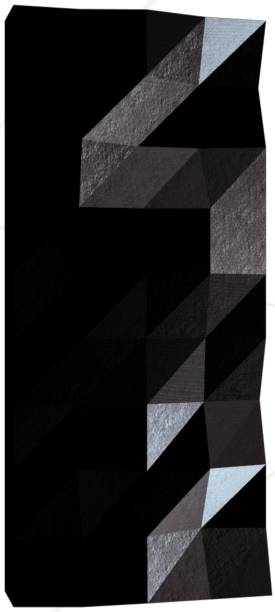

Qt essentials

Qt Core library basic features

Qt training training@qt.io
42 Pedago Staff pedago@42.fr

Summary: Qt Core library is the basis of the Qt framework. All other Qt modules (libraries) depend on Qt Core. It provides Qt basic classes, file handling, data streams, XML and JSON parsing, process and thread management, and many other features.

Contents

Chapter I

General instructions

Unless explicitely specified, the following rules will apply every day of this Piscine.

- This subject is the one and only trustable source. Don't trust any rumor.
- This subject can be updated up to one hour before the turn-in deadline.
- The assignments in a subject must be done in the given order. Later assignments won't be rated unless all the previous ones are perfectly executed.
- Be careful about the access rights of your files and folders.
- You must follow the `turn-in process` for each assignment. The url of your `GIT` repository for this day is available on your intranet.
- Your assignments will be evaluated by your Piscine peers.
- In addition to your peers evaluation, a program called the "Moulinette" will also evaluate your assignments. Fully automated, The Moulinette is tough and unforgiving in its evaluations. As a consequence, it is impossible to bargain your grade with it. Uphold the highest level of rigor to avoid unpleasant surprises.
- You must not leave in your turn-in repository any file other than the ones explicitly requested by the assignments.
- You have a question? Ask your left neighbor. Otherwise, try your luck with your right neighbor.
- Every technical answer you might need is available in the `mans` or in the Qt Documentation, which is available both in `QtCreator` IDE and on http://doc.qt.io.
- Remember to use the Piscine forum of your intranet and also Slack!
- You must read the examples thoroughly. They can reveal requirements that are not obvious in the assignment's description.
- Use the latest Qt version. Qt version 5.10 or newer is recommended.

- Many Qt classes are available as standard C++ classes. Qt classes should be preferred to standard classes, as you are supposed to learn Qt.
- Basic Qt coding style should be used, so read `http://wiki.qt.io/Qt_Coding_Style` before writing any assignments.
- Deprecated macros, functions or classes must not be used.
- Any build system, such as `qmake`, `cmake` or Qt Build System, can be used in the assignments. Instructions are based on `qmake`.
- Any editor or IDE, supporting Qt, can be used. However, `QtCreator` usage is strongly recommended.
- By Thor, by Odin! Use your brain!!!

Chapter II

Day-specific instructions

All assignments in the following days will be based on what you learn today. Make sure to learn the essential Qt features well. Some features, you will need all the time ,are the following:

- How to manage `QObject` memory using parent-child relationship or smart pointers?
- How to communicate between `QObjects` using signals/slots or events?
- How to use basic string operations? What other options Qt provides for character handling?
- How to choose a right item container for your need? How to iterate items in item containers?

All assignments for today have a similar structure, consisting of a project file (`.pro`) and a `main.cpp` file. You can create a basic project manually or use `Qt Creator` project templates to create an empty `qmake` project. The template is located under "Other Project templates". Use "Add New" to add the `main.cpp` file to the project. All code you need to write yourself.

Chapter III

Assignment 00: Hello world in Qt

In addition to debugging, logging and asserts help in developing Qt programs. Write an application, which uses `QDebug` to print "Hello World" to the debug console.

Chapter IV

Assignment 01: Memory management

Write a program, which allocates `QObjects` in the heap and uses Qt-specific ways to delete the allocated objects.

- Create a `QObject` subclass using the `Qt Creator` IDE.
- Reimplement at least the destructor in your subclass. Log the object name of the destructed object to make it easy to observe, when the object is deleted without any breakpoints.
- Allocate three instances of your subclass in the heap. Use a stack allocated parent to take care of de-allocation one object from the heap. Use two different Qt smart pointers to de-allocate two other instances. Using standard smart pointers is not allowed here.

The output of the program should be something similar to the following. Obviously, the subclass name or object names are not relevant.

```
Object "Child 3" destructed
Object "Child 2" destructed
Object "Parent" destructed
Object "Child 1" destructed
```

Chapter V

Assignment 02: String handling

Let's have fun with Qt strings. Implement the following functionality. Pay attention to the efficiency and avoid using unnecessary temporary variables. Avoid any magic numbers. After each item, log the string or result to the debug console.

- Create `QString` "Qt Rules " and concatenate an integer 42 to it.
- Check if the string contains "Qt" and if yes, add a string " always" after "Qt".
- Append the string to itself ten times. Calculate, how many times "Qt" occurs in the string.

Chapter VI

Assignment 03: Object communication - signals

It is possible to create many-to-many observer connections between `QObjects`. This is the preferred way, how objects should communicate.

Your task is to implement a Qt program, where two objects change simple data using the observer pattern, based on signals and slots. As we are not running an event loop yet, there are not many signals what we can use. That is why, you should derive two new classes from `QObject`: `Sender` and `Receiver` and add some signals and slots to them. `Sender` should have a signal to notidy `Receiver`, when its object name has changed. `Receiver` should log the observer object name to the console and change the name to a new value.

- Start by creating `Sender` and `Receiver` subclasses. Derive them directly from `QObject`.
- `Sender` should have two signals: one with a `const QString &` argument and another with two arguments: `const QString &` and an integer. `Sender` also needs a slot with a `const QString &` argument. Declare the slot to be private.
- `Receiver` should have one slot with the same two arguments as the `Sender` signal.
- When `Sender` object name changes, the matching slot should be called. The slot should emit another signal with an object name and integer value (can be a magic number).
- `Receiver` should log the received object name and integer. In addition, `Receiver` should change `Sender` object name. Do not give a pointer to `Sender` to `Receiver`, but change the `const` name argument directly.
- Log the new name in `Sender`.
- Finally, log a statement, after `Sender` has been destroyed. You are not allowed to add a new slot in `Sender`, but you need to do the logging in a lambda function.

The following log clarifies the idea, what is required.

```
Receiver says "Sender" 42
Sender says "New Sender name set by Receiver"
Sender destroyed
```

Chapter VII

Assignment 04: Object properties

As a first task in your new job in the IT department of a big company you are asked to implement a management system for sales accounts. Actually, the sales account is just a `QObject` subclass with two properties: employee and sales regions. Employee is a value type with very basic employee data, like name and salary. Sales regions are continents.

- Subclass `QObject` to create a sales account class. Add an enumeration of continents to the class. Define each value in the enumeration as a flag. Add a property to the sales account class, which defines sales regions as a combination of flags.
- Instantiate your sales account class. Connect a signal, which notifies that sales regions property has changed. Change the sales region to some value and log to the debug console the new value. Simple integer is enough. There is no need to map the flag values to any continent names.

```
MEMBER will not work for flag properties.  Declare and implement
getter and setter functions for the property using READ and WRITE.
```

- Add another property to the sales account class. The property is your own value type, containing two member variables: an employee name as `QString` and salary as `float`. Implement getter and setter functions for your members. Implement also the `!=` operator to the new class. This is needed by the Metatype system.
- Use the `setProperty()` function to set employee data object to the sales account. Read the name and salary values using `property()` function and log the values to the debug console.

```
Property types must be known by the Metatype system.
```

The log below clarifies, what the program should do.

```
Regions changed to QVariant(int, 16)
Sales account "Hard-working Developer"
has a salary of 7200.16
```

Chapter VIII

Assignment 05: Object communication - Events

Qt is an event-based system, so quite likely you need to handle events in almost any Qt application. Implement a program handling events in three different ways: reimplementing event-specific event handler. reimplementing a generic event handler, and reimplementing an event filter.

- So far we have not used any event loop in our programs, which is quite exceptional. Add and start an event loop, provided by `QCoreApplication`. If you run the program, it never finishes, unless you kill it.
- Add a `QObject` subclass to the project and re-implement the `timerEvent()` function. Start the timer and when the timer expires, say after 3000 ms, return from the main function event loop. Do not return from the event loop, if any other timer but the one you just started, expires. Use your custom class in the `main()` function to close the application. Re-implementing event-specific handler functions is the most common way to handle events in Qt.
- Subclass `QEvent` to have a custom event. Add `QString` member to identify the event and getter and setter functions for the member. Define in the `customevent.h` file a custom event type `const QEvent::Type` and assign a value to your type.
- Reimplement the `event()` function in the custom class, in which you used the timer previously. In case the event is the custom type, log the string member to the debug console. This way to handle events is less common, but needed in touch and gesture handling.
- Send the custom event first asynchronously and then synchronously to your custom `QObject`.
- Add an event filter to filter your custom event. In this case, the custom events are never handled, but other events, like timer events, are handled in a normal way. Event filters are the third way to handle events. They are useful, if similar custom behaviour is required in many different types.

Chapter IX

Assignment 06: Data streaming

Continue your task to manage sales accounts. Stream employee data into a file and read the data back from the file.

- You may re-use the same `Employee` class from the previous assignment. Just add the required functionality.
- Serialise employee data into a temporary file. Use Qt temporary file class.
- As there may be several data types, which will be needed to be serialised later, use `QVariant`. So it is not allowed to serialise `Employee` to the file directly.
- De-serialise the data from the temporary file using `QVariant`. Log the employee data, read from the file, into the debug console.

Chapter X

Assignment 07: Item containers

As a final assignment for today, you need to prepare data structures for employee data to be stored into the database. Re-use the same `Employee` class from the previous assignment. This time no changes to this class are needed.

Your task is to store and manipulate 10,000 employees using Qt item containers and Qt iterators using the following requirements.

- You must use an associative container, which stores the items in total order. The container key type must be `QPoint`.

`Qt containers do not know, how to sort QPoints, as there is no global < operator, comparing two QPoints. Compare first the x values of two points. If they are equal, compare the y values.`

- Add 10,000 employees into the container. Each employee may have the same name, but use their salary to uniquely identify each employee.
- Implement a function, which iterators through all employees and logs employee name and salary to the debug console. Call the function after adding 10,000 items to the container.
- Remove every third employee in the container. Log the result.
- Change every fifth employee so that the name will be "NN n", where n is the ordinal number of the item. Log the result.
- Use an algorithm to calculate, how many employees with the name "NN" your container has.