Exercises Lecture 4 – Datatypes, Collections and Files

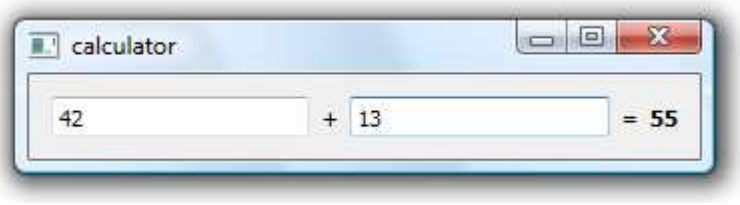
Aim: This exercise will take you through the process of loading and saving files, including custom data types and Qt collections.

Duration: 1h

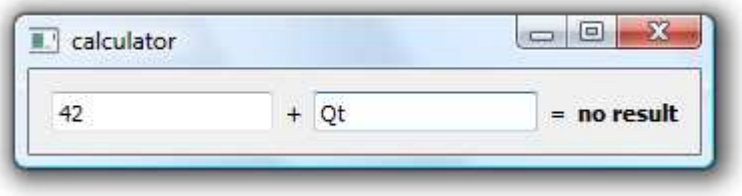
Strings and Numbers

Start from the *calculator* project. The project is implement to the extent that the user interface shows up and the `CalculatorWidget::valueChanged` slot is called when either of the value entry fields are altered.

Complete the slot so that the sum of the values entered in `m_firstValue` and `m_secondValue` is shown in the `m_result` label. The calculator only needs to handle integer values.



If either of the value field does not contains a number, the result should say “no result”.



Reading and Writing Files

Start from the *fileaccess* project. The project provides a user interface that lets the user edit a list of items. Your task is to implement the slots `FileListWidget::saveClicked` and `FileListWidget::loadClicked`.

The slots shall save the list as a text file using `QTextStream`. Each item of the list shall be given a line in the text file. This lets you verify the results using a plain text editor.

Start by implementing the `saveClicked` slot. The code shall perform the following steps:

1. Create a `QFile` object for the file name `listitems.txt` and open that file object for writing.
2. When the file has been opened, create a `QTextStream` object for that file.
3. Write the items to the stream using the `<<` operator. Use the following code to iterate over the items.

```
    for(int i=0; i<m_list->count(); ++i)
        qDebug("Item: %s", qPrintable(m_list->item(i)->text());
```

4. When all items have been written, close the file object using `QFile::close`.

Before continuing, verify that the file does contain the items of the list, one item per line.

The next step is to implement the `loadClicked` slot. The code shall be implemented much like the save slot, but read the file. Use the `QTextStream::atEnd` function to determine whether all lines of the file have been read. Use the `QTextStream::readLine` function to read a complete line of the file.

Do not forget to close the file object when the entire file has been read.

Now verify that the program loads and saves as expected. Also, ensure that you can load the

same file repeatedly correctly, i.e. press the Load button several times in a row.

Binary Files and Custom Data Types

Start from the *customtype* project. You will use this project through exercise steps 3-5. The project contains a `main.cpp` file, which you will work in, and the class `Person`. Through out these exercise steps, you will use the `Person` class in different settings. For this to work, you need to take measures to give Qt the needed tools.

To understand the first task, simply build the project. The compiler will complain about missing the two `<<` and `>>` operators for streaming objects of the type Person to and from a `QDataStream`. In the documentation for the `QDataStream` class you can find out more about the required functions. The functions to implement are declaration as shown below.

```
QDataStream &operator<<(QDataStream &, const Person &);
QDataStream &operator>>(QDataStream &, Person &);
```

Implement the functions at the designated location in the source code (look for it in `main.cpp`). Build and run the application. Verify that the output reads as follows, i.e. that the original list can be saved and reloaded properly.

```
Original people
  John Doe
  Jane Doe
  Albert A Einstein
Loaded people
  John Doe
  Jane Doe
  Albert A Einstein
```

Hashing Custom Data Types

Continuing from the last step, uncomment the body of the `hashOfPeople` function in `main.cpp`, then build the project. The newly uncommented code tries to use the `Person` class as the key in a `QHash` collection. This requires a hashing function and a comparison operator. As expected, the compiler complains about a missing `qHash(const Person&)` function and an `==` operator for the `Person` class. Add such a function at the designated location in the source code (look for it in `main.cpp`). From the documentation for the `QHash` class, you will find the proper return type of such a function.

A hashing function is often based on knowledge about the situation. Given our data type, we know that the first name and initials are usually more unique than the family name. Thus, base your return value on the `qHash(const QString&)` function and these two properties.

The `==` operator, outlined below, needs to take all properties of the `Person` class into account. Complete the operator to return `true` only if the given `Person` objects are identical.

```
bool operator==(const Person &p1, const Person &p2)
```

Now build and run the application. If something is wrong, the output of the application will tell you (by printing “Something is wrong with...”).

QVariant and Custom Data Types

Continuing from the last step, uncomment the body of the `variantsOfPeople` function in `main.cpp`. This function will put `Person` object into `QVariant` objects and then try to retrieve them.

Compiling the project with the function body uncommented, the compiler will complain about a `QMetaTypeId` for the `Person` class. This is a clue, but does not lead you all the way to the solution. The documentation for the `QVariant` class reads as follows.

> "*`QVariant` can be extended to support other types than those mentioned in the `Type` enum. See the `QMetaType` documentation for details.*"

Read the documentation on the `QMetaType` class and add the single line type registration at the designated location in the source code (look for it in `main.cpp`).

Building and running the application, the end of the output should now read as follows. This means that the `Person` objects has been properly passed through the `QVariant` objects.

```
Original variants of people
  John Doe
  Jane Doe
  Albert A Einstein
People after variants
  John Doe
  Jane Doe
  Albert A Einstein
```

Solution Tips

Step 1

Use the following line to read the text from one of the `QLineEdit` widgets.

```
    QString firstValueText = m_firstValue->text();
```

Convert a string to an integer using the `toInt` method.

```
    bool ok;
    int firstValue = firstValue.toInt(&ok);
```

Use the `setText` method to update the result.

```
    m_result->setText("no result");
```

Use the `QString::number` method to convert a number to a `QString`.

Step 2

Create a file object using the following line.

```
    QFile file("listitems.txt");
```

Do not forget to open file object with the argument `QIODevice::ReadOnly` or `QIODevice::WriteOnly` depending on what you are trying to achieve.

When writing to the file, use the following loop to retrieve the items.

```
    for(int i=0; i<m_list->count(); ++i)
        // The item's text is available from m_list->item(i)->text()
```

Add a line break at the end of each line by adding `endl` to the stream.

When loading from the file, use `m_list->addItem` to add each line to the list.

Step 3

When implementing a stream operator for a type built on existing types, e.g. a class containing strings and integers, simply use the existing stream operators in your implementation. For instance:

```
QDataStream &operator<<(QDataStream &stream, const SomeType &t)
{
    stream << t.string();
    stream << t.integer();
    stream << t.double();
    ...
```

Do not forget to return the stream object given from your stream operators.

Ensure that you read and write the items in the same order. A file works as a FIFO buffer.

When reading from a file, use a temporary of the expected type to hold the value before you pass it on to the target object.

```
QDataStream &operator>>(QDataStream &stream, SomeType &t)
```

```
{
    QString t;
    stream >> t; t.setString(t);
    ...
```

Step 4

The `qHash` function returns a `uint`.

For hashing the first name and initials, either join the strings and hash them, or join the hash values of each string.

The `operator==` simply compares the first name, initials and family name of two given objects. Unless all match, the operator should return `false`.

Step 5

Use the `Q_DECLARE_METATYPE` macro.