MULTITHREADED PROGRAMMING

This chapter presents multithreading, which is one of the core features supported by Java. The chapter introduces the need for expressing concurrency to support simultaneous operations within Java programs, especially those offering network services. It also introduces multithreading programming constructs in Java including synchronization techniques with examples.

OBJECTIVES

After learning the contents of this chapter, the reader must be able to :

- *understand the importance of concurrency*
- *understand multithreading in Java*
- *create user-defined classes with thread capability*
- *write multithreaded server programs*
- *understand the concurrent issues with thread programming*

CHAPTER

14

14.1 INTRODUCTION

In a networked world, it is common practice to share resources among multiple users. Therefore application programs designed to be deployed in a network should be designed to serve multiple users requests simultaneously. Even on desktop computers, users typically run multiple applications and carry out some operations in the background (e.g., printing) and some in the foreground (e.g., editing) simultaneously. With the increasing popularity of multicore processors, it is common to see even desktop and laptop computers with an ability to carry out multiple tasks concurrently. To meet these requirements, modern programming languages and operating systems are designed to support the development of applications containing multiple activities that can be executed concurrently.

Modern operating systems hold more than one activity (program) in memory and the processor can switch among all to execute them. This simultaneous occurrence of several activities on a computer is

known as *multitasking*. For example, you can open a file using MS Word and you can work on MS Access for creating your database. Two applications, MS Word and MS Access, are available in memory and the processor (by means of the operating system) switches to the application you are actively working on. Here two different applications are made to run concurrently by the same processor. The operating system supports multitasking in a cooperative or preemptive manner. In *cooperative multitasking* each application is responsible for relinquishing control to the processor to enable it to execute the other application. Earlier versions of operating systems followed cooperative multitasking. Modern operating systems such as Windows 95, Windows 98, Windows NT, and Windows 2000 support *preemptive multitasking*. In the preemptive type multitasking, the processor is responsible for executing each application in a certain amount of time called a *timeslice*. The processor then switches to the other applications, giving each its timeslice. The programmer is relieved from the burden of relinquishing control from one application to another. The operating system takes care of it.

A single processor computer is shared among multiple applications with preemptive multitasking. Since the processor is switching between the applications at intervals of milliseconds, you feel that all applications run concurrently. Actually, this is not so. To have true multitasking, the applications must be run on a machine with multiple processors. Multitasking results in effective and simultaneous utilization of various system resources such as processors, disks, and printers. As multitasking is managed by operating systems, we encourage readers to refer to books related to that topic. In this chapter, we focus on learning how to write an application containing multiple tasks (i.e., objects containing operations to be performed) that can be executed concurrently. In Java, this is realized by using multithreading techniques.

14.2 DEFINING THREADS

To understand multithreading, the concepts *process* and *thread* must be understood. A *process* is a program in execution. A process may be divided into a number of independent units known as *threads*. A *thread* is a dispatchable unit of work. Threads are *light-weight* processes within a process. A process is a collection of one or more threads and associated system resources. The difference between a process and a thread is shown in Fig.14.1. A process may have a number of threads in it. A thread may be assumed as a subset of a process.

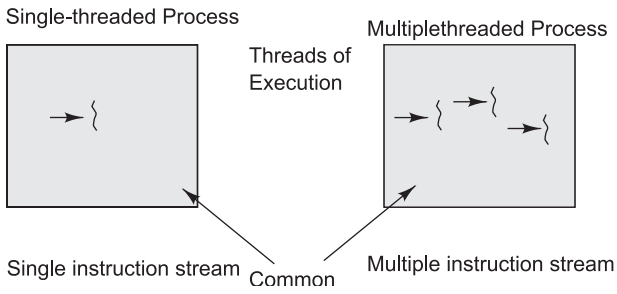

Fig. 14.1 A process containing single and multiple threads

If two applications are run on a computer (MS Word, MS Access), two processes are created. Multitasking of two or more processes is known as *process-based multitasking*. Multitasking of two or more threads is known as *thread-based multitasking*. The concept of multithreading in a programming language refers to thread-based multitasking. Process-based multitasking is totally controlled by the operating system. But thread-based multitasking can be controlled by the programmer to some extent in a program.

The concept of *context switching* is integral to threading. A hardware timer is used by the processor to determine the end of the timeslice for each thread. The timer signals at the end of the timeslice and in turn the processor saves all information required for the current thread onto a stack. Then the processor moves this information from the stack into a predefined data structure called a context structure. When the processor wants to switch back to a previously executing thread, it transfers all the information from the context structure associated with the thread to the stack. This entire procedure is known as *context switching*.

Java supports thread-based multitasking. The advantages of thread-based multitasking as compared to process-based multitasking are given below:

- Threads share the same address space.
- Context-switching between threads is normally inexpensive.
- Communication between threads is normally inexpensive.

14.3 THREADS IN JAVA

Applications are typically divided into processes during the design phase, and a master process explicitly spawns subprocesses when it makes sense to logically separate significant application functionalities. Processes, in other words, are an architectural construct. By contrast, a thread is a coding construct that does not affect the architecture of an application. A single process might contain multiple threads (see Fig. 14.2). All threads within a process share the same state and same memory space, and can communicate with each other directly, because they share the same variables.

Threads typically are spawned for a short-term benefit that is usually visualized as a serial task, but which does not have to be performed in a linear manner (such as performing a complex mathematical computation using parallelism, or initializing a large matrix), and then are absorbed when no longer required. The scope of a thread is within a specific code module—which is why we can bolt on threading without affecting the broader application.

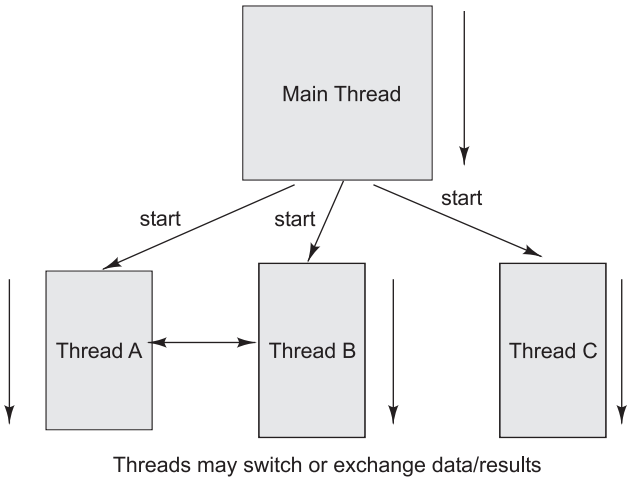

Fig. 14.2 A program with master thread and children threads

Threads are objects in the Java language. They can be created by using two different mechanisms as illustrated in Fig. 14.3:

1. Create a class that extends the standard *Thread* class.
2. Create a class that implements the standard *Runnable* interface.

That is, a thread can be defined by extending the `java.lang.Thread` class (see Fig. 14.3a) or by implementing the `java.lang.Runnable` interface (see Fig. 14.3b). The `run()` method should be overridden and should contain the code that will be executed by the new thread. This method must be public with a `void` return type and should not take any arguments. Both threads and processes are abstractions for parallelizing an application. However, processes are independent execution units that contain their own state information, use their own address spaces, and only interact with each other via interprocess communication mechanisms (generally managed by the operating system).

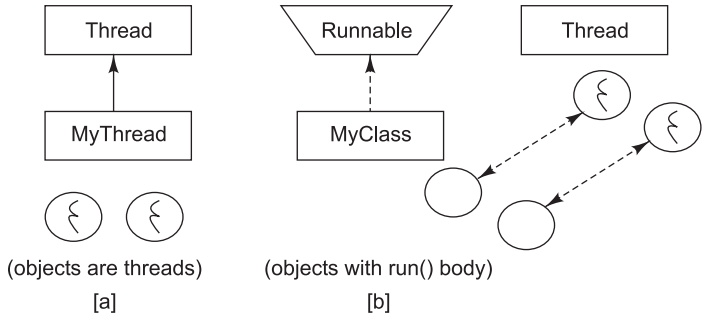

Fig. 14.3 Creation of Threads in Java

14.3.1 Extending the Thread Class

The steps for creating a thread by using the first mechanism are:

1. Create a class by extending the `Thread` class and override the `run()` method:

```
class MyThread extends Thread {
   public void run() {

        // thread body of execution
   }
}
```

2. Create a thread object:

```
MyThread thr1 = new MyThread();
```

3. Start Execution of created thread:

```
thr1.start();
```

An example program illustrating creation and invocation of a thread object is given below:

Program 14.1

```
/* ThreadEx1.java: A simple program creating and invoking a thread object by
extending the standard Thread class. */
class MyThread extends Thread {
   public void run() {
      System.out.println(" this thread is running ... ");
   }
}
class ThreadEx1 {
   public static void main(String [] args  ) {
     MyThread t = new MyThread();
     t.start();
   }
}
```

The class `MyThread` extends the standard `Thread` class to gain thread properties through inheritance. The user needs to implement their logic associated with the thread in the `run()` method, which is the body of thread. The objects created by instantiating the class `MyThread` are called threaded objects. Even though the execution method of thread is called `run`, we do not need to explicitly invoke this method directly. When the `start()` method of a threaded object is invoked, it sets the concurrent execution of the object from that point onward along with the execution of its parent thread/method.

14.3.2 Implementing the Runnable Interface

The steps for creating a thread by using the second mechanism are:

1. Create a class that implements the interface `Runnable` and override `run()` method:
   ```
   class MyThread implements Runnable {
      …
      public void run() {
         // thread body of execution
      }
   }
   ```
2. Creating Object:
   ```
   MyThread myObject = new MyThread();
   ```
3. Creating Thread Object:
   ```
   Thread thr1 = new Thread(myObject);
   ```
4. Start Execution:
   ```
   thr1.start();
   ```
   An example program illustrating creation and invocation of a thread object is given below:

Program 14.2

```
/* ThreadEx2.java: A simple program creating and invoking a thread object by
implementing Runnable interface. */
class MyThread implements Runnable  {
   public void run() {
      System.out.println(" this thread is running ... ");
```

```
    }
}
class ThreadEx2 {
   public static void main(String [] args  ) {
      Thread t = new Thread(new MyThread());
      t.start();
   }
}
```

The class `MyThread` implements standard `Runnable` interface and overrides the `run()` method and includes logic associated with the body of the thread (step 1). The objects created by instantiating the class `MyThread` are normal objects (unlike the first mechanism) (step 2). Therefore, we need to create a generic `Thread` object and pass `MyThread` object as a parameter to this generic object (step 3). As a result of this association, threaded object is created. In order to execute this threaded object, we need to invoke its `start()` method which sets execution of the new thread (step 4).

14.3.3 Thread Class versus Runnable Interface

It is a little confusing why there are two ways of doing the same thing in the threading API. It is important to understand the implication of using these two different approaches. By extending the thread class, the derived class itself is a thread object and it gains full control over the thread life cycle. Implementing the Runnable interface does not give developers any control over the thread itself, as it simply defines the unit of work that will be executed in a thread. Another important point is that when extending the Thread class, the derived class cannot extend any other base classes because Java only allows single inheritance. By implementing the Runnable interface, the class can still extend other base classes if necessary. To summarize, if the program needs a full control over the thread life cycle, extending the Thread class is a good choice, and if the program needs more flexibility of extending other base classes, implementing the Runnable interface would be preferable. If none of these is present, either of them is fine to use.

14.4 THREAD LIFE CYCLE

The life cycle of threads in Java is very similar to the life cycle of processes running in an operating system. During its life cycle the thread moves from one state to another depending on the operation performed by it or performed on it as illustrated in Fig. 14.4. A Java thread can be in one of the following states:

- *NEW*
  A thread that is just instantiated is in *new* state. When a `start()` method is invoked, the thread moves to the ready state from which it is automatically moved to runnable state by the thread scheduler.
- *RUNNABLE* (ready→running)
  A thread executing in the JVM is in running state.
- *BLOCKED*
  A thread that is blocked waiting for a monitor lock is in this state. This can also occur when a thread performs an I/O operation and moves to next (runnable) state.
- *WAITING*
  A thread that is waiting indefinitely for another thread to perform a particular action is in this state.
- *TIMED_WAITING* (sleeping)
  A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state.

- *TERMINATED* (dead)
  A thread that has exited is in this state.

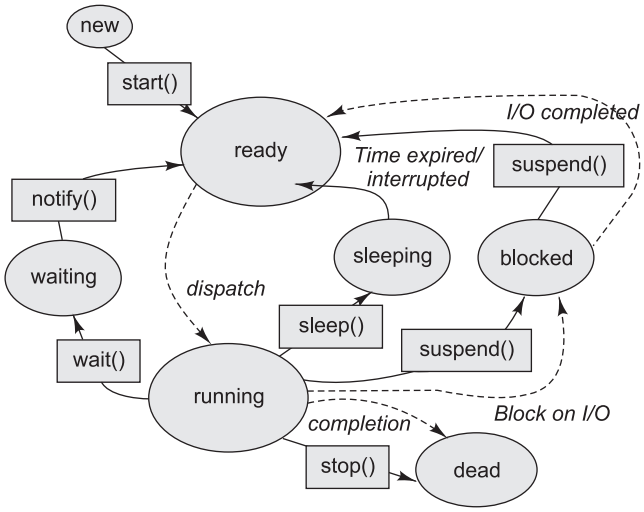

Fig. 14.4 Life cycle of Java threads

At any given time, a thread can be in only one state. These states are JVM states as they are not linked to operating system thread states.

When the object of a user `Thread` class is created, the thread moves to the `NEW` state. After invocation of the `start()` method, the thread shifts from the `NEW` to the ready (`RUNNABLE`) state and is then dispatched to running state by the JVM thread scheduler. After gaining a chance to execute, the `run()` method will be invoked. It should be noted that when the `run()` method is invoked directly (explicitly in the program), the code in it is executed by the current thread and not by the new thread (as the thread object is not assigned to the virtual machine scheduler). Depending on program operations or invocation of methods such as `wait()`, `sleep()`, and `suspend()` or an I/O operation, the thread moves to the `WAITING`, `SLEEPING`, and `BLOCKED` states respectively. It should be noted that the thread is still alive. After the completion of an operation that blocked it (I/O or sleep) or receiving an external signal that wakes it up, the thread moves to ready state. Then the JVM thread scheduler moves it to running state in order to continue the execution of remaining operations. When the thread completes its execution, it will be moved to `TERMINATED` state. A dead thread can never enter any other state, not even if the `start()` method is invoked on it.

14.5 A JAVA PROGRAM WITH MULTIPLE THREADS

To illustrate creation of multiple threads in a program performing concurrent operations, let us consider the processing of the following mathematical equation:

$$p = \sin(x) + \cos(y) + \tan(z)$$

As these trigonometric functions are independent operations without any dependencies between them, they can be executed concurrently. After that their results can be combined to produce the final result as illustrated in Fig. 14.5.

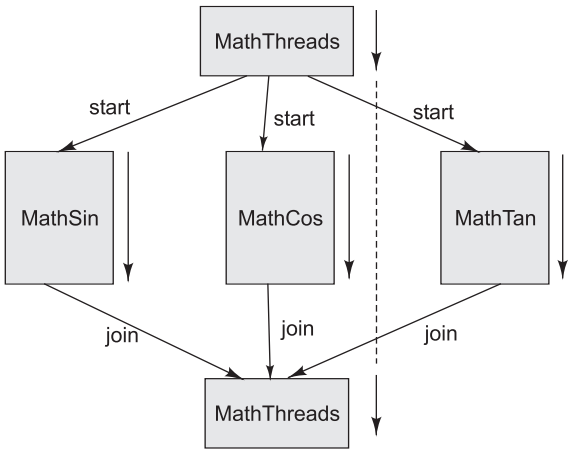

Fig. 14.5 Flow of control in a master and multiple workers threads application

The main/default thread is called `MathThreads`, which acts like a master thread. It creates three worker threads (`MathSin`, `MathCos`, and `MathTan`) and assigns them to compute values for different data inputs. All three worker threads are concurrently executed on shared or dedicated CPUs depending on the type of machine. Although the master thread can continue its execution, in this case, it needs to make sure that all operations are completed before combining individual results. This is accomplished by waiting for each thread to complete by invoking `join()` method associated with each worker thread. A complete Java program illustrating this concept is given in Program 14.3.

Program 14.3

```
/* MathThreads.java: A program with multiple threads performing concurrent
operations. */
import java.lang.Math;
class MathSin extends Thread {
   public double deg;
   public double res;

   public MathSin(int degree) {
     deg = degree;
   }
   public void run() {
            System.out.println("Executing sin of "+deg);
      double Deg2Rad = Math.toRadians(deg);
      res = Math.sin(Deg2Rad);
      System.out.println("Exit from MathSin. Res = "+res);
   }
}
class MathCos extends Thread {
   public double deg;
   public double res;
```

```
   public MathCos(int degree) {
      deg = degree;
   }
   public void run() {
      System.out.println("Executing cos of "+deg);
      double Deg2Rad = Math.toRadians(deg);
      res = Math.cos(Deg2Rad);
      System.out.println("Exit from MathCos. Res = "+res);
   }
}
class MathTan extends Thread {
   public double deg;
   public double res;

   public MathTan(int degree) {
      deg = degree;
   }
   public void run() {
      System.out.println("Executing tan of "+deg);
      double Deg2Rad = Math.toRadians(deg);
      res = Math.tan(Deg2Rad);
      System.out.println("Exit from MathTan. Res = "+res);
   }
}
class MathThreads {
   public static void main(String args[]) {
      MathSin st = new MathSin(45);
      MathCos ct = new MathCos(60);
      MathTan tt = new MathTan(30);
      st.start();
      ct.start();
      tt.start();
      try {   // wait for completion of all thread and then sum
        st.join();
        ct.join(); //wait for completion of MathCos object
        tt.join();
        double z = st.res + ct.res + tt.res;
        System.out.println("Sum of sin, cos, tan = "+z);
      }
      catch(InterruptedException IntExp) {
      }
   }
}
```

Run 1:

```
[raj@mundroo] threads [1:111] java MathThreads
Executing sin of 45.0
Executing cos of 60.0
Executing tan of 30.0
```

```
Exit from MathSin. Res = 0.7071067811865475
Exit from MathCos. Res = 0.5000000000000001
Exit from MathTan. Res = 0.5773502691896257
Sum of sin, cos, tan = 1.7844570503761732

Run 2:
[raj@mundroo] threads [1:111] java MathThreads
Executing sin of 45.0
Executing tan of 30.0
Executing cos of 60.0
Exit from MathCos. Res = 0.5000000000000001
Exit from MathTan. Res = 0.5773502691896257
Exit from MathSin. Res = 0.7071067811865475
Sum of sin, cos, tan = 1.7844570503761732

Run 3:
[raj@mundroo] threads [1:111] java MathThreads
Executing cos of 60.0
Executing sin of 45.0
Executing tan of 30.0
Exit from MathCos. Res = 0.5000000000000001
Exit from MathTan. Res = 0.5773502691896257
Exit from MathSin. Res = 0.7071067811865475
Sum of sin, cos, tan = 1.7844570503761732
```

Observations on Results A close observation of the output of three different runs reveals that the execution/completion of threads does not need to be in the order of their start (i.e., invocation of `start()` method). In Run 1, it so happened that the output of all three threads happen to be generated in the same order of their start. However, this is not the case in the next two runs. Because of various operations that are performed within JVM, threads, and different applications running at different times, it is hard to predict the precise time at which context-switching occurs. It is possible to enforce the order of execution to some extent by setting different priorities for threads.

14.6 THREAD PRIORITY

In Java, all the thread instances the developer created have the same priority, which the process will schedule fairly without worrying about the order. It is important for different threads to have different priorities. Important threads should always have higher priority than less important ones, while threads that need to run quietly as a Daemon may only need the lowest priority. For example, the garbage collector thread just needs the lowest priority to execute, which means it will not be executed before all other threads are scheduled to run. It is possible to control the priority of the threads by using the Java APIs. The `Thread.setPriority(…)` method serves this purpose. The Thread class provides 3 constants value for the priority:

```
MIN_PRIORITY = 1, NORM_PRIORITY = 5, MAX_PRIORITY = 10
```

The priority range of the thread should be between the minimum and the maximum number. The following example shows how to alter the order of the thread by changing its priority.

Program 14.5

```
/* ThreadPriorityDemo.java: A program which shows altering order of threads
by changing priority. */
class A extends Thread {
   public void run() {
      System.out.println("Thread A started");
      for(int i = 1; i <= 4; i++) {
        System.out.println("\t From ThreadA: i= " + i);
      }
      System.out.println("Exit from A");
   }
}
class B extends Thread {
   public void run() {
      System.out.println("Thread B started");
      for(int j = 1; j <= 4; j++) {
        System.out.println("\t From ThreadB: j= " + j);
      }
      System.out.println("Exit from B");
   }
}
class C extends Thread {
   public void run() {
      System.out.println("Thread C started");
      for(int k = 1; k <= 4; k++) {
        System.out.println("\t From ThreadC: k= " + k);
      }
      System.out.println("Exit from C");
   }
}
public class ThreadPriorityDemo {
   public static void main(String args[]) {
      A threadA = new A();
      B threadB = new B();
      C threadC = new C();
      threadC.setPriority(Thread.MAX_PRIORITY);
      threadB.setPriority(threadA.getPriority() + 1);
      threadA.setPriority(Thread.MIN_PRIORITY);
      System.out.println("Started Thread A");
      threadA.start();

      System.out.println("Started Thread B");
      threadB.start();
      System.out.println("Started Thread C");
      threadC.start();
      System.out.println("End of main thread");
   }
}
```

This example creates 3 threads, and adjusts the priority of each thread. The possible output of the example is shown as follows:

```
Started Thread A
Started Thread B
Started Thread C
Thread B started
End of main thread
Thread C started
    From ThreadC: k= 1
    From ThreadC: k= 2
    From ThreadC: k= 3
    From ThreadC: k= 4
Exit from C
    From ThreadB: j= 1
    From ThreadB: j= 2
    From ThreadB: j= 3
    From ThreadB: j= 4
Exit from B
Thread A started
    From ThreadA: i= 1
    From ThreadA: i= 2
    From ThreadA: i= 3
    From ThreadA: i= 4
Exit from A
```

The highest priority thread C is scheduled/executed first and the thread A which has the lowest priority is scheduled last.

14.7 THREAD METHODS

The `sleep()` method causes the current thread to sleep for a specified amount of time in milliseconds:

```
public static void sleep(long millis) throws InterruptedException
```

For example, the code below puts the thread in sleep state for 3 minutes:

```
try {
     Thread.sleep(3 * 60 * 1000); // thread sleeps for 3 minutes
} catch(InterruptedException ex){}
```

The `yield()` method causes the current thread to move from the running state to the `RUNNABLE` state, so that other threads may get a chance to run. However, the next thread chosen for running might not be a different thread:

```
public static void yield()
```

The `isAlive()` method returns true if the thread upon which it is called has been started but not moved to the dead state:

```
public final boolean isAlive()
```

When a thread calls `join()` on another thread, the currently running thread will wait until the thread it joins with has completed. It is also possible to wait for a limited amount of time instead for the thread completion.

```
void join()
void join(long millis)
void join(long millis, int nanos)
```

A thread exists in a thread group and a thread group can contain other thread groups. This example visits all threads in all thread groups.

```
// Find the root thread group
ThreadGroup root = Thread.currentThread().getThreadGroup().getParent();
while (root.getParent() != null) {
  root = root.getParent();
}
// Visit each thread group
visit(root, 0);

// This method recursively visits all thread groups under the given group.
public static void visit(ThreadGroup group, int level) {
   // get threads in the given group
   int numThreads = group.activeCount();
   Thread[] threads = new Thread[numThreads*2];
   numThreads = group.enumerate(threads, false);

   // enumerate each thread in 'group'
   for(int i=0; i<numThreads; i++) {
       // get thread
       Thread thread = threads[i];
   }

   // get thread subgroups of the given group
   int numGroups = group.activeGroupCount();
   ThreadGroup[] groups = new ThreadGroup[numGroups*2];
   numGroups = group.enumerate(groups, false);

   // recursively visit each subgroup
   for(int i=0; i<numGroups; i++) {
       visit(groups[i], level+1);
   }
}
```

Here is an example of some thread groups that contain some threads:

```
java.lang.ThreadGroup[name=system,maxpri=10]
   Thread[Reference Handler,10,system]
   Thread[Finalizer,8,system]
   Thread[Signal Dispatcher,10,system]
   Thread[CompileThread0,10,system]
java.lang.ThreadGroup[name=main,maxpri=10]
   Thread[main,5,main]
   Thread[Thread-1,5,main]
```

Another pair of methods is used for daemon threads. Threads that work in the background to support the runtime environment are called daemon threads. The setDaemon method is used to make the current thread

as a daemon thread, and the isDaemon method is used to identify whether the current thread is a daemon thread or not.

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

The usage of these two methods is intuitive. However, one important thing to notice is that a daemon thread does not mean its priority is very low, in contrast, a thread with minimum priority is not a daemon thread unless the setDaemon method is used to set it.

14.8 MULTITHREADED MATH SERVER

It is time to implement a more comprehensive threaded application by utilizing the thread programming you have learned so far. A sample multithreaded math interaction demonstrating an online math server that can perform basic math operations serving clients making simultaneous requests is shown in Fig. 14.6. The multithreaded math server extends the math server from the previous Socket chapter, it enables the math server to concurrently accept client requests and to open a new thread for each socket connection by implementing the `java.lang.Runnable` interface. The following code shows how it is implemented.

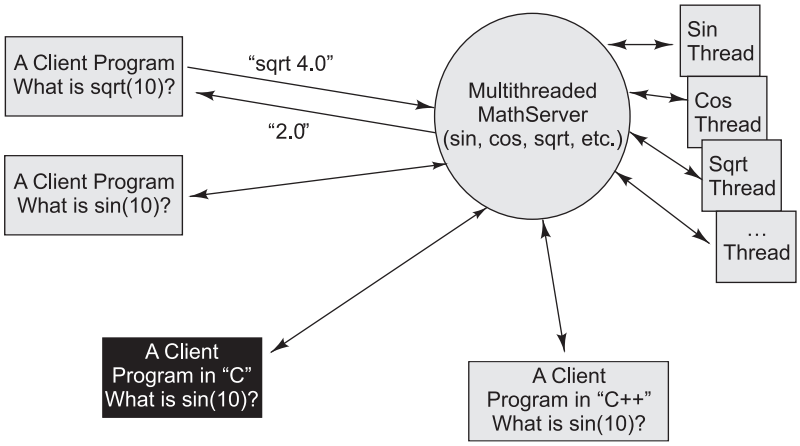

Fig. 14.6 A multithreaded math server and its clients

Program 14.4

```
/* MultiThreadMathServer.java: A program extending MathServer which
allows concurrent client requests and opens a new thread for each socket
connection. */
import java.net.ServerSocket;
import java.net.Socket;
public class MultiThreadMathServer
             extends MathServer implements Runnable {
   public void run() {
```

```
        execute();
    }
    public static void main(String [] args)throws Exception {
      int port = 10000;
      if (args.length == 1) {
          try {
             port = Integer.parseInt(args[0]);
          }
           catch(Exception e) {
          }
      }
      ServerSocket serverSocket = new ServerSocket(port);
      while(true){
        //waiting for client connection
        Socket socket = serverSocket.accept();
        socket.setSoTimeout(14000);
        MultiThreadMathServer server = new MultiThreadMathServer();
        server.setMathService(new PlainMathService());
        server.setSocket(socket);
        //start a new server thread...
        new Thread(server).start();
      }
    }
}
```

As can be seen from the program, the run method just invokes the execute method from the base `MathServer` class. The main method of the `MultiThreadMathServer` has a infinite while loop when a client request comes, it creates a new instance of the `MultiThreadMathServer` class and starts it as a separate thread. There is no need to change the client code as the multithreaded version of the Math service is totally transparent to the client. The purpose of implementing a multithreaded math server is to enable concurrent client connections, which means it now can support multiple clients at the same time compared with the single thread version. One more thing that needs to be mentioned is that every client socket has been explicitly set at a 14-seconds timeout in order to release critical resources if it is waiting for too long.

14.9 CONCURRENT ISSUES WITH THREAD PROGRAMMING

Threading is a very powerful technique which is sometimes very hard to control, especially when it is accessing shared resources. In such cases, the threads have to be coordinated, otherwise it will violate the data of the whole application. For example, a printer cannot be used to print two documents at the same time and if there are multiple printing requests, threads managing these printing operations needs to be coordinated. Another example would be simultaneously operating the same bank account: it is not correct to do both deposit and withdraw operations on a bank account at the same time. There are two very basic problems related to concurrent issues when dealing with multiple threads: read/write problem and producer-consumer problem.

14.9.1 Read/Write Problem

If one thread tries to read the data and another thread tries to update the same data, it is known as read/write problem, and it leads to inconsistent state for the shared data. This can be prevented by synchronizing access to the data via Java `synchronized` keyword. For example,

```
public synchronized void update() {
…
}
```

It is better to explain the synchronized approach to access the shared data via a simple example. Consider an example of a bank offering online access to its customers to perform transactions on their accounts from anywhere in the world, as shown in Fig. 14.7. It is possible that an account holder has given a cheque to some other account holder. If the cheque receipient happens to withdraw the money at the same time the orginal account holder deposits the money, both parties are performing simultaneous operations on the same account (as shown in Program 14.7). This situation can lead to incorrect operation on the account. This can be avoided by sychronization, which ensures that only one person is able to perform an operation on a shared data at a time (i.e., in a way operations are sequenced to avoid data inconsistency problems). The following class shows a typical invocation of banking operations via multiple threads.

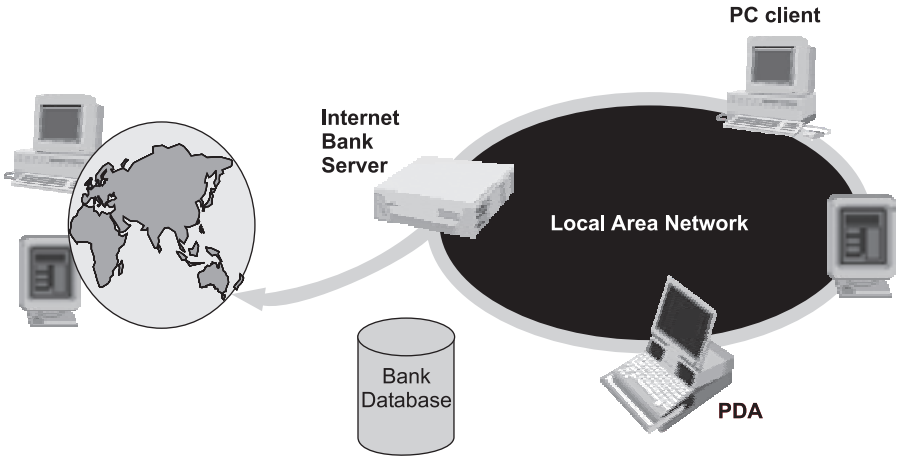

Fig. 14.7 Online Bank: Serving Many Customers and Operations

Program 14.6

```
/* InternetBankingSystem.java: A simple program showing a typical invocation of
banking operations via multiple threads. */
public class InternetBankingSystem {
  public static void main(String [] args  ) {
    Account accountObject = new Account(100);
    new Thread(new DepositThread(accountObject,30)).start();
    new Thread(new DepositThread(accountObject,20)).start();
    new Thread(new DepositThread(accountObject,10)).start();
    new Thread(new WithdrawThread(accountObject,30)).start();
```

```
    new Thread(new WithdrawThread(accountObject,50)).start();
    new Thread(new WithdrawThread(accountObject,20)).start();
  } // end main()
  }
```

There are 6 threads that access the same `Account` object simultaneously. The following code shows the operations that will be performed on each thread.

Program 14.7

```
/* WithdrawThread.java A thread that withdraw a given amount from a given
account. */
package com.javabook.threading;
public class WithdrawThread implements Runnable {
   private Account account;
   private double amount;

   public WithdrawThread(Account account, double amount) {
       this.account = account;
       this.amount = amount;
   }

   public void run() {
       //make a withdraw
       account.withdraw(amount);
   }
}//end WithdrawThread class
/* DepositThread.java A thread that deposit a given amount to a given
account. */

package com.javabook.threading;

public class DepositThread implements Runnable {
   private Account account;
   private double amount;

   public DepositThread(Account account, double amount) {
       this.account = account;
       this.amount = amount;
   }

   public void run() {
       //make a deposit
       account.deposit(amount);
   }
}//end DepositThread class
```

As can be seen (in Program 14.6), the three deposit threads are performing deposit operations on an account and simultaneously three Withdraw threads are withdrawing from the same account. If the account object is not synchronized properly, the outcome may be unpredictable while these operations are

performing simultaneously. To ensure the proper behavior of the application, it is necessary to synchronize the methods on the `Account` class as illustrated in Program 14.8.

Program 14.8

```
/* Account.java: A program with synchronized methods for the Account class.
*/
package com.javabook.threading;
public class Account {
   private double balance = 0;

   public Account(double balance) {
      this.balance = balance;
   }

   // if 'synchronized' is removed, the outcome is unpredictable
   public synchronized void deposit(double amount) {
      if (amount < 0) {
         throw new IllegalArgumentException("Can't deposit.");
      }
      this.balance += amount;
      System.out.println("Deposit "+amount+" in thread"
            +Thread.currentThread().getId()
            +", balance is " +balance);
   }

   // if 'synchronized' is removed, the outcome is unpredictable
   public synchronized void withdraw(double amount) {
      if (amount < 0 || amount > this.balance) {
         throw new IllegalArgumentException("Can't withdraw.");
      }
      this.balance -= amount;
      System.out.println("Withdraw "+amount+" in thread "
            + Thread.currentThread().getId()
            + ", balance is "+balance);
   }
}//end Account class
```

The synchronized method will ensure that only one class can access the data at one time, other operations have to wait until the first operation finishes. Simply run the InternetBankingSystem program, the following results should be shown on the systen console:

```
Deposit 30.0 in thread 8,new balance is 130.0
Withdraw 50.0 in thread 12,new balance is 80.0
Deposit 10.0 in thread 10,new balance is 90.0
Withdraw 20.0 in thread 13,new balance is 70.0
Withdraw 30.0 in thread 11,new balance is 40.0
Deposit 20.0 in thread 9,new balance is 60.0
```

This is the expected behavior when even the order of the thread execution is unknown. However, if the account class is modified without the synchronized keyword placed on the methods that access the balance

data, unpredictable behavior will occur. Run the InternetBankingSystem program again by removing the synchronized keyword; run may generate the following printout in the system console.

```
Withdraw 30.0 in thread 11, balance is 60.0
Deposit 20.0 in thread 9, balance is 60.0
Withdraw 50.0 in thread 12, balance is 60.0
Deposit 10.0 in thread 10, balance is 60.0
Withdraw 20.0 in thread 13, balance is 60.0
Deposit 30.0 in thread 8, balance is 60.0
```

The balance of each transaction is totally unpredictable and wrong compared with the expected results.

14.9.2 Producer and Consumer Problem

The producer-consumer problem (also known as the bounded-buffer problem) is another classical example of a multithread synchronization problem. The problem describes two threads, the producer and the consumer, who share a common, fixed-size buffer. The producer's job is to generate a piece of data and put it into the buffer. The consumer is consuming the data from the same buffer simultaneously. The problem is to make sure that the producer will not try to add data into the buffer if it is full and that the consumer will not try to remove data from an empty buffer.

The solution for this problem involves two parts. The producer should wait when it tries to put the newly created product into the buffer until there is at least one free slot in the buffer. The consumer, on the other hand, should stop consuming if the buffer is empty. Take a message queue as an example, the following code shows a simple implementation of such a message queue.

Program 14.9

```
/* MessageQueue.java: A message queue with synchronized methods for queuing
and consuming messages. */

package com.javabook.threading;
import java.util.ArrayList;
import java.util.List;

public class MessageQueue {
   //the size of the buffer
   private int bufferSize;

   //the buffer list of the message, assuming the string message format
   private List<String> buffer = new ArrayList<String>();

   //construct the message queue with given buffer size
   public MessageQueue(int bufferSize){
      if(bufferSize<=0)
         throw new IllegalArgumentException("Size is illegal.");
      this.bufferSize = bufferSize;
   }
   //check whether the buffer is full
   public synchronized boolean isFull() {
      return buffer.size() == bufferSize;
```

```
    }

    //check whether the buffer is empty
    public synchronized boolean isEmpty() {
        return buffer.isEmpty();
    }

    //put an income message into the queue, called by message producer
    public synchronized void put(String message) {
        //wait until the queue is not full
        while (isFull()) {
            System.out.println("Queue is full.");
            try{
                //set the current thread to wait
                wait();
            }catch(InterruptedException ex){
                //someone wake me up.
            }
        }
        buffer.add(message);
        System.out.println("Queue receives message '"+message+"'");

        //wakeup all the waiting threads to proceed
        notifyAll();
    }

    //get a message from the queue, called by the message consumer
    public synchronized String get(){
        String message = null;
        //wait until the queue is not empty
        while(isEmpty()){
            System.out.println("There is no message in queue.");
            try{
                //set the current thread to wait
                wait();
            }catch(InterruptedException ex){
                //someone wake me up.
            }
        }
        //consume the first message in the queue
        message = buffer.remove(0);

        //wakeup all the waiting thread to proceed
        notifyAll();
        return message;
    }
} //end MessageQueue class
```

The put method of the message queue object will be called by the message producer and in case the buffer is full, it invokes the wait method from the Object class, which causes the current thread to wait

until another thread invokes a notify() or notifyAll() method for this message queue object. All the other producer threads who are trying to put messages into the queue will be waiting until the notifyAll method is called by another consumer thread. If the message queue is not full, this method will skip the loop and enqueue the message into the buffer, and notify all the consumer threads who are trying to get income messages from the empty buffer, if there is any.

The get method of the message queue object will be called by the message consumer and in case the buffer is empty, it invokes the wait method similarly as in the put method, and all of the following consumer threads who are trying to consume messages from the empty buffer will wait until notifyAll method on this message queue is called by another producer thread. If the message queue is not empty, this method will skip the loop and dequeue a message from the queue, notify all the waiting producer threads who are waiting for free buffer slots, and send back the message to the consumer, if there is any.

The synchronized keyword is essential as both get/put methods are accessing the shared data which is the message queue buffer list, it will be protected against the first read/write problem. The implementation of the producer and the consumer now is pretty straightforward as the message queue is very well-protected to prevent the concurrent problems. Program 14.10 lists the implementation of the producer class.

Program 14.10

```
/* Producer.java: A producer that generates messages and put into a given
message queue. */

package com.javabook.threading;

public class Producer extends Thread{
   private static int count = 0;
   private MessageQueue queue = null;

   public Producer(MessageQueue queue){
      this.queue = queue;
   }

   public void run(){
      for(int i=0;i<10;i++){
         queue.put(generateMessage());
      }
   }

   private synchronized String generateMessage(){
      String msg = "MSG#"+count;
      count ++;

      return msg;
   }
} //end Producer class
```

The run method generates 10 messages for each producer, and synchronization is used to protect the static count field that will be accessed by multiple producer threads at the same time when generating messages. The consumer is even simpler as shown in Program 14.11.

Program 14.11

```
/* Consumer.java: A consumer that consumes messages from the queue. */
package com.javabook.threading;

public class Consumer extends Thread {
   private MessageQueue queue = null;

   public Consumer(MessageQueue queue){
       this.queue = queue;
   }

   public void run(){
       for(int i=0;i<10;i++){
           System.out.println("Consumer downloads "
              +queue.get()+ " from the queue.");
       }
   }
} //end Consumer class
```

The run method gets messages from the message queue for 10 times and prints a message if there is any. Otherwise, it will wait until the next available message is ready to pick up in the queue. Program 14.12 shows how to assemble the message queue, producer, and consumer together as a message system demo program.

Program 14.12

```
/* MessageSystem.java: A message system that demonstrate the produce and
consumer problem with the message queue example. */

package com.javabook.threading;

public class MessageSystem {
   public static void main(String[] args) {
       MessageQueue queue = new MessageQueue(5);
       new Producer(queue).start();
       new Producer(queue).start();
       new Producer(queue).start();
       new Consumer(queue).start();
       new Consumer(queue).start();
       new Consumer(queue).start();
   }
} //end MessageSystem class
```

14.10 SUMMARY

Multithreading is one of the core features supported by Java. It allows creation of multiple objects that can simultaneously execute different operations. It is often used in writing programs that perform different tasks asynchronously such as printing and editing. Network applications that are responsible for serving remote clients requests have been implemented as multithreaded applications.

14.11 EXERCISES

Objective Questions

14.1 A _____ is a program in execution. A _____ is a dispatchable unit of work.
14.2 There are two ways to define a thread in Java.
i. Define an object that extends _____ class.
ii. Define an object than implements _____ interface.
14.3 The _____ method in the thread class starts a thread.
14.4 The _____ method in the Object class will wait until the current thread completes.
14.5 Thread.sleep(100) will make the current thread sleep for _____ seconds.
14.6 Thread priority can be changed by using _____ method.
14.7 _____ keyword can be used to protect accessing the shared data.
14.8 The producer and consumer problem is also known as _____.
14.9 The thread is going to its wait state if the method _____ is called.
14.10 The thread will be waked up from the wait state to running state if _____ or _____ is called.
14.11 A process can contains more than one thread at the same time: True or False.
14.12 The start method of the Thread class needs to be overloaded by the subclass of Thread: True or False.
14.13 Java does not allow developers to control the Thread life cycle: True or False
14.14 Thread.run() can be called to start a thread without problem: True or False
14.15 Calling notify method, no exception will be generated by the waiting thread: True or False
14.16 To make a thread as a daemon thread, developers can either use setDaemon method or set The thread priority to the minimum number: True or False
14.17 The *synchronized* keyword has to be used for every method with multithread access: True or False.
14.18 To execute the following code, thread 1 will run first: True or False

```
Thread thread1 = new Thread();
Thread thread2 = new Thread();
Thread thread3 = new Thread();
thread1.start();
thread2.start();
thread3.start();
```

14.19 The read/write problem will not happen if the data is not shared between multiple threads: True or False.

14.20 The producer and consumer problem is caused by the producer who produces too many threads and the consumer cannot consume. True or False.

Review Questions

14.21 What are threads? Briefly explain the differences between a thread and a process.

14.22 Briefly explain the concept of *context-switching*.

14.23 What are the advantages of thread-based multitasking as compared to process-based multitasking?

14.24 Describe two possible ways to create threads in Java.

14.25 Briefly describe the life cycle of a thread in Java.

14.26 Assume the developer has created a thread class and that the main method is as follows:

```
public static void main(String [] args){
   Thread th1 = new MyThread();
   Thread th2 = new MyThread();
   th1.run();
   th2.run();
}
```

What will happen when executing this main method? Briefly describe the consequences.

14.27 Describe the basic methods that are supported by the `Thread` class.

14.28 Explain the meaning of the priority of the thread with a simple example.

Programming Problems

14.29 Implement a class that checks whether a given number is a prime using both the Thread class and Runnable interface

14.30 Write a simple Timer that can periodically print a timeout message.

14.31 Write a simulation program for the fruit market. The farmer will be able to produce different types of fruits (apple, orange, grape, and watermelon), and put them in the market to sell. The market has limited capacity and farmers have to stand in a queue if the capacity is exceeded to sell their fruits. Consumers can come to the market any time and purchase their desired fruits; and if the fruits they want to buy runs out, they are willing to wait until the supply of that kind is ready. (Hint: implementing this market will encounter the producer and consumer problem, and it probably needs multiple buffers for different kinds of fruits).

Mini Project

14.32 Implement the basic functions for the Online Banking Application based on the discussion in this chapter. Make sure the implementation is thread safe (all the shared data needs to be properly synchronized).