Multi Threading

Thread:-

1) Thread is nothing but a separate path of sequential execution.
2) The independent execution technical name is called thread.
3) Whenever different parts of the program are executed simultaneously, each and every part is called thread.
4) The thread is a light weight process because whenever we are creating a thread it is not occupying the separate memory it uses the same memory. Whenever the memory is shared means it is not consuming more memory.
5) Executing more than one thread a time is called multithreading.

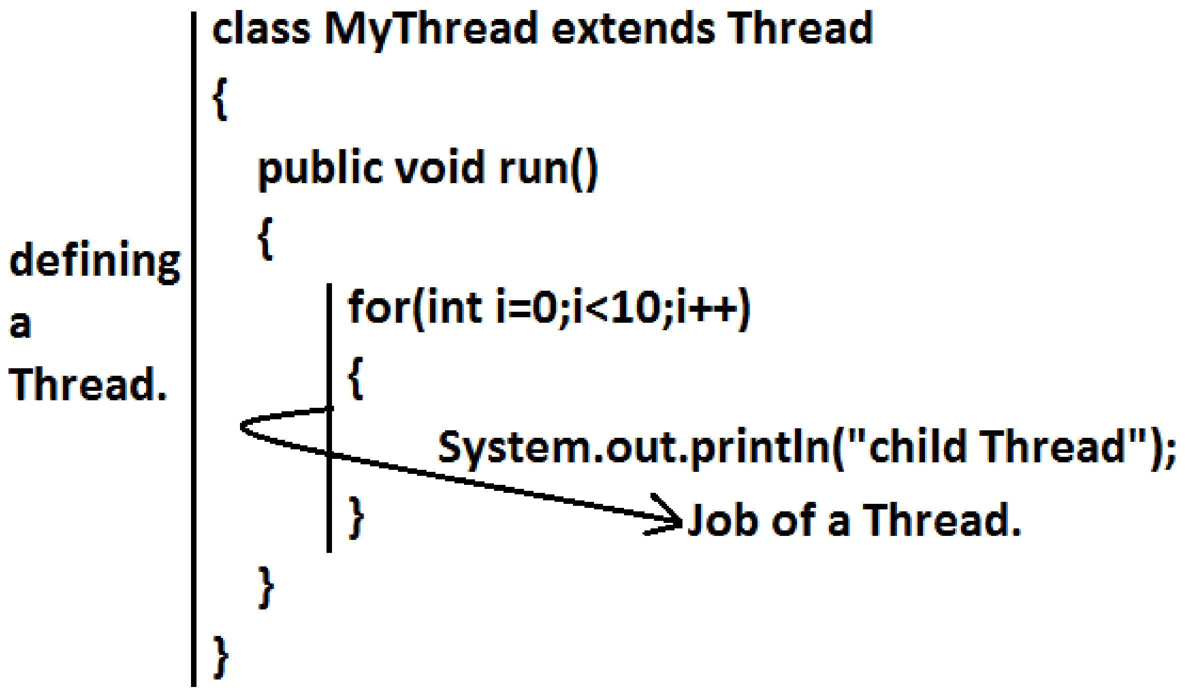
Information about main Thread
When a Java program starts one Thread is running immediately. That thread is called the main thread of your program.
1. It is used to create a new Thread(child Thread).
2. It must be the last thread to finish the execution because it performs various actions.

It is possible to get the current thread reference by using currentThread() method; it is a static public method present in Thread class.
The main important application areas of the multithreading are
1. Developing video games
2. Implementing multimedia graphics.
3. Developing animations

==

A **thread** can be created in two ways:-
1) By extending **Thread** class.
2) By *implementing* java.lang.**Runnable** interface.

1)Extending thread class



```
class MyThread extends Thread
{
  public void run()
  {
      for(int i=0;i<10;i++)
      {
          System.out.println("child Thread");
      }
  }
}
```

```
class ThreadDemo
{
public static void main(String[] args)
{
MyThread t=new MyThread();//Instantiation of a Thread
t.start();//starting of a Thread
for(int i=0;i<5;i++)
{
System.out.println("main thread");
}
}
}.
```

Thread Scheduler:

· If multiple Threads are waiting to execute then which Thread will execute 1st is decided by “Thread Scheduler” which is part of JVM.

· Which algorithm or behavior followed by Thread Scheduler we can’t expect exactly is the JVM vendor dependent hence in multithreading.

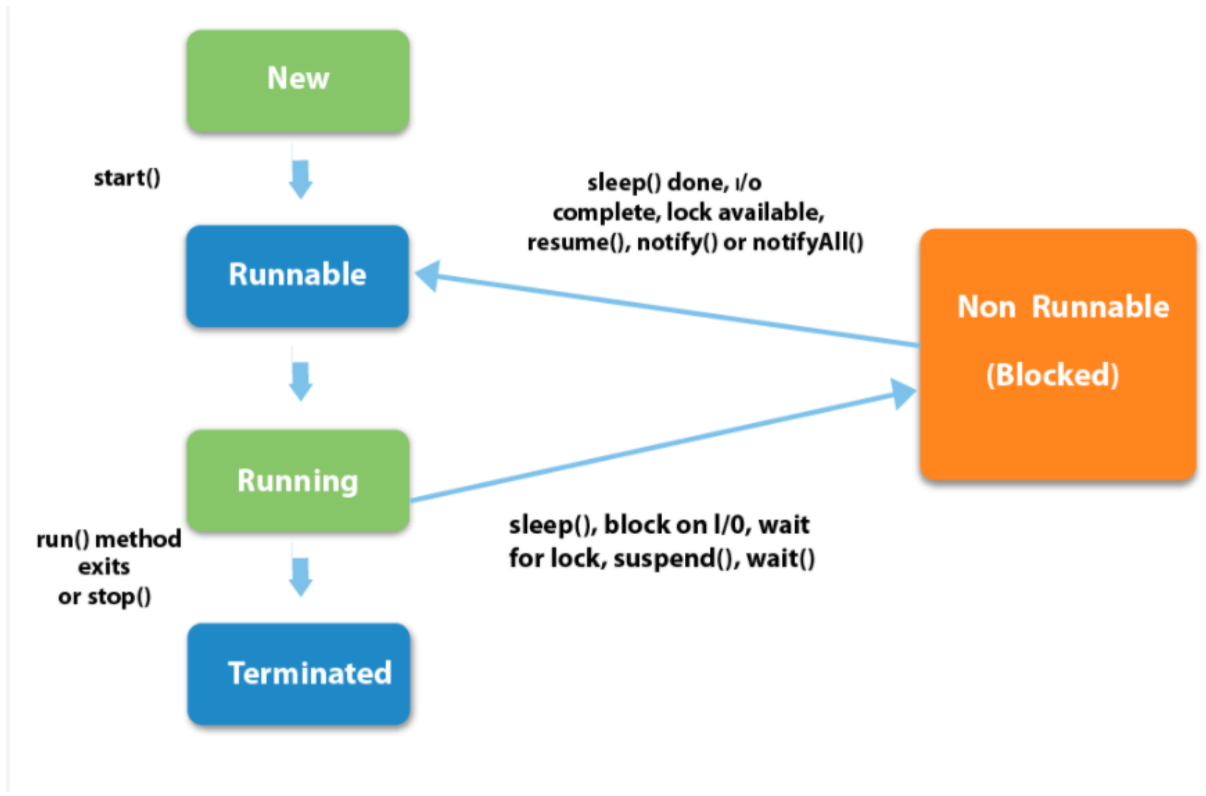
Difference between t.start() and t.run() methods.
· In the case of **t.start() a new Thread will be created** which is responsible for the execution of run() method. But in the case of t.run() no new Thread will be created and run() method will be executed just like a normal method by the main Thread.

importance of Thread class start() method.
· For every Thread the required mandatory activities like registering the Thread with Thread Scheduler will be taken care by Thread class start() method and the programmer is responsible just to define the job of the Thread inside run() method.

If we are not overriding run() method:
· If we are not overriding the run() method then the **Thread class run()** method will be executed which has empty implementation and hence we won’t get any output.

Overloading of run() method.
· We **can overload run() method** but Thread class start() method always invokes no argument run() method the other overload run() methods we have to call explicitly then only it will be executed just like normal method.

Thread Life Cycle


New
start()
Runnable
Running
run() method exits or stop()
Terminated
sleep() done, I/o complete, lock available, resume(), notify() or notifyAll()
Non Runnable (Blocked)
sleep(), block on I/0, wait for lock, suspend(), wait()

Life cycle stages are:-
1) *New*
2) Ready
3) Running state
4) Blocked / waiting / non-running mode
5) Dead state

1)**New** :- MyThread t=new MyThread();

2)**Ready** :- t.start()

3)**Running state**:- If thread scheduler allocates CPU for particular thread. Thread goes to running state
The Thread is running state means the run() is executed.

4)**Blocked State**:-
If the running thread gets interrupted or goes to a sleeping state at that moment it goes to the blocked state.

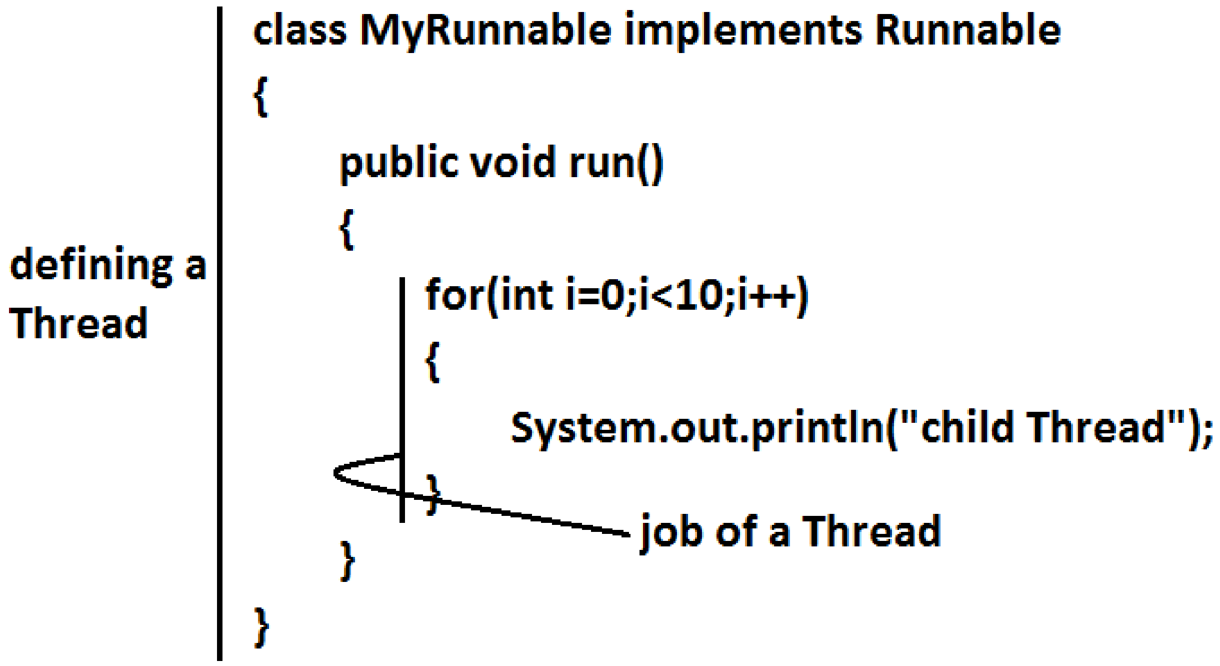
5)**Dead State**:-If the business logic of the project is completed means run() over thread goes dead state.

❖ After starting a Thread we are not allowed to restart the same Thread once again otherwise we will get a runtime exception saying "***IllegalThreadStateException***".

Second way of creating a Thread.

- We can define a Thread even by implementing **Runnable** interface also. Runnable interface present in java.lang.pkg and contains only one method **run**().

Example code :



```
class MyRunnable implements Runnable
{
      public void run()
      {
            for(int i=0;i<10;i++)
            {
                  System.out.println("child Thread");
            }
      }
}
```

```
class ThreadDemo
{
public static void main(String[] args)
{
MyRunnable r=new MyRunnable();
Thread t=new Thread(r);//here r is a Target Runnable
t.start();
for(int i=0;i<10;i++)
{
System.out.println("main thread");
}
}
}
```

So which is the best way of creating the thread ?

- Among the 2 ways of defining a Thread, **implementing a Runnable** approach is always recommended.
- · In the 1st approach our class should always **extend Thread** class there is no chance of extending any other class hence we are missing the benefits of inheritance.
- · But in the 2nd approach while **implementing Runnable** interface we can extend some other classes also.
- Hence implementing a Runnable mechanism is recommended to define a Thread.

Naming Thread

The Thread class provides methods to change and get the name of a thread. By default, each thread has a name i.e. thread-0, thread-1 and so on …

We can change the name of the thread by using setName() method. The syntax of setName() and getName() methods are given below

1. public String getName(): is used to return the name of a thread.
2. public void setName(String name): is used to change the name of a thread.

❖ We can *get current executing Thread* object reference by using **Thread.currentThread()** method.

<u>**Priority of a Thread (Thread Priority)**</u>

★ Each thread has a priority.
★ Priorities are represented by a number between 1 and 10.
★ In most cases, thread schedules the threads according to their priority (known as preemptive scheduling).
★ But it is not guaranteed depending on JVM specification which scheduling it chooses.

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**.

The Methods to Prevent a Thread from Execution:
· We can prevent(stop) a Thread execution by using the following methods.
1) yield();
2) join();
3) sleep();

yield()

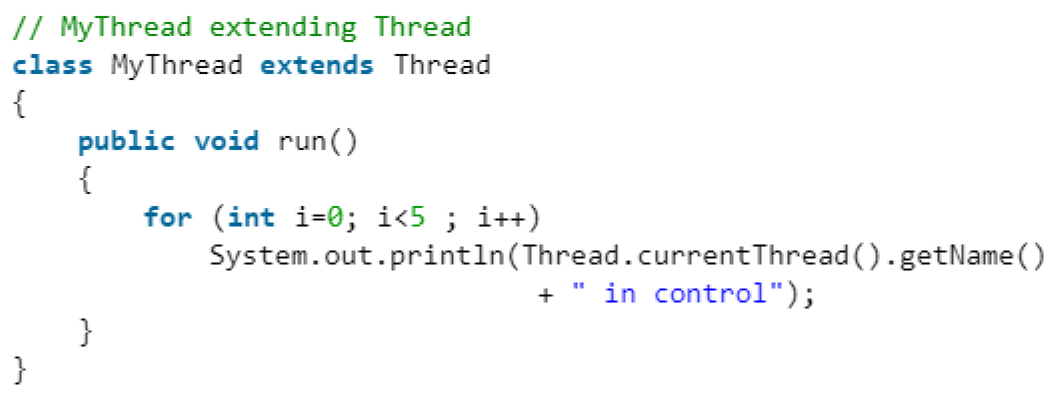
❖ Suppose there are three threads **t1, t2, and t3**. *Thread t1 gets the processor and starts its execution .*
❖ Thread t2 and t3 are in **Ready/Runnable** state.
❖ Completion time for thread t1 is 5 hour and completion time for t2 is 5 minutes. Since t1 will complete its execution after 5 hours, t2 has to wait for 5 hours to just finish the 5 minutes job.

- ❖ In such scenarios where one thread is taking too much time to complete its execution, we need a way to prevent execution of a thread in between if something important is pending.
- ❖ yeild() helps us in doing so.
- ❖ **yield()** basically means that the thread is not doing anything particularly important and if any other threads or processes need to be run, they should run. Otherwise, the current thread will continue to run.

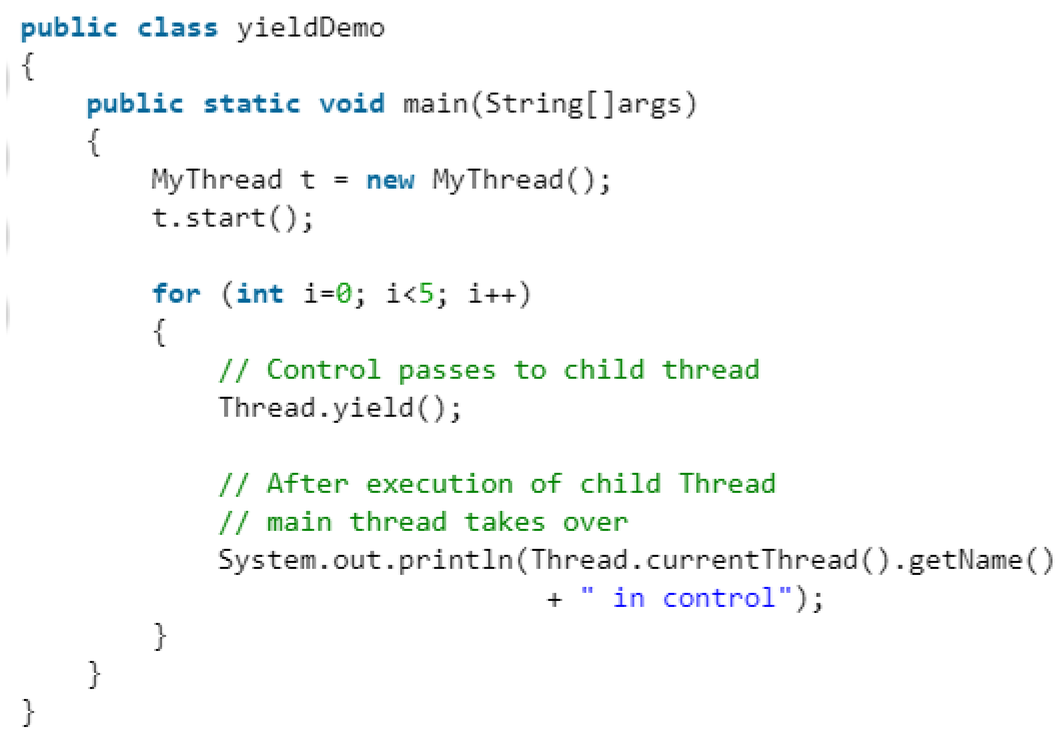
Use of yield method:

- Whenever a thread calls java.lang.Thread.yield method, it gives a hint to the thread scheduler that it is ready to pause its execution. Thread scheduler is free to ignore this hint.
- If any thread executes the yield method , the thread scheduler checks if there is any thread with same or higher priority than this thread. If the processor finds any thread with higher or same priority then it will move the current thread to Ready/Runnable state and give the processor to another thread and if not – current thread will keep executing.

Example code



```
// MyThread extending Thread
class MyThread extends Thread
{
    public void run()
    {
        for (int i=0; i<5 ; i++)
            System.out.println(Thread.currentThread().getName()
                                    + " in control");
    }
}
```



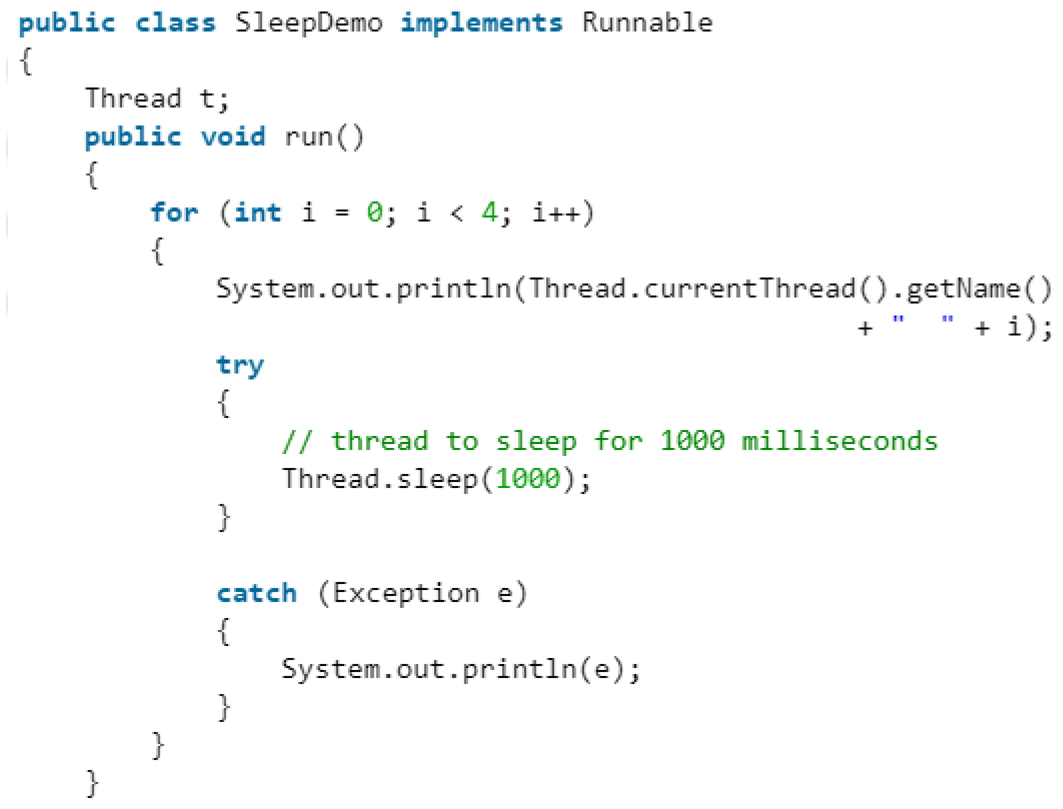
```java
public class yieldDemo
{
    public static void main(String[]args)
    {
        MyThread t = new MyThread();
        t.start();

        for (int i=0; i<5; i++)
        {
            // Control passes to child thread
            Thread.yield();

            // After execution of child Thread
            // main thread takes over
            System.out.println(Thread.currentThread().getName()
                                + " in control");
        }
    }
}
```

Note:

- Once a thread has executed the yield method and there are many threads with the same priority waiting for the processor, then we can't specify which thread will get an execution chance first.
- The thread which executes the yield method will enter in the Runnable state from Running state.
- Once a thread pauses its execution, we can't specify when it will get a chance again; it depends on the thread scheduler.
- Underlying platform must provide support for preemptive scheduling if we are using yield methods

sleep(): This method causes the currently executing thread to sleep for the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers..

```
//  sleep for the specified number of milliseconds
public static void sleep(long millis) throws
InterruptedException

//sleep for the specified number of milliseconds plus nano
seconds
public static void sleep(long millis, int nanos)
                         throws InterruptedException
```

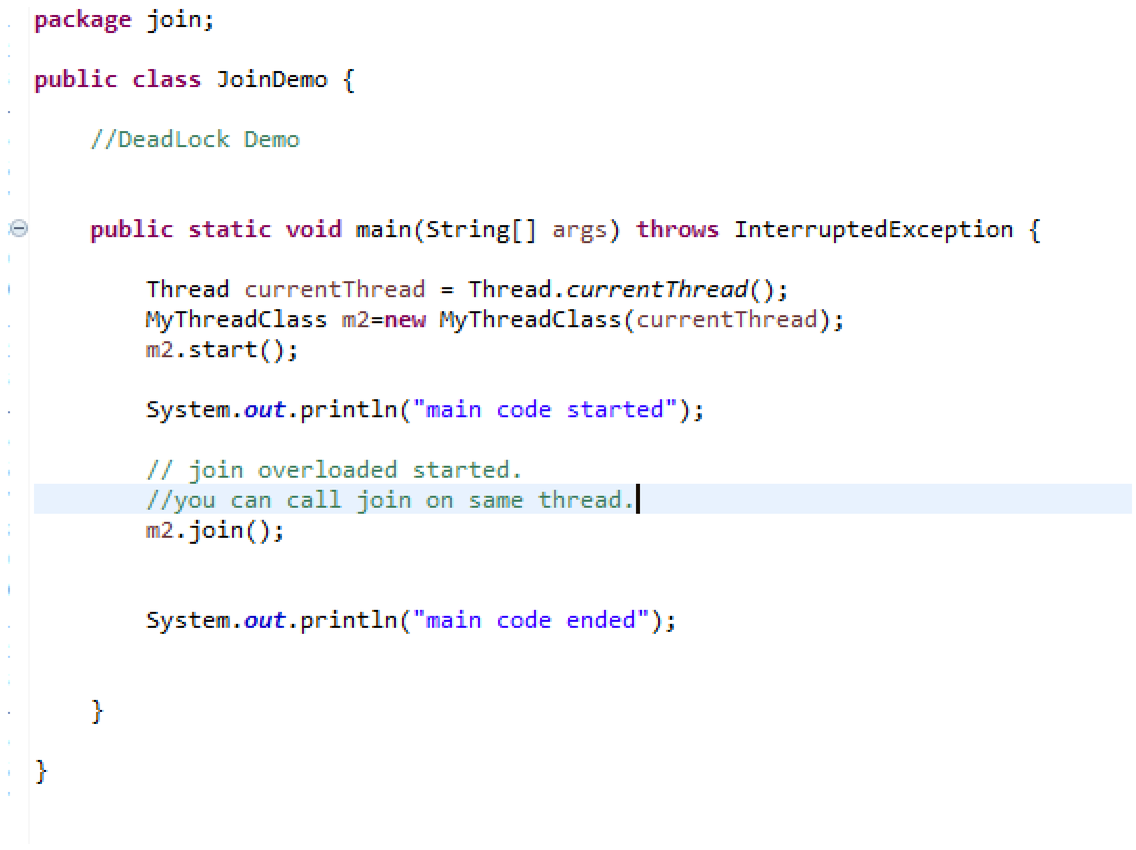


```java
public class SleepDemo implements Runnable
{
    Thread t;
    public void run()
    {
        for (int i = 0; i < 4; i++)
        {
            System.out.println(Thread.currentThread().getName()
                                                    + "  " + i);
            try
            {
                // thread to sleep for 1000 milliseconds
                Thread.sleep(1000);
            }

            catch (Exception e)
            {
                System.out.println(e);
            }
        }
    }
```

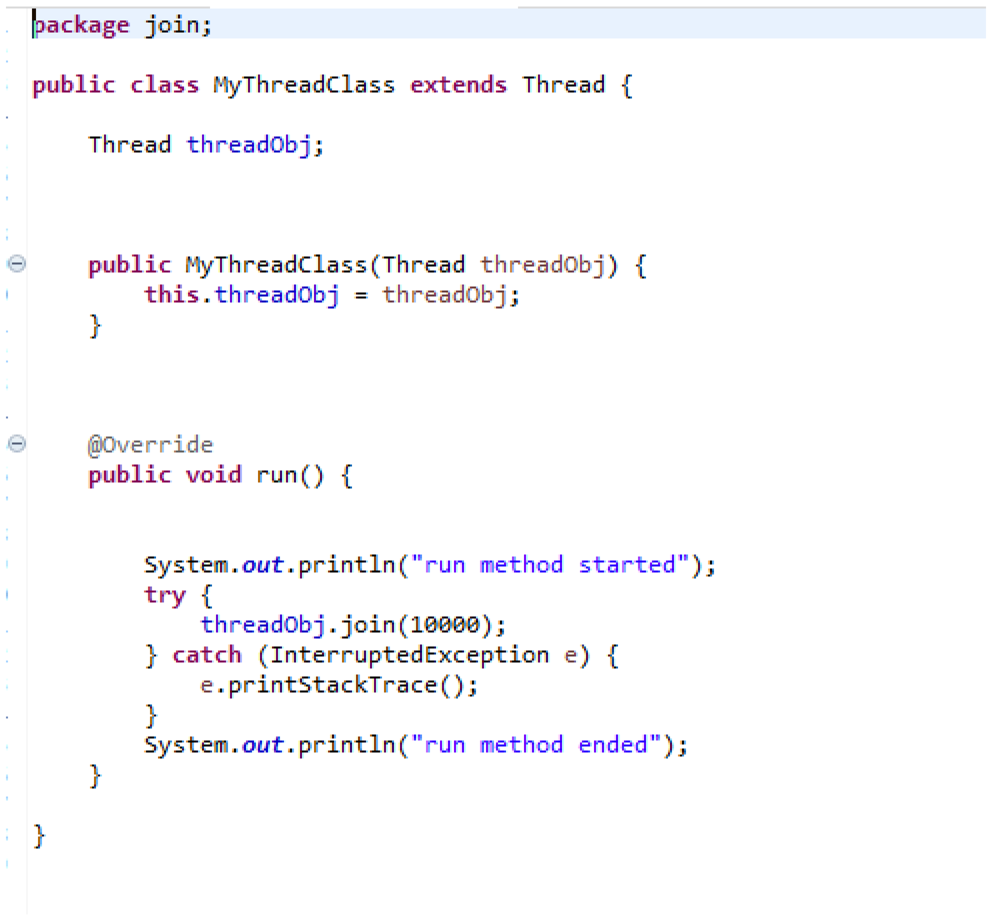
Note:

- Based on the requirement we can make a thread to be in sleeping state for a specified period of time
- Sleep() causes the thread to definitely stop executing for a given amount of time; if no other thread or process needs to be run, the CPU will be idle (and probably enter a power saving mode).

Join() : The join() method of a Thread instance is ***used to join the start of a thread's execution to the end of another thread's execution*** such that **a thread does not start running until another thread ends**. If join() is called on a Thread instance, the currently running thread will block until the Thread instance has finished executing.



```
package join;

public class JoinDemo {

    //DeadLock Demo


    public static void main(String[] args) throws InterruptedException {

        Thread currentThread = Thread.currentThread();
        MyThreadClass m2=new MyThreadClass(currentThread);
        m2.start();

        System.out.println("main code started");

        // join overloaded started.
        //you can call join on same thread.
        m2.join();


        System.out.println("main code ended");


    }

}
```



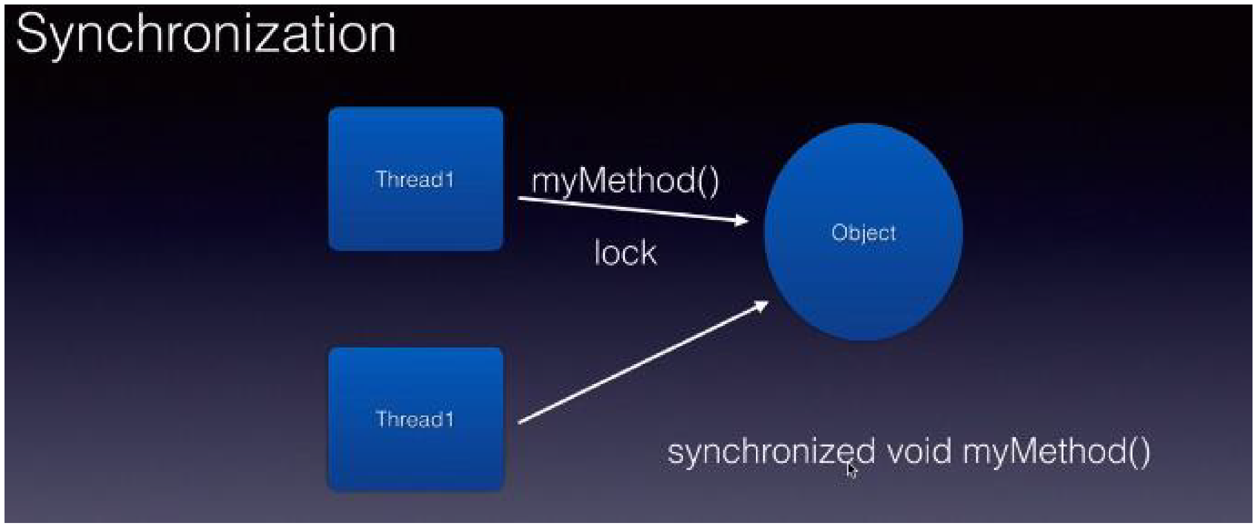
```java
package join;

public class MyThreadClass extends Thread {

    Thread threadObj;


    public MyThreadClass(Thread threadObj) {
        this.threadObj = threadObj;
    }


    @Override
    public void run() {


        System.out.println("run method started");
        try {
            threadObj.join(10000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("run method ended");
    }

}
```

=========================

Synchronization

- Synchronized is the keyword applicable for methods and blocks but not for classes and variables.
- · If a method or block is declared as synchronized then at a time only one Thread is allowed to execute that method or block on the given object.
- · The main advantage of synchronized keywords is we can resolve data inconsistency problems.
- · But the main disadvantage of synchronized keyword is it increases waiting time of the Thread and affects performance of the system.
- · Hence if there is no specific requirement then never recommend to use synchronized keywords.
- · Internally synchronization concept is implemented by using lock concept.
- · Every object in java has a unique lock. Whenever we are using synchronized keywords then only the lock concept will come into the picture.
- ·If a Thread wants to execute any synchronized method on the given object , 1st it has to get the lock of that object.

- Once a Thread gets the lock of that object then it’s allowed to execute any synchronized method on that object. If the synchronized method execution completes then automatically Thread releases lock.
- While a Thread executing any synchronized method the remaining Threads are not allowed to execute any synchronized method on that object simultaneously. But remaining Threads are allowed to execute any non-synchronized method simultaneously. [lock concept is implemented based on object but not based on method]



Example code :

```
package sync;

public class SourceSyncTest {

        public static void main(String[] args) {

                SourceTable s = new SourceTable();

                M1 m1 = new M1();
                m1.s1 = s;
                M2 m2 = new M2();
                m2.s1 = s;

                m1.start();

                m2.start();
        }

}

class M1 extends Thread {

        SourceTable s1;

        @Override
        public void run() {

                s1.tablePrint(5);

        }

}

class M2 extends Thread {

        SourceTable s1;

        @Override
        public void run() {
                s1.tablePrint(10);

        }

}
```

Source table:

```
package sync;

public class SourceTable {

        public static void tablePrint(int tableNo) {
                                for (int i = 0; i < 10; i++) {


                        System.out.println(tableNo+"  *  "+i+" = "+(tableNo*i));
```

```
                }
            }


        }

}
```

For Synchronization.

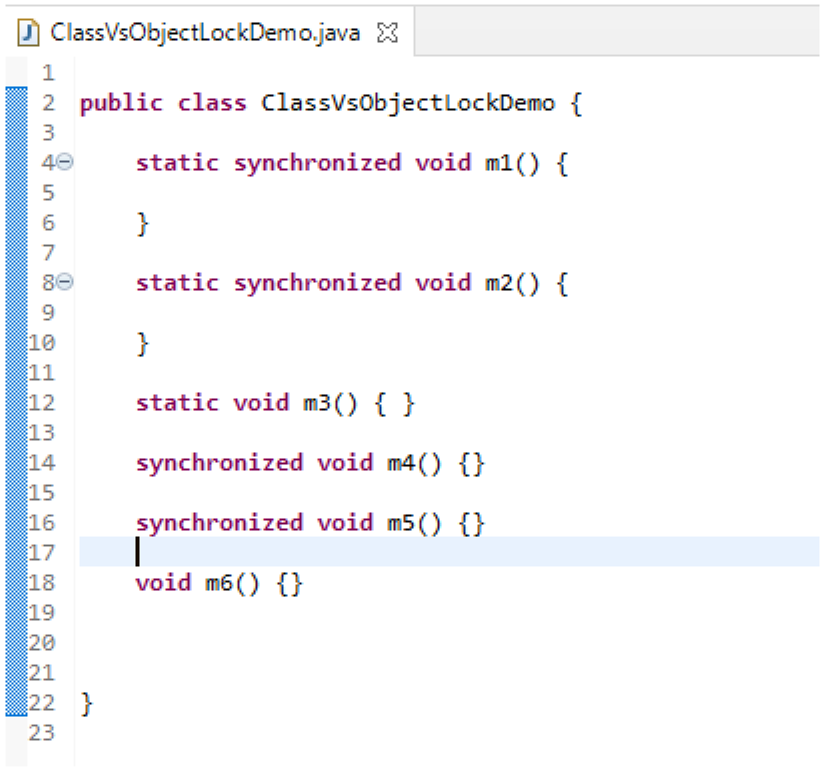
If multiple threads operate on the same java object ,then only synchronization is required ,if each thread is working on a different instance of class then there is no need of synchronization.

```
            SourceTable source1= new SourceTable();
            SourceTable source1 = new SourceTable();


            M1 m1 = new M1();
            m1.s1 = source1;
            M2 m2 = new M2();
            m2.s1 = source1;

            m1.start();

            m2.start();
```

Now both the threads are working on different objects.

Even we make it sync no use as they are operating on individual objects.

Make them as static methods and get class level locks.


ClassVsObjectLockDemo.java

```java

public class ClassVsObjectLockDemo {

    static synchronized void m1() {

    }

    static synchronized void m2() {

    }

    static void m3() { }

    synchronized void m4() {}

    synchronized void m5() {}

    void m6() {}



}

```

T1 ,T2 , T3 , T4 ⇒ 4threads.

T1 acquires class level lock while executing m1 method.

T2 which is intended to execute the m1() method will be in wait state.

T3 which is intended to execute the m2() method will be in wait state.

T4 can execute m3().

I.e : remaining threads can execute normal static methods, normal instance methods.

Class level lock in Java

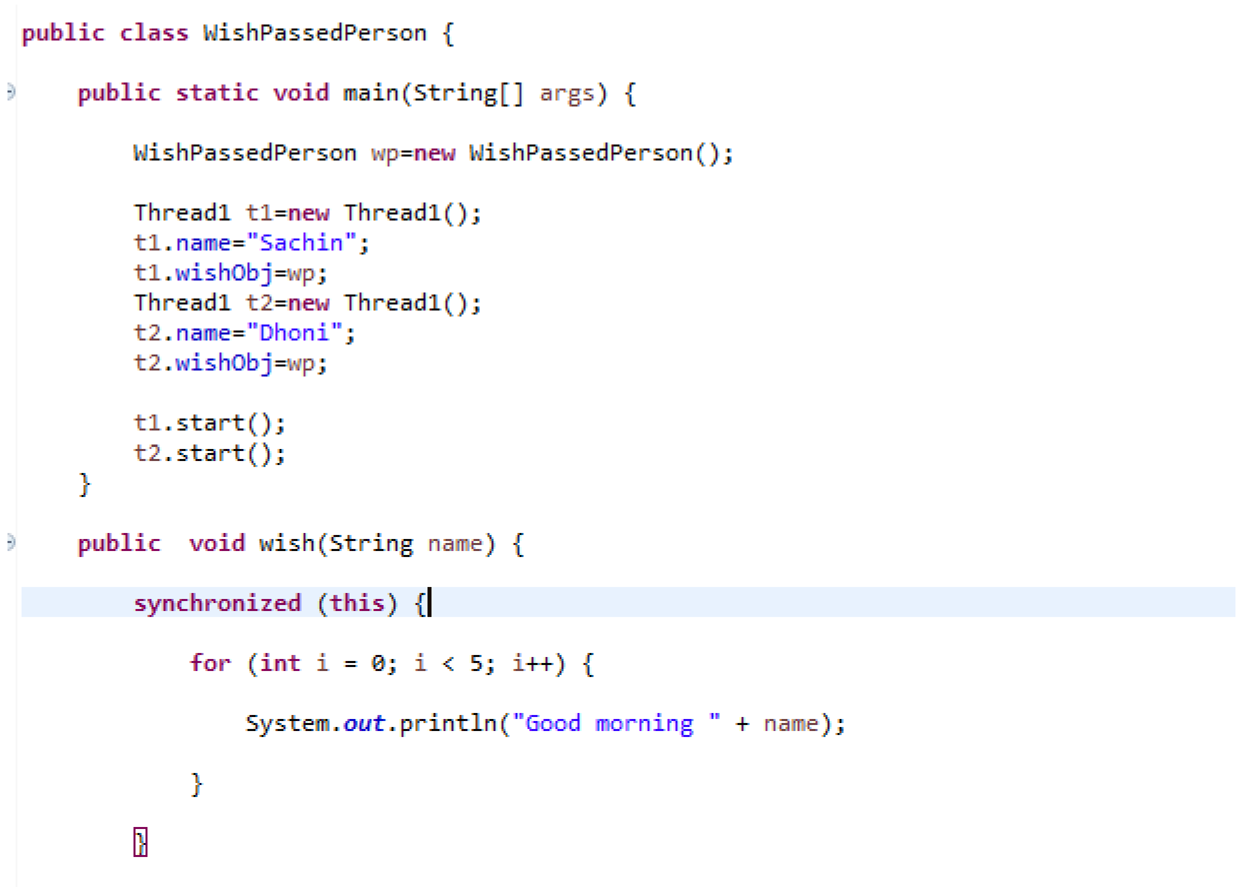
Every class in java has a unique lock called class level lock ,similarly every object in java has object level lock.
When thread is going to execute a static synchronized method/block it requires class level lock.

Class level lock prevents multiple threads to enter in synchronized block in any of all available instances of the class on runtime. This means if in

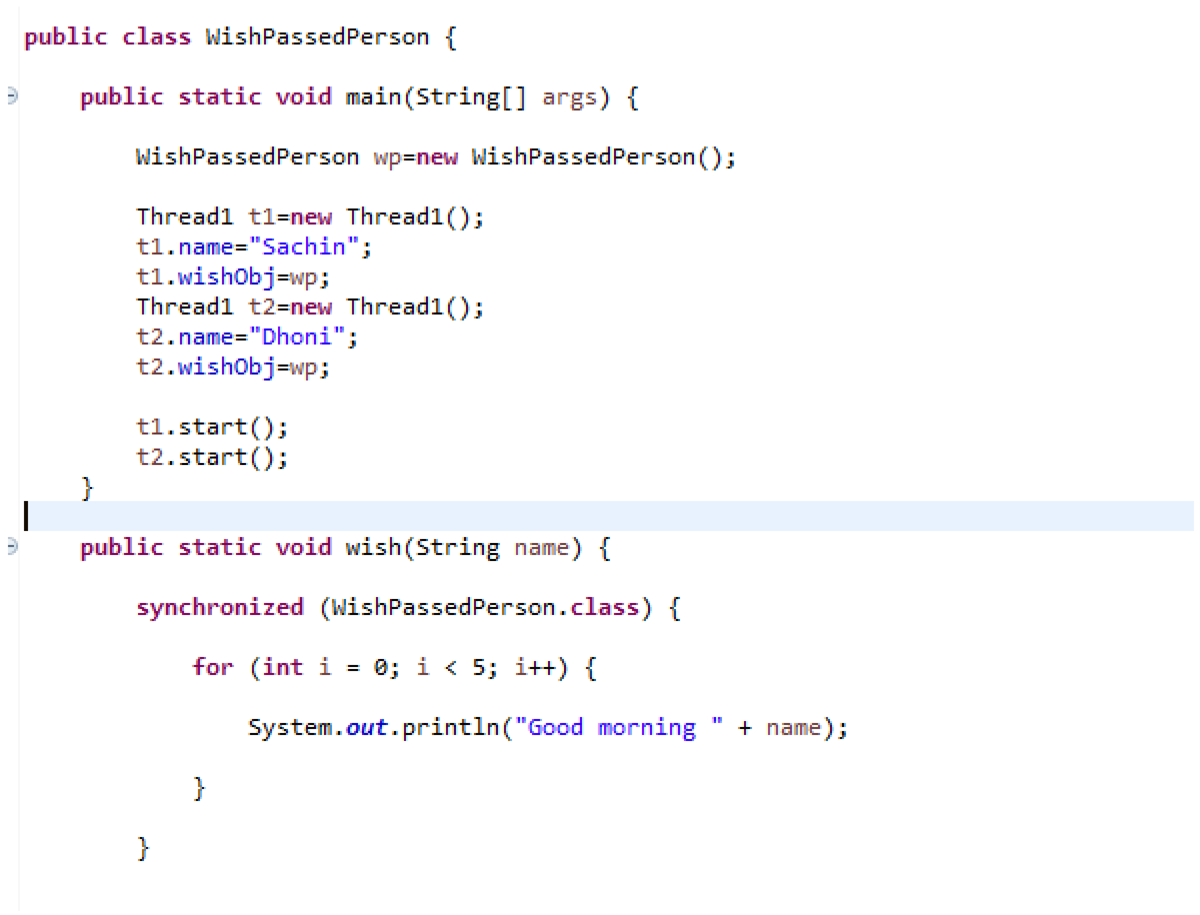
runtime there are 100 instances of DemoClass, then only one thread will be able to execute demoMethod() in any one of instance at a time, and all other instances will be locked for other threads.
Class level locking should always be done to make static data thread safe. As we know that static keywords associate data of methods to class level, so use locking at static fields or methods to make it on class level.

Object level lock in Java

Object level lock is a mechanism when **we want to synchronize a non-static method or non-static code block such that only one thread will be able to execute the code block on a given instance of the class**. This should always be done to make instance level data thread safe.



```
public class WishPassedPerson {

    public static void main(String[] args) {

        WishPassedPerson wp=new WishPassedPerson();

        Thread1 t1=new Thread1();
        t1.name="Sachin";
        t1.wishObj=wp;
        Thread1 t2=new Thread1();
        t2.name="Dhoni";
        t2.wishObj=wp;

        t1.start();
        t2.start();
    }

    public  void wish(String name) {

        synchronized (this) {

            for (int i = 0; i < 5; i++) {

                System.out.println("Good morning " + name);

            }

        }
```


```java
public class WishPassedPerson {

    public static void main(String[] args) {

        WishPassedPerson wp=new WishPassedPerson();

        Thread1 t1=new Thread1();
        t1.name="Sachin";
        t1.wishObj=wp;
        Thread1 t2=new Thread1();
        t2.name="Dhoni";
        t2.wishObj=wp;

        t1.start();
        t2.start();
    }

    public static void wish(String name) {

        synchronized (WishPassedPerson.class) {

            for (int i = 0; i < 5; i++) {

                System.out.println("Good morning " + name);

            }

        }
```

Synchronized block:

· If very few lines of the code required synchronization then it's never recommended to declare the entire method as synchronized we have to enclose those few lines of the code within the synchronized block.

· The main advantage of synchronized block over synchronized method is it reduces waiting time of Thread and improves performance of the system.

Inter Thread communication (wait(),notify(), notifyAll()):

- ❖ Two Threads can communicate with each other by using wait(), notify() and notifyAll() methods.

- · **The Thread which is expecting updation it has to call wait() method** and the **Thread which is performing updation it has to call notify()** method.
- After getting notification the waiting Thread will get those updations.
- wait(), notify() and notifyAll() methods are available in Object class but not in Thread class because Thread can call these methods on any common object.
- start(), join() we can call on only thread objects but wait , notify should be called on any java object by the waiting or notifying threads .
- To call wait(), notify() and notifyAll() methods compulsory the current **Thread should be the owner of that object** that is current Thread should have **lock of that object that is current Thread should be in synchronized** area.
- Hence we can call wait(), notify() and notifyAll() methods only from synchronized area otherwise we will get runtime exception saying **IllegalMonitorStateException.**

Notify vs NotifyAll.

notify() and notifyAll() methods with wait() method are used for communication between the threads. A thread which goes into a waiting state by calling wait() method will be in waiting state until any other thread calls either notify() or notifyAll() method on the same object.

Notifying a thread by JVM : If multiple threads are waiting for the notification and **we use notify() method then only one thread gets the notification** and the *remaining thread has to wait for further notification*. Which thread will get the notification we can't expect because it totally depends upon the JVM. But when we use notifyAll() method then multiple threads get the notification but execution of threads will be performed one by one because thread requires lock and only one lock is available for one object.