~~Quick~~ semi-quick crash course on thread closing in Java

Threads in Java (or any language really) should NOT be stopped forcibly outside the thread logic. The reasoning is how would an outside process know at which point to safely stop a thread? This is also why the stop() method is deprecated, because it does just that. These unsafe methods do not care about current thread state. Therefore, the coder needs to make sure that the thread terminates by it's own logic. How can we do this?

One way for the coder to signal to a thread that it needs to do something is by setting flags. Your thread checks the state of a flag continuously, and branches based on the flag. So if the flag is to signal termination of the thread, then setting the flag (to something that means terminate) allows the thread to read the flag at some point and do anything necessary to terminate gracefully. This technique can be generalized to provide any change of state request to a thread.

Does the flag need to be volatile? First of all, what does that mean? A variable declared volatile means that all writes are done straight to memory (no thread-local caching). Volatile variables are also (sorta) synchronized, in that reads/writes do not allow concurrent access, but no actual synchronization using locks takes place. In the case of flags, do they need to be volatile? Maybe or maybe not. Think about this. You have one thread (say the main thread) setting the value for a non-volatile flag. A thread that uses this flag happens to read it before the new value is committed to memory. This can cause the thread to miss the update, but in most cases that's not too important as it will read the new flag state on the next flag read iteration. Where volatile becomes more important is if multiple threads are going to change the value of a flag. Suppose now that the thread you want to stop can itself change the (non-volatile) flag state. If your main thread tries to modify the flag at the same time, based on the current flag state, it's possible that at the time of flag read, the actual (new) state of the flag is still cached and not committed to memory yet. Therefore the update by the main thread may not only be incorrect, but this value could then be overwritten by the previous state as it is committed. This is can cause all sorts of nastiness. When in doubt, use volatile for your flags.

What if your thread doesn't check the flag very often? This can occur if say your thread connects to a web server and downloads content. What if the content you are downloading is huge and you want to stop the thread before the download finishes? This is a case where Thread.interrupt() is useful. Thread.interrupt() is not like Thread.stop(). In fact, Thread.interrupt() works by using.... wait for it... flags! It's simply a more defined way of working with flags. Instead of defining your own flags, calling Thread.interrupt() on a thread will cause Thread.currentThread().isInterrupted() and

Thread.interrupted() to return true instead of false. Calling Thread.interrupted() returns the current flag state and then clears the flag to false - the other difference is that Thread.interrupted() is a static method which checks the current thread - equivalent to Thread.currentThread().isInterrupted(), but with flag clear support. When checking the state of a thread's interrupt flag from within the thread itself, use this.isInterrupted(). Many java methods which block support being interruptable, and many convey that they have been interrupted by throwing an InterruptedException, such as Thread.sleep(). In this way, you are not reinventing the wheel so to speak, but rather using the same flag / flag propagation system that many built-in methods already use.

One quick blurb about InterruptedExceptions. Say you code a super groovy thread which needs to sleep for some reason. Thread.sleep() throws an InterruptedException if you call interrupt() on the thread. You can choose not to handle it (catch it and do nothing), but then by doing this you completely defeat the purpose of what I was talking about in the previous paragraph. Instead, use the catch block to do any graceful cleanup or whatever else you need to do (often times when Thread.sleep() throws an InterruptedException, it can be interpreted as the thread 'waking up' by force).

Note that neither of these techniques will terminate a thread immediately. It is all dependent on the actual thread logic. Using stop() won't even terminate immediately, but its as close and forceful as you can get. Should you use stop() in your own code? As long as it works, I don't care :-)