* Multithreading :-

* Introduction :-

First, we need to know about Multitasking.

→ Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Example :- class Room student 1. Listing 2. writing

→ Multitasking can be achieved by two ways (or) Multitasking is classified into two types

- process - Based Multitasking (Multiprocessing)
- Thread - Based Multitasking (Multithreading)

* process based Multitasking :-

Executing multiple tasks simultaneously, where each task is separate independent process (or) program is called as process based Multitasking.

Example :-

1. Typing a Java program in notepad.
2. Listen audio songs.
3. Download a file from internet.

The above three tasks are performed simultaneously in a system, but there is no dependence between one task & another task.

→ process based Multitasking is best suitable at "operating system" level not at programming level.

* Thread - based Multitasking :-

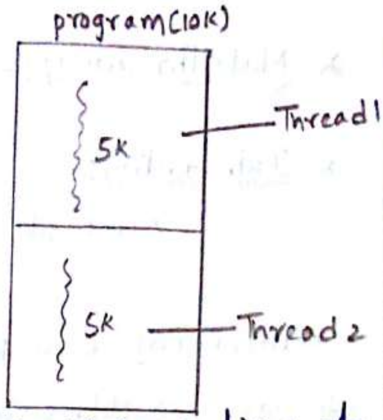
Executing multiple tasks simultaneously, where each task is a separate independent part of the same program (or) process is called Thread - based Multitasking.

→ The each independent part is called a thread.

→ Thread - based Multitasking is best suitable at programming level.

Example :-

let a program has 10K lines of code, where last 5K lines of code doesn't dependent on first 5K lines of code. then both are starts the execution simultaneously. so it takes less time to complete execution.



Note :- Any type of Multitasking is used to reduce response time of System and improves performance.

* Multithreading :-

→ A thread is a lightweight sub process, a smallest unit of processing. It is a separate path of execution.

→ Threads are independent, if there occurs exception in on thread, it doesn't affect other threads.

→ It shares a common memory area.

→ As shown in figure, thread is executed inside the process. There can be Multiple process inside the O.S and one process can have multiple threads.

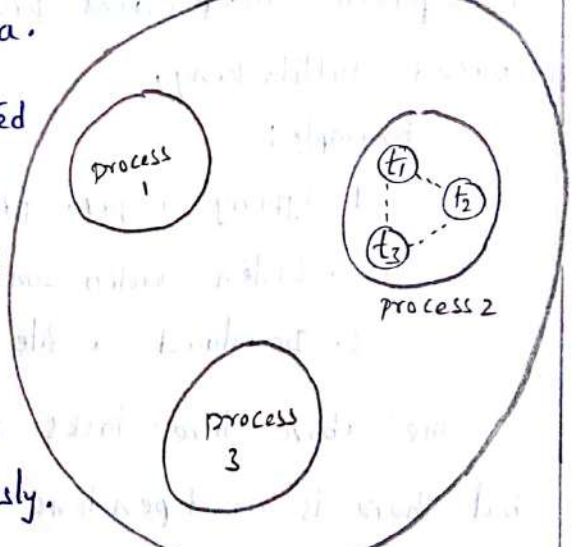


fig: operating System.

Defn:- MultiThreading is a process of executing multiple threads simultaneously.

↳ Multiprocessing and MultiThreading, both are used to achieve multitasking.

But we use multithreading than multiprocessing because threads shares a common memory area and context-switching between threads takes less time than process.

↳ The main application areas of Multithreading are

→ To develop Multi-media movies

→ To develop Video games

→ To develop web servers & Application servers, etc.

* Life Cycle of a Thread (or) A Thread Model :-

The Life cycle of the thread is controlled by JVM. And it has five states as follows

→ New
→ Runnable
→ Running
→ Non-Runnable (blocked)
→ Terminated

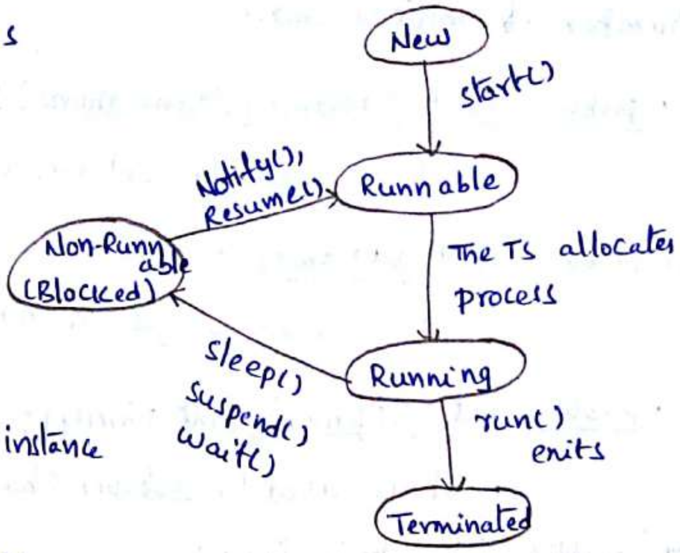


fig :- Life cycle of a Thread.

* New :- In this state, the new instance of thread class is created.
Eg :- Mythread t = new Mythread();

* Runnable :- In this, the thread is ready to enecute after invocation of start() method. Eg: t.start();

* Running :- In this, the thread is running by using run() method.

* Non-Runnable (Blocked) :-

In this, the thread is in blocked state, i.e the thread is still alive, but is currently not eligible to run.

* Terminated (Dead) :-

In this, the thread is in dead state, when run() method enits (or) completed the process.

* Thread class :-

The Thread class provides methods and constructors to create and perfom operations on a thread.

↳ Commonly used methods of Thread class :

1. public void run() : It is used to perfom action for a thread.
2. public void start() : It is used to starts the enecution of the thread. JVM calls the run() method on the thread.

3. public void sleep(long milliseconds):

To slope the execution of Thread for the specified number of milliseconds.

4. public void setName(String name):

It is used to set (or) changes the name of a Thread.

5. public void getName():

It is used to get the name of a thread.

6. public int setpriority(int priority):

It is used to set(or) change the priority of the Thread.

7. public int getpriority():

It returns the priority of the thread.

8. public boolean isAlive():

It checks if the Thread is alive or not.

9. public void yield():

It used to pause the currently executing thread and allow other Threads to execute.

10. public void suspend():

It is used to suspend the thread.

11. public void resume():

It is used to resume the suspended Thread.

12. public void stop():

It is used to stop the execution of Thread.

↳ Construdors:

1. Thread() - without parameters/arguments.
2. Thread(String name) - with one string argument.
3. Thread(Runnable r) - with runnable argument.
4. Thread(Runnable r, String name) - with runnable and string arguments.

* creating Thread :-

There are two ways to create a thread

↳ By entending Thread class.

↳ By implementing Runnable interface.

↳ By entending Thread class :-

In This, By entending Thread class we can able to create a thread and able to start a thread by calling start() method of thread class.

syntax:-

```
class class_name entends Thread
{
 ⋮
}
```

In thread class, we can mainly use two methods run() and start().

Example:-

SimpleThread.java

```
class A entends Thread
{
 public void run()
 {
  for(int i=1; i<=5; i++)
  {
   System.out.println("Thread A value ="+i);
  }
 }
}
class SimpleThread
{
public static void main(String arg[])
{
 A a=new A();
 a.start();
}
}
```

output:

```
javac SimpleThread.java
java SimpleThread
Thread A value = 1
   "           2
   "           3
   "           4
   "           5
```

↳ **By implementing Runnable interface :-**

In this, By implementing runnable interface, we can able to create & execute thread.

→ The Runnable interface Should be implemented by any class.

→ Runnable interface have only one Thread method named run(), where run() method is abstract method.

→ The programmer Should declare object for sub class, that object can be used as agument in thread class constructor from thread object.

Syntax:-

```
class class_name implements Runnable
{
 public void run()
 {
  ≡
 }
}
```

ThreadRunDemo.java

Example:-

```
class A implements Runnable
 {
  public void run()
  {
  for(int i=1; i<=5; i++)
  {
   System.out.println("Thread A value ="+i);
  }
 }
}
class ThreadRunDemo
{
public static void main(String a[])
{
 A a = new A();
 Thread t = new Thread(a);
 t.start();
}
}
```

output:-

```
javac ThreadRunDemo.java
java ThreadRunDemo
Thread A value = 1
    "          2
    "          3
    "          4
    "          5
```

* Thread priority :-

→ In java, each thread has a priority. priorities are represented by a number between 1 and 10.

→ In most cases, thread schedular schedules the threads according to their priority.

→ But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.

→ There are three constant thread priorities defined in Thread class.

1. public static int MIN_PRIORITY
2. public static int NORM_PRIORITY
3. public static int MAX_PRIORITY

→ The value of MIN_PRIORITY is 1,
The value of NORM_PRIORITY is 5,
and The value of MAX_PRIORITY is 10.

Example:

```
class Thread1 extends Thread
{
 public void run()
 {
 for (int i=1; i<=5; i++)
 {
  System.out.println("Thread1 i ="+i);
 }
  System.out.println("Thread1 is in terminated state");
 }
}
class Thread2 extends Thread
{
 public void run()
 {
```

```
for (int j=1; j<=5; j++)
 {
 System.out.println("Thread2 j=" +j);
 }
 System.out.println("Thread2 is in Terminated state");
 }
}
class Thread3 extends Thread
{
 public void run()
 {
 for (int k=1; k<=5; k++)
 {
  System.out.println("Thread3 k=" +k);
 }
 System.out.println("Thread3 is in Terminated state");
 }
}
class ThreadpriorityDemo
{
public static void main(String args[])
{
 Thread1 a = new Thread1();
 Thread2 b= new Thread2();
 Thread3 c= new Thread3();

 System.out.println("Default priority for Thread1 is:" +a.getpriority());
system.out.println("Default priority for Thread2 is:" +b.getpriority());
System.out.println("Default priority for Thread3 is:" +c.getpriority());
a.setpriority(Thread.MIN-PRIORITY);
b.setpriority(Thread.NORM-PRIORITY);
c.setpriority(Thread.MAX-PRIORITY);
```

```
System.out.println("New priority of Thread1 is:" + a.getpriority());
System.out.println("New priority of Thread2 is:" + b.getpriority());
System.out.println("New priority of Thread3 is:" + c.getpriority());

 a.start();
 b.start();
 c.start();
}
}
```

output:

```
javac ThreadpriorityDemo.java
java ThreadpriorityDemo
Default priority of Thread1 is: 5
  "        "        Thread2 is: 5
  "        "        Thread3 is: 5
New priority of Thread1 is: 1
  "        "      Thread2 is: 6
  "        "      Thread3 is: 10
 Thread3  k = 1
   "      k = 2
   "      k = 3
   "      k = 4
   "      k = 5
 Thread3  is in Terminated state
 Thread2  j = 1
   "      j = 2
   "      j = 3
   "      j = 4
   "      j = 5
 Thread2  is in Terminated state
 Thread1  i = 1
   "      i = 2
   "      i = 3
   "      i = 4
   "      i = 5
 Thread1  is in Terminated state.
```

* Methods of Thread class :-

→ sleep() :-

The sleep() method of Thread class is used to stop the execution of thread for the specified amount of time.

Syntax: sleep (long milliseconds)

The sleep() may throws an "InterruptedException". i.e we need to use try & catch statements while using sleep() method.

ThreadSleepDemo.java

Example :-

```
class sleepMethod extends Thread
{
 public void run()
 {
  for(int i=1; i<=5; i++)
  {
   try
   {
    Thread.sleep(500);
   }
   catch(InterruptedException e)
   {
    System.out.println(e);
   }
    System.out.println(i);
  }
 }
}
class ThreadSleepDemo
{
 public static void main(String arg[])
 {
  SleepMethod t1 = new SleepMethod();
  Sleep Method t2 = new SleepMethod();
   t1.start();
   t2.start();
 }
}
```

output:-

```
javac ThreadSleepDemo.java
java ThreadSleepDemo
1
1
2
2
3
3
4
4
5
5
```

Note:- If you sleep a thread for the specified time, the thread schedular picks up another thread and so on.

→ Naming Thread :-

→ The thread class provides methods to change and get the name of a thread, those are setName() and getName() methods.

→ By default, each thread has a name i.e thread-0, Thread-1 and so on.

getName(): It is used to get the name of thread.

Syntax: getName()

setName(): It is used to set (or) change the name of thread

Syntax: setName(String name)

Example:

```
class A extends Thread
{
 public void run()
 {
  System.out.println("Thread A is running...");
 }
}
class B extends Thread
{
 public void run()
 {
  System.out.println("Thread B is running...");
 }
}
class ThreadNameDemo
{
public static void main(String args[])
{
  A a=new A();
  B b=new B();
  System.out.println("Default Name of Thread A is"
                                        +a.getName());
  System.out.println("Default Name of Thread B is"
                                        +b.getName());
  a.setName("Madhu");
  b.setName("Hari");
System.out.println("New Name of Thread A is"
                                      + a.getName());

System.out.println("New Name of Thread B is"
                                      + b.getName());

 a.start();
 b.start();
}
}
```

output: javac ThreadNameDemo.java

```
java ThreadNameDemo
Default Name of Thread A is Thread-0
  "          "        B is Thread-1

New Name of Thread A is Madhu
  "    "          "   B is Havi

Thread A is running
Thread B is running
```

→ Joining Threads:-

Sometimes one thread need to know when other thread is terminating (or) not.

In Java, isAlive() and join() are two different methods that are used to check whether a thread has finished its execution or not.

* isAlive(): This method returns true if the thread still running otherwise it returns false.

Syntax: isAlive()

Example: ThreadisAliveDemo.java

```
class MyThread extends Thread
{
public void run()
{
 System.out.println("r1");
 try
 {
  Thread.sleep(500);
 }
 catch (InterruptedException e)
 {
  System.out.println(e);
 }
} } System.out.println("r2");
```

```
class ThreadisAliveDemo
{
 public static void main(String arg[])
 {
   MyThread t1 = new MyThread();
   MyThread t2 = new MyThread();
   t1.start();
  //t2.start();
   System.out.println(t1.isAlive());
   System.out.println(t2.isAlive());
  }
 }
```

output:

```
javac ThreadisAliveDemo.java
java ThreadisAliveDemo
True
false
r1
r2
```

* <u>join():</u>- This method waits for a specified thread completes its execution. It allow us to specify the time for which you want to wait for the specified thread to terminate.

<u>Syntax:</u> join()

(or)

join(long milliseconds)

→ join() method throw InterruptedException, i.e we need to use try & catch statements while using join() method.

<u>Example:</u>

```
class Thread1 extends Thread
{
 public void run()
{
for(int i=1; i<=5; i++)
{
 try
 {
 Thread.sleep(500);
```

```
  catch (InterruptedException e)
  {
   System.out.println(e);
  }
  System.out.println("Thread1 value is:"+i);
 }
 }
}
class Thread2 extends Thread
{
 public void run()
 {
 for(int i=1; i<=5; i++)
 {
  System.out.println("Thread2 value is :"+i);
 }
 }
}
class ThreadjoinDemo
{
 public static void main(String arg[])
 {
  Thread1 t1 = new Thread1();
  Thread2 t2 = new Thread2();
  t1.start();
  try
   {
   t1.join();
   }
   catch(InterruptedException e) { }
   t2.start();
  }
  }
```

output:-

```
javac ThreadjoinDemo.java
java ThreadjoinDemo
 Thread1  value  is :1
    "        "      2
    "        "      3
    "        "      4
    "        "      5
 Thread2  value  is :1
    "        "      2
    "        "      3
    "        "      4
    "        "      5
    "        "
```

Enample :- Enample for Multithreading, Race between Hare & Tortoise story.

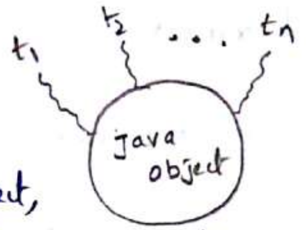
```
class Tortoise entends Thread
{
 public void run()
 {
 for(int i=1; i<=101; i++)
  {
   System.out.println("Distance covered by Tortoise = "+i);
  }
  System.out.println("Tortoise has completed the Race...");
 }
}
class Hare entends Thread
{
 public void run()
 {
 for(int j=1; j<=101; j++)
  {
   System.out.println("Distance covered by Hare = "+j);
  }
 System.out.println("Hare has completed the Race...");
 }
}
class Race
{
public static void main(String args[])
{
  Tortoise t = new Tortoise();
  Hare   h = new Hare();
  t.start();
  h.start();
}
}
```

<u>output</u>:-

```
javac Race.java
java Roce
Distance covered by Tortoise = 1
   .        .        .       2
Distance covered by Hare = 1
   .        .        .   2
Tortoise has completed the Race..
Hare has completed the Race..
```

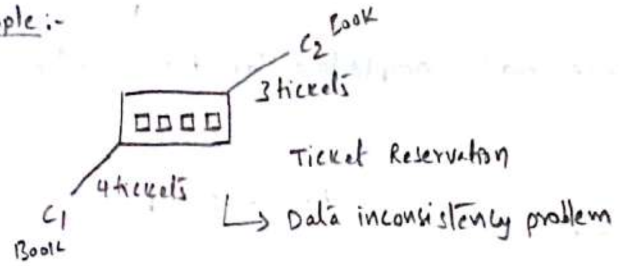
* Synchronization in Threads :-

Synchronization is a process, that allows only one thread to access shared resource at a time, if multiple threads trying to access shared resource.

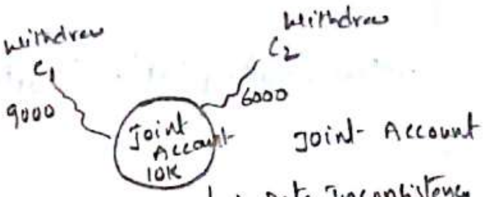


→ If multiple threads are trying to operate simultaneously on same java object, then there may be a chance of data inconsistency problem.

Example :-



↳ Data inconsistency problem



Joint Account

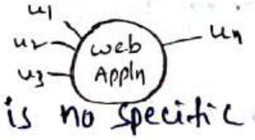
↳ Data Inconsistency problem

→ To overcome this problem, we should go for synchronized keyword.

→ Synchronized is a modifier applicable only for methods and blocks but not for classes & variables.

→ If a method declared as synchronized then at a time only one thread is allowed to execute that method on given object so that data inconsistency problem will be resolved.

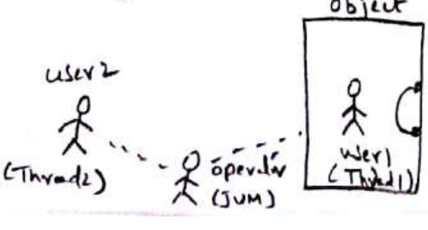
→ The main advantage of synchronized keyword is we can resolve data inconsistency problems, but the main disadvantage of synchronized keyword is it increases waiting time of threads and creates performance problems.



→ It increases response time if there are many users.

Note :- "Hence if there is no specific requirements, then it is not recomended to use synchronized keyword."

↳ Internally, Synchronization is implemented by using lock. Every object in java has an unique lock in java, whenever we are using synchronized keyword then only lock concept will come into the picture.

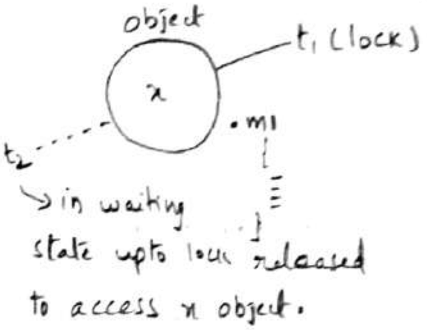


P.T.O

If a thread, want's to execute synchronized method on the given object, first it has to get lock of that object, one thread got the lock then it is allowed to execute any synchronized method on that object.

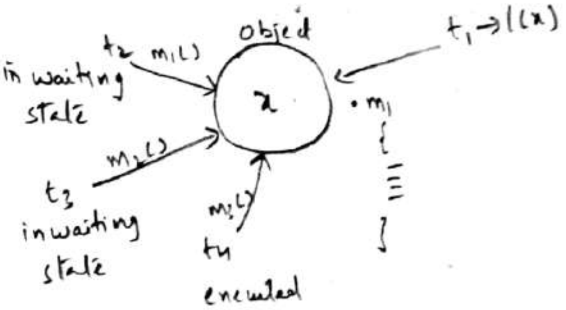
→ once method execution completes automatically thread releases the lock.

→ Acquiring & releasing lock internally takes cares by JVM not by programmer (or) Thread.



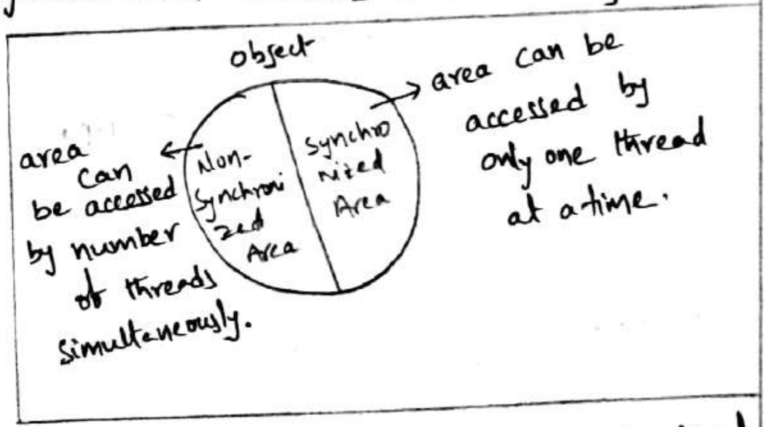
Example

```
class X
{
 synchronized m1();
 synchronized m2();
 m3();
}
```



→ While a thread executing synchronized method on the given object, the remaining threads are not allowed to execute any synchronized methods simultaneously on the same object, but remaining threads allowed to execute non-synchronized methods simulteneously.

Note:
Lock concept is implemented based on object but not based on method.


object
area can be accessed by number of threads simultaneously.
Non-Synchronized Area
synchronized Area
area can be accessed by only one thread at a time.

→ In java, any object can have two areas, those are synchronized area and non-synchronized area.

→ Synchronized area allows a thread to update (or) delete data.

→ Non-Synchronized area allows a thread to read the data (or) content.

* We should know, which method is declare as Synchronized (or) which method is declare as non-Synchronized method.

→ where ever, we are performing update operations (add/remove/replace) that methods can be declared as Synchronized (or) declared in Synchronized area.

→ where ever, we performing read operation, that methods can be declared as non-Synchronized/area.

Ex:-
```
class x
{
 Synchronized Area
 {
  //object state changing
 }
 Non-Synchronized Area
 {
  //object state won't be changed
 }
}
```

Example
```
class Banking
{
   Non-Synchronized  Balance Enquiry()
                     {
                      ⋮ Just read operation
                     }
      Synchronized   withdraw()
                     {
                      ⋮ update
                     }
}
```

Example
```
class TicketReservation
{
  Non-Synchronized  Seatavaliability()
                     {
                      ⋮ Just read operation
                     }
       Synchronized Booking()
                     {
                      ⋮ update
}                    }
```

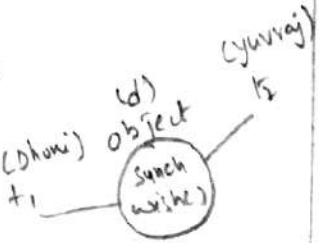
* Example :- SynchronizedDemo.java

```
class Display
{
public Synchronized void wish(String name)
{
for(int i=1; i<=5; i++)
{
System.out.println(" Good Morning :");
try
{
 Thread.sleep(2000);
}
catch(InterruptedException e){ }
System.out.println(name);
}
}
}
class MyThread extends Thread
{
 Display d;
 String name;
 MyThread(Display d, String name)
 {
  This.d=d;
  This.name=name;
 }
 public void run()
 {
 d.wish(name);
 }
}
class SynchronizedDemo
{
public static void main(String args[])
{
 Display d=new Display();
 MyThread t1=new MyThread(d,"Dhoni");
```

```
  MyThread t2 = new MyThread(d, "yuvraj");
   t1.start();
   t2.start();
  }
 }
```

output:- javac SynchronizedDemo.java

```
java SynchronizedDemo
  Good Morning : Dhoni
    "      "   : Dhoni
    "      "   : Dhoni
    "      "   : Dhoni
    "      "   : Dhoni
  Good Morning : yuvraj
    "      "   : yuvraj
    "      "   : yuvraj
    "      "   : yuvraj
    "      "   : yuvraj
```



Example :-

TSynchronizationDemo.java

```
class printTable
{
 public Synchronized void printtable(int n)
 {
  System.out.println("Table of" +n);
  for(int i=1; i<=10; i++)
  {
   System.out.println(n * i);
   try
   {
    Thread.sleep(2000);
   }
   catch(InterruptedException e) { }
  }
 }
}
class MyThread1 extends Thread
{
 printTable pt;
 MyThread1(printTable pt)
 {
  this.pt = pt;
 }
```

```
   public void run()
   {
    pt.printtable(2);
   }
  }
 class MyThread2 entends Thread
 {
  printTable pt;
  MyThread2(printTable pt)
  {
   this.pt = pt;
  }
  public void run()
  {
   pt.printtable(5);
  }
 }
class TSynchronizationDemo
{
public static void main(String args[])
{
 printTable obj=new printTable();
 MyThread1 t1 = new MyThread1(obj);
 MyThread2 t2 = new MyThread2(obj);
 t1.start();
 t2.start();
}
}
```

<u>output</u>:- javac TSynchronizationDemo.java

```
java TSynchronizationDemo
 Table of 2
  2
  4
  6
  :
  20
  Table of 5
  5
  10
  15
  :
  50
```

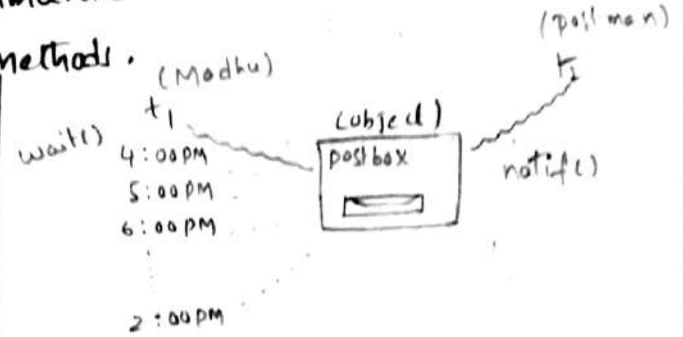
* Interthread communication :-

→ Interthread communication is a mechanism in which one thread is communicate with another thread by using wait(), notify() and notifyAll() methods.

(or)

→ Interthread communication is all about allowing Synchronized threads to communicate with each other.

→ Two threads can communicate with each other By using wait(), notify() and notifyAll() methods.

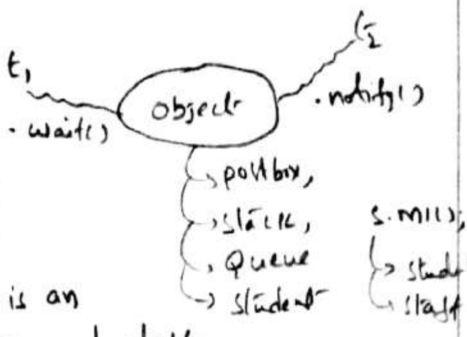


In this, t1 (Modhu) needs updation, so t1 is resposible to call wait(), And t2 (postman) needs to perform updation, so t2 is responsible to call notify().

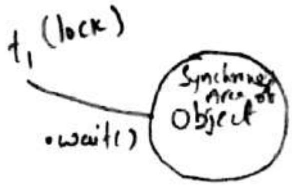
→ The thread, which is expecting updation is responsible to call wait() method, then immedially entered into waiting state

→ The thread, which is responsible to perform updation, after performing updation, it is responsible to call notify(), then waiting thread will get that notification and continue it's enecution with those updated information.

<u>Note</u>:- wait(), notify() and notifyAll() methods present in object class, but not in Thread class. Because a thread call wait(), notify() and notifyAll() on any object.



t.start()
t.join()
where t is an object of Thread class.



→ To call wait(), notify() (or) notifyAll() methods on any object, thread should be owner of that object. i.e the thread should has lock of that object, i.e the thread should be inside synchronized Area.

"Hence, we can call wait(), notify() and notifyAll() methods from synchronized Area otherwise we will get runtime Exception i.e " IllegalMonitorStateException".

```
    public void run()
    {
     pt.printtable(2);
    }
   }
  class MyThread2 enlends Thread
  {
   printTable pt;
   MyThread2(printTable pt)
   {
    this.pt = pt;
   }
   public void run()
   {
    pt.printtable(5);
   }
  }
 class TSynchronizationDemo
 {
 public static void main(String args[])
 {
  printTable obj=new printTable();
  MyThread1 t1=new MyThread1(obj);
  MyThread2 t2 = new MyThread2(obj);
  t1.start();
  t2.start();
 }
}
```

<u>output</u>:-

```
javac TSynchronizationDemo.java
jova TSynchronizationDemo
 Table of 2
  2
  4
  6
  :
  20
  Table of 5
  5
  10
  15
  :
  50
```

→ If a thread calls wait() method of any object, it immedially releases lock of That particular object and entered into waiting state.

→ If a thread calls notify() of any object, it releases lock of that object, but may not immediatly.

Note:- Except wait(), notify() & notifyAll(), There is no other method where thread releases The lock.

Methods:-

* wait()
  └→ wait() waits for until get notification (or) specified Milliseconds.

syntax:- public final void wait() throws InterruptedException

(or)

public final void wait(long ms) throws InterruptedException

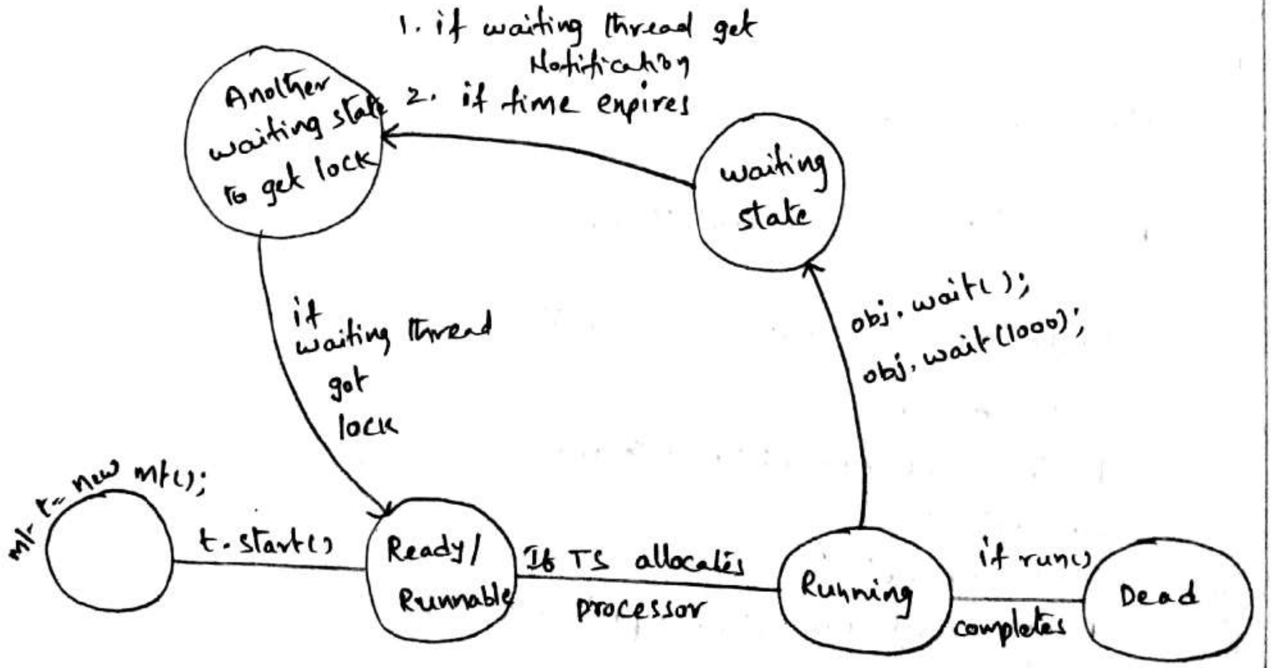
notify()
  └→ notify() gives notification to another thread

Syntax:- public final void notify()

(or)

public final void notifyAll()

→ understanding wait() & notify() methods in Thread Lifecycle



Example :-

```
class Customer
{
 int amount=10000;
 Synchronized void withdraw (int amount)
 {
  System.out.println("going to withdraw...");
  if (this.amount < amount)
  {
   System.out.println("Less Balance; waiting for deposit...");
   try
   {
    wait();
   }
   catch (Exception e) { }
  }
  this.amount -= amount;
  System.out.println ("withdraw completed");
 }
 Synchronized void deposit(int amount)
 {
  System.out.println("going to deposit...");
  This.amount += amount;
  System.out.println("Deposit completed...");
  notify();
 }
}
class ITCDemo
{
 public static void main(String args[])
 {
  customer c = new Customer();
  new Thread(){
   public void run(){c.withdraw(15000);}
   }.start();
  new Thread(){
   public void run(){c.deposit(10,000);}
   }.start();
 }
}
```

output:
```
javac ITCDemo.java
java ITCDemo
going to withdraw
Less Balance; waiting for deposit
going to deposit
deposit completed
withdraw completed.
```