Creating, Instantiating, and Starting New Threads

- Threads can be created by extending Thread and overriding the public void run() method.
- Thread objects can also be created by calling the Thread constructor that takes a Runnable argument. The Runnable object is said to be the *target* of the thread.
- You can call start() on a Thread object only once. If start() is called more than once on a Thread object, it will throw a RuntimeException.
- It is legal to create many Thread objects using the same Runnable object as the target.
- When a Thread object is created, it does not become a *thread of execution* untilits start() method is invoked. When a Thread object exists but hasn't been started, it is in the *new* state and is not considered *alive*.

Transitioning Between Thread States

- Once a new thread is started, it will always enter the runnable state.
- The thread scheduler can move a thread back and forth between the runnable state and the running state.
- Only one thread can be running at a time, although many threads may be in the runnable state.
- There is no guarantee that the order in which threads were started determines the order in which they'll run.
- There's no guarantee that threads will take turns in any fair way. It's up to the thread scheduler, as determined by the particular virtual machine implementation. If you want a guarantee that your threads will take turns regardless of the underlying JVM, you should can use the sleep() method.
- This prevents one thread from hogging the running process while another thread starves.
- A running thread may enter a blocked/waiting state by a wait(), sleep(), or join() call.
- A running thread may enter a blocked/waiting state because it can't acquire the lock for a synchronized block of code.
- When the sleep or wait is over, or an object's lock becomes available, the thread can only reenter the runnable state. It will go directly from waiting to running (well, for all practical purposes anyway).

- A dead thread cannot be started again.

Sleep, Yield, and Join

- Sleeping is used to delay execution for a period of time, and no locks are released when a thread goes to sleep.
- A sleeping thread is guaranteed to sleep for at least the time specified in the argument to the sleep method (unless it's interrupted), but there is no guarantee as to when the newly awakened thread will actually return to running.
- The sleep() method is a static method that sleeps the currently executing thread. One thread cannot tell another thread to sleep.
- The setPriority() method is used on Thread objects to give threads a priority of between 1 (low) and 10 (high), although priorities are not guaranteed, and not all JVMs use a priority range of 1-10.
- If not explicitly set, a thread's priority will be the same priority as the thread that created this thread (in other words, the thread executing the code that creates the new thread).
- The yield() method may cause a running thread to back out if there are runnable threads of the same priority. There is no guarantee that this will happen, and there is no guarantee that when the thread backs out it will be different thread selected to run. A thread might yield and then immediately reenter the running state.
- The closest thing to a guarantee is that at any given time, when a thread is running it will usually not have a lower priority than any thread in the runnable state. If a low-priority thread is running when a high-priority thread enters runnable, the JVM will preempt the running low-priority thread and put the high-priority thread in.
- When one thread calls the join() method of another thread, the currently running thread will wait until the thread it joins with has completed. Think of the join() method as saying, "Hey thread, I want to join on to the end of you. Let me know when you're done, so I can enter the runnable state."

Concurrent Access Problems and Synchronized Threads

- Synchronized methods prevent more than one thread from accessing an object's critical method code.

- You can use the synchronized keyword as a method modifier, or to start a synchronized block of code.
- To synchronize a block of code (in other words, a scope smaller than the whole method), you must specify an argument that is the object whose lock you want to synchronize on.
- While only one thread can be accessing synchronized code of a particular instance, multiple threads can still access the same object's unsynchronized code.
- When an object goes to sleep, it takes its locks with it.
- Static methods can be synchronized, using the lock from the java.lang.Class instance representing that class.

Communicating with Objects by Waiting and Notifying

- The wait() method lets a thread say, "there's nothing for me to do here, so put me in your waiting pool and notify me when something happens that I care about." Basically, a wait() call means "wait me in your pool," or "add me to your waiting list."
- The notify() method is used to send a signal to one and only one of the threads that are waiting in that same object's waiting pool.
- The method notifyAll() works in the same way as notify(), only it sends the signal to all of the threads waiting on the object.
- All three methods—wait()/notify()/notifyAll()—must be called from within a synchronized context! A thread invokes wait()/notify() on a particular object, and the thread must currently hold the lock on that object.

Deadlocked Threads

- Deadlocking is when thread execution grinds to a halt because the code is waiting for locks to be removed from objects.
- Deadlocking can occur when a locked object attempts to access another locked object that is trying to access the first locked object. In other words, both threads are waiting for each other's locks to be released; therefore, the locks will never be released!
- Deadlocking is bad. Don't do it.