THREADS

CHAPTER

26

A thread represents a separate path of execution of a group of statements. In a Java program, if we write a group of statements, then these statements are executed by JVM one by one. This execution is called a thread, because JVM uses a thread to execute these statements. This means that in every Java program, there is always a thread running internally. This thread is used by JVM to execute the program statements. What is this thread? Let us write a program to see what that thread is:

Program 1: Write a program to find the thread used by JVM to execute the statements.

```
//To find currently running thread in this program
class Current
{
        public static void main(String args[ ])
        {
                System.out.println("Let us find current thread");
                Thread t = Thread.currentThread();
                System.out.println("Current thread= "+t);
                System.out.println("Its name= "+t.getName());
        }
}
```

Output:

```
C:\> javac Current.java
C:\> java Current
Let us find current thread
Current thread= Thread[main,5,main]
Its name= main
```

In the preceding program, `currentThread()` is a static method in `Thread` class. So we called it as `Thread.currentThread()`. Then this method gave an object `t` of `Thread` class. When we displayed this object `t`, it displayed its contents as:

```
Thread[main,5,main]
```

Here, `Thread` indicates that `t` is a `Thread` class object. And the first `main` indicates the name of the thread running the current code. We get `5` which is a number representing the priority of the thread. Every thread will have a priority number associated with it. These priority numbers will range from 1 to 10. 1 is the minimum priority, and 10 is the maximum priority of a thread. If the

priority number of a thread is more, it is given more preference while execution by JVM. The default priority number of a thread is 5.

The next `main` indicates the thread group name to which this thread belongs. A thread group represents a group of threads as a single unit. The 'main' thread belongs to 'main' thread group. See the last line in the output of the program. We are displaying the currently running thread's name which is 'main'. If a thread is `t`, its name can be known using `t.getName()` method.

What does the preceding program indicate? When we write any program in Java, JVM internally uses a thread called 'main thread' to run the statements of the program. This thread is responsible for executing our statements and displaying the results.

Important Interview Question

Which thread always runs in a Java program by default?

main thread.

A thread represents execution of statements. The way the statements are executed is of two types: 1) Single tasking 2) Multi tasking.

Single Tasking

A task means doing some calculation, processing, etc. Generally, a task involves execution of a group of statements, for example executing a program. In 'single tasking' environment, only one task is given to the processor at a time. See Figure 26.1. Here, let us take an example of a student who goes to the lab to write some programs. He types his first program. It may be taking some 10 minutes. When he is typing his program, the micro processor is sitting idle, without any work. After typing is completed, he gives the program to the processor. The processor executes it within a millisecond. Since now-a-days processors can execute millions of instructions per second, even if the student has written a program of 100 lines, the processor will not take more than a fraction of a second to complete its execution. After verifying the results, the student has started typing the second program. It may take another 10 or 15 minutes. Within this time, the processor sits idle, waiting for the job. When the student submits the second program, then it works for a fraction of a second on the program. Then again it goes into idle state. From this discussion, we can understand that the processor is sitting idle without any work for most of the time.

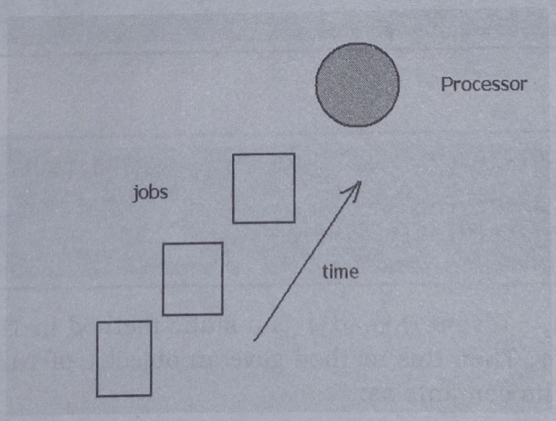


Figure 26.1 Single tasking

In single tasking, only one task is given to the processor at a time. This means we are wasting a lot of processor time and microprocessor has to sit idle without any job for a long time. This is the drawback in single tasking.

Multi Tasking

To use the processor's time in an optimum way, we can give it several jobs at a time. This is called multi tasking. But how can we give several jobs at a time? Suppose there are 4 tasks that we want to execute. We load them into the memory, as shown in Figure 26.2. The memory is divided into 4 parts and the jobs are loaded there. Now, the micro processor has to execute them all at a time. So the processor will take small time duration, like a millisecond and divide this time between the number of jobs. Here, 4 jobs are there. So we get ¼ millisecond time for each job. This small part of the processor time is called 'time slice'. It will allot exactly ¼ millisecond time for executing each of the jobs. Within this time slice, it will try to execute each job. Suppose, it started at first job, it will spend exactly ¼ millisecond time executing the first job. Within this time duration, if it could not complete the first job, then what it does? In that case, it stores the intermediate results till then it obtained in a temporary memory, and then it goes to the second task. It then spends exactly ¼ millisecond time executing the second task. Within this time, if it can complete this task, no problem. Suppose it could not complete this task, then it goes to the third task, storing the results in a temporary memory. Similarly, it will spend exactly ¼ millisecond for third task, and another ¼ millisecond for the fourth task. After executing the fourth task, it will come back to the first task, in a circular manner. This is called 'round robin' method.

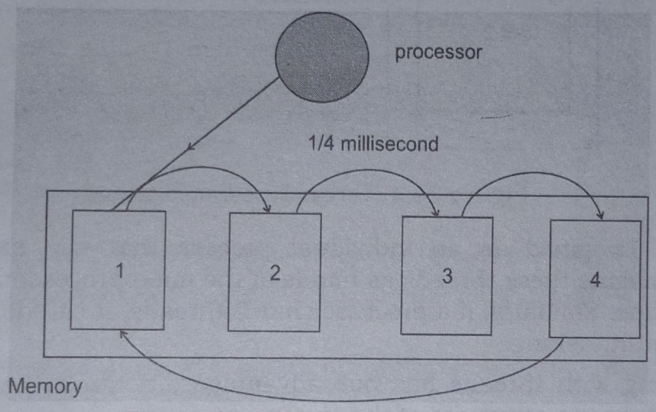


Figure 26.2 Process-based multi tasking

The processor, after returning to the first task, again starts execution from the point, where it has left that task earlier. It will execute the first task exactly for ¼ millisecond this time, and proceeds for the second task and then third and fourth before coming back to the first task in a round robin way. So if you have submitted the first job, you can understand that the processor is executing your job for ¼ millisecond and then keeping you waiting for another ¾ millisecond, while it is going and executing the other tasks. After ¾ millisecond, it is again coming back to your job and executing your job for another ¼ millisecond time. But you will not be aware that you are kept waiting for ¾ millisecond time, as this time is very small. You will feel that the processor is spending its time executing your job only. Similarly, the second person who submitted the second job will also feel that only his job is being executed by the processor. The third and fourth persons will also feel the same way. It is something like all the 4 jobs are executed by the processor simultaneously. This is called multi tasking.

Generally, we have only one processor in our Computer systems. One processor has to execute several tasks means that it has to execute them in a round robin method. Strictly speaking, this is not multi tasking, since the processor is quickly executing the tasks one by one, so quickly that we feel all the jobs are executed by the processor at a time. Multitasking cannot be a real phenomenon with single processor systems. If we want to really achieve multi tasking, we need Computers with multiple processors.

The main advantage of multi tasking is to use the processor time in a better way. We are engaging most of the processor time and it is not sitting idle. In this way, we can complete several tasks at a time, and thus achieve good performance.

Multi tasking is of two types: a) Process-based multi tasking b) Thread-based multi tasking. So far, we discussed the first type, i.e. Process-based multi tasking. Now let us think about Thread-based multi tasking.

In Process-based multi tasking, several programs are executed at a time, by the microprocessor. In Thread-based multi tasking, several parts of the same program is executed at a time, by the microprocessor. See Figure 26.3. Here, we have a program. In this program, there are 2 parts. These parts may represent two separate blocks of code or two separate methods containing code. Each part may perform a separate task. The processor should execute the two parts (tasks) simultaneously. So the processor uses 2 separate threads to execute these two parts.

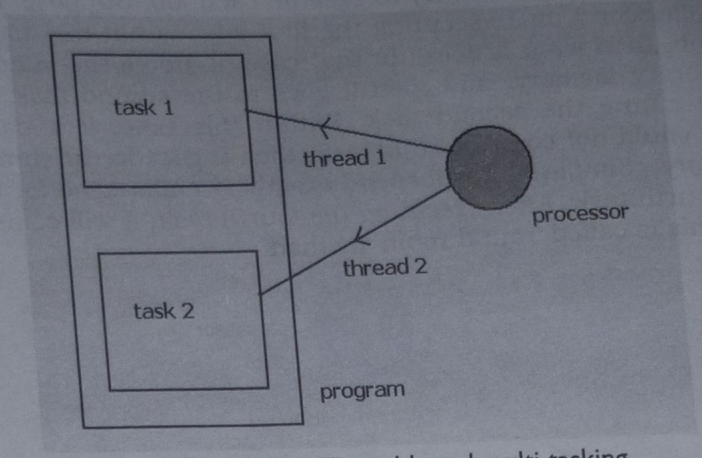


Figure 26.3 Thread-based multi tasking

Each thread can be imagined as an individual process that can execute a separate set o statements. We can imagine these threads as hands of the microprocessor. We have 2 hands, so w can do 2 things at a time. Similarly, if a processor has 2 threads, it can do 2 tasks at a time. This called Thread-based multitasking.

Achieving multi tasking with threads has one advantage ,i.e. since the threads are light weig processes, they use minimum resources of the system.

Important Interview Question

Why threads are called light-weight?

Threads are light-weight because they utilize minimum resources of the system. This means the take less memory and less processor time.

What is the difference between single tasking and multi tasking?

Executing only one job at a time is called single tasking. Executing several jobs at a time is cal multi tasking. In single tasking, the processor time is wasted, but in multi tasking, we can utilize processor time in an optimum way.

Uses of Threads

`Threads` can be used for multiple purposes. Some of the uses of threads are:

- Threads are mainly used in server-side programs to serve the needs of multiple clie network or Internet. On Internet, a server machine has to cater the needs of thou clients, at a time. For this purpose, if we use threads in the server, they can do variou a time, thus they can handle several clients.

- Threads are also used to create games and animation. Animation means moving the objects from one place to another. In many games, generally we have to perform more than one task simultaneously. There, threads will be of invaluable help. For example, in a game, a flight may be moving from left to right. A machine gun should shoot it, releasing the bullets at the flight. These 2 tasks should happen simultaneously. For this purpose, we can use 2 threads, one thread will move the flight and the other one will move the bullet, simultaneously towards the flight.

Creating a Thread and Running it

We know that in every Java program, there is a main thread available already. Apart from this main thread, we can also create our own threads in a program. The following steps should be used:

- Create a class that extends `Thread` class or implements `Runnable` interface. Both the `Thread` class and `Runnable` interface are found in `java.lang` package.

```
class Myclass extends Thread
Or, class Myclass implements Runnable
```

- Now in this class, write a `run()` method as:

```
public void run()
{
      statements;
}
```

By default, this `run()` method is recognized and executed by a thread.

- Create an object to `Myclass`, so that the `run()` method is available for execution.

```
Myclass obj = new Myclass();
```

- Now, create a thread and attach the thread to the object `obj`.

```
Thread t = new Thread(obj);
Or,  Thread t = new Thread(obj, "threadname");
```

- Run the thread. For this purpose, we should use `start()` method of `Thread` class.

```
t.start();
```

Now, the thread will start execution on the object of `Myclass`. In that object, `run()` method is found, hence it will execute the statements inside that `run()` method.

By following these steps, let us write a sample program to understand how to create a thread.

Program 2: Write a program to create `MyThread` class with `run()` method and then attach a thread to this `MyThread` class object.

```
//To create a thread and run it
//let the class extends Thread or implements Runnable
class  MyThread extends Thread
{
      //write run() method inside this class
      public void run()
      {
            //only this code is executed by the thread
            for(int i=1; i<=100000; i++)
```

```
C:\> java Demo1
1
2
3
4
5
:
:
```

In this program, we create `MyThread` class with `run()` method and then attach a thread to this `MyThread` class object. When we run the thread, it runs the `run()` method of `MyThread` object.

Here, the class `MyThread` extends `Thread` class. We can also replace this statement with a statement like `class MyThread implements Runnable`. `Thread` class and `Runnable` interface both have `public void run()` method in them. By writing `class MyThread extends Thread`, we are overriding the `run()` method of `Thread` class. By writing `class MyThread implements Runnable` interface, we are implementing the `run()` method of the `Runnable` interface.

In `Demo1` class, we created an object to `MyThread` class. This means that object contains `run()` method. Then we attached this object to thread `t`, as:

```
Thread t = new Thread(obj);
```

If we give `t.start()`, then the thread `t` starts running the code inside the `run()` method of the object `obj`. In the `run()` method, we wrote code to print numbers from 1 to 100000 using a `for` loop. As a result, the numbers will be displayed starting from 1 to 100000. If you want to terminate the program in the middle, you can press Ctrl+C from the keyboard. This leads to abnormal program termination. It means the entire program is terminated, not just the thread.

If we want to terminate only the thread that is running the code inside run() method, we should devise our own mechanism. If we press Ctrl+C, we are abnormally terminating the program. This is dangerous. Abnormal program termination may cause loss of data and lead to unreliable results. So we should terminate the thread only, not the program. How can we terminate the thread smoothly is the question now.

you stop a thread in Java?

First of all, we should create a boolean type variable which stores 'false'. When the user wants to stop the thread, we should store 'true' into the variable. The status of the variable is checked in the run() method and if it is true, the thread executes 'return' statement and then stops.

Now, let us re-write the previous program, incorporating the logic to stop the thread smoothly.

Program 3: Re-write Program 2 showing how to terminate the thread by pressing the Enter button

```
//To create a thread and run it, then stop it
import java.io.*;

class  MyThread extends Thread
{
      boolean stop = false;

      public void run()
      {
            for(int i=1; i<=100000; i++)
            {
                  System.out.println(i);
                  if(stop) return;     //come out of run()
            }
      }
}

class Demo1
{
      public static void main(String args[]) throws IOException
      {
            MyThread obj = new MyThread();
            Thread t = new Thread(obj);
            t.start();

            //stop the thread when Enter key is pressed
            System.in.read();  //wait till Enter key pressed
```

Output:

```
C:\> javac Demo1.java
C:\> java Demo1
1
2
3
4
5
6
:
:
Press <Enter> to stop the thread at any time.
```

This is same as Program 2, except that here we terminate the thread when the user presses the Enter button.

Important Interview Question

What is the difference between 'extends Thread' and 'implements Runnable' ? Which one is advantageous?

extends Thread and implements Runnable – both are functionally same. But when we write extends Thread, there is no scope to extend another class, as multiple inheritance is not supported in Java.

class Myclass extends Thread, AnotherClass //invalid

If we write implements Runnable, then still there is scope to extend another class.

class Myclass extends AnotherClass implements Runnable //valid

This is definitely advantageous when the programmer wants to use threads and also wants to access the features of another class.

Single Tasking Using a Thread

A thread can be employed to execute one task at a time. Suppose there are 3 tasks to be executed. We can create a thread and pass the 3 tasks one by one to the thread. For this purpose, we can write all these tasks separately in separate methods: `task1()`, `task2()`, `task3()`. Then these methods should be called from `run()` method, one by one. Remember, a thread executes only the code inside the `run()` method. It can never execute other methods unless they are called from `run()`.

Important Interview Question

Which method is executed by the thread by default?

public void run() method.

Program 4: Write a program showing execution of multiple tasks with a single thread.

```
//single tasking using a thread
class  MyThread  implements Runnable
{
```

```
        public void run()
        {
                //execute the tasks one by one by calling the methods.
                task1();
                task2();
                task3();
        }

        void task1()
        {
                System.out.println("This is task 1");
        }
        void task2()
        {
                System.out.println("This is task 2");
        }
        void task3()
        {
                System.out.println("This is task 3");
        }
    }

    class Single
    {
        public static void main(String args[])
        {
                //create an object to MyThread class.
                MyThread obj = new MyThread();

                //create a thread t1 and attach it to that object
                Thread t1 = new Thread(obj);

                //execute the thread t1 on that object's run() method.
                t1.start();
        }
    }
```

Output:

```
C:\> javac Single.java
C:\> java Single
This is task 1
This is task 2
This is task 3
```

In this program, a single thread `t1` is used to execute three tasks.

Multi Tasking Using Threads

In multi tasking, several tasks are executed at a time. For this purpose, we need more than one thread. For example, to perform 2 tasks, we can take 2 threads and attach them to the 2 tasks. Then those tasks are simultaneously executed by the two threads. Using more than one thread is called 'multi threading'.

When we go to a movie theatre, generally a person is there at the door—checking and cutting the tickets. When we enter the hall, there is another person who shows the seats to us. Suppose there is only one person (1 thread) doing these two tasks. He has to first cut the ticket and then come along with us to show the seat. Then he goes back to the door to cut the second ticket and then again enter the hall to show the seat for the second ticket. Like this, if he is does the things one by one, it takes a lot of time, and even though the show is over, there will be still a few people left outside the door waiting to enter the hall! This is pretty well known to the theatre management. So what they do? They employ two persons (2 threads) for this purpose. The first person will cut the

ticket, and the second one will show the seat. When the second person is showing the seat, the first person cuts the second ticket. Like this, both the persons can act simultaneously and hence there will be no wastage of time. This is shown in Program 5.

Program 5: Write a program showing two threads working simultaneously upon two objects.

```
//Two threads performing two tasks at a time - Theatre example
class MyThread implements Runnable
{
        //declare a string to represent the task
        String str;

        MyThread(String str)
        {
                this.str = str;
        }

        public void run()
        {
                for(int i=1; i<=10; i++)
                {
                        System.out.println(str+ " : "+i);
                        try{
                                Thread.sleep(2000);
                                //cease thread execution for 2000 milliseconds
                        } catch(InterruptedException ie)
                {
                        ie.printStackTrace();
                }
                } //end of for

        } //end of run()
}

class Theatre
{
        public static void main(String args[])
        {
                //create two objects to represent two tasks
                MyThread obj1 = new MyThread("Cut the ticket");
                MyThread obj2 = new MyThread("Show the seat");

                //create two threads and attach them to the two objects
                Thread t1 = new Thread(obj1);
                Thread t2 = new Thread(obj2);

                //start the threads
                t1.start();
                t2.start();

        }
}
```

Output:

```
C:\> javac Theatre.java
C:\> java Theatre
Cut the ticket: 1
Show the seat: 1
Cut the ticket: 2
Show the seat: 2
Cut the ticket: 3
Show the seat: 3
:
Cut the ticket: 10
Show the seat: 10
```

In the preceding example, first we have taken a string variable `str` in `MyThread` class. Then we passed two strings—`Cut the ticket` and `Show the seat`—into that variable from `Theatre` class. When `t1.start()` is executed, it starts execution on `run()` method code showing `Cut the ticket`. Just behind it, `t2.start()` will make the thread `t2` also execute on `run()` method, almost simultaneously. So it will display `Show the seat`. In this manner, both the threads are simultaneously doing the two tasks. Note that in `run()` method, we used:

```
try{
   Thread.sleep(2000);
}
catch(InterruptedException ie)
 {
  ie.printStackTrace();
 }
```

Here, `sleep()` is a static method in `Thread` class, which is used to suspend execution of a thread for some specified milliseconds. For example, `Thread.sleep(2000)` will stop the execution of the thread for 2000 milliseconds, i.e. 2 seconds. (1000 milliseconds = 1 second). Since this method can throw `InterruptedException`, we caught it in `catch` block.

Multiple Threads Acting on Single Object

In theatre example, we have used 2 threads on the 2 objects of `MyThread` class. It is also possible to use 2 or more threads on a single object. But in this case, sometimes we get unreliable results.

First let us see why 2 threads should share the same object (same `run()` method). We write an object to represent one task. If there is a different task, we take another object. When two people (threads) perform same task, then they need same object (`run()` method) to be executed each time. Take the case of railway reservation. Every day several people want reservation of a berth for them. The procedure to reserve the berth is same for all the people. So we need same object with same `run()` method to be executed repeatedly for all the people (threads).

Let us think that only one berth is available in a train, and two passengers (threads) are asking for that berth. In reservation counter no.1, the clerk has sent a request to the server to allot that berth to his passenger. In counter no.2, the second clerk has also sent a request to the server to allot that berth to his passenger. Let us see now see to whom that berth is allotted.

Program 6: Write a program showing two threads acting upon a single object.

```
//Thread unsafe - Two threads acting on same object
class Reserve implements Runnable
{
      //available berths are 1
      int available=1;
      int wanted;

      //accept wanted berths at run time
      Reserve(int i)
      {
            wanted=i;
      }

      public void run()
      {
            //display available berths
            System.out.println("Available Berths= "+available);
            //if available berths are more than wanted berths
            if(available >= wanted)
            {
                  //get the name of passenger
```

```
                    String name= Thread.currentThread().getName();
                    //allot the berth to him
                    System.out.println(wanted +" Berths reserved for " +name);
                    try{
                    Thread.sleep(1500); //wait for printing the ticket
                    available = available - wanted;
                    //update the no. of available berths
                    }catch(InterruptedException ie){}

                }
                //if available berths are less, display sorry
                else System.out.println("Sorry, no berths");

        }
}
class Unsafe
{
        public static void main(String args[])
        {
                //tell that 1 berth is needed
                Reserve obj = new Reserve(1);

                //attach first thread to the object
                Thread t1 = new Thread(obj);
                //attach second thread to the same object
                Thread t2 = new Thread(obj);

                //take the thread names as persons names
                t1.setName("First person");
                t2.setName("Second person");

                //send the requests for berth
                t1.start();
                t2.start();
        }
}
```

Output:

```
C:\> javac Unsafe.java
C:\> java Unsafe
Available Berths = 1
1 Berths reserved for First Person
Available Berths = 1
1 Berths reserved for Second Person
```

Please observe the output in the preceding program. It is absurd. It has allotted the same berth to both the passengers. In this program, already we have taken available berths as 1. When thread `t1` enters the `run()` method, it sees available number of berths as `1` and hence, it allots it to First Person, and displays:

```
1 Berths reserved for First Person
```

Then it enters `try{ }` block inside `run()` method, where it will sleep for 1.5 seconds. In this time, the ticket will be printed on the printer. When the first thread is sleeping, thread `t2` also enters the `run()` method, it also sees that there is 1 berth remaining. The reason is for this is that the available number of berths is not yet updated by the first thread. So the second thread also sees 1 berth as available, and it allots the same berth to the Second Person. Then the thread `t2` will also go into sleep state.

Thread t1 wakes up first, and then it updates the available number of berths as:

```
available = available - wanted;
```

Now available number of berths will become 0. But by this time, the second thread has already allotted the same berth to the Second Person also. Since both the threads are acting on the same object simultaneously, the result is unreliable.

What is the solution for this problem? Let us keep the second thread t2 wait till the first thread t1 completes and comes out. Let us not allow any other thread to enter the object till t1 comes out. This means we are preventing the threads to act on the same object simultaneously. This is called Thread Synchronization or Thread safe. See Figure 26.4.

Important Interview Question

What is Thread synchronization?

When a thread is already acting on an object, preventing any other thread from acting on the same object is called 'Thread synchronization' or 'Thread safe' The object on which the threads are synchronized is called 'synchronized object'. Thread synchronization is recommended when multiple threads are used on the same object (in multithreading).

Synchronized object is like a locked object, locked on a thread. It is like a room with only one door. A person has entered the room and locked from it from behind. The second person who wants to enter the room should wait till the first person comes out. In this way, a thread also locks the object after entering it. Then the next thread cannot enter it till the first thread comes out. This means the object is locked mutually on threads. So, this object is called 'mutex' (mutually exclusive lock).

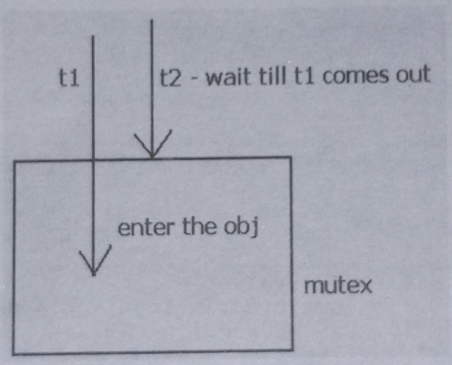


Figure 26.4 Thread synchronization

How can we synchronize the object? There are two ways of doing this.

- Using synchronized block: Here, we can embed a group of statements of the object (inside run() method) within a synchronized block, as shown here:

```
synchronized(object)
{
    statements;
}
```

Here, object represents the object to be locked or synchronized. The statements inside the synchronized block are all available to only one thread at a time. They are not available to more than one thread simultaneously.

- Using synchronized keyword: We can synchronize an entire method by using synchronized keyword. For example, if we want to synchronize the code of display() method, then add the synchronized keyword before the method name as shown here:

```
//Thread Synchronization
class Reserve implements Runnable
{
	//available berths are 1
	int available=1;
	int wanted;

	//accept wanted berths at run time
	Reserve(int i)
	{
		wanted=i;
	}

	public void run()
	{
		synchronized(this)  //synchronize the current object
		{
			//display available berths
			System.out.println("Available Berths= "+available);
			//if available berths are more than wanted berths
			if(available >= wanted)
			{
				//get the name of passenger
				String name= Thread.currentThread().getName();
				//allot the berth to him
				System.out.println(wanted +" Berths reserved for "
				 +name);
				try{
				Thread.sleep(1500); //wait for printing the ticket
				available = available - wanted;
				//update the no. of available berths
				}catch(InterruptedException ie){}

			}
			//if avaiable berths are less, display sorry
			else System.out.println("Sorry, no berths");
		} //end of synchronized block
	}
}
class Safe
{
	public static void main(String args[])
```

```
        {
                //tell that 1 berth is needed
                Reserve obj = new Reserve(1);

                //attach first thread to the object
                Thread t1 = new Thread(obj);
                //attach second thread to the same object
                Thread t2 = new Thread(obj);

                //take the thread names as persons names
                t1.setName("First person");
                t2.setName("Second person");
                //send the requests for berth
                t1.start();
                t2.start();
        }
    }
```

Output:

```
C:\> javac Safe.java
C:\> java Safe
Available Berths = 1
1 Berths reserved for First Person
Available Berths = 0
Sorry, no berths
```

Thread Class Methods

So far, we discussed some concepts of threads, it is time we listed out some important methods of `java.lang.Thread` class:

To create a thread, we can use the following forms:

```
Thread t1 = new Thread();   //thread is created without any name
Thread t2 = new Thread(obj); //here, obj is target object of the thread
Thread t3 = new Thread(obj, "thread-name");   //target object and thread name
//are given
```

- To know the currently running thread:

```
Thread t = Thread.currentThread();
```

- To start a thread:

```
t.start();
```

- To stop execution of a thread for a specified time:

```
Thread.sleep(millseconds);
```

- To get the name of a thread:

```
String name = t.getName();
```

- To set a new name to a thread:

```
t.setName("new name");
```

- To get the priority of a thread:

```
int priority_no= t.getPriority();
```

- To set the priority of a thread:

```
t.setPriotity(int priority_no);
```

Note

Thread priorities can change from 1 to 10. We can also use the following constants to represent priorities:

Thread.MAX_PRIORITY value is 10

Thread.MIN_PRIORITY value is 1

Thread.NORM_PRIORITY value is 5

- To test if a thread is still alive:

```
t.isAlive()  returns true/false.
```

- To wait till a thread dies:

```
t.join();
```

Deadlock of Threads

Even if we synchronize the threads, there is possibility of other problems like 'deadlock'. Let us understand this with an example.

Daily, thousands of people book tickets in trains and cancel tickets also. If a programmer is to develop code for this, he may visualize that booking tickets and canceling them are reverse procedures. Hence, he will write these 2 tasks as separate and opposite tasks, and assign 2 different threads to do these tasks simultaneously.

To book a ticket, the thread will enter the train object to verify that the ticket is available or not. When there is a ticket, it updates the available number of tickets in the train object. For this, it takes, say 150 milliseconds. Then it enters the compartment object. In compartment object, it should allot the ticket for the passenger and update its status to 'reserved'. This means the thread should go through both the train and compartment objects.

Similarly, let us think if a thread has to cancel a ticket, it will first enter compartment object, and updates the status of the ticket as 'available'. For this it is taking, say 200 milliseconds. Then it enters train object and updates the available number of tickets there. So, this thread also should go through both the compartment and train objects.

When the `BookTicket` thread is at train object for 150 milliseconds, the `CancelTicket` thread will be at compartment object for 200 milliseconds. Because we are using multiple (more one) threads, we should synchronize them. So, the threads will lock those objects. When 150 milliseconds time is over, `BookTicket` thread tries to come out of train object and wants to lock on compartment object, by entering it. At that time, it will find that the compartment object is already locked by another thread (`CancelTicket`) and hence it will wait. `BookTicket` thread will wait for compartment object for another 50 milliseconds.

After 200 milliseconds time is up, the `CancelTicket` thread which is in compartment object completes its execution and wants to enter and lock on train object. But it will find that the train

object is already under lock by `BookTicket` thread and hence is not available. Now, `CancelTicket` will wait for the train object which should be unlocked by `BookTicket`.

In this way, `BookTicket` thread keeps on waiting for the `CancelTicket` thread to unlock the compartment object and the `CancelTicket` thread keeps on waiting for the `BookTicket` to unlock the train object. Each thread is expecting the other thread to release the object first, and then only it is willing to release its own object. Both the threads will wait forever in this way, suspending any further execution. This situation is called 'Thread deadlock'. See Figure 26.5.

Important Interview Question

What is Thread deadlock?

When a thread has locked an object and waiting for another object to be released by another thread, and the other thread is also waiting for the first thread to release the first object, both the threads will continue waiting forever. This is called 'Thread deadlock'.

When Thread deadlock occurs, any further execution is stopped and the program will come to a halt. Thread deadlock is a drawback in a program. The programmer should take care to avoid any such deadlocks in his programs.

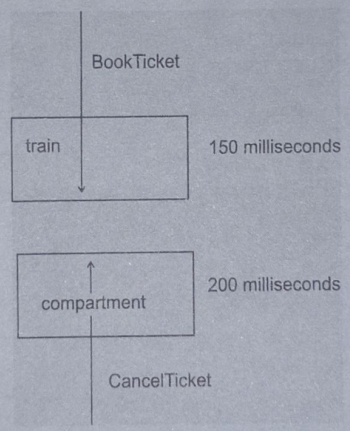


Figure 26.5 Thread deadlock

Program 8: Write a program depicting a situation in which a deadlock can occur.

```
// Thread Deadlock
class BookTicket extends Thread
{
    //we are assuming train, compartment as objects
    Object train,comp;

    BookTicket(Object train, Object comp)
    {
        this.train = train;
        this.comp = comp;
    }

    public void run()
    {
        //lock on train
        synchronized(train)
        {
            System.out.println("BookTicket locked on train");
            try{
            Thread.sleep(150);
            }catch(InterruptedException e ){}
```

```
                        System.out.println("BookTicket  now waiting to lock on
                         compartment…");

                        synchronized(comp)
                        {
                                System.out.println("BookTicket locked on
                                 compartment");
                        }
                }
        }


        class CancelTicket extends Thread
        {
                //we are assuming train, compartment as objects
                Object train,comp;

                CancelTicket(Object train, Object comp)
                {
                        this.train = train;
                        this.comp = comp;
                }

        public void run()
        {
                //lock on compartment
                synchronized(comp)
                {
                        System.out.println("CancelTicket locked on compartment");
                        try{
                        Thread.sleep(200);
                        }catch(InterruptedException e ){}

                        System.out.println("CancelTicket  now waiting to lock on
                         train…");
                        synchronized(train)
                        {
                                System.out.println("CancelTicket locked on train");
                        }
                }
        }
}
class DeadLock
{
        public static void main(String[ ] args) throws Exception
        {
                //take train, compartment as objects of Object class
                Object train = new Object();
                Object compartment = new Object();

                //create objects to BookTicket, CancelTicket classes
                BookTicket obj1 = new BookTicket (train,compartment);
                CancelTicket obj2 = new CancelTicket (train,compartment);
                //attach 2 threads to these objects
                Thread t1= new Thread(obj1);
                Thread t2= new Thread(obj2);
                //run the threads on the objects
                t1.start();
                t2.start();
        }
}
```

Output:

```
C:\> javac DeadLock.java
C:\> java DeadLock
BookTicket locked on train
```

```
CancelTicket locked on compartment
BookTicket now waiting to lock on compartment…
CancelTicket now waiting to lock on train…
```

In this program, `BookTicket` thread and `CancelTicket` threads act in reverse direction creating a deadlock situation.

Please observe the output, the program is not terminated, `BookTicket` thread is waiting for the compartment object and `CancelTicket` thread is waiting for train object. This waiting continues forever.

Avoiding Deadlocks in a Program

There is no specific solution for the problem of deadlocks. It depends on the logic used by the programmer. The programmer should design his program in such a way, that it does not form any deadlock. For example, in the preceding program, if the programmer used the threads in such a way that the `CancelTicket` thread follows the `BookTicket`, then he could have avoided the deadlock situation.

For this, we change the code in the `CancelTicket` class, so that it also follows the same sequence like `BookTicket` thread. Then what happens? First of all, `BookTicket` will enter the train object. After it processes the object, it comes out and finds that the compartment is freely available. So it enters compartment object and finally releases both the objects and comes out. Till then, `CancelTicket` thread will wait and once `BookTicket` comes out, the `CancelTicket` thread also follows the same sequence. Hence, there will not be any deadlock. So the `run()` method inside the `CancelTicket` should be changed as shown here:

```
public void run()
{
        //first lock on train like the BookTicket does
        synchronized(train)
        {
                System.out.println("CancelTicket locked on train");
                try{
                Thread.sleep(200);
                }catch(InterruptedException e ){}

                System.out.println("CancelTicket  now waiting to lock on
                 compartment…");
                synchronized(comp)
                {
                        System.out.println("CancelTicket locked on compartment");
                }
        }
}
}
```

Now run the program again to see the output:

```
BookTicket locked on train
BookTicket now waiting to lock on compartment…
BookTicket locked on compartment
CancelTicket locked on train
CancelTicket now waiting to lock on train…
CancelTicket locked on compartment
```

Thread Communication

In some cases, two or more threads should communicate with each other. For example, a Consumer thread is waiting for a Producer to produce the data (or some goods). When the Producer thread completes production of data, then the Consumer thread should take that data and use it. Let us now see how to plan the Producer class.

In the Producer class, we take a StringBuffer object to store data; in this case, we take some numbers from 1 to 10. These numbers are added to StringBuffer object. We take another boolean variable dataprodover, and initialize it to false. The idea is to make this dataprodover true when the production of numbers is completed. Producing data is done by appending numbers to StringBuffer using a for loop. This may take some time. When appending is over, we come out of for loop and then store true into dataprodover. This is shown in the following code:

```
//go on appending data (numbers) to string buffer
        for(int i= 1; i<=10; i++)
        {
                sb.append(i + ":");
                Thread.sleep(100);
                System.out.println("appending");
        }
        //data production is over, so store true into dataprodover
        dataprodover = true;
```

When the Producer is busy producing the data, now and then the Consumer will check if dataprodover is true or not. If dataprodover is true, the Consumer takes the data from StringBuffer and uses it. If the dataprodover shows false, then Consumer will sleep for some time and then again checks the dataprodover. The way to implement this logic is:

```
//sleep for 10 milliseconds while data production is not over. This while loop
//will be broken if dataprodover is true.

while( ! prod.dataprodover)
      Thread.sleep(10);
```

In this way, the Producer and Consumer can communicate with each other. But this is not an efficient way of communication. Why? Consumer checks the dataprodover at some point of time, and finds it false. So it goes into sleep for the next 10 milliseconds. Meanwhile, the data production may be over. But Consumer comes out of sleep after 10 milliseconds and then only it can find dataprodover is true. This means that there may be a time delay of 1 to 9 milliseconds to receive the data after its actual production is completed. Please see Figure 26.6.

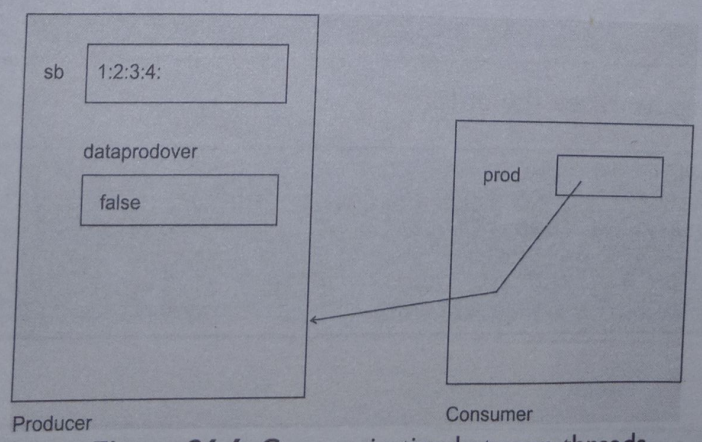


Figure 26.6 Communication between threads

Program 9: Write a program where the Consumer thread checks whether the data production is over or not every 10 milliseconds.

```
/*This program shows how two threads can communicate with each other.
      This is inefficient way of communication */
class Communicate
{
      public static void main(String[ ] args) throws Exception
      {
              //Producer produces some data which Consumer consumes
              Producer obj1 = new Producer();
              //Pass Producer object to Consumer  so that it is then available to
               Consumer
              Consumer obj2 = new Consumer(obj1);

              //create 2 threads and attach to Producer and Consumer
              Thread t1 = new Thread(obj1);
              Thread t2 = new Thread(obj2);

              //Run the threads
              t2.start();  //Consumer waits
              t1.start(); //Producer starts production
      }
}

class Producer  extends Thread
{
      //to add data, we use string buffer object
      StringBuffer sb;

      //dataprodover will be true when data production is over
      boolean dataprodover = false;

      Producer()
      {
              sb = new StringBuffer(); //allot memory
      }

      public void run()
      {
              //go on appending data (numbers) to string buffer
              for(int  i= 1; i<=10;  i++)
              {
                      try{
                      sb.append(i+":");
                      Thread.sleep(100);
                      System.out.println("appending");
                      }catch(Exception e){}
              }
              //data production is over, so store true into dataprodover
              dataprodover = true;
      }
}

class Consumer extends Thread
{
      //create Producer reference to refer to Producer object from
      //Consumer class
      Producer prod;

      Consumer(Producer prod)
      {
              this.prod = prod;
      }

      public void run()
      {
```

```
 and check
//again. Here there is a time delay of several milliseconds to
 receive data
try{
while( ! prod.dataprodover)
Thread.sleep(10);
}catch(Exception e){}

//when data production is over, display data of stringbuffer
System.out.println(prod.sb);
    }
}
```

Output:

```
C:\> javac Communicate.java
C:\> java Communicate
appending
appending
appending
appending
appending
appending
appending
appending
appending
appending
1:2:3:4:5:6:7:8:9:10:
```

How can we improve the efficiency of communication between threads? `java.lang.Object` class provides 3 methods for this purpose.

- `obj.notify()`: This method releases an object (`obj`) and sends a notification to a waiting thread that the object is available.
- `obj.notifyAll()`: This method is useful to send notification to all waiting threads at once that the object (`obj`) is available.
- `obj.wait()`: This method makes a thread wait for the object (`obj`) till it receives a notification from a `notify()` or `notifyAll()` methods.

It is recommended to use the above methods inside a synchronized block.

Let us re-write the Producer-Consumer program using the above methods so that there will not be any wastage of a single millisecond time to receive the data by the Consumer. In this program, there is no need to use `dataprodover` variable at Producer side. We can directly send a notification immediately after the data production is over, as shown here:

```
synchronized(sb)
{
    //go on appending data (numbers) to string buffer
    for(int  i= 1; i<=10;  i++)
    {
        try{
            sb.append(i+":");
            Thread.sleep(100);
            System.out.println("appending");
        }catch(Exception e){}
    }
    //data production is over, so notify to Consumer thread
    sb.notify();
}
```

Here, `sb.notify()` is sending a notification to the Consumer thread that the `StringBuffer` object `sb` is available, and it can be used now. Meanwhile, what the Consumer thread is doing? It is waiting for the notification that the `StringBuffer` object `sb` (of `Producer` class) is available. We should refer to the `sb` object at `Consumer` class as `prod.sb`.

```
synchronized(prod.sb)
{
        //wait till the sb object is released and a notification is sent
        try{
                prod.sb.wait();
        }catch(Exception e){}
}
```

Here, there is no need of using `sleep()` method to go into sleep for sometime. `wait()` method stops waiting as soon as it receives the notification. So there is no time delay to receive the data from the Producer.

Important Interview Question

What is the difference between the sleep() and wait() methods ?

Both the sleep() and wait() methods are used to suspend a thread execution for a specified time. When sleep() is executed inside a synchronized block, the object is still under lock. When wait() method is executed, it breaks the synchronized block, so that the object lock is removed and it is available.

Generally, sleep() is used for making a thread to wait for some time. But wait() is used in connection with notify() or notifyAll() methods in thread communication.

Program 10: Write a program such that the Consumer thread is informed immediately when the data production is over.

```
/* This program shows how to use wait and notify
      This is the most efficient way of thread communication */

class Communicate
{
      public static void main(String[ ] args) throws Exception
      {
              //Producer produces some data which Consumer consumes
              Producer obj1 = new Producer();
              //Pass Producer object to Consumer  so that it is then available to
               Consumer
              Consumer obj2 = new Consumer(obj1);

              //create 2 threads and attach to Producer and Consumer
              Thread t1 = new Thread(obj1);
              Thread t2 = new Thread(obj2);

              //Run the threads
              t2.start();  //Consumer waits
              t1.start(); //Producer starts production
      }
}

class Producer  extends Thread
{
      //to add data, we use string buffer object
      StringBuffer sb;

      Producer()
      {
              sb = new StringBuffer(); //allot memory
```

```
    public void run()
    {
            synchronized(sb)
            {
                    //go on appending data (numbers) to string buffer
                    for(int  i= 1; i<=10;  i++)
                    {
                            try{
                            sb.append(i+":");
                            Thread.sleep(100);
                            System.out.println("appending");
                            }catch(Exception e){}
                    }
                    //data production is over, so notify to Consumer thread
                    sb.notify();
            }
    }
}

class Consumer extends Thread
{
    //create Producer reference to refer to Producer object from
    //Consumer class
    Producer prod;

    Consumer(Producer prod)
    {
            this.prod = prod;
    }

    public void run()
    {
            synchronized(prod.sb)
            {
                    //wait till a notification is received from Producer
                     thread. Here
                    //there is no wastage of time of even a single millisecond

                    try{
                    prod.sb.wait();
                    }catch(Exception e){}

                    //when data production is over, display data of
                     stringbuffer
                    System.out.println(prod.sb);
            }
    }
}
```

Output:

```
C:\> javac Communicate.java
C:\> java Communicate
appending
appending
appending
appending
appending
appending
appending
appending
appending
appending
1:2:3:4:5:6:7:8:9:10:
```

Thread Priorities

When the threads are created and started, a 'thread scheduler' program in JVM will load them into memory and execute them. This scheduler will allot more JVM time to those threads which are having higher priorities. The priority numbers of a thread will change from 1 to 10. The minimum priority (shown by `Thread.MIN_PRIORITY`) of a thread is 1, and the maximum priority (`Thread.MAX_PRIORITIY`) is 10. The normal priority of a thread (`Thread.NORM_PRIORITY`) is 5.

Important Interview Question

What is the default priority of a thread?

When a thread is created, by default its priority will be 5.

When two tasks are assigned to two threads with different priorities, example, 2 and 5, then the thread with priority number 5 will be given more JVM time and hence it will complete the task earlier than the thread with priority number 2. See this effect in the following example to understand thread priorities. In Program 11, two threads are counting numbers from 1 to 10000. The thread with priority number 5 is completing the counting first, even if it has been started later than the thread with priority number 2.

Program 11: Write a program to understand the thread priorities. The thread with higher priority number will complete its execution first.

```
//Thread priorities
class Myclass extends Thread
{
      int count=0;   //this counts numbers

      public void run()
      {
              for(int i=1; i<=10000; i++)
              count++;   //count numbers upto 10000

              //display which thread has completed counting and its priority
              System.out.println("Completed thread: "+
               Thread.currentThread().getName());
              System.out.println("Its priority: "+
               Thread.currentThread().getPriority());

      }
}

class Prior
{
      public static void main(String args[])
      {
              Myclass obj = new Myclass();

              //create two threads
              Thread t1 = new Thread(obj, "One");
              Thread t2 = new Thread(obj, "Two");

              //set priorities for them
              t1.setPriority(2);
              t2.setPriority(Thread.NORM_PRIORITY);  //this means priority no. 5

              //start first t1 and then t2.
              t1.start();
              t2.start();
      }
}
```

```
C:\> java Prior
Completed thread: Two
Its priority: 5
Completed thread: One
Its priority: 2
```

Thread Group

A thread group represents several threads as a single group. The main advantage of taking several threads as a group is that by using a single method, we will be able to control all the threads in the group.

- To create a thread group, we should simply create an object to `ThreadGroup` class as:

```
ThreadGroup tg = new ThreadGroup("groupname");
```

Here, `tg` is the thread group object, and `groupname` is its name.

- To add a thread to this group (`tg`):

```
Thread t1 = new Thread(tg, targetobj, "threadname");
```

Here, `t1` thread is created and added to the thread group `tg`. This thread acts on `targetobj` which is the target object for the thread. The `threadname` represents the name of the thread `t1`.

- To add another thread group (`tg1`) to this group (`tg`):

```
ThreadGroup tg1 = new ThreadGroup(tg, "groupname");
```

Here we are creating and adding the thread group `tg1` to the thread group `tg`. The name of the added thread group is represented by `groupname`.

- To know the parent of a thread or a thread group, we can use `getParent()`.

```
tg.getParent();
```

This method returns `ThreadGroup` object which is the parent of `tg`.

- To know the parent thread group of a thread, we can use:

```
t.getThreadGroup();
```

This returns a `ThreadGroup` object to which the thread `t` belongs.

- To know the number of threads actively running in a thread group:

```
tg.activeCount();
```

This method returns an integer number that gives the number of threads in `tg` which are currently running in memory.

- To change the maximum priority of a thread group tg:

```
tg.setMaxPriority();
```

Normally, the maximum priority of a thread group will be 10. But this method can set it as any other number between 1 and 10.

In the following program, we are taking a thread group `tg` and in that we are adding two threads `t1` and `t2`. Then we are creating another thread group `tg1`, and adding it to `tg`. The threads `t3` and `t4` are added to the thread group `tg1`. We also used some methods which act on thread groups.

Program 12: Write a program to demonstrate the creation of thread groups and some methods which act on thread groups.

```
// Using thread groups
class TGroups
{
      public static void main(String[ ] args) throws Exception
      {
              //We should understand that the following statements are
               executed by
              //the main thread.

              Reservation res = new Reservation();
              Cancellation can = new Cancellation();

              //create a Thread group with name
              ThreadGroup tg = new ThreadGroup("First Group");

              //create 2 threads and add them to First Group
              Thread t1 = new Thread(tg, res, "First thread");
              Thread t2 = new Thread(tg, res, "Second thread");

              //create another thread group tg1 as a child to tg
              ThreadGroup tg1 = new ThreadGroup(tg, "Second Group");

              //create 2 threads and add them to Second group
              Thread t3 = new Thread(tg1, can, "Third thread");
              Thread t4 = new Thread(tg1, can, "Fourth thread");

              //find parent group of tg1
              System.out.println("Parent of tg1= "+tg1.getParent());

              //set maximum priority to tg1 as 7
              tg1.setMaxPriority(7);

              //know the thread group of t1 and t3
              System.out.println("Thread group of t1= "+ t1.getThreadGroup());
              System.out.println("Thread group of t3= "+ t3.getThreadGroup());

              //start the threads
              t1.start();
              t2.start();
              t3.start();
              t4.start();

              //find how many threads are actively running
              System.out.println("No of threads active in tg =
              "+tg.activeCount());
      }
}
class Reservation extends Thread
{
      public void run()
      {
              System.out.println("I am reservation thread");
      }
}
class Cancellation extends Thread
```

```
C:\> java TGroups
Parent of tg1= java.lang.ThreadGroup[name=First Group,maxpri=10]
Thread group of t1= java.lang.ThreadGroup[name=First Group,maxpri=10]
Thread group of t3= java.lang.ThreadGroup[name=Second Group,maxpri=7]
No of threads active in tg= 4
I am reservation thread
I am reservation thread
I am cancellation thread
I am cancellation thread
```

Daemon Threads

Sometimes, a thread has to continuously execute without any interruption to provide services to other threads. Such threads are called daemon threads. For example, `oracle.exe` is a program (a thread) that continuously executes in a computer. When the system is switched on, it also starts running and will terminate only when the system is off. Any other threads like SQL+ can communicate with it to store or retrieve data.

Important Interview Question

What is a daemon thread?

A daemon thread is a thread that executes continuously. Daemon threads are service providers for other threads or objects. It generally provides a background processing.

- To make a thread `t` as a daemon thread, we can use `setDaemon()` method as:

```
t. setDaemon(true);
```

- To know if a thread is daemon or not, `isDaemon()` is useful.

```
boolean x = t.isDaemon();
```

If `isDaemon()` returns `true`, then the thread `t` is a daemon thread, otherwise not.

Applications of Threads

In a network, a server has to render its services to several clients at a time. So, by using threads server side programs, we can make the threads serve several clients at a time. In the follow program, we are creating a server using 2 threads that can send messages to two clients at a ti First thread will contact first client, at the same time second thread will contact the second cli If third client comes, then again first thread will provide service to it. Like this, 2 threads contact the clients one by one, thus they can serve several clients. Here, we are combining `java.net` features with threads.

Program 13: Write a program to create a server with 2 threads to communicate with several clients.

```
//A server with 2 threads to contact multiple clients
import java.io.*;
import java.net.*;
class MultiServe implements Runnable
{
      static ServerSocket ss;
      static Socket s;

      public void run()
      {
            //find thread name
            String name= Thread.currentThread().getName();
            for(;;)  //server runs continuously
            {
                  try{
                  System.out.println("Thread "+name+" ready to accept...");
                  s = ss.accept();
                  System.out.println("Thread "+name+" accepted a
                   connection");
                  //for sending message
                  PrintStream ps = new PrintStream(s.getOutputStream());
                  ps.println("Thread "+name+" contacted you");

                  //close connection
                  ps.close();
                  s.close();
                  //do not close ServerSocket.
            }
            catch(Exception e) {}
      }
}

public static void main(String args[ ]) throws Exception
      {
            MultiServe ms = new MultiServe();

            //create server socket with 999 as port number
            ss = new ServerSocket(999);

            //create 2 threads
            Thread t1 = new Thread(ms,"One");
            Thread t2 = new Thread(ms,"Two");

            //start the threads
            t1.start();
            t2.start();

      }
}
```

Output:

```
C:\> javac MultiServe.java
C:\>
-------------------------------------------------------
```

Program 14: Write a program to create a client that receives message from the server.

```
//A Client that receives the messages from the server above
import java.io.*;
import java.net.*;
class MultiClient
{
```

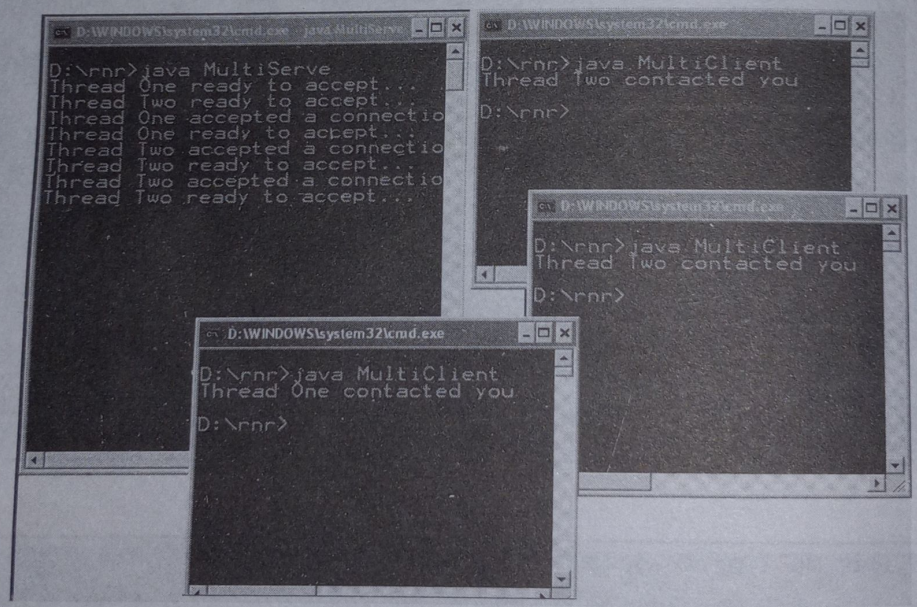
```
        public static void main(String args[ ]) throws Exception
        {
                //create Client socket with port number 999
                Socket s = new Socket("localhost", 999);

                //to accept data from server
                BufferedReader br = new BufferedReader(new  InputStreamReader
                (   s.getInputStream()));

                //receive data as long as server does not close client socket
                String str;
                while((str = br.readLine()) != null)
                System.out.println(str);

                //close connection
                br.close();
                s.close();
        }
    }
```

Output:

```
C:\> javac MultiClient.java
C:\>
```

Now run `MultiServe` server in a DOS window. Run several copies of `MultiClient` in several DOS windows. You can see the server communicating with all the clients at a time by sending a message to them.

See the Screen shot of the output of the preceding program.



Another application of threads is animation, where an object, an image or some text is moved from one place to another on the screen. Here is an example program to see a moving banner using a thread. The logic is to extract the first character from a banner string and add it at the end of the string. When the screen contents are refreshed, it appears as if the entire string will move towards left. Now extract the leftmost character and place it at the end of the banner string. Like this, repeat the same to get the effect of a moving banner.

```
import java.awt.*;  //for GUI

class Banner extends Frame implements Runnable
{
      //this is the banner string
      String str= " DREAM TECH PUBLICATIONS ";

      Banner()
      {
              setLayout(null); //dont set Layout manager
              setBackground(Color.cyan);
              setForeground(Color.red);

      }//end of constructor

      public void paint(Graphics g)
      {
              //set a font and display the banner string
              Font f = new Font("Courier", Font.BOLD, 40);
              g.setFont(f);
              g.drawString(str, 10,100);
      }

      Public void run()
      {

              for(;;) {       //move banner continuously
              repaint();  //refresh the frame contents
              try{
              Thread.sleep(400);  //give a gap of 400 millis between each
               movement
              }catch(InterruptedException ie){}
              char ch = str.charAt(0);  //extract first char from string
              str = str.substring(1,str.length());  //add to str from second
               char till end
              str = str+ch; //attach first char at the end of str
              }
      }

      public static void main(String args[ ])
      {
              Banner b = new Banner();  //b represents the frame
              b.setSize(400,400);
              b.setTitle("My banner");
              b.setVisible(true);
              //create a thread and run it
              Thread t = new Thread(b);
              t.start();
      }
}
```

Output:

```
C:\> javac Banner.java
C:\> java Banner
```

```
S   DREAMTECH PUBLICATION
```

Thread Life Cycle

Starting from the birth of a thread, till its death, a thread exists in different states which are collectively called 'Thread Life cycle'. Please see Figure 26.7.

A thread will be born when it is created using `Thread` class as:

```
Thread t = new Thread();
```

Then the thread goes into runnable state when `start()` method is applied on it. `Yield()` method may pause a thread briefly, but the thread will be still in runnable state only.

From runnable state, a thread may get into not-runnable state, when `sleep()` or `wait()` methods act on it. A thread may be occasionally blocked on some Input-Output device where it is expecting some input-output from the user. The thread would be in not-runnable state till the user provides the required input or output. After coming out from not-runnable state, again the thread comes back to runnable state.

Finally, a thread is terminated from memory only when it comes out of `run()` method. This happens when the thread completely executes the `run()` method and naturally comes out, or when the user forces it to come out of `run()` method.

All these state transitions of a thread, starting from its birth till its death are called 'thread life cycle'.

Important Interview Question

What is thread life cycle?

A thread is created using new Thread() statement and is executed by start() method. The thread enters 'runnable' state and when sleep() or wait() methods are used or when the thread is blocked on I/O, it then goes into 'not runnable' state. From 'not runnable' state, the thread comes back to the 'runnable' state and continues running the statements. The thread dies when it comes out of run() method. These state transitions of a thread are called 'life cycle of a thread'.

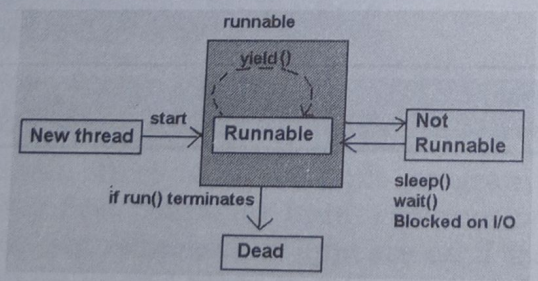


Figure 26.7 Life cycle of a thread

Conclusion

In most of the applications, different segments (parts) of the application run independently performing their duties almost at a time and thus improving performance. In such a case, we need threads to run those segments individually and at the same time. For example, a server on a network or Internet provides its services to several clients at a time. In such cases, we can use threads at the server so that a group of threads will be able to serve hundreds of clients at a time.

Threads are also useful where a process should run continuously. For example, a continuously running animation or continuously running server application needs a daemon thread which keeps it running for ever. Being light weight, threads use minimal system resources and hence they are highly preferred to actual heavy weight processes.