Multithreading

threads can be executed simultaneously in multiprocessor systems, In single-processor systems, as shown in Figure 29.1(b), the multiple threads share CPU

time known as *time sharing*, and the operating system is responsible for scheduling and allocating

resources to them

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. In

Java, each task is an instance of the Runnable interface, also called a runnable object. A

thread is essentially an object that facilitates the execution of a task.

Tasks are objects. To create tasks, you have to first define a class for tasks. A task class must

implement the Runnable interface. 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 task must be executed in a thread.

The run() method in a task specifies how to perform the task. This method is automatically invoked

by the JVM. You should not invoke it. Invoking run() directly merely executes this method

in the same thread; no new thread is started.

You can use the yield() method to temporarily release time for other threads

The **sleep(long millis)** method puts the thread to sleep for the specified time in milliseconds

to allow other threads to execute.

The **sleep** method may throw an **InterruptedException**, which is a checked exception.

Such an exception may occur when a sleeping thread’s **interrupt()** method is called. The

**interrupt()** method is very rarely invoked on a thread, so an **InterruptedException** is

unlikely to occur. But since Java forces you to catch checked exceptions, you have to put it in a

try-catch block.

You can use the **join()** method to force one thread to wait for another thread to finish

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 JVM always picks the currently runnable thread with the highest priority. A lowerpriority

thread can run only when no higher-priority threads are running. If all runnable

threads have equal priorities, each is assigned an equal portion of the CPU time in a circular

queue. This is called the *round-robin scheduling*.

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 a lower or the same priority a chance to run.

Thread Synchronization

Obviously, the problem is that Task 1 and Task 2 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

The synchronized Keyword

To avoid race conditions, it is necessary to prevent more than one thread from simultaneously

entering a certain part of the program, known as the *critical region*.

adding the keyword **synchronized** in the **deposit** method

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