bars, etc, but we aren't using any of those features here, so `QWidget` will suffice.

Run `make` to compile the program, then run the program with `./hbox`. An empty window should be displayed.

2. Add code to `hbox.cpp`, to create three `QPushButton` objects. Remember to define these using pointers. Use `button1`, `button2`, `button3` as the names of these pointer variables.
3. Now create a `QHBoxLayout` object and add the three buttons to it. The following code, put *before* the call to the `show` method, should do the trick:

```
QHBoxLayout* layout = new QHBoxLayout();
layout->addWidget(button1);
layout->addWidget(button2);
layout->addWidget(button3);
```

4. Add code to make this the layout used by the `QWidget` object representing the application window:

```
window->setLayout(layout);
```

Compile and run the program again. Try resizing the window both horizontally and vertically in various ways and observe how it behaves.

5. Change `hbox.cpp` so that it creates a label, a text field and a button, instead of three buttons. Use `QLabel` for the label and `QLineEdit` for the text field, and don't forget to add the `#include` directives for these components. The final result should look like this:

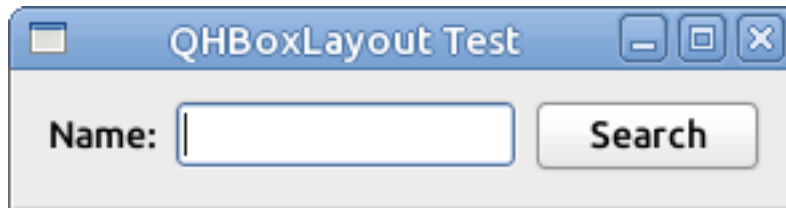


Fig. 3.1: A `QHBoxLayout` example.

Recompile and rerun your program. Try resizing the window horizontally and vertically, as you did before. How has the resizing behaviour changed?

6. Add the following between the `addWidget` method calls for the text field and button:

```
layout->addStretch();
```

Recompile and rerun your program. How has its resizing behaviour changed?

7. Move the call to `addStretch` to that it is before the line that adds the `QLabel` widget to the layout. Add a second call to `addStretch`, after the line that adds the button to the layout. Recompile and rerun your program. How has its resizing behaviour changed?

3.2 Developing a Nested Layout

1. In your terminal window, move up one level to the directory that you created for this exercise. Download the file `nested.py` into that directory. Run it with `python nested.py` and you should see a blank window appear.
2. Edit the file and, underneath the `# Create components` comment, add code to create a label and a text field. Don't forget to add the relevant `import` statement for `QLabel` and `QLineEdit` to the top of the file.

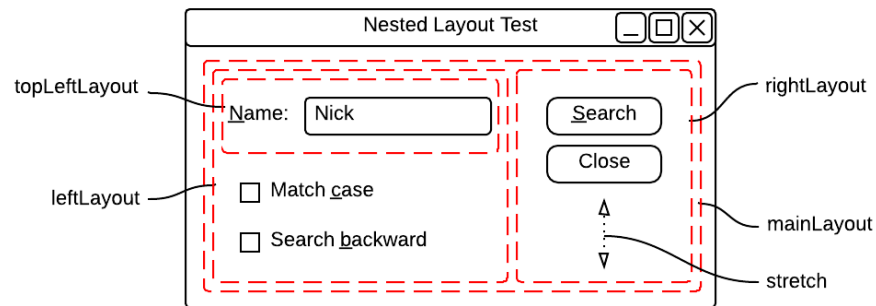
```
label = QLabel("&Name:")
nameField = QLineEdit()
label.setBuddy(nameField)
```


Notice the `&N` in the label text and how the label's `setBuddy` is called to associate the label with the text field. These two steps will allow users to shift focus to the text field using a keyboard shortcut (`Alt+N` on Linux machines).

- Now add code to create two buttons, labelled `Search` & `Close`, and two checkboxes, labelled “`Match case`” & “`Search backward`” (once again, don't forget to add the necessary `import` statements):

```
searchButton = QPushButton("&Search")
closeButton = QPushButton("Close")
caseCheck = QCheckBox("Match &case")
backwardCheck = QCheckBox("Search &backward")
```

- The next task is to arrange these components in a set of nested layouts, using the `QHBoxLayout` and `QVBoxLayout` objects shown below:



Start by adding the layout code for the left of the window, below the `# Arrange components` comment:

```
topLeftLayout = QHBoxLayout()
topLeftLayout.addWidget(label)
topLeftLayout.addWidget(nameField)

leftLayout = QVBoxLayout()
leftLayout.addLayout(topLeftLayout)
leftLayout.addWidget(caseCheck)
leftLayout.addWidget(backwardCheck)
```

Remember that you'll need `import` statements for `QHBoxLayout` and `QVBoxLayout`, too.

Note: When you are placing a layout inside another layout, you must use the `addLayout` method rather than the `addWidget` method.

- Next, add the layout code for the right-hand side of the window. See if you can figure this out for yourself! Use `rightLayout` as the name for the required layout object. Note that you can call `addStretch` on this object to add a stretch element to the layout.
- Finally, add the code that places `leftLayout` and `rightLayout` into `mainLayout` and sets this as the layout of the window:

```
mainLayout = QHBoxLayout()
mainLayout.addLayout(leftLayout)
mainLayout.addLayout(rightLayout)
window.setLayout(mainLayout)
```

Run the program with `python nested.py`. Try resizing the window.

- To stop vertical resizing from spoiling the layout, add this line before the line that shows `window`:

```
window.setFixedHeight(window.sizeHint().height())
```

Here, the chained method call `window.sizeHint().height()` returns the optimum height for the window's layout.

3.3 Using QFormLayout

You've seen some examples of putting labels alongside corresponding input fields using `QHBoxLayout`. This is such a common requirement that Qt provides a specific layout class called `QFormLayout` to support it.

`QFormLayout` can create the labels and buddy them with the associated input fields for you:

```
form = QFormLayout()
form.addRow("&Name", nameField)
```

`QFormLayout` will also align labels and input fields if there are multiple rows in the form, creating a neat two-column layout. Rows of the layout can be wrapped so that the labels appear above the corresponding input fields when horizontal space is limited.

1. Download `form.py` and study it, then run the program. You should see something like this:

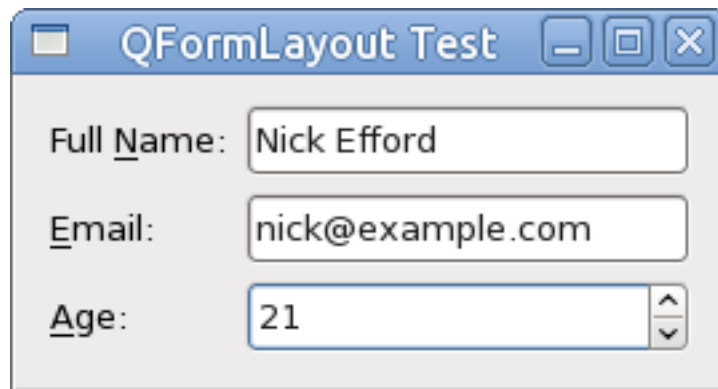


Fig. 3.2: Example of using `QFormLayout`

Try using the `Tab` key to move between input fields. Check that the keyboard shortcuts work (`Alt+N`, `Alt+E`, `Alt+A` on Linux machines).

2. Try changing the label alignment by adding the following line to the program:

```
form.setLabelAlignment(Qt.AlignRight)
```

(If labels were already right-aligned on the platform on which you are running the program, use `Qt.AlignLeft` instead.)

3. Remove the line that modified label alignment, then try changing the wrapping behaviour by adding the following line:

```
form.setRowWrapPolicy(QFormLayout.WrapAllRows)
```

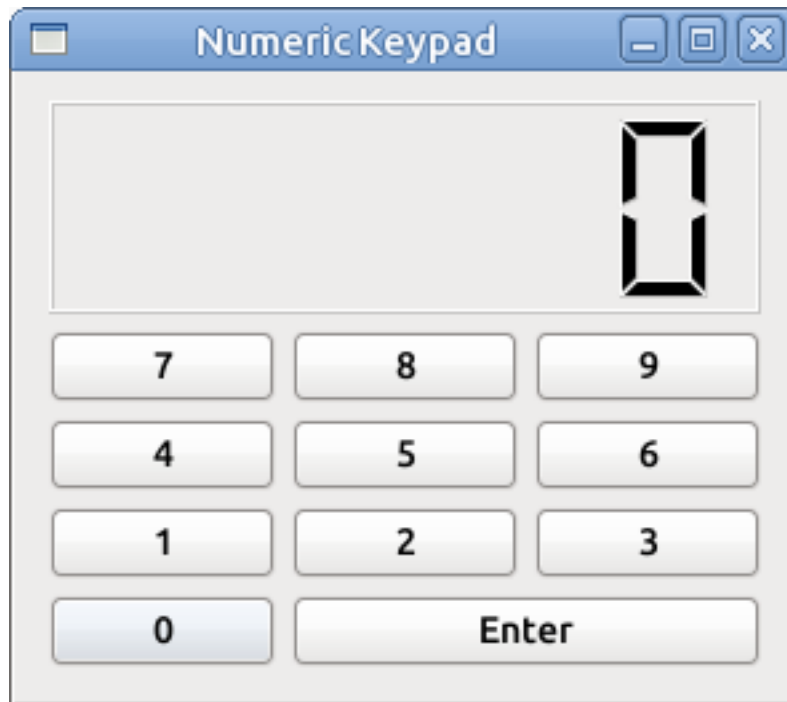
When you rerun the program, you should see that the labels are now above their corresponding input fields.

3.4 Using QGridLayout

Note: You're on your own for this one!

Give it a try, following the advice below. Use the documentation for `QGridLayout`, [on the web](#) or in *Qt Assistant*. If you get stuck, consult the sample solutions for C++ or Python.

In either C++ or Python ¹, use `QGridLayout` to implement a numeric keypad resembling the one shown below:



Your implementation should

- Use `QLCDNumber` to represent the numeric display (assume 6 digits)
- Call `setMinimumHeight` on the `QLCDNumber` component to impose a minimum height of 80 pixels
- Use `QPushButton` to represent the keys
- Add the numeric keys by calling `addWidget` on the `QGridLayout` object, passing in a `QPushButton` object, row and column
- Add the display and the Enter key to the grid by calling `addWidget` on the `QGridLayout` object, passing in the component, the row, the column, the row span and the column span

Try resizing the window after you run the application.

¹ A Python implementation will be easier!

APPLICATIONS AS CLASSES

Author Nick Efford

Contact N.D.Efford@leeds.ac.uk

Status Draft

Revised 2015-07-09

This exercise introduces a cleaner and more scalable way of structuring your Qt applications, as *classes* rather than procedural programs.

We focus here on creating subclasses of `QWidget`, which is a suitable choice for simple UIs that don't need menus, toolbars, etc. A better approach for more complex UIs is to subclass `QMainWindow` - which is covered in a [later exercise](#). We consider how subclassing is done in both Python and C++.

4.1 Subclassing QWidget in Python

4.1.1 Suggested Template

Try structuring your UI like this:

```
1 class MyWindow(QWidget):
2
3     def __init__(self):
4         super().__init__()
5         self.createWidgets()
6         self.arrangeWidgets()
7         self.setWindowTitle("My Window")
8         self.setMinimumSize(320, 200)
9
10    def createWidgets(self):
11        # Create widgets as instance variables here
12
13    def arrangeWidgets(self):
14        # Create layout for the contained widgets here
15        # Remember to call self.setLayout!
```

Several things happen in the constructor. First, on line 4, the constructor of the superclass, `QWidget`, is called. **Don't forget to include this in your own classes!** It ensures that the inherited parts of your class are initialised properly. The constructor is also a logical place to specify the text that appears in the window's title bar (line 7), or constrain the window's dimensions (line 8).

Creation and arrangement of the UI's widgets could have also been done in the constructor but instead has been delegated to two separate methods, `createWidgets` and `arrangeWidgets`, which are called from the constructor.

This isn't a requirement; it is done purely to make the constructor smaller and improve the structure and readability of the code.

Note that `createWidgets` should create the widgets of the UI as instance variables, so that other methods of the class can access them. For example, if the UI needs a text field for entry of someone's name, it should be created with a line like this:

```
self.nameField = QLineEdit()
```

The `self.` prefix here is crucial!

4.1.2 Putting It Into Practice

1. Use the template above to create the UI for a simple currency converter application. Call your class `CurrencyConverter`. Use `QDoubleSpinBox` widgets for the amounts and `QComboBox` widgets for the corresponding currencies. Use a `QHBoxLayout` to pair up each spin box with the corresponding combo box. Use a `QVBoxLayout` for the overall layout of the window.
2. Underneath the class definition, write a main program that creates an instance of your UI in the usual way. Code like this will be suitable:

```
if __name__ == "__main__":  
    import sys  
    from PyQt5.QtWidgets import QApplication  
    app = QApplication(sys.argv)  
    ui = CurrencyConverter()  
    ui.show()  
    sys.exit(app.exec_())
```

You should see something like this when you run the program:

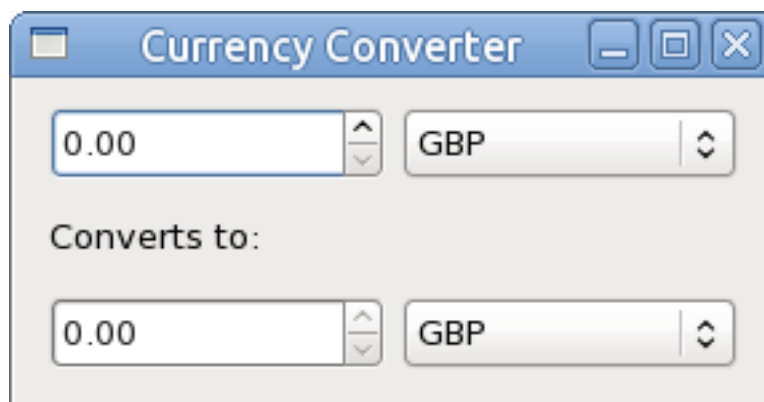


Fig. 4.1: UI of currency converter application

3. If you want more practice, try using the template to create the UI for one of the layout examples in the [previous exercise](#).

4.2 Subclassing QWidget in C++

4.2.1 Suggested Template

Try structuring your UI using three files: `window.hpp`, which defines the class; `window.cpp`, which implements the methods of the class; and `main.cpp`, which provides the main program.

`window.hpp` can have this format:

```

1  #pragma once
2
3  #include <QWidget>
4
5  // Put forward references to widget classes here
6
7  class MyWindow: public QWidget
8  {
9      public:
10         MyWindow();
11
12         private:
13         void createWidgets();
14         void arrangeWidgets();
15
16         // Specify widgets here, using pointers
17 };

```

If your UI needs a text field for entry of someone's name, you would specify a forward reference for the `QLineEdit` class like this:

```
class QLineEdit;
```

You would then specify the widget instance variable in the `private` section of the class definition like so:

```
QLineEdit* nameField;
```

Given this class definition, `window.cpp` will look something like this:

```

1  #include <QtWidgets>
2  #include "window.hpp"
3
4  MyWindow::MyWindow()
5  {
6      createWidgets();
7      arrangeWidgets();
8      setWindowTitle("My Window");
9      setMinimumSize(320, 200);
10 }
11
12 MyWindow::createWidgets()
13 {
14     // Create widgets using new here
15 }
16
17 MyWindow::arrangeWidgets()
18 {
19     // Create layout for widgets here
20     // Remember to call setLayout!
21 }

```

Finally, `main.cpp` can take this form:

```
1  #include <QApplication>
2  #include "window.hpp"
3
4  int main(int argc, char* argv[])
5  {
6      QApplication app(argc, argv);
7
8      MyWindow window;
9      window.show();
10
11     return app.exec();
12 }
```

In this case, the top-level UI object is created and manipulated on the stack (lines 8 & 9), but you could create it on the heap instead if you wanted to:

```
QWidget* window = new MyWindow();
window->show();
```

4.2.2 Putting It Into Practice

1. Create a subdirectory called `currency`. Inside this subdirectory, use the template described above to create a C++ version of the currency converter application, in three files called `window.hpp`, `window.cpp` and `main.cpp`.
2. Use `qmake -project` to generate project file `currency.pro`, then edit this project file and add the `QT += widgets` line to it. Run `qmake` to create a makefile, then run `make` to build the application.
Run the compiled application with `./currency`. The UI should look identical to the Python version that you created earlier.
3. If you want more practice, try using the template to create the UI for one of the layout examples in the [previous exercise](#).

HANDLING INTERACTION WITH SIGNALS & SLOTS

Author Nick Efford

Contact N.D.Efford@leeds.ac.uk

Status Draft

Revised 2015-07-10

You may find it useful to have The Qt Project documentation on [Signals & Slots](#) open in your browser while doing this exercise.

5.1 Using Built-in Signals & Slots in C++

1. Create a directory for this exercise and download the archive `volume.zip` into it. Then unpack this archive. This should create a subdirectory called `volume`. In a terminal window, `cd` into this subdirectory and spend a couple of minutes examining the files therein.
2. Generate a Qt project file by entering `qmake -project`. Edit the resulting file and add the `QT += widgets` line to it. Then generate the makefile with `qmake`. Finally, enter `make` to build the program and `./volume` to run it. You should see something like this:

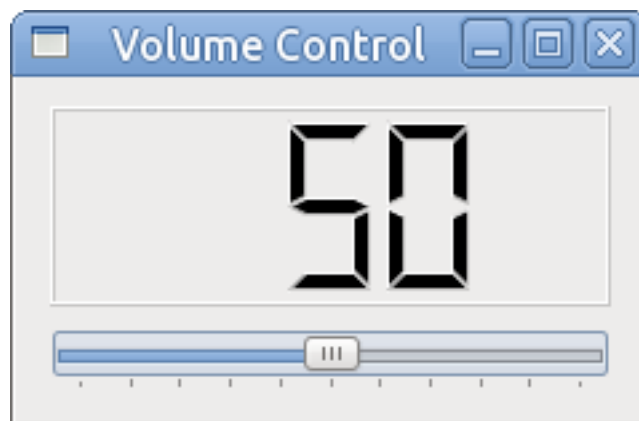


Fig. 5.1: GUI for Volume Control application

Moving the slider will currently have no effect.

3. Edit `window.hpp` and add the following prototype to the `private` section of the class definition:

```
void makeConnections();
```

This method is where the code to connect signals to slots will go.

4. Edit `window.cpp` and add a line to the constructor that calls the `makeConnections` method. Then define the `makeConnections` method at the bottom of the file, like so:

```
void VolumeControl::makeConnections()
{
    connect(slider, SIGNAL(valueChanged(int)), number, SLOT(display(int)));
}
```

The call made here to the inherited `connect` method will connect the `valueChanged(int)` signal generated by the `QSlider` widget to the `display(int)` slot provided by the `QLCDNumber` widget.

5. Enter `make` to recompile the program and try running it again. If you have connected signal and slot correctly, you should find that moving slider updates the number.

5.2 Using Built-in Signals & Slots in Python

1. Download the file `volume.py` into the directory that you created for this exercise. Edit the file and examine it. This is the PyQt equivalent of the C++ code from the previous task. Run it and you should see the same GUI as before. Once again, moving the slider will have no effect.
2. Follow the same procedure as for the C++ version of the program. Define a `makeConnections` method in the class, containing code to connect the `valueChanged(int)` signal of the slider to the `display(int)` slot of the number. In this case, the syntax required is rather different, and somewhat simpler:

```
self.slider.valueChanged.connect(self.number.display)
```

Add a line to the constructor that calls your `makeConnections` method. Run the program again and check that moving the slider now updates the number.

5.3 Custom Slots in C++

1. Download the file `dice.zip` to the directory that you created for this exercise. In a terminal window, `cd` to that directory and unpack the archive with the command `unzip dice.zip`. This should create a subdirectory called `dice`, containing several files.

Move into the `dice` subdirectory and generate the Qt project file by entering `qmake -project`. Edit the project file and add the `QT += widgets` line to it. Then generate the makefile by entering `qmake`.

Spend a few minutes examining the files `window.hpp`, `window.cpp` and `main.cpp`. Then build the application by entering `make` and run it with `./dice`. You should see the GUI shown below.

Clicking on the *Roll Dice* button will currently have no effect.

2. Edit `window.hpp`. Add the `Q_OBJECT` macro as the first item inside the `DiceRoller` class definition. Then add a prototype for a `void` method called `makeConnections` to the private section. Finally, add a new 'slots' section to the class definition, just after the private section:

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

3. Edit `window.cpp`. Add the following definition of the `rollDice` method to the bottom of the file:

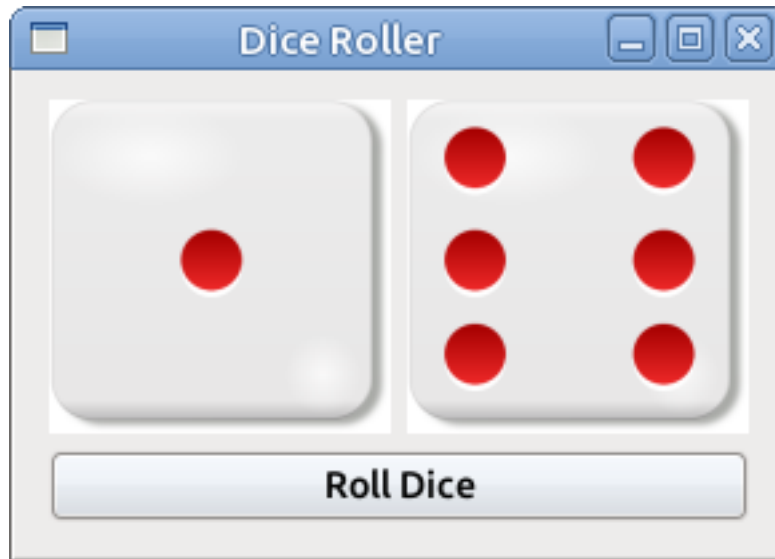


Fig. 5.2: GUI for Dice Roller application

```
void DiceRoller::rollDice()
{
    int n = qrand() % 6;
    die1->setPixmap(dieFaces[n]);
    n = qrand() % 6;
    die2->setPixmap(dieFaces[n]);
}
```

Then find the two lines near the start of the `DiceRoller` constructor that call `setPixmap` on labels `die1` and `die2`. Replace these lines with a call to the `rollDice` method.

- Now define the `makeConnections` method to look like this:

```
void DiceRoller::makeConnections()
{
    connect(rollButton, SIGNAL(clicked()), this, SLOT(rollDice()));
}
```

The call to the `connect` method looks a little different the earlier example. Here, the recipient is specified using the `this` pointer, meaning ‘this `DiceRoller` object’. The slot is the custom `rollDice` method defined in the previous step.

Don’t forget that you also need to add a line to the constructor that calls `makeConnections`!

- Use `qmake` to rebuild the makefile. This is necessary because we have added a dynamic property (a custom slot) and must therefore run Qt’s **Meta-Object Compiler** (`moc`) as part of the build process. Running `qmake` will add commands that invoke `moc` to the makefile.

Use `make` to recompile the application, then run it with `./dice`. Clicking the *Roll Dice* button should now change the die face images in a random fashion.

5.4 Custom Slots in Python

1. Edit your `volume.py` program from the *Using Built-in Signals & Slots in Python* task. Add the following to the import statements at the top of the file:

```
from PyQt5.QtCore import pyqtSlot as Slot
```

`Slot` (created as a convenient alias for `pyqtSlot`¹) is a **decorator** that can be attached to method (or function) definitions to identify them as slots.

2. Now let's define the slot. Add the following method definition to the `VolumeControl` class:

```
@Slot(int)
def numberColour(self, value):
    if value > 75:
        self.number.setStyleSheet("color:red")
    else:
        self.number.setStyleSheet("color:black")
```

`@Slot(int)` decorates `numberColour`, marking it as a slot that can receive a single integer - represented inside the method by the `value` parameter.

This slot will change the colour of the digits displayed by `number` on the basis of that received value, making the digits red if the value is above 75, black otherwise. It does this by changing the **style sheet** associated with the `QLCDNumber` widget.

3. The slider's `valueChanged(int)` signal still needs to be connected to this custom slot. Find the `makeConnections` method and add the following to it:

```
self.slider.valueChanged.connect(self.numberColour)
```

4. Run the program. Now, as you adjust the slider above the threshold value of 75, the number should change colour from black to red.

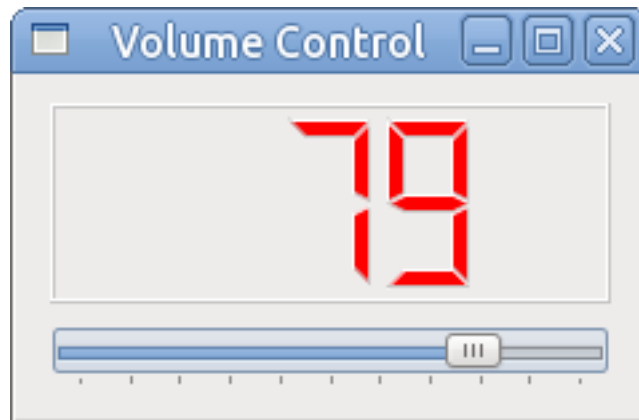


Fig. 5.3: GUI for final version of Volume Control application

¹ You could simply import `pyqtSlot` and use it as the decorator. This example renames it to `Slot` because that is the name given to it by an alternative Python binding called `PySide`. Using the same name for the decorator means that the example could be ported to `PySide` simply by changing the import statements at the top of the file.

This isn't a big deal, so you should feel free to use `pyqtSlot` if you prefer!

5.5 Other Things to Try

Here are some additional tasks if you want more practice at connecting signals to slots:

1. Modify your C++ implementation of the Volume Control application so that it has a custom slot equivalent to the Python version.
 2. Write a Python version of the Dice Roller application.
 3. Modify the currency converter application from the [previous exercise](#) so that the conversion is done whenever a new amount is entered or either currency is changed.
-

CREATING CUSTOM WIDGETS IN QT

Author Nick Efford

Contact N.D.Efford@leeds.ac.uk

Status Draft

Revised 2015-07-10

This exercise shows how subclassing can be used to create new widgets that specialise, extend or combine the capabilities of those already provided with Qt.

6.1 A Custom Widget For Text Input

Subclassing of existing Qt widgets allows us to create variants with additional or different behaviour.

1. If you've not already done so, download the [example code from Lecture 3](#) then examine the file `postcode.py`. This is a PyQt implementation of a custom version of `QLineEdit`, specialised to allow the input of UK postcodes.

Using `postcode.py` as a guide, create a new file called `examgrade.py` and in it write your own custom widget based on `QLineEdit`. Your widget class should be called `ExamGradeInput` and it should constrain input to be an integer in the range 0 to 100. In addition to a constructor, give your class a method `getGrade` that returns the exam grade as an integer. Returning `int(text())` should suffice here.

Also in `examgrade.py`, write a small program that tests your custom widget, following the approach shown in `postcode.py`. Then run the program to check that the widget works correctly.

2. Now examine the files `postcode.hpp`, `postcode.cpp` and `main.cpp` in the *postcode* subdirectory of the [example code from Lecture 3](#). These files contain a C++ implementation of the postcode entry widget and a small program to test the widget.

Using these files as a guide, create a subdirectory `examgrade` and in it create a C++ implementation of the `ExamGradeInput` widget, along with a suitable test program. Use `qmake` to create a project file and makefile, then use `make` to build an executable. Run this executable to test your widget.

6.2 Composite Widgets

We can also create **composites**, in which a set of existing related widgets are bundled together so that they can be manipulated as a single widget.

1. Download `controller.py`. This file contains an incomplete implementation of a widget called `Controller`, plus an associated test program (the code underneath the `if __name__ == "__main__":`). The `Controller` widget will use the 'volume controller' code from the [Handling Interaction With Signals &](#)

Slots exercise, but this code hasn't been added yet. `Controller` is based on `QGroupBox`, which provides a title and visual grouping for the contents of the widget. (We could have used `QWidget` as the superclass if we didn't need this visual grouping.)

Study the code. Note, in particular, how the definition of the constructor begins:

```
class Controller(QGroupBox):
    ...
    def __init__(self, title="", parent=None):
        super().__init__(title, parent)
    ...
```

The constructor takes a `parent` parameter representing the parent widget of `Controller` - i.e., the widget that contains it. The default of `None` means that there is no parent - i.e., that the `Controller` is the top-level widget of the application. The value of `parent` is passed on to the superclass constructor so that it can be used to establish the containment hierarchy of the UI.

Warning: If you want compose a UI from custom widgets, you *must* ensure that those widgets can specify their parent in this manner.

Run this program now. A window should appear, containing only the titles 'Left Channel' and 'Right Channel'.

2. In the `createWidgets` method, replace the `pass` statement with code to create and configure the `QLCDNumber` and `QSlider` widgets:

```
self.number = QLCDNumber(3)
self.number.display(50)

self.slider = QSlider(Qt.Horizontal)
self.slider.setRange(0, 100)
self.slider.setTickPosition(QSlider.TicksBelow)
self.slider.setTickInterval(10)
self.slider.setValue(50)
```

3. In the `arrangeWidgets` method, replace the `pass` statement with layout code for the two child widgets:

```
layout = QGridLayout()
layout.addWidget(self.number, 0, 0)
layout.addWidget(self.slider, 1, 0)
layout.setRowMinimumHeight(0, 75)
self.setLayout(layout)
```

4. In the `makeConnections` method, replace the `pass` statement with code that connects the slider's `valueChanged` signal to the number's `display` slot. Then run the program. You should see something like this:

Moving the slider of each `Controller` widget should update its number, independently of the other.

6.3 Adding Custom Signals & Slots

How might we link the two `Controller` widgets so that adjusting the slider of either one of them makes an identical adjustment to the other? ¹

1. One approach is to access the internal widgets of each `Controller` widget directly, connecting the `valueChanged` signal of the slider on the left to the `setValue` slot of the slider on the right, and vice versa. Implement this now, by adding the following code to the test program:

¹ This is known as 'ganging'.

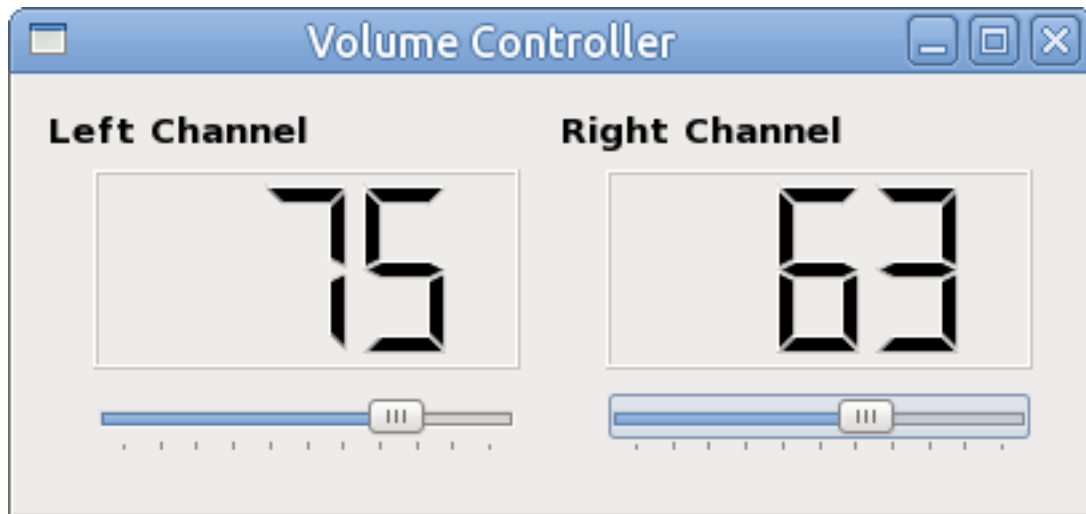


Fig. 6.1: GUI of Controller test program

```
left.slider.valueChanged.connect(right.slider.setValue)
right.slider.valueChanged.connect(left.slider.setValue)
```

Run the program again to verify that the two sliders are now ‘ganged’ together.

2. One problem with this solution is that it unnecessarily exposes an implementation detail of the `Controller` widget: namely, that it has an internal `slider` object. A cleaner solution would be to give the `Controller` widget its own custom signal and slot to handle changes in value.

Note: The key point to note here is that custom signals and slots like this provide the means to ‘wire up’ `Controller` widgets to each other and to other Qt widgets. Programs can use these custom signals and slots and do not need to know anything about what is happening inside a `Controller`. This makes it easier to change the implementation of `Controller` at a later date without breaking any code that uses the widget.

Create a suitable signal now by adding the following to the definition of `Controller`, after the docstring but before the definition of `__init__`:

```
valueChanged = Signal(int)
```

(Signal here is an alias for `pyqtSignal`, imported from `PyQt5.QtCore`.)

Next, in the `makeConnections` method, connect the `valueChanged` signal of the slider to your newly defined custom signal:

```
self.slider.valueChanged.connect(self.valueChanged)
```

Qt allows signals to be connected to other signals. In this case, it means that, for every `valueChanged` signal emitted by the slider, the `Controller` widget itself will emit an equivalent signal.

3. Now create the custom slot. Add the following method, complete with `Slot` decorator, to the `Controller` class:

```
@Slot(int)
def setValue(self, value):
    self.slider.setValue(value)
```

This provides a ‘public’ slot for `Controller`, which simply delegates to the `setValue` slot of the slider.

4. Finally, go to the main program and remove the two lines that connect the sliders of the two `Controller` widgets directly. Replace those lines with this code:

```
left.valueChanged.connect(right.setValue)
right.valueChanged.connect(left.setValue)
```

Run the program to check that it still has the same ‘ganged’ behaviour.

CREATING DIALOGS & APPLICATION WINDOWS

Author Nick Efford

Contact N.D.Efford@leeds.ac.uk

Status Draft

Revised 2015-07-14

7.1 Creating Dialogs by Subclassing QDialog

(Note: this first exercise is based on a code example in Blanchette & Summerfield's *C++ GUI Programming With Qt 4*.)

Suppose you need to create a Find dialog for a text editing application. This can be implemented as a subclass of `QDialog` with its own signals and slots. These signals and slots can be used for communication with other components of the application's user interface. Hence, the Find dialog can be self-contained and will not need to share state with other parts of the application - making the application as a whole easier to understand and maintain.

1. Create a directory for this exercise and download `dialog.zip` into it. Unpack the archive and you should have a new subdirectory called `dialog`, containing three files: `finddialog.hpp`, `finddialog.cpp` and `main.cpp`.

Run `qmake -project` to create the project file. Edit this file and add the `QT += widgets` line. Then run `qmake` to create the makefile.

2. Open `finddialog.hpp` in a text editor and study the code.

```
1 // Header file for Find dialog (implemented in finddialog.cpp)
2 // (NDE, 2014-10-27)
3
4 #pragma once
5
6 #include <QDialog>
7
8 class QCheckBox; // forward reference
9 class QLabel;
10 class QLineEdit;
11 class QPushButton;
12
13 class FindDialog: public QDialog
14 {
15     Q_OBJECT
16
17     public:
```

```

18     FindDialog(QWidget* = 0);
19
20     signals:
21         void findNext(const QString&, Qt::CaseSensitivity);
22         void findPrevious(const QString&, Qt::CaseSensitivity);
23
24     private:
25         void createWidgets();
26         void arrangeWidgets();
27         void makeConnections();
28
29         QLabel* label;
30         QLineEdit* lineEdit;
31         QCheckBox* caseBox;
32         QCheckBox* backwardBox;
33         QPushButton* findButton;
34         QPushButton* closeButton;
35
36     private slots:
37         void findClicked();
38         void enableFindButton(const QString&);
39 };

```

Line 4 is a preprocessor directive indicating to the compiler that the contents of this file should be included once only, and that subsequent includes should be ignored. (An alternative technique is to use an ‘include guard’.)

Lines 8-11 are **forward references** to the widget classes used by the dialog. This is done instead of including the header files for those classes, as a small optimisation. (The implementation file for this class will need to include them, but users of the FindDialog class won’t necessarily need to use these widgets directly, so leaving them out of finddialog.hpp makes sense.)

Line 13 indicates that the FindDialog class inherits from QDialog. **Line 18** specifies a constructor that can take a QWidget as a parameter, representing the parent of the dialog. **Lines 25-27** specify three private methods used when creating the dialog. **Lines 29-34** specify the internal widgets of the dialog as private instance variables.

Line 15 flags the fact that this class defines its own custom signals and slots. The qmake tool will recognise the presence of Q_OBJECT and add commands to the makefile that will invoke moc, the meta-object compiler. This tool adds the low-level C++ ‘plumbing’ needed for signals and slots to work.

Lines 20-22 and 36-38 specify the custom signals and slots that FindDialog needs. It will emit findNext and findPrevious signals, each including a string (the text being sought) and a flag to indicate whether the search should ignore case or not. Note that the signals are implicitly public, so that other software components can use them. The slots are used internally by the dialog and are therefore declared as private.

- Now open finddialog.cpp in your text editor. The constructor has been implemented for you. It calls methods to create the widgets, set up the widget layout and connect signals to slots, then sets the dialog window title and fixes the height of the dialog at its recommended height. You need to implement the other five methods in this file.

Start with createWidgets. Add code that

- Creates the lineEdit widget
- Creates the label widget with text of "Find &what?" and sets lineEdit as its buddy
- Creates caseBox and backwardBox, with text of "Match &case" and "Search &backward", respectively
- Creates findButton, with button text of "&Find"

- Calls the `setDefault` and `setEnabled` methods on `findButton`, passing in arguments of `true` and `false`, respectively
- Creates `closeButton`, with button text of "Close"

Check your coding by entering `make`. Fix any compiler errors before proceeding any further, but do not try running the program yet.

4. Now turn your attention to the `addWidget` method. Add code that will produce the widget layout shown below.

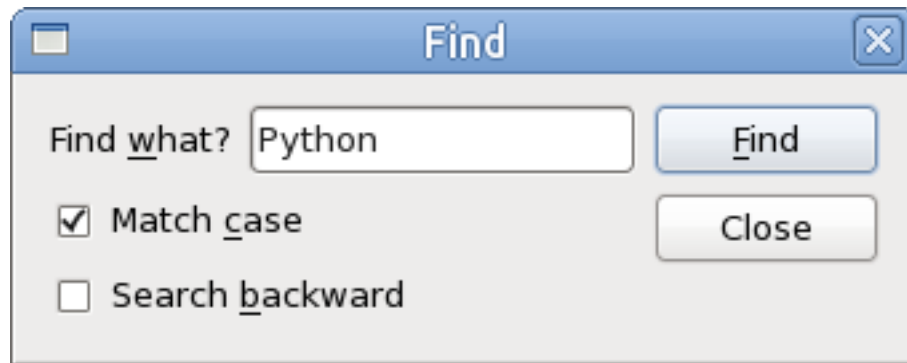


Fig. 7.1: Example of a Find dialog

You can achieve this by using a `QHBoxLayout` at the top-left, for the `label` and `lineEdit` widgets. This can be put into a `QVBoxLayout` along with the two checkboxes. The two buttons can likewise be put into a `QVBoxLayout` and these two vertical boxes can be put into another `QHBoxLayout`.

Don't forget to add some stretchable space below `closeButton`, and make sure that you end your implementation of `addWidget` with something equivalent to this:

```
setLayout(layout);
```

(This assumes that `layout` is the name you have given to the outermost `QHBoxLayout` object.)

Run `make` to recompile your code. Fix any compiler errors. When you have it compiling successfully, run the `./dialog` command to run the program and compare the resulting window with the screenshot above.

5. Wire up the dialog's widgets by adding the following code to the `makeConnections` method:

```
connect(lineEdit, SIGNAL(textChanged(const QString&)),
        this, SLOT(enableFindButton(const QString&)));

connect(findButton, SIGNAL(clicked()), this, SLOT(findClicked()));
connect(closeButton, SIGNAL(clicked()), this, SLOT(close()));
```

The first of these three connections ensures that any change to the text in the `lineEdit` widget results in the custom slot `enableFindButton` being called. (This slot has to accept a `QString` object containing the changed text, even though it won't actually use that text.)

The second call to `connect` ensures that clicking on `findButton` (once it has been enabled) results in a call to custom slot `findClicked`. The final call ensures that clicking on `closeButton` results in a call to the built-in `close` slot that `FindDialog` inherits from `Dialog`. Calling this slot will dismiss the dialog.

6. Put the following code in the body of `enableFindButton`:

```
findButton->setEnabled(not text.isEmpty());
```

This should enable the button if some text remains in the `lineEdit` widget and disable the button if the widget is empty.

Run `make` again and enter `./dialog`. Check that adding/removing text enables/disables `findButton` and that clicking `closeButton` closes the dialog.

7. Finally, add the following code to `findClicked`:

```
QString text = lineEdit->text();

Qt::CaseSensitivity cs;
if (caseBox->isChecked()) {
    cs = Qt::CaseSensitive;
}
else {
    cs = Qt::CaseInsensitive;
}

if (backwardBox->isChecked()) {
    emit findPrevious(text, cs);
}
else {
    emit findNext(text, cs);
}
```

Note, in particular, the highlighted lines. The `emit` macro is used to emit the custom signals `findPrevious` or `findNext`, depending on whether `backwardBox` has been ticked or not.

Run `make` again to check that everything still compiles. Note that you won't see any change in behaviour if you run the program again, as the emitted signals aren't connected to any slots.

7.2 Creating Application Windows by Subclassing QMainWindow

This exercise involves the creation of a small but useful image viewing application that demonstrates the most important features of `QMainWindow`. The application will be implemented using `PySide`.

1. Download `viewer.zip` into the directory you created for this set of exercises (i.e., not the `dialog` subdirectory, but its parent). This should give you a subdirectory `viewer`, containing a Python program called `viewer.py` and a few JPEG and PNG images ¹.

Open `viewer.py` in a text editor and examine it. This file defines a class called `ImageViewer` that inherits from `QMainWindow`, along with a small program that creates and displays an instance of `ImageViewer`. Many of the methods in this class are **stubs** that don't contain any useful code yet, but enough has been implemented that you can run the program. Try it now and you should see a small blank window appear.

2. Implement the `_createWidgets` method by replacing the `pass` statement with the following code:

```
self.label = QLabel()
self.label.setScaledContents(True)
self.view = QScrollArea()
self.view.setBackgroundRole(QPalette.Dark)
self.view.setWidget(self.label)
self.setCentralWidget(self.view)
```

`QMainWindow` uses the notion of a 'central widget', occupying the main area of the window. This code defines that central widget to be a `QScrollArea` widget, containing a `QLabel` widget. An image will be associated with the latter.

¹ These are images from the [Rosetta](#), [Dawn](#) and [New Horizons](#) spacecraft.

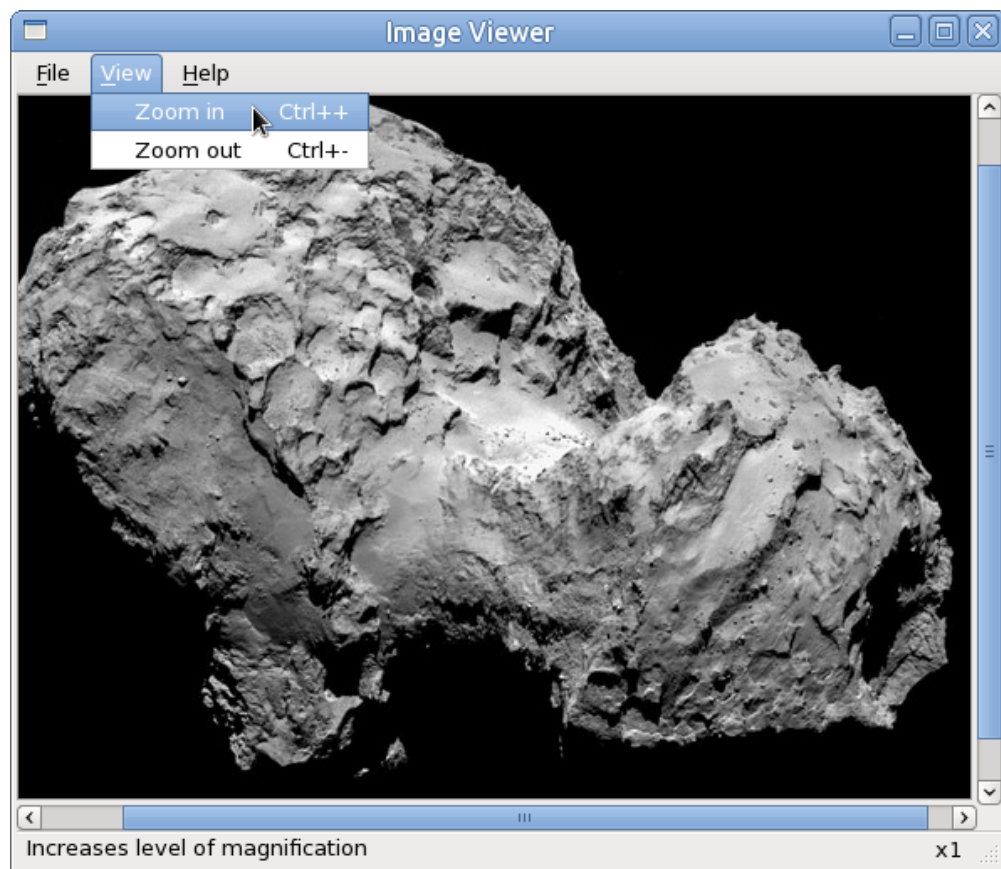


Fig. 7.2: Simple image viewer based on QMainWindow

Run the program again to check that it still creates a small blank window.

3. Add a status bar to the application by putting the following into the `_createStatusBar` method in place of the `pass` statement:

```
self.status = self.statusBar()
self.fileLabel = QLabel()
self.zoomLabel = QLabel("x1")
self.status.addWidget(self.fileLabel)
self.status.addPermanentWidget(self.zoomLabel)
```

This code gives the window a status bar containing two `QLabel` widgets. One of them, `self.fileLabel`, will be used to show the filename of the currently loaded image. The other, `self.zoomLabel`, will be used to show the current magnification factor at which the image is displayed. The latter is added as a ‘permanent widget’, meaning that it will be positioned at the far right of the status bar and will always be visible. The filename, on the other hand, will appear on the left and can be temporarily hidden - e.g., by ‘status tips’ for menu entries.

Run the program again. You should see a status bar at the bottom of the window now, with `self.zoomLabel` visible.

4. The next step is to create the **actions** that will be available on the application’s menus. To create the first two actions, replace the `pass` statement in the `_createActions` method with the following:

```
self.openAction = QAction("&Open...", self,
    shortcut=QKeySequence.Open,
    statusTip="Open an image file",
    triggered=self.openImage)

self.quitAction = QAction("Quit", self,
    shortcut=QKeySequence.Quit,
    statusTip="Quit the application",
    triggered=self.close)
```

Notice the format for creating a `QAction` object. We first specify the text that will be used in the menu and follow this with a reference to the parent of the action: the `QMainWindow` object itself. Next comes three keyword arguments, specifying the keyboard shortcut for the action, a tip to be displayed in the status bar when the action has focus and, finally, the slot to which the action’s `triggered` signal should be connected.

Now add definitions for `QAction` objects called `self.zoomAction` and `self.unzoomAction`. Specify menu entry text of "Zoom in" and "Zoom out", respectively. Specify keyboard shortcuts of `QKeySequence.ZoomIn` and `QKeySequence.ZoomOut`, likewise. Provide some appropriate text as the value of the `statusTip` keyword argument in each case, then specify slots as `self.zoom` and `self.unzoom`, respectively, for the `triggered` keyword argument.

Create the last pair of actions by adding the following code to the `_createActions` method:

```
self.aboutAction = QAction("&About", self,
    statusTip="Show information about this application",
    triggered=self.about)

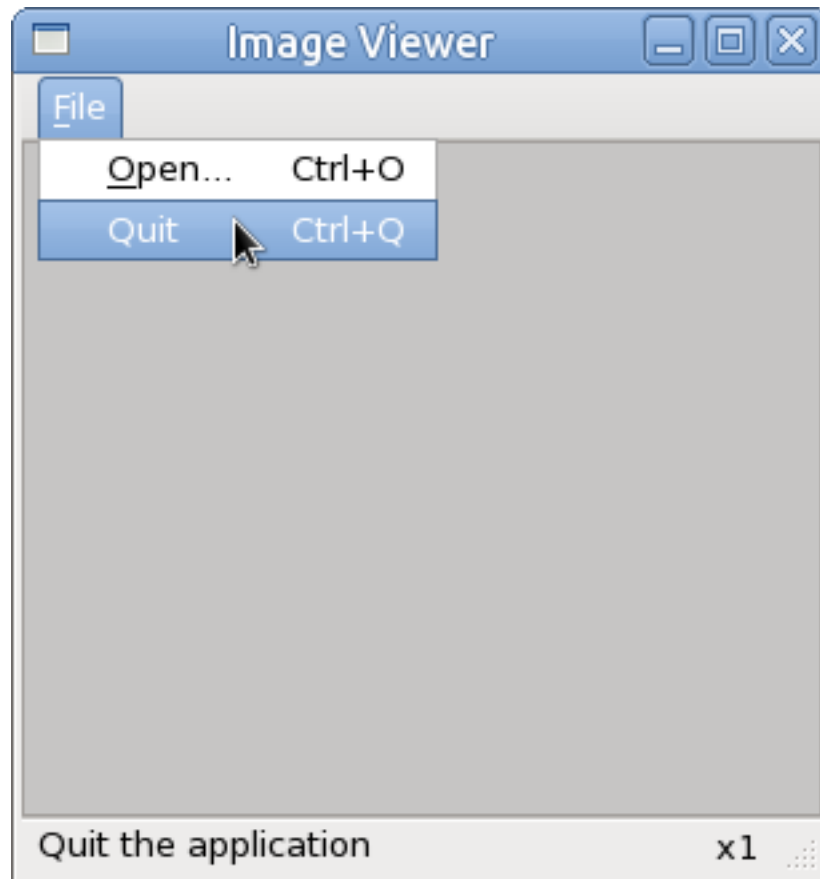
self.aboutQtAction = QAction("About &Qt", self,
    statusTip="Show information about the Qt library",
    triggered=qApp.aboutQt)
```

Try running the program again, to catch any typing errors you’ve made in the code (look and behaviour will be unchanged from before).

5. Now that the actions have been defined, it is time to create the menus that hold those actions. Create a File menu by replacing the `pass` statement in the `_createMenus` method with the following:


```
mbar = self.menuBar()
self.fileMenu = mbar.addMenu("&File")
self.fileMenu.addAction(self.openAction)
self.fileMenu.addAction(self.quitAction)
```

Run the program and interact with the File menu. You should see something similar to the screenshot below.



Note how the status tip appears in the status bar. Choosing the *Open...* action from the menu will do nothing at the moment because the slot to which it has been connected is currently a stub, but you should find that selecting *Quit* will close the window.

Use similar code to give the application a *View* menu, containing `self.zoomAction` and `self.unzoomAction`. Then give it a *Help* menu containing `self.aboutAction` and `self.aboutQtAction`. Try running the program again. The actions in the *View* menu won't be functional yet, but you should find that the actions in the *Help* menu will display suitable dialogs when triggered.

6. `ImageViewer` has two helper methods called `_updateStatus` and `_setDisplaySize` that are needed by its slots. The first is used to update the status bar with the current image zoom factor and the second is used to resize the label holding the image, according to the current image zoom factor. Add code to these methods now so that they look like this:

```
def _updateStatus(self):
    zoomText = "x{:d}".format(self.zoomFactor)
    self.zoomLabel.setText(zoomText)

def _setDisplaySize(self):
```

```
self.label.resize(self.zoomFactor*self.image.width(),
self.zoomFactor*self.image.height())
```

7. The penultimate step is to implement the `openImage` slot. Replace the `pass` statement in this method with the following code:

```
filename, _ = QFileDialog.getOpenFileName(self, "Open Image",
".", "Image files (*.jpg *.png)")
if filename:
    self.image = QPixmap(filename)
    self.fileLabel.setText(filename)
    self.label.setPixmap(self.image)
    self.zoomFactor = 1
    self._setDisplayedSize()
    self._updateStatus()
```

This code begins with a call to the `getOpenFileName` function of standard dialog class `QFileDialog`. This will open a standard dialog for selecting an image file. The string arguments are the dialog's title, the initial directory displayed by the dialog (the current directory, in this case) and a filter that restricts displayed files to those with `.jpg` and `.png` filename extensions.

If the user selects an image file, the `filename` variable will be a non-empty string and the `if` statement body will run. This will load and display the image.

Run the program now and try opening one of the provided image files. Notice how the dialog only shows these files, and how it displays small thumbnails beside their names.

8. Finally, we need to add support for zooming in and out of an image. Find the `zoom` slot and replace its `pass` statement with the the following code:

```
if self.image and self.zoomFactor < self.MAX_ZOOM:
    self.zoomFactor += 1
    self._setDisplayedSize()
    self._updateStatus()
```

Write something similar for the body of the `unzoom` slot. (Here, you will need to ensure that `self.zoomFactor` is limited to a minimum value of 1.)

Run the program, open an image file and try zooming in and out, using the menu actions and the corresponding keyboard shortcuts.

USING QT'S INTERFACE DESIGNER

Author Nick Efford

Contact N.D.Efford@leeds.ac.uk

Status Draft

Revised 2015-07-13

This exercise concerns *Qt Designer*, a visual tool that can be used to quickly create widgets and layouts, and even establish signal-slot connections. It provides a basic introduction to Designer and does not cover everything you need to know to use it effectively. For more information, you should consult the [Qt Designer Manual](#).

Note that the tasks below assume the use of Linux; you may find that keyboard shortcuts and other details are different if you are doing the work on some other platform.

8.1 Getting Started

1. Run Qt Designer from a terminal window like so:

```
designer &
```

When the 'New Form' dialog appears, select 'Widget' from the list of templates and click the *Create* button.

2. You have the option of either multiple windows or a single 'docked window' as the Designer interface, so if you are not happy with the current choice, change it by choosing *Settings* → *Preferences* and selecting the appropriate item from the 'User Interface Mode' combo box. The docked window should look something like this:
3. Spend a few minutes familiarising yourself with the features of the Designer interface. Note the blank **form** in the middle of the window. You design an interface by dragging widgets from the list on the left onto the form.

Note also the various panels to the right of the form:

- The **Object Inspector** shows the widgets and layouts of your design as a hierarchical list. You can select a particular widget or layout object either by clicking on the form or by selecting it on this list.
- The **Property Editor** allows you to view and modify the values of the various properties defined or inherited by an object in your design.
- The last panel provides three tools on different tabs. The **Signal/Slot Editor** lists connections between signals and slots. You can create them here or on the form itself. The **Action Editor** is used to define actions, which become relevant when creating a full application window with menus, etc. The **Resource Browser** allows you to manage resources such as icons that may need to be bundled with your interface.

Finally, select the *Edit* menu and note the four entries at the bottom, representing Designer's different **editing modes**.

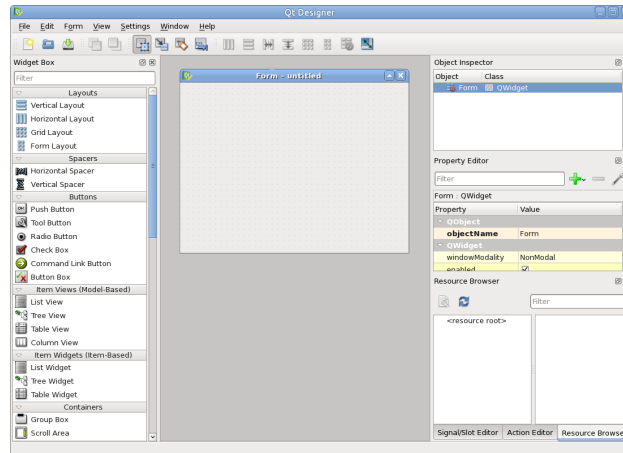


Fig. 8.1: 'Docked window' interface of Qt Designer

8.2 Simple Layout Using QWidget

As a simple example of the main features of Designer, let's explore how you would create the interface and event handling for the Volume Control application from [Handling Interaction With Signals & Slots](#).

1. Start by selecting the `QWidget` object corresponding to the form in Object Inspector. Then go to Property Editor and make the following changes to properties:

Property name	New value
<code>objectName</code>	VolumeControl
<code>windowTitle</code>	Volume Control

2. Drag 'LCD Number' from the 'Display Widgets' section of the Widget Box onto the form. Then drag 'Horizontal Slider' from the 'Input Widgets' section of the Widget Box onto the form, positioning it beneath the LCD Number widget.

Try selecting the two widgets on the form and via the Object Inspector.

3. With the Horizontal Slider selected, use Property Editor to make the following changes to properties:

Property name	New value
<code>objectName</code>	slider
<code>maximum</code>	100
<code>value</code>	50
<code>tickPosition</code>	TicksBelow
<code>tickInterval</code>	10

Then, with the LCD Number selected, use Property Editor to make the following changes

Property name	New value
<code>objectName</code>	number
<code>intValue</code>	50

4. Click on the form background so that neither widget is selected, then choose *Form* → *Lay Out in a Grid*, or press `Ctrl+5`. This has the effect of setting the parent widget layout to a `QGridLayout` with two rows containing the two widgets.
5. Preview the form by choosing *Form* → *Preview*, or by pressing `Ctrl+R`. If you try moving the slider on the previewed UI, nothing will happen because the relevant signal and slot haven't yet been connected.

- Choose *Edit* → *Edit Signals/Slots* or press F4. Move the cursor over widgets on the form and notice how each is highlighted in red as the cursor moves over it. Click on the slider and then drag upwards until the LCD Number widget is also highlighted. A red arrow will link the two widgets and the 'Configure Connection' dialog will appear.

In the dialog, select the `valueChanged(int)` signal from the list for `QSlider` on the left, then the `display(int)` slot from the list for `QLCDNumber` on the right. Then click *OK*.

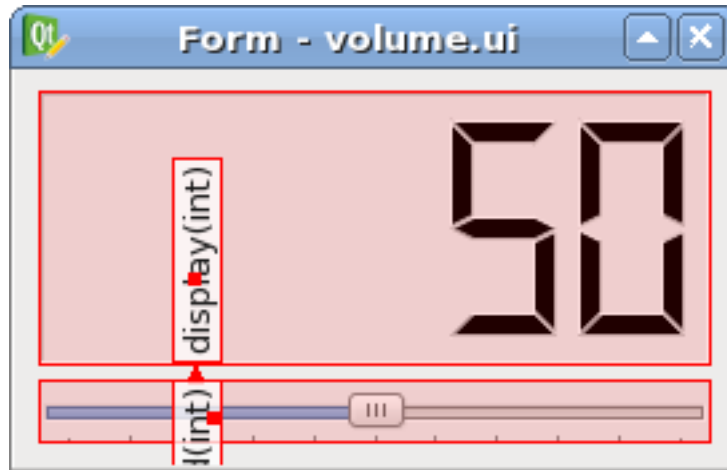


Fig. 8.2: Signal/Slot Editing Mode in Designer

Press F3 to return to widget editing mode. Preview the design again and this time, moving the slider will update the number.

- Choose *File* → *Save As* and save the form to a file called `volume.ui`, in a directory called `volume`. Examine this file outside of Designer, in a text editor. Notice the use of XML as the file format.

Clearly code will, at some point, need to be generated from this XML file. You can preview that code by choosing *Form* → *View Code*.

8.3 Using Designer-Generated Forms

8.3.1 Simple Approach

- In the directory containing `volume.ui`, Create a file called `main.cpp` containing the following:

```

1  #include <QtWidgets>
2  #include "ui_volume.h"
3
4  int main(int argc, char* argv[])
5  {
6      QApplication app(argc, argv);
7
8      Ui::VolumeControl control;
9      QWidget* window = new QWidget;
10     control.setupUi(window);
11
12     window->show();
13

```

```
14     return app.exec();
15 }
```

Here, the class `Ui::VolumeControl` is defined in header file `ui_volume.h`, which will be generated for us from `volume.ui`. This class contains all of the code to create and configure the widgets, lay them out and connect signals to slots. All we need to do is create a `Ui::VolumeControl` object and call its `setupUi` method, passing to it a pointer to the `QWidget` object that will host the UI.

2. Run `qmake -project` to generate project file `volume.pro`. Edit this file and note the lines at the end:

```
# Input
FORMS += volume.ui
SOURCES += main.cpp
```

You can see that `qmake` has recognised that the application has a form as well as a C++ file.

Add the usual `QT += widgets` line before these two lines, then save the file.

3. Run `qmake` to generate the makefile, then run `make` to build the application. Look carefully at the output that appears in the terminal window. Notice how the first command run by the makefile is called `uic`. This command is run on `volume.ui` and it generates the header file expected by `main.cpp`, before the compiler runs on `main.cpp`.

Run the application with `./volume`. It should behave the same as it did when previewed from within Designer.

8.3.2 Subclassing Approach

A better approach is to create a class that inherits from both `QWidget` and `Ui::VolumeControl`. This is more scalable and allows you to add more features, such as custom signals and slots.

1. Modify `main.cpp` so that it looks like this:

```
1  #include <QApplication>
2  #include "window.hpp"
3
4  int main(int argc, char* argv[])
5  {
6      QApplication app(argc, argv);
7
8      VolumeControl control;
9      control.show();
10
11     return app.exec();
12 }
```

2. Create a header file called `window.hpp`, containing the following:

```
1  #pragma once
2
3  #include "ui_volume.h"
4
5  class VolumeControl: public QWidget, public Ui::VolumeControl
6  {
7      public:
8          VolumeControl();
9  };
```

This declares that `VolumeControl` is a specialised `QWidget` that also contains all of the widgets and layout code defined within `Ui::VolumeControl`.

3. Now create an implementation file called `window.cpp`, containing this:

```

1  #include "window.hpp"
2
3  VolumeControl::VolumeControl()
4  {
5      setupUi(this);
6  }
```

This minimal constructor simply calls the `setupUi` method inherited from `Ui::VolumeControl`, passing it a pointer to this `VolumeControl` object; in effect, this method call is saying ‘populate this widget with the components created in Designer’.

4. Run `qmake -project` again to generate a new `volume.pro` file. This is needed because the project now consists of different source files than before. Edit `volume.pro` again and add the `QT += widgets` line. Then run `qmake` followed by `make`, as before. When you run the application, its behaviour should be unchanged.
5. Try adding a custom slot that changes number colour, as you did in Python in [an earlier exercise](#). You will need to edit `window.hpp`, making the additions highlighted below:

```

1  class VolumeControl: public QWidget, public Ui::VolumeControl
2  {
3      Q_OBJECT
4
5      public:
6          VolumeControl();
7
8      private slots:
9          void numberColour(int);
10 }
```

You will then need to make these highlighted additions to `window.cpp`:

```

1  VolumeControl::VolumeControl()
2  {
3      setupUi(this);
4      connect(slider, SIGNAL(valueChanged(int)), this, SLOT(numberColour(int)));
5  }
6
7  void VolumeControl::numberColour(int value)
8  {
9      if (value > 75) {
10         number->setStyleSheet("color:red");
11     }
12     else {
13         number->setStyleSheet("color:black");
14     }
15 }
```

Notice the variables `slider` and `number` appearing in these methods. These are the names that were given in Designer to the `QSlider` and `QLCDNumber` widgets. The names carry forward to the C++ code.

6. Do `make clean` to get rid of any generated files, then enter `qmake` followed by `make`. (Running `qmake` is necessary because you have added a custom slot, so a new makefile must be generated containing commands to run Qt’s meta-object compiler.)

When you run the compiled application, number colour should be black for values up to 75 and red for values above 75.

Note: Take a moment to think about how this application has been implemented. The widgets, their layout and some of the internal ‘wiring’ have been created rapidly using Designer. The build process turns this into usable C++ code which can be easily combined with your own C++ class representing the application. For complex UIs, this could end up being considerably quicker than hand-coding the entire interface; it certainly makes rapid prototyping easier.

The use of Designer doesn’t really limit you, as things that cannot easily be done in Designer can be implemented by hand as additional code in the class.

8.3.3 Using Forms With Python

It is possible to use Designer-generated .ui files with PyQt, as well.

1. PyQt 5 provides a tool called `pyuic5` which generates Python code from a .ui file. Run it in a terminal window like so:

```
pyuic5 volume.ui > ui_volume.py
```

Now take a look at the file `ui_volume.py` in a text editor. It defines a class called `Ui_VolumeControl`, equivalent to `Ui::VolumeControl` in the C++ example.

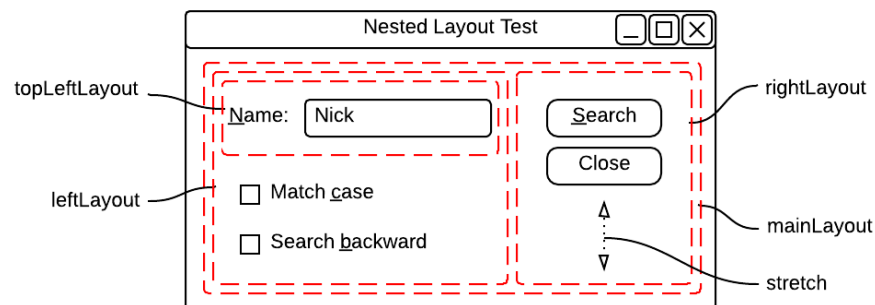
2. Download `volume.py` into the directory containing `ui_volume.py` and examine it in a text editor. This works in a similar way to the C++ example. The application is implemented as a class that inherits from both `QWidget` and `Ui_VolumeControl`. The class customises the Designer-generated UI by adding a custom slot to change number colour. The constructor calls the inherited `setupUi` method to configure the user interface and then wires up the slider to the custom slot so that number colour can change in response to slider movement.

Run `volume.py` to verify that it produces the same UI as the C++ example.

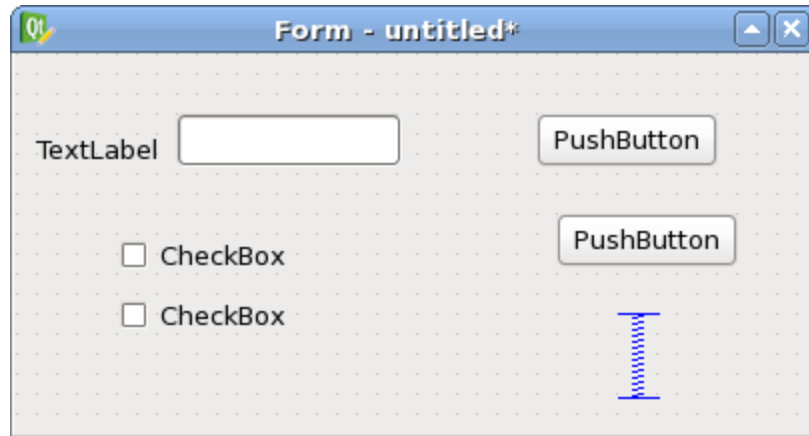
Other approaches are possible: see [Using Qt Designer](#) in the PyQt 5 documentation for further details.

8.4 A More Complex Layout

As an example of how we create more complex layouts in Designer, consider the *nested layout from an earlier exercise*:

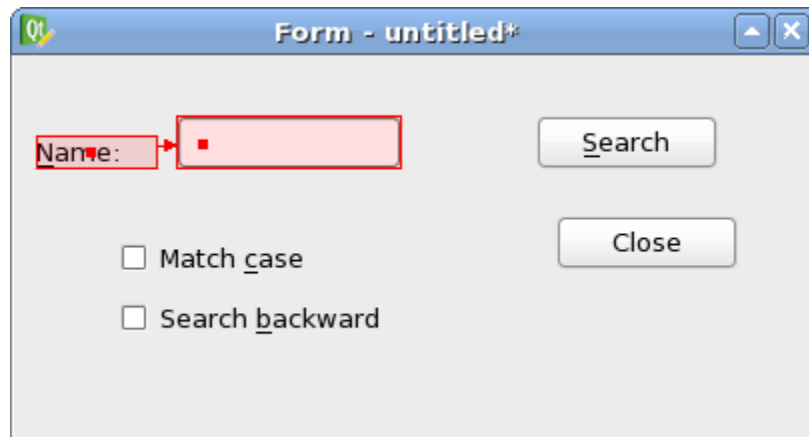


1. Close any existing forms in Designer and create a new blank form. Drag the required widgets from the Widget Box onto the form, into roughly the locations suggested by the diagram above. The ‘stretch’ noted on the diagram can be provided using ‘Vertical Spacer’, from the ‘Spacers’ section.
2. Using Property Editor, edit the `objectName` property of each widget so that it matches the variable name used in *that earlier exercise*. Then edit the text of the label, radio buttons and push buttons, so as to match the diagram.



If any of the label or checkbox text is not visible, you can use *Form* → *Adjust Size* or press `Ctrl+J` to resize the edited widget.

3. Choose *Edit* → *Edit Buddies*. Click on the Label widget and drag to the Line Edit widget:



Choose *Edit* → *Edit Widgets* or press `F3` to return to widget editing mode.

4. Now create the nested layout, from the inside out. Start with Label and Line Edit widgets. Click on the Label, then 'Ctrl+Click' on the Line Edit widget. Choose *Form* → *Layout Horizontally* or press `Ctrl+1` to put both widgets into a `QHBoxLayout`. Rename this layout object as `topLeftLayout`.

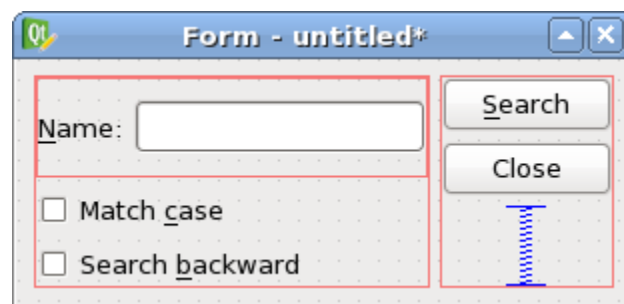
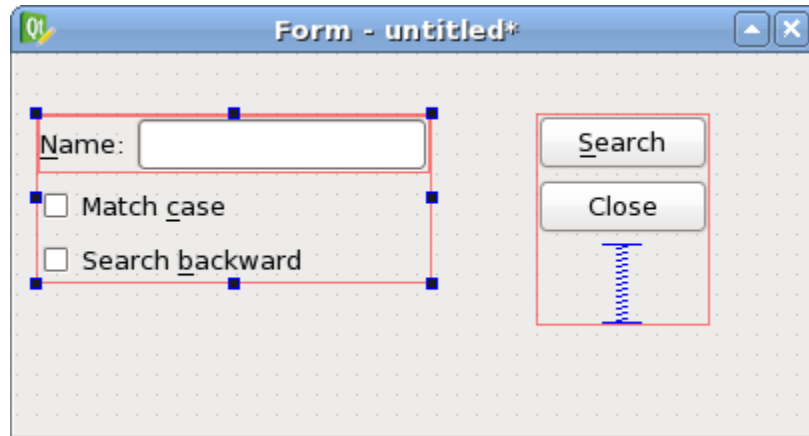
Use the same technique to select the two buttons and vertical spacer. Then choose *Form* → *Layout Vertically* or press `Ctrl+2` to put all three widgets into a `QVBoxLayout`. Rename this layout object as `rightLayout`.

Use the same technique again to put `topLeftLayout` and the two checkboxes into another `QVBoxLayout`. Rename this layout object as `leftLayout`.

5. Click on the form background, then choose *Form* → *Layout Horizontally* or press `Ctrl+1` to put the two `QVBoxLayout` objects into a final `QHBoxLayout` object and make this the top-level layout of the widget.

Finally, choose *Form* → *Adjust Size* or press `Ctrl+J`. You should end up with something like this:

Try previewing it with *Form* → *Preview* (or `Ctrl+R`) to check what the user would see.



GETTING STARTED WITH ANDROID DEVELOPMENT

Author Nick Efford

Contact N.D.Efford@leeds.ac.uk

Status Draft

Revised 2015-11-16

The goal of this exercise is to acquaint you with the Android Studio development environment and verify that you are able to run simple applications in the emulator and on a real device. Instructions are given below on accessing Android Studio from SoC Linux machines. You can obtain it for your own PC from <http://developer.android.com/sdk/>

9.1 Preparation on SoC Linux Machines

Warning: You will only need to do this once, and **you will not need to do this on your own PC.**

1. Open a terminal window and do `ls -al`. Examine the directory listing and check that there are entries named `.android`, `.AndroidStudio1.4` and `.gradle`. (The first two should be directories and the last should be a symbolic link.)

If these entries are present, then proceed to Step 2.

If none of these entries are present, download the shell script `setup.sh` to your filestore. In a terminal window, `cd` to where you downloaded it and then run the script by entering the following command:

```
bash setup.sh
```

2. Now edit your `~/.bashrc` file and add the following line to the bottom of it:

```
module add android-studio
```

Start up a new terminal window and enter `android-studio`. Android Studio will start up and take you through a series of configuration steps:

- When asked whether you wish to import settings from a previous version of Android studio, choose the ‘I do not have a previous version’ option.
- For the next step, select the ‘Custom’ setup option, then choose whichever theme you prefer. Dark and light theme options are provided.
- Next, you’ll be prompted to define SDK components. Leave these settings at their defaults.
- **IMPORTANT:** When asked to specify an SDK location, use this:

```
/vol/scratch_2/mobile-app-dev/username/sdk
```

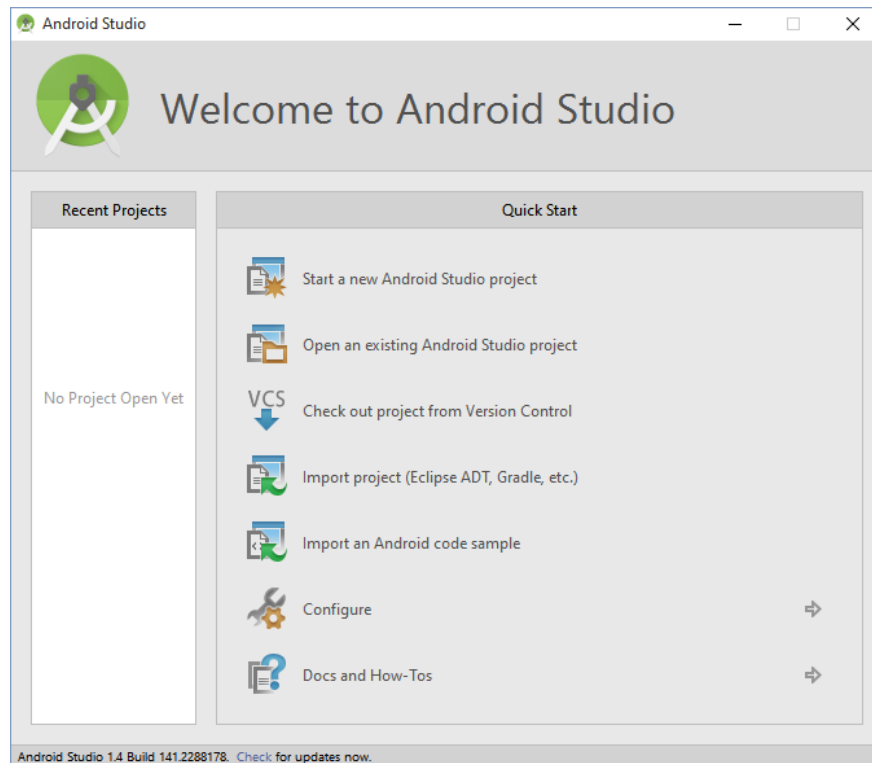
You should substitute your actual username for `username`, obviously! You can ignore the error messages that appear as you type this in, but if you’ve entered the path correctly, these should eventually disappear. You might see a message saying something like “this folder contains an existing SDK, I’ll only download updates”, which is OK. At this point you can click *Next*, followed by *Next* again on the verification screen.

- You’ll see a final screen about emulator settings, which you can ignore. Click *Finish* at this point to complete setup.

9.2 Further Configuration

Note: Start here if you are running Android Studio on your own PC.

1. If Android Studio isn’t currently running, run it now. After a splash screen and a short delay, you should see the welcome window on screen.



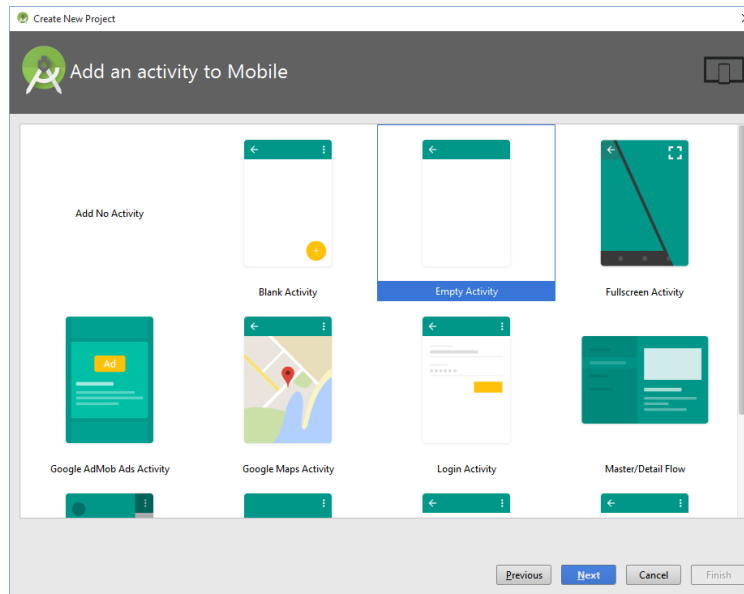
2. Choose ‘Configure’, then ‘Settings’. Spend some time exploring the different configuration possibilities. You may wish at this point to alter the theme for the IDE, change the font used to display code, set preferences for code formatting and syntax highlighting, etc. Return to the welcome screen when you are done.

9.3 Creating an Android Project

1. Choose ‘Start a new Android Studio project’. Enter ‘Echo App’ for the application name and specify either a domain you own or `example.com` as the company domain. Alter the path to the project location if you don’t

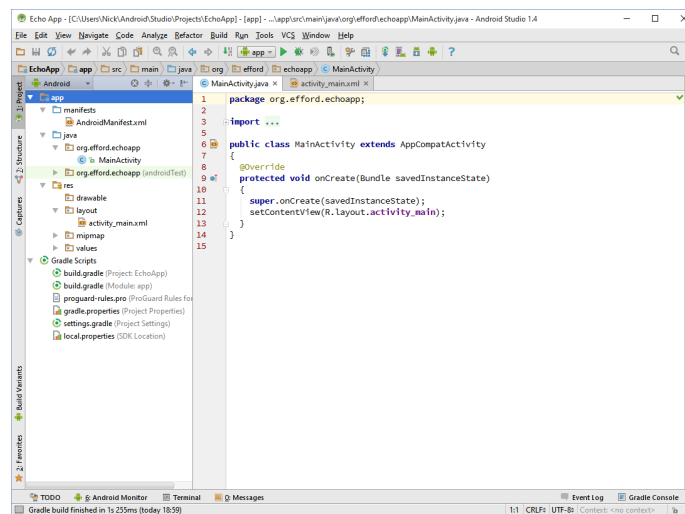
like the one that Studio has chosen for you, then click *Next*.

- On the next dialog, make sure that only the 'Phone and Tablet' checkbox is checked. Select a minimum SDK of 'API 21: Android 5.0 (Lollipop)', which will support the Nexus tablets in ENIAC. (Use a lower API level if you have wish to use your own Android device and it is running an older version of the OS.) Click *Next*.
- Next you will be prompted to select an activity template. Choose 'Empty Activity', then click the *Next* button.



On the next dialog, leave the activity name set to `MainActivity` and make sure the 'Generate Layout File' option is checked. Then click *Finish*.

- You may need to wait a while the first time that Studio starts up, but eventually the full IDE should be displayed. You will most likely see something like that screenshot below, in which the Java code for your activity is displayed in the editor.



Spend a few minutes exploring the various menus. Hover the cursor over the buttons on the toolbar until a tooltip pops up telling you what they do.

- Next, have a look at the sideways-oriented tabs on the far left. You can click on the *Project* tab (or press `Alt+1`) to toggle the visibility of a panel that shows you the files in the project. This panel can present different views of the file structure, according to what is selected from the combo box at the top of the panel. Try the 'Project' and 'Android' views. Leave the latter enabled.

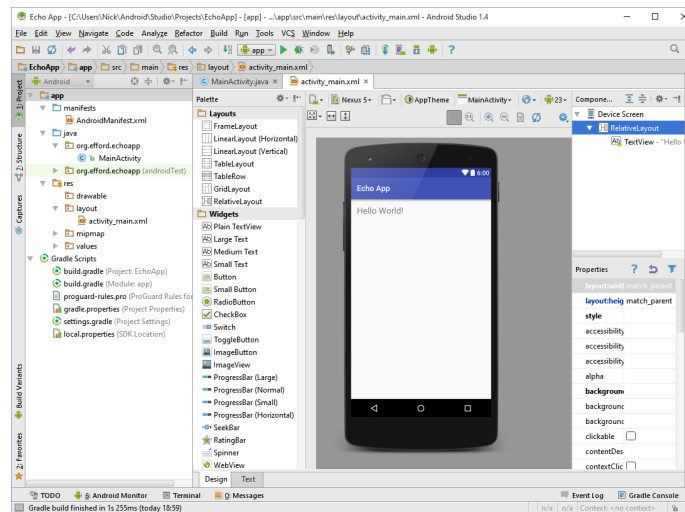
Notice how the 'Android' view gives you convenient access to the manifest (under `app/manifests`), your Java code (under `app/java`) and your app's resources (under `app/res`).

- In the Project panel, expand the 'Grade Scripts' item under 'app'. Notice the two files called `build.gradle` listed there. Double-click on each to examine their contents.

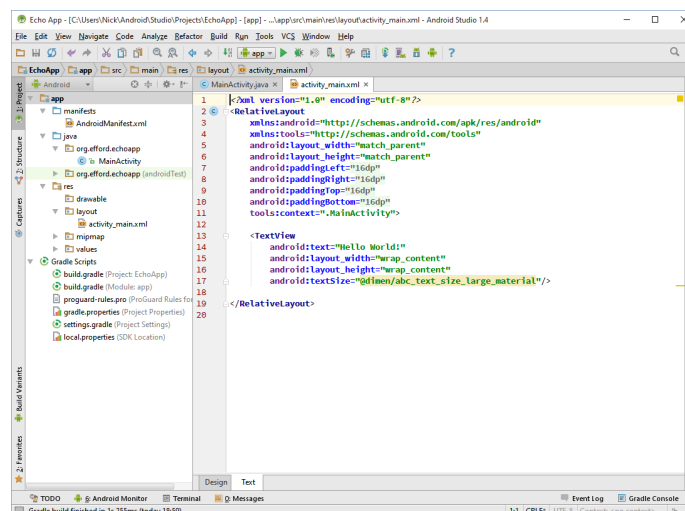
Android Studio used a build tool called **Gradle** to manage compilation and packaging of your apps. These files are used to configure the build process.

- Finally, locate `res/layout/activity_main.xml` in the Project panel and double-click on it. This gives you two different views of the UI associated with an activity. You can switch between them by click on the 'Design' or 'View' tabs at the bottom of the editing area.

The Design View shows the UI visually and allows you to develop it via drag-and-drop:



The Text View, on the other hand, shows the UI as an XML file. This is how details of widgets and layout are actually stored in your app.



The visual and XML representations are synchronised; you can make changes to either one and the other will be updated accordingly.

9.4 Running in an Emulator

The emulator allows you to test your apps if there is no real Android device available to you.

Warning: The emulator is **slow**. If you are working in ENIAC then we recommend that you simply run your apps on the Nexus 7 tablets that are plugged into the PCs.


If you do use the emulator, you should leave it running for the duration of your session so that you don't have to wait minutes for it to boot up every time you want to test your app.

On your own PC, you should make sure that hardware acceleration is enabled if possible. Android Studio should provide platform-specific advice on this as part of the installation process.


1. **If you are running Android Studio on your own PC, you may wish to skip this first step**, as you will already have a Nexus 5 AVD available to you as part of the installation process.

In your web browser, go to the [Run on the Emulator](#) section of the official Android [Building Your First App](#) tutorial. Follow the instructions there to create and start an AVD.

On SoC machines, we recommend that you choose 'Nexus 7' from the Tablet options so as to match the physical devices available in ENIAC. Also, tick the 'Use Host GPU' emulation option.

2. Choose *Run* → *Run App* or click  on the toolbar to run the app. A 'Device Chooser' dialog should appear. If the emulator is already running, select 'Choose a running device' and choose it from the available options; otherwise, select 'Launch emulator' and choose an appropriate AVD via the combo box. Then click *OK*. After a delay (which may be reasonably lengthy if the AVD has to boot up), the "Hello World!" screen of your app should appear in the emulator window.
3. In Studio, you should see the Run panel at the bottom of the IDE, showing messages logged by the emulator at it started up. You can click on the *Run* button at the bottom of the window or press `Alt+4` to toggle the visibility of the panel.

You can also use the *Android Monitor* button or `Alt+6` to toggle the visibility of the Android Monitor panel. This allows you to see messages logged by the app as it runs and examine use of resources such as memory, CPU and network bandwidth.


4. In the emulator, click on the 'Home' button (the 'disc' in the middle of the three buttons at the bottom of the screen). Then click on the 'Apps' button (the larger disc appearing just above the home button). Notice how 'Echo App' appears in the set of installed apps.
5. Click  on the Android Monitor panel to shut down the app. You can shut down the emulator via *Tools* → *Android* → *AVD Manager*, by selecting the relevant AVD and choosing 'Delete' from the drop-down list of options.

You can learn more about AVDs by reading the official Android documentation on [Managing Virtual Devices](#) and [Managing AVDs with AVD Manager](#).

9.5 Running on a Real Device

You will need to be in ENIAC lab if you wish to try this from a School of Computing PC. Make sure you are sitting at a machine that has a Nexus 7 tablet attached to it.

If you are doing this with your own PC and Android device, consult the [Run on a Real Device](#) section of the [Building Your First App](#) tutorial and follow the instructions given there.

1. Wake the tablet up by pressing the power button at the top-right edge of the device. If nothing happens because the tablet has been shut down, hold the button in for a few seconds, then wait for the tablet to boot up properly.
2. Choose *Run* → *Run App* or click  on the toolbar. When the ‘Device Chooser’ dialog appears, select ‘Choose a running device’ and then choose the device from the available list. The tablets in ENIAC will have serial numbers beginning with `asus-nexus_7`.

After a short delay, during which the app is packaged and deployed, the “Hello World!” screen should appear on the tablet.

9.6 Further Reading

For further information, see the [Building Your First App](https://developer.android.com/tutorials/building-your-first-app) tutorial at developer.android.com.

DEVELOPING A SIMPLE ANDROID APP

Author Nick Efford

Contact N.D.Efford@leeds.ac.uk

Status Draft

Revised 2015-11-15

The goal of this exercise is to acquaint you with process of using Android Studio to create a simple, single-activity app featuring multiple widgets and some event handling.

Warning: Make sure that you have completed [Getting Started With Android Development](#) successfully before starting this worksheet!

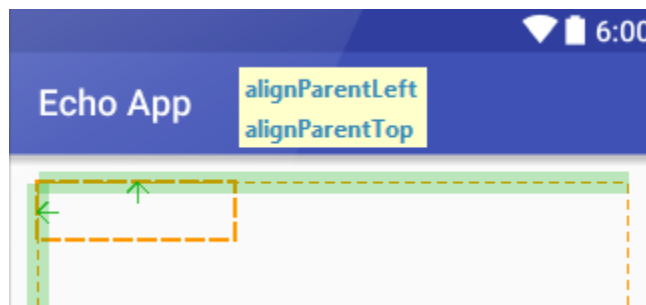
10.1 Widget Creation & Layout

1. Start Android Studio and open the ‘Echo App’ project created in the previous worksheet. Find `res/layout/activity_main.xml` in the Project panel and double-click on it to open the file.

Switch to the Design View if this isn’t currently active, then click on the ‘Hello World!’ text and remove it by choosing *Edit* → *Delete* or pressing *Delete*. Switch to the Text View and you should see that the XML markup for the widget has disappeared.

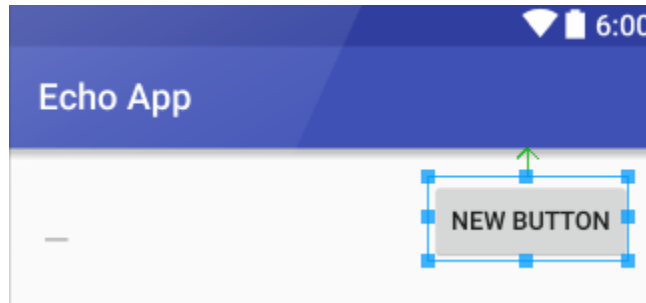
2. Switch back to the Design View. On the Palette, find the Text Fields folder and drag the item labelled ‘Plain Text’ onto the app screen. This is, in fact, an `EditText` widget. (The names on the Palette don’t correspond exactly to widget names.)

Position the widget so that it snaps into the top-left corner of the enclosing layout.



(Note: this and the other screenshots show the design rendered for the default virtual device, a Nexus 5. It won’t look quite the same if you’ve selected a different device.)

3. Drag the item labelled 'Button' from the Widgets folder in the Palette onto the Design View of your app, positioning it so that it snaps into the top-right corner of the enclosing layout.



Switch to the Text View to see how the XML markup for the layout has changed.

4. Experiment with editing the text on the button. You can do this either by double-clicking on the button or by editing the `text` property in the Properties panel at the lower-right of the window. Try both methods. Change the text to 'Send'. Ignore any warnings about the use of a hard-coded string, as you will be fixing this later.
5. Fix up the layout by selecting the `EditText` widget and then dragging its right-hand edge until it snaps onto the left-hand edge of the button.



6. Experiment with changing the virtual device used to preview the layout. You can choose different devices via a drop-down list of options at the top of the Design View. If you are in ENIAC and will be running the app on a real device (see below), check how the layout looks on a Nexus 7.
7. Run the app in the emulator or on a real device, as described in the previous worksheet. You should find that it is possible to enter text using the on-screen keyboard, but that clicking on the Send button does nothing.

Terminate the app before proceeding further (but leave the emulator running if you are using it instead of a real device).

10.2 Adding an Event Handler

1. Find `MainActivity.java` in the Project panel and double-click on it to open it in the editor. Begin defining a new method called `handleButtonClick` like so:

```
public void handleButtonClick(View view)
```

As soon as you enter `View`, Studio should flag this as an unrecognised name and offer to import the class from the `android.view` package. Accept its suggestion by pressing `Alt+Enter` when prompted.

Complete the rest of the implementation as shown below, accepting the suggested imports for `EditText` and `Toast` when prompted.

```
public void handleClick(View view)
{
    EditText et = (EditText) findViewById(R.id.editText);
    Toast toast = Toast.makeText(this, et.getText(), Toast.LENGTH_SHORT);
    toast.show();
}
```

This code creates a **toast** - Android's term for a small temporary pop-up message - using the text from the EditText widget.

Note how the reference to the widget is obtained. This is necessary because the widget is defined in XML, not in Java. In effect, we 'look up' the object via a unique identifier that is specified in the XML file.

Note: For this to work, the `android:id` attribute in `activity_main.xml` *must* have the matching value of `@+id/editText`. You should verify this now and correct any inconsistencies before attempting to run the app again.


2. In the Design View, click on the button to select it, then find the `onClick` property in the Properties panel. You should be able to choose the newly-created `handleButtonClick` method from a drop-down list to the right of the property name.

Now try running the app again. After clicking the Send button, you should see the entered text appear briefly at the foot of the screen.

10.3 Specifying Strings Properly

1. In the Design View, select the Send button. You should see a small 'light bulb' icon appear beside it. Click on this and a pop-up message should appear, warning of the use of a hard-coded string and suggesting the use of an `@string` resource instead.

Click on the pop-up message. When the Extract Resource dialog appears, enter `button_text` as the resource name and click *OK*.

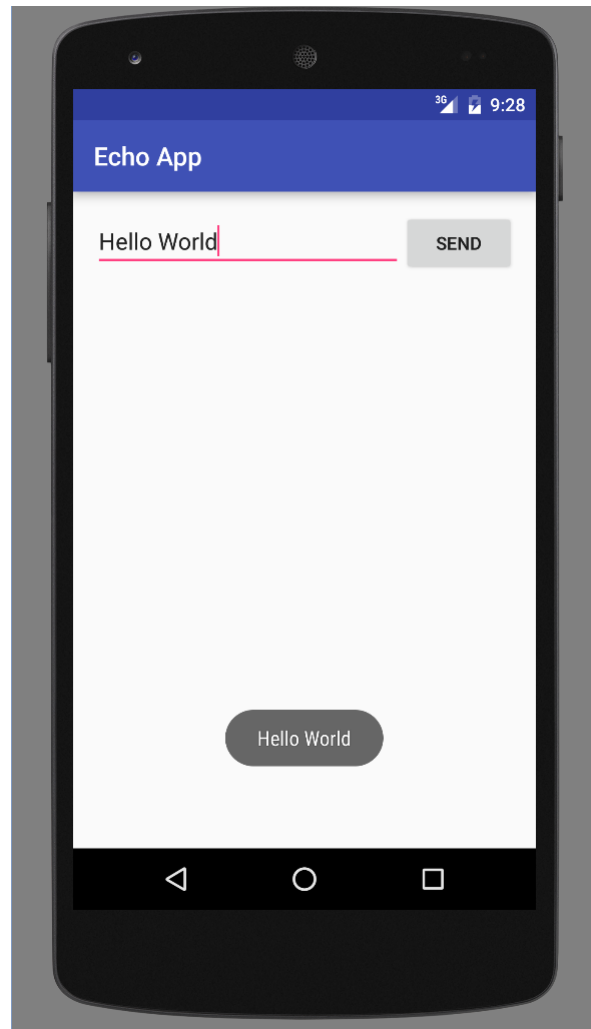
Note: If the button text changes and you see a rendering error, simply click  on the toolbar at the top of the Design View to clear the problem.

2. In the Project panel, find `res/values/strings.xml` and double-click on it to open it in the editor. You should see something like this:

```
<resources>
    <string name="app_name">Echo App</string>
    <string name="button_text">Send</string>
</resources>
```

Now check `activity_main.xml`. The `android:text` attribute of the button widget should now reference the `button_text` string resource, like this:

```
android:text="@string/button_text"
```



LINKING ACTIVITIES USING INTENTS

Author Nick Efford

Contact N.D.Efford@leeds.ac.uk

Status Draft

Revised 2015-11-15

The goal of this short exercise is to show you how one activity can start another and pass data to it using an intent. It is a simplified version of the [Starting Another Activity](#) lesson from the training materials on the Android developer web site.

Warning: Make sure that you have completed [Developing a Simple Android App](#) successfully before starting this worksheet!

11.1 Modifying The Existing App

1. Start Android Studio and open the 'Echo App' project developed in the previous worksheet. Modify the code of `MainActivity` so that the variable representing the `EditText` widget is a field (instance variable) of the class, rather than a local variable of the event handling method. The changes you need are highlighted in the code below.

```
1  class MainActivity ...
2  {
3      private EditText messageWidget;
4
5      @Override
6      protected void onCreate(Bundle savedInstanceState)
7      {
8          super.onCreate(savedInstanceState);
9          setContentView(R.layout.activity_main);
10         messageWidget = (EditText) findViewById(R.id.editText);
11     }
12
13     public void handleButtonClick(View view)
14     {
15         Toast toast = Toast.makeText(this, messageWidget.getText(), Toast.LENGTH_SHORT);
16         toast.show();
17     }
18 }
```

Generally speaking, this is the approach you should follow with your apps. Any widget that needs to be accessed from Java code should be defined as a field and initialized in `onCreate` using `findViewById`.

Note: this initialization must take place *after* the call to `setContentView`.

2. Run the app to make sure it still works, then remove (or comment out) the code inside `handleButtonClick`.

11.2 Creating The Intent

1. Add the following string constant to `MainActivity`, just inside the class definition:

```
public static final String MESSAGE_KEY = "com.example.echoapp.MESSAGE";
```

Note: this assumes that you specified `example.com` as the ‘company domain’ when creating the project; if you specified something different then alter it accordingly.

`MESSAGE_KEY` is a unique string that acts as a key for the data that will be stored in the intent. (Intent data are stored as key-value pairs.)

2. Now add code to `handleButtonClick` that will create an intent, populate it with the text that the user has entered into the `TextEditText` widget and then use the intent to start another activity. You should assume that this activity is represented by a class called `DisplayActivity`, yet to be created:

```
Intent intent = new Intent(this, DisplayActivity.class);
String message = messageWidget.getText().toString();
intent.putExtra(MESSAGE_KEY, message);
startActivity(intent);
```

Studio will flag the non-existence of `DisplayActivity` as an error, so the next step is to fix that by creating `DisplayActivity`.

11.3 Creating The Secondary Activity

1. Choose *File* → *New* → *New Activity* → *Empty Activity*. In the dialog, specify `DisplayActivity` as the name and untick the ‘Generate Layout File’ option, then click *Finish*.
2. Edit `DisplayActivity.java`. Modify the `onCreate` method by adding the following code, after the call to `super.onCreate`:

```
Intent intent = getIntent();
String message = intent.getStringExtra(MainActivity.MESSAGE_KEY);

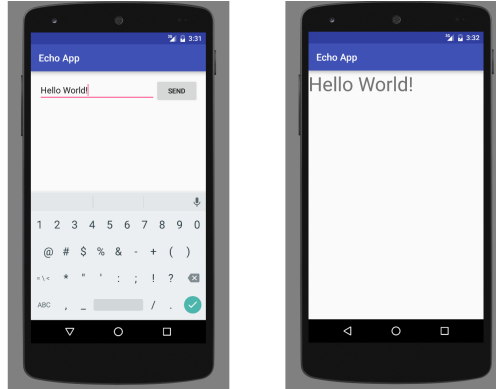
TextView view = new TextView(this);
view.setTextSize(40);
view.setText(message);

setContentView(view);
```

The first two lines obtain the intent that triggered this activity and extract the message string from it. The rest of the code creates a `TextView` widget containing the message and makes this the entire user interface for the activity. (This simplicity is the reason why XML is not used in this case.)

11.4 Testing The App

Test your app in the emulator - e.g., using one of the Nexus 7 tablets in ENIAC, or the emulator. You should see screens resembling those shown below (which are for a Nexus 5):



Try pressing the back button (the triangle from the set of three buttons displayed at the bottom of the screen) after a message has been displayed; you should find that it returns you to the main activity of the app.

11.5 Other Things To Try

- Examine the **manifest** for this app. You can find this via the Project panel; if you have 'Android' selected from the list of possible project views, the manifest file will be listed as `AndroidManifest.xml` under 'manifests'.

Another quick way of finding the manifest is to click on the small icon in the left margin next to the start of the definition of `MainActivity`. This will pop up a 'Go To Related Files' menu, from which you can choose `AndroidManifest.xml`.

When you look at the manifest, notice how both activities are declared. All components used in an app must be declared in this way. Notice also how `MainActivity` advertises itself as the activity that should be run when the app is launched.

- Try modifying the app so that it uses an XML layout file for `DisplayActivity` instead of creating the screen entirely in Java. You can create this layout file by choosing *File* → *New* → *XML* → *Layout XML File*.