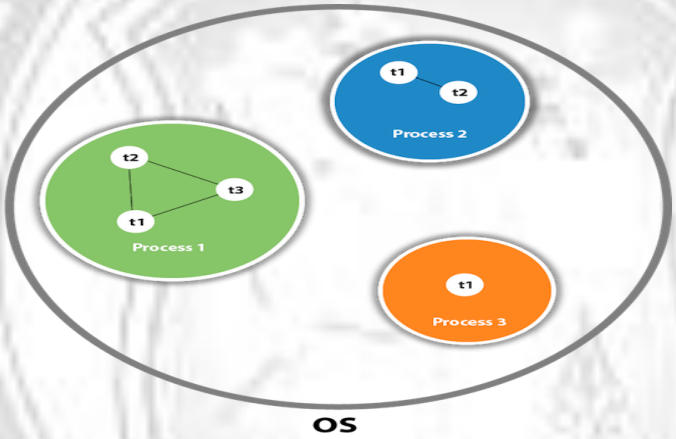
3.6: MULTITHREADED PROGRAMMING

INTRODUCTION TO THREAD

Definition: THREAD

A thread is a lightweight sub-process that defines a separate path of execution. It is the smallest unit of processing that can run concurrently with the other parts (other threads)

- ✓ Threads are independent.
- ✓ If there occurs exception in one thread, it doesn't affect other threads.
- ✓ It uses a shared memory area.
- ✓ As shown in the above figure, a thread is executed inside the process.



- ✓ There is context-switching between the threads.
- ✓ There can be multiple processes inside the OS, and one process can have multiple threads.

DIFFERENCE BETWEEN THREAD AND PROCESS:

S.NO	PROCESS	THREAD
1)	Process is a heavy weight program	Thread is a light weight process
2)	Each process has a complete set of its own variables	Threads share the same data
3)	Processes must use IPC (Inter-Process Communication) to communicate with sibling processes	Threads can directly communicate with each other with the help of shared variables
4)	Cost of communication between processes is high.	Cost of communication between threads is low.
5)	Process switching uses interface in operating system.	Thread switching does not require calling an operating system.
6)	Processes are independent of one another	Threads are dependent of one another
7)	Each process has its own memory and resources	All threads of a particular process shares the common memory and Resources
8)	Creating & destroying processes takes more overhead	Takes less overhead to create and destroy individual threads

MULTITHREADING

A program can be divided into a number of small processes. Each small process can be addressed as a single thread.

Definition: Multithreading

Multithreading is a technique of executing more than one thread, performing different tasks, simultaneously.

Multithreading enables programs to have more than one execution paths which executes concurrently. Each such execution path is a thread. For example, one thread is writing content on a file at the same time another thread is performing spelling check.

Advantages of Threads / Multithreading:

1. Threads are light weight compared to processes.
2. Threads share the same address space and therefore can share both data and code.
3. Context switching between threads is usually less expensive that between processes.
4. Cost of thread communication is low than inter-process communication.
5. Threads allow different tasks to be performed concurrently.
6. Reduces the computation time.
7. Through multithreading, efficient utilization of system resources can be achieved.

MULTITASKING

Definition: Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to maximize the utilization of CPU.

Multitasking can be achieved in two ways:

1) **Process-based Multitasking (Multiprocessing):-**
   - **It is a feature of executing two or more programs concurrently.**
   - For example, process-based multitasking enables you to run the Java compiler at the same time that you are using a text editor or visiting a web site.
2) **Thread-based Multitasking (Multithreading):-**
   - **It is a feature that a single program can perform two or more tasks simultaneously.**
   - For instance, a text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads.

Differences between multi-threading and multitasking

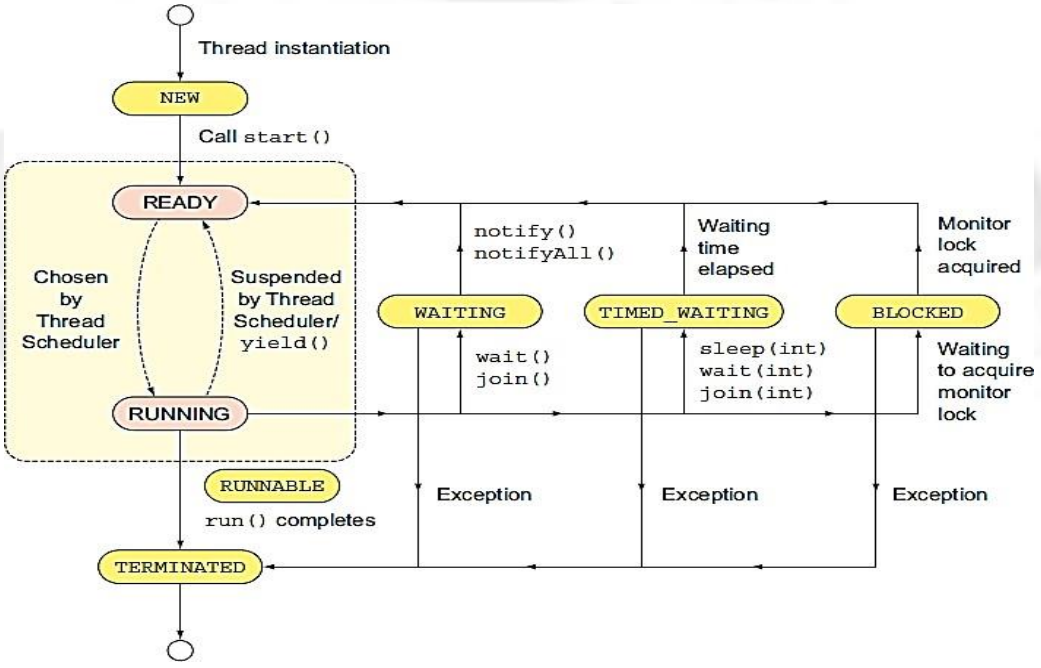
Characteristics	Multithreading	Multitasking
Meaning	A process is divided into several different sub-processes called as threads, which has its own path of execution. This concept is called as multithreading.	The execution of more than one task simultaneously is called as multitasking.

Number of CPU	Can be one or more than one	One
Number of process being executed	Various components of the same process are being executed at a time.	One by one job is being executed at a time.
Number of users	Usually one.	More than one.
Memory Space	Threads are lighter weight. They share the same address space	Processes are heavyweight tasks that require their own separate address spaces.
Communication between units	Interthread communication is Inexpensive	Interprocess communication is expensive and limited.
Context Switching	Context switching from one thread to the next is lower in cost.	Context switching from one process to another is also costly.

3.7: Thread Model / Thread Life Cycle (Different states of a Thread)

Different states, a thread (or applet/servlet) travels from its object creation to object removal (garbage collection) is known as life cycle of thread. A thread goes through various stages in its life cycle. At any time, a thread always exists in any one of the following state:

1. New State
2. Runnable State
3. Running State
4. Waiting/Timed Waiting/Blocked state
5. Terminated State/ dead state



1. New State:

A new thread begins its life cycle in the new state. It remains in this state until the

program starts the thread by calling **start()** method, which places the thread in the **runnable** state.

- ✓ A new thread is also referred to as a born thread.
- ✓ When the thread is in this state, only **start()** and **stop()** methods can be called. Calling any other methods causes an **IllegalThreadStateException**.
- ✓ Sample Code: **Thread myThread=new Thread();**

2. Runnable State:

After a newly born thread is started, the thread becomes runnable or running by calling the **run()** method.

- ✓ A thread in this state is considered to be executing its task.
- ✓ Sample code: **myThread.start();**
- ✓ The **start()** method creates the system resources necessary to run the thread, schedules the thread to run and calls the thread's run() method.

3. Running state:

- ✓ **Thread scheduler** selects thread to go from runnable to running state. In running state Thread starts executing by entering **run()** method.
- ✓ Thread scheduler selects thread from the runnable pool on basis of priority, if priority of two threads is same, threads are scheduled in unpredictable manner. Thread scheduler behaviour is completely unpredictable.
- ✓ When threads are in running state, **yield()** method can make thread to go in Runnable state.

4. Waiting/Timed Waiting/Blocked State :

❖ **Waiting State:**

Sometimes one thread has to undergo in waiting state because another thread starts executing. A runnable thread can be moved to a waiting state by calling the **wait**() method.

- ✓ A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.
- ✓ A call to **notify()** and **notifyAll()** may bring the thread from waiting state to runnable state.

❖ **Timed Waiting:**

A runnable thread can enter the timed waiting state for a specified interval of time by calling the **sleep()** method.

- ✓ After the interval gets over, the thread in waiting state enters into the runnable state.
- ✓ Sample Code:

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

❖ **Blocked State:**

When a particular thread issues an I/O request, then operating system moves the thread to blocked state until the I/O operations gets completed.

- ✓ This can be achieved by calling **suspend()** method.
- ✓ After the I/O completion, the thread is sent back to the runnable state.

5. Terminated State:

A runnable thread enters the terminated state when,

(i) It completes its task (when the run() method has finished)

public void run() { }

(ii) Terminates (when the stop() is invoked) – **myThread.stop();**

A terminated thread cannot run again.

New : A thread begins its life cycle in the new state. It remains in this state until the start() method is called on it.
Runnable : After invocation of start() method on new thread, the thread becomes runnable.
Running : A thread is in running state if the thread scheduler has selected it.
Waiting : A thread is in waiting state if it waits for another thread to perform a task. In this stage the thread is still alive.
Terminated : A thread enter the terminated state when it complete its task.

THE "main" THREAD

The "main" thread is a thread that begins running immediately when a java program starts up. The "main" thread is important for two reasons:
1. It is the thread form which other child threads will be spawned.
2. It must be the last thread to finish execution because it performs various shutdown actions.

- ✓Although the main thread is created automatically when our program is started, it can be controlled through a **Thread** object.
- ✓To do so, we must obtain a reference to it by calling the method **currentThread()**.

Example:

```
class CurrentThreadDemo {
public static void main(String args[])
{
Thread t=Thread.currentThread();
System.out.println("Current Thread: "+t);
```

```
// change the name of the main thread
t.setName("My Thread");
System.out.println("After name change : "+t);



try {
for(int n=5;n>0;n--) {
System.out.println(n);
Thread.sleep(1000);// delay for 1 second
}
    } catch(InterruptedException e) {
          System.out.println("Main Thread Interrrupted");
  }
}
}
```

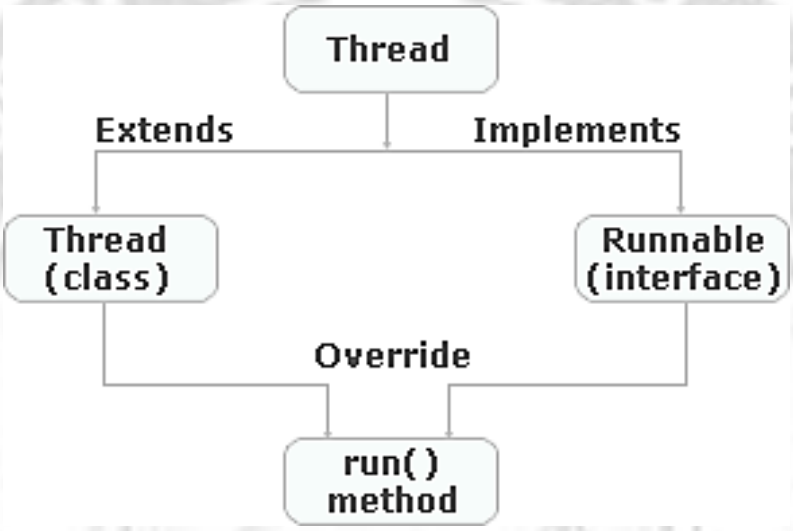
Output:

```
Current Thread: Thread[main,5,main]
After name change: Thread[My Thread,5,main]
5
4
3
2
1
```

3.8: Creating Threads

We can create threads by instantiating an object of type **Thread.** Java defines two ways to create threads:

1. By implementing **Runnable** interface (**java.lang.Runnable)**
2. By extending the **Thread** class (**java.lang.Thread)**



1. Creating threads by implementing Runnable interface:

- The Runnable interface should be implemented by any class whose instances are intended to be executed as a thread.
- Implementing thread program using Runnable is preferable than implementing it by extending Thread class because of the following two reasons:
  1. If a class extends a Thread class, then it cannot extend any other class.
  2. If a class Thread is extended, then all its functionalities get inherited. This is an expensive operation.
- The Runnable interface has only one method that must be overridden by the class which implements this interface:

```
public void run()// run() contains the logic of the thread
{
// implementation code
}
```

- **Steps for thread creation:**

1. Create a class that implements **Runnable** interface. An object of this class is **Runnable** object.

```
public class MyThread implements Runnable
{
---
}
```

2. Override the **run()** method to define the code executed by the thread.
3. Create an object of type Thread by passing a Runnable object as argument.

Thread t=new Thread(Runnable threadobj, String threadName);

4. Invoke the **start()** method on the instance of the Thread class.
   t.start();

- **Example:**

```
class MyThread implements Runnable
{
public void run()
{
for(int i=0;i<3;i++)
{
System.out.println(Thread.currentThread().getName()+" # Printing "+i);
try
{
Thread.sleep(1000);
}catch(InterruptedException e)
{
System.out.println(e);
  }
    }
}
}
public class RunnableDemo {
public static void main(String[] args)
{
MyThread obj=new MyThread();
MyThread obj1=new MyThread();
Thread t=new Thread(obj,"Thread-1");
t.start();
Thread t1=new Thread(obj1,"Thread-2");
t1.start();
}
}
```

Output:

```
Thread-0 # Printing 0
Thread-1 # Printing 0
Thread-1 # Printing 1
Thread-0 # Printing 1
Thread-1 # Printing 2
Thread-0 # Printing 2
```

2. Creating threads by extending Thread class:

Thread class provide constructors and methods to create and perform operations on a thread.

Commonly used Constructors of Thread class:

- Thread()
- Thread(String name)
- Thread(Runnable r)
- Thread(Runnable r, String name)

All the above constructors creates a new thread.

Commonly used methods of Thread class:

1. **public void run():** is used to perform action for a thread.
2. **public void start():** starts the execution of the thread.JVM calls the run() method on the thread.
3. **public void sleep(long miliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
4. **public void join():** waits for a thread to die.
5. **public void join(long miliseconds):** waits for a thread to die for the specified miliseconds.
6. **public int getPriority():** returns the priority of the thread.
7. **public int setPriority(int priority):** changes the priority of the thread.
8. **public String getName():** returns the name of the thread.
9. **public void setName(String name):** changes the name of the thread.
10. **public Thread currentThread():** returns the reference of currently executing thread.
11. **public boolean isAlive():** tests if the thread is alive.
12. **public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.
13. **public void suspend():** is used to suspend the thread(depricated).
14. **public void resume():** is used to resume the suspended thread(depricated).
15. **public void stop():** is used to stop the thread(depricated).
16. **public boolean isDaemon():** tests if the thread is a daemon thread.
17. **public void setDaemon(boolean b):** marks the thread as daemon or user thread.
18. **public void interrupt():** interrupts the thread.
19. **public boolean isInterrupted():** tests if the thread has been interrupted.
20. **public static boolean interrupted():** tests if the current thread has been interrupted.

- **<u>Steps for thread creation:</u>**

  1. Create a class that extends **java.lang.Thread** class.

  ```
  public class MyThread extends Thread
  {
  ---
  }
  ```

  2. Override the **run()** method in the sub class to define the code executed by the thread.

  3. Create an object of this sub class.

  MyThread t=new MyThread(String threadName);

4. Invoke the **start()** method on the instance of the subclass to make the thread for running.

start();

- **Example:**

```
class SampleThread extends Thread
{
public void run()
{
for(int i=0;i<3;i++)
{
System.out.println(Thread.currentThread().getName()+" # Printing "+i);
try
{
Thread.sleep(1000);
}catch(InterruptedException e)
{
System.out.println(e);
}
}
}
}
public class ThreadDemo {
public static void main(String[] args) {
SampleThread obj=new SampleThread();
obj.start();
SampleThread obj1=new SampleThread();
obj1.start();
}
}
```

Output:

```
Thread-0 # Printing
0Thread-1 # Printing
0Thread-1 # Printing
1Thread-0 # Printing
1Thread-0 # Printing
2Thread-1 # Printing
2
```

3.9: THREAD PRIORITY

- ✓ Thread priority determines how a thread should be treated with respect to others.
- ✓ Every thread in java has some priority, it may be default priority generated by JVM or customized priority provided by programmer.
- ✓ Priorities are represented by a number between 1 and 10.

  1 – Minimum Priority5 – Normal Priority 10 – Maximum Priority
- ✓ Thread scheduler will use priorities while allocating processor. The thread which is having highest priority will get the chance first.
- ✓ Higher priority threads get more CPU time than lower priority threads.
- ✓ A higher priority thread can also preempt a lower priority thread. For instance, when a lower priority thread is running and a higher priority thread resumes (for sleeping or waiting on I/O), it will preempt the lower priority thread.

> - ✓ **Thread scheduler** is a part of Java Virtual Machine (JVM). It decides which thread should execute first among two or more threads that are waiting for execution.
> - ✓ It is decided based on the priorities that are assigned to threads. The thread having highest priority gets a chance first to execute.
> - ✓ If two or more threads have same priorities, we can't predict the execution of waiting threads. It is completely decided by thread scheduler. It depends on the type of algorithm used by thread scheduler.

- ✓ If two or more threads have same priorities, we can't predict the execution of waiting threads. It is completely decided by thread scheduler. It depends on the type of algorithm used by thread scheduler.
- ✓ **3 constants defined in Thread class:**

  1.public static int MIN_PRIORITY
  2.public static int NORM_PRIORITY
  3.public static int MAX_PRIORITY
- ✓ Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

- ✓ To set a thread's priority, use the **setPriority()** method.
- ✓ To obtain the current priority of a thread, use **getPriority()** method.

- ✓ **Example:**

```
class TestMultiPriority1 extends Thread{
 public void run(){
  System.out.println("running thread name is:"+Thread.currentThread().getNam
e());
  System.out.println("running thread priority is:"+
                                    Thread.currentThread().getPriority());
 }


 public static void main(String args[]){
  TestMultiPriority1 m1=new TestMultiPriority1();
  TestMultiPriority1 m2=new TestMultiPriority1();
  m1.setPriority(Thread.MIN_PRIORITY);
  m2.setPriority(Thread.MAX_PRIORITY);
  m1.start();
  m2.start();
 }
}
```

Output:

```
running thread name is:Thread-0
running thread priority is:10
running thread name is:Thread-1
running thread priority is:1
```

3.10: Thread Synchronization

Definition: Thread Synchronization

Thread synchronization is the concurrent execution of two or more threads that share critical resources.

When two or more threads need to use a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process of ensuring single thread access to a shared resource at a time is called **synchronization.**

Threads should be synchronized to avoid critical resource use conflicts. Otherwise, conflicts may arise when parallel-running threads attempt to modify a common variable at the same time.

- ✓ **Why use Synchronization**

The synchronization is mainly used to

1. To prevent thread interference.
2. To prevent consistency problem.

✓ Thread Synchronization

There are two types of thread synchronization mutual exclusive and inter-thread communication.

1. Mutual Exclusive
    1. Synchronized method.
    2. Synchronized block.
    3. static synchronization.
2. Cooperation (Inter-thread communication in java)

✓ Mutual Exclusive

Mutual Exclusive helps keep threads from interfering with one another while sharing data. This can be done by two ways in java:

1. by synchronized method
2. by synchronized block

✓ Concept of Lock in Java

Synchronization is built around an internal entity known as the lock or monitor. Every object has a lock associated with it. By convention, a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with them.

1. Java synchronized method

- ✓ If you declare any method as synchronized, it is known as synchronized method. Synchronized method is used to lock an object for any shared resource.
- ✓ When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

Syntax to use synchronized method:

```
Access_modifier synchronized return_type method_name(parameters)
{ …….. }
```

Example of java synchronized method:

```
class Table{
 synchronized void printTable(int n)//synchronized method
{
for(int i=1;i<=5;i++) {
   System.out.println(n*i);
   try{   Thread.sleep(400);
          }
      catch(Exception e) { System.out.println(e); }
    }
 }
}
```

```
class MyThread1 extends
Thread {Table t;
MyThread1(Table t){this.t=t;
}
public void
run(){
t.printTable(5);
}
}
class MyThread2 extends
Thread{Table t;
MyThread2(Table
t){this.t=t;
}
public void run(){
t.printTable(100);
}
}
public class TestSynchronization2{
public static void main(String
args[]){
Table obj = new Table(); //only one
objectMyThread1 t1=new
MyThread1(obj); MyThread2 t2=new
MyThread2(obj); t1.start();
t2.start();
}}
```

Output:

```
 5
10
15
20
25
100
200
300
400
500
```

2. Synchronized block in java

- ✓ Synchronized block can be used to perform synchronization on any specific resource of the method.
- ✓ Suppose you have 50 lines of code in your method, but you want to synchronize only 5 lines, you can use synchronized block.
- ✓ If you put all the codes of the method in the synchronized block, it will work same as the synchronized method.

Points to remember for Synchronized block

- Synchronized block is used to lock an object for any shared resource.
- Scope of synchronized block is smaller than the method.

Syntax to use synchronized block

1. **synchronized (object reference expression) {**
2. **//code**

block3. }

Example of synchronized block

```
class Table{
 void printTable(int n)
{
  synchronized(this)  //synchronized block
  {
    for(int i=1;i<=5;i++){
     System.out.println(n*i);
     try{       Thread.sleep(400);       }catch(Exception e){System.out.println(e);}
    }
  }
 }//end of the method
}
class MyThread1 extends Thread{
        Table t;
        MyThread1(Table t){
                this.t=t;
        }
        public void run(){
                t.printTable(5);
        }
}
class MyThread2 extends Thread{
        Table t;
        MyThread2(Table t){
                this.t=t;
        }
        public void run(){
                t.printTable(100);
        }
}
public class TestSynchronizedBlock1
{
public static void main(String args[])
{
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
```

```
}
}
```

Output:

```
5
10
15
20
25
100
200
300
400
500
```

Difference between synchronized method and synchronized block:

Inter-Thread Communication Synchronized method	Synchronized block
1. Lock is acquired on whole method. 2. Less preferred. 3. Performance will be less as compared to synchronized block.	1. Lock is acquired on critical block of code only. 2. Preferred. 3. Performance will be better as compared to synchronized method.

3.11: Inter-Thread Communication

Inter-Thread Communication or Co-operation is all about allowing synchronized threads to communicate with each other.

Definition:

Inter-thread communication is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.

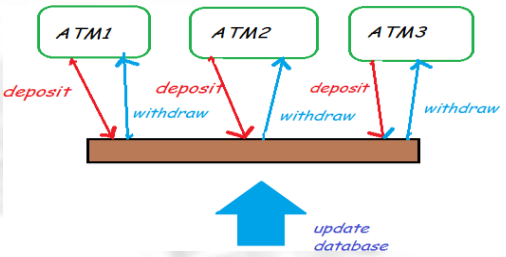
It is implemented by following methods of Object class and all these methods can be called only from within a synchronized context.

S.No.	Method & Description
1	**public final void wait() throws InterruptedException** Causes the current thread to wait until another thread invokes the notify().
2	**public final void wait(long timeout) throws InterruptedException** Causes current thread to wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed. **Parameters:** **timeout** – *the maximum time to wait in milliseconds.*
3	**public final void notify()** Wakes up a single thread that is waiting on this object's monitor.

4	**Public final void notifyAll()** Wakes up all the threads that called wait( ) on the same object.

Difference between wait() and sleep()		
Parameter	**wait()**	**sleep()**
Synchonized	wait should be called from synchronized context i.e. from block or method, If you do not call it using synchronized context, it will throw IllegalMonitorStateException	It need not be called from synchronized block or methods
Calls on	wait method operates on Object and defined in Object class	Sleep method operates on current thread and is in java.lang.Thread
Release of lock	wait release lock of object on which it is called and also other locks if it holds any	Sleep method does not release lock at all
Wake up condition	until call notify() or notifyAll() from Object class	Until time expires or calls interrupt()
static	wait is non-static method	sleep is static method

Example: The following program illustrates simple bank transaction operations with inter-thread communication:




```
class Customer{
int Balance=10000;

 synchronized void withdraw(int amount)
{
  System.out.println("going to withdraw..."+amount);

  if(Balance<amount)
  {
          System.out.println("Less balance; Balance = Rs. "+Balance+"\nWaiting for
  deposit...\n");
          try
{
  wait();
                  }
catch(Exception e){}
  }
  Balance-=amount;
```

```
  System.out.println("withdraw completed...");
}
 synchronized void deposit(int amount)
{
  System.out.println("going to deposit... Rs. "+amount);
  Balance+=amount;
  System.out.println("deposit completed... Balance = "+Balance);
  notify();
}
}
class ThreadCommn
{
  public static void main(String args[]) {
        Customer c=new Customer();
        new Thread()
        {
              public void run(){c.withdraw(20000);}
        }.start();
        new Thread(){
              public void run(){c.deposit(15000);}
        }.start();
  }
}
```

Output:

```
going to withdraw...20000
Less balance;  Balance = Rs. 10000
Waiting for deposit...
going to deposit... Rs. 15000
deposit completed... Balance = 25000
withdraw completed...
```

3.12: Suspending, Resuming and Stopping threads

The functions of Suspend, Resume and Stop a thread is performed using Boolean-type flags in a multithreading program. These flags are used to store the current status of the thread.

1. If the suspend flag is set to true, then run() will suspend the execution of the currently running thread.
2. If the resume flag is set to true, then run() will resume the execution of the suspended thread.

3. If the stop flag is set to true, then a thread will get terminated.

Example

```
class NewThread implements Runnable
{
   String name;     //name of
   threadThread thr;
   boolean suspendFlag;
   boolean stopFlag;




   NewThread(String threadname)
   {
      name = threadname;
      thr = new Thread(this, name);
      System.out.println("New thread : " + thr);
      suspendFlag = false;
      stopFlag = false;
      thr.start();    // start the thread
   }

   /* this is the entry point for thread */
   public void run()
   {
      try
      {
         for(int i=1; i<10; i++)
         {
            System.out.println(name + " : " + i);
            Thread.sleep(1000);
            synchronized(this)
            {
               while(suspendFlag)
               {
                  wait();
               }
               if(stopFlag)
                  break;
            }
         }
      }

      catch(InterruptedException e)
      {
         System.out.println(name + " interrupted");
      }
      System.out.println(name + " exiting...");
   }

   synchronized void mysuspend()
```

```
    {
       suspendFlag = true;
    }

    synchronized void myresume()
    {
       suspendFlag = false;
       notify();
    }




     synchronized void mystop()
    {
     suspendFlag=false;
     stopFlag=true;
     notify();
     System.out.println("Thread "+name+" Stopped!!!");
    }
}
 class SuspendResumeThread
{
    public static void main(String args[])
    {
       NewThread obj1 = new NewThread("One");
       NewThread obj2 = new NewThread("two");
       try
       {
          Thread.sleep(1000);
          obj1.mysuspend();
          System.out.println("Suspending thread One...");
          Thread.sleep(1000);
          obj1.myresume();
          System.out.println("Resuming thread One...");
          obj2.mysuspend();
          System.out.println("Suspending thread Two...");
          Thread.sleep(1000);
          obj2.myresume();
          System.out.println("Resuming thread Two...");
          obj2.mystop();
       }
       catch(InterruptedException e)
       {
          System.out.println("Main thread Interrupted..!!");
       }

       System.out.println("Main thread exiting...");

    }
```

}

Output:

New thread : Thread[One,5,main]
New thread : Thread[two,5,main]
One : 1
two : 1
One : 2

Suspending thread One...
two : 2
two : 3
Resuming thread One...
One : 3
Suspending thread Two...
One : 4
Resuming thread Two...
two : 4
Thread two Stopped!!!
Main thread exiting...
two exiting...
One : 5
One : 6
One : 7
One : 8
One : 9
One exiting...