Chapter 15

THREADS

In this chapter, you will learn about multithreaded programming in Java. A single task can be divided into a number of small sub-tasks. Each sub-task can be executed as an independent process. Such a process is known as a thread. In this chapter, creation of multiple threads and managing them are discussed.

15.1 Multitasking

Before starting the thread concept, let us begin with a known multitasking process. When more than one task is processed by a computer, it is called multitasking. This is being done to utilize the idle time of a CPU more effectively. Each task (heavy-weight process) has its own set of variables and separate memory location for them. A multitasking process is shown in fig.15.1.

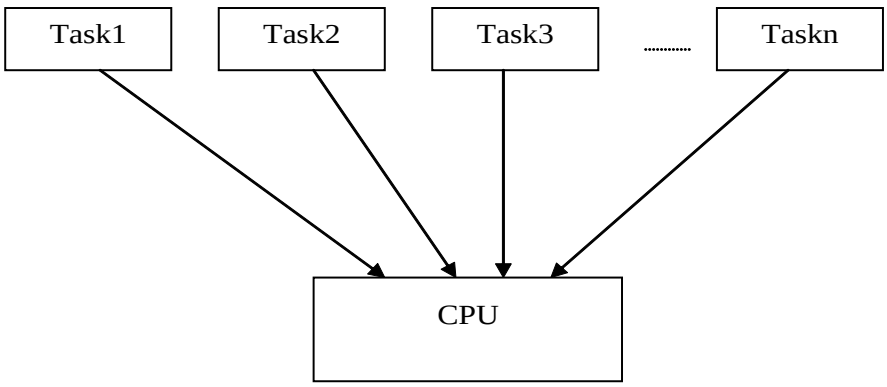

Fig.15.1 The Multitasking Process

The underlying operating system does the multitasking process by bringing tasks to the CPU and executing them by using appropriate schedule.

Multithreading

In many real-life situations, a single task (heavy-weight process) may comprise of many sub-tasks (light-weight process), such as reading an input stream, processing data, drawing graphics on screen, etc. Each sub-task is carried out one after another. But certain sub-tasks, like reading an input stream, may have to keep waiting till a byte is read. In such occasions, the CPU will be idling without doing any work.

This idling time of the CPU can be utilized if some other sub-task, say drawing a graphic on a screen, can be executed in the CPU. Java language provides a mechanism by which each sub-task can be treated as a separate process. Several such processes originating from a single task, can be simultaneously started and handled by Java. This mechanism of treating a single task as several independent processes simultaneously is called multithreading. Each independent process is called a thread. Each thread is executed one at a time in the CPU. Fig. 15.2 shows the multithreading process.

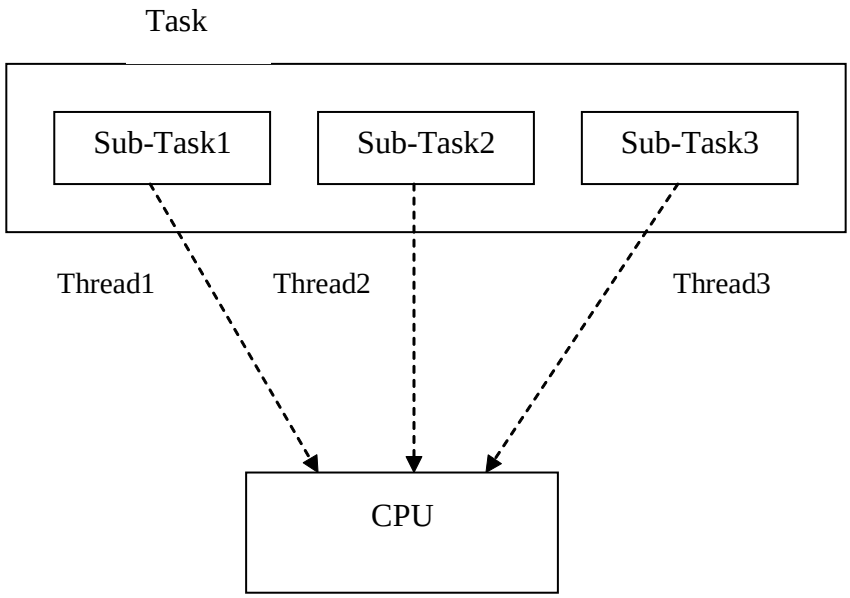

Fig.15.2 The Multithreaded Process

In multithreading, same set of variables and memory space is shared by the threads. When a thread dealing with a sub-task, say reading an input stream, has to wait for the arrival of a byte, such a thread will exit the CPU and another thread, say drawing a graphic on a screen, may enter the CPU and continue its task. Thus Java's thread provides a mechanism to utilize the CPU time more optimally. Multithreading finds use in variety of applications. The

threading mechanism is supported by Java's **Thread** class contained in java.lang package.

15.2. Creating a Thread

A Java thread can be created in two different ways. They are:

i) By extending **Thread** class
ii) By implementing **Runnable** interface

Though both methods can be used to create a thread, implementing **Runnable** interface has certain advantages, like subclassing an existing class. This method is recommended for creating a **Thread.**

Some of the methods defined in a **Thread** class to create and execute a thread are given in table 15.1.

Table 15.1 Some of the Methods Defined in a Thread Class

	Method	Purpose of the Method
1.	String getName()	Returns the name of the invoking Thread object
2.	boolean isAlive()	Returns true if the thread has started and has not yet terminated
3.	void join()	Waits for a thread to terminate
4.	void run()	Contains the statements that are to be executed in a thread
5.	void start()	Starts a thread by calling the run() method
6.	void sleep(long ms)	Makes the currently executing thread to sleep for ms milli seconds

Let us see how to create a thread from a **Thread** class. Create a subclass of the **Thread** class. Write a **run()** method and place all the statements that are to be executed by a thread. Write a constructor for the class and create an instance of the **Thread** class by passing the **run()** method as a parameter. This is done through the use of **this** keyword. Optionally, a name for the thread can also be assigned by placing a String as a second parameter for the **Thread** instance. Start the thread by calling the **start()** method of the **Thread** class. Program 15.1 is an illustration for creating a thread by this way. The program finds the sum of the natural numbers from 1 up to n, with n varying from 1 to 5.

Program 15.1

```
// Creation of a Thread  by extending the Thread class
// The thread is created and started in the constructor.
class Sumthread
    extends Thread
    {
    int i, sum = 0;
    Thread th;
    Sumthread()
       {
        th = new Thread(this, “My thread1”);
        System.out.println(“\n The thread created is : “ +
                                th.getName());
        th.start();
       }
    public void run()
        {
        for (i = 1; i <= 5; i++)
            {
            sum += i;
             System.out.println(“\n Sum of numbers from 1 up
                                to “ + i + “  = “ + sum);
            }
        }
    }
class Threadcreate1
    {
    public static void main(String args [])
        {
        new Sumthread();
        }
    }
```

The above program gives the following output:

```
The thread created is : My thread1
Sum of numbers from 1 up to  1  = 1
Sum of numbers from 1 up to  2  = 3
Sum of numbers from 1 up to  3  = 6
Sum of numbers from 1 up to  4  = 10
Sum of numbers from 1 up to  5  = 15
```

In the above program 15.1, the **run()** method is executed by its own class object. The **run()** method is not directly called. When the **start()** method is called, the **run()** method is automatically called. It is also possible that the **run()** method can be executed by some other **Thread** object. Program 15.2 shows how a **run()** method is executed by an object other than by its own object, where the **run()** method is defined. The problem dealt in this program 15.2 is the same as that of program 15.1.

Program 15.2

```
// Creation of a Thread  by extending the Thread class
// The thread is created and started in main method of
// another class.
class Sumthread
    extends Thread
    {
    int i, sum = 0;
    public void run()
        {
        for (i = 1; i <= 5; i++)
            {
            sum += i;
             System.out.println("\n Sum of numbers from 1 up
                                to " + i + "  = " + sum);
            }
        }
    }
class Threadcreate2
    {
    public static void main(String args [])
       {
        Sumthread st = new Sumthread();
        Thread th = new Thread(st, "My thread2");
        System.out.println("\n The thread created is : "
                                + th.getName());
        th.start();
       }
    }
```

The above program gives the following output:

```
The thread created is : My thread2
Sum of numbers from 1 up to  1  = 1
Sum of numbers from 1 up to  2  = 3
Sum of numbers from 1 up to  3  = 6
Sum of numbers from 1 up to  4  = 10
Sum of numbers from 1 up to  5  = 15
```

Now, let us see the second way to create a thread by implementing the interface **Runnable.** Here also, the **run()** method is placed inside the class implementing **Runnable**. A constructor is used to create a **Thread** object and start it. One advantage of implementing a **Runnable** interface is that a thread can be created in a subclass of an existing class. This enables to write several threads for the same superclass. Program 15.3 shows the creation of a thread by implementing the interface **Runnable**. Another method, **isAlive(),** is also used to know the status of the thread.

Program 15.3

```
//  Creation of a Thread  by implementing the interface
//  Runnable
//  The thread is created and started by the constructor of
//  the same class.
class Sumthread
    implements Runnable
    {
    int i, sum = 0;
    Thread th;
    Sumthread()
      {
      th = new Thread(this, “My thread3”);
      System.out.println(“\n The thread created is : “ +
           th.getName());
      System.out.println(“\n Before starting the thread\n
           Is the thread alive? :” + th.isAlive());
      th.start();
      System.out.println(“\n After starting the thread\n
                Is the thread alive? :” + th.isAlive());
      }
    public void run()
        {
        for (i = 1; i <= 5; i++)
           {
           sum += i;
           System.out.println(“\n Sum of numbers from 1
                     up to “ + i + “  = “ + sum);
           }
        }
    }
class Threadcreate3
    {
    public static void main(String args [])
        {
        new Sumthread();
        }
    }
```

The above program gives the following output:

```
The thread created is : My thread3
Before starting the thread
Is the thread alive? :false
After starting the thread
Is the thread alive? :true
Sum of numbers from 1 up to  1  = 1
Sum of numbers from 1 up to  2  = 3
Sum of numbers from 1 up to  3  = 6
Sum of numbers from 1 up to  4  = 10
Sum of numbers from 1 up to  5  = 15
```

We write another program in which the **run()** method is executed by another **Thread** object. Program 15.4 shows the second form of executing the **run()** method using **Runnable** interface.

The codes that are to be executed in a thread are to be placed in the **run()** method.

Program 15.4

```
//  Creation of a Thread  by implementing the interface
//  Runnable. The thread is created in one class and
//  started in another class

class Sumthread
    implements Runnable
    {
    int i, sum = 0;
    public void run()
        {
        for (i = 1; i <= 5; i++)
           {
           sum += i;
           System.out.println("\n Sum of numbers from 1
                     up to  " + i + "  = " + sum);
           }
        }
    }
class Threadcreate4
    {
    public static void main(String args [])
      {
      Sumthread st = new Sumthread();
      Thread th = new Thread(st, "My thread4");
      System.out.println("\n The thread created is : " +
                          th.getName());
      System.out.println("\n Before starting the
                          thread\n Is the thread alive? :" +
                          th.isAlive());
      th.start();
      System.out.println("\n After starting the thread\n Is
                     the thread alive? :" + th.isAlive());
      }
  }
```

The above program 15.4 gives the same result as that of program 15.3.

15.3 States of a Thread

When a thread is created, it goes to different states before it completes its task and is dead. The different states are:

- new
- runnable
- running
- blocked
- dead

new threads

When a thread is created, it is in a new state. In this state the thread will not be executed.

runnable threads

When the **start()** method is called on the thread object, the thread is in runnable state. A runnable thread not necessarily be executed by the CPU. A runnable thread joins the collection of threads that are ready for execution. Which runnable thread takes up the CPU is determined by the underlying operating system.

running threads

A thread currently being executed by the CPU is in a running state.

blocked threads

A running thread may go to a blocked state due to any of the following conditions.

- **wait** method is called by the thread
- the thread performs I/O operation
- **sleep** method is called by the thread

Fig. 15.3 shows the different states of a thread.

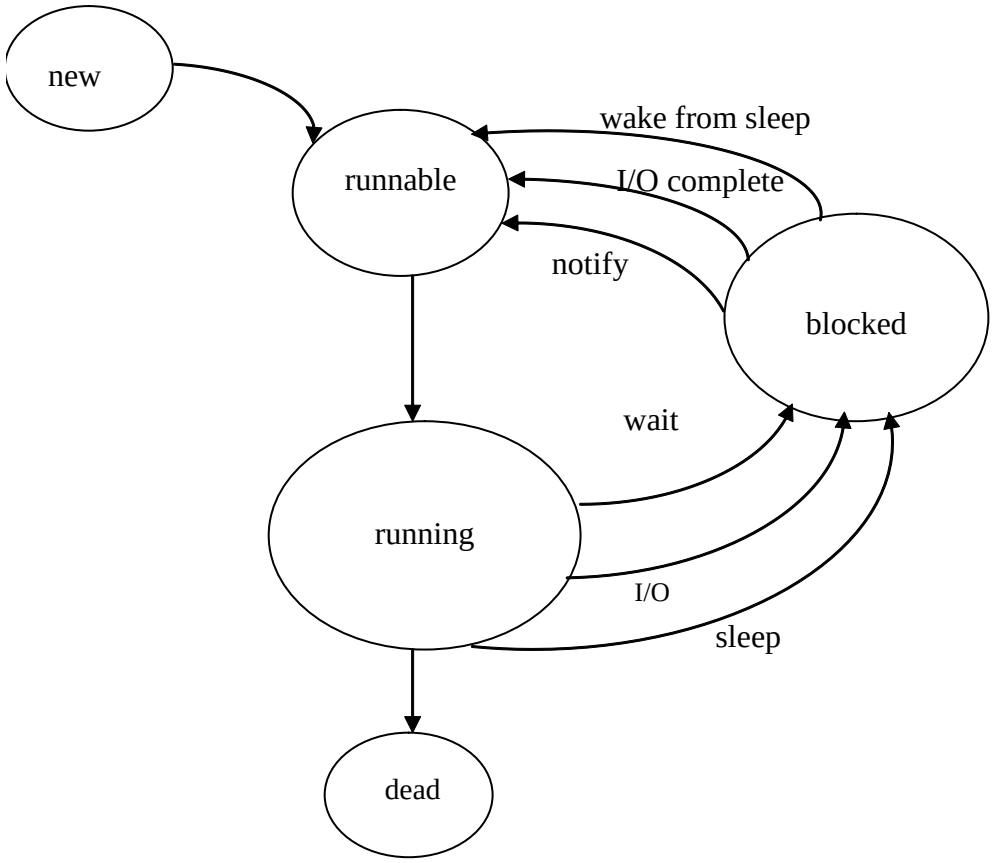

Fig.15.3 The Different States of a Thread

When a blocked thread is unblocked, it goes to runnable state and not to running state. Among the runnable threads, the one which has higher priority (set by the programmer) will go to the running state. This scheduling is done by the operating system.

When a blocked thread is unblocked, it goes to a runnable state and not to a running state.

dead thread

A thread becomes dead on two occasions. In the first case, a thread completes its task, exits the running state and becomes dead. In the other case, the **run** method is aborted, due to the occurrence of an exception and the thread becomes dead.

15.4 Multithreaded Programming

In earlier examples, we have seen programs with only one thread. Now, we will see how to write a multithreaded program. When a program is started, a thread is automatically created. It is called the main thread. The main thread

can be accessed using the **currentThread()** method of the **Thread** class. Program 15.5 illustrates the creation of two child threads, one finds the sum of natural numbers and the other finds the factorial of numbers.

Program 15.5

```
// This program creates two threads.
class Sumthread
    implements Runnable
    {
    int i, sum = 0;
    public void run()
        {
        for (i = 1; i <= 5; i++)
           {
           sum += i;
           System.out.println(" Sum of numbers from 1 up
                              to " + i + "  = " + sum);
           if (i == 4)
                Thread.yield();
           }
        }
    }
class Factthread
    implements Runnable
    {
    int i, n, fact = 1;
    public void run()
        {
        for (i = 1; i <= 5; i++)
           {
           fact *= i;
           System.out.println(" Factorial of  " + i + " = "
                                    + fact);
           }
        }
    }

class Twothread
    {
    public static void main(String args [])
      {
      Thread ct = Thread.currentThread();
      System.out.println("\n The main thread  is : "
                                    + ct.getName());
      Sumthread st = new Sumthread();
      Factthread ft = new Factthread();
      Thread sumt = new Thread(st, "Sum thread");
      Thread factt = new Thread(ft, "Factorial thread");
```

```
        sumt.start();
        System.out.println("\n The thread created is : "
                                        + sumt.getName());
        factt.start();
        System.out.println("\n The thread created is : "
                                        + factt.getName());
        }
    }
```

The above program gives the following output:

```
The main thread  is : main
The thread created is : Sum thread
The thread created is : Factorial thread
Sum of numbers from 1 up to  1  = 1
Sum of numbers from 1 up to  2  = 3
Sum of numbers from 1 up to  3  = 6
Sum of numbers from 1 up to  4  = 10
Sum of numbers from 1 up to  5  = 15
Factorial of  1 = 1
Factorial of  2 = 2
Factorial of  3 = 6
Factorial of  4 = 24
Factorial of  5 = 120
```

When you repeatedly run the above program 15.5, you may get the results interleaved or in a different order. It is due to the fact that when threads are created a default priority is allotted to each of the thread. Since both sum thread and factorial thread take equal priorities, the CPU time is allotted equally to each thread. Hence, results may be interleaved.

15.5 Thread Priorities

Threads can be assigned priorities. The priorities are used by the operating system to find out which runnable thread is to be given the CPU time. Priorities are given by priority numbers ranging from 1 to 10. A thread with higher priority number will be given preference over the threads with lower priority numbers. The priority of a thread can be set using the method:

setPriority(int pri-numb)

There are three pre-defined priorities. **MIN_PRIORITY** with a value 1, **NORM_PRIORITY** with a value 5 and **MAX_PRIORITY** with a value 10. When a thread is created, a default priority **NORM_PRIORITY** is given. To obtain the priority of a thread, the

int getPriority()

method can be used. When two threads have equal priorities, the underlying

operating system decides which thread is to be given the CPU. The scheduling of threads to running state depends on the OS. It is not guaranteed that threads of equal priorities are given equal CPU time. Program 15.6 shows the creation of two threads and assigning them priorities.

Program 15.6

```
// This program sets priorities for the threads created.
class Sumthread
    implements Runnable
    {
    int i, sum = 0;
    public void run()
        {
        for (i = 1; i <= 5; i++)
           {
           sum += i;
           System.out.println("\n Sum of numbers from 1
                          up to  " + i + "  = " + sum);
           }
        }
    }
class Factthread
    implements Runnable
    {
    int i, n, fact = 1;
    public void run()
        {
        for (i = 1; i <= 5; i++)
           {
           fact *= i;
           System.out.println("\n Factorial of  " + i + "
                                    = "  + fact);
           }
        }
    }

class Threadpriority
    {
    public static void main(String args [])
        {
         Sumthread st = new Sumthread();
         Factthread ft = new Factthread();
         Thread sumt = new Thread(st, "Sum thread");
         Thread factt = new Thread(ft, "Factorial thread");
         sumt.setPriority(Thread.NORM_PRIORITY - 2);
         factt.setPriority(Thread.NORM_PRIORITY + 2);
         System.out.println("\n The thread created is : " +
                               sumt.getName());
```

```
        System.out.println("\n The thread created is : " +
                                factt.getName());
        System.out.println("\n The priority value of  " +
                    sumt.getName() + " is  =  " +
                    sumt.getPriority());
        System.out.println("\n The priority value of  "
                    + factt.getName() + " is  =  " +
                    factt.getPriority());
        sumt.start();
        factt.start();
       }
}
```

The above program gives the following output:

```
The thread created is : Sum thread
The thread created is : Factorial thread
The priority value of  Sum thread is  =  3
The priority value of  Factorial thread is  =  7
Factorial of  1 = 1
Factorial of  2 = 2
Factorial of  3 = 6
Factorial of  4 = 24
Factorial of  5 = 120
Sum of numbers from 1 up to  1  = 1
Sum of numbers from 1 up to  2  = 3
Sum of numbers from 1 up to  3  = 6
Sum of numbers from 1 up to  4  = 10
Sum of numbers from 1 up to  5  = 15
```

In the above program 15.6, the Factthread is executed first because it has been assigned higher priority 7, than the Sumthread. You will get the same result in the same order if the program is executed any number of times. This is in contrast to the result for the program 15.5.

15.6 Waiting for a thread-join Method

In some problems, it may be required to wait for a particular thread to complete its task before another thread to proceed with. In such occasions, the **join** method of **Thread** class can be used. When **join** method is called on a thread object, the control waits for the thread to complete its task and becomes a dead thread. Then the control proceeds with the normal course of execution. In program 15.7, two threads are created. One thread is started and the **join** method is called on it. Until the thread completes its task, the control waits at that point. As soon as the corresponding thread comes to dead state, the control proceeds with the execution of the other statements.

Program 15.7

```
// This program sets priorities for the threads created.
//  An indirect wait is made till the other thread proceeds
//  using the join methods.

class Sumthread
    implements Runnable
    {
    int i, sum = 0;
    public void run()
        {
        for (i = 1; i <= 5; i++)
           {
           sum += i;
            System.out.println("\n Sum of numbers from 1
                up to " + i + "  = " + sum);
           }
        }
    }
class Factthread
    implements Runnable
    {
    int i, n, fact = 1;
    public void run()
        {
        for (i = 1; i <= 5; i++)
           {
           fact *= i;
           System.out.println("\n Factorial of  " + i
                     + " = " + fact);
            }
        }
    }
class Threadjoin
    {
    public static void main(String args [])
      {
      Sumthread st = new Sumthread();
      Factthread ft = new Factthread();
      Thread sumt = new Thread(st, "Sum thread");
      Thread factt = new Thread(ft, "Factorial thread");
      sumt.setPriority(Thread.NORM_PRIORITY - 2);
      factt.setPriority(Thread.NORM_PRIORITY + 2);
      System.out.println("\n The priority value of  "
                        + sumt.getName() + " is  =  "
                        + sumt.getPriority());
      System.out.println("\n The priority value of  "
                        + factt.getName() + " is  =  "
                        + factt.getPriority());
```

```
        sumt.start();
        System.out.println("\n The thread created is : "
                            + sumt.getName());
        try
              {
              sumt.join();
              }
        catch (InterruptedException e)
              {
              ;
              }
        factt.start();
        System.out.println("\n The thread created is : "
                                  + factt.getName());
        }
  }
```

The above program gives the following output:

```
The priority value of  Sum thread is  =  3
The priority value of  Factorial thread is  =  7
The thread created is : Sum thread
Sum of numbers from 1 up to  1  = 1
Sum of numbers from 1 up to  2  = 3
Sum of numbers from 1 up to  3  = 6
Sum of numbers from 1 up to  4  = 10
Sum of numbers from 1 up to  5  = 15
The thread created is : Factorial thread
Factorial of  1 = 1
Factorial of  2 = 2
Factorial of  3 = 6
Factorial of  4 = 24
Factorial of  5 = 120
```

You will observe from the result that when the Sum thread is started and called with **join** method, the control waits till the Sum thread completes its task. After that the Factorial thread is started.

15.7 Controlling the Threads

The objective of multithreading is to deal with sub-tasks independent of each other and to utilize the CPU time optimally. In a multithreaded program, if a particular thread is assigned a higher priority and if it needs longer time to complete its task, then the other runnable threads have to keep waiting. No benefit will be derived if one thread is allowed to consume much of CPU time and keep other threads waiting. Fair chances are to be given to every thread in a program. In any multithread program, a few of the threads in their course of action, such as doing I/O operation, may go from running state to blocked state,

thereby creating opportunities for other threads to enter running state. In the absence of any such natural event, the programmer must find ways to make a CPU-hungry thread to give way for other threads to enter the CPU. Java has two methods which can change the running state of a thread to blocked state. These methods are given in table 15.2.

Table 15.2. Methods that Control the Thread

Method	Purpose of the Method
1. static void sleep(long ms)	Makes the thread to sleep for ms milliseconds
2. static void yield()	Causes the running thread to yield to other runnable thread

The sleep method makes the thread to sleep (lying idle) for a specified amount of time. Once a thread calls a **sleep** method, it goes from running state to blocked state. It remains sleeping (blocked) till the specified amount of time, wakes up and goes to runnable state. This is one way of making a thread to give way for other runnable thread to enter the running state. Program 15.8 shows the use of **sleep** method. In this program, two child threads, Sum thread and Factorial thread are created in addition to the main thread. Each of the child thread is made to sleep 25 milliseconds after two or three calculations.

Program 15.8

```
// This program illustrates the use of sleep method.
class Sumthread
    implements Runnable
    {
    int i, sum = 0;
    public void run()
        {
        for (i = 1; i <= 5; i++)
            {
            sum += i;
            try
                {
                if (i % 2 == 0)
                    Thread.sleep(25);
                }
            catch (InterruptedException e)
                {
                ;
                }
            System.out.println("\n Sum of numbers from 1
                      up to " + i + "  = " + sum);
```

```
            }
        }
    }
class Factthread
    implements Runnable
    {
    int i, n, fact = 1;
    public void run()
        {
        for (i = 1; i <= 5; i++)
            {
            fact *= i;
            try
                {
                if (i % 3 == 0)
                    Thread.sleep(25);
                }
            catch (InterruptedException e)
                {
                ;
                }
            System.out.println("\n Factorial of  " + i + "
                                    = " + fact);
            }
        }
    }
class Threadsleep
    {
    public static void main(String args [])
      {
      Thread ct = Thread.currentThread();
      System.out.println("\n The main thread  is : " +
                              ct.getName());
      Sumthread st = new Sumthread();
      Factthread ft = new Factthread();
      Thread sumt = new Thread(st, "Sum thread");
      Thread factt = new Thread(ft, "Factorial thread");
      System.out.println("\n The main thread priority  is :
                " + ct.getPriority());
      System.out.println("\n The Sum thread priority  is :
                " + sumt.getPriority());
      System.out.println("\n The Factorial thread priority
                is : " + factt.getPriority());
      sumt.start();
      factt.start();
      }
 }
```

The above program gives the following output:

```
The main thread  is : main
The main thread priority  is : 5
The Sum thread priority  is : 5
The Factorial thread priority  is : 5
Sum of numbers from 1 up to  1  = 1
Factorial of  1 = 1
Factorial of  2 = 2
Sum of numbers from 1 up to  2  = 3
Sum of numbers from 1 up to  3  = 6
Factorial of  3 = 6
Factorial of  4 = 24
Factorial of  5 = 120
Sum of numbers from 1 up to  4  = 10
Sum of numbers from 1 up to  5  = 15
```

In the above program, three threads, main, Sum and Factorial threads, are created. All the three threads take up priority 5. The first four-line outputs are done by the main thread. Thereafterwards, only the two child threads, Sum and Factorial, with equal priority compete for the CPU. The Sum thread is made to sleep for 25 milliseconds whenever even numbers are encountered and the Factorial thread is made to sleep whenever the processing number is divisible by 3. Thus the Sum thread and Factorial thread alternatively provide opportunity to each other (with equal priority) to enter the running state.

When a program is executed, a thread is created in the main() method implicitly. This thread is called main thread. A reference can be obtained to this thread using **currentThread()** method.

The **yield** method makes the running thread to yield to other runnable threads. If there are runnable threads which have the same or higher priority of the yielding thread, one of them will be scheduled to run in the CPU. The yielding thread will be in runnable state until it gets its next turn. If there are no runnable threads with the same or higher priority, the yielding thread will again come back to running state and continue to run in the CPU. The program 15.9 illustrates the use of **yield** method. The problem dealt is the same as that for program 15.8.

Program 15.9

```
// This program illustrates the use of yield method.
class Sumthread
    implements Runnable
    {
    int i, sum = 0;
```

```
    public void run()
        {
        for (i = 1; i <= 5; i++)
            {
            sum += i;
            if (i % 2 == 0)
                Thread.yield();
             System.out.println(“\n Sum of numbers from 1
                                up to “ + i + “  = “ + sum);
            }
        }
    }
class Factthread
    implements Runnable
    {
    int i, n, fact = 1;
    public void run()
        {
        for (i = 1; i <= 5; i++)
            {
            fact *= i;
            if (i % 3 == 0)
                Thread.yield();
             System.out.println(“\n Factorial of  “ + i + “
                                = “ + fact);
            }
        }
    }
class Threadyield
    {
    public static void main(String args [])
      {
      Thread ct = Thread.currentThread();
      System.out.println(“\n The main thread  is : “
                                + ct.getName());
      Sumthread st = new Sumthread();
      Factthread ft = new Factthread();
      Thread sumt = new Thread(st, “Sum thread”);
      Thread factt = new Thread(ft, “Factorial thread”);
      System.out.println(“\n The main thread priority
                         is : “ + ct.getPriority());
      System.out.println(“\n The Sum thread priority
                         is : “ + sumt.getPriority());
      System.out.println(“\n The Factorial thread priority
                         is : “ + factt.getPriority());
      sumt.start();
      factt.start();
      }
  }
```

The above program gives the following output:

```
The main thread  is : main
The main thread priority  is : 5
The Sum thread priority  is : 5
The Factorial thread priority  is : 5
Sum of numbers from 1 up to  1  = 1
Factorial of  1 = 1
Factorial of  2 = 2
Sum of numbers from 1 up to  2  = 3
Sum of numbers from 1 up to  3  = 6
Factorial of  3 = 6
Factorial of  4 = 24
Factorial of  5 = 120
Sum of numbers from 1 up to  4  = 10
Sum of numbers from 1 up to  5  = 15
```

In the above program, three threads main, Sum and Factorial are created. All the three have the same priority 5. As in the previous program 15.8, the Sum thread is made to yield whenever the number processed is even and the Factorial thread is made to yield whenever the processed number is divisible by 3. The result obtained is the same as the previous program. It is to be noted that when the program is repeated many times, the sequence of the output may be different from the one shown above. It is due to the fact that scheduling the thread is carried out by the underlying OS. Hence, with different OS the interleaving sequence of output may differ.

In multithread program, if there are threads that have no chance of being blocked and have a higher priority, such threads should be sent to blocked state by calling sleep or yield method. Only then other threads will have a chance to enter CPU.

15.8 Synchronizing Methods

Classes can be defined with methods which can do generalized tasks such as sorting a given set of numbers, calculating the compound interest, etc. Objects of such classes can be created and be used as a common resource. Threads can make use of such objects for carrying out their tasks. When two or more threads use the same object to do their task, it is likely that the task of one thread is mixed with the task of other thread and gives incorrect results. This may happen due to any of the following situations:

- Threads using the common object may have same priority and, therefore, the CPU may switch from one thread to another thread before the method completes the task in the running thread.
- A thread using the common object may have a higher priority than another thread which is also using the common object and may preempt the lower priority thread, in which the method has not yet completed the task.
- A thread using the common object may go to a blocked state before the method completes the task and another thread using the same object may start executing its task.

Program 15.10 illustrates the use of a common object by three threads and the resulting interleaved output for the threads. This program has a class Printing with a method printnumber that prints numbers from the given number down to 1, by decrementing one in each step. Halfway through, the process is made to sleep for 100 milliseconds (to simulate a blocked state) and then to complete the remaining task. Another class Threadserve is used to create a thread. This class has a constructor, through which an object of type Printing and an int type number is passed as parameter. The **run()** method makes use of the printnumber method of the object. In the main method, an object of the type Printing is created. Three instances of the Threadserve are created to which the same object of type Printing is passed as parameter with int number 16 for the first, 8 for the second and 10 for the third. These three objects create threads and are supposed to print from 16 to 1 by the first, 8 to 1 by the second and 10 to 1 by the third.

Program 15.10

```
// This program illustrates the use of unsynchronized method.
class Printing
    {
    void printnumber(int n)
        {
         System.out.println(" Start");
        for (int j = n; j > 0; j–)
            {
            try
                {
                if (j == n / 2)
                    Thread.sleep(100);
                }
            catch (InterruptedException e)
                {
                ;
                }
```

```
            System.out.print("  " + j);
            }
        System.out.println(" End");
        }
    }
class Threadserve implements Runnable
    {
    int n;
    Printing pt;
    Thread th;
    Threadserve(Printing p, int x)
        {
        n = x;
        pt = p;
        th = new Thread(this);
        th.start();
        }
    public void run()
        {
        pt.printnumber(n);        }
    }
class Threadsynchro
    {
    public static void main(String args [])
        {
         Printing p = new Printing();
         Threadserve ts1 = new Threadserve(p, 16);
         Threadserve ts2 = new Threadserve(p, 8);
         Threadserve ts3 = new Threadserve(p, 10);
        }
    }
```

The above program gives the following output:

```
Start
16 15 14 13 12 11 10 9 Start
8 7 6 5 Start
10 9 8 7 6 8 7 6 5 4 3 2 1 End
4 3 2 1 End
5 4 3 2 1 End
```

From the above results, it can be seen that printnumber method used by the first thread prints from 16 to 9, jumps to the second thread and prints from 8 to 5, then jumps to the third thread and prints from 10 to 6, goes back to first thread and prints from 8 to 1 and completes the task, then switches to second thread, prints from 4 to 1 and completes the task and comes to the third thread, prints from 5 to 1 and completes the printing. As it can be seen, the results are mixed up, contrary to our expectations. There should be a way out to eliminate this problem.

To avoid methods serving many threads to leave in the middle before completing the task, Java provides a mechanism called synchronization. To do this, the keyword **synchronized** is used. A **synchronized** method, once having taken up a task, cannot be accessed by other threads until it completes the task.

To prevent a method to serve many threads simultaneously, declare it as synchronized.

The synchronization is achieved through the **monitor** concept. When a method is declared as **synchronized**, the object to which it is a member is supposed to have a lock and key. When the method is not accessed by any thread, the key is available to any thread. When a thread wants to make use of the method, it takes the key and locks the object. While the method is doing its task, no other thread can access the method. Once the task is over, the thread releases the object and leaves the key free.

The keyword **synchronized** is used in two forms. In the first form, **synchronized** word is used while declaring the method itself.

synchronized return-type methodname (para-list) {
…
…
method body
}
In the second form, the declaration is done as follows:
synchronized (object) {
statements for synchronization;
}

While an object is locked, other threads cannot access only the synchronized method, but can access the other methods. An object can have any number of synchronized methods. When such an object is locked by one thread for want of one synchronized method, all other synchronized methods of the object cannot be accessed by other threads. When one thread owns the lock, it can access all synchronized methods of that object.

When an object having more than one synchronized method is locked, all synchronized methods cannot be accessed by other threads. However, other non-synchronized methods of that object can be accessed.

Program 15.11 shows the use of **synchronized** method, which is a modified form of program 15.10.

Program 15.11

```
// This program illustrates the use of synchronized method.
class Printing
    {
    synchronized void printnumber(int n)
      {
      System.out.println("Start");
      for (int j = n; j > 0; j–)
            {
            try
                {
                if (j == n / 2)
                    Thread.sleep(100);
                }
            catch (InterruptedException e)
                {
                ;
                }
            System.out.print("  " + j);
            }
        System.out.println("End");
        }
    }
class Threadserve
    implements Runnable
    {
    int n;
    Printing pt;
    Thread th;
    Threadserve(Printing p, int x)
        {
        n = x;
        pt = p;
        th = new Thread(this);
        th.start();
        }
    public void run()
        {
        pt.printnumber(n);
        }
    }
class Threadsynchro1
    {
    public static void main(String args [])
      {
```

```
    Printing p = new Printing();
    Threadserve ts1 = new Threadserve(p, 16);
    Threadserve ts2 = new Threadserve(p, 8);
    Threadserve ts3 = new Threadserve(p, 10);
    }
  }
```

The above program gives the following output:

```
Start
16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 End
Start
8 7 6 5 4 3 2 1 End
Start
10 9 8 7 6 5 4 3 2 1 End
```

You will observe from the above result that the printing of numbers, once started in a thread by the synchronized method, will not be left halfway by the thread without completing it. In this program, the **synchronized** keyword is used in the declaration of the method itself. Alternatively, it can be written in the following form also:

```
.
.
.
class Printing {
void printnumber(int n) {
.
.
.
public void run() {
synchronized(pt) {
pt.printnumber(n);
}
}
.
.
.
```

The above modification in program 15.11 will give the same result as that for 15.11.

15.9 Inter-Thread Communication

Threads are created to carry out light-weight process independently. In certain problems, two or more threads may use an object as a common resource. In order to avoid a mix-up of the task of one thread with that of another thread,

the resource object is synchronized. When one thread is using the synchronized object, the monitor is locked and another thread needing to use this object has to keep waiting. A synchronized object may have more than one synchronized method. One thread may need to use one synchronized method, while another thread may need another synchronized method of the same object. But when a synchronized object is used by one thread, it cannot be accessed by any other thread, even if a different method of the shared object is needed. It may happen that only after an action has taken place in one thread, the other thread can proceed. If the currently running thread can proceed only after an action in another non-running thread, the running thread has to keep waiting infinitely. To avoid such problem, Java provides inter-thread communication methods, which can send messages from one thread to another thread, which uses the same object.

The methods used for inter-thread communication are :

1. wait()
   This method makes the calling thread to give up monitor and go to sleep until some other thread wakes it up.

2. notify()
   This method wakes up the first thread which called **wait()** on the same object.

3. notifyAll()

   This method wakes up all the threads that called **wait()** method on the same object.

All the three methods can be called only inside a synchronized code and are applicable to threads that share the same object.

To illustrate the inter-thread communication, consider a problem of handling a message. It is needed to receive a message from a keyboard and then print it out. It is obvious that the message can be printed only after receiving a message. To start with, we develop a simple program to handle this problem and show that unsynchronized shared object cannot meet our requirement. Program 15.12 illustrates this. The class Message contains two methods Readmesg and Printmesg. Readmesg reads the keyboard and stores the message and Printmesg prints the stored message. To deal with the read process, the thread Receive is developed and another thread Print is developed to deal with the printing process. Both share the same object of the class Message. The two threads are invoked, three times in the main method.

Program 15.12

```
/* This program  reads a message from a keyboard
 and prints it on the screen. Unsynchronized object is used.
*/
import java.io.*;
class Message
    {
    String mesg;
    void Readmesg()
        {
        try
           {
           InputStreamReader ins = new
                     InputStreamReader(System.in);
           BufferedReader br = new BufferedReader(ins);
           System.out.print("Type in a message: ");
           mesg = br.readLine();
           }
        catch (IOException e)
            {
            System.out.println("IO Error");
            }
        }

    void Printmesg()
        {
        System.out.println("The received message is :"
                                + mesg);
        }
    }
class Receive
    implements Runnable
    {
    Message ms;
    Thread th;
    Receive(Message ms)
      {
      this.ms = ms;
      th = new Thread(this);
      th.start();
      }
    public void run()
        {
        ms.Readmesg();
        }
    }
class Print
    implements Runnable
    {
```

```
        Message ms;
        Thread th;
        Print(Message ms)
            {
            this.ms = ms;
            th = new Thread(this);
            th.start();
            }
        public void run()
            {
            ms.Printmesg();
            }
        }
    class Threadcomm1
        {
        public static void main(String args [])
            {
            Message mymsg = new Message();
            for (int i = 0; i < 3; i++)
                {
                new Print(mymsg);
                new Receive(mymsg);
                }
            }
        }
```

The output and the input typed in for the above program are given below:

```
The received message is :null
Type in a message: The received message is :null
Type in a message: The received message is :null
Type in a message: hello
how
are
```

In the output, the words, hello, how and are, are typed separately on the keyboard. The result shows that the Receive thread did not wait for key press nor the Print thread waited for the receipt of the message. Both threads executed their process without waiting for the message. The message printed is a default null value. Only after both threads completed their task, the three words are sensed at the keyboard. The Print thread is called first and the Receive thread is placed second. This is done purposely to show the need for inter-thread communication. To improve this, the methods Readmesg and Printmesg of the class Message is declared as synchronized.

.

.

synchronized void Readmesg() {

```
.
.
}
synchronized void Printmesg() {
.
.
}
```

With the above modification in program 15.12 the output obtained is :

```
The received message is :null
Type in a message: hello
The received message is :hello
Type in a message: how
The received message is :how
Type in a message: are
```

The output shows the first printing is with "null". The Receive thread is waiting for the keyboard press, as the thread uses a synchronized object. Though the Receive thread worked properly, the Print thread did not. It prints the previously received message and not the current message. Now, we make use of **wait** and **notify** methods. Program 15.13 shows the program with **wait** and **notify** methods. The thread containing Readmesg method is made to wait (gives up monitor lock so that another thread can make use of the synchronized object) when a message is already received from the keyboard. If no fresh message is received, it reads the keyboard and calls the **notify** method to inform the other thread that there is a message to print. The thread using Printmesg method is made to wait when there is no message for printing. If there is a message, it prints the message and calls the **notify** method to inform the other thread that the message has been printed and new message can be received. You will notice that the message is received from the keyboard first and printed out. This is the expected result.

Program 15.13

```
/* This program  reads a message from a keyboard
    and prints it on the screen. Synchronized object with
    wait and notify method is used.
*/
import java.io.*;
class Message
    {
    String mesg;
    boolean received = false;
    synchronized void Readmesg()
        {
```

```
        try
            {
            while (received)
                wait();
            }
        catch (InterruptedException e)
            {
            System.out.println("Thread interrupted while
                                waiting");
            }
        try
            {
             InputStreamReader ins = new
                InputStreamReader(System.in);
             BufferedReader br = new BufferedReader(ins);
             System.out.print("Type in a message: ");
             mesg = br.readLine();
            }
        catch (IOException e)
            {
            System.out.println("IO Problem");
            }
        received = true;
        notify();
        }
    synchronized void Printmesg()
        {
        try
            {
            while (!received)
                wait();
            }
        catch (InterruptedException e)
            {
            System.out.println("Thread interrupted while
                                waiting");
            }
        System.out.println("The received message is :"
                           + mesg);
        received = false;
        notify();
        }
    }
class Receive
    implements Runnable
    {
    Message ms;
    Thread th;
    Receive(Message ms)
```

```
        {
        this.ms = ms;
        th = new Thread(this);
        th.start();
        }
    public void run()
        {
        ms.Readmesg();
        }
    }
class Print
    implements Runnable
    {
    Message ms;
    Thread th;
    Print(Message ms)
        {
        this.ms = ms;
        th = new Thread(this);
        th.start();
        }
    public void run()
        {
        ms.Printmesg();
        }
    }
class Threadcomm
    {
    public static void main(String args [])
        {
        Message mymsg = new Message();
        for (int i = 0; i < 3; i++)
            {
            new Print(mymsg);
            new Receive(mymsg);
            }
        }
    }
```

The above program gives the following output:

```
Type in a message: hello
The received message is :hello
Type in a message: how
The received message is :how
Type in a message: are
The received message is :are
```

The wait(), notify(), notifyAll() methods can be called only inside synchronized method.

After reading this chapter, you should have learned the following:

- ➲ Thread is an individual subtask running independently.
- ➲ A program may contain many threads to perform various subtasks.
- ➲ Threads can communicate with other threads.

In the next chapter, you will learn about applets.

Worked Out Problems–15

Problem 15.1w

Write a Java program to compute the Sine and Cosine values, using the formula given in problem 5.1w. Use one thread to compute Sine function and another thread to compute Cosine function.

Program 15.1w

```
/* ------------------------------------------------------------
 This program calculates the sin(x) and cos(x) functions by
 computing the sin series and cos series functions using
 thread technique.

 somasundaramk@yahoo.com
------------------------------------------------------------ */

class Trigno
    {
    double x;
    int i, n = 5;
    long factn;
    Trigno(double x)
        {
        this.x = x;
        }
    long Factfun(int nmax)
        {
        long fact = 1;
        for (i = 2; i <= nmax; i++)
```

```
            fact *= i;
        return fact;
        }
    synchronized double Sinfun()
        {
        double six = x;
        for (int k = 1; k <= n; k++)
            {
            factn = Factfun(2 * k + 1);
            six = six + Math.pow(-1, k) * Math.pow(x, (2 * k +
                1)) / factn;
            }
        return six;
        }
    synchronized double Cosfun()
        {
        double cox = 1;
        for (int k = 1; k <= n; k++)
            {
            factn = Factfun(2 * k);
          cox = cox + Math.pow(-1, k) * Math.pow(x,2* k)/factn;
            }
        return cox;
        }
    }
class Sine
    implements Runnable
    {
    double sinval;
    Trigno trgno;
    double Sincomp(Trigno trg)
     {
     trgno = trg;
     Thread th = new Thread(this);
     th.start();
     try
            {
            th.sleep(50);
            }
        catch (InterruptedException ie)
            {       ;           }
        return sinval;
        }
    public void run()
        {
        sinval = trgno.Sinfun();
        }
    }
class Cosine
    implements Runnable
```

```
    {
    double cosval;
    Trigno trgno;
    double Coscomp(Trigno trg)
        {
        trgno = trg;
        Thread th = new Thread(this);
        th.start();
        try
            {
            th.sleep(50);
            }
        catch (InterruptedException ie)
            {         ;            }
        return cosval;
        }
    public void run()
        {
        cosval = trgno.Cosfun();
        }
    }
class Prob151
    {
    public static void main(String args [])
          {
          int pn = 80;
          double x, sinx, cosx, jsx, jcx;
          Trigno trg;
          Sine sino;
          Cosine coso;
          for (int i = 0; i < pn; i++)
             System.out.print("-");
          System.out.println("\n");
          System.out.println("x \t my sinx \t Java sinx \t my
                    cosx \t Java cosx\n");
          for (int i = 0; i < pn; i++)
             System.out.print("-");
          System.out.println("\n");
          for (x = 0; x < 1.6; )
               {
               trg = new Trigno(x);
               sino = new Sine();
               coso = new Cosine();
               sinx = sino.Sincomp(trg);
               cosx = coso.Coscomp(trg);
               // Reduce the fractional digits for display
               double sx = (int)(sinx * 1000);
               double cx = (int)(cosx * 1000);
               sinx = sx / 1000;
               cosx = cx / 1000;
```

```
                jsx = (int)(Math.sin(x) * 1000);
                jcx = (int)(Math.cos(x) * 1000);
                jsx = jsx / 1000;
                jcx = jcx / 1000;
                System.out.println(x + "\t" + sinx + "\t\t" +
                     jsx + "\t\t" + cosx + "\t\t" + jcx);
                x = x + 0.5;
                }
    for (int i = 0; i < pn; i++)
            System.out.print("-");
     System.out.println("\n");
    }
 }
```

The above program gives the following output:

---

x	my sinx	Java sinx	my cosx	Java cosx
0.0	0.0	0.0	1.0	1.0
0.5	0.479	0.479	0.877	0.877
1.0	0.841	0.841	0.54	0.54
1.5	0.997	0.997	0.07	0.07

Problem 15.2w

Write a Java program to make two player number game. Each player has to feed an integer when his/her turn comes (to keep the players interacting with the progam). A random number will be generated in the range 0 to 200. A player who gets a larger number is the winner. A winner gets points equal to the difference in the random numbers.

Program 15.2w

```
/* ------------------------------------------------------------
This program makes two player number game. Players are to feed in
an integer number which is not used in the program, but keeps the
player engaged in the game. For each player, a random number is
generated. The player who  gets higher number is the winner.  The
winner gets  points equal  to the difference between the two random
numbers.

somasundaramk@yahoo.com
------------------------------------------------------------ */
```

```
import java.util.*;
import java.io.*;
import java.text.*;
class Game
    {
    int sn1, rand1;
    int sn2, rand2;
    Random r1, r2;
    boolean played = false;
    DecimalFormat df = new DecimalFormat();
    Number n1, n2;
    String str;
    synchronized int Play1()
        {
        try
            {
            while (played)
                wait();
            }
        catch (InterruptedException e)
            {
            System.out.println("Thread interrupted while
                    waiting");
            }
        try
          {
          InputStreamReader ins = new
               InputStreamReader(System.in);
          BufferedReader br = new BufferedReader(ins);
          System.out.print("Player 1: Type in an integer
                                    number ");
          str = br.readLine();
          n1 = df.parse(str);
          sn1 = n1.intValue();
          // genarate random number
          r1 = new Random();
          // get random numbers from 0 to 200
          rand1 = r1.nextInt(200);
          }
        catch (IOException e)
            {
            System.out.println("IO Problem");
            }
        catch (ParseException pe)
            {
            System.out.println("Parsing error");
            }
        played = true;
        notify();
        return rand1;
```

```
        }
    synchronized int Play2()
        {
        // keep waiting till the other player has played
        try
            {
            while (!played)
                wait();
            }
        catch (InterruptedException e)
            {
            System.out.println("Thread interrupted while
                                        waiting");
            }
        try
          {
          InputStreamReader ins = new
               InputStreamReader(System.in);
          BufferedReader br = new BufferedReader(ins);
          System.out.print("Player 2: Type in an integer
               number ");
          str = br.readLine();
          n2 = df.parse(str);
          sn2 = n2.intValue();
          // genarate random number
          r2 = new Random();
          // get random numbers from 0 to 200
          rand2 = r2.nextInt(200);
          }
        catch (IOException e)
            {
            System.out.println("IO Problem");
            }
        catch (ParseException pe)
            {
            System.out.println("Parsing error");
            }
        played = false;
        notify();
        return rand2;
        }
    }
class Player1
    implements Runnable
    {
    Game gm;
    int score;
    Thread th;
    int Play(Game gme)
          {
```

```
          gm = gme;
          th = new Thread(this);
          th.start();
          try
          {
          th.join();
          }


        catch (InterruptedException ie)
            {
            ;
            }
        return score;
        }
    public void run()
        {
        score = gm.Play1();
        }
    }
class Player2
    implements Runnable
    {
    Game gm;
    int score;
    Thread th;
    int Play(Game gme)
          {
          gm = gme;
          th = new Thread(this);
          th.start();
          try
          {
          th.join();
          }
        catch (InterruptedException ie)
            {
            ;
            }
        return score;
        }
    public void run()
        {
        score = gm.Play2();
        }
    }
class Prob152
    {
    public static void main(String args [])
          {
```

```
        Game gam = new Game();
        Player1 p1;
        Player2 p2;
        int points1 = 0;
        int points2 = 0;
        int score1, score2;
        int gamecount = 0;

        while (true)
        {
        p1 = new Player1();
        score1 = p1.Play(gam);
        p2 = new Player2();
        score2 = p2.Play(gam);
        points1 += ((score1 > score2) ? score1 - score2 : 0);
        points2 += ((score2 > score1) ? score2 - score1 : 0);
        gamecount++;
        System.out.println("\nThis game:\n Player 1 score : "
                  + score1 + "\n Player 2 score : " + score2);
             if (score1 > score2)
                  System.out.println("\nPlayer1 wins");
             else
                  System.out.println("\nPlayer2 wins");
        System.out.println("\nTotal  Points after " +
                                  gamecount + " games");
        System.out.println("\n Player 1 : " + points1 + "
             points" + "\n Player 2 : " + points2 + "
             points");
        try
             {
             InputStreamReader ins = new
                   InputStreamReader(System.in);
             BufferedReader br = new BufferedReader(ins);
             System.out.print("\nDo you want to continue
                  another (y/n): ");
             String str = br.readLine();
             if (str.equals("n"))
                  break;
             }
          catch (IOException e)
             {
             System.out.println("IO Problem");
             }
          }
     }
}
```

The above program gives the following output:

```
Player 1: Type in an integer number 33
Player 2: Type in an integer number 55

This game:
 Player 1 score : 70
 Player 2 score : 99

Player2 wins

Total  Points after 1 games

 Player 1 : 0 points
 Player 2 : 29 points

Do you want to continue another (y/n):y
Player 1: Type in an integer number 79
Player 2: Type in an integer number 0

This game:
 Player 1 score : 157
 Player 2 score : 190

Player2 wins

Total  Points after 2 games

 Player 1 : 0 points
 Player 2 : 62 points

Do you want to continue another (y/n):y
Player 1: Type in an integer number 1234
Player 2: Type in an integer number 965

This game:
 Player 1 score : 175
 Player 2 score : 137

Player1 wins

Total  Points after 3 games
 Player 1 : 38 points
 Player 2 : 62 points

Do you want to continue another (y/n): y
Player 1: Type in an integer number 654
Player 2: Type in an integer number 67
```

```
This game:
 Player 1 score : 85
 Player 2 score : 25

Player1 wins

Total  Points after 4 games
 Player 1 : 98 points
 Player 2 : 62 points

Do you want to continue another (y/n): y
Player 1: Type in an integer number 89
Player 2: Type in an integer number 432

This game:
 Player 1 score : 154
 Player 2 score : 126

Player1 wins

Total  Points after 5 games

 Player 1 : 126 points
 Player 2 : 62 points

Do you want to continue another (y/n): y
Player 1: Type in an integer number 45
Player 2: Type in an integer number 99

This game:
 Player 1 score : 55
 Player 2 score : 20

Player1 wins

Total  Points after 6 games

 Player 1 : 161 points
 Player 2 : 62 points

Do you want to continue another (y/n): n
```

Exercise-15

I. Fill in the blanks

15.1. When several tasks are handled by a single CPU, it is called _____ .

15.2. When several sub-tasks of one task is handled in a CPU, it is called _____ .

15.3. Threads can be created by extending _____ class.

15.4. Threads can be created by implementing _____ interface.

15.5. All codes that are to be executed in a thread are to be placed inside the _____ method.

15.6. When a thread is created using new operator, the thread is in _____ state.

15.7. To bring a thread to a runnable state, the _____ method is to be called.

15.8. When a thread is waiting for an action (like I/O operation) it is said to be in _____ state.

15.9. At any one instance of time _____ thread(s) will be in running state.

15.10. When an object is used as a common resource, the object is to be _____ to avoid mix-up of tasks.

15.11. wait() and notify() methods can be used only inside a _____ code.

15.12. The decision, which thread is to enter the CPU, is made by the _____ .

15.13. A thread with lower priority value will be given _____ priority than a thread with higher priority value.

II. Write Java program for the following:

15.14. Write a Java program to convert the sequence of characters AB*CD/+ representing the Polish notation to the original expression A*B+C/D. Use two threads to perform the evaluation.

15.15. Write a Java program to compute the first 25 prime numbers. Also compute the first 50 Fibonacci numbers given by $f_n = f_{n-1} + f_{n-2}$, with

f1=f2=1. Create two threads to compute each one of them. Set the priority of thread that computes Fibonacci number to 8 and the other to 5. After calculating 50 Fibonacci numbers, make that thread to sleep and take up the prime number computation. After computing the 25 prime numbers continue the Fibonacci number computing.

15.16. A bank account is operated by a father and his son. The account is opened with an initial deposit of Rs. 600. Thereafter, the father deposits a random amount between Re 1 and Rs 200 each time, until the account balance crosses Rs. 2,000. The son can start withdrawing the amount only if the balance exceeds Rs. 2,000. Thereafter, the son withdraws random amount between Re 1 and Rs 150, until the balance goes below Rs. 500. Once the balance becomes less than Rs 500, the father deposits amount till it crosses Rs. 2,000 and the process continues. Write a Father and Son thread to carry out the above process.

* * * * * *