

### **\*\*\*Chapter in the Advanced Version**

## **Part**

## **VI**

### **Threads, Multimedia, and Internationalization**

This part of the book is devoted to three unique and useful features in Java. The subject treated include the use of multithreading to make programs more responsive and interactive in Chapter 19, the incorporation of sound and images to make programs user-friendly in Chapter 20, and the use of internationalization support to develop projects for international audiences in Chapter 21.

Chapter 19

Multithreading

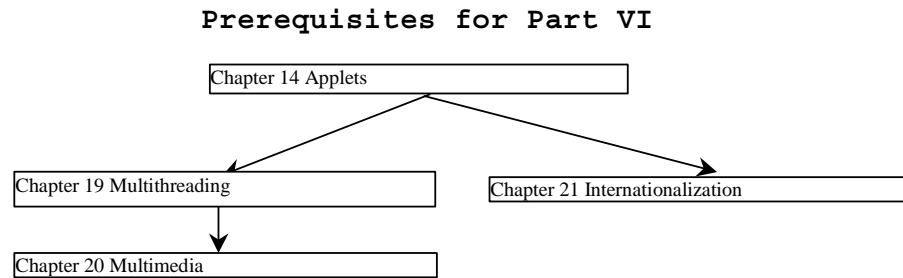
Chapter 20

Multimedia: Audio, Images, and Animations

Chapter 21

Internationalization

**\*\*\*PD: Put this in the center of the back of the Part-opening page. AU**



## CHAPTER

### 19

## Multithreading

### Objectives

[BL]To understand the concept of multithreading and apply it to develop animation.

[BL]To develop thread classes by extending the Thread class.

[BL]To develop thread classes by implementing the Runnable interface in cases of multiple inheritance.

[BL]To describe the life-cycle of thread states and set thread priorities.

[BL]To know how to control threads: starting, stopping, suspending, and resuming threads.

[BL]To use synchronized methods or block to synchronize threads to avoid race conditions.

[BL]To use wait(), notify(), and notifyAll() to facilitate thread cooperation.

[BL]To use the resource ordering technique to avoid deadlock.

[BL]To use the Timer class to simplify the control of Java animations.

[BL]To display the completion status of a task using

JProgressBar.

## 19.1 Introduction

One of the important features of Java is its built-in support for multithreading. *Multithreading* is the capability of running multiple tasks concurrently within a program. In many programming languages, you have to invoke system-dependent procedures and functions to implement multithreading. This chapter introduces the concepts of threads and how to develop multithreading programs in Java.

## 19.2 Thread Concepts

A *thread* is the flow of execution, from beginning to end, of a task in a program. With Java, you can launch multiple threads from a program concurrently. These threads can be executed simultaneously in multiprocessor systems, as shown in Figure 19.1.

\*\*\* Same as Fig 18.1 in introjb3e p784



**Figure 19.1**

*Here, multiple threads are running on multiple CPUs.*

In single-processor systems, as shown in Figure 19.2, the multiple threads share CPU time, and the operating system is responsible for scheduling and allocating resources to them. This arrangement is practical because most of the time the CPU is idle. It does nothing, for example, while waiting for the user to enter data.

\*\*\*Same as Fig 18.2 in introjb3e p784



**Figure 19.2**

*Here, multiple threads share a single CPU.*

Multithreading can make your program more responsive and interactive, as well as enhance performance. For example, a good word processor lets you print or save a file while you are typing. In some cases, multithreaded programs run faster than single-threaded programs even on single-processor

systems. Java provides exceptionally good support for creating and running threads and for locking resources to prevent conflicts.

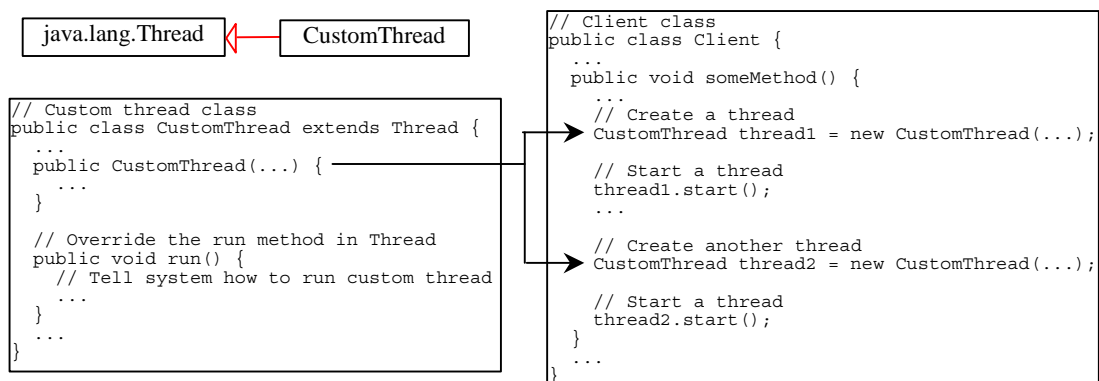
When your program executes as an application, the Java interpreter starts a thread for the main method. When your program executes as an applet, the Web browser starts a thread to run the applet. You can create additional threads to run concurrent tasks in the program. Each new thread is an object of a class that implements the Runnable interface or extends a class that implements the Runnable interface. This new object is referred to as a *runnable object*.

You can create threads either by extending the Thread class or by implementing the Runnable interface. Both Thread and Runnable are defined in the java.lang package. Thread actually implements Runnable. In the following sections, you will learn how to use the Thread class and the Runnable interface to write multithreaded programs.

### 19.3 Creating Threads by Extending the Thread Class

The Thread class contains the constructors for creating threads, as well as many useful methods for controlling threads. To create and run a thread, first define a class that extends the Thread class. Your thread class must override the run() method, which tells the system how the thread will be executed when it runs. You can then create an object running on the thread.

A template for developing a custom thread class that extends the Thread class and for creating threads from the custom thread class is shown in Figure 19.3. The thread is a runnable object created from the CustomThread class. The start method tells the system that the thread is ready to run.



**Figure 19.3**

Define a thread class by extending the Thread class.

### Example 19.1

## Using the Thread Class to Create and Launch Threads

## Problem

Write a program that creates and runs three threads:

[BL]The first thread prints the letter a one hundred times.

[BL]The second thread prints the letter *b* one hundred times.

[BL]The third thread prints the integers 1 through 100.

### Solution

The program has three independent threads. To run them concurrently, it needs to create a runnable object for each thread. Because the first two threads have similar functionality, they can be defined in one thread class.

The program is given here, and its output is shown in Figure 19.4.

```
C:\book>java TestThread
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16ababababababababaaaaa
aaaaaaaaaaaaabbbbbbbbbbbbbbbbbbaaaaaaaaaaaaaabbbbbbbbbbbbbbb
bbbaaaaaaaaaaaaaabbbbbbbbbbbbbbbbbbaaaaaaaaaaaaaabbbbbbbbbbb
bbbbbbbaaaaaaaaaaaaaabbbbbbbbbbbbbbbbbbaaaaaaaaaaaaaabbbbbbbb
17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36
37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56
57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76
77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96
97 98 99 100
C:\book>
```

Figure 19.4

*Threads printA, printB, and print100 are executed simultaneously to display the letter a one hundred times, the letter b one hundred times, and the numbers from 1 to 100.*

\*\*\*PD: Please add line numbers in the following code\*\*\*

```

// TestThread.java: Define threads using the Thread class
public class TestThread {
    /** Main method */
    public static void main(String[] args) {
        // Create threads
        PrintChar printA = new PrintChar('a', 100);
        PrintChar printB = new PrintChar('b', 100);
        PrintNum print100 = new PrintNum(100);

        // Start threads
        print100.start();
        printA.start();
        printB.start();
    }
}

// The thread class for printing a specified character
// in specified times
class PrintChar extends Thread {
    private char charToPrint; // The character to print
    private int times; // The times to repeat

    /** Construct a thread with specified character and number of
        times to print the character
    */
    public PrintChar(char c, int t) {
        charToPrint = c;
        times = t;
    }

    /** Override the run() method to tell the system
        what the thread will do
    */
    public void run() {
        for (int i = 0; i < times; i++)
            System.out.print(charToPrint);
    }
}

// The thread class for printing number from 1 to n for a given n
class PrintNum extends Thread {
    private int lastNum;

    /** Construct a thread for print 1, 2, ... i */
    public PrintNum(int n) {
        lastNum = n;
    }

    /** Tell the thread how to run */
    public void run() {
        for (int i = 1; i <= lastNum; i++) {
            System.out.print(" " + i);
        }
    }
}

```

## Review

If you run this program on a multiple-CPU system, all three threads will execute simultaneously. If you run the program on a single-CPU system, the three threads will share the CPU and take turns printing letters and numbers on the console. This is known as *time-sharing*.

The program creates thread classes by extending the Thread class. The PrintChar class (Lines 19-38), derived from the Thread class, overrides the run() method (Lines 34-37) with the print-character action. This class provides a framework for printing any single character a given number of times. The runnable objects printA and printB are instances of

the user-defined thread class PrintChar.

The PrintNum class (Lines 41-54) overrides the run() method (Lines 50-53) with the print-number action. This class provides a framework for printing numbers from 1 to *n*, for any integer *n*. The runnable object print100 is an instance of the user-defined thread class printNum.

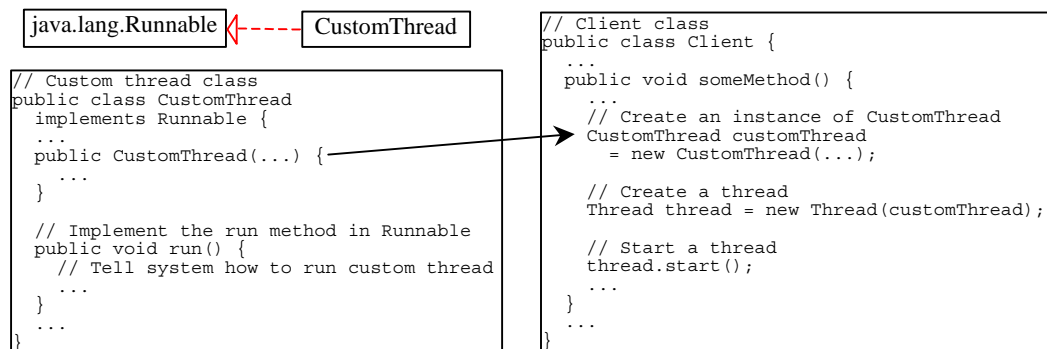
In the client program, the program creates a thread, printA, for printing the letter *a*, and a thread, printB, for printing the letter *b*. Both are objects of the PrintChar class. The print100 thread object is created from the PrintNum class.

The start() method (Lines 11-13) is invoked to start a thread that causes the run() method to execute. When the run() method completes, the threads terminate.

#### 19.4 Creating Threads by Implementing the Runnable Interface

In the preceding section, you created and ran a thread by declaring a user thread class that extends the Thread class. This approach works well if the user thread class inherits only from the Thread class, but not if it inherits multiple classes, as in the case of an applet. To inherit multiple classes, you have to implement interfaces. Java provides the Runnable interface as an alternative to the Thread class.

The Runnable interface is rather simple. All it contains is the run method. You need to implement this method to tell the system how your thread is going to run. A template for developing a custom thread class that implements the Runnable interface and for creating threads from the custom thread class is shown in Figure 19.5.



**Figure 19.5**

Define a thread class by implementing the Runnable interface.

To start a new thread with the Runnable interface, you must first create an instance of the class that implements the Runnable interface, then use the Thread class constructor to construct a thread.

The following example demonstrates how to create threads using the Runnable interface.

### **Example 19.2**

#### **Using the Runnable Interface to Create and Launch Threads**

##### **Problem**

Modify Example 19.1, "Using the Thread Class to Create and Launch Threads," to create and run the same threads using the Runnable interface.

##### **Solution**

The following code gives the solution to the problem.

**\*\*\*PD: Please add line numbers in the following code\*\*\***

```
// TestRunnable.java: Define threads using the Runnable interface
public class TestRunnable {
    // Create threads
    Thread printA = new Thread(new PrintChar('a', 100));
    Thread printB = new Thread(new PrintChar('b', 100));
    Thread print100 = new Thread(new PrintNum(100));

    /** Main method */
    public static void main(String[] args) {
        new TestRunnable();
    }

    /** Default constructor */
    public TestRunnable() {
        // Start threads
        print100.start();
        printA.start();
        printB.start();
    }

    // The thread class for printing a specified character
    // in specified times
    class PrintChar implements Runnable {
        private char charToPrint; // The character to print
        private int times; // The times to repeat

        /** Construct a thread with specified character and number of
         * times to print the character
         */
        public PrintChar(char c, int t) {
            charToPrint = c;
            times = t;
        }

        /** Override the run() method to tell the system
         * what the thread will do
         */
    }
}
```



```

    public void run() {
        for (int i = 0; i < times; i++)
            System.out.print(charToPrint);
    }
}

// The thread class for printing number from 1 to n for a given n
class PrintNum implements Runnable {
    private int lastNum;

    /** Construct a thread for print 1, 2, ... i */
    public PrintNum(int n) {
        lastNum = n;
    }

    /** Tell the thread how to run */
    public void run() {
        for (int i = 1; i <= lastNum; i++)
            System.out.print(" " + i);
    }
}

```

## Review

The program creates thread classes by implementing the Runnable interface.

This example performs the same task as in Example 19.1. The classes PrintChar and PrintNum are the same as in Example 19.1 except that they implement the Runnable interface rather than extend the Thread class.

The PrintChar and PrintNum classes are implemented as inner classes to avoid naming conflicts with the PrintChar and PrintNum classes in Example 19.1. Threads printA, printB, and print100 are created in the constructor instead of directly in the main method. This is because the main method is static and the inner classes PrintChar and PrintNum are nonstatic; you cannot reference nonstatic members of a class in a static method.

An instance of the class that extends the Thread class is a thread, which can be started using the start() method in the Thread class. But an instance of the class that implements the Runnable interface is not yet a thread. You have to wrap it, using the Thread class, to construct a thread for the instance, such as

```
Thread printA = new Thread(new PrintChar('a', 100));
```

## 19.5 Thread Controls and Communications

The Thread class contains the methods for controlling threads, as shown in Figure 19.6.

java.lang.Thread	
+Thread(tagert: Runnable)	Creates a new thread to run the target object.
+run(): void	Invoked by the Java runtime system to execute the thread. You must override this method and provide the code you want your thread to execute in your thread class. This method is never directly invoked by the runnable object in a program, although it is an instance method of a runnable object.
+start(): void	Starts the thread that causes the run() method to be invoked by JVM.
+interrupt(): void	Interrupts this thread. If the thread is blocked, it is ready to run again.
+isAlive(): boolean	Tests whether the thread is currently running.
+setPriority(p: int): void	Sets priority p (ranging from 1 to 10) for this thread.
+join(): void	Waits for this thread to finish.
+sleep(millis: long): void	Puts the runnable object to sleep for a specified time in milliseconds.
+yield(): void	Causes this thread to temporarily pause and allow other threads to execute.
+isInterrupted(): Boolean	Tests whether the current thread has been interrupted.
+currentThread(): Thread	Returns a reference to the currently executing thread object.

**Figure 19.6**

*The Thread class contains the methods for controlling threads.*

NOTE: The Thread class also contains the stop(), suspend(), and resume() methods. As of Java 2, these methods are *deprecated* (or *outdated*) because they are known to be inherently unsafe. You should assign null to a Thread variable to indicate that it is stopped rather than use the stop() method.

You can use the yield() method to temporarily release time for other threads. For example, suppose you modify the code in Lines 56-60 in TestRunnable.java in Example 19.2 as follows:

```

public void run() {
    for (int i = 1; i <= lastNum; i++) {
        System.out.print(" " + i);
        Thread.yield();
    }
}

```

Every time a number is printed, the print100 thread is yielded. So, the numbers are printed after the characters.

The sleep(long mills) method puts the thread to sleep for the specified time in milliseconds. For example, suppose you modify the code in Lines 56-60 in TestRunnable.java in Example 19.2 as follows:

```

public void run() {
    for (int i = 1; i <= lastNum; i++) {
        System.out.print(" " + i);
    }
}

```

```

    try {
        if (i >= 50) Thread.sleep(1);
    }
    catch (InterruptedException ex) {
    }
}
}

```

Every time a number ( $\geq 50$ ) is printed, the print100 thread is put to sleep for 1 millisecond.

You can use the join() method to force one thread to wait for another thread to finish. For example, suppose you modify the code in Lines 56-60 in TestRunnable.java in Example 19.2 as follows:

```

public void run() {
    for (int i = 1; i <= lastNum; i++) {
        System.out.print(" " + i);
        try {
            if (i == 50) printA.join();
        }
        catch (InterruptedException ex) {
        }
    }
}
}

```

The numbers after 50 are printed after thread printA is finished.

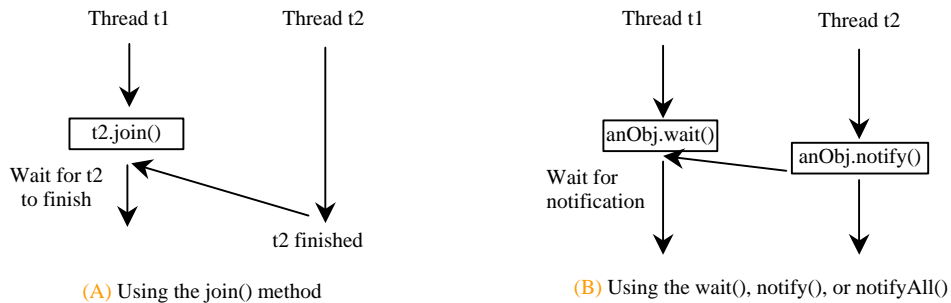
The wait(), notify(), and notifyAll() methods defined in the Object class are also used with threads to facilitate communications among the active threads:

[BL] public final void wait() throws InterruptedException  
 Forces the thread to wait until the notify or notifyAll method is called for the object to which wait is called.

[BL] public final void notify()  
 Awakens one of the threads that are waiting on this object. Which one is notified depends on the system implementation.

[BL] public final void notifyAll()  
 Awakens all the threads that are waiting on this object.

While the join() method forces one thread to wait for another thread to finish, the wait(), notify(), and notifyAll() are used to cooperate among the active threads, as shown in Figure 19.7. More detail on these methods is discussed in Section 19.7.3, "Cooperation Among Threads."

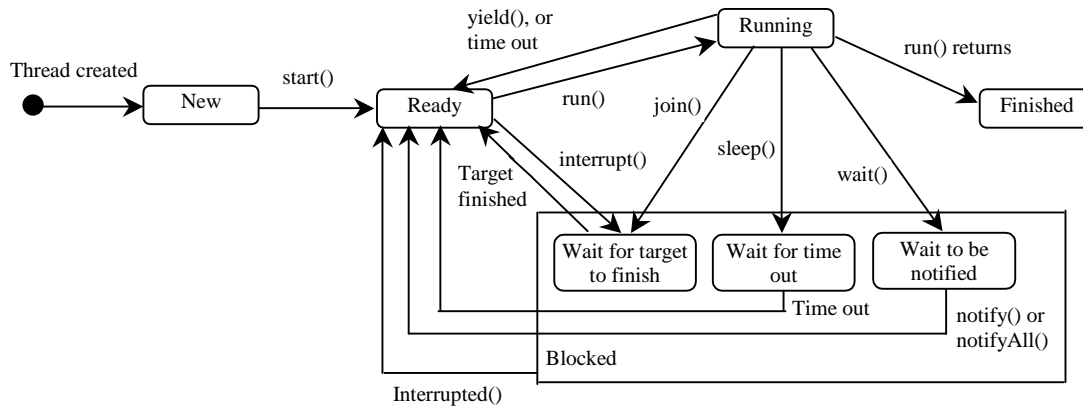


**Figure 19.7**

The `join()`, `wait()`, `notify()`, and `notifyAll()` are used to communicate among threads.

### 19.5.1 Thread States

Threads can be in one of five states: **New**, **Ready**, **Running**, **Blocked**, or **Finished** (see Figure 19.8).



**Figure 19.8**

A thread can be in one of five states: **New**, **Ready**, **Running**, **Blocked**, or **Finished**.

When a thread is newly created, it enters the **New** state. After a thread is started by calling its `start()` method, it enters the **Ready** state. A ready thread is runnable but may not be running yet. The operating system has to allocate CPU time to it.

When a ready thread begins executing, it enters the **Running** state. A running thread may enter the ready state if its given CPU time expires or its `yield()` method is called.

A thread can enter the **Blocked** state (i.e., become inactive) for several reasons. It may have invoked the `join()`,

sleep(), or wait() method, or some other thread may have invoked these methods. It may be waiting for an I/O operation to finish. A blocked thread may be reactivated when the action inactivating it is reversed. For example, if a thread has been put to sleep and the sleep time has expired, the thread is reactivated and enters the ready state.

Finally, a thread is *finished* if it completes the execution of its run() method.

The isAlive() method is used to find out the state of a thread. It returns true if a thread is in the **Ready**, **Blocked**, or **Running** state; it returns false if a thread is new and has not started or if it is finished.

The interrupt() method interrupts a thread in the following way: If a thread is currently in the **Ready** or **Running** state, its interrupted flag is set; if a thread is currently blocked, it is awakened and enters the **Ready** state, and an java.io.InterruptedExcep~~tion~~ is thrown.

### 19.5.2 Thread Priorities

Java assigns every thread a priority. By default, a thread inherits the priority of the thread that spawned it. You can increase or decrease the priority of any thread by using the setPriority method, and you can get the thread's priority by using the getPriority method. Priorities are numbers ranging from 1 to 10. The Thread class has the int constants MIN\_PRIORITY, NORM\_PRIORITY, and MAX\_PRIORITY, representing 1, 5, and 10, respectively. The priority of the main thread is Thread.NORM\_PRIORITY.

The Java runtime system always picks the currently runnable thread with the highest priority. If several runnable threads have equally high priorities, the CPU is allocated to all of them in round-robin fashion. A lower-priority thread can run only when no higher-priority threads are running. For example, suppose you insert the following code in Line 20 in TestRunnable.java in Example 19.2:

```
printA.setPriority(Thread.MAX_PRIORITY);
```

The print100 thread will be finished first.

#### TIP

The priority numbers may be changed in a future version of Java. To minimize the impact of any changes, use the constants in the Thread class to specify thread priorities.

## TIP

A thread may never get a chance to run if there is always a higher-priority thread running or a same-priority thread that never yields. This situation is known as *contention* or *starvation*. To avoid contention, the thread with high-priority must periodically invoke the `sleep` or `yield` method to give a thread with lower or same priority a chance to run.

## 19.6 Thread Groups

A *thread group* is a set of threads. Some programs contain quite a few threads with similar functionality. For convenience, you can group them together and perform operations on the entire group. For example, you can suspend or resume all of the threads in a group at the same time.

Listed below are the guidelines for using thread groups:

1. Use the `ThreadGroup` constructor to construct a thread group:

```
ThreadGroup g = new ThreadGroup("thread_group");
```

This creates a thread group `g` named "thread\_group". The name is a string and must be unique.

2. Using the `Thread` constructor, place a thread in a thread group:

```
Thread t = new Thread(g, new ThreadClass(), "This thread");
```

This statement creates a thread and places it in the thread group `g`. You can add a thread group under another thread group to form a tree in which every thread group except the initial one has a parent.

3. To find out how many threads in a group are currently running, use the `activeCount()` method. The following statement displays the active number of threads in group `g`.

```
System.out.println("The number of runnable threads in the group " + g.activeCount());
```

4. Each thread belongs to a thread group. By default, a newly created thread becomes a member of the current thread group that spawned it. To find which group a thread belongs to, use the `getThreadGroup()` method.

## NOTE

You have to start each thread individually. There is no `start()` method in `ThreadGroup`.

In the next section, you will see an example that uses the

ThreadGroup class.

## 19.7 Synchronization and Cooperation Among Threads

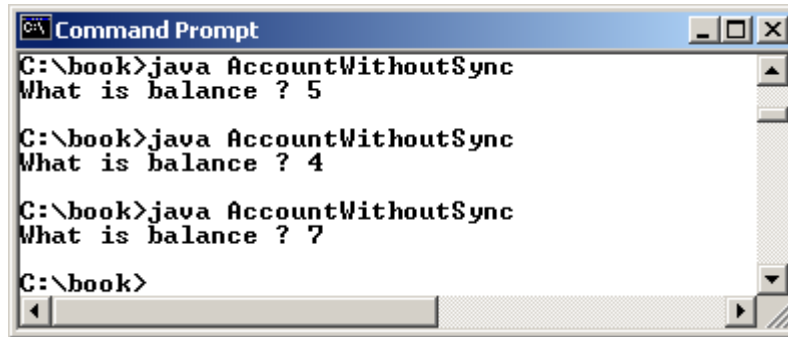
A shared resource may be corrupted if it is accessed simultaneously by multiple threads. The following example demonstrates the problem.

### *Example 19.3*

#### *Showing Resource Conflict*

##### **Problem**

Write a program that demonstrates the problem of resource conflict. Suppose that you create and launch one hundred threads, each of which adds a penny to an account. Assume that the account is initially empty. A sample run of the program is shown in Figure 19.9.



```
C:\book>java AccountWithoutSync
What is balance ? 5

C:\book>java AccountWithoutSync
What is balance ? 4

C:\book>java AccountWithoutSync
What is balance ? 7

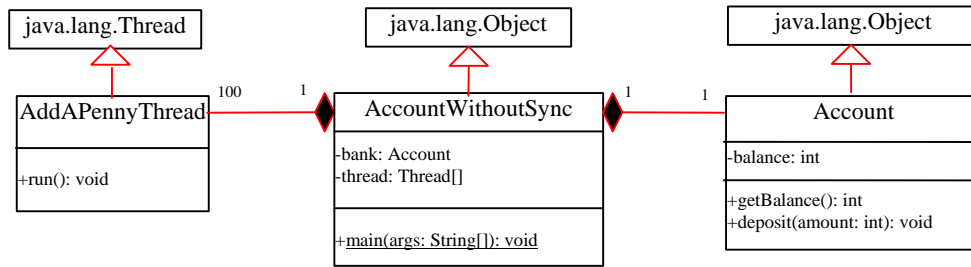
C:\book>
```

**Figure 19.9**

*The AccountWithoutSync program causes data inconsistency.*

##### **Solution**

Create a class named Account to model the account, a class named AddAPennyThread to add a penny to the account, and a main class that creates and launches threads. The relationships of these classes are shown in Figure 19.10.



**Figure 19.10**

*AccountWithoutSync* contains an instance of *Account*, and one hundred threads of *AddAPennyThread*.

**\*\*\*AU: Mark the critical region in the code\*\*\***

**\*\*\*PD: Please add line numbers in the following code\*\*\***

```

// AccountWithoutSync.java: Demonstrate resource conflict
public class AccountWithoutSync {
    private Account account = new Account();
    private Thread[] thread = new Thread[100];

    public static void main(String[] args) {
        AccountWithoutSync test = new AccountWithoutSync();
        System.out.println("What is balance ? " +
            test.account.getBalance());
    }

    public AccountWithoutSync() {
        ThreadGroup g = new ThreadGroup("group");
        boolean done = false;

        // Create and launch 100 threads
        for (int i = 0; i < 100; i++) {
            thread[i] = new Thread(g, new AddAPennyThread(), "t");
            thread[i].start();
        }

        // Check if all the threads are finished
        while (!done)
            if (g.activeCount() == 0)
                done = true;
    }

    // A thread for adding a penny to the account
    class AddAPennyThread extends Thread {
        public void run() {
            account.deposit(1);
        }
    }

    // An inner class for account
    class Account {
        private int balance = 0;

        public int getBalance() {
            return balance;
        }

        public void deposit(int amount) {
            int newBalance = balance + amount;

            // This delay is deliberately added to magnify the
            // data-corruption problem and make it easy to see.
            try {
                Thread.sleep(5);
            }
        }
    }
}

```



```

    }
    catch (InterruptedException ex) {
    }
    balance = newBalance;
}
}
}

```

## Review

The program creates one hundred threads in the array `thread`, and groups all of them into a thread group `g` (Lines 13-20). The `activeCount()` method is used to count the active threads. When all the threads are finished, it returns 0.

The balance of the account is initially 0 (Line 48). When all the threads are finished, the balance should be 100, but the output is unpredictable. As can be seen in Figure 19.10, the answers are wrong in the sample run. This demonstrates the data-corruption problem that occurs when all the threads have access to the same data source simultaneously.

Lines 46-56 could be replaced by one statement:

```
balance = balance + amount;
```

However, it is highly unlikely to replicate the problem using this single statement, although plausible. These statements in Lines 46-56 are deliberately designed to magnify the data-corruption problem and make it easy to see. If you run the program several times but still do not see the problem, increase the sleep time. This will increase the chances for showing the problem of data inconsistency.

## \*\*\*End of Example box

What, then, caused the error in Example 19.3? Here is a possible scenario, as shown in Figure 19.11.

Step	balance	thread[i]	thread[j]
1	0	newBalance = balance + 1;	
2	0		newBalance = balance + 1;
3	1	balance = newBalance;	
4	1		balance = newBalance;

**Figure 19.11**

thread[i] and thread[j] both add 1 to the same balance.

In Step 1, thread[i], for some i, gets the balances from the account. In Step 2, thread[j], for some j, gets the same balances from the account. In Step 3, thread[i] writes a new balance to the account. In Step 4, thread[j] writes a new balance to the account.

The effect of this scenario is that thread thread[i] did nothing, because in Step 4 thread thread[j] overrides thread[i]'s result. Obviously, the problem is that thread[i] and thread[j] are accessing a common resource in a way that causes conflict. This is a common problem known as a *race condition* in multithreaded programs. A class is said to be *thread-safe* if an object of the class does not cause a race condition in the presence of multiple threads. As demonstrated in the preceding example, the Account class is not thread-safe.

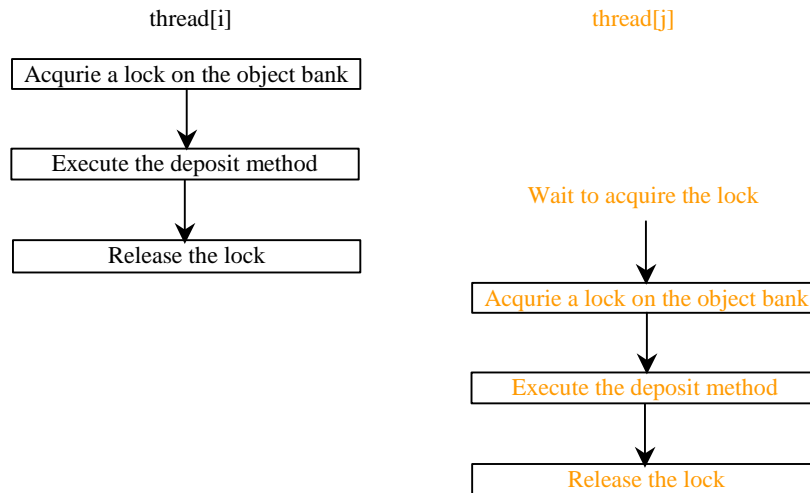
### **19.7.1 Synchronizing Instance and Static Methods**

To avoid race conditions, more than one thread must be prevented from simultaneously entering certain part of the program, known as *critical region*. The critical region in Example 19.3 is the entire deposit method. You can use the synchronized keyword to synchronize the method so that only one thread can access the method at a time. There are several ways to correct the problem in Example 19.3, one approach is to make Account thread-safe by adding the synchronized keyword in the deposit method in Line 45 as follows:

```
public synchronized void deposit(double amount)
```

A synchronized method acquires a lock before it executes. In the case of an instance method, the lock is on the object for which the method was invoked. In the case of a static method, the lock is on the class. If one thread invokes a synchronized instance method (respectively, static method) on an object, the lock of that object (respectively, class) is acquired first, then the method is executed, and finally the lock is released. Another thread invoking the same method of that object (respectively, class) is blocked until the lock is released.

With the deposit method synchronized, the preceding scenario cannot happen. If thread thread[j] starts to enter the method, and thread thread[i] is already in the method, thread thread[j] is blocked until thread thread[i] finishes the method, as shown in Figure 19.12.



**Figure 19.12**

*thread[i] and thread[j] are synchronized.*

Suppose you are not allowed to modify `Account`, you could add a new synchronized method that invokes `deposit(1)` and invoke this new method from the `run()` method. See Exercise 19.11.

### 19.7.2 Synchronizing Statements

Invoking a synchronized instance method of an object acquires a lock on the object, and invoking a synchronized static method of a class acquires a lock on the class. A synchronized statement can be used to acquire a lock on any object, not just *this* object, when executing a block of the code in a method. This block is referred to as a *synchronized block*. The general form of a synchronized statement is as follows:

```

synchronized (expr) {
    statements;
}
  
```

The expression `expr` must evaluate to an object reference. If the object is already locked by another thread, the thread is blocked until the lock is released. When a lock is obtained on the object, the statements in the synchronized block are executed, and then the lock is released.

Synchronized statements enable you to synchronize part of the code in a method instead of the entire method. This increases concurrency. Synchronized statements enable you to acquire a lock on any object so that you can synchronize the access to an object instead of to a method. You can make Example 19.3 thread-safe by placing the statement in Line 32 inside a synchronized block as follows:

```

synchronized (account) {
    account.deposit(1);
}
  
```

## NOTE

Any synchronized instance method can be converted into a synchronized statement. Suppose that the following is a synchronized instance method:

```
public synchronized void xMethod() {  
    // method body  
}
```

This method is equivalent to

```
public void xMethod() {  
    synchronized (this) {  
        // method body  
    }  
}
```

\*\*\*End of NOTE

### 19.7.3 Cooperation Among Threads

Thread synchronization suffices to avoid race conditions by ensuring the mutual exclusion in the critical region in multiple threads, but sometimes you also need a way to cooperate among threads. The wait(), notify(), and notifyAll() methods can be used to facilitate communication among threads. The wait() method lets the thread wait until some condition occurs. When it occurs, you can use the notify() or notifyAll() methods to notify the waiting threads to resume normal execution. The notifyAll() method wakes up all waiting threads, while notify() picks up only one thread from a waiting queue.

The wait(), notify(), and notifyAll() methods must be called in a synchronized method or a synchronized block on the receiving object of these methods. Otherwise, an IllegalMonitorStateException would occur. The template for invoking these methods is shown in Figure 19.13.

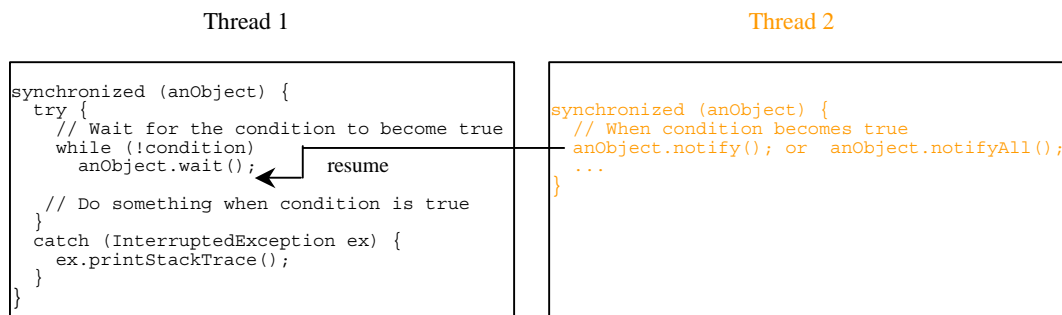


Figure 19.13

The wait(), notify(), and notifyAll() methods coordinate thread communication.

NOTE: A synchronization lock must be obtained on the object to be waited or notified. When wait() is invoked, it pauses the thread and simultaneously releases the lock on the object. When the thread is restarted after being notified, the lock is automatically reacquired.

#### Example 19.4

#### Thread Cooperation

##### Problem

Write a program that demonstrates thread cooperation. Suppose that you create and launch two threads, one deposit to an account, and the other withdraws from the same account. The second thread has to wait if the amount to be withdrawn is more than the current balance in the account. Whenever new fund is deposited to the account, the first thread notifies the second thread to resume. If the amount is still not enough for a withdrawal, the second thread has to continue to wait for more fund in the account. Assume the initial balance is 0 and the amount to deposit and to withdraw is randomly generated, a sample run of the program is shown in Figure 19.14.

Thread 1	Thread 2	Balance
		7
Deposit 7		8
Deposit 1		18
Deposit 10		9
	Withdraw 9	5
	Withdraw 4	2
	Withdraw 3	11
Deposit 9		6
	Withdraw 5	4
	Withdraw 2	7
Deposit 3		

Figure 19.14

*Thread 2 waits if there is not sufficient fund to withdraw.*

##### Solution

Create a new inner class named Account to model the account with two synchronized methods deposit(int) and withdraw(int), a class named DepositThread to add

an amount to the balance and a class name WithdrawThread to , and a main class that creates and launches two threads. The program is given as follows:

**\*\*\*PD: Please add line numbers in the following code\*\*\***

```
// ThreadCooperation.java: Demonstrate thread cooperation
public class ThreadCooperation {
    private Account account = new Account();
    private Thread thread1 = new DepositThread();
    private Thread thread2 = new WithdrawThread();

    public static void main(String[] args) {
        ThreadCooperation test = new ThreadCooperation();
        System.out.println("Thread 1\t\tThread 2\t\tBalance");
    }

    public ThreadCooperation() {
        thread1.start();
        thread2.start();
    }

    // A thread for adding an amount to the account
    class DepositThread extends Thread {
        public void run() {
            while (true) {
                account.deposit((int)(Math.random() * 10) + 1);
                try { // Purposely delay it to let the withdraw method wait
                    Thread.sleep(1000);
                }
                catch (InterruptedException ex) {
                    ex.printStackTrace();
                }
            }
        }
    }

    // A thread for subtracting an amount from the account
    class WithdrawThread extends Thread {
        public void run() {
            while (true) {
                account.withdraw((int)(Math.random() * 10) + 1);
            }
        }
    }

    // An inner class for account
    class Account {
        private int balance = 0;

        public int getBalance() {
            return balance;
        }

        public synchronized void deposit(int amount) {
            balance += amount;
            System.out.println("Deposit " + amount +
                "\t\t\t\t" + account.getBalance());
            notifyAll();
        }

        public synchronized void withdraw(int amount) {
            try {
                while (balance < amount)
                    wait();
            }
            catch (InterruptedException ex) {
                ex.printStackTrace();
            }

            balance -= amount;
            System.out.println("\t\t\tWithdraw " + amount +
                "\t\t" + account.getBalance());
        }
    }
}
```

```
}
I
```

## Review

The program creates thread1 for deposit (Line 4) and thread2 for withdrawal (Line 5). thread1 is purposely put to sleep (Line 23) to let thread2 run. When there is not enough fund to withdraw, thread2 waits (Line 59) for notification of the balance change from thread1 (Line 53).

What would happen if you replace the while loop in Lines 58-59 by the following if statement?

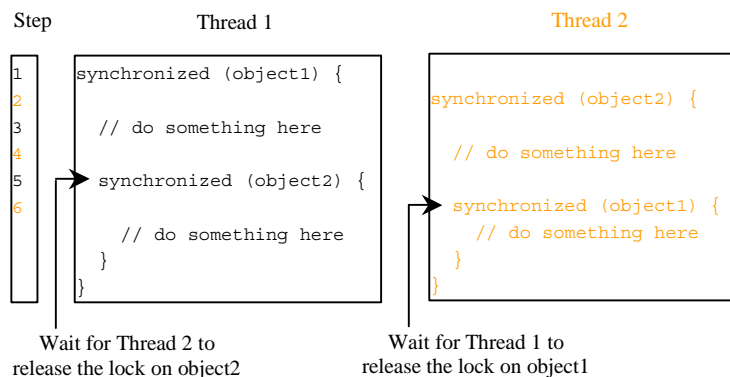
```
if (balance < amount)
    wait();
```

thread2 notifies thread1 whenever balance changed. (balance < amount) may be still true when thread1 is awakened. So, you should always use test the condition in a loop.

\*\*\*End of Example box

### 19.7.4 Deadlock

Sometimes two or more threads need to acquire the locks on several shared objects. This could cause *deadlock*, in which each thread has the lock on one of the objects and is waiting for the lock on the other object. Consider the scenario with two threads and two objects, as shown in Figure 19.15. Thread 1 acquired a lock on object1 and Thread 2 acquired a lock on object2. Now Thread 1 is waiting for the lock on object2 and Thread 2 for the lock on object1. The two threads wait for each other to release the in order to get the lock, and neither can continue to run.



**Figure 19.15**

*Thread 1 and Thread 2 are deadlocked.*

Deadlock can be easily avoided by using a simple technique known as *resource ordering*. With this technique, you assign an order on all the objects whose locks must be acquired and ensure that each thread acquires the locks in that order. For the example in Figure 19.15, suppose the objects are ordered as object1 and object2. Using the resource ordering technique, Thread 2 must acquire a lock on object1 first, then on object2. Once Thread 1 acquired a lock on object1, Thread 2 has to wait for a lock on object1. So Thread 1 will be able to acquire a lock on object2 and no deadlock would occur.

## 19.8 Controlling Animation Using Threads

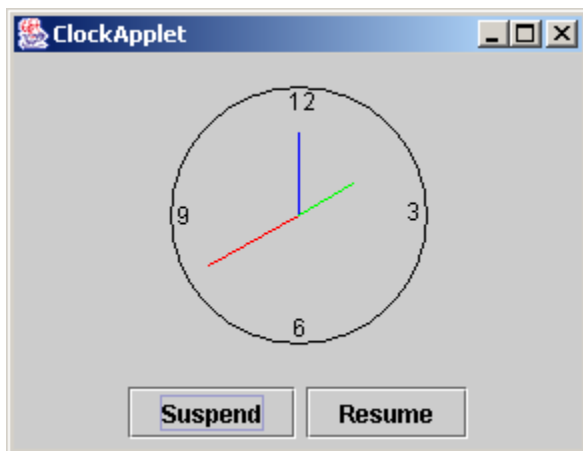
In Example 11.6, "Drawing a Clock," you drew a clock to show the current time. The clock does not tick after it is displayed. What can you do to make the clock display a new current time every second? The key to making the clock tick is to repaint it every second with a new current time. You can use a thread to control how to repaint the clock.

### *Example 19.5*

#### *Displaying a Running Clock in an Applet*

##### **Problem**

Write an applet that displays a runnable clock. Use two buttons to suspend and resume the clock, as shown in Figure 19.16.



**Figure 19.16**

*You can click the Suspend button to suspend the clock and the Resume button to resume the clock.*



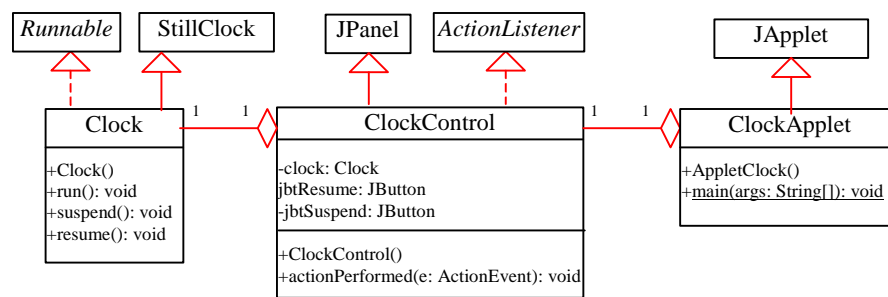
## Solution

Here are the major steps to complete this example:

1. Create a subclass of StillClock named Clock to enable the clock to run.
2. Create a subclass of JPanel named ClockControl to contain the clock with two control buttons *Suspend* and *Resume*.
3. Create an applet named ClockApplet to contain an instance of ClockControl and enable the applet to run standalone.

The relationship among these classes is shown in Figure 19.17.

**\*\*\* Same as Fig 18.12 in introjb3e p800**



**Figure 19.17**

ClockApplet contains ClockControl, and ClockControl contains Clock.

**\*\*\*PD: Please add line numbers in the following code\*\*\***

```
public class Clock extends StillClock implements Runnable {
    private boolean suspended;

    public Clock() {
        new Thread(this).start();
    }

    public void run() {
        while (true) {
            setCurrentTime();
            repaint();

            try {
                Thread.sleep(1000);
                waitForNotificationToResume();
            } catch (InterruptedException ex) {
            }
        }
    }
}
```

```

    public synchronized void suspend() {
        suspended = true;
    }

    public synchronized void resume() {
        if (suspended) {
            suspended = false;
            notifyAll();
        }
    }

    private synchronized void waitForNotificationToResume()
        throws InterruptedException {
        while (suspended)
            wait();
    }
}

```

**\*\*\*PD: Please insert a separator line**

**\*\*\*PD: Please add line numbers in the following code\*\*\***

```

1  import javax.swing.*;
2  import java.awt.event.*;
3  import java.awt.BorderLayout;

4  public class ClockControl extends JPanel implements ActionListener {
5      private Clock clock = new Clock();
6      private JButton jbtSuspend = new JButton("Suspend");
7      private JButton jbtResume = new JButton("Resume");

8      public ClockControl() {
9          // Group buttons in a panel
10         JPanel panel = new JPanel();
11         panel.add(jbtSuspend);
12         panel.add(jbtResume);

13         // Add clock and buttons to the panel
14         setLayout(new BorderLayout());
15         add(clock, BorderLayout.CENTER);
16         add(panel, BorderLayout.SOUTH);

17         // Register listeners
18         jbtSuspend.addActionListener(this);
19         jbtResume.addActionListener(this);
20     }

21     public void actionPerformed(ActionEvent e) {
22         if (e.getSource() == jbtSuspend)
23             clock.suspend();
24         else if (e.getSource() == jbtResume)
25             clock.resume();
26     }
27 }

```

**\*\*\*PD: Please insert a separator line**

**\*\*\*PD: Please add line numbers in the following code\*\*\***

```

1  // ClockApplet.java: Display a running clock on the applet
2  public class ClockApplet extends JApplet {
3      public ClockApplet() {
4          getContentPane().add(new ClockControl());
5      }
6  }

```

## Review

The Clock class extends StillClock to display the

clock and implements the Runnable interface to enable the clock to run. Since the suspend and resume methods in the Thread class are deprecated, you must create new methods for resuming and suspending threads. The variable suspended is declared as a data member of the class (Line 4), which indicates the state of the thread. The keyword synchronized ensures that the resume() and suspend() methods are synchronized to avoid race conditions that could result in an inconsistent value for the variable suspended.

Line 7 creates and starts the thread. The run method (Lines 10-22) is implemented to set a new time and repaint the clock every one second continuously. In the while loop body, the thread is blocked if suspended is true. The waitForNotificationToResume() (Line 17) method causes the thread to suspend and wait for notification by the notifyAll() method (Line 31) invoked from the resume() method.

The ClockControl class extends JPanel to display the clock and two control buttons and implements the ActionListener to handle action events from the buttons. When the Suspend button is clicked, the clock's suspend method is invoked to suspend the clock. When the Resume button is clicked, the clock's resume method is invoked to resume the clock.

The ClockApplet class simply places an instance of ClockControl in the applet's content pane. The main method is provided in the applet so that you can also run it standalone.

## 19.9 Controlling Animation Using the Timer Class

The preceding example creates a thread to run a while loop that repaints the clock in a panel every one second. Java animations frequently repaint panels at a predefined rate. For this reason, Java provides the javax.swing.Timer class, which can be used to control repainting a panel at a predefined rate. Using the Timer class dramatically simplifies the program. A Timer object serves as the source of an ActionEvent. It fires an ActionEvent at a fixed rate. The listeners for the event are registered with the Timer object. When you create a Timer object, you have to specify the delay and a listener using the following constructor:

```
public Timer(int delay, ActionListener listener)
```

where delay specifies the number of milliseconds between two action events. You can add additional listeners using the addActionListener method, and adjust the delay using the setDelay method. To start the timer, use the start method. To stop the timer, use the stop method. In the action

listener class, you can implement the `actionPerformed` handler by invoking the `repaint` method to repaint the panel. This `actionPerformed` method is invoked at a rate determined by the delay.

Using the `Timer` class, the preceding `Clock` class can be modified as follows:

```
import java.awt.event.*;
import javax.swing.Timer;

public class ClockWithTimer extends StillClock
    implements ActionListener {
    // Create a timer with delay 1000 ms
    protected Timer timer = new Timer(1000, this);

    public void suspend() {
        timer.stop(); // Suspend clock
    }

    public void resume() {
        timer.start(); // Resume clock
    }

    /** Handle the action event */
    public void actionPerformed(ActionEvent e) {
        // Set new time and repaint the clock to display current time
        setCurrentTime();
        repaint();
    }
}
```

#### **NOTE**

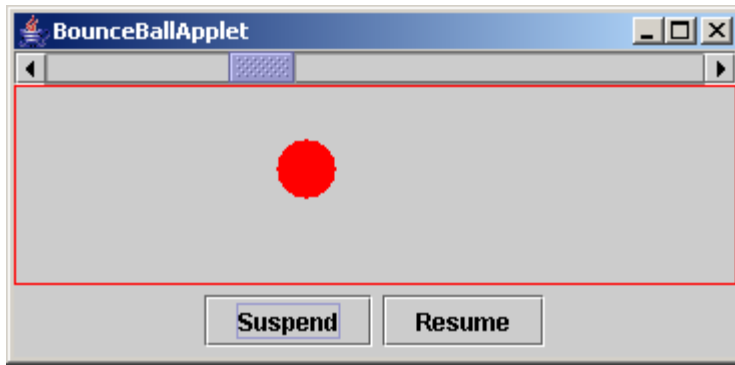
Using a thread to control animations and using a timer to control animations are two entirely different approaches. The thread approach places the animation control on a separate thread so that it does not interfere with the drawing. The timer approach uses the action event to control the drawing. If your drawings are painted at a fixed rate, use the timer approach, because it can greatly simplify programming.

#### **Example 19.6**

#### ***Displaying a Bouncing Ball***

##### **Problem**

Write an applet that displays a ball bouncing in a panel. Use two buttons to suspend and resume the movement and use a scroll bar to control the bouncing speed, as shown in Figure 19.18.



**Figure 19.18**

*The ball's movement is controlled by the Suspend and Resume buttons and the scroll bar.*

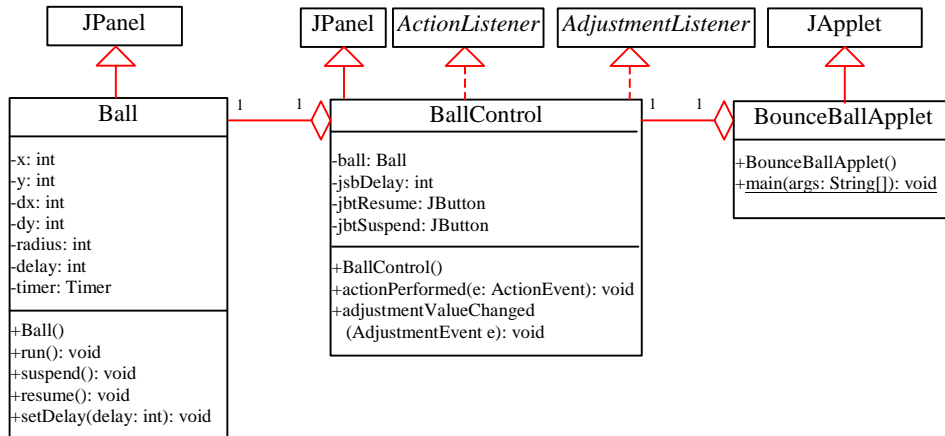
### **Solution**

Here are the major steps to complete this example:

1. Create a subclass of `JPanel` named `Ball` to display a ball bouncing.
2. Create a subclass of `JPanel` named `BallControl` to contain the ball with a scroll bar and two control buttons *Suspend* and *Resume*.
3. Create an applet named `BounceBallApplet` to contain an instance of `BallControl` and enable the applet to run standalone.

The relationship among these classes is shown in Figure 19.19.

**\*\*\* Same as Fig 18.14 in introjb3e p805**



**Figure 19.19**

*BounceBallApplet* contains *BallControl*, and *BallControl* contains *Ball*.

**\*\*\*PD: Please add line numbers in the following code\*\*\***

```

import java.awt.event.*;
import javax.swing.Timer;
import java.awt.*;
import javax.swing.*;

public class Ball extends JPanel implements ActionListener {
    private int delay = 10;

    // Create a timer with delay 1000 ms
    protected Timer timer = new Timer(delay, this);

    private int x = 0; private int y = 0; // Current ball position
    private int radius = 15; // Ball radius
    private int dx = 2; // Increment on ball's x-coordinate
    private int dy = 2; // Increment on ball's y-coordinate

    public Ball() {
        timer.start();
    }

    /** Handle the action event */
    public void actionPerformed(ActionEvent e) {
        repaint();
    }

    public void paintComponent(Graphics g) {
        super.paintComponent(g);

        g.setColor(Color.red);

        // Check boundaries
        if (x < radius) dx = Math.abs(dx);
        if (x > getWidth() - radius) dx = -Math.abs(dx);
        if (y < radius) dy = Math.abs(dy);
        if (y > getHeight() - radius) dy = -Math.abs(dy);

        // Adjust ball position
        x += dx;
        y += dy;
        g.fillOval(x - radius, y - radius, radius * 2, radius * 2);
    }

    public void suspend() {
        timer.stop(); // Suspend clock
    }
}

```

```

    public void resume() {
        timer.start(); // Resume clock
    }

    public void setDelay(int delay) {
        this.delay = delay;
        timer.setDelay(delay);
    }
}

```

**\*\*\*PD: Please insert a separator line**

**\*\*\*PD: Please add line numbers in the following code\*\*\***

```

1 import javax.swing.*;
2 import java.awt.event.*;
3 import java.awt.*;
4
5 public class BallControl extends JPanel
6     implements ActionListener, AdjustmentListener {
7     private Ball ball = new Ball();
8     private JButton jbtSuspend = new JButton("Suspend");
9     private JButton jbtResume = new JButton("Resume");
10    private JScrollBar jsbDelay = new JScrollBar();
11
12    public BallControl() {
13        // Group buttons in a panel
14        JPanel panel = new JPanel();
15        panel.add(jbtSuspend);
16        panel.add(jbtResume);
17
18        // Add ball and buttons to the panel
19        ball.setBorder(new javax.swing.border.LineBorder(Color.red));
20        jsbDelay.setOrientation(JScrollBar.HORIZONTAL);
21        ball.setDelay(jsbDelay.getMaximum());
22        setLayout(new BorderLayout());
23        add(jsbDelay, BorderLayout.NORTH);
24        add(ball, BorderLayout.CENTER);
25        add(panel, BorderLayout.SOUTH);
26
27        // Register listeners
28        jbtSuspend.addActionListener(this);
29        jbtResume.addActionListener(this);
30        jsbDelay.addAdjustmentListener(this);
31    }
32
33    public void actionPerformed(ActionEvent e) {
34        if (e.getSource() == jbtSuspend)
35            ball.suspend();
36        else if (e.getSource() == jbtResume)
37            ball.resume();
38    }
39
40    public void adjustmentValueChanged(AdjustmentEvent e) {
41        ball.setDelay(jsbDelay.getMaximum() - e.getValue());
42    }
43 }

```

**\*\*\*PD: Please insert a separator line**

**\*\*\*PD: Please add line numbers in the following code\*\*\***

```

1 import java.awt.*;
2 import java.awt.event.*;
3 import java.applet.*;
4 import javax.swing.*;
5
6 public class BounceBallApplet extends JApplet {
7     public BounceBallApplet() {
8         getContentPane().add(new BallControl());
9     }
10 }

```



## Review

The Ball class extends JPanel to display a bouncing ball. A Timer object is created to enable the ball to be displayed continuously at a fixed rate. The center of the ball is at  $(x, y)$ , which changes to  $(x + dx, y + dy)$  on the next display. The suspend and resume methods (Lines 45-51) can be used to stop and start the timer.

The BallControl class extends JPanel to display the clock with a scroll bar and two control buttons, implements the ActionListener to handle action events from the buttons, and implements the AdjustmentListener to handle value change events from the scroll bar. When the Suspend button is clicked, the ball's suspend method is invoked to suspend the ball movement. When the Resume button is clicked, the ball's resume method is invoked to resume the ball movement. The bouncing speed can be changed using the scroll bar.

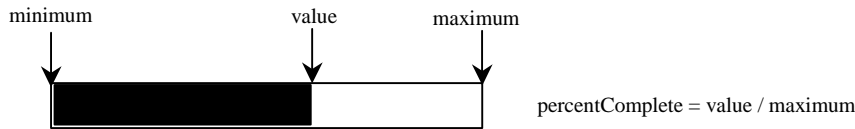
The BounceBallApplet class simply places an instance of BallControl in the applet's content pane. The main method is provided in the applet so that you can also run it standalone.

### 19.10 JProgressBar

JProgressBar is a component that displays a value graphically within a bounded interval. A progress bar is typically used to show the percentage of completion of a lengthy operation; it comprises a rectangular bar that is "filled in" from left to right horizontally or from bottom to top vertically as the operation is performed. It provides the user with feedback on the progress of the operation. For example, when a file is being read, it alerts the user to the progress of the operation, thereby keeping the user attentive.

JProgressBar is often implemented using a thread to monitor the completion status of other threads. The progress bar can be displayed horizontally or vertically, as determined by its orientation property. The minimum, value, and maximum properties determine the minimum, current, and maximum length on the progress bar, as shown in Figure 9.20. Figure 19.21 lists frequently used features of JProgressBar.





**Figure 19.20**

*JProgressBar displays the progress of a task.*

javafx.swing.JComponent	
javafx.swing.JProgressBar	
+JProgressBar()	Creates a horizontal progress bar with min 0 and max 100.
+JProgressBar(min: int, max: int)	Creates a horizontal progress bar with specified min and max.
+JProgressBar(orient: int)	Creates a progress bar with min 0 and max 100 and a specified orientation.
+JProgressBar(orient: int, min: int, max: int)	Creates a progress bar with a specified orientation, min, and max.
+getMaximum(): int	Gets the maximum value. (default: 100)
+setMaximum(n: int): void	Sets a new maximum value.
+getMinimum(): int	Gets the minimum value. (default: 0)
+setMinimum(n: int): void	Sets a new minimum value.
+getOrientation(): int	Gets the orientation value. (default: HORIZONTAL)
+setOrientation(orient: int): void	Sets a new minimum value.
+getPercentComplete():double	Returns the percent complete for the progress bar. 0 <= a value <= 1.0.
+getValue(): int	Returns the progress bar's current value
+setValue(n: int): void	Sets the progress bar's current value.
+getString(): String	Returns the current value of the progress string.
+setString(s: String): void	Sets the value of the progress string.
+isStringPainted(): Boolean	Returns the value of the stringPainted property.
+setStringPainted(b: boolean): void	Sets the value of the stringPainted property, which determines whether the progress bar should render a progress percentage string. (default: false)

**Figure 19.21**

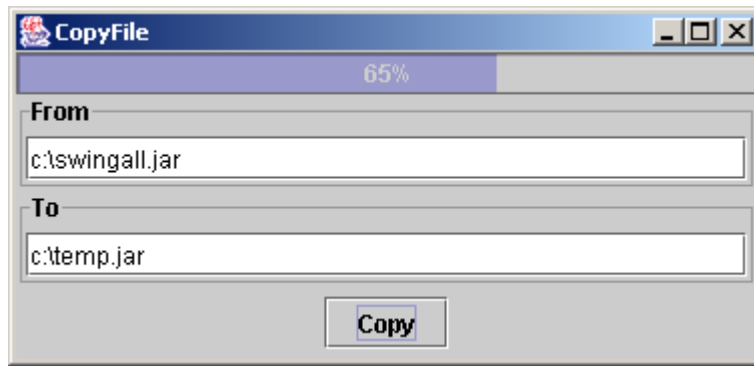
*JProgressBar is a Swing component with many properties that enables you to customize a progress bar.*

### Example 19.7

#### JProgressBar Demo

#### Problem

Write a GUI application that lets you copy files. A progress bar is used to show the progress of the copying operation, as shown in Figure 9.22.



**Figure 9.22**

*The user enters the files in the text fields and clicks the Copy button to start copying files.*

### Solution

Place a JProgressBar in the north of the frame. Place a button in a panel and place the panel in the source of the frame. Place the two text fields in two panels, set the titles on the borders of the panels, and place the panels in another panel of the GridLayout with two rows. Place the panel in the center of the frame.

While copying data from a source file to a destination file on one thread, the progress bar is updated on another thread. You need to create a thread that copies a file and another thread that updates the progress bar. Every time some bytes of the file are copied, the current value in the progress is updated to show the progress.

Create a main class named CopyFile that lays out the user interface. Create an inner class named CopyFileThread that extends Thread to copy files when the Copy button is pressed. Create another inner class named UpdateProgressBar inside CopyFileThread that updates the progress bar. The complete program is given as follows:

**\*\*\*PD: Please add line numbers in the following code\*\*\***

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;
import java.io.*;

public class CopyFile extends JFrame implements ActionListener {
```

```

private JProgressBar jpb = new JProgressBar();
private JButton jbtCopy = new JButton("Copy");
private JTextField jtfFrom = new JTextField();
private JTextField jtfTo = new JTextField();

public CopyFile() {
    JPanel jPanel2 = new JPanel();
    jPanel2.setLayout(new BorderLayout());
    jPanel2.setBorder(new TitledBorder("From"));
    jPanel2.add(jtfFrom, BorderLayout.CENTER);

    JPanel jPanel3 = new JPanel();
    jPanel3.setLayout(new BorderLayout());
    jPanel3.setBorder(new TitledBorder("To"));
    jPanel3.add(jtfTo, BorderLayout.CENTER);

    JPanel jPanel1 = new JPanel();
    jPanel1.setLayout(new GridLayout(2, 1));
    jPanel1.add(jPanel2);
    jPanel1.add(jPanel3);

    JPanel jPanel4 = new JPanel();
    jPanel4.add(jbtCopy);

    this.getContentPane().add(jpb, BorderLayout.NORTH);
    this.getContentPane().add(jPanel1, BorderLayout.CENTER);
    this.getContentPane().add(jPanel4, BorderLayout.SOUTH);

    jpb.setStringPainted(true); // Paint the percent in a string

    jbtCopy.addActionListener(this);
}

public void actionPerformed(ActionEvent e) {
    // Create a thread for copying files
    new CopyFileThread().start();
}

public static void main(String[] args) {
    CopyFile frame = new CopyFile();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setTitle("CopyFile");
    frame.setSize(400, 180);
    frame.setVisible(true);
}

// Copy file and update progress bar in a separate thread
class CopyFileThread extends Thread {
    private int currentValue;

    public void run() {
        BufferedInputStream in = null;
        BufferedOutputStream out = null;
        try {

```



thread. What would happen if copying file is not run on a separate thread? The progress bar will not be updated until the copy ends. This is because the paint method for repainting the progress bar runs on the thread with the actionPerformed method. As long as the copy operation continues, the progress bar never gets chance to be repainted. Running the copy operation on a separate thread would enable the progress bar to be repainted simultaneously as the copy operation progresses.

### Key Classes and Methods

- **java.lang.Thread** A class that contains the constructors for creating threads, as well as many useful methods for controlling threads.
- **java.lang.Runnable** An interface that provides a common protocol for runnable objects. It contains the run() method.
- **javax.swing.Timer** Sets a timer that fires one or more action events after a specified delay.
- **javax.swing.JProgressBar** A Swing component that can be used to graphically display the completion status of a thread.

### Key Terms

- **multithreading** The capability of a program to perform several tasks simultaneously within a program.
- **thread** A flow of execution of a task, with a beginning and an end, in a program. A thread must be an instance of java.lang.Thread.
- **synchronized** A keyword to specify a synchronized method or a synchronized block. A synchronized instance method acquires a lock on this object and a synchronized static method acquires a lock on the class. A synchronized block acquires a lock on a specified object, not just this object before executing the statements in the block.
- **race condition** A situation that causes data corruption due to unsynchronized access of data by multiple threads.
- **thread-safe** A class is said to be thread-safe if an object of the class does not cause a race condition in the presence of multiple threads.
- **deadlock** A situation in which two or more threads acquire locks on multiple objects and each has the lock on one object and is waiting for the lock on the other object.

### Chapter Summary

- You can derive your thread class from the Thread class and create a thread instance to run a task on a separate thread. If your class needs to inherit

multiple classes, implement the Runnable interface to run multiple tasks in the program simultaneously.

- After a thread object is created, use the start() method to start a thread, and the sleep(long) method to put a thread to sleep so that other threads get a chance to run. Since the stop, suspend, and resume methods are deprecated in Java 2, you need to implement these methods to stop, suspend, and resume a thread, if necessary.
- A thread object never directly invokes the run method. The Java runtime system invokes the run method when it is time to execute the thread. Your class must override the run method to tell the system what the thread will do when it runs.
- A thread can be in one of five states: **New**, **Ready**, **Running**, **Blocked**, or **Finished**. When a thread is newly created, it enters the **New** state. After a thread is started by calling its start() method, it enters the **Ready** state. A **Ready** thread is runnable but may not be running yet. The operating system has to allocate CPU time to it.
- When a ready thread begins executing, it enters the **Running** state. A running thread may reenter the **Ready** state if its given CPU time expires or its yield() method is called.
- A thread can enter the **Blocked** state (i.e., become inactive) for several reasons. It may have invoked the sleep(long), wait(), or interrupt() method, or some other thread may have invoked its sleep or interrupt() method. It may be waiting for an I/O operation to finish. A **Blocked** thread may be reactivated when the action inactivating it is reversed. For example, if a thread has been put to sleep and the sleep time has expired, the thread is reactivated and enters the **Ready** state.
- A thread is **Finished** if it completes the execution of its run() method.
- Threads can be assigned priorities. The Java runtime system always executes the ready thread with the highest priority. You can use a thread group to put relevant threads together for group control.
- To prevent threads from corrupting a shared resource, use synchronized methods or blocks. A synchronized method acquires a lock before it executes. In the case

of an instance method, the lock is on the object for which the method was invoked. In the case of a static (class) method, the lock is on the class.

- A synchronized statement can be used to acquire a lock on any object, not just *this* object, when executing a block of the code in a method. This block is referred to as a *synchronized block*.
- *Deadlock* occurs in the case that two or more threads acquire locks on multiple objects and each has the lock on one object and is waiting for the lock on the other object. The *resource ordering technique* can be used to avoid deadlock.
- You can use either the thread approach or the timer approach to control Java animations. Using a thread to control animations and using a timer to control animations are two entirely different approaches. The thread approach places the animation control on a separate thread so that it does not interfere with the drawing. The timer approach uses the action event to control the drawing. If your drawings are painted at a fixed rate, use the timer approach, because it can greatly simplify programming.

## Review Questions

### Sections 19.1-19.4

19.1

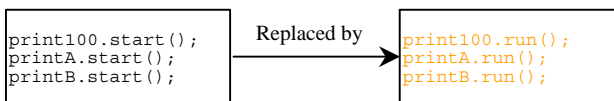
Why do you need multithreading? How can multiple threads run simultaneously in a single-processor system?

19.2

What are two ways to create threads? When do you use the Thread class, and when do you use the Runnable interface? What are the differences between the Thread class and the Runnable interface?

19.3

How do you create a thread and launch a thread object? What would happen if you replace the start() method by the run() method in Lines 11-13 in Example 19.1?



### Section 19.5 Thread Controls and Communications

19.4

Why does the following class have a runtime error?

**\*\*\*PD: Please add line numbers in the following code\*\*\***

```
public class Test extends Thread {  
    public static void main(String[] args) {  
        Test t = new Test();  
        t.start();  
        t.start();  
    }  
  
    public void run() {  
        System.out.println("test");  
    }  
}
```

19.5

Which of the following methods are instance methods in java.lang.Thread? Which method may throw an InterruptedException? Which of them are deprecated in Java 2?

run, start, stop, suspend, resume, sleep,  
interrupt, isInterrupted, yield, join,  
currentThread

19.6

Can the wait(), notify(), and notifyAll() be invoked from any object? What is the purpose of these methods?

19.7

Explain the life-cycle of a thread object. How do you set a priority for a thread? What is the default priority?

### *Section 19.6 Thread Groups*

19.8

Describe a thread group. How do you create a thread group?

19.9

How do you start the threads in a group? How do you find the number of active threads in a group of threads?

### *Section 19.7 Synchronization and Cooperation Among Threads*

19.10

Give some examples of possible resource corruption when running multiple threads. How do you synchronize conflict threads?



19.11

Suppose you place the statement in Line 32 inside a synchronized block to avoid race conditions in Example 19.3 as follows:

```
synchronized (this) {  
    account.deposit(1);  
}
```

Does it work?

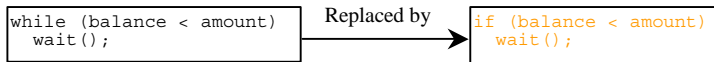
19.12

What is wrong in the following code?

```
synchronized (object1) {  
    try {  
        while (!condition) object2.wait();  
    }  
    catch (InterruptedException ex) {  
    }  
}
```

19.13

What would happen if the while loop in Lines 58-59 in Example 19.4 is changed to the if statement?



19.14

Why does the following class have a syntax error?

**\*\*\*PD: Please add line numbers in the following code\*\*\***

```
import javax.swing.*;  
  
public class Test extends JApplet implements Runnable {  
    public void init() throws InterruptedException {  
        Thread thread = new Thread(this);  
        thread.sleep(1000);  
    }  
  
    public synchronized void run() {  
    }  
}
```

19.15

What is deadlock? How can you avoid deadlock?

*Sections 19.8-19.10*

19.16

How do you override the methods init, start, stop, and destroy in the Applet class to work well with the threads in the applets?

19.17

Will the program behave differently if `Thread.sleep(1000)` is replaced by `thread.sleep(1000)` in Example 19.5, "Displaying a Running Clock in an Applet"?

19.18

How do you use the `Timer` class to control Java animations?

19.19

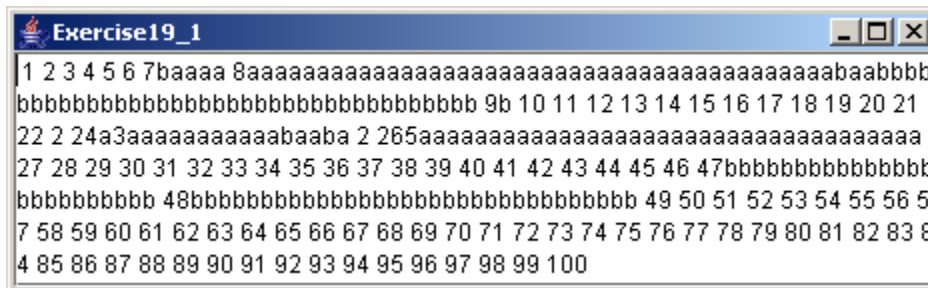
What is the property that displays the percentage of work completed in `JProgressBar`?

## Programming Exercises

### Sections 19.1-19.4

19.1

(Revising Example 19.1 "Using the `Thread` Class to Create and Launch Threads") Rewrite Example 19.1 to display the output in a text area, as shown in Figure 19.23.



**Figure 19.23**

*The output from three threads is displayed in a text area.*

### Section 19.7 Synchronization and Cooperation Among Threads

19.2

(Synchronizing threads) Write a program that launches one hundred threads. Each thread adds 1 to a variable `sum` that initially is zero. You need to pass `sum` by reference to each thread. In order to pass it by reference, define an `Integer` wrapper object to hold `sum`. Run the program with and without synchronization to see its effect.

19.3

(Modifying Example 19.3 "Showing Resource Conflict") Suppose

you are not allowed to modify Account, add a new synchronized method that invokes deposit(1) and invoke this new method from the run() method.

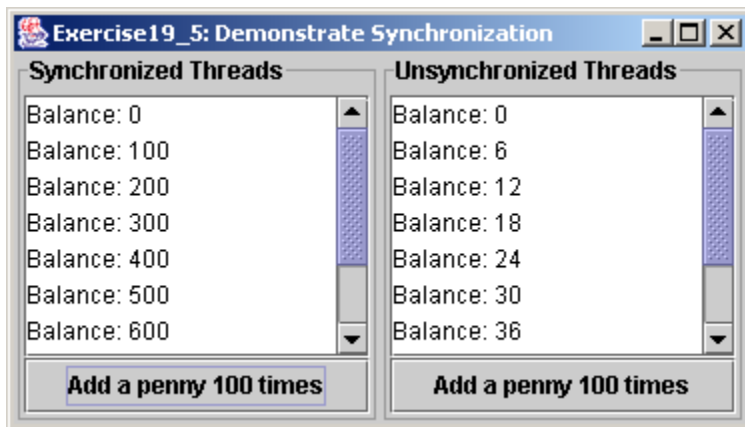
19.4

(Using synchronized statements) Modify Example 19.3, "Showing Resource Conflict," using a synchronized statement to synchronize access to the Account object in the run() method.

19.5

(Revising Example 19.3 "Showing Resource Conflict") Modify Example 19.3 as follows:

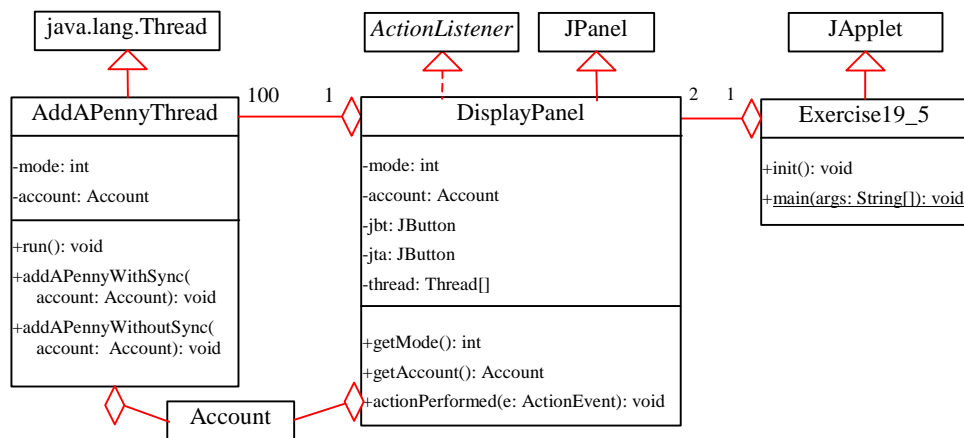
[BL]Create two panels with the titles "Synchronized Threads" and "Unsynchronized Threads", as shown in Figure 19.24. The Synchronized Threads panel displays an account balance after a penny has been added one hundred times using synchronized threads. The Unsynchronized Threads panel displays an account balance after a penny has been added one hundred times using unsynchronized threads.



**Figure 19.24**

*The program shows the effect of executing the threads with and without synchronization.*

[BL]Since the two panels are very similar, you can create a class to model them uniformly, as shown in Figure 19.20. Use a variable named mode to indicate whether synchronized threads or unsynchronized threads are used in the panel. Invoke the method addAPennyWithSync or addAPennyWithoutSync, depending on the mode.



Same as in Example 19.3

**Figure 19.25**

*DisplayPanel* displays the balance in the text area after a penny is added to the balance one hundred times.

19.6

(Demonstrating ConcurrentModificationException) The iterator is *fail-fast*. This means that if you are using an iterator to traverse a collection while the underlying collection is being modified by another thread, then the iterator fails immediately by throwing java.util.ConcurrentModificationException. Since this exception is a subclass of RuntimeException, you don't have to place the methods of Iterator in a try-catch block. Create a program with two threads concurrently accessing and modifying a set. The first thread creates a hash set filled with numbers, and adds a new number to the set every second. The second thread obtains an iterator for the set and traverses the set back and forth through the iterator every second. You will receive a ConcurrentModificationException, because, while the set in the second thread is being traversed, the underlying set is being modified in the first thread.

19.7

(Using synchronized sets) Correct the problem in the preceding exercise using synchronization so that the second thread does not throw ConcurrentModificationException. You will have to use the Collections.synchronizedSet(set) method to obtain a synchronized set and acquire a lock on the returned set when traversing it. It is, however, imperative for a thread to acquire a lock on the synchronized list, set, or map when traversing it through an iterator, as shown

in the following code:

```
Set hashSet = Collections.synchronizedSet(new HashSet());  
synchronized (hashSet) { // Must synchronize it  
    Iterator iterator = hashSet.iterator();  
  
    while (iterator.hasNext()) {  
        System.out.println(iterator.next());  
    }
```

Failure to do so may result in nondeterministic behavior, such as ConcurrentModificationException.

## 19.8

(Producer and consumer) A classic exercise to use thread communication is the producer and consumer type of program. Write a program with two threads named producer and consumer. A producer produces data and the consumer consumes the data. Assume that the data is stored in a stack. When the stack is empty, the consumer waits. Whenever the producer adds data to the stack, it notifies the consumer.

## 19.9

(Demonstrating deadlock) Write a program that demonstrates deadlock.

## *Sections 19.8-19.10*

## 19.10

(Displaying a flashing label) Write an applet that displays a flashing label. Enable it to run standalone.

### **HINT**

To make the label flash, you need to repaint the panel alternately with the label and without the label (blank screen) at a fixed rate. Use a boolean variable to control the alternation.

## 19.11

(Displaying a moving label) Write an applet that displays a moving label. The label continuously moves from right to left in the applet's viewing area. Whenever the label disappears at the far left of the viewing area, it reappears again on the right-hand side. The label freezes when the mouse is pressed, and moves again when the button is released. Enable it to run as an application.

### **HINT**

Redraw the label with a new x coordinate at a fixed rate.

19.12

(Simulating a stock ticker) Write a Java applet that displays a stock index ticker (see Figure 19.26). The stock index information is passed from the `<param>` tag in the HTML file. Each index has four parameters: Index Name (e.g., S&P 500), Current Time (e.g., 15:54), the index from the previous day (e.g., 919.01), and Change (e.g., 4.54). Enable the applet to run standalone.

Use at least five indexes, such as Dow Jones, S&P 500, NASDAQ, NIKKEI, and Gold & Silver Index. Display positive changes in green, and negative changes in red. The indexes move from right to left in the applet's viewing area. The applet freezes the ticker when the mouse button is pressed; it moves again when the mouse button is released.



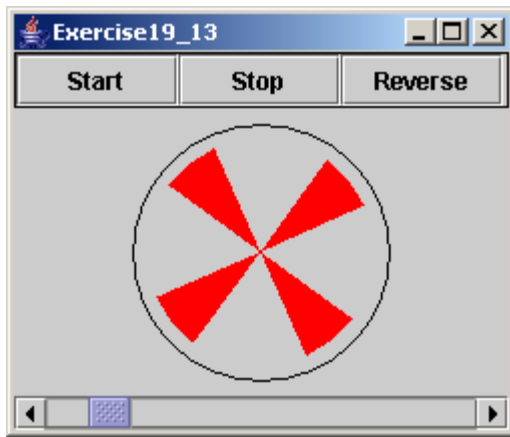
**Figure 19.26**

*The program displays a stock index ticker.*

19.13

(Showing a running fan) Write a Java applet that simulates a running fan, as shown in Figure 19.27. The buttons Start, Stop, and Reverse control the fan. The scrollbar controls the fan's speed. Create a class named `Fan`, a subclass of `JPanel`, to display the fan. This class also contains the methods to suspend and resume the fan, set its speed, and reverse its direction. Create a class named `FanControl` that contains a fan, and three buttons and a scroll bar to control the fan. Create a Java applet that contains an instance of `FanControl`. Enable the applet to run standalone. The relationships of these classes are shown in Figure

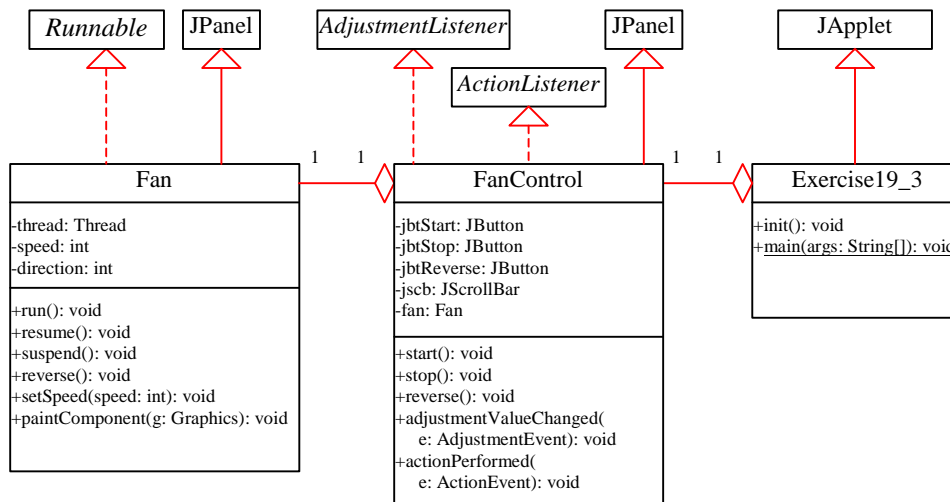
19.28.



**Figure 19.27**

*The program simulates a running fan.*

**\*\*\* Same as Fig 18.18 in introjb3e p813**



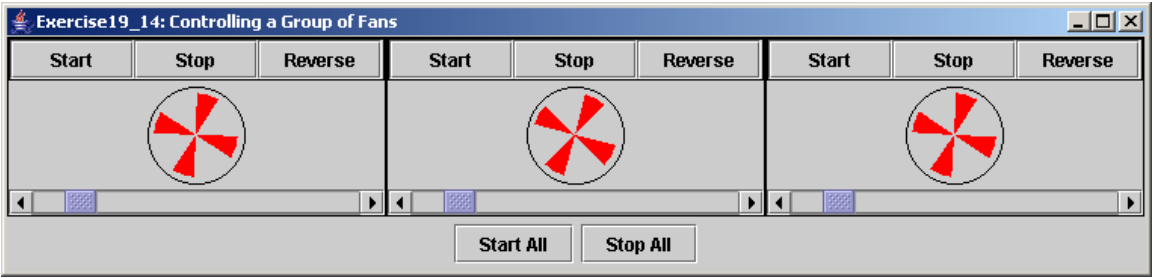
**Figure 19.28**

*The FanControl class contains a fan, three buttons, and a scroll bar to control the fan; and the Fan class displays a running fan.*

19.14

(Controlling a group of fans) Write a Java applet that displays three fans in a group, with control buttons to start and stop all of them, as shown in Figure 19.29. Use

the `FanControl` to control and display a single fan. Enable the applet to run standalone.

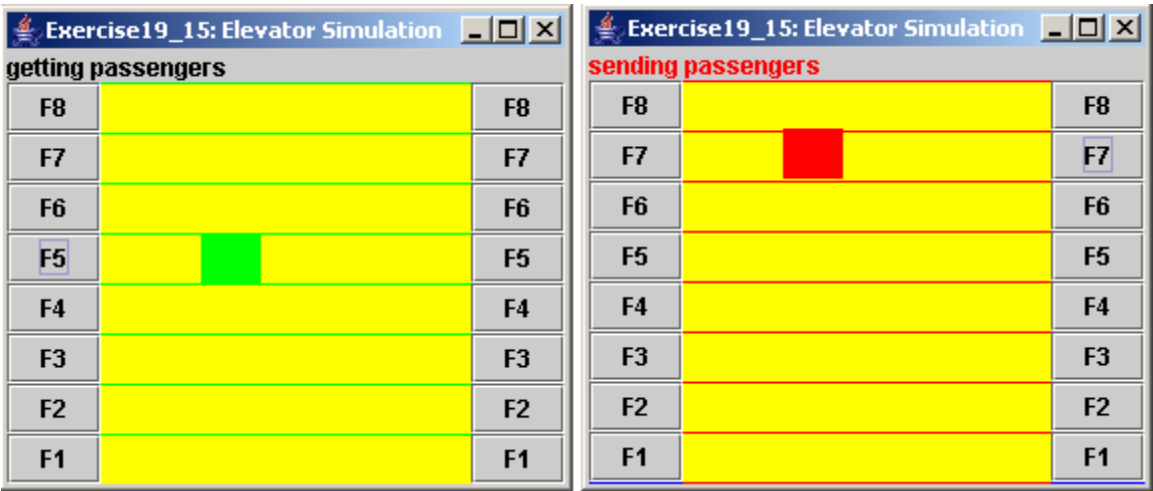


**Figure 19.29**

*The program runs and controls a group of fans.*

19.15

(Creating an elevator simulator) Write an applet that simulates an elevator going up and down (see Figure 19.30). The buttons on the left indicate the floor where the passenger is now located. The passenger must click a button on the left to request that the elevator come to his or her floor. On entering the elevator, the passenger clicks a button on the right to request that it go to the specified floor. Enable the applet to run standalone.



**Figure 19.30**

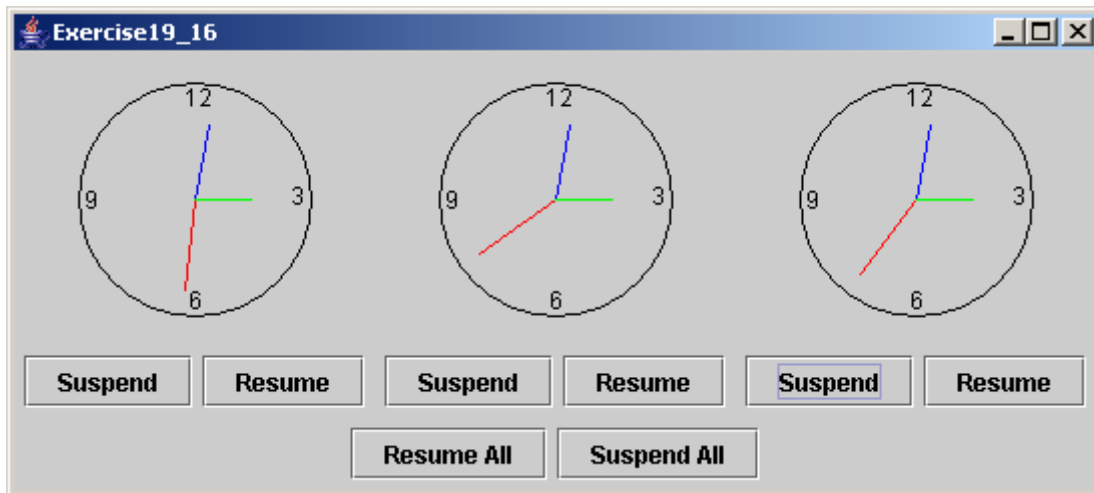
*The program simulates elevator operations.*

19.16

(Controlling a group of clocks) Write a Java applet that



displays three clocks in a group, with control buttons to start and stop all of them, as shown in Figure 19.31. Use the ClockControl to control and display a single clock. Enable the applet to run standalone.



**Figure 19.31**

*Three clocks run independently with individual control and group control.*

#### *Section 19.10 JProgressBar*

19.17 (Using JProgressBar) Create a program that displays an instance of JProgressBar and sets its value property randomly every 500 milliseconds infinitely.