Threads and terminology

- A reentrant method is one that can be called simultaneously by multiple threads, provided no two invocations of the method attempt to reference the same data.

?

```
static int sum = 0;

int increment(int i) {
  sum += i;
  return sum;
}
```

?

```
int increment(int sum, int i) {
    return sum + i;
}
```

Threads and terminology

- A reentrant method is one that can be called simultaneously by multiple threads, provided no two invocations of the method attempt to reference the same data.

Non-Reentrant

```
static int sum = 0;

int increment(int i) {
  sum += i;
  return sum;
}
```

Reentrant

```
int increment(int sum, int i) {
    return sum + i;
}
```

Threads and terminology

- A thread-safe method can be called simultaneously by multiple threads at any time, because any shared data is protected somehow (e.g., by a mutex) from simultaneous accesses.
- A thread-safe object is one that can be accessed concurrently by multiple threads and is guaranteed to always be in a "valid" state.

QT Processes

- QProcess is the class for starting and controlling other processes.
- A QProcess can launch another process using the *start()* function.
  - The new process is a child process that terminates when the parent process does.

Example of QProcess

```
LogTail::LogTail(QString fn) {
    ...
    QStringList argv;

    argv << "-f" << fn;
    start("tail", argv);
}
```

Qt Thread (or QThread)

- Qt's thread model permits the prioritizing and control of threads.
- One important class in this namespace is QThreadPool, a class that manages a pool of threads.
  - Every Qt application has a QThreadPool with a suggested maximum thread count that defaults, on most systems, to the number of cores.
- Qt uses a Message Queue to pass signals between threads.

How Qthread works

- Qthread start() method creates a new thread, which will execute the run() method and ends when it finishes

```
class MyThread : public QThread {
    Q_OBJECT
    protected: void run();
};

void MyThread::run() { ... }
```

Synchronizing Threads

- In Qt, the following mechanisms are available:
  - Qmutex
  - QReadWriteLock
  - QSemaphore
  - QWaitCondition

QMutex

- Provides a Mutual exclusive lock for shared data
- Methods:
  - lock
  - tryLock(int timeout)
    - Attempts to lock the mutex. This function returns true if the lock was obtained; otherwise it returns false. If another thread has locked the mutex, this function will wait for at most timeout milliseconds for the mutex to become available.
  - unlock
- If the mutex is already locked, any other thread trying to lock it will sleep until it is unlocked
- When the mutex is unlocked, a single thread that is blocked on lock will be released and start its work

QReadWriteLock

- Works a bit like QMutex but differentiates read and write operations
- Allows multiple readers and a single writer
  - *"readers attempting to obtain a lock will not succeed if there is a blocked writer waiting for access, even if the lock is currently only accessed by other readers. Also, if the lock is accessed by a writer and another writer comes in, that writer will have priority over any readers that might also be waiting."*
- Methods:
  - lockForRead
  - lockForWrite
  - try...
  - unlock

QSemaphore

- A general purpose counting semaphore
- Methods:
  - acquire(int n=1)
  - release(int n=1)
  - int available()

QWaitCondition

- A general purpose signal/wait mechanism
  - Basically a conditional variable
- Allows a thread to tell another thread that a condition has been met, and it is now safe to resume work
- Perfect for Producer/Consumer
- Methods:
  - wait
  - wakeAll()
  - wakeOne()

Example of QWaitCondition

- *keyPressed* variable is a global variable of type QWaitCondition.

Thread 1

```
…
keyPressed.wait(&mutex);
    do_something();
…
…
```

Thread 2

```
…
    getchar();
keyPressed.wakeAll();
…
```

A common design pattern to avoid direct use of threads

- An event-loop is a well-known design pattern to produce multi-threaded code without managing threads explicitly

```
while (is_active)
{
    while (!event_queue_is_empty)
        dispatch_next_event();

}
```

Event Loop instead of threading

- A dispatcher is in charge of activating a new thread (usually from a pool) once a new event comes
  - Event can be handled blocking or non-blocking
- An event may be produced as a consequence of:
  - Timer expiration (Qtimer)
  - User-defined signal that identifies the arrival of a new even

Even loop Scheme