26

Multithreading: Solutions

The most general definition of beauty…Multeity in Unity.
—Samuel Taylor Coleridge

Do not block the way of inquiry.
—Charles Sanders Peirce

A person with one watch knows what time it is; a person with two watches is never sure.
—Proverb

Learn to labor and to wait.
—Henry Wadsworth Longfellow

The world is moving so fast these days that the man who says it can't be done is generally interrupted by someone doing it.
—Elbert Hubbard

Objectives

In this chapter you'll learn:

- What threads are and why they're useful.
- How threads enable you to manage concurrent activities.
- The life cycle of a thread.
- To create and execute `Runnable`s.
- Thread synchronization.
- What producer/consumer relationships are and how they're implemented with multithreading.
- To enable multiple threads to update Swing GUI components in a thread-safe manner.

Self-Review Exercises

26.1 Fill in the blanks in each of the following statements:

a) A thread enters the *terminated* state when _____.

ANS: its `run` method ends.

b) To pause for a designated number of milliseconds and resume execution, a thread should call method _____ of class _____.

ANS: `sleep`, `Thread`.

c) Method _____ of class `Condition` moves a single thread in an object's *waiting* state to the *runnable* state.

ANS: `signal`.

d) Method _____ of class `Condition` moves every thread in an object's *waiting* state to the *runnable* state.

ANS: `signalAll`.

e) A(n) _____ thread enters the _____ state when it completes its task or otherwise terminates.

ANS: *runnable*, *terminated*.

f) A *runnable* thread can enter the _____ state for a specified interval of time.

ANS: *timed waiting*.

g) At the operating-system level, the *runnable* state actually encompasses two separate states, _____ and _____.

ANS: *ready*, *running*.

h) `Runnable`s are executed using a class that implements the _____ interface.

ANS: `Executor`.

i) `ExecutorService` method _____ ends each thread in an `ExecutorService` as soon as it finishes executing its current `Runnable`, if any.

ANS: `shutdown`.

j) A thread can call method _____ on a `Condition` object to release the associated `Lock` and place that thread in the _____ state.

ANS: `await`, *waiting*.

k) In a(n) _____ relationship, the _____ generates data and stores it in a shared object, and the _____ reads data from the shared object.

ANS: producer/consumer, producer, consumer.

l) Class _____ implements the `BlockingQueue` interface using an array.

ANS: `ArrayBlockingQueue`.

m) Keyword _____ indicates that only one thread at a time should execute on an object.

ANS: `synchronized`.

26.2 State whether each of the following is *true* or *false*. If *false*, explain why.

a) A thread is not *runnable* if it has terminated.

ANS: True.

b) Some operating systems use timeslicing with threads. Therefore, they can enable threads to preempt threads of the same priority.

ANS: False. Timeslicing allows a thread to execute until its timeslice (or quantum) expires. Then other threads of equal priority can execute.

c) When the thread's quantum expires, the thread returns to the *running* state as the operating system assigns it to a processor.

ANS: False. When a thread's quantum expires, the thread returns to the *ready* state and the operating system assigns to the processor another thread.

d) On a single-processor system without timeslicing, each thread in a set of equal-priority threads (with no other threads present) runs to completion before other threads of equal priority get a chance to execute.

ANS: True.

Exercises

NOTE: Solutions to the programming exercises are located in the `ch26solutions` folder. Each exercise has its own folder named `ex26_##` where ## is a two-digit number representing the exercise number. For example, exercise 26.8's solution is located in the folder `ex26_08`.

26.3 State whether each of the following is *true* or *false*. If *false*, explain why.

a) Method `sleep` does not consume processor time while a thread sleeps.

ANS: True.

b) Declaring a method `synchronized` guarantees that deadlock cannot occur.

ANS: False. Deadlocks can occur if the lock on an object is never released.

c) Once a `ReentrantLock` has been obtained by a thread, the `ReentrantLock` object will not allow another thread to obtain the lock until the first thread releases it.

ANS: True.

d) Swing components are thread safe.

ANS: False. Swing components are not thread safe. All interactions with Swing GUI components should be performed in the event-dispatching thread.

26.4 Define each of the following terms.

a) thread

ANS: An individual execution context of a program.

b) multithreading

ANS: The ability of more than one thread to execute concurrently.

c) *runnable* state

ANS: A state in which the thread is capable of running (if the processor becomes available).

d) *timed waiting* state

ANS: A state in which the thread cannot use the processor because it is waiting for a time interval to expire or a notification from another thread.

e) preemptive scheduling

ANS: A thread of higher priority enters a *running* state and is assigned to the processor. The thread preempted from the processor is placed back in the *ready* state according to its priority.

f) `Runnable` interface

ANS: An interface that provides a `run` method. By implementing the `Runnable` interface, any class can be executed as a separate thread.

g) `notifyAll` method

ANS: Transitions all threads waiting on an object's monitor to the *runnable* state.

h) producer/consumer relationship

ANS: A relationship in which a producer and a consumer share common data. The producer typically wants to "produce" (add information) and the consumer wants to "consume" (remove information).

i) quantum

ANS: A small amount of processor time, also called a time slice.

26.5 Discuss each of the following terms in the context of Java's threading mechanisms:

a) `synchronized`

ANS: When a method or block is declared `synchronized` and it is running, the object is locked. Other threads cannot access the other `synchronized` methods of the object until the lock is released.

b) producer

ANS: A thread that writes data to a shared memory resource.

c) consumer

ANS: A thread that reads data from a shared memory resource.

d) `wait`

ANS: Places a thread in the *waiting* state until another thread calls `notify` or `notifyAll` on the same object or until a specified amount of time elapses.

e) `notify`

ANS: Wake a thread currently waiting on the given object.

f) `Lock`

ANS: An interface implemented by objects that control access to a resource shared among multiple threads. Only one thread can be holding a `Lock` at one time—a second thread calling the `lock` method will block until the `unlock` method is called. Using the `Lock` interface is more complicated than using the `synchronized` keyword, but is more flexible.

g) `Condition`

ANS: Objects of this interface represent condition variables that can be used with `Locks` to manage access to a shared resource.

26.6 List the reasons for entering the *blocked* state. For each of these, describe how the program will normally leave the *blocked* state and enter the *runnable* state.

ANS: A thread transitions to the *blocked* state when it attempts to perform a task that cannot be completed immediately and the thread must temporarily wait until that task completes. For example, when a thread issues an input/output request, the operating system blocks the thread from executing until that I/O request completes—at that point, the *blocked* thread transitions to the *runnable* state, so it can resume execution. A thread also transitions to the *blocked* state when it attempts to acquire a monitor lock that is not currently available. When the lock becomes available, the thread returns to the *runnable* state and attempts to acquire the lock.

26.7 Two problems that can occur in systems that allow threads to wait are deadlock, in which one or more threads will wait forever for an event that cannot occur, and indefinite postponement, in which one or more threads will be delayed for some unpredictably long time. Give an example of how each of these problems can occur in multithreaded Java programs.

ANS: Deadlock: If we have two threads named `thread1` and `thread2`, deadlock might occur in the following situation: If `thread1` is waiting for `thread2` to complete a task, and `thread2` is waiting for `thread1` to complete a task, then neither thread can continue. Since both threads are in the waiting state, neither thread can be signaled to continue executing. To help prevent deadlocks, ensure locks are always taken in the same order and released in the opposite order they were taken.

Indefinite Postponement: This typically occurs because threads of higher priority are scheduled before threads of lower priority.