PGJC3.book Page 613 Tuesday, November 25, 2008 9:10 PM

13 Threads

Exam Objectives
4.1 Write code to define, instantiate, and start new threads using both `java.lang.Thread` and `java.lang.Runnable`.
4.2 Recognize the states in which a thread can exist, and identify ways in which a thread can transition from one state to another.
4.3 Given a scenario, write code that makes appropriate use of object locking to protect static or instance variables from concurrent access problems.
4.4 Given a scenario, write code that makes appropriate use of `wait`, `notify`, or `notifyAll`.
Supplementary Objectives
• Recognize conditions that might prevent a thread from executing.
• Write code to start and stop a thread.
• Understand aspects of thread behavior that are not guaranteed.

PGJC3.book Page 614 Tuesday, November 25, 2008 9:10 PM

13.1 Multitasking

Multitasking allows several activities to occur concurrently on the computer. A distinction is usually made between:

- Process-based multitasking
- Thread-based multitasking

At the coarse-grain level there is *process-based* multitasking, which allows processes (i.e., programs) to run concurrently on the computer. A familiar example is running the spreadsheet program while also working with the word-processor. At the fine-grain level there is *thread-based* multitasking, which allows parts of the *same* program to run concurrently on the computer. A familiar example is a word-processor that is printing and formatting text at the same time. This is only feasible if the two tasks are performed by two independent paths of execution at runtime. The two tasks would correspond to executing parts of the program concurrently. The sequence of code executed for each task defines a separate path of execution, and is called a *thread* (*of execution*).

In a single-threaded environment only one task at a time can be performed. CPU cycles are wasted, for example, when waiting for user input. Multitasking allows idle CPU time to be put to good use.

Some advantages of thread-based multitasking as compared to process-based multitasking are:

- threads share the same address space
- context switching between threads is usually less expensive than between processes
- the cost of communication between threads is relatively low

Java supports thread-based multitasking and provides high-level facilities for multithreaded programming. *Thread safety* is the term used to describe the design of classes that ensure that the state of their objects is always consistent, even when the objects are used concurrently by multiple threads.

13.2 Overview of Threads

A thread is an independent sequential path of execution within a program. Many threads can run concurrently within a program. At runtime, threads in a program exist in a common memory space and can, therefore, share both data and code (i.e., they are *lightweight* compared to processes). They also share the process running the program.

Every thread in Java is created and controlled by a unique object of the `java.lang.Thread` class. Often the thread and its associated `Thread` object are thought of as being synonymous.

PGJC3.book Page 615 Tuesday, November 25, 2008 9:10 PM

Threads make the runtime environment asynchronous, allowing different tasks to be performed concurrently. Using this powerful paradigm in Java centers around understanding the following aspects of multithreaded programming:

- creating threads and providing the code that gets executed by a thread (see Section 13.4, p. 615)
- accessing common data and code through synchronization (see Section 13.5, p. 626).
- transitioning between thread states (see Section 13.6, p. 634).

13.3 The Main Thread

The runtime environment distinguishes between *user threads* and *daemon threads*. As long as a user thread is alive, the JVM does not terminate. A daemon thread is at the mercy of the runtime system: it is stopped if there are no more user threads running, thus terminating the program. Daemon threads exist only to serve user threads.

When a standalone application is run, a user thread is automatically created to execute the `main()` method of the application. This thread is called the *main thread*. If no other user threads are spawned, the program terminates when the `main()` method finishes executing. All other threads, called *child* threads, are spawned from the main thread, inheriting its user-thread status. The `main()` method can then finish, but the program will keep running until all user threads have completed. Calling the `setDaemon(boolean)` method in the `Thread` class marks the status of the thread as either daemon or user, but this must be done before the thread is *started*. Any attempt to change the status after the thread has been started, throws an `IllegalThreadStateException`. Marking all spawned threads as daemon threads ensures that the application terminates when the main thread dies.

When a GUI application is started, a special thread is automatically created to monitor the user–GUI interaction. This user thread keeps the program running, allowing interaction between the user and the GUI, even though the main thread might have completed after the `main()` method finished executing.

13.4 Thread Creation

A thread in Java is represented by an object of the `Thread` class. Implementing threads is achieved in one of two ways:

- implementing the `java.lang.Runnable` interface
- extending the `java.lang.Thread` class

PGJC3.book Page 616 Tuesday, November 25, 2008 9:10 PM

Implementing the `Runnable` Interface

The `Runnable` interface has the following specification, comprising one abstract method declaration:

```
public interface Runnable {
  void run();
}
```

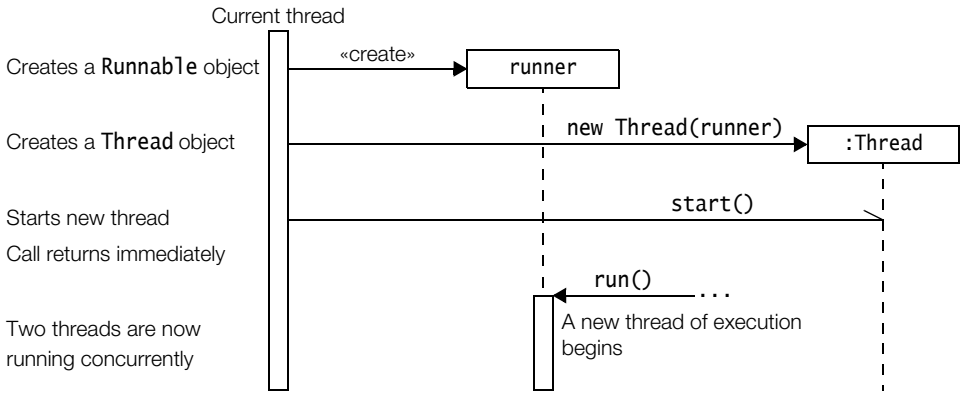
A thread, which is created based on an object that implements the `Runnable` interface, will execute the code defined in the public method `run()`. In other words, the code in the `run()` method defines an independent path of execution and thereby the entry and the exits for the thread. A thread ends when the `run()` method ends, either by normal completion or by throwing an uncaught exception.

The procedure for creating threads based on the `Runnable` interface is as follows:

1. A class implements the `Runnable` interface, providing the `run()` method that will be executed by the thread. An object of this class is a `Runnable` object.
2. An object of the `Thread` class is created by passing a `Runnable` object as an argument in the `Thread` constructor call. The `Thread` object now has a `Runnable` object that implements the `run()` method.
3. The `start()` method is invoked on the `Thread` object created in the previous step. The `start()` method returns immediately after a thread has been spawned. In other words, the call to the `start()` method is asynchronous.

When the thread, represented by the `Thread` object on which the `start()` method was invoked, gets to run, it executes the `run()` method of the `Runnable` object. This sequence of events is illustrated in Figure 13.1.

Figure 13.1 *Spawning Threads Using a* `Runnable` *Object*



PGJC3.book Page 617 Tuesday, November 25, 2008 9:10 PM

The following is a summary of important constructors and methods from the `java.lang.Thread` class:

```
Thread(Runnable threadTarget)
Thread(Runnable threadTarget, String threadName)
```

The argument `threadTarget` is the object whose `run()` method will be executed when the thread is started. The argument `threadName` can be specified to give an explicit name for the thread, rather than an automatically generated one. A thread's name can be retrieved by calling the `getName()` method.

```
static Thread currentThread()
```

This method returns a reference to the `Thread` object of the currently executing thread.

```
final String getName()
final void setName(String name)
```

The first method returns the name of the thread. The second one sets the thread's name to the specified argument.

```
void run()
```

The `Thread` class implements the `Runnable` interface by providing an implementation of the `run()` method. This implementation in the `Thread` class does nothing and returns. Subclasses of the `Thread` class should override this method. If the current thread is created using a separate `Runnable` object, the `run()` method of the `Runnable` object is called.

```
final void setDaemon(boolean flag)
final boolean isDaemon()
```

The first method sets the status of the thread either as a daemon thread or as a user thread, depending on whether the argument is `true` or `false`, respectively. The status should be set before the thread is started. The second method returns `true` if the thread is a daemon thread, otherwise, `false`.

```
void start()
```

This method spawns a new thread, i.e., the new thread will begin execution as a child thread of the current thread. The spawning is done asynchronously as the call to this method returns immediately. It throws an `IllegalThreadStateException` if the thread is already started.

In Example 13.1, the class `Counter` implements the `Runnable` interface. At (1), the class defines the `run()` method that constitutes the code to be executed in a thread. In each iteration of the `while` loop, the current value of the counter is printed and incremented, as shown at (2). Also, in each iteration, the thread will sleep for 250 milliseconds, as shown at (3). While it is sleeping, other threads may run (see Section 13.6, p. 640).

PGJC3.book Page 618 Tuesday, November 25, 2008 9:10 PM

The code in the `main()` method ensures that the `Counter` object created at (4) is passed to a new `Thread` object in the constructor call, as shown at (5). In addition, the thread is enabled for execution by the call to its `start()` method, as shown at (6).

The static method `currentThread()` in the `Thread` class can be used to obtain a reference to the `Thread` object associated with the current thread. We can call the `getName()` method on the current thread to obtain its name. An example of its usage is shown at (2), that prints the name of the thread executing the `run()` method. Another example of its usage is shown at (8), that prints the name of the thread executing the `main()` method.

Example 13.1 *Implementing the* `Runnable` *Interface*

```
class Counter implements Runnable {
  private int currentValue;
  public Counter() { currentValue = 0; }
  public int getValue() { return currentValue; }

  public void run() {                           // (1) Thread entry point
    try {
      while (currentValue < 5) {
        System.out.println(
            Thread.currentThread().getName()    // (2) Print thread name.
            + ": " + (currentValue++)
        );
        Thread.sleep(250);                      // (3) Current thread sleeps.
      }
    } catch (InterruptedException e) {
      System.out.println(Thread.currentThread().getName() + " interrupted.");
    }
    System.out.println("Exit from thread: " + Thread.currentThread().getName());
  }
}
//_____________________________________________________________________________
public class Client {
  public static void main(String[] args) {
    Counter counterA = new Counter();                 // (4) Create a counter.
    Thread worker = new Thread(counterA, "Counter A");// (5) Create a new thread.
    System.out.println(worker);
    worker.start();                                   // (6) Start the thread.

    try {
      int val;
      do {
        val = counterA.getValue();                // (7) Access the counter value.
        System.out.println(
            "Counter value read by " +
            Thread.currentThread().getName() +    // (8) Print thread name.
            ": " + val
        );
        Thread.sleep(1000);                       // (9) Current thread sleeps.
      } while (val < 5);
    } catch (InterruptedException e) {
```

PGJC3.book Page 619 Tuesday, November 25, 2008 9:10 PM


```
            System.out.println("The main thread is interrupted.");
        }

        System.out.println("Exit from main() method.");
    }
}
```

Possible output from the program:

```
Thread[Counter A,5,main]
Counter value read by main thread: 0
Counter A: 0
Counter A: 1
Counter A: 2
Counter A: 3
Counter value read by main thread: 4
Counter A: 4
Exit from thread: Counter A
Counter value read by main thread: 5
Exit from main() method.
```

The `Client` class uses the `Counter` class. It creates an object of the class `Counter` at (4) and retrieves its value in a loop at (7). After each retrieval, the main thread sleeps for 1,000 milliseconds at (9), allowing other threads to run.

Note that the main thread executing in the `Client` class sleeps for a longer time between iterations than the `Counter A` thread, giving the `Counter A` thread the opportunity to run as well. The `Counter A` thread is a *child* thread of the main thread. It inherits the user-thread status from the main thread. If the code after the statement at (6) in the `main()` method was removed, the main thread would finish executing before the child thread. However, the program would continue running until the child thread completed its execution.

Since thread scheduling is not predictable (Section 13.6, p. 638) and Example 13.1 does not enforce any synchronization between the two threads in accessing the counter value, the output shown may vary. The first line of the output shows the string representation of the `Thread` object associated with the counter: its name (`Counter A`), its priority (`5`), and its parent thread (`main`). The output from the `main` thread and the `Counter A` thread is interspersed. It also shows that the value in the `Counter A` thread was incremented faster than the `main` thread could access the counter's value after each increment.

Extending the `Thread` Class

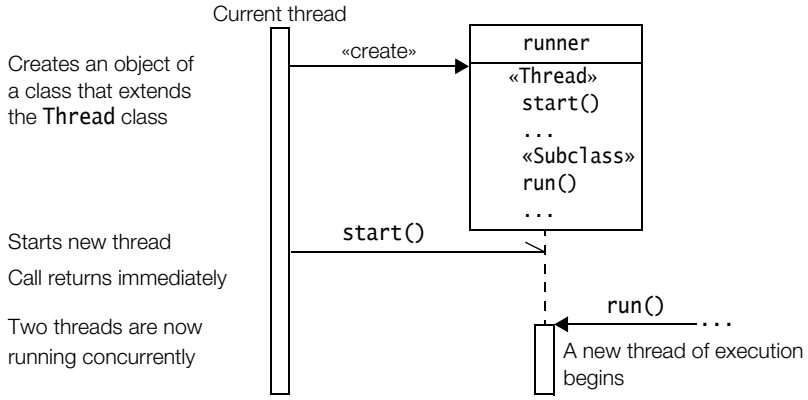
A class can also extend the `Thread` class to create a thread. A typical procedure for doing this is as follows (see Figure 13.2):

1. A class extending the `Thread` class overrides the `run()` method from the `Thread` class to define the code executed by the thread.
2. This subclass may call a `Thread` constructor explicitly in its constructors to initialize the thread, using the `super()` call.

PGJC3.book Page 620 Tuesday, November 25, 2008 9:10 PM

3. The `start()` method inherited from the `Thread` class is invoked on the object of the class to make the thread eligible for running.

Figure 13.2 *Spawning Threads—Extending the* `Thread` *Class*



In Example 13.2, the `Counter` class from Example 13.1 has been modified to illustrate creating a thread by extending the `Thread` class. Note the call to the constructor of the superclass `Thread` at (1) and the invocation of the inherited `start()` method at (2) in the constructor of the `Counter` class. The program output shows that the `Client` class creates two threads and exits, but the program continues running until the child threads have completed. The two child threads are independent, each having its own counter and executing its own `run()` method.

The `Thread` class implements the `Runnable` interface, which means that this approach is not much different from implementing the `Runnable` interface directly. The only difference is that the roles of the `Runnable` object and the `Thread` object are combined in a single object.

Adding the following statement before the call to the `start()` method at (2) in Example 13.2:

```
setDaemon(true);
```

illustrates the daemon nature of threads. The program execution will now terminate after the main thread has completed, without waiting for the daemon `Counter` threads to finish normally:

```
Method main() runs in thread main
Thread[Counter A,5,main]
Thread[Counter B,5,main]
Counter A: 0
Exit from main() method.
Counter B: 0
```

PGJC3.book Page 621 Tuesday, November 25, 2008 9:10 PM

Example 13.2 *Extending the* `Thread` *Class*

```
class Counter extends Thread {

  private int currentValue;

  public Counter(String threadName) {
    super(threadName);                          // (1) Initialize thread.
    currentValue = 0;
    System.out.println(this);
//  setDaemon(true);
    start();                                    // (2) Start this thread.
  }

  public int getValue() { return currentValue; }

  public void run() {                           // (3) Override from superclass.
    try {
      while (currentValue < 5) {
        System.out.println(getName() + ": " + (currentValue++));
        Thread.sleep(250);                      // (4) Current thread sleeps.
      }
    } catch (InterruptedException e) {
      System.out.println(getName() + " interrupted.");
    }
    System.out.println("Exit from thread: " + getName());
  }
}
//___________________________________________________________________________
public class Client {
  public static void main(String[] args) {

    System.out.println("Method main() runs in thread " +
        Thread.currentThread().getName());      // (5) Current thread

    Counter counterA = new Counter("Counter A"); // (6) Create a thread.
    Counter counterB = new Counter("Counter B"); // (7) Create a thread.

    System.out.println("Exit from main() method.");
  }
}
```

Possible output from the program:

```
Method main() runs in thread main
Thread[Counter A,5,main]
Thread[Counter B,5,main]
Exit from main() method.
Counter A: 0
Counter B: 0
Counter A: 1
Counter B: 1
Counter A: 2
Counter B: 2
Counter A: 3
```

PGJC3.book Page 622 Tuesday, November 25, 2008 9:10 PM


```
Counter B: 3
Counter A: 4
Counter B: 4
Exit from thread: Counter A
Exit from thread: Counter B
```

When creating threads, there are two reasons why implementing the `Runnable` interface may be preferable to extending the `Thread` class:

- Extending the `Thread` class means that the subclass cannot extend any other class, whereas a class implementing the `Runnable` interface has this option.
- A class might only be interested in being runnable and, therefore, inheriting the full overhead of the `Thread` class would be excessive.

The two previous examples illustrated two different ways to create a thread. In Example 13.1 the code to create the `Thread` object and call the `start()` method to initiate the thread execution is in the client code, not in the `Counter` class. In Example 13.2, this functionality is in the constructor of the `Counter` class, not in the client code.

Inner classes are useful for implementing threads that do simple tasks. The anonymous class below will create a thread and start it:

```
(new Thread() {
   public void run() {
     for(;;) System.out.println("Stop the world!");
   }
 }
).start();
```

Review Questions

13.1 Which is the correct way to start a new thread?

Select the one correct answer.

(a) Just create a new `Thread` object. The thread will start automatically.
(b) Create a new `Thread` object and call the method `begin()`.
(c) Create a new `Thread` object and call the method `start()`.
(d) Create a new `Thread` object and call the method `run()`.
(e) Create a new `Thread` object and call the method `resume()`.

13.2 When extending the `Thread` class to implement the code executed by a thread, which method should be overridden?

Select the one correct answer.

(a) `begin()`
(b) `start()`
(c) `run()`

PGJC3.book Page 623 Tuesday, November 25, 2008 9:10 PM

(d) `resume()`
(e) `behavior()`

13.3 Which statements are true?

Select the two correct answers.

(a) The class `Thread` is `abstract`.
(b) The class `Thread` implements `Runnable`.
(c) The `Runnable` interface has a single method named `start`.
(d) Calling the method `run()` on an object implementing `Runnable` will create a new thread.
(e) A program terminates when the last user thread finishes.

13.4 What will be the result of attempting to compile and run the following program?

```
public class MyClass extends Thread {
  public MyClass(String s) { msg = s; }
  String msg;
  public void run() {
    System.out.println(msg);
  }

  public static void main(String[] args) {
    new MyClass("Hello");
    new MyClass("World");
  }
}
```

Select the one correct answer.

(a) The program will fail to compile.
(b) The program will compile without errors and will print `Hello` and `World`, in that order, every time the program is run.
(c) The program will compile without errors and will print a never-ending stream of `Hello` and `World`.
(d) The program will compile without errors and will print `Hello` and `World` when run, but the order is unpredictable.
(e) The program will compile without errors and will simply terminate without any output when run.

13.5 What will be the result of attempting to compile and run the following program?

```
class Extender extends Thread {
  public Extender() { }
  public Extender(Runnable runnable) {super(runnable);}
  public void run() {System.out.print("|Extender|");}
}

public class Implementer implements Runnable {
  public void run() {System.out.print("|Implementer|");}
```

PGJC3.book Page 624 Tuesday, November 25, 2008 9:10 PM


```
  public static void main(String[] args) {
    new Extender(new Implementer()).start();              // (1)
    new Extender().start();                               // (2)
    new Thread(new Implementer()).start();                // (3)
  }
}
```

Select the one correct answer.

(a) The program will fail to compile.
(b) The program will compile without errors and will print `|Extender|` twice and `|Implementer|` once, in some order, every time the program is run.
(c) The program will compile without errors and will print`|Extender|` once and `|Implementer|` twice, in some order, every time the program is run.
(d) The program will compile without errors and will print `|Extender|` once and `|Implementer|` once, in some order, every time the program is run
(e) The program will compile without errors and will simply terminate without any output when run.
(f) The program will compile without errors, and will print `|Extender|` once and `|Implementer|` once, in some order, and terminate because of an runtime error.

13.6 What will be the result of attempting to compile and run the following program?

```
class R1 implements Runnable {
  public void run() {
    System.out.print(Thread.currentThread().getName());
  }
}
public class R2 implements Runnable {
  public void run() {
    new Thread(new R1(),"|R1a|").run();
    new Thread(new R1(),"|R1b|").start();
    System.out.print(Thread.currentThread().getName());
  }

  public static void main(String[] args) {
    new Thread(new R2(),"|R2|").start();
  }
}
```

Select the one correct answer.

(a) The program will fail to compile.
(b) The program will compile without errors and will print `|R1a|` twice and `|R2|` once, in some order, every time the program is run.
(c) The program will compile without errors and will print`|R1b|` twice and `|R2|` once, in some order, every time the program is run.
(d) The program will compile without errors and will print `|R1b|` once and `|R2|` twice, in some order, every time the program is run.
(e) The program will compile without errors and will print `|R1a|` once, `|R1b|` once, and `|R2|` once, in some order, every time the program is run.

PGJC3.book Page 625 Tuesday, November 25, 2008 9:10 PM

13.7 What will be the result of attempting to compile and run the following program?

```
public class Threader extends Thread {
  Threader(String name) {
    super(name);
  }
  public void run() throws IllegalStateException {
    System.out.println(Thread.currentThread().getName());
    throw new IllegalStateException();
  }
  public static void main(String[] args) {
    new Threader("|T1|").start();
  }
}
```

Select the one correct answer.

(a) The program will fail to compile.
(b) The program will compile without errors, will print `|T1|`, and terminate normally every time the program is run.
(c) The program will compile without errors, will print`|T1|`, and throw an `IllegalStateException`, every time the program is run.
(d) None of the above.

13.8 What will be the result of attempting to compile and run the following program?

```
public class Worker extends Thread {
  public void run() {
    System.out.print("|work|");
  }
  public static void main(String[] args) {
    Worker worker = new Worker();
    worker.start();
    worker.run();
    worker.start();
  }
}
```

Select the one correct answer.

(a) The program will fail to compile.
(b) The program will compile without errors, will print `|work|` twice, and terminate normally every time the program is run.
(c) The program will compile without errors, will print`|work|` three times, and terminate normally every time the program is run.
(d) The program will compile without errors, will print`|work|` twice, and throw an `IllegalStateException`, every time the program is run.
(e) None of the above.

PGJC3.book Page 626 Tuesday, November 25, 2008 9:10 PM

13.5 Synchronization

Threads share the same memory space, i.e., they can share resources. However, there are critical situations where it is desirable that only one thread at a time has access to a shared resource. For example, crediting and debiting a shared bank account concurrently among several users without proper discipline, will jeopardize the integrity of the account data. Java provides high-level concepts for *synchronization* in order to control access to shared resources.

Locks

A *lock* (also called a *monitor*) is used to synchronize access to a shared resource. A lock can be associated with a shared resource. Threads gain access to a shared resource by first acquiring the lock associated with the resource. At any given time, at most one thread can hold the lock and thereby have access to the shared resource. A lock thus implements *mutual exclusion* (also known as *mutex*).

In Java, *all* objects have a lock—including arrays. This means that the lock from any Java object can be used to implement mutual exclusion. By associating a shared resource with a Java object and its lock, the object can act as a *guard*, ensuring synchronized access to the resource. Only one thread at a time can access the shared resource guarded by the *object lock*.

The object lock mechanism enforces the following rules of synchronization:

- A thread must *acquire* the object lock associated with a shared resource, before it can *enter* the shared resource. The runtime system ensures that no other thread can enter a shared resource if another thread already holds the object lock associated with it. If a thread cannot immediately acquire the object lock, it is *blocked*, i.e., it must wait for the lock to become available.
- When a thread *exits* a shared resource, the runtime system ensures that the object lock is also relinquished. If another thread is waiting for this object lock, it can try to acquire the lock in order to gain access to the shared resource.

It should be made clear that programs should not make any assumptions about the order in which threads are granted ownership of a lock.

Classes also have a class-specific lock that is analogous to the object lock. Such a lock is actually a lock on the `java.lang.Class` object associated with the class. Given a class `A`, the reference `A.class` denotes this unique `Class` object. The class lock can be used in much the same way as an object lock to implement mutual exclusion.

The keyword `synchronized` and the lock mechanism form the basis for implementing synchronized execution of code. There are two ways in which execution of code can be synchronized, by declaring *synchronized methods* or *synchronized code blocks*.

PGJC3.book Page 627 Tuesday, November 25, 2008 9:10 PM

Synchronized Methods

If the methods of an object should only be executed by one thread at a time, then the declaration of all such methods should be specified with the keyword `synchronized`. A thread wishing to execute a `synchronized` method must first obtain the object's lock (i.e., hold the lock) before it can enter the object to execute the method. This is simply achieved by calling the method. If the lock is already held by another thread, the calling thread waits. No particular action on the part of the program is necessary. A thread relinquishes the lock simply by returning from the `synchronized` method, allowing the next thread waiting for this lock to proceed.

Synchronized methods are useful in situations where methods can manipulate the state of an object in ways that can corrupt the state if executed concurrently. A stack implementation usually defines the two operations push and pop as synchronized, ensuring that pushing and popping of elements are mutually exclusive operations. If several threads were to share a stack, then one thread would, for example, not be able to push an element on the stack while another thread was popping the stack. The integrity of the stack is maintained in the face of several threads accessing the state of the same stack. This situation is illustrated by Example 13.3.

The code in Example 13.3 is intentionally non-generic in order to avoid generic considerations getting in the way. The `main()` method in class `Mutex` creates a stack at (6), which is used by the two threads created at (7) and (8). The two threads continually push and pop the stack. The non-`synchronized` `push()` and `pop()` methods at (2a) and (4a) intentionally sleep at (3) and (5), respectively, between an update and the use of the value in the field `topOfStack`. This setup increases the chances for the state of the stack being corrupted by one of the threads, while the other one is sleeping. The output from the program in Example 13.3 bears this out when the methods are not declared `synchronized`. Non-synchronized updating of the value in the field `topOfStack` between the two threads is a disaster waiting to happen. This is an example of what is called a *race condition*. It occurs when two or more threads simultaneously update the same value and, as a consequence, leave the value in an undefined or inconsistent state.

From the output shown in Example 13.3, we can see that the main thread exits right after creating and starting the threads. The threads push and pop the stack. The stack state eventually gets corrupted, resulting in an `ArrayOutOfBoundsException` in the Pusher thread. The uncaught exception results in the demise of the Pusher thread, but the Popper thread continues.

Running the program in Example 13.3 with the `synchronized` version of the `push()` and `pop()` methods at (2b) and (4b), respectively, avoids the race condition. The method `sleep()` does not relinquish any lock that the thread might have on the current object. It is only relinquished when the `synchronized` method exits, guaranteeing mutually exclusive push and pop operations on the stack.

PGJC3.book Page 628 Tuesday, November 25, 2008 9:10 PM

Example 13.3 *Mutual Exclusion*

```
class StackImpl {                                          // (1)
  private Object[] stackArray;
  private int topOfStack;

  public StackImpl(int capacity) {
    stackArray = new Object[capacity];
    topOfStack = -1;
  }

  public boolean push(Object element) {                    // (2a) non-synchronized
//public synchronized boolean push(Object element) {       // (2b) synchronized
    if (isFull()) return false;
    ++topOfStack;
    try { Thread.sleep(1000); } catch (Exception e) { } // (3) Sleep a little.
    stackArray[topOfStack] = element;
    return true;
  }

  public Object pop() {                                    // (4a) non-synchronized
//public synchronized Object pop() {                       // (4b) synchronized
    if (isEmpty()) return null;
    Object obj = stackArray[topOfStack];
    stackArray[topOfStack] = null;
    try { Thread.sleep(1000); } catch (Exception e) { } // (5) Sleep a little.
    topOfStack--;
    return obj;
  }

  public boolean isEmpty() { return topOfStack < 0; }
  public boolean isFull()  { return topOfStack >= stackArray.length - 1; }
}
//_____________________________________________________________________________
public class Mutex {
  public static void main(String[] args) {

    final StackImpl stack = new StackImpl(20);      // (6) Shared by the threads.

    (new Thread("Pusher") {                         // (7) Thread no. 1
      public void run() {
        for(;;) {
          System.out.println("Pushed: " + stack.push(2008));
        }
      }
    }).start();

    (new Thread("Popper") {                         // (8) Thread no. 2
      public void run() {
        for(;;) {
          System.out.println("Popped: " + stack.pop());
        }
      }
```

PGJC3.book Page 629 Tuesday, November 25, 2008 9:10 PM


```
        }).start();

        System.out.println("Exit from main().");
      }
    }
```

Possible output from the program when run with (2a) and (4a):

```
Exit from main().
...
Pushed: true
Popped: 2008
Popped: 2008
Popped: null
...
Popped: null
java.lang.ArrayIndexOutOfBoundsException: -1
        at StackImpl.push(Mutex.java:15)
        at Mutex$1.run(Mutex.java:41)
Popped: null
Popped: null
...
```

While a thread is inside a `synchronized` method of an object, all other threads that wish to execute this `synchronized` method or any other `synchronized` method of the object will have to wait. This restriction does not apply to the thread that already has the lock and is executing a `synchronized` method of the object. Such a method can invoke other `synchronized` methods of the object without being blocked. The non-`synchronized` methods of the object can always be called at any time by any thread.

Static methods synchronize on the class lock. Acquiring and relinquishing a class lock by a thread in order to execute a `static synchronized` method is analogous to that of an object lock for a `synchronized` instance method. A thread acquires the class lock before it can proceed with the execution of any static `synchronized` method in the class, blocking other threads wishing to execute any static synchronized methods in the same class. This does not apply to static, non-synchronized methods, which can be invoked at any time. A thread acquiring the lock of a class to execute a static `synchronized` method has no effect on any thread acquiring the lock on any object of the class to execute a `synchronized` instance method. In other words, synchronization of static methods in a class is independent from the synchronization of instance methods on objects of the class.

A subclass decides whether the new definition of an inherited `synchronized` method will remain `synchronized` in the subclass.

Synchronized Blocks

Whereas execution of `synchronized` methods of an object is synchronized on the lock of the object, the `synchronized` block allows execution of arbitrary code to be

PGJC3.book Page 630 Tuesday, November 25, 2008 9:10 PM

synchronized on the lock of an arbitrary object. The general form of the `synchronized` statement is as follows:

```
synchronized (<object reference expression>) { <code block> }
```

The *<object reference expression>* must evaluate to a non-`null` reference value, otherwise a `NullPointerException` is thrown. The code block is usually related to the object on which the synchronization is being done. This is analagous to a `synchronized` method, where the execution of the method is synchronized on the lock of the current object. The following code is equivalent to the `synchronized pop()` method at (4b) in Example 13.3:

```
public Object pop() {
  synchronized (this) {           // Synchronized block on current object
    // ...
  }
}
```

Once a thread has entered the code block after acquiring the lock on the specified object, no other thread will be able to execute the code block, or any other code requiring the same object lock, until the lock is relinquished. This happens when the execution of the code block completes normally or an uncaught exception is thrown. In contrast to `synchronized` methods, this mechanism allows fine-grained synchronization of code on arbitrary objects.

Object specification in the `synchronized` statement is mandatory. A class can choose to synchronize the execution of a part of a method by using the `this` reference and putting the relevant part of the method in the `synchronized` block. The braces of the block cannot be left out, even if the code block has just one statement.

```
class SmartClient {
  BankAccount account;
  // ...
  public void updateTransaction() {
    synchronized (account) {       // (1) synchronized block
      account.update();            // (2)
    }
  }
}
```

In the previous example, the code at (2) in the `synchronized` block at (1) is synchronized on the `BankAccount` object. If several threads were to concurrently execute the method `updateTransaction()` on an object of `SmartClient`, the statement at (2) would be executed by one thread at a time only after synchronizing on the `BankAccount` object associated with this particular instance of `SmartClient`.

Inner classes can access data in their enclosing context (see Section 8.1, p. 352). An inner object might need to synchronize on its associated outer object in order to ensure integrity of data in the latter. This is illustrated in the following code where the `synchronized` block at (5) uses the special form of the `this` reference to synchronize on the outer object associated with an object of the inner class. This setup ensures that a thread executing the method `setPi()` in an inner object can only

PGJC3.book Page 631 Tuesday, November 25, 2008 9:10 PM

access the `private double` field `myPi` at (2) in the synchronized block at (5) by first acquiring the lock on the associated outer object. If another thread has the lock of the associated outer object, the thread in the inner object has to wait for the lock to be relinquished before it can proceed with the execution of the synchronized block at (5). However, synchronizing on an inner object and on its associated outer object are independent of each other, unless enforced explicitly, as in the following code:

```
class Outer {                           // (1) Top-level Class
  private double myPi;                  // (2)

  protected class Inner {               // (3) Non-static member Class
    public void setPi() {               // (4)
      synchronized(Outer.this) {        // (5) Synchronized block on outer object
        myPi = Math.PI;                 // (6)
      }
    }
  }
}
```

Synchronized blocks can also be specified on a class lock:

```
synchronized (<class name>.class) { <code block> }
```

The block synchronizes on the lock of the object denoted by the reference *<class name>*`.class`. This object (of type `Class`) represents the class in the JVM. A `static synchronized` method `classAction()` in class `A` is equivalent to the following declaration:

```
static void classAction() {
  synchronized (A.class) {       // Synchronized block on class A
    // ...
  }
}
```

In summary, a thread can hold a lock on an object

- by executing a synchronized instance method of the object
- by executing the body of a synchronized block that synchronizes on the object
- by executing a synchronized static method of a class (in which case, the object is the `Class` object representing the class in the JVM)

Review Questions

13.9 Given the following program, which statements are guaranteed to be true?

```
public class ThreadedPrint {
  static Thread makeThread(final String id, boolean daemon) {
    Thread t = new Thread(id) {
      public void run() {
        System.out.println(id);
      }
    };
    t.setDaemon(daemon);
```

PGJC3.book Page 632 Tuesday, November 25, 2008 9:10 PM


```
    t.start();
    return t;
  }

  public static void main(String[] args) {
    Thread a = makeThread("A", false);
    Thread b = makeThread("B", true);
    System.out.print("End\n");
  }
}
```

Select the two correct answers.

(a) The letter `A` is always printed.
(b) The letter `B` is always printed.
(c) The letter `A` is never printed after `End`.
(d) The letter `B` is never printed after `End`.
(e) The program might print `B`, `End`, and `A`, in that order.

13.10 Given the following program, which alternatives would make good choices to synchronize on at (1)?

```
public class Preference {
  private int account1;
  private Integer account2;

  public void doIt() {
    final Double account3 = new Double(10e10);
    synchronized(/* ___(1)___ */) {
      System.out.print("doIt");
    }
  }
}
```

Select the two correct answers.

(a) Synchronize on `account1`.
(b) Synchronize on `account2`.
(c) Synchronize on `account3`.
(d) Synchronize on `this`.

13.11 Which statements are not true about the synchronized block?

Select the three correct answers.

(a) If the expression in a synchronized block evaluates to `null`, a `NullPointerException` will be thrown.
(b) The lock is only released if the execution of the block terminates normally.
(c) A thread cannot hold more than one lock at a time.
(d) Synchronized statements cannot be nested.
(e) The braces cannot be omitted even if there is only a single statement to execute in the block.

PGJC3.book Page 633 Tuesday, November 25, 2008 9:10 PM

13.12 Which statement is true?

Select the one correct answer.

(a) No two threads can concurrently execute synchronized methods on the same object.
(b) Methods declared `synchronized` should not be recursive, since the object lock will not allow new invocations of the method.
(c) Synchronized methods can only call other synchronized methods directly.
(d) Inside a `synchronized` method, one can assume that no other threads are currently executing any other methods in the same class.

13.13 Given the following program, which statement is true?

```
public class MyClass extends Thread {
  static Object lock1 = new Object();
  static Object lock2 = new Object();
  static volatile int i1, i2, j1, j2, k1, k2;
  public void run() { while (true) { doIt(); check(); } }
  void doIt() {
    synchronized(lock1) { i1++; }
    j1++;
    synchronized(lock2) { k1++; k2++; }
    j2++;
    synchronized(lock1) { i2++; }
  }
  void check() {
    if (i1 != i2) System.out.println("i");
    if (j1 != j2) System.out.println("j");
    if (k1 != k2) System.out.println("k");
  }
  public static void main(String[] args) {
    new MyClass().start();
    new MyClass().start();
  }
}
```

Select the one correct answer.

(a) The program will fail to compile.
(b) One cannot be certain whether any of the letters `i`, `j`, and `k` will be printed during execution.
(c) One can be certain that none of the letters `i`, `j`, and `k` will ever be printed during execution.
(d) One can be certain that the letters `i` and `k` will never be printed during execution.
(e) One can be certain that the letter `k` will never be printed during execution.

13.14 Given the following program, which code modifications will result in *both* threads being able to participate in printing one smiley (`:-)`) per line continuously?

```
public class Smiley extends Thread {

  public void run() {                                    // (1)
    while(true) {                                        // (2)
```

PGJC3.book Page 634 Tuesday, November 25, 2008 9:10 PM


```
            try {                                                    // (3)
                System.out.print(":");                               // (4)
                sleep(100);                                          // (5)
                System.out.print("-");                               // (6)
                sleep(100);                                          // (7)
                System.out.println(")");                             // (8)
                sleep(100);                                          // (9)
            } catch (InterruptedException e) {
              e.printStackTrace();
            }
          }
        }

        public static void main(String[] args) {
          new Smiley().start();
          new Smiley().start();
        }
    }
```

Select the two correct answers.

(a) Synchronize the `run()` method with the keyword `synchronized`, (1).
(b) Synchronize the `while` loop with a `synchronized(Smiley.class)` block, (2).
(c) Synchronize the `try-catch` construct with a `synchronized(Smiley.class)` block, (3).
(d) Synchronize the statements (4) to (9) with one `synchronized(Smiley.class)` block.
(e) Synchronize each statement (4), (6), and (8) individually with a `synchronized (Smiley.class)` block.
(f) None of the above will give the desired result.

13.6 Thread Transitions

Thread States

Understanding the life cycle of a thread is valuable when programming with threads. Threads can exist in different states. Just because a thread's `start()` method has been called, it does not mean that the thread has access to the CPU and can start executing straight away. Several factors determine how it will proceed.

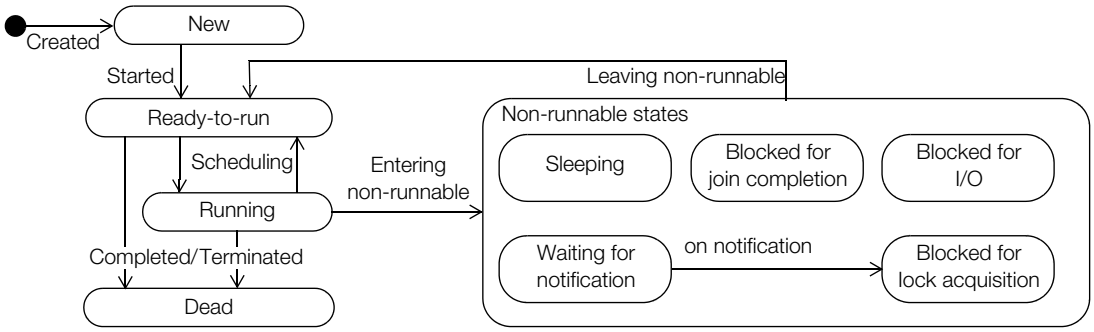
Figure 13.3 shows the states and the transitions in the life cycle of a thread.

- *New state*
  A thread has been *created*, but it has not yet *started*. A thread is started by calling its `start()` method, as explained in Section 13.4.
- *Ready-to-run state*
  A thread starts life in the Ready-to-run state (see p. 639).

PGJC3.book Page 635 Tuesday, November 25, 2008 9:10 PM

- *Running state*
  If a thread is in the Running state, it means that the thread is currently executing (see p. 639).
- *Dead state*
  Once in this state, the thread cannot ever run again (see p. 650).
- *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. 640).
  - *Blocked for I/O*: The thread waits for a *blocking* operation to complete (see p. 649).
  - *Blocked for join completion*: The thread awaits *completion* of another thread (see p. 647).
  - *Waiting for notification*: The thread awaits *notification* from another thread (see p. 640).
  - *Blocked for lock acquisition*: The thread waits to *acquire* the lock of an object (see p. 626).

Figure 13.3 *Thread States*



The `Thread` class provides the `getState()` method to determine the state of the current thread. The method returns a constant of type `Thread.State` (i.e., the type `State` is a static inner `enum` type declared in the `Thread` class). The correspondence between the states represented by its constants and the states shown in Figure 13.3 is summarized in Table 13.1.

PGJC3.book Page 636 Tuesday, November 25, 2008 9:10 PM

Table 13.1 *Thread States*

Constant in the `Thread.State` enum type	State in Figure 13.3	Description of the thread
`NEW`	*New*	Created but not yet started.
`RUNNABLE`	*Runnable*	Executing in the JVM.
`BLOCKED`	*Blocked for lock acquisition*	Blocked while waiting for a lock.
`WAITING`	*Waiting for notify,* *Blocked for join completion*	Waiting indefinitely for another thread to perform a particular action.
`TIMED_WAITING`	*Sleeping,* *Waiting for notify,* *Blocked for join completion*	Waiting for another thread to perform an action for up to a specified time.
`TERMINATED`	*Dead*	Completed execution.

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, i.e., it is not in the Dead state.

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

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

```
Thread.State getState()
```

This method returns the state of this thread (see Table 13.1). It should be used for monitoring the state and not for synchronizing control.

```
static void yield()
```

This method causes the current thread to temporarily pause its execution and, thereby, allow other threads to execute. It is up to the JVM to decide if and when this transition will take place.

```
static void sleep (long millisec) throws InterruptedException
```

The current thread sleeps for the specified time before it becomes eligible for running again.

PGJC3.book Page 637 Tuesday, November 25, 2008 9:10 PM


```
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`.

Example 13.4 illustrates transitions between thread states. A thread at (1) sleeps a little at (2) and then does some computation in a loop at (3), after which the thread terminates. The `main()` method monitors the thread in a loop at (4), printing the thread state returned by the `getState()` method. The output shows that the thread goes through the `RUNNABLE` state when the `run()` method starts to execute and then transits to the `TIMED_WAITING` state to sleep. On waking up, it computes the loop in the `RUNNABLE` state, and transits to the `TERMINATED` state when the `run()` method finishes.

Example 13.4 *Thread States*

```
public class ThreadStates {

  private static Thread t1 = new Thread("T1") {    // (1)
    public void run() {
      try {
        sleep(2);                                  // (2)
        for(int i = 10000; i > 0; i--);            // (3)
      } catch (InterruptedException ie){
        ie.printStackTrace();
      }
    }
  };

  public static void main(String[] args) {
    t1.start();
    while(true) {                                  // (4)
      Thread.State state = t1.getState();
      System.out.println(state);
      if (state == Thread.State.TERMINATED) break;
    }
  }
}
```

Possible output from the program:

```
RUNNABLE
TIMED_WAITING
```

PGJC3.book Page 638 Tuesday, November 25, 2008 9:10 PM


```
...
TIMED_WAITING
RUNNABLE
...
RUNNABLE
TERMINATED
```

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. The 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));
```

The `setPriority()` method is an *advisory* method, meaning that it provides a hint from the program to the JVM, which the JVM is in no way obliged to honor. The method can be used to fine-tune the *performance* of the program, but should not be relied upon for the *correctness* of the program.

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, 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.

PGJC3.book Page 639 Tuesday, November 25, 2008 9:10 PM

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

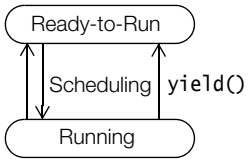
Running and Yielding

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, i.e., it waits for its turn to get CPU time. The thread scheduler decides which thread runs and for how long.

Figure 13.4 illustrates the transitions between the Ready-to-Run and Running states. A call to the static method `yield()`, defined in the `Thread` class, may cause the current thread in the Running state to transit to the Ready-to-run state, thus relinquishing the CPU. If this happens, the thread is then at the mercy of the thread scheduler as to when it will run again. It is possible that if there are no threads in the Ready-to-run state, this thread can continue executing. If there are other threads in the Ready-to-run state, their priorities can influence which thread gets to execute.

As with the `setPriority()` method, the `yield()` method is also an advisory method, and therefore comes with no guarantees that the JVM will carry out the call's bidding. A call to the `yield()` method does not affect any locks that the thread might hold.

Figure 13.4 *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 cancelling 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
    }
```

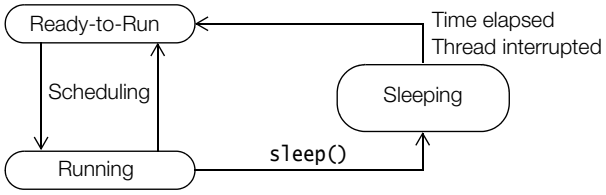
PGJC3.book Page 640 Tuesday, November 25, 2008 9:10 PM


```
    } catch (InterruptedException ie) {
      doCleaningUp();
    }
  }
```

Sleeping and Waking Up

Transitions by a thread to and from the Sleeping state are illustrated in Figure 13.5.

Figure 13.5 *Sleeping and Waking up*



A call to the static method `sleep()` in the `Thread` class will cause the currently running thread to temporarily pause its execution and transit to the Sleeping state. This 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 two overloaded versions of the `sleep()` method in the `Thread` class, allowing time to be specified in milliseconds, and additionally in nanoseconds.

Usage of the `sleep()` method is illustrated in Examples 13.1, 13.2, and 13.3.

Waiting and Notifying

Waiting and notifying provide means of communication between threads that *synchronize on the same object* (see Section 13.5, p. 626). 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 (in other words, in synchronized code), 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 current object.

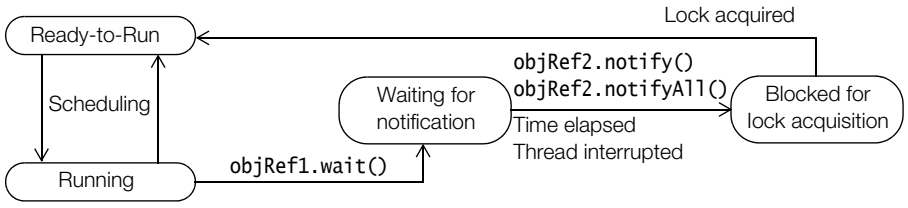
```
final void notify()
final void notifyAll()
```

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

PGJC3.book Page 641 Tuesday, November 25, 2008 9:10 PM

Communication between threads is facilitated by waiting and notifying, as illustrated by Figures 13.6 and 13.7. 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 13.6 *Waiting and Notifying*



Transition to the Waiting-for-notification state and relinquishing the object lock are completed as one *atomic* (non-interruptible) 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 relinquishes only the lock of the object on which the `wait()` method was invoked. It does not relinquish any other object locks that it might hold, which will 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.

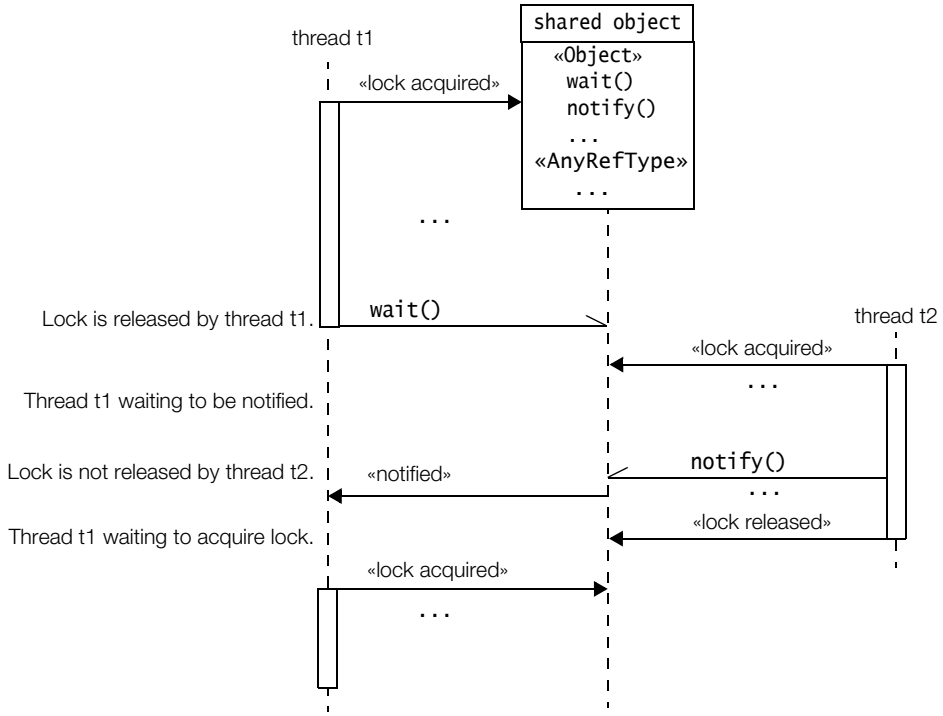
Figure 13.7 shows a thread t_1 that first acquires a lock on the shared object, and afterward invokes the `wait()` method on the shared object. This relinquishes the object lock and the thread t_1 awaits to be notified. While the thread t_1 is waiting, another thread t_2 can acquire the lock on the shared object for its own purposes.

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.

PGJC3.book Page 642 Tuesday, November 25, 2008 9:10 PM

Figure 13.7 *Thread Communication*



Notified

Invoking the `notify()` method on an object wakes up a single thread that is waiting for 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 at its own discretion, and the awakened thread will not be able to run until the notifying thread relinquishes the object lock.

When the notified thread obtains the object lock, it is enabled for execution, waiting in the Ready-to-run state for its turn to execute again. Finally, when it does execute, the call to the `wait()` method returns and the thread can continue with its execution.

From Figure 13.7 we see that thread t_2 does not relinquish the object lock when it invokes the `notify()` method. Thread t_1 is forced to wait in the Blocked-for-lock-acquisition state. It is shown no privileges and must compete with any other threads waiting for lock acquisition.

PGJC3.book Page 643 Tuesday, November 25, 2008 9:10 PM

A call to the `notify()` method has no effect if there are no threads in the wait set of the object.

In contrast to the `notify()` method, the `notifyAll()` method wakes up *all* threads in the wait set of the shared object. They will all transit to the Blocked-for-lock-acquisition state and contend for the object lock as explained earlier.

It should be stressed that a program should not make any assumptions about the order in which threads awaken in response to the `notify()` or `notifyAll()` method and transit to the Blocked-for-lock-acquisition state.

Timed-out

The `wait()` call specified the time the thread should wait before being timed out, if it was not awakened by being notified. The awakened thread competes in the usual manner to execute again. Note that the awakened thread has no way of knowing whether it was timed out or woken up by one of the notification methods.

Interrupted

This means that another thread invoked the `interrupt()` method on the waiting thread. The awakened thread is enabled as previously explained, but the return from the `wait()` call will result in an `InterruptedException` if and when the awakened thread finally gets a chance to run. The code invoking the `wait()` method must be prepared to handle this checked exception.

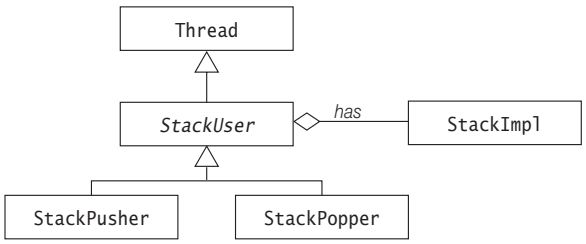
Using Wait and Notify

In Example 13.5, three threads are manipulating the same stack. Two of them are pushing elements on the stack, while the third one is popping elements off the stack. The class diagram for Example 13.5 is shown in Figure 13.8. The example comprises the following classes:

- The subclasses `StackPopper` at (9) and `StackPusher` at (10) extend the `abstract` superclass `StackUser` at (5).
- Class `StackUser`, which extends the `Thread` class, creates and starts each thread.
- Class `StackImpl` implements the `synchronized` methods `pop()` and `push()`.

Again, the code in Example 13.5 has not been generified in order to keep things simple.

Figure 13.8 *Stack Users*



PGJC3.book Page 644 Tuesday, November 25, 2008 9:10 PM

In Example 13.5, the field `topOfStack` in class `StackImpl` is declared `volatile`, so that read and write operations on this variable will access the *master* value of this variable, and not any copies, during runtime (see Section 4.10, p. 153).

Since the threads manipulate the same stack object, and the `push()` and `pop()` methods in the class `StackImpl` are `synchronized`, it means that the threads synchronize on the same object. In other words, the mutual exclusion of these operations is guaranteed on the same stack object.

Example 13.5 *Waiting and Notifying*

```
class StackImpl {
  private Object[] stackArray;
  private volatile int topOfStack;

  StackImpl (int capacity) {
    stackArray = new Object[capacity];
    topOfStack = -1;
  }

  public synchronized Object pop() {
    System.out.println(Thread.currentThread() + ": popping");
    while (isEmpty())
      try {
        System.out.println(Thread.currentThread() + ": waiting to pop");
        wait();                                          // (1)
      } catch (InterruptedException ie) {
        System.out.println(Thread.currentThread() + " interrupted.");
      }
    Object element = stackArray[topOfStack];
    stackArray[topOfStack--] = null;
    System.out.println(Thread.currentThread() +
                       ": notifying after popping");
    notify();                                            // (2)
    return element;
  }

  public synchronized void push(Object element) {
    System.out.println(Thread.currentThread() + ": pushing");
    while (isFull())
      try {
        System.out.println(Thread.currentThread() + ": waiting to push");
        wait();                                          // (3)
      } catch (InterruptedException ie) {
        System.out.println(Thread.currentThread() + " interrupted.");
      }
    stackArray[++topOfStack] = element;
    System.out.println(Thread.currentThread() +
                       ": notifying after pushing");
    notify();                                            // (4)
  }

  public boolean isFull() { return topOfStack >= stackArray.length -1; }
```

PGJC3.book Page 645 Tuesday, November 25, 2008 9:10 PM


```
  public boolean isEmpty() { return topOfStack < 0; }
}
//_____________________________________________________________________________
abstract class StackUser implements Runnable {       // (5) Stack user

  protected StackImpl stack;                         // (6)

  StackUser(String threadName, StackImpl stack) {
    this.stack = stack;
    Thread worker = new Thread(this, threadName);
    System.out.println(worker);
    worker.setDaemon(true);                          // (7) Daemon thread status
    worker.start();                                  // (8) Start the thread
  }
}
//_____________________________________________________________________________
class StackPopper extends StackUser {                // (9) Popper
  StackPopper(String threadName, StackImpl stack) {
    super(threadName, stack);
  }
  public void run() { while (true) stack.pop(); }
}
//_____________________________________________________________________________
class StackPusher extends StackUser {                // (10) Pusher
  StackPusher(String threadName, StackImpl stack) {
    super(threadName, stack);
  }
  public void run() { while (true) stack.push(2008); }
}
//_____________________________________________________________________________
public class WaitAndNotifyClient {
  public static void main(String[] args)
                    throws InterruptedException {    // (11)

    StackImpl stack = new StackImpl(5);              // Stack of capacity 5.

    new StackPusher("A", stack);
    new StackPusher("B", stack);
    new StackPopper("C", stack);
    System.out.println("Main Thread sleeping.");
    Thread.sleep(10);
    System.out.println("Exit from Main Thread.");
  }
}
```

Possible output from the program:

```
Thread[A,5,main]
Thread[B,5,main]
Thread[C,5,main]
Main Thread sleeping.
...
Thread[A,5,main]: pushing
Thread[A,5,main]: waiting to push
Thread[B,5,main]: pushing
Thread[B,5,main]: waiting to push
```

PGJC3.book Page 646 Tuesday, November 25, 2008 9:10 PM


```
Thread[C,5,main]: popping
Thread[C,5,main]: notifying after pop
Thread[A,5,main]: notifying after push
Thread[A,5,main]: pushing
Thread[A,5,main]: waiting to push
Thread[B,5,main]: waiting to push
Thread[C,5,main]: popping
Thread[C,5,main]: notifying after pop
Thread[A,5,main]: notifying after push
...
Thread[B,5,main]: notifying after push
...
Exit from Main Thread.
...
```

Example 13.5 illustrates how a thread waiting as a result of calling the `wait()` method on an object is notified by another thread calling the `notify()` method on the same object, in order for the first thread to start running again.

One usage of the `wait()` call is shown in Example 13.5 at (1) in the `synchronized pop()` method. When a thread executing this method on the `StackImpl` object finds that the stack is empty, it invokes the `wait()` method in order to wait for some thread to push something on this stack first.

Another use of the `wait()` call is shown at (3) in the `synchronized push()` method. When a thread executing this method on the `StackImpl` object finds that the stack is full, it invokes the `wait()` method to await some thread removing an element first, in order to make room for a push operation on the stack.

When a thread executing the `synchronized` method `push()` on the `StackImpl` object successfully pushes an element on the stack, it calls the `notify()` method at (4). The wait set of the `StackImpl` object contains all waiting threads that have earlier called the `wait()` method at either (1) or (3) on this `StackImpl` object. A single thread from the wait set is enabled for running. If this thread was executing a pop operation, it now has a chance of being successful because the stack is not empty at the moment. If this thread was executing a push operation, it can try again to see if there is room on the stack.

When a thread executing the `synchronized` method `pop()` on the `StackImpl` object successfully pops an element off the stack, it calls the `notify()` method at (2). Again assuming that the wait set of the `StackImpl` object is not empty, one thread from the set is arbitrarily chosen and enabled. If the notified thread was executing a pop operation, it can proceed to see if the stack still has an element to pop. If the notified thread was executing a push operation, it now has a chance of succeeding, because the stack is not full at the moment.

Note that the waiting condition at (1) for the pop operation is executed in a loop. A waiting thread that has been notified is not guaranteed to run right away. Before it gets to run, another thread may synchronize on the stack and empty it. If the notified thread was waiting to pop the stack, it would now incorrectly pop the stack,

PGJC3.book Page 647 Tuesday, November 25, 2008 9:10 PM

because the condition was not tested after notification. The loop ensures that the condition is always tested after notification, sending the thread back to the Waiting-on-notification state if the condition is not met. To avert the analogous danger of pushing on a full stack, the waiting condition at (3) for the push operation is also executed in a loop.

The behavior of each thread can be traced in the output from Example 13.5. Each push-and-pop operation can be traced by a sequence consisting of the name of the operation to be performed, followed by zero or more wait messages, and concluding with a notification after the operation is done. For example, thread `A` performs two pushes as shown in the output from the program:

```
Thread[A,5,main]: pushing
Thread[A,5,main]: waiting to push
...
Thread[A,5,main]: notifying after push
Thread[A,5,main]: pushing
Thread[A,5,main]: waiting to push
...
Thread[A,5,main]: notifying after push
```

Thread `B` is shown doing one push:

```
Thread[B,5,main]: pushing
Thread[B,5,main]: waiting to push
...
Thread[B,5,main]: notifying after push
```

Whereas thread `C` pops the stack twice without any waiting:

```
Thread[C,5,main]: popping
Thread[C,5,main]: notifying after pop
...
Thread[C,5,main]: popping
Thread[C,5,main]: notifying after pop
```

When the operations are interweaved, the output clearly shows that the pushers wait when the stack is full, and only push after the stack is popped.

The three threads created are daemon threads. Their status is set at (7). They will be terminated if they have not completed when the main thread dies, thereby stopping the execution of the program.

Joining

A thread can invoke the overloaded method `join()` on another thread in order to wait for the other thread to complete its execution before continuing, i.e., the first thread waits for the second thread to *join it after completion.* A running thread t_1 invokes the method `join()` on a thread t_2. The `join()` call has no effect if thread t_2 has already completed. If thread t_2 is still alive, thread t_1 transits to the Blocked-for-join-completion state. Thread t_1 waits in this state until one of these events occur (see Figure 13.9):

PGJC3.book Page 648 Tuesday, November 25, 2008 9:10 PM

- Thread t_2 completes.

  In this case thread t_1 moves to the Ready-to-run state, and when it gets to run, it will continue normally after the call to the `join()` method.

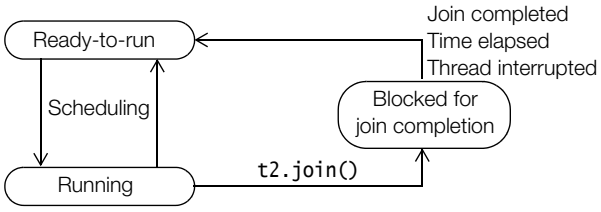
- Thread t_1 is timed out.

  The time specified in the argument of the `join()` method call has elapsed without thread t_2 completing. In this case as well, thread t_1 transits to the Ready-to-run state. When it gets to run, it will continue normally after the call to the `join()` method.

- Thread t_1 is interrupted.

  Some thread interrupted thread t_1 while thread t_1 was waiting for join completion. Thread t_1 transits to the Ready-to-run state, but when it gets to execute, it will now throw an `InterruptedException`.

Figure 13.9 *Joining of Threads*



Example 13.6 illustrates joining of threads. The `AnotherClient` class below uses the `Counter` class, which extends the `Thread` class from Example 13.2. It creates two threads that are enabled for execution. The main thread invokes the `join()` method on the `Counter A` thread. If the `Counter A` thread has not already completed, the main thread transits to the Blocked-for-join-completion state. When the `Counter A` thread completes, the main thread will be enabled for running. Once the main thread is running, it continues with execution after (5). A parent thread can call the `isAlive()` method to find out whether its child threads are alive before terminating itself. The call to the `isAlive()` method on the `Counter A` thread at (6) correctly reports that the `Counter A` thread is not alive. A similar scenario transpires between the main thread and the `Counter B` thread. The main thread passes through the Blocked-for-join-completion state twice at the most.

Example 13.6 *Joining of Threads*

```
class Counter extends Thread { /* See Example 13.2. */ }
//_____________________________________________________________________________
public class AnotherClient {
  public static void main(String[] args) {

    Counter counterA = new Counter("Counter A");
```

PGJC3.book Page 649 Tuesday, November 25, 2008 9:10 PM


```
    Counter counterB = new Counter("Counter B");

    try {
      System.out.println("Wait for the child threads to finish.");
      counterA.join();                                  // (5)
      if (!counterA.isAlive())                          // (6)
        System.out.println("Counter A not alive.");
      counterB.join();                                  // (7)
      if (!counterB.isAlive())                          // (8)
        System.out.println("Counter B not alive.");
    } catch (InterruptedException ie) {
      System.out.println("Main Thread interrupted.");
    }
    System.out.println("Exit from Main Thread.");
  }
}
```

Possible output from the program:

```
Thread[Counter A,5,main]
Thread[Counter B,5,main]
Wait for the child threads to finish.
Counter A: 0
Counter B: 0
Counter A: 1
Counter B: 1
Counter A: 2
Counter B: 2
Counter A: 3
Counter B: 3
Counter A: 4
Counter B: 4
Exit from Counter A.
Counter A not alive.
Exit from Counter B.
Counter B not alive.
Exit from Main Thread.
```

Blocking for I/O

A running thread, on executing a *blocking operation* requiring a resource (like a call to an I/O method), will transit to the Blocked-for-I/O state. The blocking operation must complete before the thread can proceed to the Ready-to-run state. An example is a thread reading from the standard input terminal which blocks until input is provided:

```
int input = System.in.read();
```

PGJC3.book Page 650 Tuesday, November 25, 2008 9:10 PM

Thread Termination

A thread can transit to the Dead state from the Running or the Ready-to-run states. The thread dies when it completes its `run()` method, either by returning normally or by throwing an exception. Once in this state, the thread cannot be resurrected. There is no way the thread can be enabled for running again, not even by calling the `start()` method again on the thread object.

Example 13.7 illustrates a typical scenario where a thread can be controlled by one or more threads. Work is performed by a loop body, which the thread executes continually. It should be possible for other threads to start and stop the *worker* thread. This functionality is implemented by the class `Worker` at (1), which has a private field `theThread` declared at (2) to keep track of the `Thread` object executing its `run()` method.

The `kickStart()` method at (3) in class `Worker` creates and starts a thread if one is not already running. It is not enough to just call the `start()` method on a thread that has terminated. A new `Thread` object must be created first. The `terminate()` method at (4) sets the field `theThread` to `null`. Note that this does not affect any `Thread` object that might have been referenced by the reference `theThread`. The runtime system maintains any such `Thread` object; therefore, changing one of its references does not affect the object.

The `run()` method at (5) has a loop whose execution is controlled by a special condition. The condition tests to see whether the `Thread` object referenced by the reference `theThread` and the `Thread` object executing now, are one and the same. This is bound to be the case if the reference `theThread` has the same reference value that it was assigned when the thread was created and started in the `kickStart()` method. The condition will then be `true`, and the body of the loop will execute. However, if the value in the reference `theThread` has changed, the condition will be `false`. In that case, the loop will not execute, the `run()` method will complete and the thread will terminate.

A client can control the thread implemented by the class `Worker`, using the `kickStart()` and the `terminate()` methods. The client is able to terminate the running thread at the start of the next iteration of the loop body by calling the `terminate()` method that changes the value of the `theThread` reference to `null`.

In Example 13.7, a `Worker` object is first created at (8) and a thread started on this `Worker` object at (9). The main thread invokes the `sleep()` method at (10) to temporarily cease its execution for 2 milliseconds giving the thread of the `Worker` object a chance to run. The main thread, when it is executing again, terminates the thread of the `Worker` object at (11), as explained earlier. This simple scenario can be generalized where several threads, sharing a single `Worker` object, could be starting and stopping the thread of the `Worker` object.

PGJC3.book Page 651 Tuesday, November 25, 2008 9:10 PM

Example 13.7 *Thread Termination*

```
class Worker implements Runnable {                                  // (1)
  private volatile Thread theThread;                                // (2)

  public void kickStart() {                                         // (3)
    if (theThread == null) {
      theThread = new Thread(this);
      theThread.start();
    }
  }

  public void terminate() {                                         // (4)
    theThread = null;
  }

  public void run() {                                               // (5)
    while (theThread == Thread.currentThread()) {                   // (6)
      System.out.println("Going around in loops.");
    }
  }
}
//_____________________________________________________________________________
public class Controller {
  public static void main(String[] args) {                          // (7)
    Worker worker = new Worker();                                   // (8)
    worker.kickStart();                                             // (9)
    try {
      Thread.sleep(2);                                              // (10)
    } catch(InterruptedException ie) {
      ie.printStackTrace();
    }
    worker.terminate();                                             // (11)
  }
}
```

Possible output from the program:

```
Going around in loops.
Going around in loops.
Going around in loops.
Going around in loops.
Going around in loops.
```

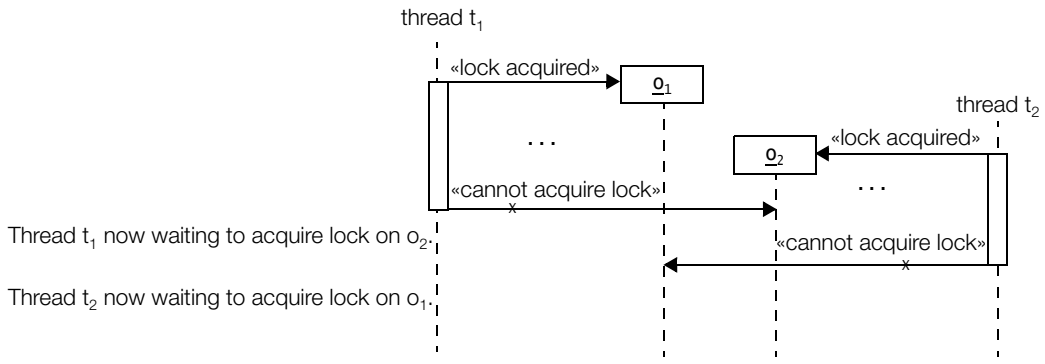
Deadlocks

A deadlock is a situation where a thread is waiting for an object lock that another thread holds, and this second thread is waiting for an object lock that the first thread holds. Since each thread is waiting for the other thread to relinquish a lock, they both remain waiting forever in the Blocked-for-lock-acquisition state. The threads are said to be *deadlocked*.

PGJC3.book Page 652 Tuesday, November 25, 2008 9:10 PM

A deadlock is depicted in Figure 13.10. Thread t_1 has a lock on object o_1, but cannot acquire the lock on object o_2. Thread t_2 has a lock on object o_2, but cannot acquire the lock on object o_1. They can only proceed if one of them relinquishes a lock the other one wants, which is never going to happen.

Figure 13.10 *Deadlock*



The situation in Figure 13.10 is implemented in Example 13.8. Thread `t1` at (3) tries to synchronize at (4) and (5), first on string `o1` at (1) then on string `o2` at (2), respectively. The thread `t2` at (6) does the opposite. It synchronizes at (7) and (8), first on string `o2` then on string `o1`, respectively. A deadlock can occur as explained previously.

However, the potential of deadlock in the situation in Example 13.8 is easy to fix. If the two threads acquire the locks on the objects in the same order, then mutual lock dependency is avoided and a deadlock can never occur. This means having the same locking order at (4) and (5) as at (7) and (8). In general, the cause of a deadlock is not always easy to discover, let alone easy to fix.

Example 13.8 *Deadlock*

```
public class DeadLockDanger {

  String o1 = "Lock " ;                          // (1)
  String o2 = "Step ";                           // (2)

  Thread t1 = (new Thread("Printer1") {          // (3)
    public void run() {
      while(true) {
        synchronized(o1) {                       // (4)
          synchronized(o2) {                     // (5)
            System.out.println(o1 + o2);
          }
        }
```

PGJC3.book Page 653 Tuesday, November 25, 2008 9:10 PM


```
        }
      }
    });

    Thread t2 = (new Thread("Printer2") {        // (6)
      public void run() {
        while(true) {
          synchronized(o2) {                     // (7)
            synchronized(o1) {                   // (8)
              System.out.println(o2 + o1);
            }
          }
        }
      }
    });

    public static void main(String[] args) {
      DeadLockDanger dld = new DeadLockDanger();
      dld.t1.start();
      dld.t2.start();
    }
  }
```

Possible output from the program:

```
...
Lock Step
Lock Step
Lock Step
Lock Step
Lock Step
_
```

Review Questions

13.15 Which one of these events will cause a thread to die?

Select the one correct answer.

(a) The method `sleep()` is called.
(b) The method `wait()` is called.
(c) Execution of the `start()` method ends.
(d) Execution of the `run()` method ends.
(e) Execution of the thread's constructor ends.

13.16 Which statements are true about the following code?

```
public class Joining {
  static Thread createThread(final int i, final Thread t1) {
    Thread t2 = new Thread() {
      public void run() {
        System.out.println(i+1);
```

PGJC3.book Page 654 Tuesday, November 25, 2008 9:10 PM


```
        try {
          t1.join();
        } catch (InterruptedException ie) {
        }
        System.out.println(i+2);
      }
    };
    System.out.println(i+3);
    t2.start();
    System.out.println(i+4);
    return t2;
  }

  public static void main(String[] args) {
    createThread(10, createThread(20, Thread.currentThread()));
  }
}
```

Select the two correct answers.

(a) The first number printed is 13.
(b) The number 14 is printed before the number 22.
(c) The number 24 is printed before the number 21.
(d) The last number printed is 12.
(e) The number 11 is printed before the number 23.

13.17 Which statements are true about the following program?

```
public class ThreadAPI {
  private static Thread t1 = new Thread("T1") {
    public void run() {
      try { wait(1000); } catch (InterruptedException ie){}
  }};

  private static Thread t2 = new Thread("T2") {
    public void run() {
      notify();
  }};

  private static Thread t3 = new Thread("T3") {
    public void run() {
      yield();
  }};

  private static Thread t4 = new Thread("T4") {
    public void run() {
      try { sleep(100); } catch (InterruptedException ie){}
  }};

  public static void main(String[] args) {
    t1.start(); t2.start(); t3.start(); t4.start();
    try { t4.join(); } catch (InterruptedException ie){}
  }
}
```

PGJC3.book Page 655 Tuesday, November 25, 2008 9:10 PM

Select the three correct answers.

(a) The program will compile and will run and terminate normally.
(b) The program will compile but thread `t1` will throw an exception.
(c) The program will compile but thread `t2` will throw an exception.
(d) The program will compile but thread `t3` will throw an exception.
(e) Enclosing the call to the `sleep()` method in a `try-catch` construct in thread `t4` is unnecessary.
(f) Enclosing the call to the `join()` method in a `try-catch` construct in the main thread is necessary.

13.18 Which code, when inserted at (1), will result in the program compiling and printing `Done` on the standard input stream, and then all threads terminating normally?

```
public class RunningThreads {

  private static Thread t1 = new Thread("T1") {
    public void run() {
      synchronized(RunningThreads.class) {
        try {
          // (1) INSERT CODE HERE ...
        } catch (InterruptedException ie){
          ie.printStackTrace();
        }
        System.out.println("Done");
      }}};

  public static void main(String[] args) {
    t1.start();
    try {
      t1.join();
    } catch (InterruptedException ie){
      ie.printStackTrace();
    }
  }
}
```

Select the two correct answers.

(a) `wait();`
(b) `wait(100);`
(c) `RunningThreads.class.wait();`
(d) `RunningThreads.class.wait(100);`
(e) `yield();`
(f) `sleep(100);`

13.19 What can be guaranteed by calling the method `yield()`?

Select the one correct answer.

(a) All lower priority threads will be granted CPU time.
(b) The current thread will sleep for some time while some other threads run.
(c) The current thread will not continue until other threads have terminated.

PGJC3.book Page 656 Tuesday, November 25, 2008 9:10 PM

(d) The thread will wait until it is notified.
(e) None of the above.

13.20 In which class or interface is the `notify()` method defined?

Select the one correct answer.

(a) `Thread`
(b) `Object`
(c) `Appendable`
(d) `Runnable`

13.21 How can the priority of a thread be set?

Select the one correct answer.

(a) By using the `setPriority()` method in the `Thread` class.
(b) By passing the priority as an argument to a constructor of the `Thread` class.
(c) Both of the above.
(d) None of the above.

13.22 Which statements are true about locks?

Select the two correct answers.

(a) A thread can hold more than one lock at a time.
(b) Invoking `wait()` on a `Thread` object will relinquish all locks held by the thread.
(c) Invoking `wait()` on an object whose lock is held by the current thread will relinquish the lock.
(d) Invoking `notify()` on a object whose lock is held by the current thread will relinquish the lock.
(e) Multiple threads can hold the same lock at the same time.

13.23 What will be the result of invoking the `wait()` method on an object without ensuring that the current thread holds the lock of the object?

Select the one correct answer.

(a) The code will fail to compile.
(b) Nothing special will happen.
(c) An `IllegalMonitorStateException` will be thrown if the `wait()` method is called while the current thread does not hold the lock of the object.
(d) The thread will be blocked until it gains the lock of the object.

13.24 Which of these are plausible reasons why a thread might be alive, but still not be running?

Select the four correct answers.

(a) The thread is waiting for some condition as a result of a `wait()` call.
(b) The execution has reached the end of the `run()` method.
(c) The thread is waiting to acquire the lock of an object in order to execute a certain method on that object.

PGJC3.book Page 657 Tuesday, November 25, 2008 9:10 PM

(d) The thread does not have the highest priority and is currently not executing.
(e) The thread is sleeping as a result of a call to the `sleep()` method.

13.25 What will the following program print when compiled and run?

```
public class Tank {
  private boolean isEmpty = true;

  public synchronized void emptying() {
    pause(true);
    isEmpty = !isEmpty;
    System.out.println("emptying");
    notify();
  }

  public synchronized void filling() {
    pause(false);
    isEmpty = !isEmpty;
    System.out.println("filling");
    notify();
  }

  private void pause(boolean flag) {
    while(flag ? isEmpty : !isEmpty) {
      try {
        wait();
      } catch (InterruptedException ie) {
        System.out.println(Thread.currentThread() + " interrupted.");
      }
    }
  }

  public static void main(String[] args) {
    final Tank token = new Tank();
    (new Thread("A") { public void run() {for(;;) token.emptying();}}).start();
    (new Thread("B") { public void run() {for(;;) token.filling();}}).start();
  }
}
```

Select the one correct answer.

(a) The program will compile and continue running once started, but will not print anything.
(b) The program will compile and continue running once started, printing only the string `"emptying"`.
(c) The program will compile and continue running once started, printing only the string `"filling"`.
(d) The program will compile and continue running once started, always printing the string `"filling"` followed by the string `"emptying"`.
(e) The program will compile and continue running once started, printing the strings `"filling"` and `"emptying"` in some order.

PGJC3.book Page 658 Tuesday, November 25, 2008 9:10 PM

13.26 What will the following program print when compiled and run?

```
public class Syncher2 {
  final static int[] intArray = new int[2];

  private static void pause() {
    while(intArray[0] == 0) {
      try { intArray.wait(); }
      catch (InterruptedException ie) {
        System.out.println(Thread.currentThread() + " interrupted.");
      }
    }
  }

  public static void main (String[] args) {

    Thread runner = new Thread() {
      public void run() {
        synchronized (intArray) {
          pause();
          System.out.println(intArray[0] + intArray[1]);
    }}};

    runner.start();
    intArray[0] = intArray[1] = 10;
    synchronized(intArray) {
      intArray.notify();
    }
  }
}
```

Select the one correct answer.

(a) The program will not compile.
(b) The program will compile, but throw an exception when run.
(c) The program will compile and continue running once started, but will not print anything.
(d) The program will compile and print `0` and terminate normally, when run.
(e) The program will compile and print `20` and terminate normally, when run.
(f) The program will compileand print some other number than `0` or `20`, and terminate normally, when run.

Chapter Summary

The following information was included in this chapter:

- creating threads by extending the `Thread` class or implementing the `Runnable` interface
- writing synchronized code using synchronized methods and synchronized blocks to achieve mutually exclusive access to shared resources

PGJC3.book Page 659 Tuesday, November 25, 2008 9:10 PM

- understanding thread states, and the transitions between them, and thread communication
- understanding which thread behavior a program must not take as guaranteed

Programming Exercises

13.1 Implement three classes: `Storage`, `Counter`, and `Printer`. The `Storage` class should store an integer. The `Counter` class should create a thread that starts counting from 0 (0, 1, 2, 3, ...) and stores each value in the `Storage` class. The `Printer` class should create a thread that keeps reading the value in the `Storage` class and printing it.

Write a program that creates an instance of the `Storage` class and sets up a `Counter` and a `Printer` object to operate on it.

13.2 Modify the program from the previous exercise to ensure that each number is printed exactly once, by adding suitable synchronization.