Thread pool goals

Creating a thread is a non-trivial task at the OS level.

Rather than running one task and dying, a thread can execute multiple `Runnable` tasks sequentially, and those tasks can be pulled from a queue structure (typically a `BlockingQueue`)

If a number of worker threads are attached to a single `BlockingQueue`, the result is known as a ***thread-pool***.

Thread pools are provided by the Java APIs, implementing the interfaces `Executor`, and/or `ExecutorService`.

Using `ExecutorService`

Create in various ways, commonly using factories in the `Executors` class

Send work to the pool using either `submit` or `execute` methods

The `submit` method accepts an instance of `Callable<E>`, similar to `Runnable`, but returns a value (of type `E`), and can throw checked exceptions. (Can also submit a `Runnable`)

The `submit` method returns a `Future<E>`, which is a handle on the submitted job.

Using `Future<E>`

The `isDone` method polls to see if the submitted task has completed

The `isCancelled` method polls to see if the submitted task was canceled

The `get` methods (an overload provides a timeout) block until the job is completed, and returns the result of that job, or throws an `ExecutionException` if the job threw an exception

> `ExecutionException` has the job's exception embedded as the "cause".

The `cancel` method attempts to cancel the job, this might remove it from the input queue, or if it has started can send an interrupt to ask the job to shutdown (controlled by a boolean argument).

Thread shutdown

Threads should not be killed from outside, they might hold locks, or have data or devices in transactionally unsafe states.

Instead, they should be sent a message requesting the thread clean up and shut itself down.

The conventional notification is to send an interrupt; interrupts should not be used for other purposes.

Library code should clean up that method call, and perhaps that library, but rethrow the InterruptedException to the caller, so that the business logic can shut itself down cleanly too.

`ReentrantLock`

The `synchronized`/`wait`/`notify` mechanism has several significant issues:

The only way forward from a `synchronized` call is successfully obtaining the lock. This can cause problems shutting down threads.

Similarly, there is no timeout on a call to `synchronized`

There is only one "condition variable" that a thread can block (wait) on, this is rarely enough to model any real situation.

Using `notifyAll` to solve the previous issue impedes scalability.

The `ReentrantLock` API addresses these problems.

Using `ReentrantLock`

Locks must be released reliably in all situations. Use a `try`/`finally` structure to achieve this:

```
lock.lock();
try {
    // …
} finally {
    lock.unlock();
}
```

This ensures lock release even in the face of unhandled exceptions or premature return.

Timing with `ReentrantLock`

Create one or more rendezvous objects using `aLock.newCondition()`

While holding the lock call `aCondition.signal()` or `aCondition.await()`

Behavior is parallel to `notify()` and `wait()` of `Object`

`StampedLock`

StampedLock allows us to build access control based on three modes

- Exclusive (write) lock
- Non exclusive (read) lock
- Optimistic lock

Lock using: `[write|read]Lock()`, `[write|read]LockInterruptibly()`, `try[Write|Read|OptimisticRead]Lock()`. These return a `long` value identifying the lock obtained, or zero if the attempt failed.

Release the lock using `unlock[Write|Read](lock)`, verify optimistic lock using `validate(lock)`

Semaphore

`Semaphore` is a classic mutual exclusion construct. It often serves better as a resource counter.

Can be tricky to use for mutual exclusion as it is not reentrant.

`sem.acquire()` attempts to decrement the counter of the semaphore, if the counter would reduce below zero, then the thread is blocked until another thread causes a sufficient increase in the count.

`sem.release()` increments the counter of the semaphore, possibly releasing one or more threads blocked on acquire calls.

`try`/`finally` constructs can be used to ensure `release()` is called reliably.