Figure 9.3 shows the states and the transitions in the life cycle of a thread.

- *Ready-to-run state*

  A thread starts life in the Ready-to-run state (see p. 369).

- *Running state*

  If a thread is in the Running state, it means that the thread is currently executing (see p. 369).

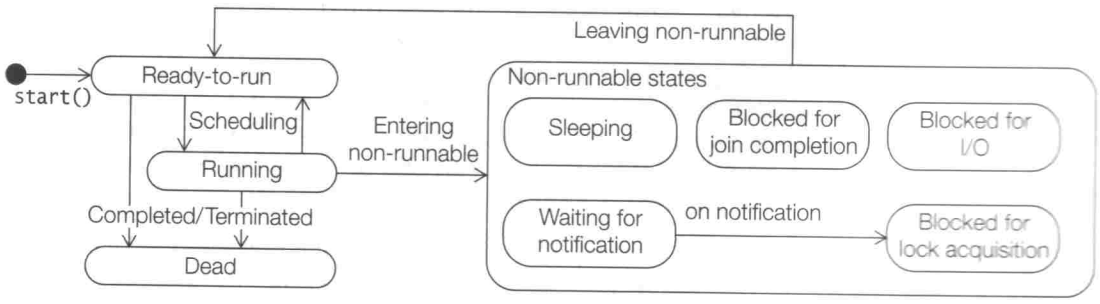
- *Dead state*

  Once in this state, the thread cannot ever run again (see p. 380).

- *Non-runnable states*

  A running thread can transit to one of the non-runnable states, depending on the circumstances. A thread remains in a non-runnable state until a special transition occurs. A thread does not go directly to the Running state from a non-runnable state, but transits first to the Ready-to-run state.
  The non-runnable states can be characterized as follows:
  - *Sleeping*: The thread *sleeps* for a specified amount of time (see p. 370).
  - *Blocked for I/O*: The thread waits for a *blocking* operation to complete (see p. 380).
  - *Blocked for join completion*: The thread awaits *completion* of another thread (see p. 377).
  - *Waiting for notification*: The thread awaits *notification* from another thread (see p. 370).
  - *Blocked for lock acquisition*: The thread waits to *acquire* the lock of an object (see p. 359).

Figure 9.3 *Thread States*



Various methods from the `Thread` class are presented next. Examples of their usage are presented in subsequent sections.

`final boolean isAlive()`

This method can be used to find out if a thread is alive or dead. A thread is *alive* if it has been started but not yet terminated, that is, it is not in the Dead state.

```
final int getPriority()
final void setPriority(int newPriority)
```

The first method returns the priority of the current thread. The second method changes its priority. The priority set will be the minimum of the two values: the specified `newPriority` and the maximum priority permitted for this thread.

`static void yield()`

This method causes the current thread to temporarily pause its execution and, thereby, allow other threads to execute.

`static void sleep (long millisec) throws InterruptedException`

The current thread sleeps for the specified time before it takes its turn at running again.

```
final void join() throws InterruptedException
final void join(long millisec) throws InterruptedException
```

A call to any of these two methods invoked on a thread will wait and not return until either the thread has completed or it is timed out after the specified time, respectively.

`void interrupt()`

The method interrupts the thread on which it is invoked. In the Waiting-for-notification, Sleeping, or Blocked-for-join-completion states, the thread will receive an `InterruptedException`.

Thread Priorities

Threads are assigned priorities that the thread scheduler *can* use to determine how the threads will be scheduled. The thread scheduler can use thread priorities to determine which thread gets to run. The thread scheduler favors giving CPU time to the thread with the highest priority in the Ready-to-run state. This is not necessarily the thread that has been the longest time in the Ready-to-run state. Heavy reliance on thread priorities for the behavior of a program can make the program unportable across platforms, as thread scheduling is host platform–dependent.

Priorities are integer values from 1 (lowest priority given by the constant `Thread.MIN_PRIORITY`) to 10 (highest priority given by the constant `Thread.MAX_PRIORITY`). The default priority is 5 (`Thread.NORM_PRIORITY`).

A thread inherits the priority of its parent thread. Priority of a thread can be set using the `setPriority()` method and read using the `getPriority()` method, both of which are defined in the `Thread` class. The following code sets the priority of the

thread `myThread` to the minimum of two values: maximum priority and current priority incremented to the next level:

```
myThread.setPriority(Math.min(Thread.MAX_PRIORITY, myThread.getPriority()+1));
```

Thread Scheduler

Schedulers in JVM implementations usually employ one of the two following strategies:

- Preemptive scheduling.

  If a thread with a higher priority than the current running thread moves to the Ready-to-run state, then the current running thread can be *preempted* (moved to the Ready-to-run state) to let the higher priority thread execute.

- Time-Sliced or Round-Robin scheduling.

  A running thread is allowed to execute for a fixed length of time, after which it moves to the Ready-to-run state to await its turn to run again.

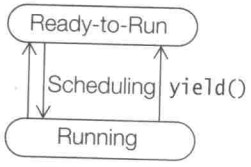
It should be pointed out that thread schedulers are implementation- and platform-dependent; therefore, how threads will be scheduled is unpredictable, at least from platform to platform.

Running and Yielding

After its `start()` method has been called, the thread starts life in the Ready-to-run state. Once in the Ready-to-run state, the thread is eligible for running, that is, it waits for its turn to get CPU time. The thread scheduler decides which thread gets to run and for how long.

Figure 9.4 illustrates the transitions between the Ready-to-Run and Running states. A call to the static method `yield()`, defined in the `Thread` class, will cause the current thread in the Running state to transit to the Ready-to-run state, thus relinquishing the CPU. The thread is then at the mercy of the thread scheduler as to when it will run again. If there are no threads waiting in the Ready-to-run state, this thread continues execution. If there are other threads in the Ready-to-run state, their priorities determine which thread gets to execute.

Figure 9.4 *Running and Yielding*



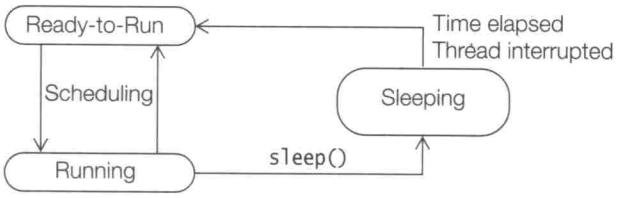
By calling the static method `yield()`, the running thread gives other threads in the Ready-to-run state a chance to run. A typical example where this can be useful is

when a user has given some command to start a CPU-intensive computation, and has the option of canceling it by clicking on a Cancel button. If the computation thread hogs the CPU and the user clicks the Cancel button, chances are that it might take a while before the thread monitoring the user input gets a chance to run and take appropriate action to stop the computation. A thread running such a computation should do the computation in increments, yielding between increments to allow other threads to run. This is illustrated by the following `run()` method:

```
public void run() {
    try {
        while (!done()) {
            doLittleBitMore();
            Thread.yield();          // Current thread yields
        }
    } catch (InterruptedException e) {
        doCleaningUp();
    }
}
```

Sleeping and Waking up

Figure 9.5 *Sleeping and Waking up*



A call to the static method `sleep()` in the `Thread` class will cause the currently running thread to pause its execution and transit to the Sleeping state. The method does not relinquish any lock that the thread might have. The thread will sleep for at least the time specified in its argument, before transitioning to the Ready-to-run state where it takes its turn to run again. If a thread is interrupted while sleeping, it will throw an `InterruptedException` when it awakes and gets to execute.

There are serveral overloaded versions of the `sleep()` method in the `Thread` class.

Usage of the `sleep()` method is illustrated in Examples 9.1, 9.2, and 9.3.

Waiting and Notifying

Waiting and notifying provide means of communication between threads that *synchronize on the same object* (see Section 9.4, p. 359). The threads execute `wait()` and `notify()` (or `notifyAll()`) methods on the shared object for this purpose. These `final` methods are defined in the `Object` class, and therefore, inherited by all objects.

These methods can only be executed on an object whose lock the thread holds, otherwise, the call will result in an `IllegalMonitorStateException`.

```
final void wait(long timeout) throws InterruptedException
final void wait(long timeout, int nanos) throws InterruptedException
final void wait() throws InterruptedException
```

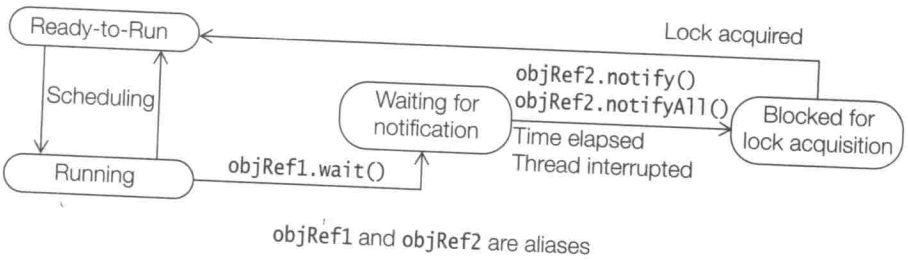
A thread invokes the `wait()` method on the object whose lock it holds. The thread is added to the *wait set* of the object.

```
final void notify()
final void notifyAll()
```

A thread invokes a notification method on the object whose lock it holds to notify thread(s) that are in the wait set of the object.

Communication between threads is facilitated by waiting and notifying, as illustrated by Figures 9.6 and 9.7. A thread usually calls the `wait()` method on the object whose lock it holds because a condition for its continued execution was not met. The thread leaves the Running state and transits to the Waiting-for-notification state. There it waits for this condition to occur. The thread relinquishes ownership of the object lock.

Figure 9.6 *Waiting and Notifying*



Transition to the Waiting-for-notification state and relinquishing the object lock are completed as one *atomic* (non-interruptable) operation. The releasing of the lock of the shared object by the thread allows other threads to run and execute synchronized code on the same object after acquiring its lock.

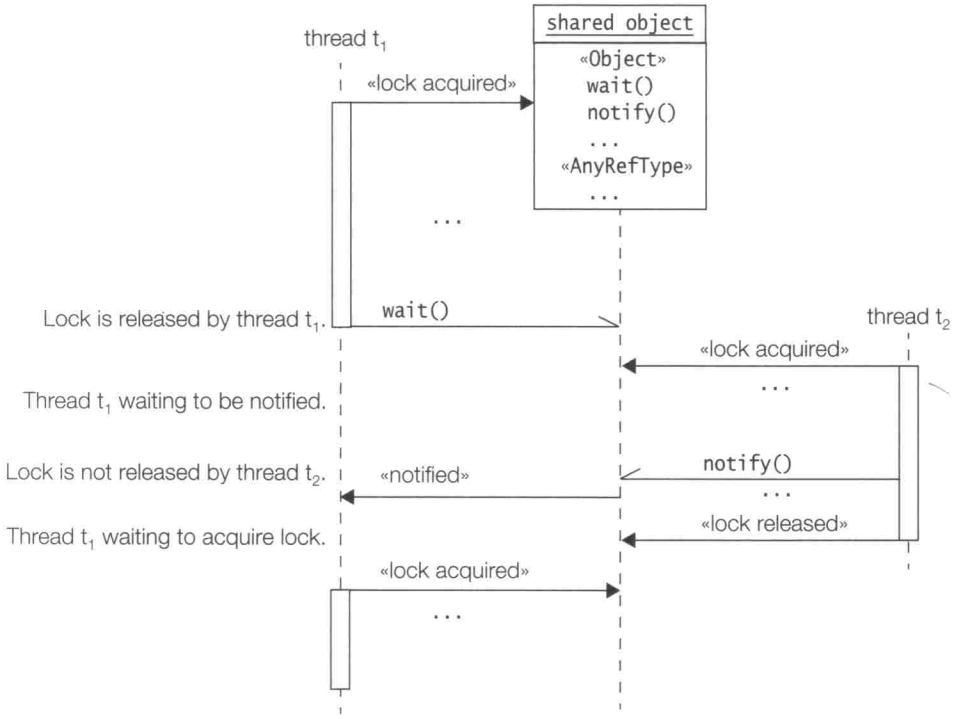
Note that the waiting thread does not relinquish any other object locks that it might hold, only that of the object on which the `wait()` method was invoked. Objects that have these other locks remain locked while the thread is waiting.

Each object has a *wait set* containing threads waiting for notification. Threads in the Waiting-for-notification state are grouped according to the object whose `wait()` method they invoked.

Figure 9.7 shows a thread t_1 that first acquires a lock on the shared object, and afterwards invokes the `wait()` method on the shared object. This relinquishes the object

lock and the thread t_1 awaits to be notified. While the thread t_1 is waiting, another thread t_2 can acquire the lock on the shared object for its own purpose.

Figure 9.7 *Thread Communication*



A thread in the Waiting-for-notification state can be awakened by the occurrence of any one of these three incidents:

1. Another thread invokes the `notify()` method on the object of the waiting thread, and the waiting thread is selected as the thread to be awakened.
2. The waiting thread times out.
3. Another thread interrupts the waiting thread.

Notify

Invoking the `notify()` method on an object wakes up a single thread that is waiting on the lock of this object. The selection of a thread to awaken is dependent on the thread policies implemented by the JVM. On being *notified*, a waiting thread first transits to the Blocked-for-lock-acquisition state to acquire the lock on the object, and not directly to the Ready-to-run state. The thread is also removed from the wait set of the object. Note that the object lock is not relinquished when the notifying thread invokes the `notify()` method. The notifying thread relinquishes the lock