Chapter 24 Multithreading

1. Multithreading can make your program more responsive and interactive, and enhance the performance. Multithreading is needed in many situations, such as animation and client/server computing. Because most of time the CPU is idle--for example, the CPU is doing nothing while the user enters data--it is practical for multiple threads to share the CPU time in single-processor systems.

2. You can create a task classes by implementing the Runnable interface and create a thread for a task using the constructor new Thread(task).

3. If you replace the start() method by the run() method in Lines 11-13 in Example 19.1, the run() method are executed in sequence. The threads are not executed concurrently.

4. (a) new Test() is recursively called inside the constructor. To fix it, delete the highlighted line and use new Thread(this).start(). (b) An illegal java.lang.IllegalThreadStateException may be thrown because you just started thread and thread might have not yet finished before you start it again. To fix it, delete one t.start().

5. All of the methods shown in Figure 23.4 except yield() and sleep(long) are instance methods. sleep() and join() may throw an InterruptedException. The methods stop(), suspend() and resume() are deprecated in JDK 1.2.

6. If the loop is outside the try-catch block, the thread may continue to execute even though it is being interrupted.

7. You use the setPriority() method to set the priority for a thread. The default priority of the thread is Thread.NORM_PRIORITY (5).

8. You can create a thread for each task. This approach is convenient for a single task execution, but it is not efficient for a large number of tasks, because you have to create a thread for each task. Starting a new thread for each task could limit throughput and cause poor performance. A thread pool is ideal to manage the number of tasks executing concurrently.

9. GUI event handling and painting code executes in a single thread, called the event dispatcher thread. This ensures that each event handler finishes executing before the next one executes and the painting isn’t interrupted by events. You can use the static methods, invokeLater and invokeAndWait, in the javax.swing.SwingUtilities class to run the code in the event dispatcher thread.

10. If you need a precise delay time or a quick response, it is better to use a thread. Otherwise, using a timer is simpler and more efficient than using a thread. Timers
consume less system resources than threads because timers run on the GUI event
dispatcher thread so you don’t need to spawn new threads for timers.

11. To create a thread pool with three threads, use

```java
ExecutorService executor = Executors.newFixedThreadPool(3);
```

To submit a task, use

```java
executor.execute(task);
```

To check whether all tasks in the pool are finished, invoke isTerminated() method.

12. See the section "Synchronization" for examples and solutions.

13. No. synchronized (this) acquires a lock on a thread (an instance of
    AddAPennyThread). Each thread still can access the object bank concurrently. To
    fix the problem, acquire the lock on bank using synchronized (bank).

14. To create a lock, use the `new ReentrantLock()`. To acquire the lock, invoke its
    lock() method and to release it, invoke its unlock() method.

15. A condition on a lock can be created using lock.newCondition(). The await() method
    causes the current thread to wait until the condition is signaled. The
    signal() method wakes up one waiting thread, and the signalAll() method wakes
    up all waiting threads.

16. When a thread notify a waiting thread, you cannot assume the balance >= amount
    for the waiting thread. The condition (balance < amount) may be still true when
    the thread is awakened.

17. To override the init() method defined in the Applet class, you have to use the
    exact signature. The init() method does not claim throwing exceptions.

18. Yes. These methods are defined in the Object class and they are used for thread
    communication.

19. If you invoke the methods on a condition without first acquiring a lock for the
    condition, an `IllegalMonitorStateException` would be thrown.

20. object1 and object2 are not the same. A lock must be acquired on the receiving
    object of the wait(), notify(), and notifyAll() methods before invoking these
    methods.
21. The ArrayBlockingQueue, LinkedBlockingQueue, and PriorityBlockingQueue are supported in JDK 1.5. All are concrete classes of the BlockingQueue interface.

22. Use put(element) method to add an element to the queue. If the queue is full, the thread is blocked.

23. Use take() method to retrieve an element from the queue. If the queue is empty, the thread is blocked.

24. Lock and semaphore can both be used to restrict access to a shared resource. Using a lock on a resource ensures only one thread can access it. Using a semaphore on a resource allows one or more specified number of threads to access a resource.

25. Use new Semaphore(numberOfPermits) to create a semaphore. Invoking acquire() to get a semaphore and invoking release() to release a semaphore.

26. Deadlock occurs in the case that two or more threads acquire locks on multiple objects and each has the lock on one object and is waiting for the lock on the other object. The resource ordering technique can be used to avoid deadlock.

27. A synchronized collection is thread-safe, i.e., it can be accessed by multiple threads concurrently without being corrupted. ArrayList is not thread-safe. There are several ways to make it safe. You may obtain a lock before accessing it, or use the Collections.synchronizedList(list) to return a synchronized list.

28. An iterator is fail-fast. This means that if you are using an iterator to traverse a collection while the underlying collection is being modified by another thread, then the iterator will immediately fail by throwing java.util.ConcurrentModificationException.

29. The stringPainted property. The setOrientation method