Consumer

```
class Consumer : public QThread
{
    Q_OBJECT
public:
    explicit Consumer( int *buffer, int bufferSize,QSemaphore * freeSpace,
                       QSemaphore * availableSpace,
                       bool * atEnd,QObject *parent = nullptr);

    // QThread interface
protected:
    void run() override;

private:
    int * m_buffer;
    int m_BUFFER_SIZE;
    QSemaphore * m_free_space;// Space where the producer can write data
    QSemaphore * m_available_space;//Space where consumer can read data from
    bool * m_at_end;
};
```

Consumer

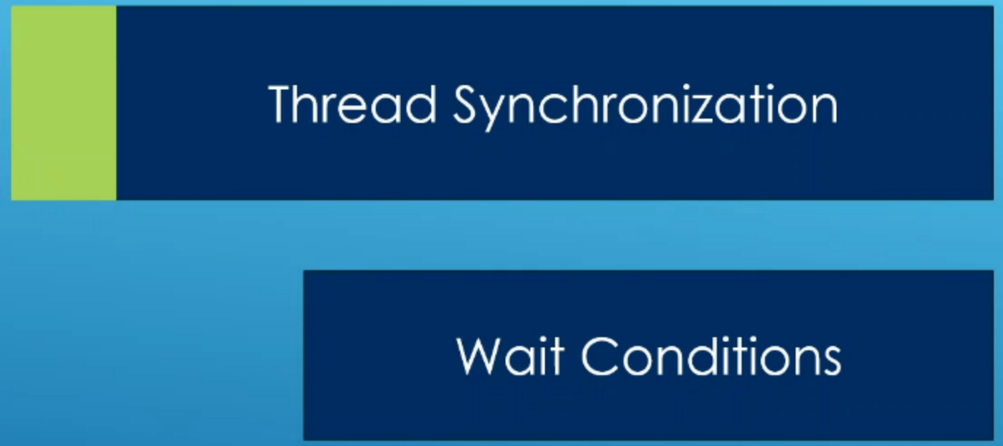
```
Consumer::Consumer( int * buffer,int bufferSize,
                    QSemaphore *freeSpace, QSemaphore *availableSpace,
                    bool *atEnd, QObject *parent) : QThread (parent),
        m_buffer(buffer),m_BUFFER_SIZE(bufferSize),
        m_free_space(freeSpace),
        m_available_space(availableSpace),
        m_at_end(atEnd)

{

}

void Consumer::run()
{
    int i{0};

    while(!(*m_at_end) || m_available_space->available()){

        m_available_space->acquire();
        qDebug() << "[" << i << "]" << m_buffer[i];//Consume data

        i =((i+1) % (m_BUFFER_SIZE));
        //Signal that there is one more slot of free space for the producer to use
        m_free_space->release();
    }
    qDebug() << "Consumer done using data!";
}
```

Wrap it all up

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

    dataSource << 1 << 2 << 34 <<55 << 4<< 11 << 22 << 34 <<55 <<
                  4<< 1 << 2 << 34 <<55 << 4<< 1 << 2 << 34 <<55 << 4<<2222;

    atEnd = false;

    freeSpace = new QSemaphore(BUFFER_SIZE);

    avSpace = new QSemaphore(0);

    producer = new Producer(dataSource,bufferArray,BUFFER_SIZE,freeSpace,avSpace,&atEnd);

    consumer = new Consumer(bufferArray,BUFFER_SIZE,freeSpace,avSpace,&atEnd);
}
```

Off it goes !

```
void Widget::on_startButton_clicked()
{
    producer->start();
    consumer->start();

    producer->wait();
    consumer->wait();

    atEnd = false;
}
```



Course content

Code execution is going to FREEZE when a call wo QWaitCondition::wait() is made.

Code execution RESUMES when a call is make to QWaitCondition::wake somewhere in the app, on the SAME QWaitCondition object

Look for: QWai
qWait
QWaitCondition
qWaitFor
qWaitForWindowActive
qWaitForWindowExposed
qWaitForWindowShown
~QWaitCondition

Detailed Description

The QWaitCondition class provides a condition variable for synchronizing threads.

QWaitCondition allows a thread to tell other threads that some sort of condition has been met. One or many threads can block waiting for a QWaitCondition to set a condition with wakeOne() or wakeAll(). Use wakeOne() to wake one randomly selected thread or wakeAll() to wake them all.

For example, let's suppose that we have three tasks that should be performed whenever the user presses a key. Each task could be split into a thread, each of which would have a run() body like this:

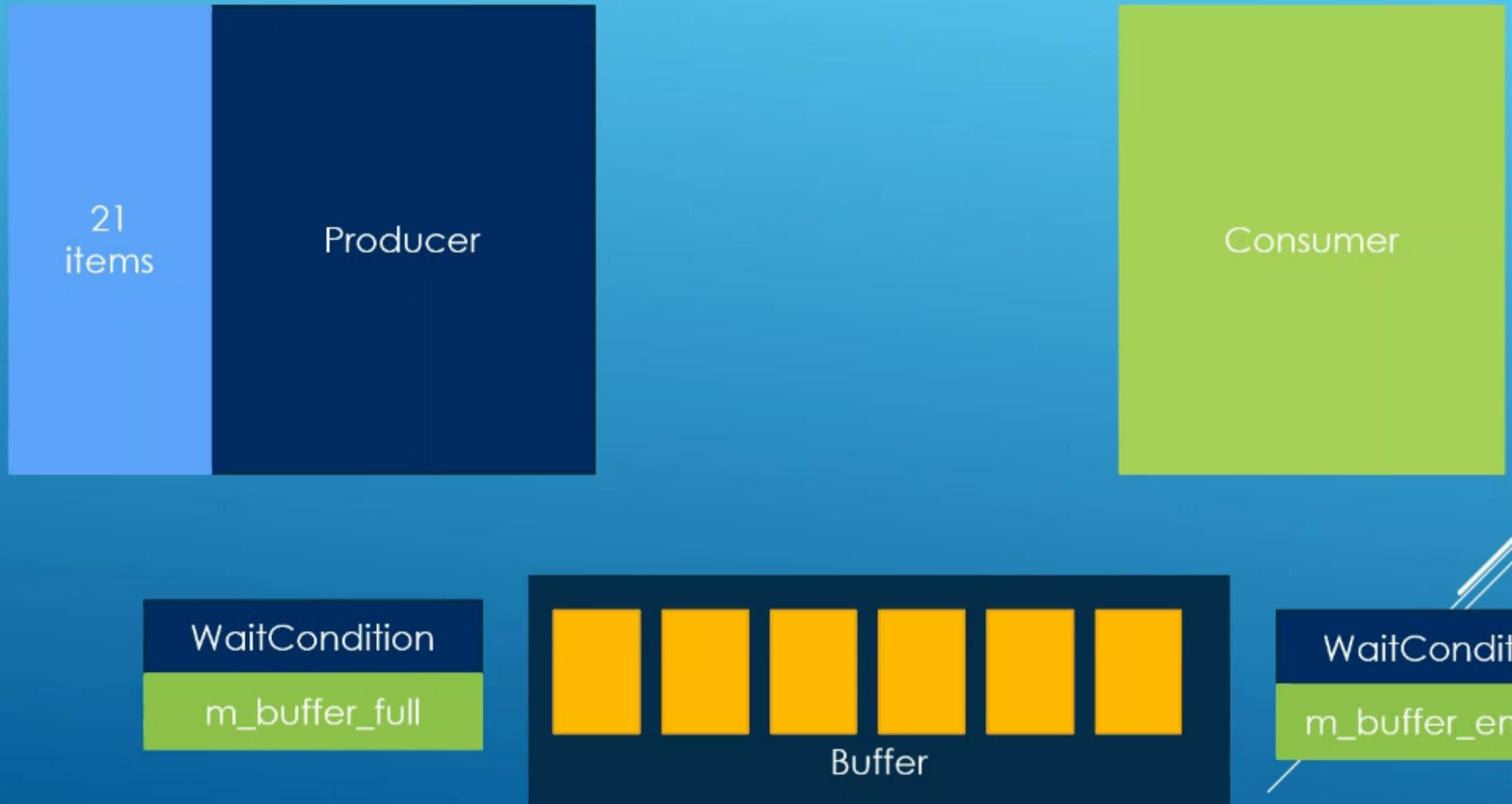
```
forever {
    mutex.lock();
    keyPressed.wait(&mutex);
    do_something();
    mutex.unlock();
}
```

Here, the `keyPressed` variable is a global variable of type QWaitCondition.

A fourth thread would read key presses and wake the other three threads up every time it receives one, like this:

```
forever {
    getchar();
    keyPressed.wakeAll();
}
```

The order in which the three threads are woken up is undefined. Also, if some of the threads are still in `do_something()` when the key is pressed, they won't be woken up (since they're not waiting on the condition variable) and so the task will not be performed for that key press. This issue can be solved using a counter


21
items
Producer
Consumer
WaitCondition
m_buffer_full
Buffer
WaitCondition
m_buffer_empty

```
const int M_BUFFER_SIZE = 6000;
class Widget : public QWidget
{
    Q_OBJECT

public:
    explicit Widget(QWidget *parent = nullptr);
    ~Widget();

private slots:
    void on_startButton_clicked();

private:
    Ui::Widget *ui;

    int totalData;
    char buffer[M_BUFFER_SIZE];
    QWaitCondition bufferFull;
    QWaitCondition bufferEmpty;
    QMutex mutex;
    int usedSlots;
    Producer * producer;
    Consumer * consumer;
};
```

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

     totalData = 100000;
     usedSlots = 0;

     producer = new Producer(buffer,M_BUFFER_SIZE,&usedSlots,totalData,
                             &mutex,&bufferFull,&bufferEmpty);

     consumer = new Consumer(buffer,M_BUFFER_SIZE,&usedSlots,totalData,
                             &mutex,&bufferFull,&bufferEmpty);
     connect(consumer,&QThread::finished,[](){
         qDebug() << "Consumer done!";
     });
}
```

Producer

```
void Producer:: run() {

    for (int i = 0; i < m_total_data; ++i) {
        m_mutex->lock();
        if ((*m_used_slots) == m_buffer_size)
            m_buffer_full->wait(m_mutex);
        m_mutex->unlock();

        //qDebug() << "Producer, buffer used slots : " << (*m_used_slots);
        m_buffer[i % m_buffer_size] = "BACK"[QRandomGenerator::global()->bounded(4)];

        m_mutex->lock();
        ++(*m_used_slots);
        m_buffer_empty->wakeAll();
        m_mutex->unlock();
    }
}
```

Producer

```
void Producer:: run() {

    for (int i = 0; i < m_total_data; ++i) {
        m_mutex->lock();
        if ((*m_used_slots) == m_buffer_size)
            m_buffer_full->wait(m_mutex);
        m_mutex->unlock();

        //qDebug() << "Producer, buffer used slots : " << (*m_used_slots);
        m_buffer[i % m_buffer_size] = "BACK"[QRandomGenerator::global()->bounded(4)];

        m_mutex->lock();
        ++(*m_used_slots);
        m_buffer_empty->wakeAll();
        m_mutex->unlock();
    }
}
```

Consumer

```
void Consumer::run(){

    for (int i = 0; i < m_total_data; ++i) {
        m_mutex->lock();
        if ((*m_used_slots) == 0)
            m_buffer_empty->wait(m_mutex);
        m_mutex->unlock();

        qDebug() << "Consumer data : " << m_buffer[i % m_buffer_size];

        m_mutex->lock();
        --(*m_used_slots);
        m_buffer_full->wakeAll();
        m_mutex->unlock();
    }
}
```

Look for: QWai
qWait
QWaitCondition
qWaitFor
qWaitForWindowActive
qWaitForWindowExposed
qWaitForWindowShown
~QWaitCondition

Detailed Description

The QWaitCondition class provides a condition variable for synchronizing threads.

QWaitCondition allows a thread to tell other threads that some sort of condition has been met. One or many threads can block waiting for a QWaitCondition to set a condition with wakeOne() or wakeAll(). Use wakeOne() to wake one randomly selected thread or wakeAll() to wake them all.

For example, let's suppose that we have three tasks that should be performed whenever the user presses a key. Each task could be split into a thread, each of which would have a run() body like this:

```
forever {
    mutex.lock();
    keyPressed.wait(&mutex);
    do_something();
    mutex.unlock();
}
```

Here, the `keyPressed` variable is a global variable of type QWaitCondition.

A fourth thread would read key presses and wake the other three threads up every time it receives one, like this:

```
forever {
    getchar();
    keyPressed.wakeAll();
}
```

The order in which the three threads are woken up is undefined. Also, if some of the threads are still in `do_something()` when the key is pressed, they won't be woken up (since they're not waiting on the condition variable) and so the task will not be performed for that key press. This issue can be solved using a counter

A call to QWaitCondition FREEZES code execution

Thread with Pause Resume

```
class WorkerThread : public QThread
{
    Q_OBJECT
public:
    explicit WorkerThread(QObject *parent = nullptr);
    ~WorkerThread() override;


signals:
    void currentCount(int value);

public slots:
    void pause();
    void resume();

    // QThread interface
protected:
    void run() override;
private:
    QMutex m_mutex;
    QWaitCondition m_pause_condition;
    bool m_pause;
};
```

Thread with Pause Resume

```
WorkerThread::WorkerThread(QObject *parent) : QThread(parent)
{
    m_pause = false;
}

WorkerThread::~WorkerThread()
{
    qDebug() << "WorkerThread object destroyed";
}

void WorkerThread::pause()
{
    m_mutex.lock();
    m_pause = true;
    m_mutex.unlock();
}

void WorkerThread::resume()
{
    m_mutex.lock();
    m_pause = false;
    m_mutex.unlock();
    m_pause_condition.wakeAll();
}
```

Thread with Pause Resume

```
void WorkerThread::run()
{
    for(int i{0} ; i < 1000000001 ; i++){
        /* ...
        m_mutex.lock();
        if(m_pause)
            // in this place, your thread will stop to execute until someone calls resume
            m_pause_condition.wait(&m_mutex);
        m_mutex.unlock();

        /* ...
        if((i%100000) == 0){
            double percentage = ((i/1000000000.0)) * 100;
            emit currentCount(QVariant::fromValue(percentage).toInt());
        }
    }
}
```

Thread with Pause Resume

```
WorkerThread::WorkerThread(QObject *parent) : QThread(parent)
{
    m_pause = false;
}

WorkerThread::~WorkerThread()
{
    qDebug() << "WorkerThread object destroyed";
}

void WorkerThread::pause()
{
    m_mutex.lock();
    m_pause = true;
    m_mutex.unlock();
}

void WorkerThread::resume()
{
    m_mutex.lock();
    m_pause = false;
    m_mutex.unlock();
    m_pause_condition.wakeAll();
}
```

Thread Synchronization

Chapter Review

QMutex

QWaitCondition

QReadWriteLocker

QMutexLocker

QSemaphore

Pause Resume

Thread Safety and Reentrancy

Course content

Section 4: Thread Safety and Reentrancy

0 / 6 | 1hr 10min

- 23. Thread Safety and Reentrancy Overview — 16min
- 24. Cross Thread Signals and Slots - Example1 — 22min — Resources
- 25. Cross Thread Signals and Slots - Example2 — 14min — Resources
- 26. Cross Thread Signals and Slots - Example3 — 7min — Resources
- 27. Slots in QThread Subclass — 10min — Resources
- 28. Thread Safety and Reentrancy - Chapter Review — 1min

Section 5: Qt Concurrent

0 / 12 | 2hr 37min

Thread Safe Function	Can be safely invoked at the same time, from multiple threads, on the same data, without synchronization

Reentrant Function	Can be safely invoked at the same time, from multiple threads, on different data, otherwise external synchronization is needed

Non Reentrant Function (Thread unsafe)	Can NEVER be invoked from more than one thread

By extension, a class is said to be *reentrant* if its member functions can be called safely from multiple threads, as long as each thread uses a *different* instance of the class. The class is *thread-safe* if its member functions can be called safely from multiple threads, even if all the threads use the *same* instance of the class.

Thread Safety and Reentrancy

https://doc.qt.io/qt-5/threads-reentrancy.html

Reentrant Function	Can be safely invoked at the same time, from multiple threads, on different data, otherwise external synchronization is needed

By extension, a class is said to be *reentrant* if its member functions can be called safely from multiple threads, as long as each thread uses a *different* instance of the class. The class is *thread-safe* if its member functions can be called safely from multiple threads, even if all the threads use the *same* instance of the class.

Thread Safety and Reentrancy

https://doc.qt.io/qt-5/threads-reentrancy.html

Thread Safe
(Examples)

- QMutex
- QObject::connect
- QMetaObject::invokeMethod
- QCoreApplication::postEvent

Reentrant
(Examples)

- QString
- QImage
- Value classes

Non Reentrant
(Examples)

- QWidget (and subclasses)
- QPixmap
- All GUI classes only usable in main thread

QObject Thread Affinity

- Each QObject knows the thread it's living in
- It holds a reference to the thread it was created in (QObject::thread()). We say that the object has affinity with that thread
- We can move the affinity of an object to another thread with QObject::moveToThread()

QObject Class

The QObject class is the base class of all Qt objects. More...

Header: #include <QObject>

qmake: QT += core

Inherited By: QAbstractAnimation, QAbstractEventDispatcher, QAbstractItemModel, QAbstractState, QAbstractTransition, QCoreApplication, QEventLoop, QFileSelector, QFileSystemWatcher, QIODevice, QItemSelectionModel, QLibrary, QMimeData, QObjectCleanupHandler, QPluginLoader, QSettings, QSharedMemory, QSignalMapper, QSocketNotifier, QThread, QThreadPool, QTimeLine, QTimer, QTranslator, and QWinEventNotifier

- List of all members, including inherited members
- Obsolete members

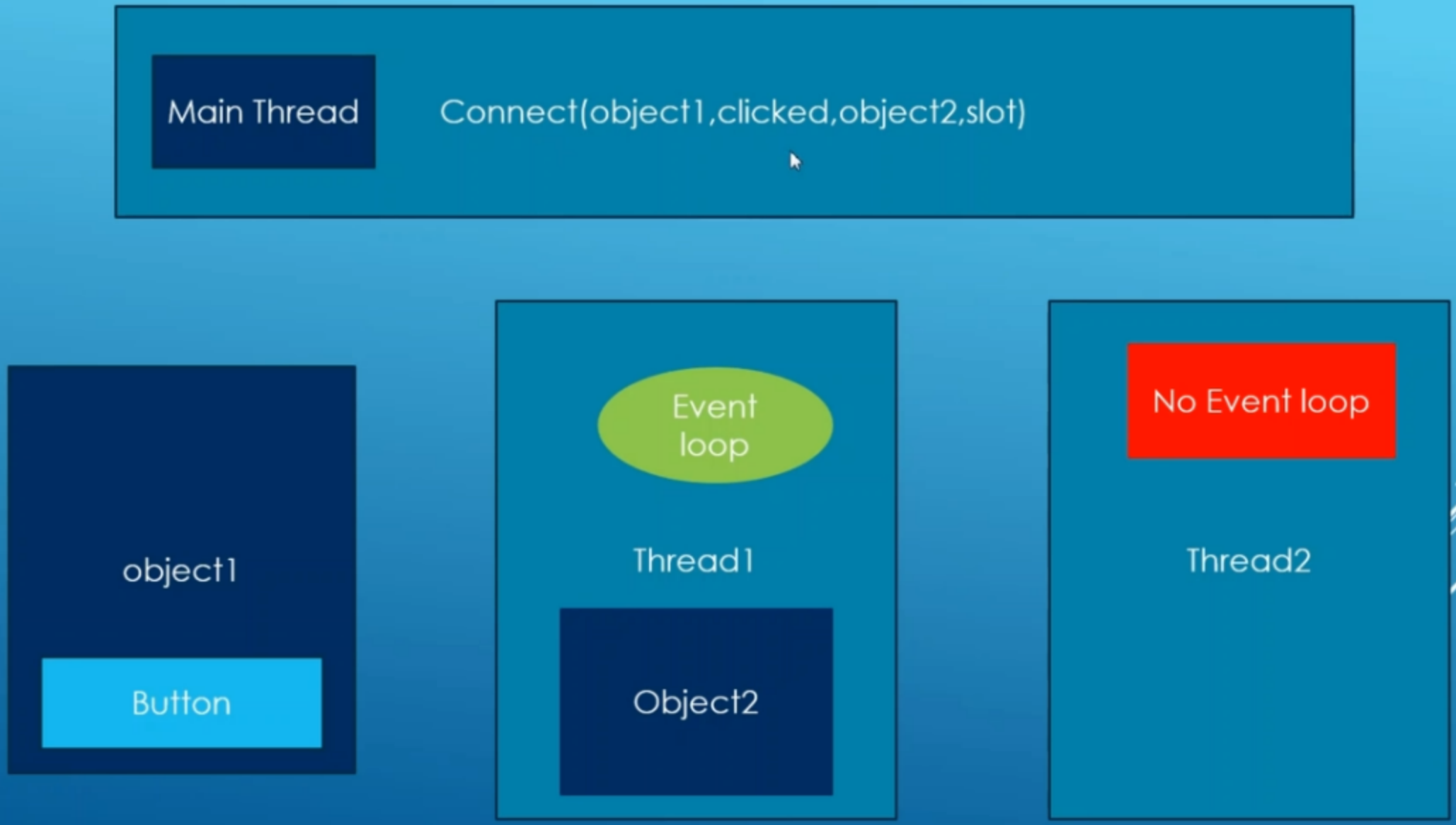
Note: All functions in this class are reentrant.

Note: These functions are also thread-safe:

- Event based classes like timers and sockets are not reentrant : You can't call methods on them from threads other than the one they have affinity with

* If you violate this guideline, the behavior is undefined. Sometimes you get a warning on the console

Event dispatching for QObject instances happens in the thread it has affinity with


Main Thread
Connect(object1,clicked,object2,slot)
object1
Button
Event loop
Thread1
Object2
No Event loop
Thread2

Event dispatching for QObject instances happens in the thread it has affinity with

All objects in the same parent/child tree must have the same thread affinity

WorkerThread lives in main thread

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

    workerThread = new WorkerThread(this);
}
```

Object lives in secondary thread

```
void WorkerThread::run()
{
    QObject * object = new QObject(this);
}
```

- You must delete all QObjects living in a QThread before deleting the thread object itself
- You can call moveToThread on a QObject from the same thread the QObject instance has affinity with

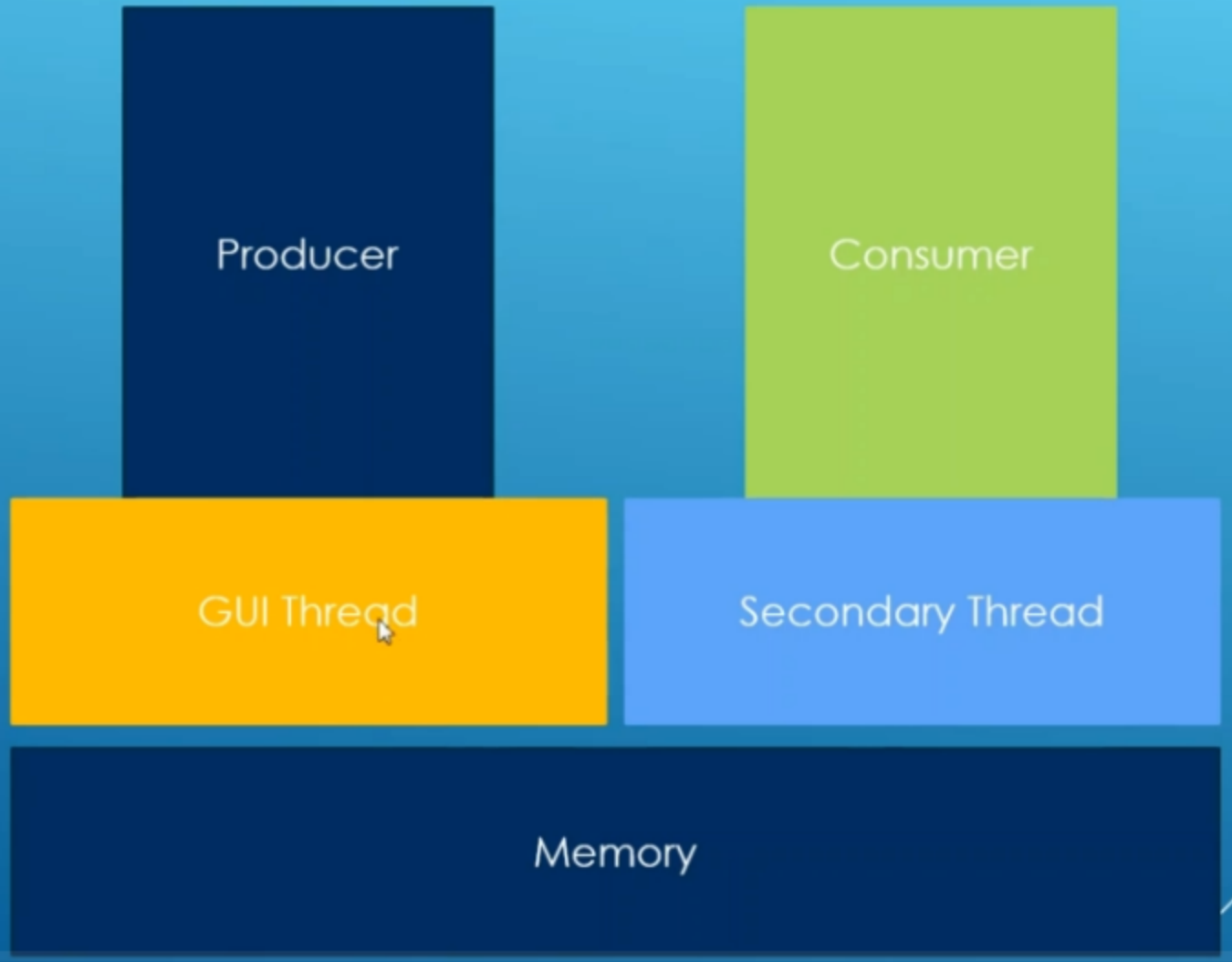
- Think of QObject as non reentrant in practice. It will make things easier.
- Only touch a QObject from the thread where it is living

We can emit the signal from one thread, and have the slot invoked in another thread (the thread the receiver object has affinity with)

- Connection is queued if receiver lives in different thread that object emitting signal
- For queued connections, a metacall event is posted to the receiver's event queue
- The event will be dispatched to the receiver object by the correct thread
- Metacall events are handled by calling the intended slot
- Because events are involved, the receiver's thread needs an event loop running

Cross Thread Signals and Slots

Example 1


Producer
Consumer
GUI Thread
Secondary Thread
Memory

Producer

```
class Producer : public QPushButton
{
    Q_OBJECT
public:
    explicit Producer(QWidget *parent = nullptr);

signals:
    void data(const CustomData &);

protected slots:
    void sendData();

private:
    int m_integer;
};
```

Producer

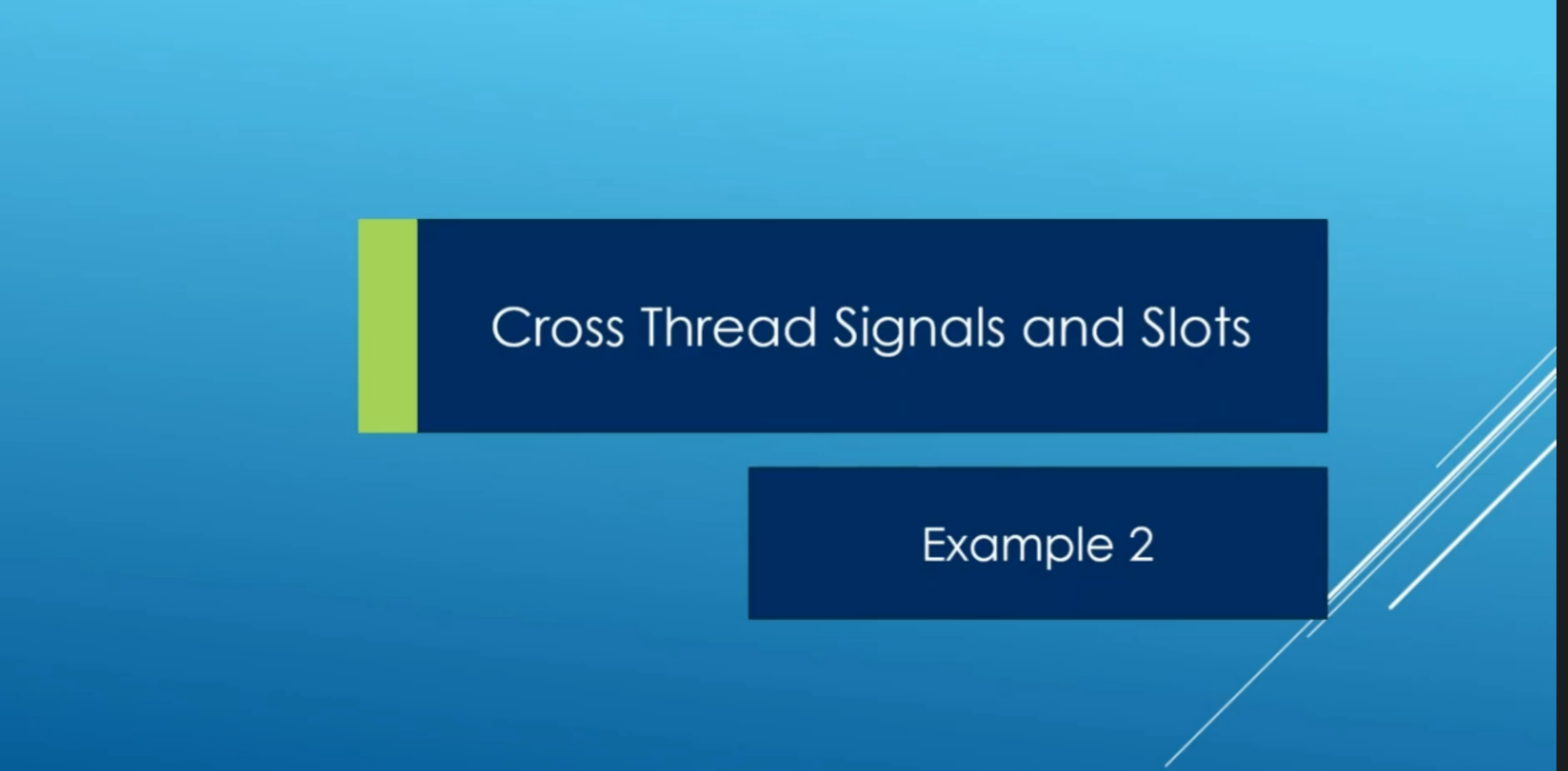
```
Producer::Producer(QWidget *parent) : QPushButton("Push me", parent)
{
    connect(this, &Producer::clicked, this, &Producer::sendData);

    m_integer = 100;
}

void Producer::sendData()
{
    qDebug() << "Producer producing data , thread :" << QThread::currentThread();
    CustomData cd(10, &m_integer, "Produced data");
    emit data(cd);
}
```

Consumer

```
Consumer::Consumer(QObject *parent) : QObject(parent),
    m_counter(0)
{
}

void Consumer::data(const CustomData &cd)
{
    qDebug() << "Consumer message : " << cd.m_integer << *cd.m_pointer << cd.m_string
             << " thread : " << QThread::currentThread();
    // Kill the thread
    if (++m_counter > 10) {
        qDebug() << "Thread about to be killed";
        thread()->quit();
    }
}
```

Connection

```
producer = new Producer(this);
ui->verticalLayout->addWidget(producer);

consumer = new Consumer();

thread = new QThread(this);

connect(thread, &QThread::finished, [=](){
    qDebug()<< "Quitting application from lambda, thread : " << QThread::currentThread();
    QApplication::quit();
});

//Cross thread connection
connect(producer, &Producer::data, consumer, &Consumer::data);
consumer->moveToThread(thread);

thread->start();
```



Course content

Section 3: Thread Synchronization
8 / 8 | 2hr 9min

Section 4: Thread Safety and Reentrancy
2 / 6 | 1hr 10min

- [x] 23. Thread Safety and Reentrancy Overview
  16min
- [x] 24. Cross Thread Signals and Slots - Example1
  22min Resources
- [ ] 25. Cross Thread Signals and Slots - Example2
  14min Resources
- [ ] 26. Cross Thread Signals and Slots - Example3
  7min Resources
- [ ] 27. Slots in QThread Subclass
  10min Resources
- [ ] 28. Thread Safety and Reentrancy - Chapter Review
  1min

Subclass Qthread and declare consumer in run() method

```
void Thread::run()
{
    Consumer consumer;
    connect(m_producer, &Producer::data, &consumer, &Consumer::data);
    exec();
}
```

Connection

```
producer = new Producer();

ui->verticalLayout->addWidget(producer);

thread = new Thread(producer);

thread->start();

producer->startProduction();
```

Course content

Section 3: Thread Synchronization
8 / 8 | 2hr 9min

Section 4: Thread Safety and Reentrancy
3 / 6 | 1hr 10min

- [x] 23. Thread Safety and Reentrancy Overview
  16min
- [x] 24. Cross Thread Signals and Slots - Example1
  22min Resources
- [x] 25. Cross Thread Signals and Slots - Example2
  14min Resources
- [ ] 26. Cross Thread Signals and Slots - Example3
  7min Resources
- [ ] 27. Slots in QThread Subclass
  10min Resources
- [ ] 28. Thread Safety and Reentrancy - Chapter Review
  1min

Setup connections beforehand, making sure consumer never misses unit from producer

```
producer = new Producer(this);
ui->verticalLayout->addWidget(producer);

consumer = new Consumer();//Don't give a parent to this as parents
                          //to objects can't span multiple threads

thread = new QThread(this);

connect(producer, &Producer::data, consumer, &Consumer::data);
connect(thread,&QThread::started,producer,&Producer::startProduction);
connect(thread, &QThread::finished,consumer, &QObject::deleteLater);

consumer->moveToThread(thread);

thread->start();
```

```
producer = new Producer(this);
ui->verticalLayout->addWidget(producer);

consumer = new Consumer();//Don't give a parent to this as parents
                          //to objects can't span multiple threads

thread = new QThread(this);

connect(producer, &Producer::data, consumer, &Consumer::data);
connect(thread,&QThread::started,producer,&Producer::startProduction);
connect(thread, &QThread::finished,consumer, &QObject::deleteLater);

consumer->moveToThread(thread);

thread->start();
```

Section 3: Thread Synchronization

8 / 8 | 2hr 9min

Section 4: Thread Safety and Reentrancy

4 / 6 | 1hr 10min

- [x] 23. Thread Safety and Reentrancy Overview
  16min
- [x] 24. Cross Thread Signals and Slots - Example1
  22min Resources
- [x] 25. Cross Thread Signals and Slots - Example2
  14min Resources
- [x] 26. Cross Thread Signals and Slots - Example3
  7min Resources
- [ ] 27. Slots in QThread Subclass
  10min Resources
- [ ] 28. Thread Safety and Reentrancy - Chapter Review
  1min

```
class Thread : public QThread
{
    Q_OBJECT
public:
    explicit Thread(QObject *parent = nullptr);

signals:

private slots:
    void increment();

private :
    QTimer * m_timer;
    int m_count;

    // QThread interface
protected:
    void run() override;
};
```

Thread Safety and Reentrancy

Chapter Review

Thread Safety

Reentrancy

Non Reentrancy

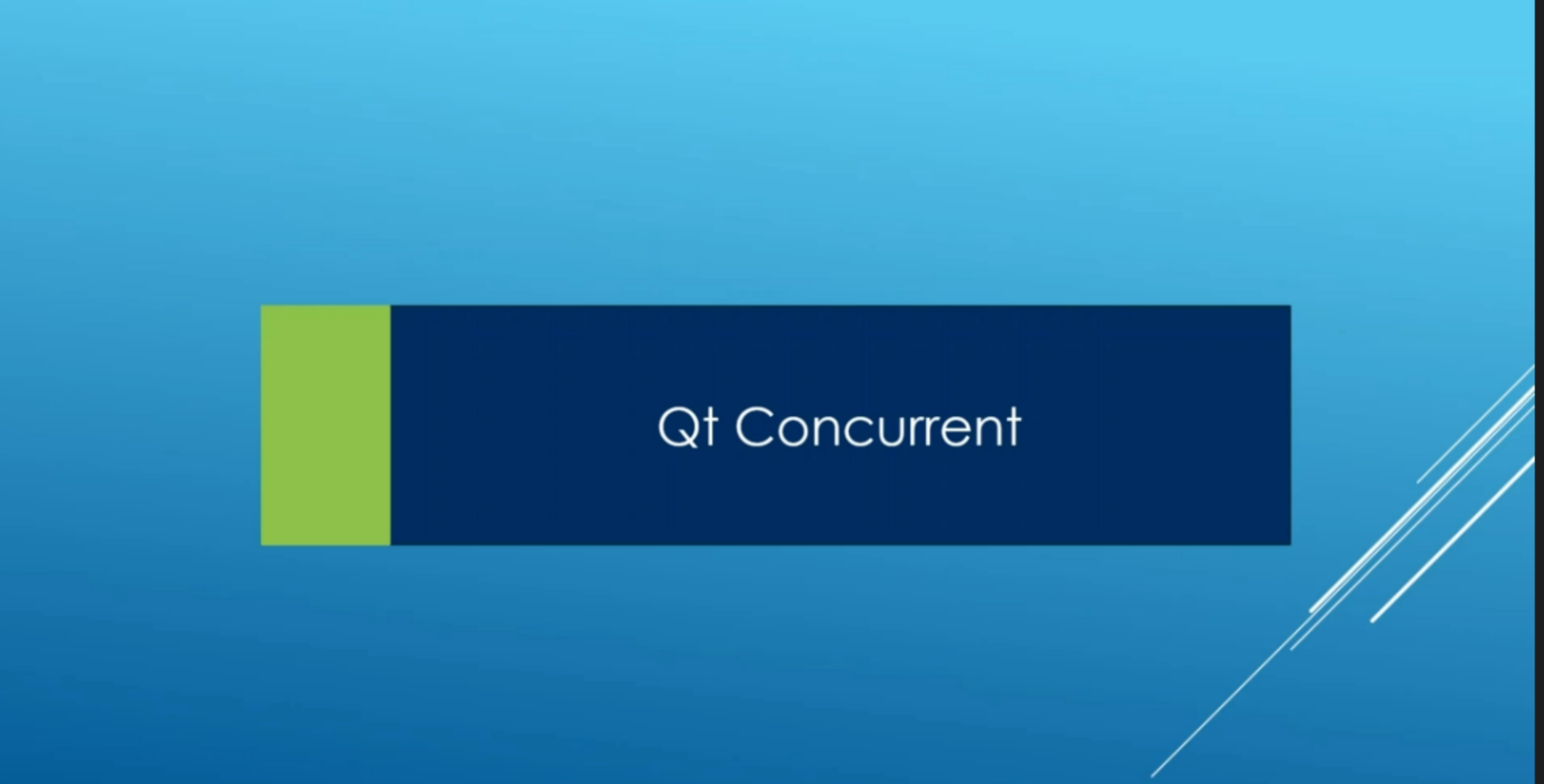
Thread Affinity

Get info from Qt Docs

Event Loop

Signals and Slots

Slots in Qthread Subclass



Course content

29. Qt Concurrent Overview
    6min
30. Qt Concurrent-run-synchronous
    14min Resources
31. Qt Concurrent Asynchronous - Return values
    23min Resources
32. Qt Concurrent-map
    13min Resources
33. Qt Concurrent-maped
    15min Resources
34. Qt Concurrent-mapReduced
    33min Resources
35. Qt Concurrent-Filter
    17min Resources
36. Qt Concurrent-Filtered
    9min Resources

The QtConcurrent namespace provides high-level APIs that make it possible to write multi-threaded programs without using low-level threading primitives such as mutexes, read-write locks, wait conditions, or semaphores. Programs written with QtConcurrent automatically adjust the number of threads used according to the number of processor cores available. This means that applications written today will continue to scale when deployed on multi-core systems in the future.

Header: #include <QtConcurrent>

qmake: QT += concurrent

Qt Creator

File Edit Build Debug Analyze Tools Window Help

Index | Qt Concurrent 5.12.3 | Unfiltered

Look for: Qt Concu

Qt Concurrent

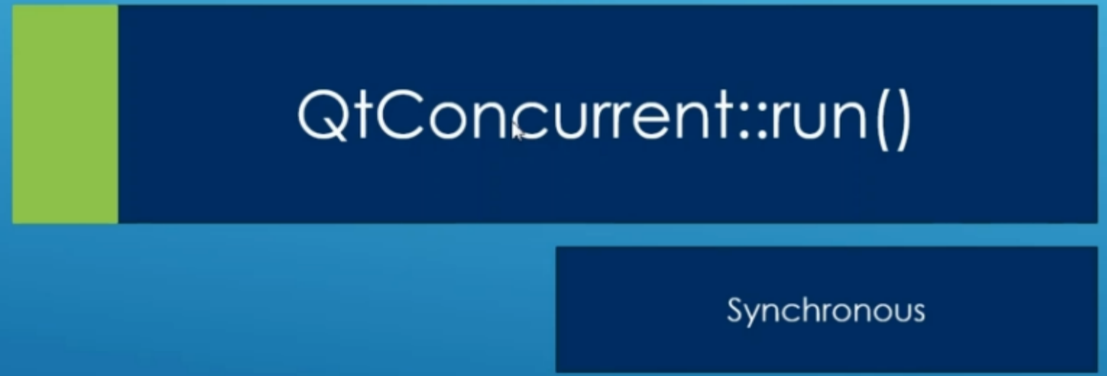
Qt Concurrent

Getting Started

The QtConcurrent namespace provides high-level APIs that make it possible to write multi-threaded programs without using low-level threading primitives such as mutexes, read-write locks, wait conditions, or semaphores. Programs written with QtConcurrent automatically adjust the number of threads used according to the number of processor cores available. This means that applications written today will continue to scale when deployed on multi-core systems in the future.

QtConcurrent includes functional programming style APIs for parallel list processing, including a MapReduce and FilterReduce implementation for shared-memory (non-distributed) systems, and classes for managing asynchronous computations in GUI applications:

- Concurrent Map and Map-Reduce
  - QtConcurrent::map() applies a function to every item in a container, modifying the items in-place.
  - QtConcurrent::mapped() is like map(), except that it returns a new container with the modifications.
  - QtConcurrent::mappedReduced() is like mapped(), except that the modified results are reduced or folded into a single result.
- Concurrent Filter and Filter-Reduce
  - QtConcurrent::filter() removes all items from a container based on the result of a filter function.
  - QtConcurrent::filtered() is like filter(), except that it returns a new container with the filtered results.
  - QtConcurrent::filteredReduced() is like filtered(), except that the filtered results are reduced or folded into a single result



Course content

- 30. Qt Concurrent-run-synchronous — 14min — Resources
- 31. Qt Concurrent Asynchronous - Return values — 23min — Resources
- 32. Qt Concurrent-map — 13min — Resources
- 33. Qt Concurrent-maped — 15min — Resources
- 34. Qt Concurrent-mapReduced — 33min — Resources
- 35. Qt Concurrent-Filter — 17min — Resources
- 36. Qt Concurrent-Filtered — 9min — Resources
- 37. Qt Concurrent-FilterReduce — 10min — Resources

Load Qt Concurrent Module

```
6
7 QT       += core gui
8 QT       += concurrent
9
```

Workload we need to run in secondary threads

```
void Widget::heavyWork()
{
    qDebug() << "Heavy work running in thread : " << QThread::currentThread();
    for(int i{0} ; i < 1000000001 ; i++){

        if((i%100000) == 0){
            double percentage = ((i/1000000000.0)) * 100;
            qDebug() << "Percentage : " << QVariant::fromValue(percentage).toInt()
                     << " | Thread : " << QThread::currentThread();


        }
    }
}
```

Off we go!

```
QtConcurrent::run(heavyWork);
```

Handle to a computation

QFuture

```
future = QtConcurrent::run(heavyWork);
```

Progress Monitoring

Synchronous

```
future = QtConcurrent::run(heavyWork);
future.waitForFinished();
qDebug() << "Done waiting for heavy work";
```

Detailed Description

The QFuture class represents the result of an asynchronous computation.

To start a computation, use one of the APIs in the Qt Concurrent framework.

QFuture allows threads to be synchronized against one or more results which will be ready at a later point in time. The result can be of any type that has a default constructor and a copy constructor. If a result is not available at the time of calling the result(), resultAt(), or results() functions, QFuture will wait until the result becomes available. You can use the isResultReadyAt() function to determine if a result is ready or not. For QFuture objects that report more than one result, the resultCount() function returns the number of continuous results. This means that it is always safe to iterate through the results from 0 to resultCount().

QFuture provides a Java-style iterator (QFutureIterator) and an STL-style iterator (QFuture::const_iterator). Using these iterators is another way to access results in the future.

QFuture also offers ways to interact with a runnning computation. For instance, the computation can be canceled with the cancel() function. To pause the computation, use the setPaused() function or one of the pause(), resume(), or togglePaused() convenience functions. Be aware that not all asynchronous computations can be canceled or paused. For example, the future returned by QtConcurrent::run() cannot be canceled; but the future returned by QtConcurrent::mappedReduced() can.

Progress information is provided by the progressValue(), progressMinimum(), progressMaximum(), and progressText() functions. The waitForFinished() function causes the calling thread to block and wait for the computation to finish, ensuring that all results are available.

The state of the computation represented by a QFuture can be queried using the isCanceled(), isStarted(), isFinished(), isRunning(), or isPaused() functions.

Detailed Description

The QFuture class represents the result of an asynchronous computation.

To start a computation, use one of the APIs in the Qt Concurrent framework.

QFuture allows threads to be synchronized against one or more results which will be ready at a later point in time. The result can be of any type that has a default constructor and a copy constructor. If a result is not available at the time of calling the result(), resultAt(), or results() functions, QFuture will wait until the result becomes available. You can use the isResultReadyAt() function to determine if a result is ready or not. For QFuture objects that report more than one result, the resultCount() function returns the number of continuous results. This means that it is always safe to iterate through the results from 0 to resultCount().

QFuture provides a Java-style iterator (QFutureIterator) and an STL-style iterator (QFuture::const_iterator). Using these iterators is another way to access results in the future.

QFuture also offers ways to interact with a runnning computation. For instance, the computation can be canceled with the cancel() function. To pause the computation, use the setPaused() function or one of the pause(), resume(), or togglePaused() convenience functions. Be aware that not all asynchronous computations can be canceled or paused. For example, the future returned by QtConcurrent::run() cannot be canceled; but the future returned by QtConcurrent::mappedReduced() can.

Progress information is provided by the progressValue(), progressMinimum(), progressMaximum(), and progressText() functions. The waitForFinished() function causes the calling thread to block and wait for the computation to finish, ensuring that all results are available.

The state of the computation represented by a QFuture can be queried using the isCanceled(),

Progress Monitoring

Asynchronous

```
QFutureWatcher<void> watcherNoReturn;
```

```
future = QtConcurrent::run(heavyWork);

//Be notified when work is done asynchronously through signals and slots.
watcherNoReturn.setFuture(future);
```

```
connect(&watcherNoReturn,&QFutureWatcher<void>::finished,[=](){
    qDebug() << "asynchronous : heavy work done.";
});
```

- 9 protected functions inherited from QObject

Detailed Description

The QFutureWatcher class allows monitoring a QFuture using signals and slots.

QFutureWatcher provides information and notifications about a QFuture. Use the setFuture() function to start watching a particular QFuture. The future() function returns the future set with setFuture().

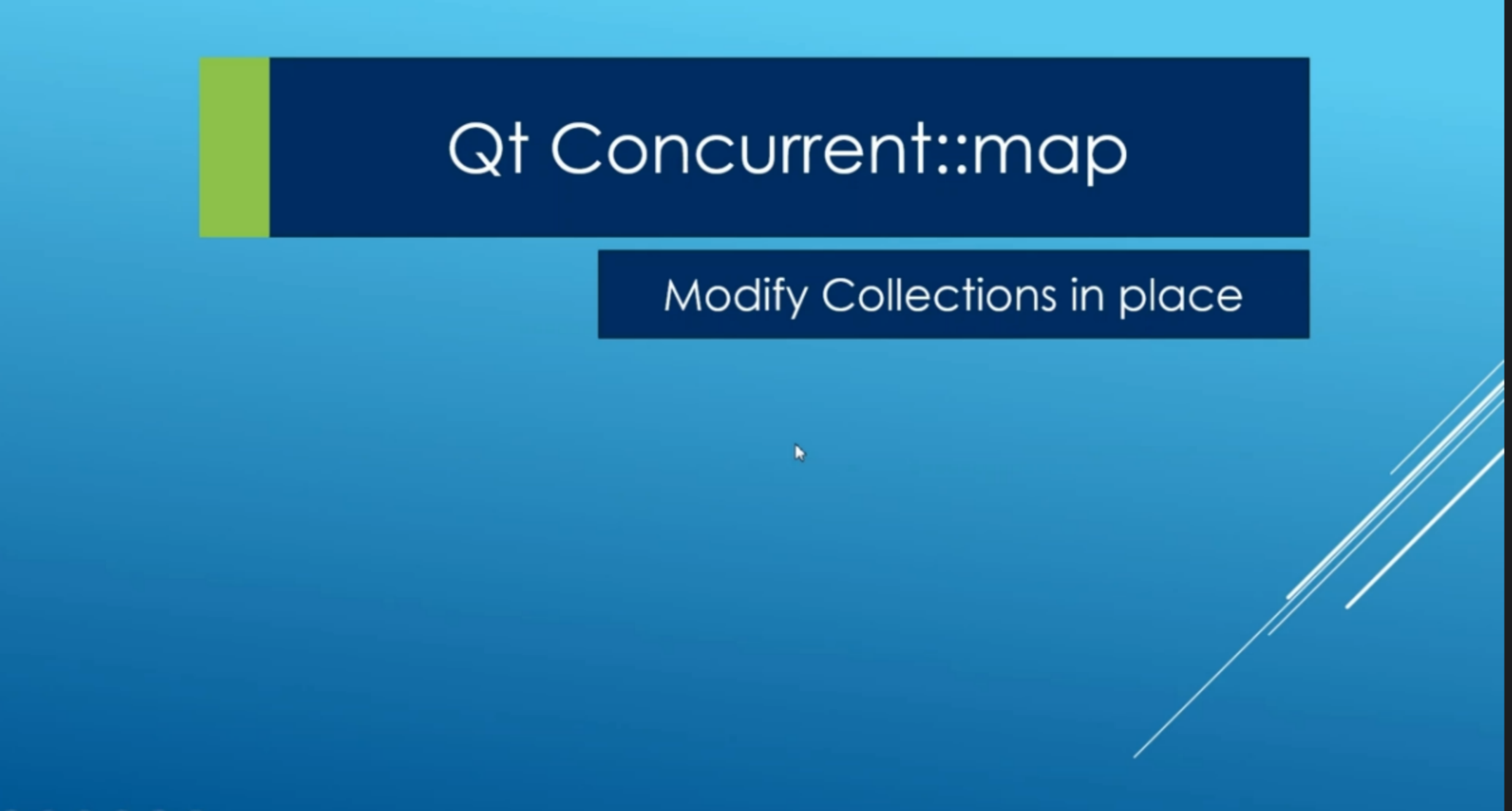
For convenience, several of QFuture's functions are also available in QFutureWatcher: progressValue(), progressMinimum(), progressMaximum(), progressText(), isStarted(), isFinished(), isRunning(), isCanceled(), isPaused(), waitForFinished(), result(), and resultAt(). The cancel(), setPaused(), pause(), resume(), and togglePaused() functions are slots in QFutureWatcher.

Status changes are reported via the started(), finished(), canceled(), paused(), resumed(), resultReadyAt(), and resultsReadyAt() signals. Progress information is provided from the progressRangeChanged(), void progressValueChanged(), and progressTextChanged() signals.

Throttling control is provided by the setPendingResultsLimit() function. When the number of pending resultReadyAt() or resultsReadyAt() signals exceeds the limit, the computation represented by the future will be throttled automatically. The computation will resume once the number of pending signals drops below the limit.

Example: Starting a computation and getting a slot callback when it's finished:

```
// Instantiate the objects and connect to the finished signal.
MyClass myObject;
QFutureWatcher<int> watcher;
connect(&watcher, SIGNAL(finished()), &myObject, SLOT(handleFinished()));

// Start the computation.
```



Course content

- 32. Qt Concurrent-map — 13min — Resources
- 33. Qt Concurrent-maped — 15min — Resources
- 34. Qt Concurrent-mapReduced — 33min — Resources
- 35. Qt Concurrent-Filter — 17min — Resources
- 36. Qt Concurrent-Filtered — 9min — Resources
- 37. Qt Concurrent-FilterReduce — 10min — Resources
- 38. Qt Concurrent-QFutureSynchronizer — 7min — Resources
- 39. Qt Concurrent : Feedback — 4min
- 40. Threading Overview-Comparison

Modify collection data in place (large numbers of items)

```
qDebug() << "Populating list with data . GUI Thread : " << QThread::currentThread();
//Populate list
for( int i {0} ;  i < 30000000 ; i++){
    list << i;
}

qDebug() << "Original value : " << list.last();

connect(&futureWatcher, &QFutureWatcher<void>::started,[=](){
    qDebug() << "asynchronous : Work started.";
});

connect(&futureWatcher, &QFutureWatcher<void>::finished,[=](){
    qDebug() << "asynchronous : Work done.";
});

connect(&futureWatcher, &QFutureWatcher<void>::progressValueChanged,[=](int value){
    qDebug() << "Progress : " << value;
});
```

Modify collection data in place (large numbers of items)

```
void Widget::on_modifyButton_clicked()
{
    //Define function

    auto modify = [](int &value){
        qDebug() << "Modifying " << value << " result : " << value * 10
                 << " Thread :" << QThread::currentThread();
        value = value * 10;
    };

   future = QtConcurrent::map(list,modify);
   futureWatcher.setFuture(future);
}
```

Edit Build Debug Analyze Tools Window Help

Projects

- 5-2ConcurrentRunSync
- 5-3ConcurrentRunAsyc
- **5-4ConcurrentMap**
  - 5-4ConcurrentMap.pro
  - Headers
    - widget.h
  - Sources
    - main.cpp
    - widget.cpp
  - Forms
    - widget.ui

Open Documents

- 5-3ConcurrentRunAsyc.pro
- 5-4ConcurrentMap.pro
- widget.cpp
- widget.h
- widget.ui

5-4ConcurrentMap.pro — Line: 8, Col: 23

```
#-------------------------------------------------
#
# Project created by QtCreator 2019-10-29T06:27:02
#
#-------------------------------------------------

QT       += core gui
QT       += concurrent

greaterThan(QT_MAJOR_VERSION, 4): QT += widgets

TARGET = 5-4ConcurrentMap
TEMPLATE = app

# The following define makes your compiler emit warnings if you use
# any feature of Qt which has been marked as deprecated (the exact warnings
# depend on your compiler). Please consult the documentation of the
# deprecated API in order to know how to port your code away from it.
DEFINES += QT_DEPRECATED_WARNINGS

# You can also make your code fail to compile if you use deprecated APIs.
# In order to do so, uncomment the following line.
# You can also select to disable deprecated APIs only up to a certain version of Qt.
#DEFINES += QT_DISABLE_DEPRECATED_BEFORE=0x060000    # disables all the APIs deprecated before Qt

CONFIG += c++11

SOURCES += \
        main.cpp \
        widget.cpp
```

Course content

- [x] 32. Qt Concurrent-map
  13min — Resources
- [ ] 33. Qt Concurrent-maped
  15min — Resources
- [ ] 34. Qt Concurrent-mapReduced
  33min — Resources
- [ ] 35. Qt Concurrent-Filter
  17min — Resources
- [ ] 36. Qt Concurrent-Filtered
  9min — Resources
- [ ] 37. Qt Concurrent-FilterReduce
  10min — Resources
- [ ] 38. Qt Concurrent-QFutureSynchronizer
  7min — Resources
- [ ] 39. Qt Concurrent : Feedback
  4min

Return new modified collection

```
void Widget::on_modifyButton_clicked()
{
    std::function<int(const int&)> modify = [](const int & value) -> int{
        qDebug() << "Modifying " << value << " result : " << value * 10
                 << " Thread :" << QThread::currentThread();
        return value * 10;
    };

    future =  QtConcurrent::mapped(list,modify);
    futureWatcher.setFuture(future);

}
```

Get notified asynchronously in a slot

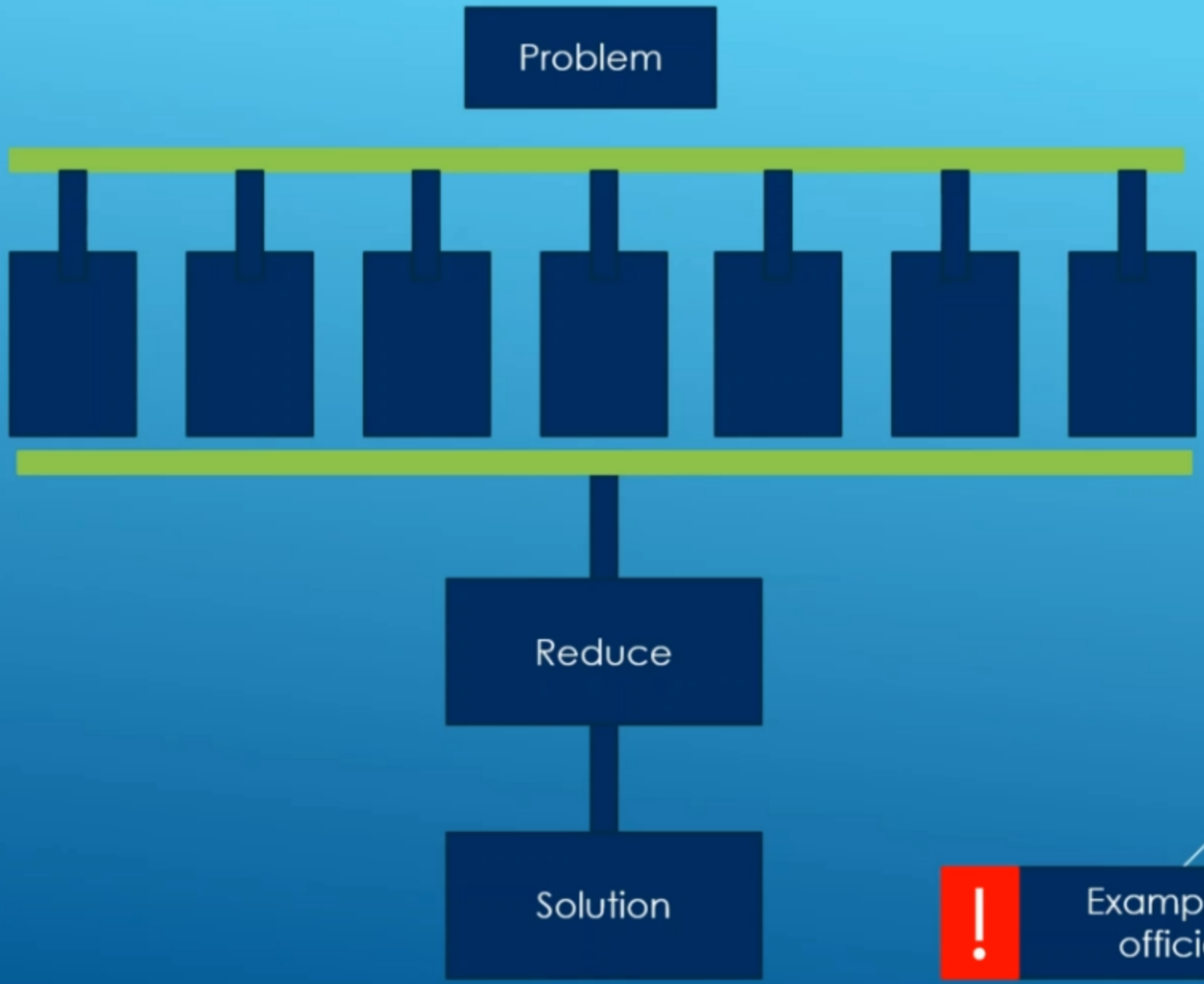
```
connect(&futureWatcher, &QFutureWatcher<void>::finished,[=](){

    qDebug() << "asynchronous : Work done.";

    qDebug() << "Result count : " << future.resultCount();

    qDebug() << future.results();

    for( int i{0} ; i < future.resultCount(); i ++){
        qDebug() << "Result " << i << " :" << future.resultAt(i);
    }

});
```

Go synchronous

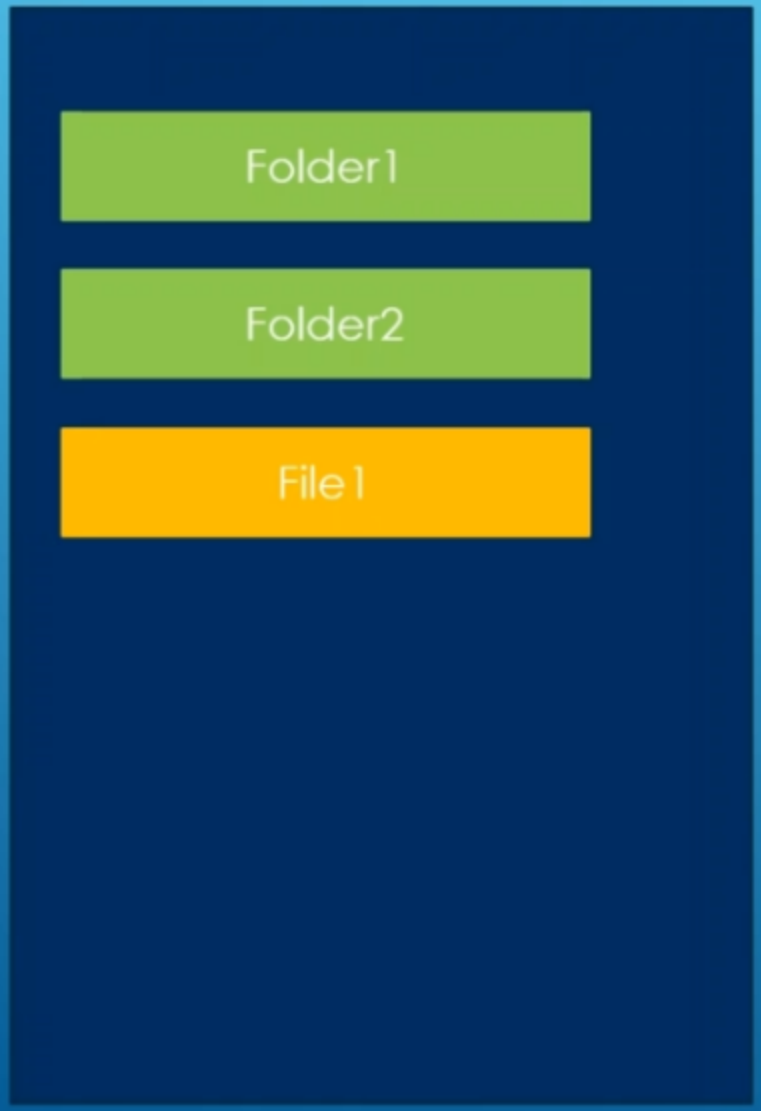
```
std::function<int(const int&)> modify = [](const int & value) -> int{
    qDebug() << "Modifying " << value << " result : " << value * 10
             << " Thread :" << QThread::currentThread();
    return value * 10;
};

future =  QtConcurrent::mapped(list,modify);
future.waitForFinished();
qDebug() << "Result count : " << future.resultCount();
qDebug() << future.results();



for( int i{0} ; i < future.resultCount(); i ++){
    qDebug() << "Result " << i << " :" << future.resultAt(i);
}
```

Course content

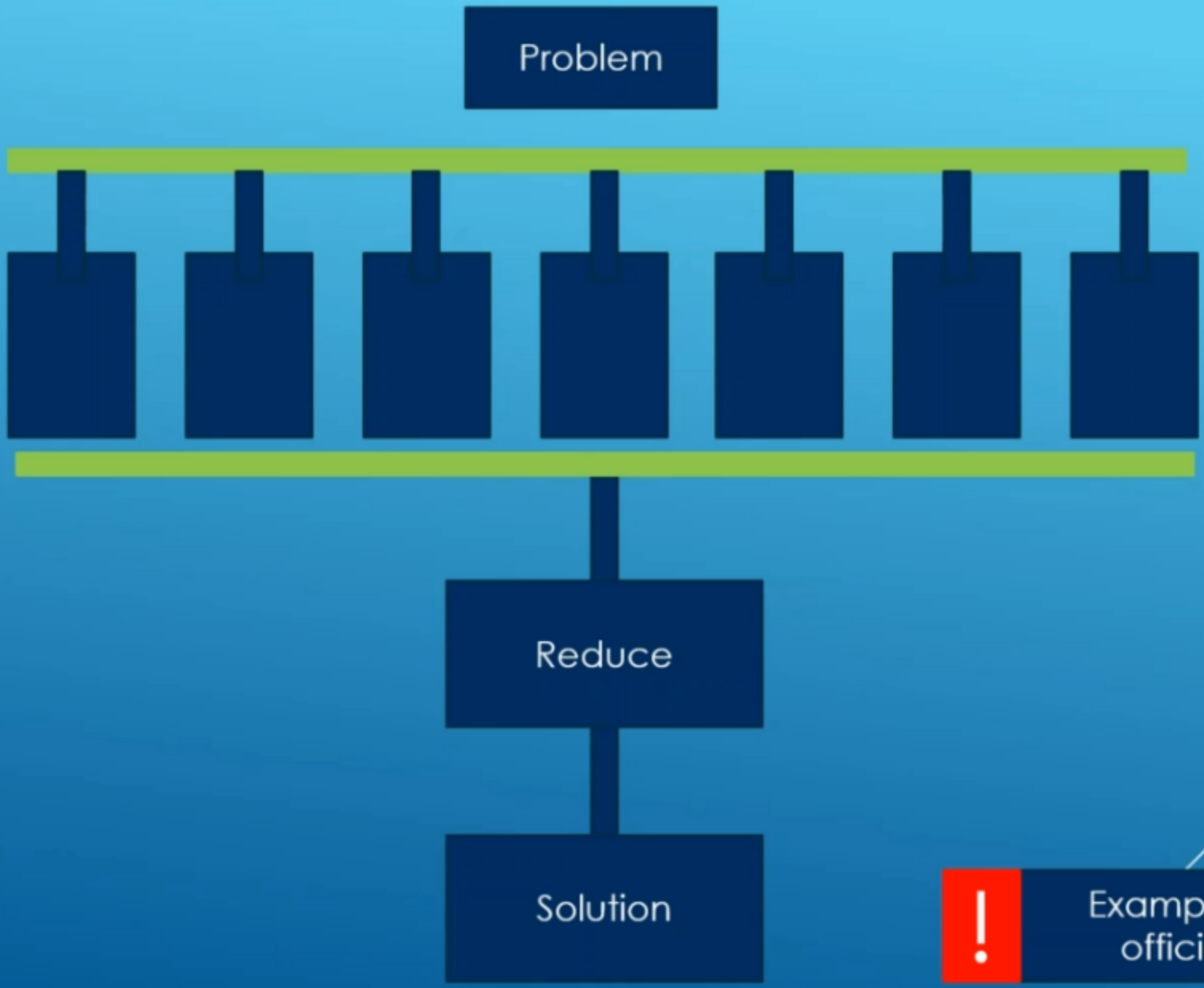
- [x] 32. Qt Concurrent-map
  13min — Resources
- [x] 33. Qt Concurrent-maped
  15min — Resources
- [ ] 34. Qt Concurrent-mapReduced
  33min — Resources
- [ ] 35. Qt Concurrent-Filter
  17min — Resources
- [ ] 36. Qt Concurrent-Filtered
  9min — Resources
- [ ] 37. Qt Concurrent-FilterReduce
  10min — Resources
- [ ] 38. Qt Concurrent-QFutureSynchronizer
  7min — Resources
- [ ] 39. Qt Concurrent : Feedback
  4min



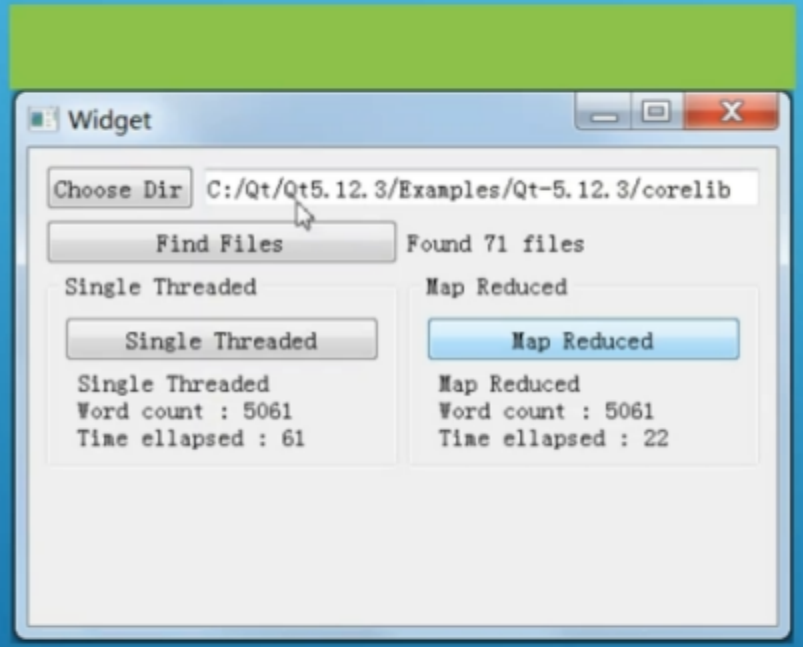
Example adapted from official Qt Examples



word	frequency
while	233
for	34
if	50
join	11



! Example adapted from official Qt Examples


Widget
Choose Dir
C:/Qt/Qt5.12.3/Examples/Qt-5.12.3/corelib
Find Files
Found 71 files
Single Threaded
Map Reduced
Single Threaded
Map Reduced
Single Threaded
Word count : 5061
Time ellapsed : 61
Map Reduced
Word count : 5061
Time ellapsed : 22

Single Threaded

```
QMap<QString, int> Widget::singleThreadedWordCount(const QStringList &files)
{
    QMap<QString, int> wordCount;
    for (const QString &file : files) {
        QFile f(file);
        f.open(QIODevice::ReadOnly);
        QTextStream textStream(&f);
        while (!textStream.atEnd()) {
            const auto words =  textStream.readLine().split(' ');
            for (const QString &word : words)
                wordCount[word] += 1;
        }
    }
    return wordCount;
}
```

Map	Reduce
Count words in each file, store results in map	Put together results from all files

Map

```
QMap<QString, int> Widget::countWords(const QString &file)
{
    QFile f(file);
    f.open(QIODevice::ReadOnly);
    QTextStream textStream(&f);
    QMap<QString, int> wordCount;

    while (!textStream.atEnd()) {
        const auto words =  textStream.readLine().split(' ');
        for (const QString &word : words)
            wordCount[word] += 1;
    }

    return wordCount;
}
```

Reduce

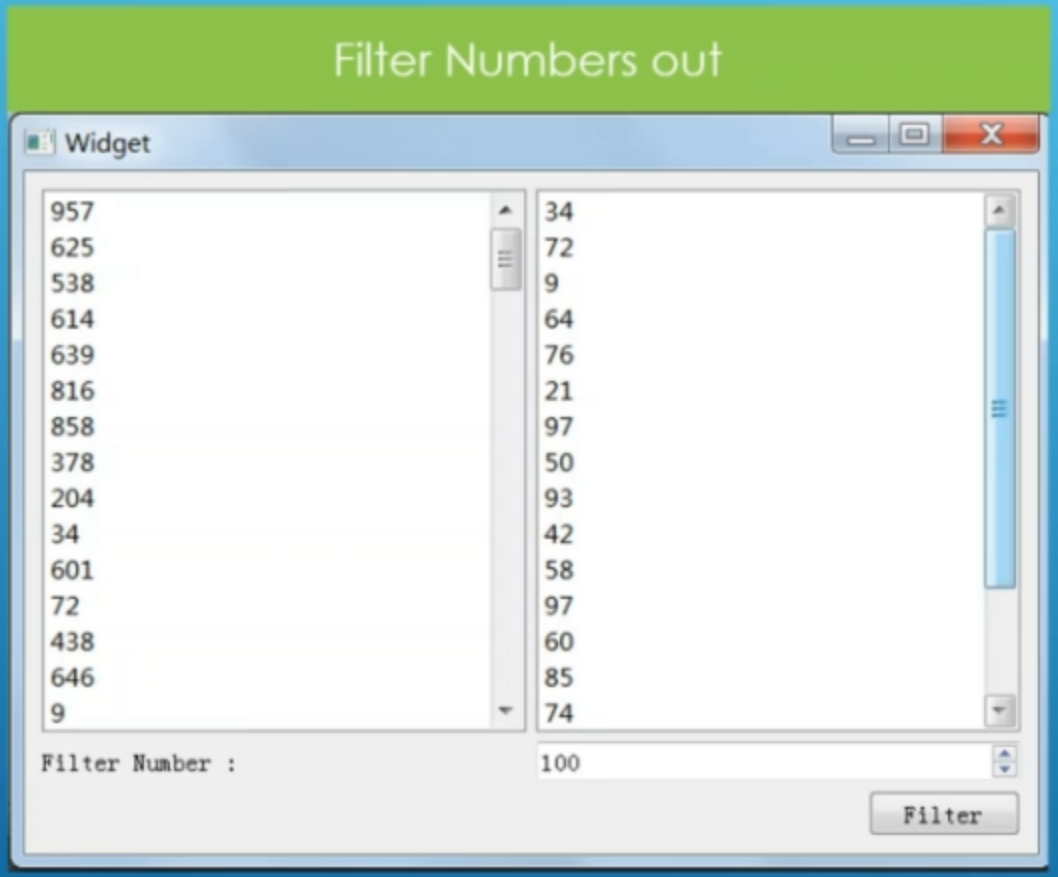
```
void Widget::reduce(QMap<QString, int> &result, const QMap<QString, int> &w)
{
    QMapIterator<QString, int> i(w);
    while (i.hasNext()) {
        i.next();
        result[i.key()] += i.value();
    }
}
```

QtConcurrent::mapReduced()

```
void Widget::on_mapReducedButton_clicked()
{

    QTime time;
    time.start();

    future = QtConcurrent::mappedReduced(files, countWords, reduce);
    future.waitForFinished();

    int timeEllapsed = time.elapsed();

    QMap<QString, int> total = future.result();

    QString message = " Map Reduced \n Word count : "+
            QString::number(total.keys().count()) +
            " \n Time ellapsed : " + QString::number(timeEllapsed);
    ui->mapReducedTimeLabel->setText(message);

}
```

Qt Concurrent::filter

Filter Numbers out



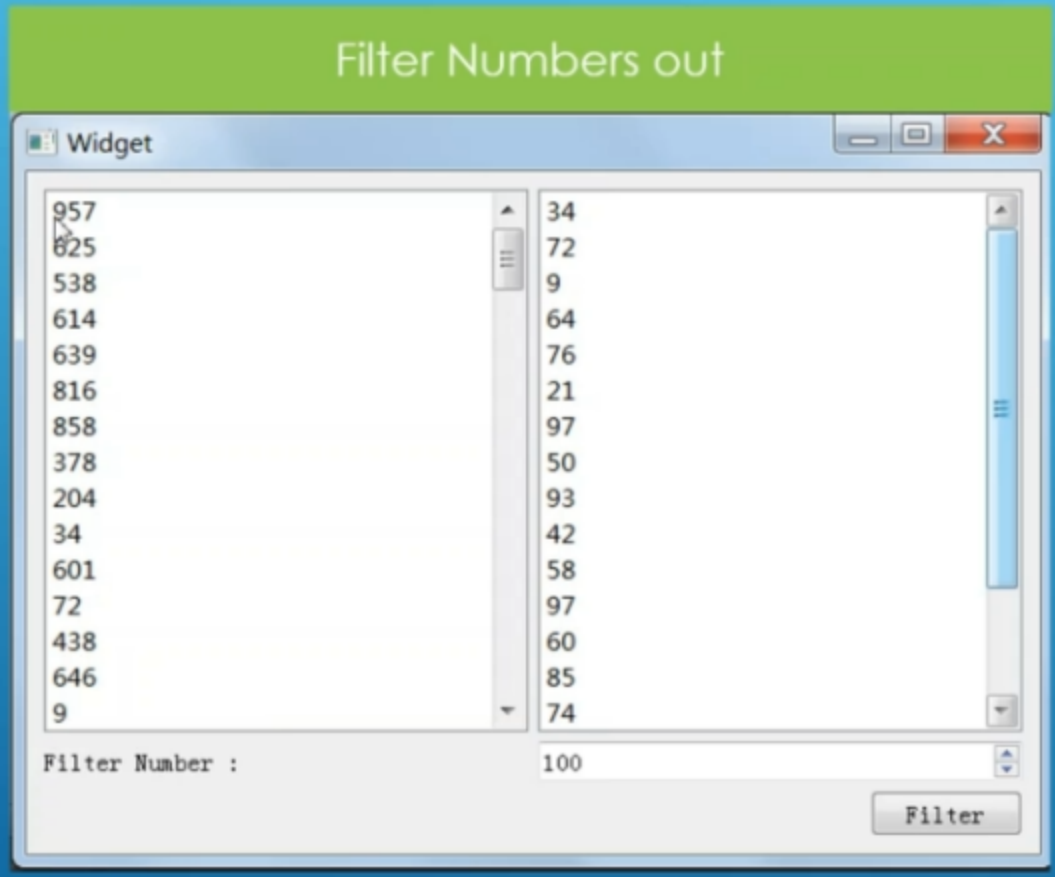
Filter

```
void Widget::on_filterButton_clicked()
{
    auto filter = [=](const int value){

        if(value >= filterValue){
            return false;
        }
        return true;
    };
      //This is for QtConcurrent::filter()
    ui->filteredList->clear();

    QFuture<void> future =QtConcurrent::filter(intList,filter);
    future.waitForFinished();

    qDebug() << "After , item count : " << intList.count();
    foreach (int value, intList) {
        ui->filteredList->addItem(QString::number(value));
    }
}
```

Course content

- [x] 32. Qt Concurrent-map
  13min — Resources
- [x] 33. Qt Concurrent-maped
  15min — Resources
- [x] 34. Qt Concurrent-mapReduced
  33min — Resources
- [x] 35. Qt Concurrent-Filter
  17min — Resources
- [ ] 36. Qt Concurrent-Filtered
  9min — Resources
- [ ] 37. Qt Concurrent-FilterReduce
  10min — Resources
- [ ] 38. Qt Concurrent-QFutureSynchronizer
  7min — Resources
- [ ] 39. Qt Concurrent : Feedback
  4min

Filter Numbers out



Filtered

```
void Widget::on_filterButton_clicked()
{
    auto filter = [=](const int value){

        if(value >= filterValue){

            return false;
        }
        return true;
    };
    ui->filteredList->clear();
    future =QtConcurrent::filtered(intList,filter);

    future.waitForFinished();

    QList<int> newList = future.results();
    qDebug() << "After , item count : " << newList.count();
    foreach (int value, newList) {
        ui->filteredList->addItem(QString::number(value));
    }
}
```