QML & git

Applied Computer Systems Laboratory 2

Laboratory Guide

#6

Author: László Blázovics PhD

Version: 1.2

2020.

Budapest University of Technology and Economics
Department of Automation and Applied Informatics

Exercise 6

Git practice - QML functions, signals and slots

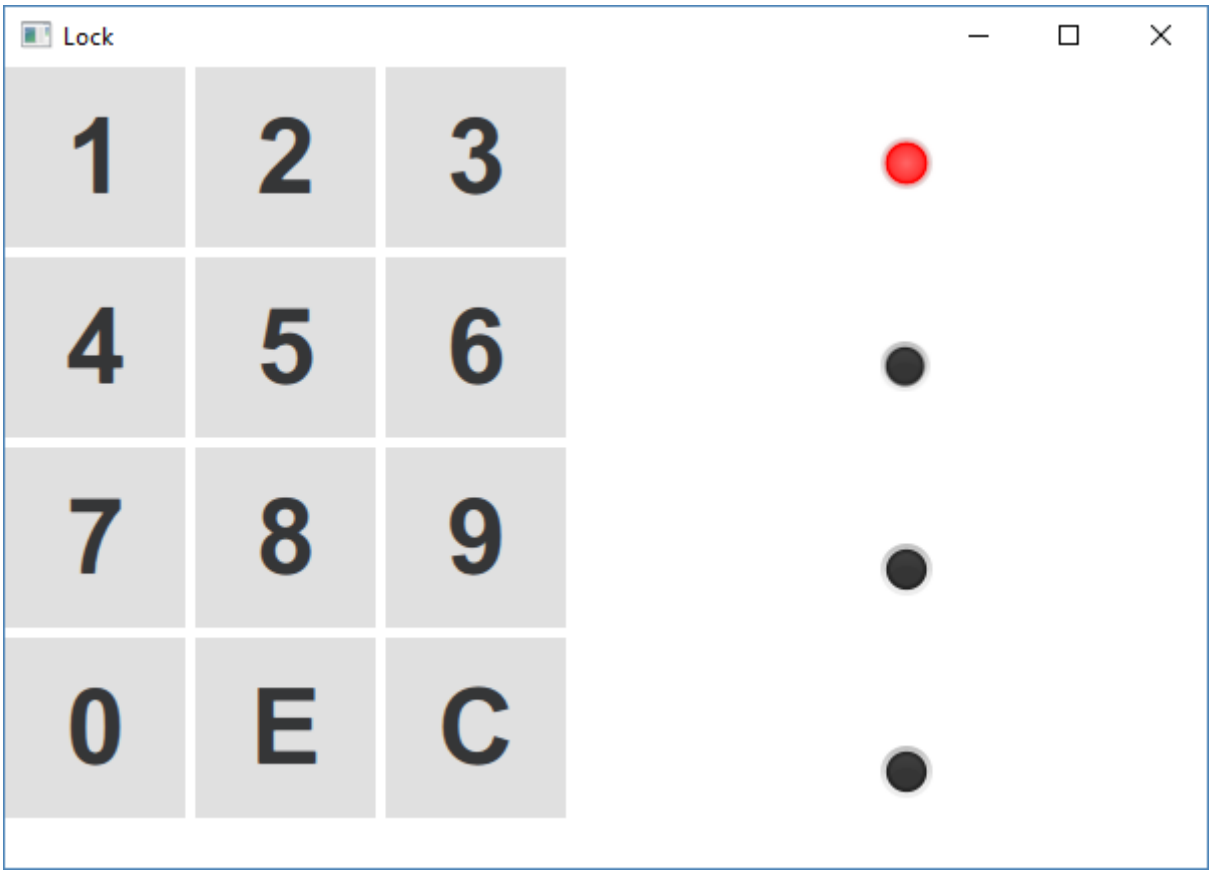
This lesson is a brief introduction of the QML side programming. It will discuss the usage of *properties*, *signals* and their *slots* and *functions* too.

During this exercise, a "security lock panel" will be built up from QML components. If the user gives the correct four-digit passcode, the "lock" will open (the status light will change from red to green).

Git preparation

First let us a create a repository for version control our project

1. Create an empty Github repository.
2. Clone the repository to your machine by using a git client appllication. Use the `git clone` command with the proper argument. (i.e. the link of your repository which can be copied to the clipboard on the github webpage)
3. Use the downloaded repository folder as the root folder of your project.



4. Download, unzip and open the RAL6 QtQuick project.
5. As it can be seen, there are four qml files in the root of the qml.qrc. Next to the default main.qml, there are following three files:

- LockDashboard.qml, which now contains four status lights in a ColumnLayout.
- Numberpad.qml, which is a 3x4 grid of NumberpadButtons.
- NumberpadButton.qml which is an extension of the QML Button.

6. At first, try to build and compile the application. If it succeeded, it could be seen that all buttons have the same "X" text and they cannot take effect on the indicator lights. That is what will be altered in the following section.

7. Go back to the `Git Bash` and check the status of the repository by executing the `git status` command. You might see that there are generated resource files (.o, .exe files) among the untracked files. And if you are using the *Shadow Build* build option (which is set by default), these generated files can be found under the dedicated *shadow build folder(s)*

   As we learned, these files should be omitted from the git repository, so add either the whole folders (similar to this: *build-RAL6-Desktop_Qt_5_12_0_MinGW_64_bit-Debug*) or the specific extension types to the **.gitignore** file. See this the link for further details.

   For editing use a simple text editor like *Notepad++* or similar.

8. Now check the status again. If you made it right, you should see only the source files. If not, check it again.

9. Now make your first `commit` then `push` it to your repository too.

10. Because we would like to change the text of our custom button, first open the **NumberpadButton.qml** and add a *string property* called *number* to the base Item.

```
property string number
```

11. Now, we have a property which can be reached from outside. (Mark that the properties of the nested Button inside the Item are not reachable from outside, just from inside this file.)

12. Change the default value of the *text* property of the *Button* to this newly created *number*.

```
anchors.fill: parent
text: number; // instead of "X"
highlighted: false
```

13. If we have our property in the *NumberpadButton*, go to the Numberpad.qml and add set the *number* of all the 10 number button of Numberpad:

```
...
NumberpadButton {
    id: btn1
    number: qsTr("1") //do it for all the numbers 0-9
...
NumberpadButton {
```

```
    id: btn0
    number: qsTr("0") //do it for all the numbers 0-9
...
```

14. For the last two buttons - because they are special ones for control - set the followings:

```
...
NumberpadButton {
    id: btnEnter
    number: qsTr("E")
}
NumberpadButton {
    id: btnClear
    number: qsTr("C")
```

15. It is worth to compile and run the application to check our work.

16. Now make another `commit` and then `push` it to your repository.

17. Now, we have a text, let us create a signal which can be reached from outside too. Besides the fact that someone has clicked on the button, we would like to send the "number" of the button also. Therefore add a *signal* called *clickSignal* with one *string* type parameter named *number* to the base Item.

```
signal clickSignal(string number)
```

18. Let us connect the "click event handler" of the Button (which is the slot for the signal **clicked**) with our newly created one. This will be done in the "slot" *onClicked:* inside the Button. Because our signal has one parameter, we will pass our *number* property to it.

```
onClicked: parent.clickSignal(number)
```

19. Now, that we have a custom button which emits a signal with its "identifier", let us extend the *Numberpad* with the following three signals:

```
Item{
    ...
    signal buttonPressed(int id)
    signal enterPressed()
    signal clearPressed()
```

The first one is used by the numbers the other two - without parameter - are used for signalling when the *E = Enter* and the *C = Clear* buttons are pressed.

20. Our last task with the *Numberpad* is to connect the signal of each custom button with the proper signals of our *Numberpad*

```
    ...
    number: qsTr("1")
    onClickSignal: buttonPressed(parseInt(number)) //do it for all the
numbers 0-9
    ...
    number: qsTr("0")
    onClickSignal: buttonPressed(parseInt(number)) //do it for all the
numbers 0-9
    ...
    NumberpadButton {
        id: btnEnter
        number: qsTr("E")
        onClickSignal: enterPressed() //special signal #1
    }
    NumberpadButton {
        id: btnClear
        number: qsTr("C")
        onClickSignal: clearPressed() //special signal #2
    }
```

21. Now `commit` and `push` your work to your repository again.

22. Switch to the *main.qml* and handle the signals of our *Numberpad* component. At this stage, we only print out the some debug message. However, we will extend these functions soon.

```
Numberpad {
    id: numberpad

    onButtonPressed: {
        console.log(id)
    }
    onClearPressed: {
        console.log("clear")
    }
    onEnterPressed: {
        console.log("enter")
    }
```

23. Compile and run the application to test whether the signals are properly handled.

24. Now, that our *Numberpad* is working, let us implement our *LockDashboard*. Therefore switch to the *LockDashboard.qml* file.

    The inner logic of the lock is the following:

    - The default state is locked which is indicated by the **StatusIndicator** called *lockedIndicator*.

- When a user presses the *C* button, the lock begins to wait for the four-digit passcode. This will be indicated by the yellow *unlockkingIndicator*.
- If the user successfully enters the correct passcode, the lock will turn to green. (indicated by the green *unlockedIndicator*. However, when the user failed to enter the passcode, the lock will return to the locked state.
- When the lock is unlocked, and the user presses the *E* button, the lock will enter into a programming state indicated by the blue *programmingIndicator*. In this state, when the user enters four digits on the *Numberpad* the passcode will be overwritten, and the lock will return to the locked state.

25. Let's `commit` and `push` again.

26. Start the implementation of the "lock logic" with the addition of a *QtObject* into the Item. This object will contain all properties and functions which should not be reached from outside of the module. Unfortunately, there is no other way to crate *private* properties of functions.

```
Item {
    ...
    QtObject {
        id: priv // let the id of this object priv
    ```
```

27. First create an array, which will store the passcode which is 1,2,3,4:

```
QtObject {
    property variant passcode: [1,2,3,4]
```

28. Now add a property which will store the index of the next number of the passcode. Let the initial value of it -1, which will indicate that no number was given.

```
    property variant passcode: [1,2,3,4]
    property int inputIndex: -1 //no number was entered
```

29. Then add two boolean property which will store the unlocked and programming states.

```
    property int inputIndex: -1 //no number was entered
    property bool unlocked: false
    property bool programming: false
```

30. Finally add a function called *unlock*, which will set the unlocked state.
```

```
function unlock(){
    unlocked = true;
}
```

31. Outside of the *QtObject* create a function called startUnlocking, which will be called when the user presses the *C* button.

    ```
    function startUnlocking(){

    }
    ```

    Inside this function set the *inputIndex* to **0** and the *unlocked* to **false**. Keep in mind that these are the properties of the embedded *QtObject* with the id *priv*!

32. And here comes another `commit` and `push`.

33. Now implement the *lock* function which will reset the lock to its default state:

    ```
    function lock(){

    }
    ```

    Inside it reset the *inputIndex* to its default state and the *unlocked* property to **false**.

34. Now, go to the bottom of the file and set the *active* property of each *StatusIndicator*

    - The *lockedIndicator* should be active only when the *unlocked* property is **false**:

      ```
      active: !priv.unlocked
      ```

    - The *unlockedIndicator* should be active only when the *unlocked* property is **true**
    - The *programmingIndicator* should be active only when the *programming* property is **true**
    - The *unlockingIndicator* should be active only when the *unlocked* property is **false** and the inputIndex **greater or equal to 0**

      ```
      active: (!priv.unlocked && priv.inputIndex >= 0)
      ```

35. Now, let us implement the function which will handle the number inputs. Therefore this function will have one input parameter.

    ```

    ```

```
function numberInput(number)
{

}
```

36. Inside the function first, check if the *inputIndex* is *less than 0*. If that is true, the function should *return*, because the lock is not in unlocking state.

37. If the lock is in unlocking state then compare the given *number* with the proper value of the passcode.

```
if( number !== priv.passcode[priv.inputIndex])
{

}
```

    If they are not equal call the *lock()* function and *return*, because the user does not know the correct passcode.

38. If the two values are equal, check whether the inputIndex has increased to 3, which means that the fourth digit has been successfully entered. Therefore the lock should enter the unlocked state. Otherwise, increment the *inputIndex*!

```
if(priv.inputIndex == 3)
{
    priv.unlock();
}
else{
    priv.inputIndex++
}
```

39. Now go back to the *main.qml* file and the proper function calls to the event handlers:

```
onButtonPressed: {
    dashboard.numberInput(id);
}
onClearPressed: {
    dashboard.startUnlocking();
}
```

40. Build and run the application and try to unlock the lock!

41. `commit` and `push`!

42. Now go back to the *LockDashboard.qml* and extend the "lock logic" with the programming phase. First, create a function called *startProgramming()*. Inside the function set the *programming* property to *true* and the *inputIndex* to **0**, but only in that case when the *unlocked* property is *true* (i.e. the lock is unlocked).

43. Go back inside the *QtObject* because we will implement a "private-like" event handler for the *programming* property. Create an event handler (technically a slot) for the signal *unlockedChanged*.

```
onUnlockedChanged: {
    programming = false;
}
```

This will set the *programming* property false, whenever the state of the *unlocked* property has changed.

44. Also in the *QtObject* add the following array property to it, because it will store the newly given passcode which should overwrite the old one:

```
property variant newPasscode: [1,2,3,4]
```

45. Now, go to the *numberInput* function and extend it with the following lines:

This will store the numbers of the new passcode.

```
if(priv.programming)
{
    priv.newPasscode[priv.inputIndex] = number;
}
else{
    if( number !== priv.passcode[priv.inputIndex])
    {
        ...
    }
}
```

This will copy the newly given passcode to the old one's place.

```
if(priv.inputIndex == 3)
{
    if(priv.programming)
    {
        for(var i=0; i<4; i++)
        {
            priv.passcode[i] = priv.newPasscode[i];
        }
        lock();
    }
```

```
        else
        {
            priv.unlock();
        }
    }
    else {
        priv.inputIndex++
    }
```

46. Now, go back again to the *main.qml* and inside the *onEnterPressed:* event handler call the *dasboard*'s *startProgramming()* function.

47. Build and run the application and test the passcode "programming"!

48. And finally `commit` and `push`, as usual.