Exercises Lecture 3 – Layouts and widgets

Aim: This exercise will help you explore and understand Qt's widgets and the layout approach to designing user interfaces.

Duration: 2h

Widgets Manually

Start by creating a new, empty Qt 4 project in Qt Creator. Add the following to your project:

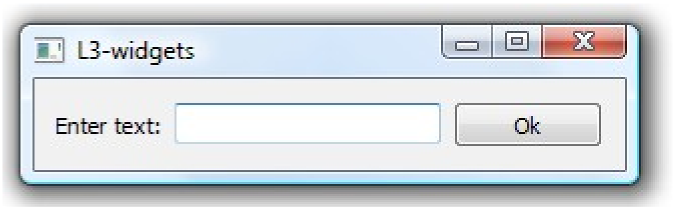
- A C++ class, `ManualWidget`, inheriting from `QWidget`
- A C++ source file, `main.cpp`

In `main.cpp`, create a main function and include the `ManualWidget` header file. In your `main` function, implement the following function body. You might have to add more include files to meet all dependencies.

```
{
    QApplication a(argc, argv);
    ManualWidget mw;
    mw.show();
    return a.exec();
}
```

Running this code should produce an empty window. Closing the window terminates the application.

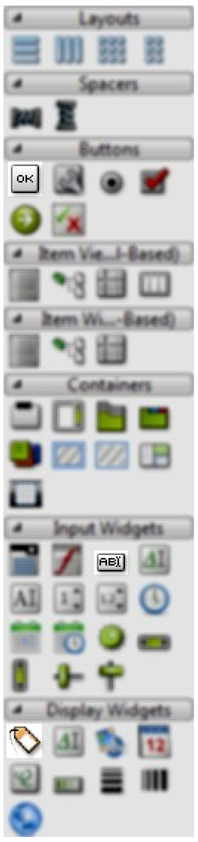
The next step is to place a set of widgets inside layouts in the window, creating a basic dialog. All this takes place in the constructor of the `ManualWindow` class. The goal of this exercise is shown below.



The contents of the window is a `QLabel`, a `QLineEdit` and a `QPushButton`. Start by creating the three widgets, then add them all to a horizontal box layout (`QHBoxLayout`) that is applied to the `ManualWidget` instance itself (i.e. `this`).

Running the code should yield the result shown above.

Widgets Using Designer



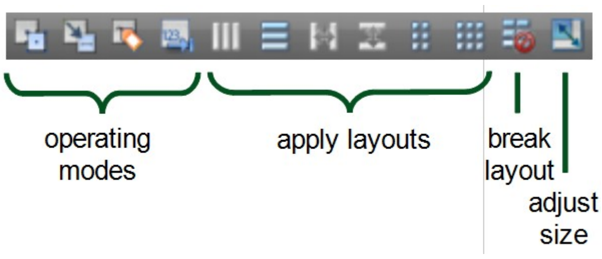
Keeping the project from the previous step, add a new Qt Designer Form Class to the project. Ensure the following:

- The form template is a widget
- The class is called `DesignerWidget`

Now, open the form in Designer. Drag and drop a `QLabel`, `QLineEdit` and `QPushButton` onto the form and apply a horizontal layout (from the toolbar on the top of the screen, see legend further down). You can see the widgets highlighted in the illustration to the left.

To apply a layout, select the containing widget (i.e. the form itself) and click on the appropriate layout button in the toolbar.

To alter the text of the label and button, simply double click on the widget.



If the widgets don't arrange themselves properly in the layout, you can either break the layout using the break layout button in the toolbar, or drag and drop the widgets within the layout to move them about.

To compact the size of the form, adjust the size using the toolbar button.

You can try out the design using the form preview function reached using Ctrl+Alt+R, or *Tools --> Form editor --> Preview...*. Using the latter, you can also choose to preview the form using different styles such as Plastique, ClearLooks, etc.

When you are happy with the result, add the following lines to your main function, just before the call to `exec`.

```
    ...
    DesignerWidget dw;
    dw.show();
    return a.exec();
}
```

Running the resulting code should produce two more or less identical windows – only the window title telling them apart (unless you changed it in Designer).

Try looking at the code how the user interface is handled through the `ui` class variable. The class for `ui` is forward declared in the header, instantiated in the constructor and deleted in the destructor. You can read more on how you can use Designer user interfaces from your code here: http://doc.trolltech.com/4.6/designer-using-a-ui-file.html.

Reacting and Accessing Manually

Building on the code from the last two steps, you will now add code to react to user actions.

The goal is to let the user enter some text and press the button. This will result in a dialog with the text and a short message.

To achieve this, you need to start by connecting a slot to the `clicked()` signal of the button. Start with the `ManualWidget`.

As the message to the user is supposed to contain the text entered by the user, a pointer to the `QLineEdit` widget must be kept as a class variable (otherwise it will not be accessible outside the constructor). Add that variable and call it `m_lineEdit` and update the constructor to use it as the pointer to the widget.

If you need to access the contents of a widget outside the constructor, you need a class variable pointer to it. Passive widgets (e.g. labels and frames) and widgets only used to trigger slots (e.g. buttons) can be left to Qt's parent-child memory management mechanism.

In the header file, add a private slot called `showMessage()`. Implement the slot with the following body.

```
{
    QMessageBox::information(this, "Message", "The text entered in the "
        "manual widget window is:\n" + m_lineEdit->text());
}
```

Now add a connection in the constructor by calling `connect`. Connect the `clicked()` signal from the push button to the `showMessage()` slot of `this`.

Running the code should now give you a working manual widget window.

However, if the user enters some text and presses return, the dialog does not pop up. The user is required to click the button (or use tab to reach it and then activate it). Luckily, the `QLineEdit` class provides the signal `returnPressed()`. Connect it to the same slot and the window behaves as expected.

Reacting and Accessing Using Designer

This step is about reproducing what you just did with the `ManualWidget` for the `DesignerWidget` class. It will highlight some of the differences, but also the similarities between the two approaches to user interface implementation.

Start by opening the form in Designer. Right click on the push button and pick the *Go to slot...* menu option. This results in a list of available signals. Pick `clicked()` from it. This results in the creation of the `on_pushButton_clicked()` slot.

In the slot body, enter the following code.

```
    QMessageBox::information(this, "Message", "The text entered in the "
        "designer widget window is:\n" + ui->lineEdit->text());
```

Notice that all widgets in the design are available through the `ui` variable.

To support the user pressing return when in the line edit widget you have two options.

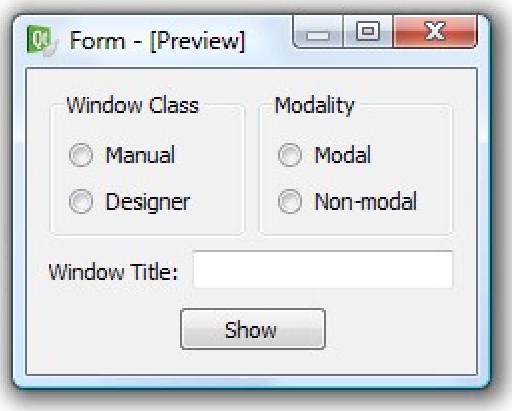
- Add a connect call in the constructor connecting the `returnPressed()` signals with the `on_pushButton_clicked()` slot
- Add a new slot through Designer and call `on_pushButton_clicked()` from that slot

Implement one of these solutions and test the code. The resulting two windows, the `ManualWidget` and the `DesignerWidget`, should behave identical to the user.

Container Widgets

When designing user interfaces, grouping related options together helps the user understand the context better. From a Qt memory hierarchy perspective, the interface goes from a flat structure with a single parent to all widgets, to a tree where widgets contain other widgets.

Use the same project as in the previous steps and add another Qt Designer form class. Again base it on a `QWidget` and name it `MultiChoiceWindow`. The goal is to populate the widget to look like the illustration below.



Start the design process by dragging two `QGroupBox` widgets onto the form. Change their text properties to “Window Class” and “Modality”.

> Group boxes are widgets that contain other widgets. These widgets are referred to as container widgets.

In each group box, add two `QRadioButton` widgets and alter their texts according to the illustration above.

Beneath the two group boxes, add a `QLabel` and a `QLineEdit`. Change the text of the label to “Window Title:”.

Finally, add a `QPushButton` at the very bottom of the dialog with the text “Show”.

Now, apply layouts from the inside out, i.e. layout the interior of container widgets before placing the container widgets themselves in a layout.

Apply vertical layouts to the group boxes.

Place the two group boxes in a horizontal layout. Do this by selecting the two widgets (click to select the first, Ctrl+click to select more) and then apply the layout. You can place layouts inside layouts, thus building hierarchies of layouts.

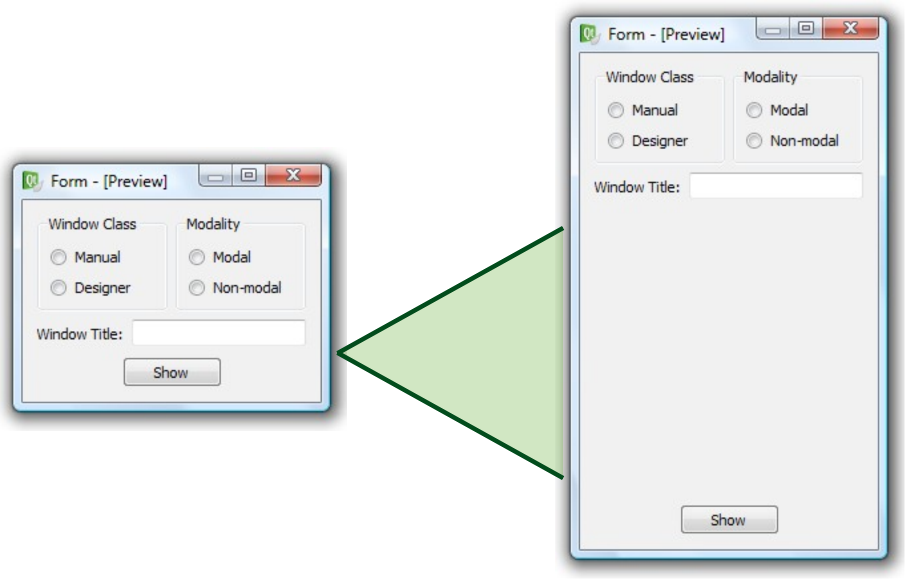
Place the label and line edit in another horizontal layout.

Apply a vertical layout to the entire form and then adjust the size of the form.

Previewing the widget, the result looks similar to the illustration above, but the button does not look right.

Break the layout of the form (select the form and click the break layout button on the toolbar), then place the button and two horizontal springs in a horizontal layout before reapplying the vertical layout to the form.

Previewing the widget now looks OK, but when resizing, the stretch does not look good. Drag and drop a vertical spring into position to make the widget stretch as shown below (you do not have to



break any layout to achieve this, but it might be easier).

The design is now complete, but we must still name the widgets we plan to access. By selecting the widgets one by one, the `objectName` can be changed using the property editor. Name the widgets according to this table.

Widget	Name
Radio button "Manual"	`radioManual`
Radio button "Designer"	`radioDesigner`
Radio button "Modal"	`radioModal`
Radio button "Non-modal"	`radioNonModal`
Line edit	`titleEdit`

Also, make sure that the `radioManual` and `radioModal` are checked, i.e. the `checked` property is set.

When all the settings have been applied, continue by creating a slot for the push button's `clicked()` event (using the context menu).

In the slot function, add the following code. You will have to include the headers for the two widget classes in the file as well.

```
void MultiChoiceWindow::on_pushButton_clicked()
{
    QWidget *w=0;

    if(ui->radioManual->isChecked())
        w = new ManualWidget();
    else
        w = new DesignerWidget();

    if(ui->radioModal->isChecked())
        w->setWindowModality(Qt::ApplicationModal);

    w->setWindowTitle(ui->titleEdit->text());
    w->show();
}
```

This code creates the requested window, sets it modality and title before showing it.

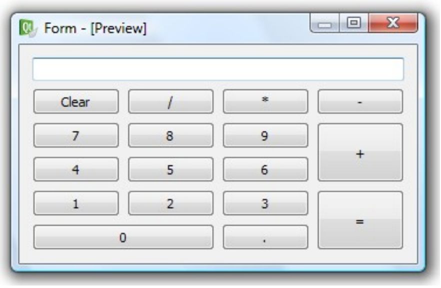
The last step on the way is to alter the main function to show the `MultiChoiceWindow` instead of the `ManualWidget` and `DesignerWidget`.

Layouts and Size Policies

Layouts and size policies work together to negotiate sizes and locations for the widgets of a window. In this step you will create two dialogs using different approaches. The goal is to learn about the design process, so we will not use the windows from any actual code. Instead, we will use Designer to create the interfaces and them preview them, again using Designer, to ensure that they look and behave as expected.

The Calculator

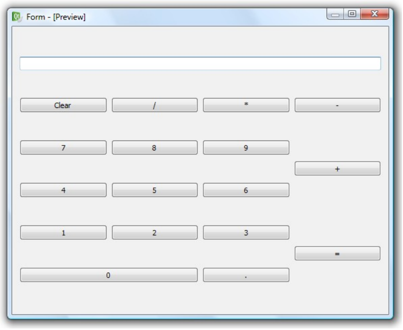
Using the project from the previous step, start by adding a new Qt Designer Form to the project. Call it `calculator.ui` and use the QWidget template. The first design that you will create is the calculator interface shown here.

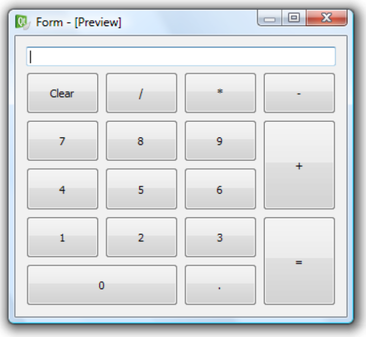


Start by dragging the required `QPushButton` widgets and the `QLineEdit` onto the form and place them roughly in the right place. Ensure that the buttons 0, =, + and the line edit widget are stretched to span over multiple columns or rows.

When you are happy with the placement of the buttons, apply a grid layout to the entire form.

When previewing the form you will have a result stretching like shown to the left below, which the expected behavior is shown to the right.



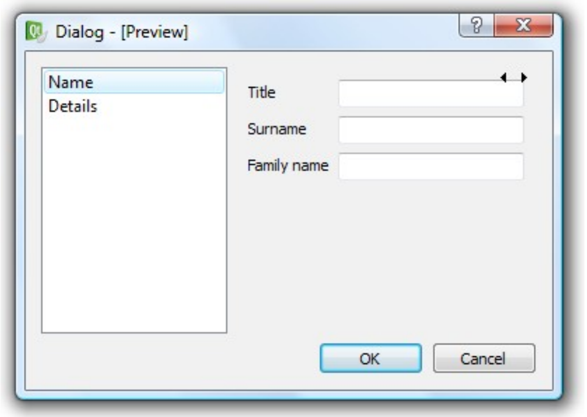


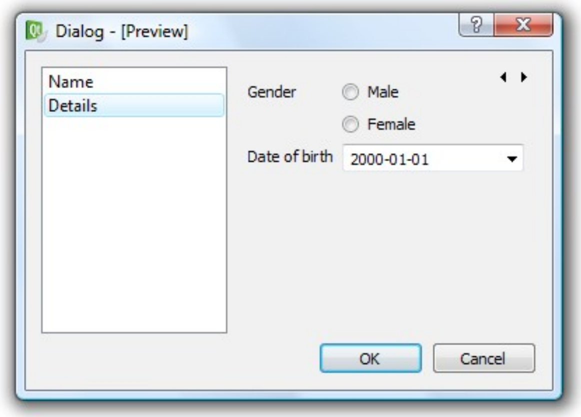
Discuss and explain why the buttons do not stretch in the vertical direction.

Try altering the vertical size policy property of the buttons to Minimum. You can do this for all buttons at once if you select them all together. Now preview the design again. Discuss and explain the changes and why it works. What happens if the `QLineEdit` is given a vertical `Expanding` policy? Why?

Invisible Pages

The next dialog is a slightly more complex design: a configuration dialog with two different pages, as shown below. The active page is selected from the list to the left.





Start by adding a new Qt Designer form to your project. This time, call it `configuration.ui`, and use the dialog with buttons at the bottom template as a starting point.

When the form is shown, first notice that the buttons are contained by a `QDialogButtonBox` widget. You can add or remove buttons from it using the property editor.

Now, drag a `QListWidget` and a `QStackWidget` () onto the form. The stack widget can be hard to see as it does not have an outer frame when not selected, but you can always spot the two arrows in its upper right corner.

A stack of widgets is a set of pages from which only one is shown. Right clicking on the stack lets you insert, remove, sort and navigate between the pages. You can also use the arrows to navigate between the pages. The arrows are not visible in the final design.

Now, drop three line edit widgets and three labels onto the first page of the widget stack. Then switch to the second page and drop two labels, two radio buttons and one date editing widget () onto it. When all widgets are in roughly the right place (compare to the illustration above), apply form layouts to both pages.

Now, switch your attention to the list widget. Double click on it and add two items – “Name” and “Details” to the list. Again the result should look like the illustration above.

Now place the stack of widgets and the list widget in a horizontal layout, before applying a vertical layout to the entire form.

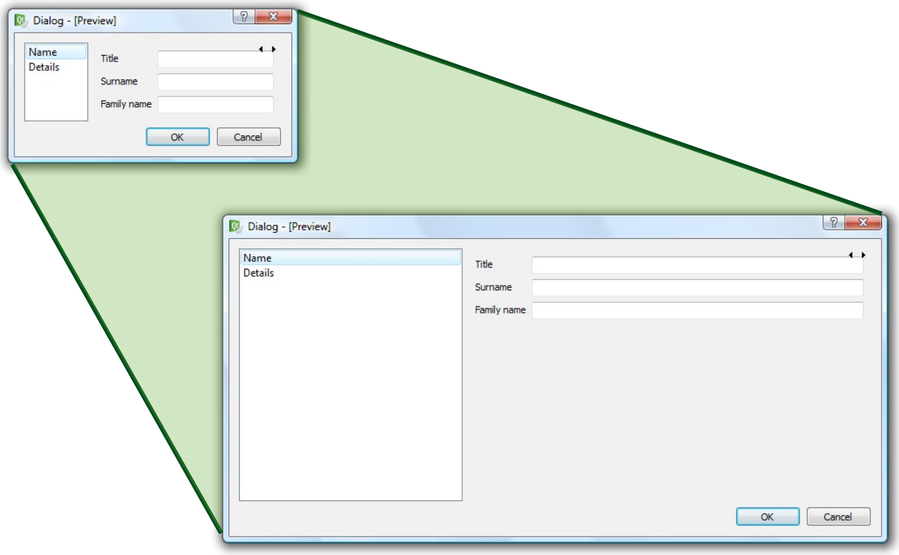
The final step before we can tweak the dialog is to connect the list widget and the widget stack. You do this by switching the operating mode of designer from widget editing to signal/slot editing from the buttons in the toolbar.

In signal/slot edit mode, drag from the list widget (signal source) to the widget stack (destination slot). Make sure not to drag onto one of the child widgets of the stack. In the dialog that pops up, pick the `currentRowChanged(int)` signal and the `setCurrentIndex(int)` slot. This makes the list widget control which pane is shown.

Test the dialog by previewing it. Notice that the arrows in the top-right corner will not be present if the design is used in an actual application. Instead, try using the list widget to switch between the pages.

Also try stretching the dialog from small to large. Notice how the list widget feels too wide at times. It would be better to let the widgets in the widget stack utilize the screen estate.

By setting the horizontal size policy of the widget stack to minimum expanding and the corresponding policy of the list widget to maximum, the behavior will feel more natural. The resulting stretch is shown below.



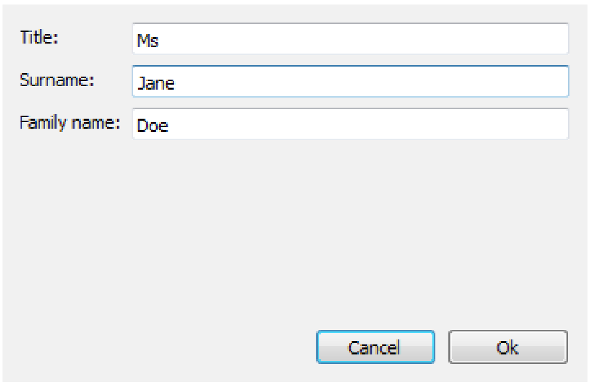
Style Sheets

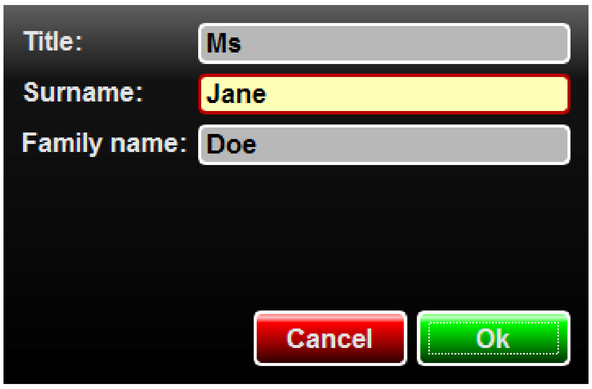
In this final step, you will try tweaking the look of a user interface using style sheets.

Start by adding a Qt Designer form to the project. Use the dialog without buttons template and name it `styleddialog.ui`.

Start by adding two `QPushButton` widgets, three `QLineEdit` widgets and three `QLabel` widgets. Put the buttons together with a spacer spring in a horizontal layout. Put the labels and line edits in a form layout and finally apply a vertical layout to the entire form.

Alter the texts so that the dialog looks like the example to the left below. The goal will be to alter the look to resemble the right dialog.

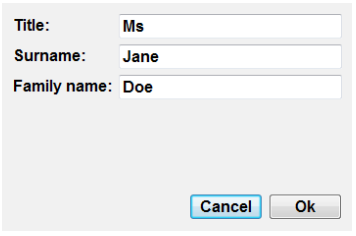




The first step will be to replace the font used through-out the widgets. Start by selecting the form itself and then right click an choose *Change styleSheet...* from the popup menu. Enter the following code in the style sheet editing dialog and click apply.

```
QWidget {
    font: bold 12pt "Arial";
}
```

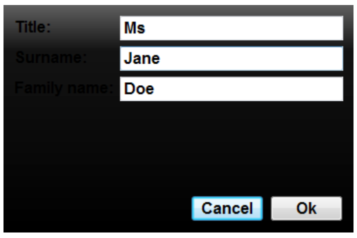
The code *selects* all widgets that are `QWidget` decendants (including action `QWidget` instances), and alters the font property.



The change is shown at once in the form editing window, but you can also preview the dialog to inspect the result. Notice that the font property is inherited from QWidget to all other widgets, so this style sheet applies to all buttons.

Continue by altering the background of the dialog itself. Add the following rule to the style sheet.

```
QWidget[objectName="StyledDialog"] {
    background-color: qlineargradient(x1:0, y1:0, x2:0, y2:1,
        stop:0 rgba(90, 90, 90, 255),
        stop:0.2 rgba(16, 16, 16, 255),
        stop:1 rgba(0,0,0,255));
}
```



The rule will apply a linear gradient to the dialog. You can read more on `qlineargradient` rules here: http://doc.trolltech.com/4.6/stylesheet-reference.html#gradient .

The selector used, “`QWidget[objectName="StyledDialog"]`”, selects all `QWidget` decendants with the property “`objectName`” set to “`StyledDialog`”. If the rule does not change the background of your form, then check the name in the property editor.

The result from this is a dialog where it is more than hard to read the text of the labels. To fix this, add the following line to the original `QWidget` style rule.

```
color: rgb(226, 226, 226);
```

Now it is impossible to read the text on the buttons and line editors. Let's take care of that with the following rule. Make sure to add this rule to the bottom of the style sheet.

```
QLineEdit {
    color: black;
}
```

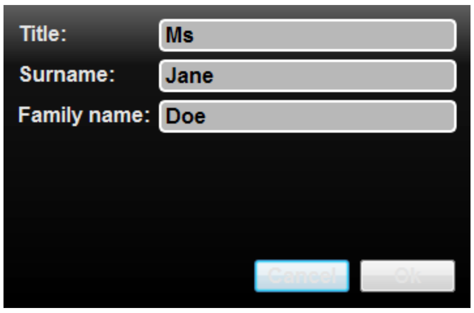
Changing the text color of a `QLineEdit` overrides the color set for `QWidget` if it appears last in the style sheet. The styles are applied as the style sheet is interpreted, so order is important.

While we are modifying the line editor widgets, lets add the following lines to the same rule.

```
border: 2px solid white;
border-radius: 5px;
background-color: rgb(180, 180, 180);
```

This alters the border of the widget to a two pixels wide, solid, white line. The corners of the border are rounded with a radius of five pixels. Finally, the background color of the widgets in question is altered as well.

Previewing the dialog will yield something like the illustration below.


Title: Ms
Surname: Jane
Family name: Doe

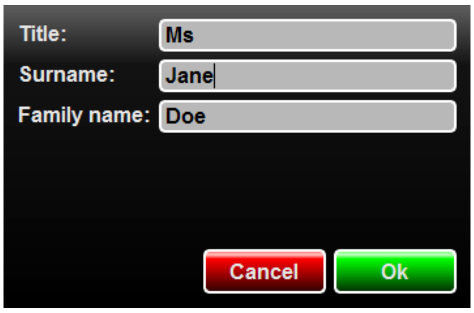
The text of the buttons can still not be read – and the shape of the buttons look out of place. As both buttons will have the same shape, start by adding the following rule to the bottom of the stylesheet.

```
QPushButton {
    border: 2px solid white;
    border-radius: 5px;
    padding: 5px;
    min-width: 80px;
    min-height: 20px;
}
```

Now, all that is left is to apply a green gradient to the Ok button and a red gradient to the Cancel button. The following rules take care of that.

```
QPushButton[text="Ok"] {
    background-color: qlineargradient(x1:0, y1:0, x2:0, y2:1,
        stop:0 rgba(64, 255, 64, 255),
        stop:0.1 rgba(200, 255, 200, 255),
        stop:0.2 rgba(0, 255, 0, 255),
        stop:0.90 rgba(0, 64, 0, 255),
        stop:1 rgba(0, 32, 0, 255));
}
QPushButton[text="Cancel"] {
    background-color: qlineargradient(x1:0, y1:0, x2:0, y2:1,
        stop:0 rgba(255, 64, 64, 255),
        stop:0.1 rgba(255, 200, 200, 255),
        stop:0.2 rgba(255, 0, 0, 255),
        stop:0.90 rgba(64, 0, 0, 255),
        stop:1 rgba(32, 0, 0, 255));
}
```

Previewing the dialog again, the only difference to the goal dialog is that one of the line edits are highlighted.

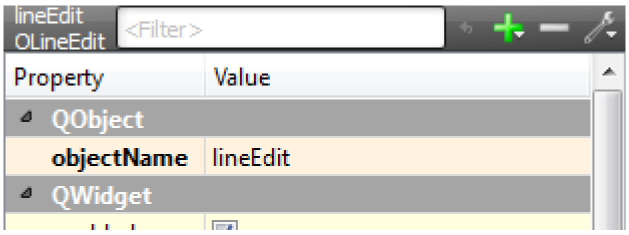


Achieving highlighting is a two stage process. First add the highlight rule to the bottom of the style sheet.

```
QLineEdit[highlight="true"] {
    background-color: rgb(255, 255, 180);
    border-color: rgb(180, 0, 0);
}
```

The rule will highlight all `QLineEdit` widgets with the property `highlight` set to true. The only problem is that the `QLineEdit` widget does not have a property called `highlight`, but we can add this dynamically.

Start by selecting the widget in question and click the small plus sign at the top of the property editor.



Property	Value
objectName	lineEdit

In the drop down menu, choose to add a boolean property and name it highlight. When added, simply check the property and the widget is highlighted.