Chapter 12

Multithreaded Programming

12.1 Introduction

Those who are familiar with the modern operating systems such as Windows 95 and Windows XP may recognize that they can execute several programs simultaneously. This ability is known as *multitasking*. In system's terminology, it is called *multithreading*.

Multithreading is a conceptual programming paradigm where a program (process) is divided into two or more subprograms (processes), which can be implemented at the same time in parallel. For example, one subprogram can display an animation on the screen while another may build the next animation to be displayed. This is something similar to dividing a task into subtasks and assigning them to different people for execution independently and simultaneously.

In most of our computers, we have only a single processor and therefore, in reality, the processor is doing only one thing at a time. However, the processor switches between the processes so fast that it appears to human beings that all of them are being done simultaneously.

Java programs that we have seen and discussed so far contain only a single sequential flow of control. This is what happens when we execute a normal program. The program begins, runs through a sequence of executions, and finally ends. At any given point of time, there is only one statement under execution.

A *thread* is similar to a program that has a single flow of control. It has a beginning, a body, and an end, and executes commands sequentially. In fact, all main programs in our earlier examples can be called *single-threaded* programs. Every program will have at least one thread as shown in Fig. 12.1.

A unique property of Java is its support for multithreading. That is, Java enables us to use multiple flows of control in developing programs. Each flow of control may be thought of as a separate tiny

program (or module) known as a *thread* that runs in parallel to others as shown in Fig. 12.2. A program that contains multiple flows of control is known as *multithreaded program*. Figure 12.2 illustrates a Java program with four threads, one main and three others. The main thread is actually the **main** method module, which is designed to create and start the other three threads, namely A, B and C.

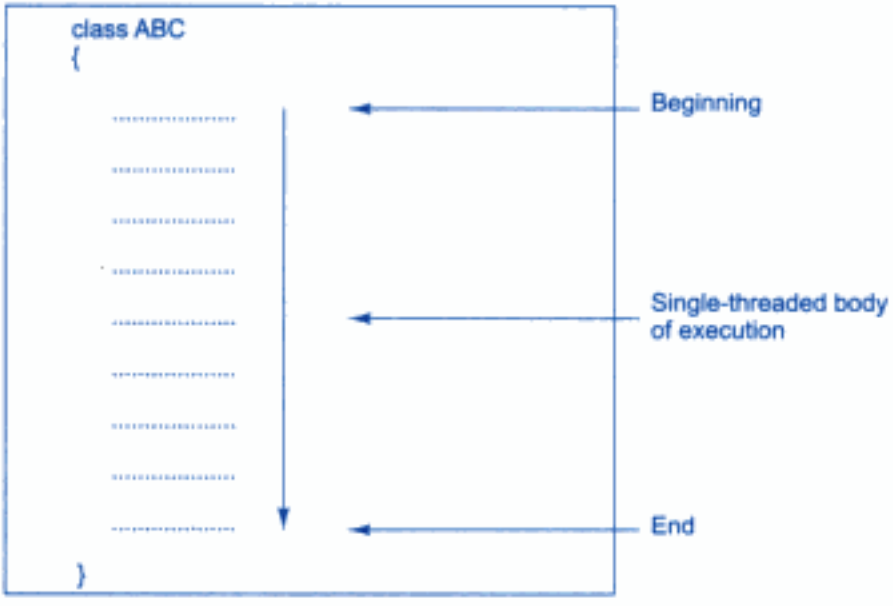

Fig. 12.1 *Single-threaded program*

Once initiated by the main thread, the threads A, B, and C run concurrently and share the resources jointly. It is like people living in joint families and sharing certain resources among all of them. The ability of a language to support multithreads is referred to as *concurrency*. Since threads in Java are subprograms of a main application program and share the same memory space, they are known as *lightweight threads* or *lightweight processes*.

It is important to remember that 'threads running in parallel' does not really mean that they actually run at the same time. Since all the threads are running on a single processor, the flow of execution is shared between the threads. The Java interpreter handles the switching of control between the threads in such a way that it appears they are running concurrently.

Multithreading is a powerful programming tool that makes Java distinctly different from its fellow programming languages. Multithreading is useful in a number of ways. It enables programmers to do multiple things at one time. They can divide a long program (containing operations that are conceptually concurrent) into threads and execute them in parallel. For example, we can send tasks such as printing into the background and continue to perform some other task in the foreground. This approach would considerably improve the speed of our programs.

Threads are extensively used in Java-enabled browsers such as HotJava. These browsers can download a file to the local computer, display a Web page in the window, output another Web page to a printer and so on.

Any application we are working on that requires two or more things to be done at the same time is probably a best one for use of threads.

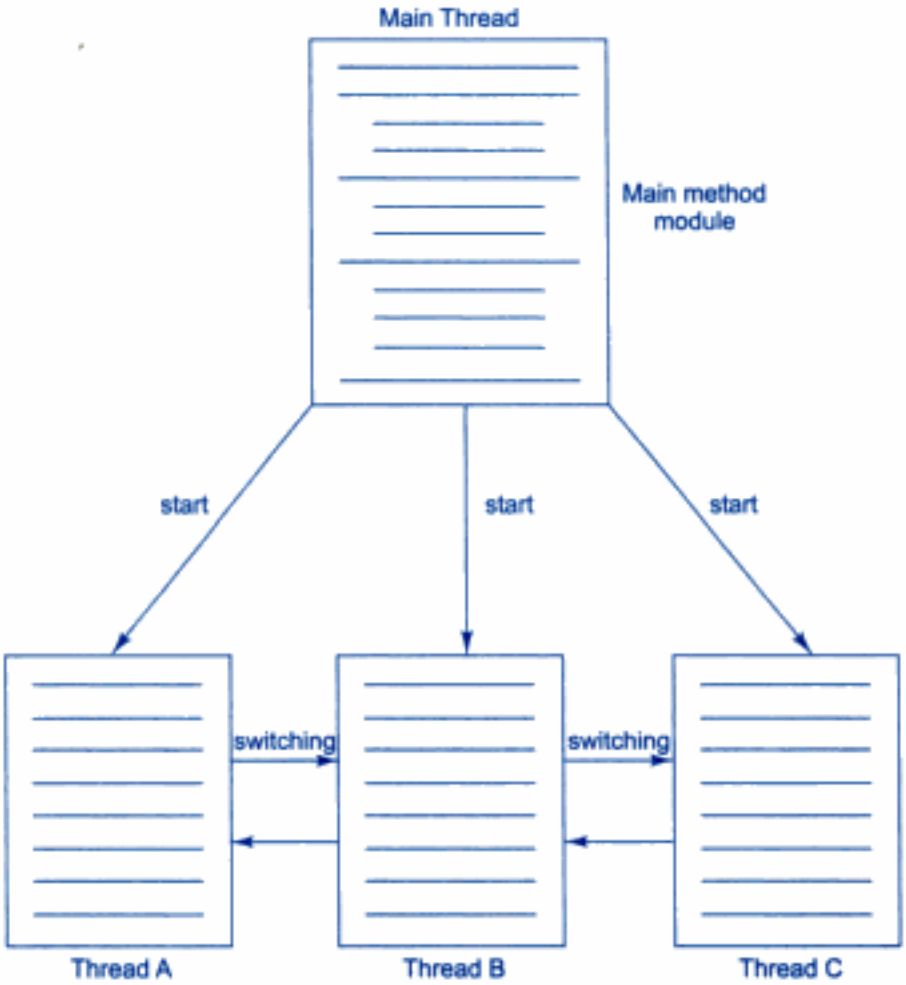

Fig. 12.2 *A Multithreaded program*

12.2 Creating Threads

Creating threads in Java is simple. Threads are implemented in the form of objects that contain a method called **run( )**. The **run( )** method is the heart and soul of any thread. It makes up the entire body of a thread and is the only method in which the thread's behaviour can be implemented. A typical **run( )** would appear as follows:

```
public void run( )
{
     .....
     ..... (statements for implementing thread)
     .....
}
```

The **run( )** method should be invoked by an object of the concerned thread. This can be achieved by creating the thread and initiating it with the help of another thread method called **start ( ).**

A new thread can be created in two ways.

1. ***By creating a thread class:*** Define a class that extends **Thread** class and override its **run( )** method with the code required by the thread.
2. ***By converting a class to a thread:*** Define a class that implements **Runnable** interface. The **Runnable** interface has only one method, **run( )**, that is to be defined in the method with the code to be executed by the thread.

The approach to be used depends on what the class we are creating requires. If it requires to extend another class, then we have no choice but to implement the **Runnable** interface, since Java classes cannot have two superclasses.

12.3 Extending the Thread Class

We can make our class runnable as thread by extending the class **java.lang.Thread.** This gives us access to all the thread methods directly. It includes the following steps:

1. Declare the class as extending the **Thread** class.
2. Implement the **run( )** method that is responsible for executing the sequence of code that the thread will execute.
3. Create a thread object and call the **start( )** method to initiate the thread execution.

Declaring the Class

The **Thread** class can be extended as follows:

```
class MyThread extends Thread
{
    ..........
    ..........
    ..........
}
```

Now we have a new type of thread **MyThread.**

Implementing the *run()* Method

The **run( )** method has been inherited by the class **MyThread.** We have to override this method in order to implement the code to be executed by our thread. The basic implementation of **run( )** will look like this:

```
public void run( )
{
    ..........
    ..........   // Thread code here
    ..........
}
```

When we start the new thread, Java calls the thread's **run( )** method, so it is the **run( )** where all the action takes place.

Starting New Thread

To actually create and run an instance of our thread class, we must write the following:

```
MyThread aThread = new MyThread( );
aThread.start( );                    // invokes run() method
```

The first line instantiates a new object of class **MyThread.** Note that this statement just creates the object. The thread that will run this object is not yet running. The thread is in a *newborn* state.

The second line calls the **start( )** method causing the thread to move into the *runnable* state. Then, the Java runtime will schedule the thread to run by invoking its **run( )** method. Now, the thread is said to be in the *running* state.

An Example of Using the Thread Class

Program 12.1 illustrates the use of **Thread** class for creating and running threads in an application. The program creates three threads A, B, and C for undertaking three different tasks. The **main** method in the **ThreadTest** class also constitutes another thread which we may call the "main thread".

The main thread dies at the end of its **main** method. However, before it dies, it creates and starts all the three threads A, B, and C. Note the statements like

```
new A( ).start( );
```

in the main thread. This is just a compact way of starting a thread. This is equivalent to:

```
A threadA = new A( );
threadA.start( );
```

Immediately after the thread A is started, there will be two threads running in the program: the main thread and the thread A. The **start( )** method returns back to the main thread immediately after invoking the **run( )** method, thus allowing the main thread to start the thread B.

Program 12.1 ***Creating threads using the thread class***

```
class A extends Thread
{
     public void run( )
     {
          for (int i=1; i<=5; i++)
          {
                    System.out.println("\tFrom ThreadA : i = " + i);
          }
          System.out.println("Exit form A ");
     }
}
class B extends Thread
{
```

(Continued)

Program 12.1 *(Continued)*

```
        public void run( )
        {
            for(int j=1; j<=5; j++)
            {
                    System.out.println("\tFrom Thread B :j = " + j);
            }
            System.out.println("Exit from B ");
        }
}
class B extends Thread
{
        public void run( )
        {
            for(int k=1; k<=5; k++)
            {
                System.out.println("\tFrom Thread C : k = " + k);
            }
            System.out.println("Exit from C ");
        }
}
class ThreadTest
{
        public static void main(String args[ ])
        {
            new A( ).start( );
            new B( ).start( );
            new C( ).start( );
        }
}
```

Output of Program 12.1 would be:

```
First run
        From    Thread      A     :      i     =      1
        From    Thread      A     :      i     =      2
        From    Thread      B     :      j     =      1
        From    Thread      B     :      j     =      2
        From    Thread      C     :      k     =      1
        From    Thread      C     :      k     =      2
        From    Thread      A     :      i     =      3
        From    Thread      A     :      i     =      4
        From    Thread      B     :      j     =      3
        From    Thread      B     :      j     =      4
        From    Thread      C     :      k     =      3
        From    Thread      C     :      k     =      4
```

```
  From    Thread      A     :     i     =     5
Exit from A
  From    Thread      B     :     j     =     5
Exit from B
  From    Thread      C     :     k     =     5
Exit from C
```

Second run

```
  From    Thread      A     :     i     =     1
  From    Thread      A     :     i     =     2
  From    Thread      C     :     k     =     1
  From    Thread      C     :     k     =     2
  From    Thread      A     :     i     =     3
  From    Thread      A     :     i     =     4
  From    Thread      B     :     j     =     1
  From    Thread      B     :     j     =     2
  From    Thread      C     :     k     =     3
  From    Thread      C     :     k     =     4
  From    Thread      A     :     i     =     5
Exit from A
  From    Thread      B     :     k     =     4
  From    Thread      B     :     j     =     5
  From    Thread      C     :     k     =     5
Exit from C
  From    Thread      B     :     j     =     5
Exit from B
```

Similarly, it starts C thread. By the time the main thread has reached the end of its **main** method, there are a total of four separate threads running in parallel.

We have simply initiated three new threads and started them. We did not hold on to them any further. They are running concurrently on their own. Note that the output from the threads are not specially sequential. They do not follow any specific order. They are running independently of one another and each executes whenever it has a chance. Remember, once the threads are started, we cannot decide with certainty the order in which they may execute statements. Note that a second run has a different output sequence.

12.4 Stopping and Blocking a Thread

Stopping a Thread

Whenever we want to stop a thread from running further, we may do so by calling its **stop( )** method, like:

```
aThread.stop( );
```

This statement causes the thread to move to the ***dead*** state. A thread will also move to the dead state automatically when it reaches the end of its method. The **stop( )** method may be used when the *premature death* of a thread is desired.

Blocking a Thread

A thread can also be temporarily suspended or blocked from entering into the runnable and subsequently running state by using either of the following thread methods:

```
sleep( )          // blocked for a specified time
suspend( )        // blocked until further orders
wait( )           // blocked until certain condition occurs
```

These methods cause the thread to go into the ***blocked*** (or ***not-runnable***) state. The thread will return to the runnable state when the specified time is elapsed in the case of **sleep( )**, the **resume( )** method is invoked in the case of **suspend( )**, and the **notify( )** method is called in the case of **wait( ).**

12.5 Life Cycle of a Thread

During the life time of a thread, there are many states it can enter. They include:

1. Newborn state
2. Runnable state
3. Running state
4. Blocked state
5. Dead state

A thread is always in one of these five states. It can move from one state to another via a variety of ways as shown in Fig. 12.3.

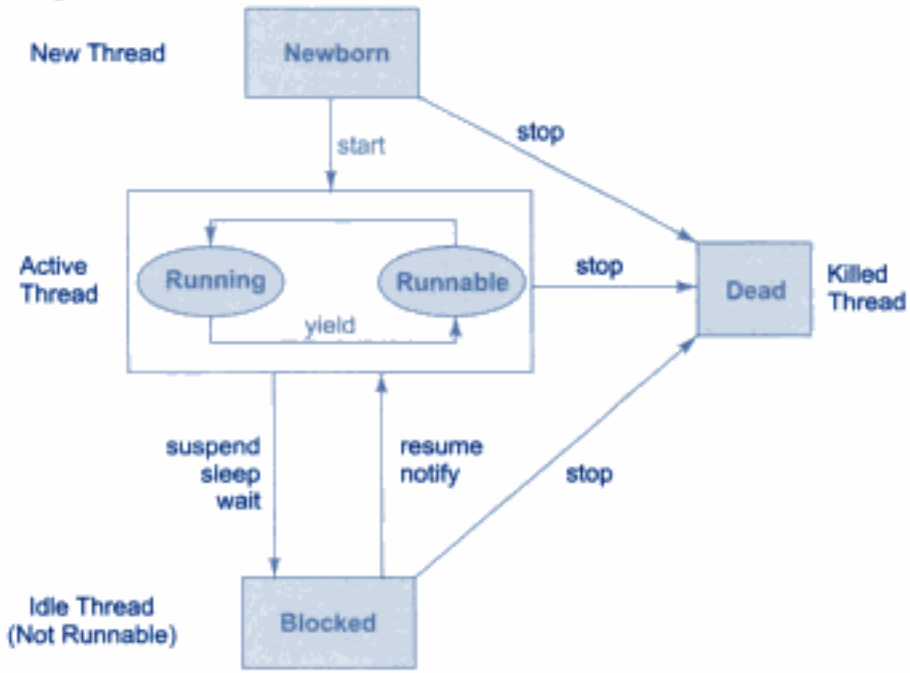

Fig. 12.3 *State transition diagram of a thread*

Newborn State

When we create a thread object, the thread is born and is said to be in *newborn* state. The thread is not yet scheduled for running. At this state, we can do only one of the following things with it:

- Schedule it for running using **start( )** method.
- Kill it using **stop( )** method.

If scheduled, it moves to the runnable state (Fig. 12.4). If we attempt to use any other method at this stage, an exception will be thrown.

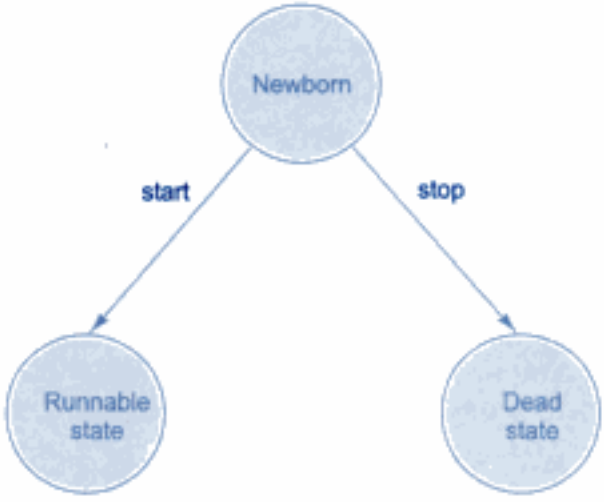

Fig. 12.4 *Scheduling a newborn thread*

Runnable State

The *runnable* state means that the thread is ready for execution and is waiting for the availability of the processor. That is, the thread has joined the queue of threads that are waiting for execution. If all threads have equal priority, then they are given time slots for execution in round robin fashion, i.e., first-come, first-serve manner. The thread that relinquishes control joins the queue at the end and again waits for its turn. This process of assigning time to threads is known as *time-slicing*.

However, if we want a thread to relinquish control to another thread to equal priority before its turn comes, we can do so by using the **yield( )** method (Fig. 12.5).

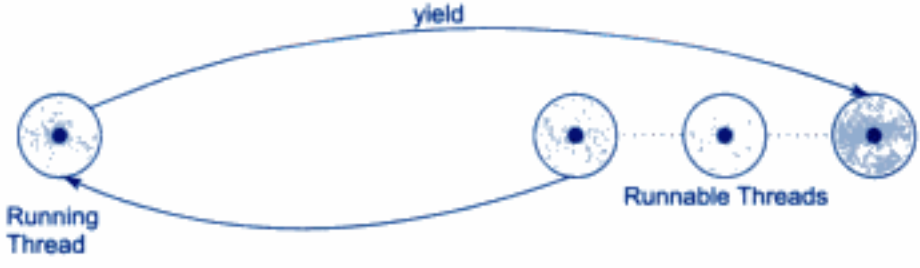

Fig. 12.5 *Relinquishing control using yield( ) method*

Running State

Running means that the processor has given its time to the thread for its execution. The thread runs until it relinquishes control on its own or it is preempted by a higher priority thread. A running thread may relinquish its control in one of the following situations.

1. It has been suspended using **suspend( )** method. A suspended thread can be revived by using the **resume( )** method. This approach is useful when we want to suspend a thread for some time due to certain reason, but do not want to kill it.

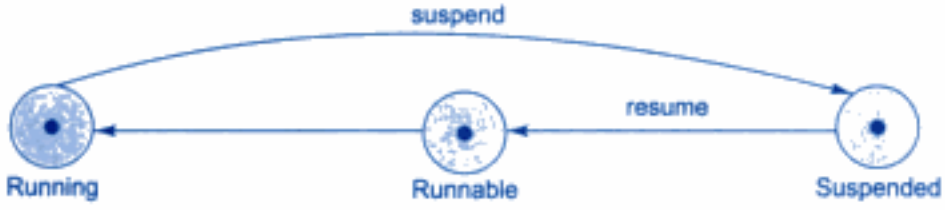

Fig. 12.6 *Relinquishing control using suspend( ) method*

2. It has been made to sleep. We can put a thread to sleep for a specified time period using the method **sleep**(*time*) where *time* is in milliseconds. This means that the thread is out of the queue during this time period. The thread re-enters the runnable state as soon as this time period is elapsed.

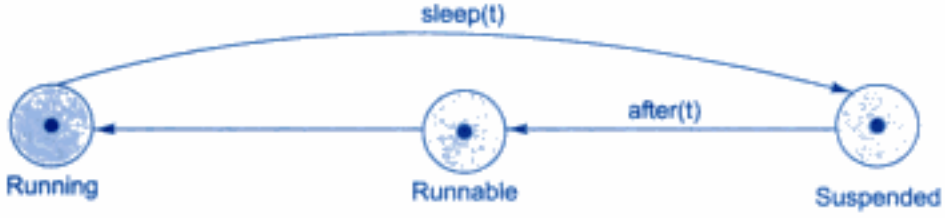

Fig. 12.7 *Relinquishing control using sleep( ) method*

3. It has been told to wait until some event occurs. This is done using the **wait( )** method. The thread can be scheduled to run again using the **notify( )** method.

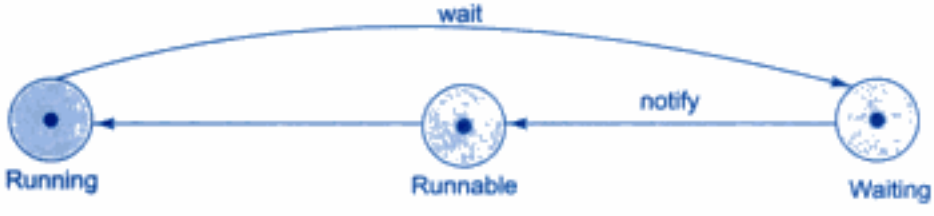

Fig. 12.8 *Relinquishing control using wait( ) method*

Blocked State

A thread is said to be *blocked* when it is prevented from entering into the runnable state and subsequently the running state. This happens when the thread is suspended, sleeping, or waiting in order to satisfy certain requirements. A blocked thread is considered "not runnable" but not dead and therefore fully qualified to run again.

Dead State

Every thread has a life cycle. A running thread ends its life when it has completed executing its **run( )** method. It is a natural death. However, we can kill it by sending the stop message to it at any state thus causing a premature death to it. A thread can be killed as soon it is born, or while it is running, or even when it is in "not runnable" (blocked) condition.

12.6 Using Thread Methods

We have discussed how **Thread** class methods can be used to control the behaviour of a thread. We have used the methods **start( )** and **run( )** in Program 12.1. There are also methods that can move a thread from one state to another. Program 12.2 illustrates the use of **yield( )**, **sleep( )** and **stop( )** methods. Compare the outputs of Programs 12.1 and 12.2.

Program 12.2 ***Use of yield( ), stop( ), and sleep( ) methods***

```
class A extends Thread
{
    public void run( )
    {
        for(int i = 1; i<=5; i++)
        {
            if(i==1) yield( );
            System.out.println("\tFrom Thread A : i = " +i);
        }
      System.out.println("exit from A ");
    }
}
class B extends Thread
{
    public void run( )
        {
            for(int i=1; i<=5; j++)
            {
                System.out.println("\tFrom Thread B : j = " + j);
                if(j==3) stop( );
        }
```

(Continued)

Program 12.2 *(Continued)*

```
            System.out.println("Exit from B ");
        }
    }
    class C extends Thread
    {
        public void run( )
        {
            for (int k=1; k<=5; k++)
            {
                System.out.println("\tFrom Thread C : k = " +k);
                if(k==1)
                try
                    {
                        sleep(1000);
                    }
                catch (Exception e)
                {
                }
            }
            System.out.println("Exit from C ");
        }
    }
    class ThreadMethods
    {
        public static void main(String args[ ])
        {
            A threadA = new A( );
            B threadB = new B( );
            C threadC = new C( );

            System.out.println("Start thread A");
            threadA.start( );

            System.out.println("Start thread B");
            threadB.start( );

            System.out.println("Start thread C");
            threadC.start( );

            System.out.println("End of main thread");
        }
    }
```

Here is the output of Program 12.2:

```
Start thread A
Start thread B
Start thread C
    From Thread B : j = 1
    From Thread B : j = 2
    From Thread A : i = 1
    From Thread A : i = 2
End of main thread
    From Thread C : k = 1
    From Thread B : j = 3
    From Thread A : i = 3
    From Thread A : i = 4
    From Thread A : i = 5
Exit from A
    From Thread C : k = 2
    From Thread C : k = 3
    From Thread C : k = 4
    From Thread C : k = 5
Exit from C
```

Program 12.2 uses the **yield( )** method in thread A at the iteration i = 1. Therefore, the thread A, although started first, has relinquished its control to the thread B. The **stop( )** method in thread B has killed it after implementing the **for** loop only three times. Note that it has not reached end of **run( )** method. The thread C started sleeping after executing the for loop only once. When it woke up (after 1000 milliseconds), the other two threads have already completed their runs and therefore was running alone. The main thread died much earlier than the other three threads.

12.7 Thread Exceptions

Note that the call to **sleep( )** method is enclosed in a **try** block and followed by a **catch** block. This is necessary because the **sleep( )** method throws an exception, which should be caught. If we fail to catch the exception, program will not compile.

Java run system will throw **IllegalThreadStateException** whenever we attempt to invoke a method that a thread cannot handle in the given state. For example, a sleeping thread cannot deal with the **resume( )** method because a sleeping thread cannot receive any instructions. The same is true with the **suspend( )** method when it is used on a blocked (Not Runnable) thread.

Whenever we call a thread method that is likely to throw an exception, we have to supply an appropriate exception handler to catch it. The **catch** statement may take one of the following forms:

```
catch (ThreadDeath e)
{
    ..........
    ..........          // Killed thread
}
```

```
catch (InterruptedException e)
{
     ..........
     ..........          // Cannot handle it in the current state
}
catch (IllegalArgumentException e)
{
     ..........
     ..........          // Illegal method argument
}
catch (Exception e)
{
     ..........
     ..........          // Any other
}
```

Exception handling is discussed in detail in Chapter 13.

12.8 Thread Priority

In Java, each thread is assigned a priority, which affects the order in which it is scheduled for running. The threads that we have discussed so far are of the same priority. The threads of the same priority are given equal treatment by the Java scheduler and, therefore, they share the processor on a first-come, first-serve basis.

Java permits us to set the priority of a thread using the **setPriority( )** method as follows:

```
ThreadName.setPriority(intNumber);
```

The **intNumber** is an integer value to which the thread's priority is set. The **Thread** class defines several priority constants:

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

The **intNumber** may assume one of these constants or any value between 1 and 10. Note that the default setting is NORM_PRIORITY.

Most user-level processes should use NORM_PRIORITY, plus or minus 1. Back-ground tasks such as network I/O and screen repainting should use a value very near to the lower limit. We should be very cautious when trying to use very high priority values. This may defeat the very purpose of using multithreads.

By assigning priorities to threads, we can ensure that they are given the attention (or lack of it) they deserve. For example, we may need to answer an input as quickly as possible. Whenever multiple threads are ready for execution, the Java system chooses the highest priority thread and executes it. For a thread of lower priority to gain control, one of the following things should happen:

1. It stops running at the end of **run( )**.
2. It is made to sleep using **sleep( )**.
3. It is told to wait using **wait( )**.

However, if another thread of a higher priority comes along, the currently running thread will be *preempted* by the incoming thread thus forcing the current thread to move to the runnable state. Remember that the highest priority thread always preempts any lower priority threads.

Program 12.3 and its output illustrate the effect of assigning higher priority to a thread. Note that although the thread A started first, the higher priority thread B has preempted it and started printing the output first. Immediately, the thread C that has been assigned the highest priority takes control over the other two threads. The thread A is the last to complete.

Program 12.3 ***Use of priority in threads***

```
class A extends Thread
{
    public void run( )
    {
        System.out.println("threadA started");
        for(int i=1; i<=4; i++)
        {
            System.out.println("\tFrom Thread A : i = " +i);
        }
        System.out.println("Exit from A ");
    }
}
class B extends Thread
{
    public void run( )
    {
        System.out.println("threadB started");
        for(int j=1; j<=4; j++)
        {
            System.out.println("\tFrom Thread B : j = " + j);
        }
        System.out.println("Exit from B ");
    }
}
class C extends Thread
{
    public void run( )
    {
        System.out.println("threadC started");
        for(int k=1; k<=4; k++)
        {
            System.out.println("\tFrom Thread C : k = " + k);
    }
```

(Continued)

Program 12.3 *(Continued)*

```
        System.out.println("Exit from C ");
        }
}
class ThreadPriority
{
    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.setPrioirty(Thread.MIN_PRIORITY);

        System.out.println("Start thread A");
        threadA.start( );

        System.out.println("Start thread B");
        threadB.start( );

        System.out.println("Start thread C");
        threadC.start( );

        System.out.println("End of main thread");
    }
}
```

Output of Program 12.3:

```
Start thread A
Start thread B
Start thread C
threadB started
    From Thread B : j = 1
    From Thread B : j = 2
threadC started
    From Thread C : k = 1
    From Thread C : k = 2
    From Thread C : k = 3
    From Thread C : k = 4
Exit from C
End of main thread
    From Thread B : j = 3
    From Thread B : j = 4
```

```
Exit from B
threadA started
     From Thread A : i = 1
     From Thread A : i = 2
     From Thread A : i = 3
     From Thread A : i = 4
Exit from A
```

12.9 Synchronization

So far, we have seen threads that use their own data and methods provided inside their **run( )** methods. What happens when they try to use data and methods outside themselves? On such occasions, they may compete for the same resources and may lead to serious problems. For example, one thread may try to read a record from a file while another is still writing to the same file. Depending on the situation, we may get strange results. Java enables us to overcome this problem using a technique known as *synchronization*.

In case of Java, the keyword **synchronised** helps to solve such problems by keeping a watch on such locations. For example, the method that will read information from a file and the method that will update the same file may be declared as **synchronized**. Example:

```
synchronized void update( )
{
     ..........
     ..........   // code here is synchronized
     ..........
}
```

When we declare a method synchronized, Java creates a "monitor" and hands it over to the thread that calls the method first time. As long as the thread holds the monitor, no other thread can enter the synchronized section of code. A monitor is like a key and the thread that holds the key can only open the lock.

It is also possible to mark a block of code as synchronized as shown below:

```
synchronized (lock-object)
{
     ..........   // code here is synchronized
     ..........
}
```

Whenever a thread has completed its work of using synchronized method (or block of code), it will hand over the monitor to the next thread that is ready to use the same resource.

An interesting situation may occur when two or more threads are waiting to gain control of a resource. Due to some reasons, the condition on which the waiting threads rely on to gain control does not happen. This results in what is known as *deadlock*. For example, assume that the thread A must access Method1 before it can release Method2, but the thread B cannot release Method1 until it gets hold of Method2. Because these are mutually exclusive conditions, a deadlock occurs. The code below illustrates this:

Thread A

```
synchronized method2( )
{
    synchronized method1( )
    {
        ..........
        ..........
    }
}
```

Thread B

```
synchronized method1( )
{
    synchronized method2( )
    {
        ..........
        ..........
    }
}
```

12.10 Implementing the 'Runnable' Interface

We stated earlier that we can create threads in two ways: one by using the extended **Thread** class and another by implementing the **Runnable** interface. We have already discussed in detail how the **Thread** class is used for creating and running threads. In this section, we shall see how to make use of the **Runnable** interface to implement threads.

The **Runnable** interface declares the **run( )** method that is required for implementing threads in our programs. To do this, we must perform the steps listed below:

1. Declare the class as implementing the **Runnable** interface.
2. Implement the **run( )** method.
3. Create a thread by defining an object that is instantiated from this "runnable" class as the target of the thread.
4. Call the thread's **start( )** method to run the thread.

Program 12.4 illustrates the implementation of the above steps. In main method, we first create an instance of **X** and then pass this instance as the initial value of the object **threadX** (an object of **Thread** Class). Whenever, the new thread **threadX** starts up, its **run( )** method calls the **run( )** method of the target object supplied to it. Here, the target object is **runnable**. If the direct reference to the thread **threadX** is not required, then we may use a shortcut as shown below:

```
new Thread (new X( )).start( );
```

Program 12.4 ***Using Runnable interface***

```
class X implements Runnable                                    // Step 1
{
    public void run( )                                         // Step 2
    {
        for(int i = 1; i<=10; i++)
        {
            System.out.println("\tThreadX : " +i);
        }
        System.out.println("End of ThreadX");
    }
}
class RunnableTest
{
    public static void main(String args[ ])
    {
        X runnable = new X( );
        Thread threadX = new Thread(runnable);                 // Step 3
        threadX.start( );                                      // Step 4
        System.out.println("End of main Thread");
    }
}
```

Output of Program 12.4

```
End of main Thread
    ThreadX : 1
    ThreadX : 2
    ThreadX : 3
    ThreadX : 4
    ThreadX : 5
    ThreadX : 6
    ThreadX : 7
    ThreadX : 8
    ThreadX : 9
    ThreadX : 10
End of ThreadX
```

12.11 Summary

A thread is a single line of execution within a program. Multiple threads can run concurrently in a single program. A thread is created either by subclassing the **Thread** class or implementing the **Runnable** interface. We have discussed both the approaches in detail in this chapter. We have also learned the following in this chapter:

- How to synchronize threads,
- How to set priorities for threads, and
- How to control the execution of threads

Careful application of multithreading will considerably improve the execution speed of Java programs.

Key Terms

Thread, Multithread, Multitask, Concurrency, Lightweight Processes, Time-slicing, Priority, Synchronization, Deadlock.

Review Questions

12.1 What is a thread?
12.2 What is the difference between multiprocessing and multithreading? What is to be done to implement these in a program?
12.3 What Java interface must be implemented by all threads?
12.4 How do we start a thread?
12.5 What are the two methods by which we may stop threads?
12.6 What is the difference between suspending and stopping a thread?
12.7 How do we set priorities for threads?
12.8 Describe the complete life cycle of a thread.
12.9 What is synchronization? When do we use it?
12.10 Develop a simple real-life application program to illustrate the use of multithreads.

Debugging Exercises

12.1 Sleep( ) method has been demonstrated in the following code. Will this code compile successfully?

```
class A extends Thread
{
    public void run()
    {
        for(int i=1; i<=5; i++)
        {
            System.out.println("\tFrom threadA. i = " +i);
            Thread.sleep(100);
        }
    }
}
class ThreadClass
{
    public static void main(String[] args)
    {
        A a = new A();
        a.start();
    }
}
```

12.2 Debug the given code which creates two different classes, one extending 'Thread' class and other, implementing 'Runnable' interface.

```
class multi1 extends Thread
{
    public void run()
        {
            for(int i=1; i<=5; i++)
            {
                System.out.println("\tFrom Thread 1 i = " +i);
            }
    }
};
class multi2 implements Runnable
{
    public void run()
        {
            for(int j=1; j<=5; j++)
            {
                System.out.println("\tFrom Thread 1 j = " +j);
        }
    }
};
class threadcheck
{
    public static void main(String[] args)
    {
        multi1 m1 = new multi1();
        m1.start();
        multi2 m2 = new multi2();
        m2.start();
    }
}
```

12.3 Will the code compile? If not, why?

```
class multi1 extends Thread
{
    public void start()
    {
        for(int i=1; i<=5; i++)
        {
            System.out.println("\tFrom Thread 1 i = " +i);
        }
    }
};
class multi2 implements Runnable
{
```

```
    public void start()
    {
        for(int j=1; j<=5; j++)
        {
            System.out.println("\tFrom Thread 1 j = " +j);
        }
    }
};
class runthread
{
    public static void main(String[] args)
    {
        multi1 m1 = new multi1();
        m1.start();
    }
}
```

12.4 The code given below calls the run( ) method of two threads while setting their priority. Will this code compile successfully? If not, correct the code.

```
class t1 extends Thread
{
    public void run()
    {
        System.out.println("This is Thread1 class");
    }
}
class t2 extends Thread
{
    public void run()
    {
        System.out.println("This is Thread2 Class");
    }
}
public class ThreadP
{
    public static void main(String s[])
    {
        t1 t = new t1();
        t2 tt = new t2();
        t.setPriority(Thread.MIN_PRIORITY);
        tt.setPriority(Thread.MIN_PRIORITY);
        t1.run();
        t2.run();
    }
}
```

12.5 Priority of a thread is defined in the given code. Debug the code.

```
class thread1 extends Thread
{
    public void run()
    {
        System.out.println("This is Thread1 class");
    }
}
class thread2 extends Thread
{
    public void run()
    {
        System.out.println("This is Thread2 Class");
    }
}
public class ThreadPrior
{
    public static void main(String s[])
    {
        thread1 t1 = new thread1();
        thread2 t2 = new thread2();
        t1.setPriority(Thread.Max_Priority);
        t1.run();
        t2.run();
    }
}
```