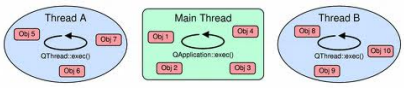
Qt Fundamentals: Threading

Multithreading

Threading can be difficult to get right - so don't use it unless you really need it.

- Most GUI applications have a single thread of execution in which the event loop is running
- However, if the user invokes a time consuming operation the interface freezes. We can work around this in different ways:
  - Using the QApplication::processEvent() during long tasks to make sure events (key, window, etc.) are delivered and the UI stays responsive.
  - Using threads to perform the long running tasks. Qt has a number of options for this.



Threading Concepts

Reentrancy:
- A reentrant function/class can be called simultaneously from multiple threads, but only if each thread uses its own instance / data.

Thread-safe:
- A thread-safe function/class can be called simultaneously from multiple threads, even if each thread access the same instance / data.

The Qt documentation typically specify whether a class/function is thread-safe or reentrant.

Qt Threading Classes

- QAtomicInt / QAtomicPointer
  - Atomic operations on integers and pointers
- QMutex (helper: QMutexLocker)
  - Coordination between threads
- QReadWriteLock (helper: QReadLocker and QWriteLocker)
  - Concurrent access between readers and writers
- QSemaphore
  - Resource counting semaphore
- QWaitCondition
  - Condition variable for synchronizing threads
- QThread
  - Basic thread implementaion
- QThreadPool / QRunnable
  - Manage worker threads and jobs
- QtConcurrent
  - High-level concurrency API

Simple Threading Example

```
class TestThread : public QThread
{
   Q_OBJECT
public:
   TestThread();

   void setMessage(const QString &message);
   void stop();

protected:
   void run();

private:
   QMutex m_mutex;
   bool m_stop;

   QString m_message;
};
```

- Using RAII structures like QMutexLocker simplifies code a lot.

```
TestThread::TestThread() : m_stop(false)
{}

void TestThread::setMessage(const QString &message)
{
   QMutexLocker l(&m_mutex);
   m_message = message;

}

void TestThread::stop()
{
   QMutexLocker l(&m_mutex);
   m_stop = true;
}


void TestThread::run()
{
   {
      QMutexLocker l(&m_mutex);
      m_stop = false;
   }

   forever
   {
      {
         QMutexLocker l(&m_mutex);

         if(m_stop) break;

         qDebug() << m_message;
      }

      QThread::msleep(1);
   }
}
```


Using Qt's Classes in Threads

As QObject is reentrant it is possible to use QObject in other threads than the main thread.
- An exception to this the Q(Core)Application, QPixmap and Qt GUI classes i.e. QWidget derived classes.

A Object "lives" in the thread in which it was created
- Therefore it must also be deleted in that thread (can easily be achieved by creating them on the stack in the run() method).
- Child QObjects must be created in the parent's thread

Signals and Slots for Threads

- We can use signals and slots safely across thread boundaries. However there are a couple of things we should know:
  - Each thread can have its own event loop, this is needed to deliver events and to invoke slots
    - Some classes e.g. QTimer, and network related classes require the event loop running.
  - A thread does not need the event loop if it just emit's signals.

```
bool QObject::connect ( const QObject * sender, const char * signal, const QObject * receiver,
                        const char * method, Qt::ConnectionType type = Qt::AutoConnection )
```

- QThread::exec() starts the loop

Qt::QueuedConnection