Exercises Lecture 5 – Custom Widgets and Painting

Aim: This exercise will take you through the process of creating custom widgets. You will try both composing a widget from existing components and implementing a new widget completely from scratch.

Duration: 1-1.5h

Composing Widgets

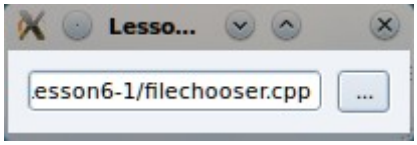
The exercise is coupled to a source code package. Extract the contents of that package before you start. The first exercise step is based on the *filechooser* project.

The project is a starting point. It shows a basic user interface for picking file names. If you resize the window you will see that its the size of the window and the size of the contents are not connected in any way. All widgets are placed and sized statically.

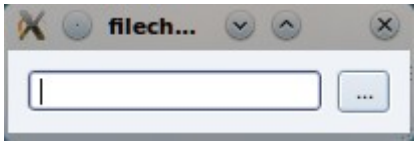
You will now correct the resizing issue by creating and setting up a horizontal layout for managing the position and size of the widgets in the window. You do that in the constructor of the `FileChooser` class.

First, create a `QHBoxLayout` object, then use the `addWidget` method to add the widgets to it in the right order (from the left to the right).

In the provided source code, the button has no action associated with it. To sort this out, start by finding the `chooseFile()` slot. It will be used to respond to the user clicking the button. In it, ask the user for a file path using the `QFileDialog` class. The path is then set to the text of the `lineEdit` object. Add the code to do so. The easiest approach is to use the `QFileDialog::getOpenFileName` method. If the user cancels the `QFileDialog`, make sure to clear the contents of the `lineEdit`.



To invoke the slot when the button is clicked, connect the button's `clicked` signal to the `chooseFile` slot in the constructor of the `FileChooser` class.



Right now, the `FileChooser` class can be used as is. However, by adding the methods shown below to the public section of the class declaration, you can make it reusable.

```
    QString file() const;
    void setFile(const QString &file);
```

Implement these two methods to return and set the value of the line edit, and you are done. What

you just did was to add a public API to the group of widgets that you have just composed into your very own `FileChooser` widget. Every time you want to provide file choosing capabilities from your user interface, you can now use the `FileChooser` class.

Creating a Simple Smiley

The remaining part of this exercise will be about the creation of a completely custom widget – the `Smiley`. The starting point is the *smiley1* project.

Right now, the project creates and shows a smiley, but the result is only an empty window. Your task is to draw a smiley inside the window by implementing the `paintEvent()` method and drawing a face. To do this, you will implement the two support methods `paintEye()` and `paintSmile()`. Throughout the code in the *smiley1* project, you will find connects that suggest appropriate size values to be used.

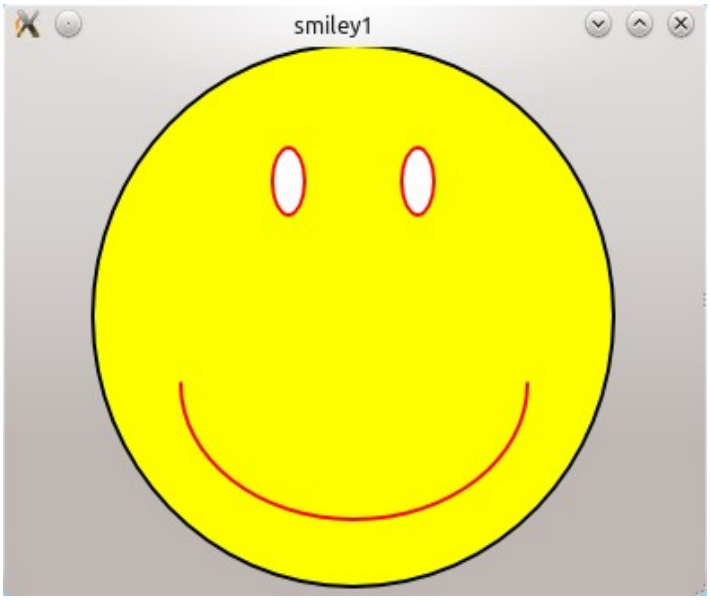
First, start by looking at the `paintEvent` method. There, setup a pen and brush for the background of the smiley and use the `QPainter::drawEllipse` method to draw a circle around the origin `center` with the radius `size`. The `size` and `center` variables have been calculated for you.

```
QPen pen;
pen.setWidth(2);
pen.setColor(Qt::black);
painter.setPen(pen);
painter.setBrush(Qt::yellow);
painter.drawEllipse( …
```

When the face has been painted, continue by implementing the `paintEye` method. In it, again setup an appropriate pen and brush (yellow eyes do not look very healthy) and draw ellipses around the given center (`pt`) with the given `size`.

Finally, implement the `paintSmile` method. Setup a pen, then draw the smile using the `QPainter::drawArc(QRect, int start, int length)` method. Paint it from 0 degrees and make it 180 degrees wide. The length is specified in 16ths of a degree and in the wrong direction for our needs, to make sure to set the length to `-180*16`.

Running the application now should give you a window filled with a happy face!



Handling Mouse Events

The next stage of the smiling widget project is to add eye movement tracking the mouse pointer while a mouse button is being pressed. The project is a continuation of the previous step, but there is support code that is added to it. The best way forward is to start from the *smiley2* project.

The first thing that you need to do is to track the mouse position while a mouse button is being pressed. To do that, we use the `QPoint` variable called `focusPoint`. It holds the last known position of the mouse. For each of the event handles; mousePressEvent, mouseMoveEvent and mouseReleaseEvent, set the focus point to the current mouse location. You can get the mouse location from the `QMouseEvent::pos()` given.

Each time the position is changed, you need to tell Qt to redraw the widget. You do this by calling the `QWidget::update()` method. This should give you a smiley with mouse tracking eye movements.

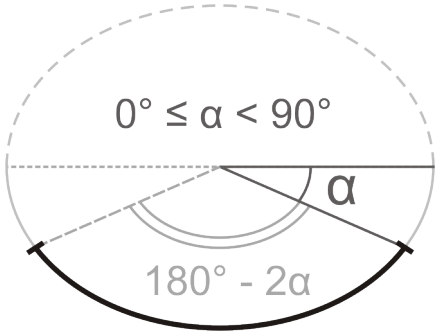
Adding signals and slots

The final step in creating your own widget is about signals and slots. Usually, you connect the right signal to the right slot, but when creating custom widgets you must also provide the right signals and slots.

Again, this is a continuation from the previous step, but with added support code. This means that you should start from the *smiley3* project provided in the source code package.

The goal of this step is to modify the `Smiley` class so that the extent of the smile can be adjusted using signals and slots. This means that when incorporating the smiley widget in a user interface, the smile can be adjusted in a reusable way.

The first point of change is the `paintSmile()` method. It needs to be modified to accept an angle which will determine the starting angle and arc span for painting. The size of the smile is stored in the `m_smileSize` variable.



Implement the `paintSmile()` method so that it follows the figure above, where the argument `angle` corresponds to alpha. You can test your implementation by changing the value of `m_smileSize` in the class constructor. Valid values for the value are 0-100.

The next step is to expose an appropriate API for modifying the size of the smile. The size of the smile is kept in the `m_smileSize` variable and is measured as percentage of the full smile (so its values in range 0-100).

To indicate that the class has custom signals and slots, start by adding the `Q_OBJECT` macro at the top of the class implementation. Having done this, you have to re-run the qmake command on the project. In Qt Creator, you do this by right clicking on the project and selecting the *Run qmake in...* option

The first API function to implement is the public `setSmileSize()` slot. Make sure that only valid values are accepted. When implemented, you can test your slot by dragging the slider under the smiley in the window. As the slot alters the `m_smileSize` variable, which in turn affects the appearance of the widget, make sure to call `update` in the slot as well.

When providing a slot for altering a member variable, it is common practice to extend the class with a corresponding reading function (already present in the *smiley3* project) and a signal emitted when the value is changed.

First add code to emit the signal `smileSizeChanged(int)` whenever the smile size changes. Start by declaring the signal in the class header, then modify the setSmileSize slot to emit the signal. Make sure that the signal only is emitted when the value is changed, not when it is set to the same value as it already was.

To take the smile size from a set of methods and a private variable to a Qt property, you need to declare it as such.You can do that by adding the following line right after the `Q_OBJECT` macro in the `Smiley` class declaration.

```
Q_PROPERTY(int smileSize READ smileSize WRITE setSmileSize)
```

This informs Qt there is a `smileSize` property of type `int` that can be accessed using `smileSize` and `setSmileSize` methods. This makes it possible to integrate the widget with Qt Designer, as well as accessing the property using the Qt meta-object interfaces.

Solution Tips

Step 1

If you pass `this` to the constructor of the layout object, the layout will handle the layout of the current widget.

```
    QHBoxLayout *layout = new QHBoxLayout(this);
    layout->addWidget(lineEdit);
    layout->addWidget(button);
```

Use `QFileDialog::getOpenFileName()` to query the user for a file name.

```
    QString path = QFileDialog::getOpenFileName(this);
```

Use the `setText` method to update the line edit.

```
    lineEdit->setText(path);
```

Use `QObject::connect` method to set up a connection between the button's `clicked()` signal and `chooseFile()` slot of the main widget.

```
    connect(button, SIGNAL(clicked()), this, SLOT(chooseFile()));
```

The `file` and `setFile` methods are used to access the text property of the `lineEdit`. This makes them very simple to implement.

```
QString FileChooser::file() const
{
    return lineEdit->text();
}

void FileChooser::setFile(const QString &file)
{
    lineEdit->setText(file);
}
```

Step 2

Draw the face with `QPainter::drawEllipse()` passing it the center of the circle and its radius.

```
    painter.drawEllipse(center, size/2, size/2);
```

Use similar code for the eyes. Make the eye width half the size of its height.

```
    painter->drawEllipse(pt, size/4, size/2);
```

The smile can be drawn as an arc by calling `QPainter::drawArc()`. Make it draw from the right, downwards and to the left with a total of 180 degrees span. The angles are passed in 1/16th of degree and counted clock-wise (pass negative values for counter-clockwise direction).

```
    painter->drawArc(r, 0, -180*16);
```

Step 3

To store the current position of the cursor, assign the point retrieved from the mouse event to a

member variable.

```
    focusPoint = me->pos();
```

If you want an indifferent look when the mouse button is released, you can assign an empty (invalid) point to the `focusPoint`.

```
    void Smiley::mouseReleaseEvent(QMouseEvent *) {
        focusPoint = QPoint();
        update();
    }
```

Step 4

To make sure the value passed to the slot is within appropriate range you can use qBound() macro passing it the minimum, current and maximum values. It will return the current value bounded to the minimum-maximum range.

```
    size = qBound(0, size, 100);
```

Check that you only emit a signal when the smile size changes. You can simply return from the method upon detecting that no change occurs.

```
    if(size == m_smileSize)
        return;
    m_smileSize = size;
    emit smileSizeChanged(size);
```