Chapter - 9

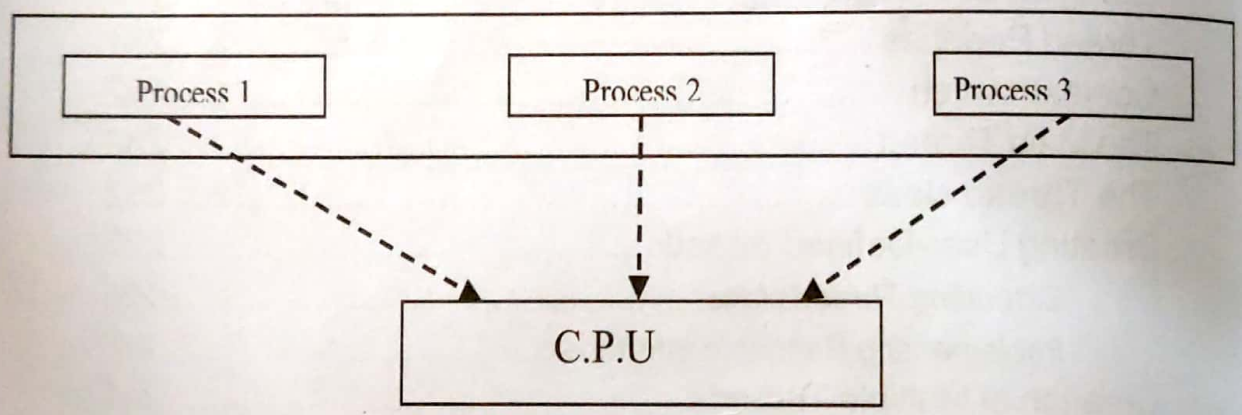
Multithreading

Multithreading

Introduction

We have different types of operating systems out of which some supports single tasking and most supports multitasking. In single tasking operating systems at any given time only one program or process (A program that is executing) executes i.e. one program after the other executes. When one program is executing other programs can not be executed. For example when a file is sent to the printer for printing we can not create or execute the programs. In this case while the printer is printing the file the CPU sits idle which is wastage of CPU time. In order to utilize the CPU time efficiently the operating systems are designed to support multitasking where at any given time two or more programs or processes can be executed. This multitasking is of two distinct types: ***process-based*** multitasking and ***thread-based*** multitasking.

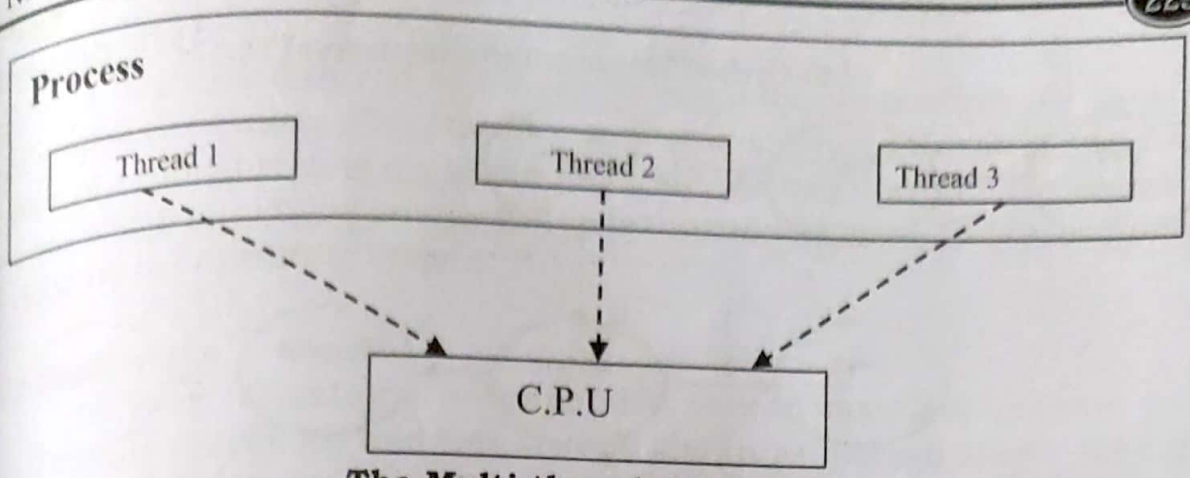
The ***process-based*** multitasking is the feature that allows the computer to run two or more programs concurrently. For example, in process-based multitasking we can send a file to the printer and at the same time we can create and execute java programs. In this case the CPU time is utilized efficiently.



The Multi-tasking process

Process

In some situations we want different parts of the same program execute simultaneously and it can be implemented using ***thread-based*** multitasking. For example, a text editor can format text at the same time that it is printing, as long these two actions are being performed by two separate threads. Using threads CPU idle time can be utilized efficiently i.e. multithreading enables to write very efficient programs that make maximum use of the CPU time, because idle time can be kept to a minimum.



The Multi-threaded Process

Unlike many other programming languages, Java provides built-in support for ***multithreaded programming***. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a *thread,* and each thread defines a separate path of execution. Thus, multithreading is a specialized form of multitasking.

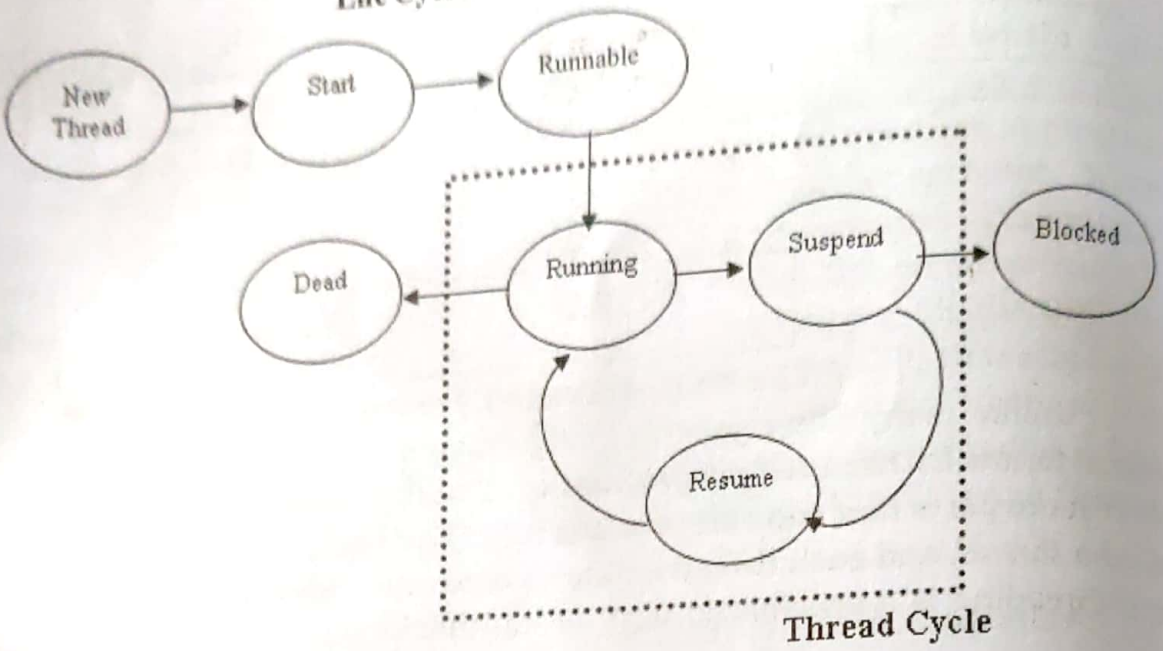
Multitasking threads requires less overhead than multitasking processes. Processes are heavyweight tasks that require their own separate address spaces. Interprocess communication is expensive and limited. Context switching from one process to another is also costly. Where as threads are lightweight and they share the same address space and cooperatively share the same heavyweight process. Interthread communication is inexpensive, and context switching from one thread to the next is low cost. The Java programs make use of process-based multitasking environments, process-based multitasking is not under the control of Java where as multithreaded multitasking is under the control of Java.

Life Cycle of Thread or Differrent states of Thread

When a thread is created, it goes into different states before it completes its task and it is called as life cycle. The different states are:

1. New
2. Runnable
3. Running
4. Suspend
5. Resume
6. Blocked
7. Dead

Life Cycle of Thread or Different stages of Thread



New state

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

Runnable state

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

Running state

The thread currently being executed by the CPU is in a running state. In this state the thread actually performs the task that is assigned to it.

Suspend state

A running thread may stop its execution temporarily for some time and therefore it is in suspend state. A thread may suspend because of the following conditions.

sleep() method is called by the thread

the thread performs I/O operation

wait() method is called by the thread

Resume state

A suspended thread continues its execution from where the thread was stopped execution. This state is resume state. When a thread is resumed it goes to a runnable state not to running state. Among the runnable threads, the one which has higher priority will go to the running state. This scheduling is done by the operating system.

Blocked state

In some situations a suspended thread may not resume i.e. permanently suspended and this state is known as blocked state. We can not do any thing in this state except to stop the program.

Dead state

A thread becomes dead(termination of thread) 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.

In the above states new, blocked, dead states are executes only once where as other states runnable, running, suspend, resume executes continuously until the thread is successfully terminated or blocked.

Thread Priorities

Java assigns each thread with a number which is known as **Priority Number**. This number exist between **1** to **10**. The priority number of a thread determines how a thread should be treated with request to other threads. The execution speed of the threads does not depend on the priority numbers, but a thread with higher priority number executes more time in the CPU than lower priority number thread. When a thread is created, a default priority number 5 is given. This priority number can be changed by the programmer according to his requirements.

For example, if CPU contains one second(**1000 milli-seconds**) of idle time, this time is divided into parts called as time slices, let say **1000 milli-seconds** are divided into **100** parts (may not be equal parts). If two threads(**thread1**,**thread2**) are executing in the CPU with different priority numbers(**3**,**8**) then **CPU** allots more time slices to **thead2** to execute than the **thread1**. In this way a higher priority thread execute more time than lower priority thread. The thread's priority number decides when to switch from one running thread to the next. This is known as ***context switch.***

Context Switch

Context switch is the concept of switching from one running thread to other by the CPU. Context switch may takes place because of any one of the following reasons.

1. ***A thread can voluntarily relinquish control:*** If a running thread gets into yielding, sleeping, or blocking on pending I/O than the CPU examines all other threads and the higher priority thread that is ready to run is given the CPU.

2. ***A thread can be preempted by a higher-priority thread:*** In this case when a lower-priority thread is executing and it is not yielding the processor and at the same time if higher-priority thread want to execute therefore the higher-priority thread makes the lower-priority thread to suspend and the higher-priority thread executes in CPU. This is called as preemptive multitasking.

In case, if two or more threads have same priority numbers than the OS uses different techniques such as round-robin, **first-in first-out** etc, to allot the CPU cycles.

The Main Thread

When a java program starts its execution, the jvm creates a thread which is called *main thread*. This is the first thread that is executed and all other child-threads start from it. We should see that the main thread is terminated after child threads are terminated because main thread performs various shutdown actions.

The Thread Class

The **Thread** class is a predefined class present in **java.lang** package. This class contains various methods and properties that encapsulates the mechanism of multithreading. Using these methods we can write the multithreaded programs in Java. The methods in the **Thread** class are:

Methods	Meaning
public String getName()	Returns thread's name
public void setName(String tame)	Sets thread's name
public int getPriority()	Returns thread's priority number
public void setPriority(int n)	Sets thread's priority number

public static Thread currentThread()	Returns reference of running thread
public static void sleep (int millisec)	Suspends thread for given time
public boolean isAlive()	Returns true if thread is not terminated
public void join()	Waits for the thread to terminate
public void start()	Start a thread by calling its run()
public void run()	Entry point for the thread. It contains thread code.

Constructors of **Thread** class are :

Constructors	Meaning
Thread(String tname)	Constructs the thread with tname (used with, when user-defined thread class extends Thread class)
Thread(Runnable ob, String tname)	Constructs ob as thread with tame (used with, when user-defined thread class implements from Runnable interface)

A program to demonstrate the main thread

Bin>edit mainthread.java

```
class mainthread
{
   public static void main(String  argv[ ])
   {
         System.out.print("\n Main thread started..");
         Thread t=Thread.currentThread();
         System.out.print("\n Main thread details : " + t);
         System.out.print("\n Thread name:"+t.getName());
         System.out.print("\n Priority number:"+
                              t.getPriority());

         t.setName("Apex");
         t.setPriority(7);
         System.out.print("\n Thread details after change:" +t);
         System.out.print("\n Thread name:"+t.getName());
         System.out.print("\n Priority number : "+
                              t.getPriority());

         try
         {
```

```
            for(int  i=1 ; i<=5 ; i++)
            {
              System.out.print("\n Main Thread : " + i );
              Thread.sleep(1000);
            }
        }
        catch(InterruptedException e)
        {
            System.out.print("Error : " + e );
        }
        System.out.print("\n Main terminates.");
    }
}
```

Bin> javac mainthread.java
Bin> java mainthread

```
Main thread started..
Main thread details : Thread [main, 5, main]
Thread name : main
Priority number : 5
Thread details after change : [Apex, 7, main]
Thread name : Apex
Priority number : 7
Main Thread : 1
Main Thread : 2
Main Thread : 3
Main Thread : 4
Main Thread : 5
Main terminates.
```

Explanation:

When the above program is executed, main() method is started and the jvm creates the main-thread. The first statement prints the message **Main thread started..** at the statement **Thread t=Thread.currentThread();** the currentThread() which is public static method of Thread class returns the reference of main-thread object that is executing and assign to **t**. At the statement **System.out.print("\n Main thread details : " + t);** the **t** object prints the details of main-thread as

Main thread details : Thread [main, 5, main]

Where first **main** is thread name and second **main** is the parent thread name, as the main does not have parent therefore parent is also displayed as main. Where **5** is priority number of main-thread, which is default priority number.

The methods **t.getName()** and **t.getPriority()** returns the name(**main**) and priority number(**5**) of the main-thread and are printed as

Main thread details : Thread [main, 5, main]
Thread name : main
Priority number : 5

The methods **t.setName("Apex")** and **t.setPriority(7)** sets main-thread name as **"Apex"** and priority number as **7** and the next print statements prints the same.

The for loop prints the numbers **1**, **2**, **3**, **4**, **5** for very one second, because the **Thread.sleep(1000);** statement suspends the thread for one second. While the thread is executing, because of any reasons if the running thread gets interrupted then the jvm throws an **InterruptedException** and therefore it is handled by the catch block.

Finally, the last print statement prints the message **Main terminates** and the program terminates.

Creating User-Defined threads

User can also define thread classes in java either extending Thread class or implementing Runnable interface.

1. Extending Thread class

The Thread class of **java.lang** package contains multithreaded features therefore any user-defined thread class should inherit from **Thread** class. Any class that extends from Thread class can behave like a threaded class. The user-defined thread class should override the **run()** method because it is the actual method that contains thread code.

Syntax:

```
class <userthreadclass> extends Thread
{
        :
        :
}
```

2. Implementing Runnable interface

A user-defined thread class can also be defined implementing from **Runnable** interface. This interface contains only one abstract method i.e. **run()**, therefore any class that implements **Runnable** interface must override this method otherwise the compiler generates error. However the user-defined class must inherit from the **Thread** class. When a thread class implements from **Runnable** interface we have inherit the **Thread** class using indirect inheritance i.e. **deligation**.

Syntax:

```
class <userthreadclass> implements Runnable
{
        Thread t;
        :
        :
}
```

This method of creating thread class is useful when we want the sub-class to make as a thread class. For example, consider the following

```
class <userthreadclass> extends super-class implements Runnable
{
        Thread t;
        :
        :
}
```

But the following is wrong because Java does not support multiple inheritance.

```
class <userthreadclass> extends super-class extends Thread
{
        :
        :
}
```

A program demonstrates creation of user-defined thread class extending Thread class

Bin>edit thread1.java

```
class userthread extends Thread
{
   public userthread(String tname)
   {
         super(tname);
         System.out.print("\n child thread created:" + this);
         start();
   }
   public void run()
   {
         System.out.print("\n child thread started..");
         try
         {
               for(int i=1;i<=5;i++)
               {
                     System.out.print("\n Child :" + i );
                     Thread.sleep(1000);
               }
         }
         catch(InterruptedException e)
         {
               System.out.print("\n Error:"+e);
         }
         System.out.print("\n Child thread terminates..");
   }
}
class thread1
{
   public static void main(String argv[])
   {
         System.out.print("\n Main Thread started");
         userthread obj=new userthread("apex");
         try
         {
               for(int i=1; i<=5; i++)
               {
                     System.out.print("\n Main : " + i );
                     Thread.sleep(2000);
               }
         }
```

```
            catch(InterruptedException e)
            {
                System.out.print("\n Error : " + e );
            }
            System.out.print("\n Main thread terminates..");
        }
    }
```

Bin>javac thread1.java
Bin> java thread1
Main thread started
child thread created:Thread[apex, 5, main]
Main :1
child thread started..
Child :1
Child :2
Main :2
Child :3
Child :4
Main :3
Child :5
Child thread terminates..
Main :4
Main :5
Main thread terminates..

Explanation:

When the above program is executed, the main thread is created by the jvm and main() executes. The first statement in the main prints the message **Main thread started...** The statement **userthread obj=userthread("apex");** creates an object **obj** of **userthread** class and executes the constructor where "**apex**" is passed to **tname.** In the constructor **super(tname);** calls the constructor of superclass **Thread** which actually constructs the **obj** into thread object. In the statement **System.out.print("\n child thread created:" + this); this** reference variable represents the **obj** which print the object details i.e. thread details as **child thread created:Thread[apex, 5, main]** where **apex** is the thread name and **main** is the parent thread name and **5** is priority number. The **start()**; makes the thread as runnable thread which joins the collection of threads that are ready for execution. When this runnable thread comes into execution it calls the **run()** method. Now two threads in the program execution, one is

main thread and other is child(**Apex**) thread. These two threads executes concurrently, the main thread suspends two seconds for each cycle where as child thread suspends one second for each cycle and we get the output as shown above. When child thread for loop terminates it comes out of try and prints a message **child thread terminates..** and run method terminates as a result the thread dies(terminates). There after the main thread executes for remaining cycles and finally it also dies.

Note: In multithreading the output may not be same for every time. It may changes from execution to execution that depends on processor speed or operating system etc..

A program demonstrates creation of user-defined thread class implementing Runnable interface.

Bin>edit thread2.java

```
class userthread implements Runnable
{
   Thread t;
   public userthread(String tname)
   {
         t=new Thread(this,tname);
         System.out.print("\n child thread created:" + t );
         t.start();
   }
   public void run()
   {
         System.out.print("\n child thread started..");
         try
         {
               for(int i=1;i<=5;i++)
               {
                     System.out.print("\n Child :" + i );
                     Thread.sleep(1000);
               }
         }
         catch(InterruptedException e)
         {
               System.out.print("\n Error:"+e);
         }
         System.out.print("\n Child thread terminates..");
   }
```

```
class thread2
{
      public static void main(String argv[])
      {
            System.out.print("\n Main Thread started");
            userthread obj=new userthread("apex");
            try
            {
                  for(int i=1; i<=5; i++)
                  {
                        System.out.print("\n Main:"+i);
                        Thread.sleep(2000);
                  }
            }
            catch(InterruptedException e)
            {
                  System.out.print("\n Error : " + e );
            }
            System.out.print("\nMainthread terminates..");
      }
}
```

Bin>javac thread2.java
Bin> java thread2

Explanation:

In the above program, the **userthread** class implements from **Runnable** interface. In this case the **userthread** class have to implement indirect inheritance (**object deligation**) from **Thread** class and it does by declaring **Thread** class reference variable **t**. The statement **userthread obj=userthread("apex");** creates an object **obj** of **userthread** class and executes the constructor where **"apex"** is passed to **tname.** In the constructor the statement **t=new Thread(this,tame);** creates an object of **Thread** class as a result it calls the constructor of Thread class and passes the reference of **obj**(this), **"apex"**(tname) which actually constructs the **obj** into thread object. In the statement **System.out.print("\n child thread created:" + t);** t reference variable represents the thread details as **child thread created:Thread[apex, 5, main]** where **apex** is the thread name and **main** is the parent thread name and **5** is priority number. The **t.start()**; makes the thread as runnable thread which joins the collection of threads that are ready for execution. When this runnable thread comes into execution it

calls the **run()** method. The remaining execution and output is similar to the previous program.

Creating Multiple Threads

Up to now we have created and executed only two threads i.e. one is main and other is child thread. It is possible to create any number of child threads and can be executed in a single program. The following is the program that demonstrates multiple threads.

Bin>edit mthreads.java

```
class mythread implements Runnable
{
   Thread t;
   String threadname;
   public mythread(String tname)
   {
         t=new Thread(this,tname);
         threadname=tname;
         System.out.print("\n Thread created:"+threadname );
         t.start();
   }
   public void run()
   {
         System.out.print("\n Thread started :" + threadname);
         try
         {
               for(int i=1;i<=5;i++)
               {
                  System.out.print("\n"+threadname +":"+i);
                  Thread.sleep(1000);
               }
         }
         catch(InterruptedException e)
         {
               System.out.print("\n Error:"+e);
         }
         System.out.print("\n Thread terminates:"+threadname);
   }
}
```

```
class mthreads
{
   public static void main(String argv[])
   {
         System.out.print ("\n Main Thread started…");
         mythread t1=new mythread("first");
         mythread t2=new mythread("second");
         mythread t3=new mythread("Third");
         try
         {
         System.out.print("\n Main thread suspending..");
         Thread.sleep(10000);
         System.out.print("\n Main thread resumed..");
         }
         catch(InterruptedException e)
         {
               System.out.print("\n Error :"+e);
         }
         System.out.print("\n Main thread terminates…");
   }
}
```

Bin>javac mthreads.java
Bin>java mthreads

```
Main Thread started...
Thread created :First
Thread created :Second
Thread created :Third
Main thread suspending..
Thread started : First
First : 1
Thread started : Second
Second :1
Thread started : Third
Third : 1
First :2
Second :2
Third :2
First :3
Second :3
Third :3
```

Thread: path followed by executing a pgm: main thread


```
First :4
Second :4
Third :4
First :5
Thread terminates :First
Second :5
Thread terminates :Second
Third :5
Thread terminates :Third
                                    // suspends main for 5 seconds

Main thread resumed..
Main thread terminates..
```

Explanation:

In the above program, three child threads(**First**, **Second**, **Third**) are created and executes. These three thread shares the **CPU** time equally and we get the output as shown above. Child threads complete their execution in **5** seconds where as main thread suspends for **10** seconds to ensure that main terminates after child threads. Therefore main thread waits for **5** seconds of time even after the child threads are terminated which is waste of time.

join() and isAlive() methods

In some situations, it may be required to wait for a particular thread to complete its task before another thread to proceed with. In this case, 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 and then the control proceeds with the normal course of execution. The general syntax of **join()** method is..

```
public void join()
```

The **isAlive()** method of Thread class returns **true** if the thread is not terminated and returns false if the thread is terminated. The syntax of **isAlive()** is.

```
public boolean isAlive()
```

Bin>edit mthreads1.java

```
class userthread implements Runnable
{
    Thread t;
    String threadname;
    public userthread(String tname)
    {
          t=new Thread(this,tname);
          threadname=tname;
          System.out.print("\n Thread created:"+threadname );
          t.start();
    }
    public void run()
    {
          System.out.print("\n Thread started :" + threadname);
          try
          {
                for(int i=1;i<=5;i++)
                {
                   System.out.print("\n"+threadname+":"+i);
                   Thread.sleep(1000);
                }
          }
          catch(InterruptedException e)
          {
                System.out.print("\n Error:"+e);
          }
          System.out.print("\n Thread terminates:"+threadname);
    }
}
class mthreads1
{
    public static void main(String argv[])
    {
          System.out.print("\n Main Thread is started.....");
          userthread t1=new userthread("First");
          userthread t2=new userthread("Second");
          userthread t3=new userthread("Third");
          System.out.print("\n t1 isAlive:"+t1.t.isAlive());
          System.out.print("\n t2 isAlive:"+t2.t.isAlive());
          System.out.print("\n t3 isAlive:"+t3.t.isAlive());
```

```
            try
            {
              System.out.print("\n Main thread suspending..");
              t1.t.join();
              t2.t.join();
              t3.t.join();
              System.out.print("\n Main thread resumed..");
            }
            catch(InterruptedException e)
            {
              System.out.print("\nError:"+e);
            }
            System.out.print("\n t1 isAlive:"+t1.t.isAlive());
            System.out.print("\n t2 isAlive:"+t2.t.isAlive());
            System.out.print("\n t3 isAlive:"+t3.t.isAlive());
            System.out.print("\n Main Thread terminates..");
    }
}
```

Bin>javac mthreads1.java
Bin>java mthreads1

```
Main Thread started...
Thread created :First
Thread created :Second
Thread created :Third
t1 isAlive:true
t2 isAlive:true
t3 isAlive:true
Main thread suspending..
Thread started : First
First : 1
Thread started : Second
Second :1
Thread started : Third
Third : 1
First :2
Second :2
Third :2
First :3
Second :3
Third :3
```

```
First :4
Second :4
Third :4
First :5
Second :5
Third :5
Thread terminates :First
Thread terminates :Second
Thread terminates :Third
Main thread resumed..
t1 isAlive:false
t2 isAlive:false
t3 isAlive:false
// main resumes immediately after termination of child
//threads and no waste of CPU time
Main thread terminates..
```

Explanation:

The above program is similar to its previous program and we get the output as shown above. The **isAlive()** methods before try returns **true,** because the three child threads are started but not terminated. In the try block, each **join()** method call suspends the main thread until the threads are terminated. Immediately after termination of child threads the main resumes. The **isAlive()** methods after try block returns **false** because after **join()** methods three threads terminates. Immediately the main thread terminates as a result no wastage of time.

Thread Priorities

Not all threads are created equal. Sometimes we want to give one thread more time than another. Threads that interact with the user should get very high priorities. Threads that calculate in the background should get low priorities.

Each thread can be assigned priority number. Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run i.e. when to give the CP time. A thread that has higher-priority number is executed more time in the CPU as compared to lower-priority thread. The priority numbers ranges from **1** to **10**. Priority numbers can be set to the threads using **setPriority()** method of Thread class and its signature is..

```
public void setPriority(int priorityno);
```

The Thread class contains public static final variables which are set with some priority numbers and they are,

```
MIN_PRIORITY =1      // minimum priority number of thread
MAX_PRIORITY=10      // maximum priority number of thread
NORM_PRIORITY=5      //default priority number of thread
```

The **getPriority()** method of Thread class returns the priority number of a thread and its signature is

public int getPriority();

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.

The following is the program to demonstrate thread priorities

Bin>edit tprior1.java

```
class userthread implements Runnable
{
   Thread t;
   long count=0;
   volatile boolean flag=false;
   public userthread(int pno)
   {
         t=new Thread(this, "racing");
         t.setPriority(pno);
         flag=true;
   }
   public void begin()
   {
         t.start();
   }
   public void stop()
   {
         flag=false;
   }
   public void run()
   {
         while(flag)
         {
```

```
                count++;
            }
        }
    }
class tprior1
{
    public static void main(String argv[])
    {
    userthread t1=new userthread(3);
    userthread t2=new userthread(7);
    System.out.print("\n t1 and t2 threads are starting.....");
    t1.begin();
    t2.begin();
    try
    {
    System.out.print("\n Main Thread suspending for 5 sec");
    Thread.sleep(5000);
    }
        catch(InterruptedException e)
        {
            System.out.print("\nError:"+e);
        }
        t1.stop();
        t2.stop();
        try
        {
            t1.t.join();
            t2.t.join();
        }
        catch(InterruptedException e1)
        {
            System.out.print("\n Error :"+e1);
        }
        System.out.print("\n Main thread resumed..");
        System.out.print("\n count of t1 ="+t1.count);
        System.out.print("\n count of t2 ="+t2.count);
    }
}
```

Bin>javac tprior1.java
Bin>java tprior

```
t1 and t2 threads are starting.....
Main Thread suspending for 5 sec
Main thread resumed..
count of t1 = 43063383
count of t2 =1963588708
```

Explanation:

In the above program, **t1** thread is created with priority number **3** and **t2** with **7**. The statement **t1.begin();** calls the **begin()** method which starts the threads i.e. enters into runnable state. Similarly, **t2** thread also starts its execution. The main thread is suspending for **5000** milli-seconds and in this time the two threads executes in CPU according to its priorities. Which ever the thread that gets the CPU time executes where whose count is continuously incremented as shown in **run()**. After **5** seconds, the main thread resumes and the statement **t1.stop()** makes **flag** of **t1** thread to **false** as a result the while loop in the run() method terminates and the thread dies. Similarly, with **t2** thread. In these **5** seconds of time, **t2** thread executes more time in the CPU then **t1,** because **t2** priority is higher than **t1**. Therefore **t2.count** value is higher than **t1.count** as shown in the above output.

In the main, even after stopping the **t1** and **t2** threads, we are calling **join()** methods on threads because we want to ensure that threads terminate completely before the main thread continue its execution. This is because when **t1** calls **stop()** method which makes **flag** to **false** and it should reflect in while loop to terminate, all this takes some time and similarly with **t2** thread. Mean while we don't want the main thread to continue its execution, that is why we are suspending the main thread till **t1** and **t2** threads terminate completely and proceed further.

Synchronization

It is possible that two or more threads can share the common resource(methods, objects) at the same time. In this situation one thread is mixed with the task of other threads and gives incorrect results. This may happen because of any one of the following situations.

Threads using the common object may have same priority and, therefore, the CPU may switch form 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.

Let us, demonstrate threads that share the common resource and are not synchronized.

In the following program, we pass different files from different threads to **print()** method for printing at the same time and the **print()** method is not synchronized. Here **print()** method is the resource shared by threads.

Bin>edit unsyn.java

```
class printer
{
   public void print(String filename)
   {
         System.out.print("\n Printing ……"+filename);
         try
         {
               Thread.sleep(2000);
         }
         catch(InterruptedException e)
         {
               System.out.print("\n Error: "+e);
         }
         System.out.print("\n Completed ……"+filename);
   }
}
class computer implements Runnable
{
   printer pobj;
   String filename;
   Thread t;
   public computer(printer  p, String fname)
   {
         pobj=p;
         filename=fname;
         t=new Thread(this,"unsyn");
         t.start();
   }
   public void run()
```

```
        {
                pobj.print(filename);
        }
}
class unsyn
{
        public static void main(String argv[])
        {
                printer prn=new printer();
                computer c1=new computer(prn,"A.java");
                computer c2=new computer(prn,"B.java");
                computer c3=new computer(prn,"C.java");
                try
                {
                        c1.t.join();
                        c2.t.join();
                        c3.t.join();
                }
                catch(InterruptedException  e)
                {
                        System.out.print("\nError:"+e);
                }
        }
}
```

Bin>javac unsyn.java
Bin>java unsyn
Printing...A.java
Printing...B.java
Printing...C.java
Completed...A.java
Completed...B.java
Completed...C.java

Explanation :

In the above program, the statement **printer prn=new printer();** creates an object to **printer** class. The statement **computer c1=new computer(prn,"A.java");** creates an object to **computer** class and invokes constructor **computer(printer p,String fname);** where **p** is passed with the reference of **prn** and **fname** with **"A.java".** In the constructor, **p** reference is assigned to **pobj, fname(A.java)** to **filename** and thread object **t** is defined and thereafter **start()** method calls the **run()** method where the statement

pobj.print(filename) calls **print(String filename)** which prints the message **Printing...A.java** and the **c1** thread suspends for **2** seconds.

The main thread resumes and the statement **computer c2=new computer(prn,"B.java");** creates **c2** object and invokes **computer(printer p,String fname)** constructor where **p** is passed with same **prn** object reference, **fname** with **B.java**. In the constructor, **p** reference assigns to **pobj**, **fname(B.java)** to **filename** and thread object **t** is defined and thereafter **start()** method calls the **run()** method where the statement **pobj.print(filename)** calls **print(String filename)** which prints the message **Printing...B.java** and **c2** thread suspends for **2** seconds.

Similarly, **c3** thread prints the message **Printing...C.java** and it also suspends for **2** seconds.

After **2** seconds, the first **thread(c1)** which was suspended resumes and prints the message **Completed...A.java** and **print()** method terminates as a result **c1** thread dies. The second **thread(c2)** resumes and prints the message **Completed...B.java** and **c2** thread also dies. Finally, third **thread(c3)** resumes and prints the message **Completed...C.java**. We get the output as shown above.

In the above program three threads are sharing the common resource (**print()**) at the same time as a result we are getting the wrong output. To overcome this problem, synchronization concept can be implemented.

Synchronization

Synchronization is the concept of allowing two or more thread share the common resource one thread after the other so that the resource is utilize properly. The synchronization is achieved through ***monitor*** (also called as ***semaphore***) concept. A **monitor** is an object that is used as a mutually exclusive *lock* or ***mutex***. At any given time only one thread can enter into monitor. When a thread enters into monitor, it is locked and other threads that come to the same monitor for accessing of resource will wait until the thread in the monitor exit from monitor and unlocks it. The first thread that comes to the monitor is given first preference and allows into monitor and locks it. Other threads will be in the waiting state until it come out of monitor. In this way the monitor allows one thread after the other to share the resource at any given time.

Synchronization can be implemented in two ways in Java

1. Method Synchronization
2. Synchronized Statement

Method Synchronization

In the method synchronization, the method should be declared with the keyword synchronized. 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.

While an object is locked, other threads cannot access only the synchronized methods, but can access the other methods. An object can have any number of synchronized methods. When such object is locked by one thread for want fo 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.

Syntax:

```
synchronized return-type methodname( [parameters-list] )
{
        :
        :
}
```

A program that demonstrates method synchronization

To clear the problem in the above program, declare the **print()** method with the **synchronized** keyword as shown below and it becomes synchronized method. Therefore **print()** method is restricted to access to only one thread at a time.

```
class printer
{
        synchronized public void print(String filename)
        {
                :
        }
}
```

The above synchronized method prevents other threads from entering **print()** while another thread is using it. After synchronized has been added to **print()**, the output of the program is as follows.

```
Printing...A.java
Completed...A.java
Printing...B.java
Completed...B.java
Printing...C.java
Completed...C.java
```

Synchronized Statement

The method synchronization can be achieved when the class and the method is written by us and it is the easiest method. But, there are situations where the class is written by third party and not declared the method as synchronized. We want to use this class by creating objects and want to synchronize access to these objects. As we are creating the objects to the class, we can synchronize the objects using the synchronized statement.

Syntax :

```
synchronized(object)
{
        ://statements to be synchronized
}
```

In the above syntax, **object** is a reference to the object being synchronized. A synchronized block at any given time allows only one thread to enter into it and access the method of the object.

A program to demonstrate synchronized statement

Re-execute the **unsyn.java** program by replacing the statement in the **run()** method as follows..

```
public void run()
{
        synchronized(pobj)
        {
                pobj.print(filename);
        }
}
```

In the above **run()** method, before the **print()** is called, the **pobj** object is synchronized. Therefore one thread after the other access the **print()** and we get the same output as synchronized methods example.

Interthread Communication

Threads are created to carry out process independently. In some situations, two or more threads may want to use common object as resource. In order to overcome 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, it will be locked in the monitor and other threads which want to use the same object has to be in waiting state. A synchronized object can have any number of synchronized methods. 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 has to keep waiting infinitely. To avoid such problem, Java provides interthread communication methods, which can send messages from one thread to another thread, which uses the same object.

The following methods are used for interthread communication.

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.

These methods **wait()**, **notify()** and **notifyAll()** should be used only in synchronized methods. These methods are defined in **Object** class of **java.lang** package, the **Object** class is the supermost class of any class(pre-defined or user-defined) in Java, therefore these methods are available to all classes in Java.

A program demonstrate without interthread communication through consumer producer problem

In this problem, the **Producer** produces products and **Consumer** consumes them. But, the rule is, consumer should consume all the products in the same order in which the producer produce the products. One more rule is, consumer should not consume a product more than once. In this program, we create a **shop** class which is shared by two threads **Producer** and **Consumer**. The producer produce the product and puts in the **shop** using **put()** method and consumer uses **get()** method to consume. But, these threads do not communicate with each other i.e. they execute independently and therefore gives incorrect output.

Bin>edit uncomm.java

```
class shop
{
    int pno=0;
    synchronized public void put(int n)
    {
          pno=n;
          System.out.print("\n Produced : "+pno);
    }
    synchronized public void get()
    {
          System.out.print("\n Consumed : "+pno);
    }
}
class producer implements Runnable
{
   Thread t;
   shop sobj;
   public producer(shop ob)
   {
         sobj=ob;
         t=new Thread(this, "producer");
         t.start();
   }
   public void run()
   {
         try
         {
               for(int i=1;i<=5;i++)
               {
```

```
                    Thread.sleep(2000);
                    sobj.put(i);
                }
            }
            catch(InterruptedException e)
            {
                System.out.print("\nError : "+ e);
            }
        }
}
class consumer implements Runnable
{
    Thread t;
    shop sobj;
    public consumer(shop ob)
    {
        sobj=ob;
        t=new Thread(this,"consumer");
        t.start();
    }
    public void run()
    {
        try
        {
            for(int i=1;i<=5;i++)
            {
                sobj.get();
                Thread.sleep(1000);
            }
        }
        catch(InterruptedException e)
        {
            System.out.print("\nError : "+ e);
        }
    }
}
class nocomm
{
    public static void main(String argv[ ])
    {
```

```
        shop sobj=new shop();
        producer p =new producer(sobj);
        consumer c=new consumer(sobj);
    }
}
```

Bin>javac nocomm.java
Bin>java nocomm

```
Consumed: 0
Consumed : 0
Produced : 1
Consumed : 1
Consumed : 1
Produced : 2
Consumed : 2
Produced : 3
Produced : 4
Produced : 5
```

Explanation :

In the main(), **sobj** is an object defined to **shop** class. The statement producer **p =new producer(sobj);** creates an object of producer class and to the constructor **sobj** is passed. In the constructor thread object is defined and starts the thread.

The statement consumer **c=new consumer(sobj);** creates an object of consumer class and to the constructor the same **sobj** is passed and the constructor starts the thread. Two threads are using the same **shop** class object **sobj**.

The two threads are started, but the producer produces the products for every **2** seconds where as the consumer thread consumes for every **1** second. As the timing of two thread are different and no communication exist between the threads, we get the output as shown above.

A program demonstrates inter-thread communication through consumer producer problem

The above program is modified using the **wait()** and **notify()** to establish the communication between the threads to get the correct output as explained in the previous program.

Bin>edit comm.java

```
class shop
{
   int pno=0;
   boolean flag=false;
   synchronized public void put(int n)
   {
         try
         {
               if(flag==true)
                  wait();
         }
         catch(InterruptedException e)
         {
               System.out.print("\nError : " +e);
         }
         pno=n;
         System.out.print("\n Produced : "+pno);
         flag=true;
         notify();
   }
   synchronized public void get()
   {
         try
         {
               if(flag==false)
                  wait();
         }
         catch(InterruptedException e)
         {
               System.out.print("Error : "+e);
         }
         System.out.print("\n Consumed d : "+pno);
         flag=false;
         notify();
   }
}
class producer implements Runnable
{
   Thread t;
```

```
    shop sobj;
    public producer(shop ob)
    {
          sobj=ob;
          t=new Thread(this, "producer");
          t.start();
    }
    public void run()
    {
          try
          {
                for(int i=1;i<=5;i++)
                {
                      Thread.sleep(2000);
                      sobj.put(i);
                }
          }
          catch(InterruptedException e)
          {
                System.out.print("\nError : "+ e);
          }
    }
}
class consumer implements Runnable
{
    Thread t;
    shop sobj;
    public consumer(shop ob)
    {
          sobj=ob;
          t=new Thread(this,"consumer");
          t.start();
    }
    public void run()
    {
          try
          {
                for(int i=1;i<=5;i++)
                {
                      sobj.get();
```

```
                    Thread.sleep(1000);
                }
            }
            catch(InterruptedException e)
            {
                System.out.print("\nError : "+ e);
            }
        }
    }
    class comm
    {
        public static void main(String argv[ ])
        {
            shop sobj=new shop();
            producer p =new producer(sobj);
            consumer c=new consumer(sobj);
        }
    }
```

Bin>javac comm.java
Bin>java comm

```
Produced : 1
Consumed : 1
Produced : 2
Consumed : 2
Produced : 3
Consumed : 3
Produced : 4
Consumed : 4
Produced : 5
Consumed : 5
```

Explanation :

In the main(), **sobj** is an object defined to **shop** class. The statement **producer p =new producer(sobj);** creates an object of producer class and to the constructor **sobj** is passed. In the constructor thread object is defined and starts the thread.

The statement **consumer c=new consumer(sobj);** creates an object of consumer class and to the constructor the same **sobj** is passed and the constructor starts the thread. Two threads are using the same **shop** class object **sobj**.

When two threads(**producer** and **consumer**) are started, the producer produces the products for every **2** seconds where as the consumer thread consumes for every **1** second. As the timing of two threads are different and communication establishes using **wait()** and **notify()** methods and we get the output as shown above.

In the **put()** method, the **wait()** suspends the producer thread until the **notify()** from **get()** method is called by consumer thread. Similarly, in the **get()** method the **wait()** suspends the consumer thread until the **notify()** from the put() method is called by the producer thread. In this way two threads communicates with each other using **wait()** and **notify()**.

Daemon Threads

Threads that work in the background to support the runtime environment are called *daemon threads*. For example, the clock handler thread, the idle thread, the screen updater thread, and the garbage collector thread are all daemon threads. The virtual machine exits whenever all **non-daemon** threads have completed.

public final void setDaemon(boolean isDaemon)

public final boolean isDaemon()

By default a thread you create is not a daemon thread. However you can use the **setDaemon(true)** method to make the thread into demon.

Deadlock

Other threads that want access to a synchronized object must wait for the first thread to release the object before they can continue. When a thread is waiting for another thread to release the lock on an object it is ***blocked***.

Synchronization is a dangerous thing and should be avoided where possible. Two threads can each lock an object the other thread needs, and thus prevent both threads from running. This is called *deadlock*.

Deadlock is a special type of error that occurs in multithreading. Deadlock occurs when two threads have a circular dependency on a pair of synchronized objects. For example, one thread enters into monitor on object **A** and another thread enters into monitor on object **B**. If the thread in **A** tries to call any synchronized method on **B**, it will be blocked in the monitor. At this time, if the thread in **B**, calls any synchronized method on **A**, the thread waits forever, because to access **A**, it would have to release its own lock on **B** so that the first thread could complete.

Once deadlock situation occurs in multithreading than we can't do anything, except to terminate the program. We should see that not to get the thread deadlock. The dead lock occurs because to two reasons:

1. If two threads time-slice is just the same way.
2. If two threads involves in two synchronized objects.

A program demonstrates deadlock

In this program, we take to classes **printer** and **scanner**. The printer class contain two synchronized methods **print()** and **printresource()**. The scanner class contain two synchronized methods **scan()** and **scanresource()**. The printer object calls the **print()** and scanner object calls the **scan()**. Therefore the two objects locks in the monitor.

In this situation, from the **print()** method the scanner object calls the **scanresouce()** for sharing, as a result the object waits because the scanner object is already in the monitor and can't enter into other monitor. At this time, if printer object calls the **printresource()** from the **scan()** for sharing then the printer object goes into waiting state, because the printer object was already in the monitor and can't enter into monitor again. In this way two objects printer and scanner gets into waiting state indefinitely and are blocked.

The **printresource()** will be executed if the printer object comes out of the monitor and it comes out of the monitor when **scanresource()** is executed. Similarly, the **scanresource()** will be executed if the scanner object comes out of monitor and it comes out when **printresource()** is executed. As there is a circular dependency of resources among two synchronized object the two object gets locked forever which is ***deaklock***.

Bin>edit dead1.java

```
class printer
{
   synchronized void print(scanner sobj)
   {
         System.out.print("\n printing…");
         try
         {
               Thread.sleep(2000);
         }
         catch(InterruptedException e)
         {
               System.out.print("\nError:"+e);
```

```
            }
            sobj.scanresource();
            System.out.print("\n completed printing…");
      }
      synchronized void printresource()
      {
            System.out.print("\n resource of printer…");
      }
}
class scanner
{
      synchronized void scan(printer pobj)
      {
            System.out.print("\n scanning…");
            try
            {
                  Thread.sleep(2000);
            }
            catch(InterruptedException e)
            {
                  System.out.print("\nError:"+e);
            }
            pobj.printresource();
            System.out.print("\n completed scanning…");
      }
      synchronized void scanresource()
      {
            System.out.print("\n resource of scanner…");
      }
}
class deadlock implements Runnable
{
      printer pobj=new printer();
      scanner sobj=new scanner();
      Thread t;
      public deadlock()
      {
            t=new Thread(this,"deadlock");
            t.start();
            sobj.scan(pobj);
      }
```

```
    public void run()
    {
          pobj.print(sobj);
    }
}
class dead1
{
    public static void main(String argv[])
    {
          deadlock dobj=new deadlock();
    }
}
```

Bin>javac dead1.java
Bin>java dead1
printing...
scanning...