Guide to threading examples

Three ways of starting a thread
(in directory: *1 Starting a thread*)

- We can derive from the Thread class: MyThread.java
- We can implement the Runnable interface and pass our new object to an object of class Thread: MyThread2.java
- We can create an instance of a Runnable interface on the fly by implementing its run method and passing the new object to an object of the class Thread: MyThread3.java

Stopping a thread
(in directory: *2 Stopping a thread*)

- Example
  - StoppingMyThread.java
- In the above, there is a flag, *done*, that is set by a public method
- The thread is executing a loop and, in that loop, is every so often checking for the state of the *done* flag.
- That loop should not be too tight. In other words, it's a bad idea to continuously check for the state of the *done* flag. It should be checked at intervals, else the amount of processing power that will be wasted on this activity will be too great.

Pausing a thread
(in directory: *3 Pausing a thread*)

- A *MainThread* object starts another thread in which it runs an object of class *OtherThread*.
- The *MainThread* object puts the *OtherThread* object into a wait state by setting a flag. When *OtherThread* detects this flag, it calls the *wait* method on its thread. This puts it into a dormant state.
- *MainThread* does some work then calls *OtherThread*'s *notify* method to wake it up.
- Both classes send messages to the console to indicate their current state making it possible to observe the sequence of messages.
- *wait/notify* is the right way to implement loops in which one thread waits for another thread to give it something to do. A blocking queue is another way we can implement the same functionality.

Waiting for thread to finish
(in directory: *4 Pausing a thread*)

- *WaitingForThreadToFinish.java* shows how one thread can use the *join* method to wait for another thread to finish

Failing to synchronize
(in directory: *5 Synchronization*)

- *SynchronizationError.java* shows a typical synchronization error. When two threads can access the same object, it may lead to unpredictable results:
  - Suppose one thread is running the *adjust* method
  - The value for variable *size* is retrieved and 1 is added to it
  - Before the resulting value can be put back into variable *size*, this thread is interrupted by a second thread
  - That thread calls reset and *size* is set to zero
  - It is then interrupted by the first thread, and the value previously computed by adding 1 to *size* is put back into size
  - It's as if the call to reset never took place!

Synchronization
(in directory: *5 Synchronization*)

- SynchronizationError_Fixed_Big.java shows how to prevent an error similar to the one shown in SynchronizationError.java
  - By using the *synchronize* block with *this*, we prevent two or more threads from accessing the blocked code at the same time. One thread has access to the block while others must wait for it to complete.
  - This may be overkill. In other words, there may be no reason to stop threads from accessing methods that cannot create a synchronization error of the type shown.
- *SynchronizationError_Fixed_Methods.java* shows how to prevent the above errors by using the *synchronize* keyword with methods that may interfere with each other. If, in this case, we had methods that could not interfere with each other, they would not require the *synchronize* keyword, and could be accessed by multiple threads.
- SynchronizationError_Fixed_Small.java shows how to use the *synchronize* keyword on individual variables rather than on the object as a whole, a much more granular solution. This allows some blocks to synchronize on one variable while other blocks synchronize on another.

Synchronization
(in directory: *5 Synchronization*)

- SynchronizationError_Fixed_Big.java shows how to prevent an error similar to the one shown in SynchronizationError.java
  - By using the *synchronize* block with *this*, we prevent two or more threads from accessing the blocked code at the same time. One thread has access to the block while others must wait for it to complete.
  - This may be overkill. In other words, there may be no reason to stop threads from accessing methods that cannot create a synchronization error of the type shown.
- *SynchronizationError_Fixed_Methods.java* shows how to prevent the above errors by using the *synchronize* keyword with methods that may interfere with each other. If, in this case, we had methods that could not interfere with each other, they would not require the *synchronize* keyword, and could be accessed by multiple threads.
- SynchronizationError_Fixed_Small.java shows how to use the *synchronize* keyword on individual variables rather than on the object as a whole, a much more granular solution. This allows some blocks to synchronize on one variable while other blocks synchronize on another.

Locks

- Locks are another synchronization tool and may be found in the Java concurrent tools libary. A good explanation of locks can be found here:
  - http://tutorials.jenkov.com/java-concurrency/locks.html
- Locks are used as follows:

  -
    ```
    lock.lock();
    int newCount = ++count;
    lock.unlock();
    return newCount;
    ```
- The main differences between a lock and a synchronized block are as follows.
  - The condition that triggers a call to *lock.unlock()* does not have to reside in the same method as the condition that triggers a call to *lock.lock(),* making it possible for external events to cause an unlock.
  - A lock may have a tryLock() or tryLock(timeOut) methods that fail to obtain a lock, either immediately or after some timeout. This can be very handy for choosing a course of action that is more reasonable than waiting for a long synchronized process to complete.

Thread-safe data structures

- The java concurrent library has a set of collection classes that were specifically designed to be thread safe. For example:
  - ConcurrentHashMap – a thread safe version of a hash map
  - ConcurrentLinkedQueue – a thread safe version of a linked list
- Objects of these classes can be accessed by multiple threads without concern that the internal state may be compromised. Hence, it is not necessary to use synchronize with these objects.
- Also part of this library are the blocking queues
  - LinkedBlockingDequeue
  - LinkedBlockingQueue
- When a variable is assigned the return value of the *consume* method of a blocking queue, it waits – without looping – until that queue has something it can return. This is an efficient way to implement communication between objects in your system without using *wait/notify*.
- Note: If we want to shut down an object that is listening to a queue, we can put a *null* on that queue after having set the object's *done* flag.