Exercises Lecture 2 – The Qt Object Model and Signal Slot Mechanism

Aim: This exercise will help you explore the Qt object model (inheritance, properties, memory management) and the signal slot mechanism.

Duration: 1h

NOKIA

1 Exploring Inheritance

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

- A C++ class, `ValueObject`, inheriting from `QObject`
- A C++ source file, `main.cpp`

In `main.cpp`, create a main function and include the `ValueObject` header file. In your `main` function, implement the following function body.

```
{
    ValueObject o;

    // Insert code here

    return 0;
}
```

Using the function `QObject::inherits(const char *className)`, write `qDebug` statements testing if `o` inherits `QObject` and `ValueObject`.

> **Note!** If using Qt Creator on Windows, you might have to run your code in the debugger to see the `qDebug` printouts!

Try commenting out the `Q_OBJECT` macro from the `valueobject.h` header file. Discuss and explain the results. Remember to re-enable the `Q_OBJECT` macro after having tested this.

Now, try testing a `QFile` object. Does it inhert `QIODevice`, `QDataStream`, `QObject` or `QTemporaryFile`? Can you find this information in the documentation as well as through testing?

2 Properties

For this step, we keep the class that we created in the previous step.

Start by adding two member functions to the `ValueObject` class:

- `setValue`, setting an integer value of the object
- `value`, returning the set integer value of the object

These methods set and get an `int` value. You will have to add a private `int` member variable to keep that value. Do not forget to initialize the private value in the constructor.

NOKIA

Now, modify your `main` function to look like this.

```
int main(int argc, char **argv)
{
    ValueObject o;

    qDebug("Value: %d = %d", o.value(), o.property("value").toInt());
    o.setValue(42);
    qDebug("Value: %d = %d", o.value(), o.property("value").toInt());
    o.setProperty("value", 11);
    qDebug("Value: %d = %d", o.value(), o.property("value").toInt());

    return 0;
}
```

You will have to include the `QVariant` header file as well for the `property` calls to work.

Run the code, discuss and explain the results.

Now, add a `Q_PROPERTY` macro defining the `value` property with a `READ` and a `WRITE` function. Add the macro just after the `Q_OBJECT` macro in the class declaration.

Re-run the code, discuss and explain the different result.

3 Memory Management

Continuing from the previous step, we now add a `qDebug` statement in the constructor along with a destructor with a `qDebug` statement. Make sure that the destructor is virtual. The `qDebug` statements should inform you when an object is constructed and when it is destructed. In the destructor, add a reference to the `QObject::objectName` property as well.

Replace the property setting and debugging from the last step with the following line.

```
    o.setObjectName("root");
```

Running the code should now yield an output looking something like this.

```
ValueObject constructed.
ValueObject root destructed.
```

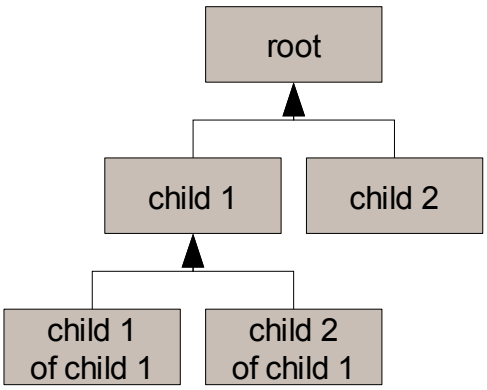
Now, add the following code just after the setting of the object name of of `o`.

```
    ValueObject *c1 = new ValueObject();
    c1->setObjectName("child 1");
    ValueObject *c2 = new ValueObject();
    c2->setObjectName("child 2");
    ValueObject *c1c1 = new ValueObject();
    c1c1->setObjectName("child 1 of child 1");
    ValueObject *c2c1 = new ValueObject();
    c2c1->setObjectName("child 2 of child 1");
    ValueObject *c = new ValueObject();
    c->setObjectName("child");
```

Run the code and evaluate the result. The code is leaking memory.

NOKIA

Now, provide a parent pointer to the constructors of `c1`, `c2`, `c1c1` and `c2c1` so that the ownership tree looks like the illustration below.



Finally, assign `c2` as the parent of `c` using the `QObject::setParent` method.

Run the code and verify that all objects are properly destructed. Discuss and explain the order of destruction, as well as why having the root object on the stack instead of the heap simplifies the situation.

4 Signals and Slots

Again, building on the results from the last step, we will now add signals and slots to our `ValueObject` class.

Set functions are natural slots, and are also natural points for adding signals tracking the value of the property being set. Update the `ValueObject` class declaration with these changes.

- Move `setValue` to the `public slots` section
- Add a signal called `valueChanged` carrying an integer argument

Alter `setValue` so that it prints the new value and emits the `valueChanged` signal upon the value changing.

Now, replace the contents of your main function with the following code.

```
int main(int argc, char **argv)
{
    ValueObject o1;
    ValueObject o2;

    o1.setValue(1);
    o2.setValue(2);

    // Connect here

    qDebug("o1: %d, o2: %d", o1.value(), o2.value());

    o1.setValue(42);
    qDebug("o1: %d, o2: %d", o1.value(), o2.value());

    o2.setValue(11);
    qDebug("o1: %d, o2: %d", o1.value(), o2.value());

    return 0;
}
```

NOKIA

Running the code shows that `o1` and `o2` are completely independent.

Now, establish a connection from `o1` to `o2`, making the value of `o2` follow `o1`.

Run the code, discuss and explain the behavior.

Now establish another connection from `o2` to `o1`, making the two `ValueObject` instances follow each other.

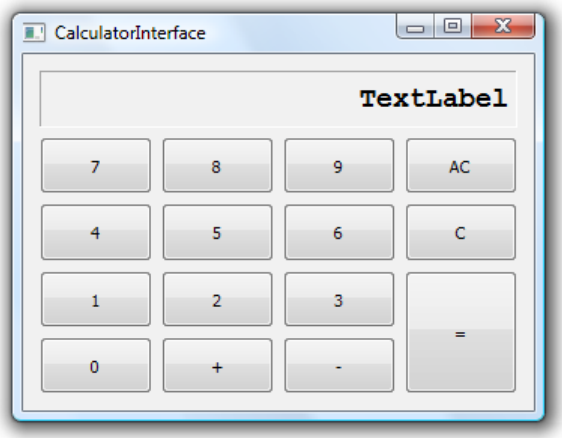
Run the code, discuss and explain the behavior.

> If the program hangs here, setting the same value over and over again, you have forgotten to add a check in your `setValue`. If the value isn't changed, the `setValue` slot should simply return.

Try to come up with situations in a user interface where the two connection relationships are useful, i.e. one object following another, versus two objects following each other.

5 The Signal Mapper

This step uses the *calculator* project as the base. Please open the project in Qt Creator and familiarize yourself with the components of the project.



The project is divided into three parts.

- `main.cpp`, containing a boilerplate `main` function
- The class `CalculatorInterface`, providing a user interface for a basic calculator
- The class `Calculator`, providing a trivial calculator engine implementation

In the file `calculatorinterface.cpp`, you will find the constructor shown below.

```
CalculatorInterface::CalculatorInterface(QWidget *parent) :
    QWidget(parent),
    ui(new Ui::CalculatorInterface),
    m_calculator(new Calculator(this))
{
    ui->setupUi(this);

    // Add code here

    m_calculator->allClear();
}
```

The user interface, made available through the `ui` variable, holds the following set of widgets:

- `buttonZero`
- `buttonOne`
- `buttonTwo`
- `buttonThree`
- `buttonFour`
- `buttonFive`
- `buttonSix`
- `buttonSeven`
- `buttonEight`
- `buttonNine`
- `buttonClear`
- `buttonAllClear`
- `buttonAdd`
- `buttonSubtract`
- `buttonCalculate`
- `entryLabel`

The `Calculator` class, object `m_calculator`, holds the following slots:

- `numEntered(int)` – the number provided has been entered (0-9)
- `clear()` - clear the current entry
- `clearAll()` - clear the calculator state completely
- `additionMode()` - switch to addition mode, calculates before switching mode
- `subtractionMode()` - switch to subtraction mode, calculates before switching mode
- `calculate()` - calculate the next result, clears the current entry

The class also provides a signal, `displayChanged(QString)`, used to update any display showing the status of the calculator.

In the `CalculatorInterface` constructor, make all the connections you can see fit. Run the code and discuss the result.

6 Extending the Calculator

The Calculator class is far from perfect. The interested student can extend the calculator in various ways:

- Add more operations (e.g. multiplication and division)
- Add support for entering negative numbers
- Add memory operation (M+, M-, MC, MR)
- Add support for larger numbers (the current limit is the size of `int`)

NOKIA

Solution Tips

Step 1 - Inheritance

Test if the object `o` inherits from `QObject` using the following line.

```
    qDebug("ValueObject inherits QObject: %s",
           o.inherits("QObject")?"yes":"no");
```

Step 2 – Properties

The class declaration looks like this when the member functions and private variable have been added.

```
class ValueObject : public QObject
{
Q_OBJECT
public:
    explicit ValueObject(QObject *parent = 0);

    void setValue(int value);
    int value() const;

private:
    int m_value;
};
```

The property declaration to be added at the end of the exercise reads like this.

```
Q_PROPERTY(int value READ value WRITE setValue)
```

Step 3 – Memory Management

The constructors need to be called with the following parent pointers.

```
    ValueObject *c1 = new ValueObject(&o);
    ValueObject *c2 = new ValueObject(&o);
    ValueObject *c1c1 = new ValueObject(c1);
    ValueObject *c2c1 = new ValueObject(c1);
```

The parent of c is assigned using the following line.

```
    c->setParent(c2);
```

NOKIA

Step 4 – Signals and Slots

The class declaration looks like this when the slot and signal have been added.

```
class ValueObject : public QObject
{
Q_OBJECT

Q_PROPERTY(int value READ value WRITE setValue)

public:
    explicit ValueObject(QObject *parent = 0);
    virtual ~ValueObject();

    int value() const;

public slots:
    void setValue(int value);

signals:
    void valueChanged(int);

private:
    int m_value;
};
```

The connections for interconnecting `o1` and `o2` look like this.

```
    QObject::connect(&o1, SIGNAL(valueChanged(int)),
                     &o2, SLOT(setValue(int)));
    QObject::connect(&o2, SIGNAL(valueChanged(int)),
                     &o1, SLOT(setValue(int)));
```

The implemented slot looks like this.

```
void ValueObject::setValue(int value)
{
    if(m_value == value)
        return;

    m_value = value;
    qDebug("Value set to %d", m_value);
    emit valueChanged(m_value);
}
```

NOKIA