CHAPTER:4 : MULTITHREADING

Introduction

- **Multitasking**: Executing several tasks simultaneously is the concept of multitasking.
- There are two types of multitasking's.
  1. Process based multitasking.
  2. Thread based multitasking.

Process based multitasking:

- Executing several tasks simultaneously where each task is a separate independent process such type of multitasking is called process based multitasking.
  Example:
  - While typing a java program in the editor we can able to listen mp3 audio songs at the same time we can download a file from the net all these tasks are independent of each other and executing simultaneously and hence it is Process based multitasking.
  - This type of multitasking is best suitable at "os level".

Thread based multitasking:

- Executing several tasks simultaneously where each task is a separate independent part of the same program, is called Thread based multitasking. And each independent part is called a "Thread".
  - This type of multitasking is best suitable for "programmatic level".
  - When compared with "C++", developing multithreading examples is very easy in java because java provides in built support for multithreading through a rich API (Thread, Runnable,ThreadGroup, ThreadLocal...etc).
  - In multithreading on 10% of the work the programmer is required to do and 90% of the work will be down by java API.
  - The main important application areas of multithreading are:
    1. To implement multimedia graphics.
    2. To develop animations.
    3. To develop video games etc.
    4. To develop web and application servers
    5. Whether it is process based or Thread based the main objective of multitasking is to improve performance of the system by reducing response time.

CHAPTER:4 : MULTITHREADING

The ways to define instantiate and start a new Thread:

- We can define a Thread in the following 2 ways.
  1. By extending Thread class.
  2. By implementing Runnable interface.

(1) Defining a Thread by extending "Thread class":

Example: Write a program to create a child thread to print integer numbers 1 to 10

```
class MyThread extends Thread
{
    public void run()
    {
        for (int i = 1; i <= 10; i++)
        {
            System.out.println(i);
        }
    }
}
class ChildThreadExample
{
    public static void main(String[] args)
    {
        MyThread t1= new MyThread();//Instiation of Thread
        t1.start(); // Start the child thread
    }
}
```

Job of Thread

Defining A Thread

Example: Understanding the Execution of thread

```
class MyThread extends Thread
{
    public void run()
    {
        for (int i = 1; i <= 5; i++)
        {
            System.out.println("Child Thread: " + i);
        }
    }
}
class Main
```

```
{
    public static void main(String[] args)
    {
        MyThread t1= new MyThread();
        t1.start(); // Start the child thread
        for (int i = 1; i <= 5; i++)
        {
            System.out.println("Main Thread: " + i);
        }
    }
}
```

- We cannot except exact output The following are various possible outputs for the above program.

Output:1	**Output:2**	**Output:3**
Main Thread: 1	Child Thread: 1	Main Thread: 1
Child Thread: 1	Main Thread: 1	Main Thread: 2
Main Thread: 2	Child Thread: 2	Main Thread: 3
Child Thread: 2	Child Thread: 3	Main Thread: 4
Main Thread: 3	Child Thread: 4	Main Thread: 5
Child Thread: 3	Child Thread: 5	Child Thread: 1
Main Thread: 4	Main Thread: 2	Child Thread: 2
Main Thread: 5	Main Thread: 3	Child Thread: 3
Child Thread: 4	Main Thread: 4	Child Thread: 4
Child Thread: 5	Main Thread: 5	Child Thread: 5

Case 1: 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 it is the JVM vendor dependent hence in multithreading examples we can't expect exact execution order and exact output.

Case 2: Difference between t1.start() and t1.run() methods.

- In the case of t1.start() a new Thread will be created which is responsible for the execution of run() method.
- But in the case of t1.run() no new Thread will be created and run() method will be executed just like a normal method by the main Thread.

- In the above program if we are replacing t1.start() with t1.run() the following is the output.
- Entire output produced by only main Thread.

Example: Difference between t1.start() and t1.run() methods

```
class MyThread extends Thread
{
    public void run()
    {
        for (int i = 1; i <= 5; i++)
        {
            System.out.println("Child Thread: " + i);
        }
    }
}
class Main
{
    public static void main(String[] args)
    {
        MyThread t1= new MyThread();
        //t1.start(); // Start the child thread
        t1.run();
        for (int i = 1; i <= 5; i++)
        {
            System.out.println("Main Thread: " + i);
        }
    }
}
```

Output:

```
Child Thread: 1
Child Thread: 2
Child Thread: 3
Child Thread: 4
Child Thread: 5
Main Thread: 1
Main Thread: 2
Main Thread: 3
Main Thread: 4
Main Thread: 5
```

Case 3: Importance of Thread class start() method.

- For every Thread the required mandatory activities like registering the Thread with Thread Scheduler will takes care by Thread class start() method and

programmer is responsible just to define the job of the Thread inside run() method.

- That is start() method acts as best assistant to the programmer.

Example:

```
start()
{
1. Register Thread with Thread Scheduler
2. All other mandatory low level activities.
3. Invoke or calling run() method.
}
```

- We can conclude that without executing Thread class start() method there is no chance of starting a new Thread in java. Due to this start() is considered as heart of multithreading.

Case 4: If we are not overriding run() method:

- If we are not overriding run() method then Thread class run() method will be executed which has empty implementation and hence we won't get any output.

Example: Not overriding of run() method

```
class MyThread extends Thread
{

}
class Main
{
    public static void main(String[] args)
    {
        MyThread t1= new MyThread();
        t1.start(); // Start the child thread
    }
}
```

- It is highly recommended to override run() method. Otherwise don't go for multithreading concept.

Case 5: Overloding of run() method.

- We can overload run() method but Thread class start() method always invokes noargument run() method the other overload run() methods we have to call explicitly then only it will be executed just like normal method.

Example: Overloding of run() method

```
class MyThread extends Thread
{
    @Override
    public void run()
    {
        System.out.println("No arg Method");
    }
    public void run(int i)
    {
        System.out.println("Int arg Method");
    }
}
class Main
{
    public static void main(String[] args)
    {
        MyThread t1= new MyThread();
        t1.start(); // Start the child thread
    }
}
output: No arg Method
```

Case 6: overriding of start() method:

- If we override start() method then our start() method will be executed just like a normal method call and no new Thread will be started.

Example:1: overriding of start() method

```
class MyThread extends Thread
{
    public void start()
    {
        System.out.println("start method");
    }
    public void run()
    {
        System.out.println("run method");
    }
}
class Main
```

```
{
    public static void main(String[] args)
    {
        MyThread t1= new MyThread();
        t1.start(); // Start the child thread
        System.out.println("main method");
    }
}
output:
start method
main method
```

- Entire output produced by only main Thread.
- Note : It is never recommended to override start() method.

Example:2: overriding of start() method

```
class MyThread extends Thread
{
    public void start()
    {
        super.start();
        System.out.println("start method");
    }
    public void run()
    {
        System.out.println("run method");
    }
}
class Main
{
    public static void main(String[] args)
    {
        MyThread t1= new MyThread();
        t1.start(); // Start the child thread
        System.out.println("main method");
    }
}
output:
start method
main method
run method
```

- Here t1.start() calls the overridden start() method of MyThread.
- Inside the overridden start(): super.start() is called → This creates a new thread and invokes the run() method in that new thread.

- After the new thread has been started, the run() method will invoked.
- The output may vary slightly due to thread scheduling However, it's not guaranteed run method might print before or after main method, depending on how the JVM schedules threads.

Case 7:Restarting the thread again

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

Example: Restarting thread again

```
class MyThread extends Thread
{
    public void run()
    {
        System.out.println("run method");
    }
}
class Main
{
    public static void main(String[] args)
    {
        MyThread t1= new MyThread();
        t1.start(); // Start the child thread
        System.out.println("main method");
        t1.start();
    }
}
output:
main method
run method
Exception in thread "main" java.lang.IllegalThreadStateException
      at java.lang.Thread.start(Thread.java:710)
      at Main.main(PbEx1.java:15)
```

(2) Defining a Thread by implementing Runnable interface:

- We can define a Thread even by implementing Runnable interface also.Runnable interface present in java.lang.pkg and contains only one method **public abstract void run(); ,** So it is necessary to override the run method while defining the thread.

Example: Understanding Execution of Thread

```
class MyThread implements Runnable
{
    public void run()//cumpulsory
    {
        for (int i=1 ; i<=5 ;i++)
        {
            System.out.println("child Thread: "+i);
        }
    }
}
class Main
{
    public static void main(String[] args)
    {
        MyThread t1= new MyThread();
        Thread t2=new Thread(t1);//here t1 is a Target Runnable
        t2.start();// Start the child thread
        for (int i=1 ; i<=5 ;i++)
        {
            System.out.println("Main Thread: "+i);
        }

    }
}
```

Defining A Thread

Job of Thread

- We cannot except exact output The following are various possible outputs for the above program.

Output:1
Main Thread: 1
Child Thread: 1
Main Thread: 2
Child Thread: 2
Main Thread: 3
Child Thread: 3
Main Thread: 4
Main Thread: 5
Child Thread: 4
Child Thread: 5

Output:2
Child Thread: 1
Main Thread: 1
Child Thread: 2
Child Thread: 3
Child Thread: 4
Child Thread: 5
Main Thread: 2
Main Thread: 3
Main Thread: 4
Main Thread: 5

Output:3
Main Thread: 1
Main Thread: 2
Main Thread: 3
Main Thread: 4
Main Thread: 5
Child Thread: 1
Child Thread: 2
Child Thread: 3
Child Thread: 4
Child Thread: 5

Case study:

Example:

```
class MyThread implements Runnable
{
    public void run()
    {
            System.out.println("child Thread:");
    }
}
class Main
{
    public static void main(String[] args)
    {
        MyThread t1= new MyThread();
        Thread t2=new Thread();
        Thread t3=new Thread(t1);

        System.out.println("Main Thread");
    }
}
```

Case 1: t2.start():

- A new Thread will be created which is responsible for the execution of Thread class run() method.

Case 2: t2.run():

- No new Thread will be created but Thread class run() method will be executed just like a normal method call.

Case 3: t3.start():

- New Thread will be created which is responsible for the execution of MyThread run() method.

Case 4: t3.run():

- No new Thread will be created and MyThread run() method will be executed just like a normal method call.

Case 5: t1.start():

- We will get compile time error saying start()method is not available in MyThread class.

Case 6: t1.run():

- No new Thread will be created and MyThread class run() method will be executed just like a normal method call.

Best approach to define a Thread:

- Among the 2 ways of defining a Thread, implements Runnable approach is always recommended.
- In the 1st approach our class should always extends 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 class also. Hence implements Runnable mechanism is recommended to define a Thread.

Thread class constructors:

1. Thread t=new Thread();
2. Thread t=new Thread(Runnable r);
3. Thread t=new Thread(String name);
4. Thread t=new Thread(Runnable r,String name);

Getting and setting name of a Thread:

- Every Thread in java has some name it may be provided explicitly by the programmer or automatically generated by JVM.
- Thread class defines the following methods to get and set name of a Thread.
- Methods:

1. **public final String getName()**
2. **public final void setName(String name)**

Example:1

```
class MyThread extends Thread
{

}
class ThreadDemo
{
    public static void main(String[] args)
    {
        System.out.println(Thread.currentThread().getName());
        Thread.currentThread().setName("LJ");
        System.out.println(Thread.currentThread().getName());
        //System.out.println(10/0);
        MyThread t=new MyThread();
        System.out.println(t.getName());//Thread-0
        t.setName("hridank");
        System.out.println(t.getName());
    }
}
Output:
main
LJ
Thread-0
Hridank
```

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

Example:2

```
class MyThread extends Thread
{
    public MyThread(String name) {
        super(name);
    }
}
class ThreadDemo
{
    public static void main(String[] args)
    {
        MyThread t=new MyThread("Hridank");
        System.out.println(t.getName());//Hridank
    }
}
```

Example:3

```
class MyThread extends Thread
{
    @Override
    public void run() {
        Thread t=Thread.currentThread();
        t.setName("Hridank");
        //Thread.currentThread().setName("Hridank");
        System.out.println(t.getName());
    }
}
class ThreadDemo
{
    public static void main(String[] args)
    {
        MyThread t=new MyThread();
        t.start();
        /*for(int i=0;i<100;i++)
        {
            System.out.println("Main");
        }*/
        System.out.println(t.getName());
    }
}
```

Example:4

```
class MyThread implements Runnable
{
    public void run() {
        System.out.println("Child");
    }
}
class ThreadDemo{
    public static void main(String[] args)
    {
        MyThread t=new MyThread();
        Thread t1=new Thread(t,"Hridank");
        t1.start();
        System.out.println(t1.getName());
    }
}
```

Thread Priorities

- Every Thread in java has some priority it may be default priority generated by JVM (or) explicitly provided by the programmer.
- The valid range of Thread priorities is 1 to 10[but not 0 to 10] where 1 is the least
- priority and 10 is highest priority.
- Thread class defines the following constants to represent some standard
- priorities.
  1. Thread. MIN_PRIORITY----------1
  2. Thread. MAX_PRIORITY----------10
  3. Thread. NORM_PRIORITY--------5
- There are no constants like **Thread.LOW_PRIORITY, Thread.HIGH_PRIORITY**
- Thread scheduler uses these priorities while allocating CPU.
- The Thread which is having highest priority will get chance for 1st execution.
- If 2 Threads having the same priority then we can't expect exact execution order it depends on Thread scheduler whose behavior is vendor dependent.
- We can get and set the priority of a Thread by using the following methods.
  1. public final int getPriority()
  2. public final void setPriority(int newPriority);
- The allowed values are 1 to 10 otherwise we will get runtime exception saying **"IllegalArgumentException".**

Default priority:

- The default priority only for the main Thread is 5. But for all the remaining Threads the default priority will be inheriting from parent to child. That is whatever the priority parent has by default the same priority will be for the child also.

Example 1:

```
class MyThread extends Thread
{
}
class ThreadDemo
{
    public static void main(String[] args)
    {

System.out.println(Thread.currentThread().getPriority());
```

```
        MyThread t=new MyThread();
        System.out.println(t.getPriority());
    }
}
output:
5
5
```

Example 2:

```
class MyThread extends Thread
{
}
class ThreadDemo
{
    public static void main(String[] args)
    {
        Thread.currentThread().setPriority(9);

System.out.println(Thread.currentThread().getPriority());
        MyThread t=new MyThread();
        System.out.println(t.getPriority());
    }
}
output:
9
9
```

Example:Understanding the concept of priority

```
class MyThread extends Thread
{
    public void run()
    {
        for(int i=1;i<=5;i++)
        {
            System.out.println("child thread: "+i);
        }
    }
}
class ThreadPriorityDemo
{
    public static void main(String[] args)
    {
```

```
        MyThread t=new MyThread();

        //t.setPriority(10); //----> 1
        t.start();
        for(int i=1;i<=5;i++)
        {
            System.out.println("main thread: "+i);
        }
    }
}
```

- If we are commenting line 1 then both main and child Threads will have the same priority and hence we can't expect exact execution order.
- If we are not commenting line 1 then **child Thread has the priority 10** and **main Thread has the priority 5** hence **child Thread will get chance for execution** and after completing child Thread main Thread will get the chance in this the output is:

Output

Child Thread: 1
Child Thread: 2
Child Thread: 3
Child Thread: 4
Child Thread: 5
Main Thread: 1
Main Thread: 2
Main Thread: 3
Main Thread: 4
Main Thread: 5

- But we cannot get the above output and we cannot expect exact output because, Some operating **systems(like windowsXP)** may not provide proper support for Thread priorities. We have to **install separate bats** provided by vendor to provide support for priorities.

The Methods to Prevent a Thread from Execution:

- We can prevent(stop) a Thread execution by using the following methods.

    1. yield();
    2. sleep();
    3. join();

(1) yield()

- Yield() method causes "to pause current executing Thread for giving the chance of remaining waiting Threads of same priority".
- If all waiting Threads have the low priority or if there is no waiting Threads then the same Thread will be continued its execution.
- If several waiting Threads with same priority available then we can't expect exact which Thread will get chance for execution.
- The Thread which is yielded when it get chance once again for execution is depends on mercy of the Thread scheduler.
- **public static native void yield();**

Example: Execution of yield()

```
class MyThread extends Thread
{
    public void run()
    {
        for(int i=1;i<=5;i++)
        {
            Thread.yield();
            System.out.println("child thread: "+i);
        }
    }
}
class ThreadYieldDemo
{
    public static void main(String[] args)
    {
        MyThread t=new MyThread();
        t.start();
```

```
        for(int i=1;i<=5;i++)
        {
            System.out.println("main thread: "+i);
        }
    }
}
expected output:
Main Thread: 1
Main Thread: 2
Main Thread: 3
Main Thread: 4
Main Thread: 5
Child Thread: 1
Child Thread: 2
Child Thread: 3
Child Thread: 4
Child Thread: 5
```

- In the above program child Thread always calling yield() method and hence main Thread will get the chance more number of times for execution. Hence the chance of completing the main Thread first is high.
- **Note : Some operating systems may not provide proper support for yield() method.**

(2) Sleep() method:

- If a Thread don't want to perform any operation for a particular amount of time then we should go for sleep() method.
  1. public static native void sleep(long ms) throws InterruptedException
  2. public static void sleep(long ms,int ns)throws InterruptedException

Example: Sleep demo

```
public class SleepDemo
{
    public static void main(String[] args) throws
InterruptedException
    {
        System.out.println("J");
        Thread.sleep(1000);

        System.out.println("A");
```

```
        Thread.sleep(1000);

        System.out.println("V");
        Thread.sleep(1000);

        System.out.println("A");
    }
}
```

Interrupting a Thread:

- How a Thread can interrupt another thread ?
- If a Thread can interrupt a sleeping or waiting Thread by using interrupt() (break off) method of Thread class.
- **public void interrupt();**

Example: Interrupting A thread

```
class MyThread extends Thread{
    public void run()
    {
        try{
            for(int i=0;i<5;i++)
            {
                System.out.println("i am lazy Thread :"+i);
                Thread.sleep(2000);
            }
        }
        catch (InterruptedException e){
            System.out.println("i got interrupted");
        }
    }
}
class ThreadInterruptDemo
{
    public static void main(String[] args)
    {
        MyThread t=new MyThread();
        t.start();
        //t.interrupt(); //--->1
        System.out.println("end of main thread");
    }
}
```

- If we are commenting line 1 then main Thread won't interrupt child Thread and hence child Thread will be continued until its completion.
- If we are not commenting line 1 then main Thread interrupts child Thread and hence child Thread won't continued until its completion in this case the output is:
  end of main thread
  i am lazy Thread: 0
  Iigot interrupted

Note:

- Whenever we are calling interrupt() method we may not see the effect immediately, if the target Thread is in sleeping or waiting state it will be interrupted immediately.
- If the target Thread is not in sleeping or waiting state then interrupt call will wait until target Thread will enter into sleeping or waiting state. Once target Thread entered into sleeping or waiting state it will effect immediately.
- In its lifetime if the target Thread never entered into sleeping or waiting state then there is no impact of interrupt call simply interrupt call will be wasted.

(3) join()

- If a Thread wants to wait until completing some other Thread then we should go for join() method.
- Example: If a Thread t1 executes t2.join() then t1 should go for waiting state until completing t2.
- The following are the three overloaded join() method

  1. public final void join()throws InterruptedException

  2. public final void join(long ms) throws InterruptedException

  3. public final void join(long ms,int ns) throws InterruptedException

- Every join() method throws InterruptedException, which is checked exception hencecompulsory we should handle either by try catch or by throws keyword.Otherwise we will get compiletime error.

Example:

```
class MyThread extends Thread
{
    public void run()
    {
        for(int i=0;i<5;i++)
        {
            System.out.println("Child Thread");
        }
    }
}
class ThreadJoinDemo
{
    public static void main(String[] args)throws
InterruptedException
    {
        MyThread t=new MyThread();
        t.start();
        //t.join(); //--->1
        for(int i=0;i<5;i++)
        {
            System.out.println("Main Thread");
        }
    }
}
```

- If we are commenting line 1 then both Threads will be executed simultaneously and we can't expect exact execution order.
- If we are not commenting line 1 then main Thread will wait until completing child Thread in this the output is **Child Thread** 5 times followed by **Main Thread** 5 times.

Waiting of child Thread untill completing main Thread :

Example

```
class MyThread extends Thread
{
    static Thread mt;

    public static void setMt(Thread mt1) {
        mt = mt1;
```

```
    }
    public void run()
    {
        try
        {
            mt.join();
        }
        catch (InterruptedException e){}
        for(int i=0;i<5;i++)
        {
            System.out.println("Child Thread");
        }
    }
}
class ThreadJoinDemo
{
    public static void main(String[] args)throws
InterruptedException
    {
        Thread mt=Thread.currentThread();
        MyThread t=new MyThread();
        t.setMt(mt);
        t.start();
        for(int i=0;i<5;i++)
        {
            Thread.sleep(2000);
            System.out.println("Main Thread");
        }
    }
}
```

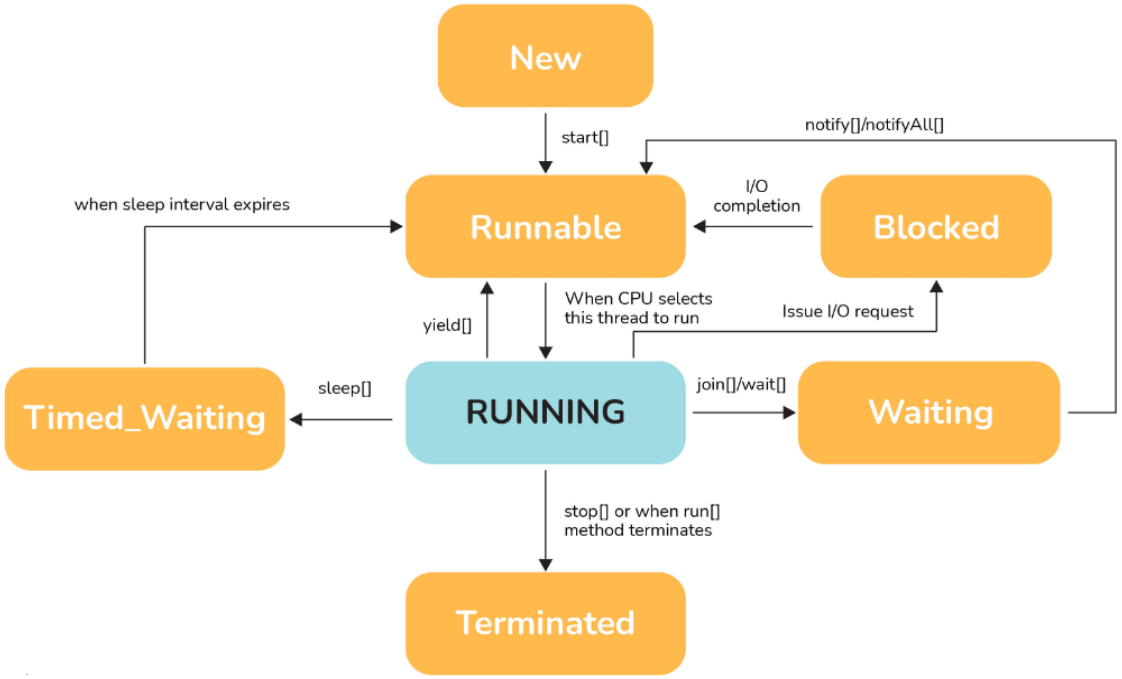
Life Cycle of Thread

There are multiple states of the thread in a lifecycle as mentioned below:

- **New Thread:** When a new thread is created, it is in the new state. The thread has not yet started to run when the thread is in this state. When a thread lies in the new state, its code is yet to be run and has not started to execute.
- **Runnable State:** A thread that is ready to run is moved to a runnable state. In this state, a thread might actually be running or it might be ready to run at any instant of time. It is the responsibility of the thread scheduler to give the thread, time to run. A multi-threaded program allocates a fixed amount of time to each individual thread. Each and every thread get a small amount of time to run. After

running for a while, a thread pauses and gives up the CPU so that other threads can run.

- **Blocked:** The thread will be in blocked state when it is trying to acquire a lock but currently the lock is acquired by the other thread. The thread will move from the blocked state to runnable state when it acquires the lock.
- **Waiting state:** The thread will be in waiting state when it calls wait() method or join() method. It will move to the runnable state when other thread will notify or that thread will be terminated.
- **Timed Waiting:** A thread lies in a timed waiting state when it calls a method with a time-out parameter. A thread lies in this state until the timeout is completed or until a notification is received. For example, when a thread calls sleep or a conditional wait, it is moved to a timed waiting state.
- **Terminated State:** A thread terminates because of either of the following reasons:
  - Because it exits normally. This happens when the code of the thread has been entirely executed by the program.
  - Because there occurred some unusual erroneous event, like a segmentation fault or an unhandled exception.



Thread Life cycle

Synchronization

- Synchronized is the keyword applicable for methods and blocks but not for classes and variables.
- If a method or block declared as the synchronized then at a time only one Thread is allow to execute that method or block on the given object.
- The main advantage of synchronized keyword is we can resolve date inconsistency problems.
- But the main disadvantage of synchronized keyword is it increases waiting time of the Thread and effects performance of the system. Hence if there is no specific requirement then never recommended to use synchronized keyword. Internally synchronization concept is implemented by using lock concept.
- Every object in java has a unique lock. Whenever we are using synchronized keyword then only 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 got the lock of that object then it's allow 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 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 onmethod].

Example

```
class Display
{
    public synchronized void wish(String name)
    {
        for(int i=0;i<5;i++)
        {
            System.out.print("good morning:");
            try
            {
                Thread.sleep(1000);
            }
            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 d1=new Display();
        MyThread t1=new MyThread(d1,"dhoni");
        MyThread t2=new MyThread(d1,"yuvaraj");
        t1.start();
        t2.start();
    }
}
```

- If we are not declaring wish() method as synchronized then both Threads will be executed simultaneously and we will get irregular output.

Output:

good morning:good morning:yuvaraj
good morning:dhoni
good morning:yuvaraj
good morning:dhoni
good morning:yuvaraj
good morning:dhoni
good morning:yuvaraj
good morning:dhoni
good morning:yuvaraj
dhoni

- If we declare wish()method as synchronized then the Threads will be executed one byone that is until completing the 1st Thread the 2nd Thread will wait in this case we will get regular output which is nothing but

Output:

good morning:dhoni
good morning:dhoni
good morning:dhoni
good morning:dhoni
good morning:dhoni
good morning:yuvaraj
good morning:yuvaraj
good morning:yuvaraj
good morning:yuvaraj
good morning:yuvaraj

Case Study: 1

Example

```
class Display
{
    public synchronized void wish(String name)
    {
        for(int i=0;i<5;i++)
        {
            System.out.print("good morning:");
            try
            {
                Thread.sleep(1000);
            }
            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 d1=new Display();
        Display d2=new Display();
        MyThread t1=new MyThread(d1,"dhoni");
        MyThread t2=new MyThread(d2,"yuvaraj");
        t1.start();
        t2.start();
    }
}
```

- Even though we declared wish() method as synchronized but we will get irregular output in this case, because both Threads are operating on different objects.
- Conclusion : If multiple threads are operating on multiple objects then there is no impact of Syncronization. If multiple threads are operating on same java objects then syncronized concept is required(applicable).

Class level lock:

- Every class in java has a unique lock. If a Thread wants to execute a static synchronized method then it required class level lock.
- Once a Thread got class level lock then it is allow to execute any static synchronized method of that class.
- While a Thread executing any static synchronized method the remaining Threads are not allow to execute any static synchronized method of that class simultaneously.
- But remaining Threads are allowed to execute normal synchronized methods, normal static methods, and normal instance methods simultaneously.
- Class level lock and object lock both are different and there is no relationship between these two.

Example:

```
class Display
{
    public static synchronized void wish(String name)
    {
        for(int i=0;i<5;i++)
        {
            System.out.print("good morning:");
            try
            {
                Thread.sleep(1000);
            }
            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 d1=new Display();
        Display d2=new Display();
        MyThread t1=new MyThread(d1,"dhoni");
        MyThread t2=new MyThread(d2,"yuvaraj");
        t1.start();
        t2.start();
    }
}
```

Example: Movie ticket booking using Synchronized Method

```
class MovieTicketBooking
{
    private int availableSeats = 5;

    // Synchronized method to ensure thread safety
    public synchronized void bookTicket(String userName, int
seatsRequested)
    {
        System.out.println(userName + " is trying to book " +
seatsRequested + " seat(s).");

        if (seatsRequested <= availableSeats)
        {
            System.out.println("Seats available. Booking for " +
userName + " in progress...");
            try
            {
                Thread.sleep(1000); // Simulate delay
            } catch (InterruptedException e)
            {
                e.printStackTrace();
            }
            availableSeats -= seatsRequested;
            System.out.println("Booking successful for " + userName +
". Remaining seats: " + availableSeats);
        }
        else
        {
            System.out.println("Booking failed for " + userName + ".
Not enough seats.");
        }
    }
}

class User extends Thread
{
    private MovieTicketBooking bookingSystem;
    private int seats;

    public User(MovieTicketBooking bookingSystem, String name, int
seats) {
        super(name);
        this.bookingSystem = bookingSystem;
        this.seats = seats;
    }


    public void run()
```

```
    {
        bookingSystem.bookTicket(getName(), seats);
    }
}

class TicketBookingApp
{
    public static void main(String[] args)
    {
        MovieTicketBooking bookingSystem = new MovieTicketBooking();

        User user1 = new User(bookingSystem, "Alice", 2);
        User user2 = new User(bookingSystem, "Bob", 2);
        User user3 = new User(bookingSystem, "Charlie", 2);

        user1.start();
        user2.start();
        user3.start();
    }
}
```

Synchronized block

- If very few line of the code is required to be synchronized then its never recommended to declare entire method synchronized enclose those few lines of the code with in synchronized block
- The main Advantage of synchronized block over synchronized method is it reduces waiting time of Thread and improves performance of system

Example: To get the lock of current object we can declare the synchronized block as follows:
synchronized(this){}

```
class Display
{
    public void wish(String name)
    {
        ;;;;;;;;;;;;;;;//1 lack line of code
        synchronized(this)
        {
            for(int i=0;i<5;i++)
            {
                System.out.print("Good Morning:");
                try {
                    Thread.sleep(1000);
                } catch (Exception e) {
```

```
                    // TODO: handle exception
                }
                System.out.println(name);
            }
        }
        ;;;;;;;;;;;;;;;//1 lack line of code
    }
}
class Mythread extends Thread
{
    Display d;
    String name;
    public Mythread(Display d,String name)
    {
        this.d = d;
        this.name = name;
    }
    public void run()
    {
        d.wish(name);
    }
}
class SyncDemo1
{
    public static void main(String[] args) {
        Display d=new Display();
        Mythread t1=new Mythread(d,"Dhoni");
        Mythread t2=new Mythread(d,"yuvi");
        t1.start();
        t2.start();
    }
}
```

Example: To get the class level lock we can declare the synchronized block as follows
//synchronized(ClassName.class){}

```
class Display
{
    public void wish(String name)
    {
        ;;;;;;;;;;;;;;;//1 lack line of code
        synchronized(Display.class)
        {
            for(int i=0;i<5;i++)
            {
```

```
                System.out.print("Good Morning:");
                try {
                    Thread.sleep(1000);
                } catch (Exception e) {
                    // TODO: handle exception
                }
                System.out.println(name);
            }
        }
        ;;;;;;;;;;;;;;;//1 lack line of code
    }
}
class Mythread extends Thread
{
    Display d;
    String name;
    public Mythread(Display d,String name)
    {
        this.d = d;
        this.name = name;
    }
    public void run()
    {
        d.wish(name);
    }
}
class SyncDemo2
{
    public static void main(String[] args) {
        Display d1=new Display();
        Display d2=new Display();
        Mythread t1=new Mythread(d1,"Dhoni");
        Mythread t2=new Mythread(d2,"yuvi");
        t1.start();
        t2.start();
    }
}
```

Example: To get the lock of a particular object 'd' we can declare the synchronized block as follows
//synchronized(d){}

```
class Display
{
```

```
public void wish(Display d,String name)
    {
        ;;;;;;;;;;;;;;;//1 lack line of code
        synchronized(d)
        {
            for(int i=0;i<5;i++)
            {
                System.out.print("Good Morning:");
                try {
                    Thread.sleep(1000);
                } catch (Exception e) {
                    // TODO: handle exception
                }
                System.out.println(name);
            }
        }
        ;;;;;;;;;;;;;;;//1 lack line of code
    }
}
class Mythread extends Thread
{
    Display d;
    String name;
    public Mythread(Display d,String name)
    {
        this.d = d;
        this.name = name;
    }
    public void run()
    {
        d.wish(d,name);
    }
}
class SyncDemo3
{
    public static void main(String[] args) {
        Display d=new Display();
        Mythread t1=new Mythread(d,"Dhoni");
        Mythread t2=new Mythread(d,"yuvi");
        t1.start();
        t2.start();
    }
}
```

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

- Two Threads can communicate with each other by using wait(), notify() and notifyAll() methods.
- The Thread which is required updation it has to call wait() method on the required object then immediately the Thread will entered into waiting state. The Thread which is performing updation of object, it is responsible to give notification by calling 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.
- To call wait(), notify() and notifyAll() methods compulsory the current Thread should be owner of that object.
  - i.e., current Thread should has lock of that object
  - i.e., 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.**
- Once a Thread calls wait() method on the given object 1st it releases the lock of that object immediately and entered into waiting state.
- Once a Thread calls notify() (or) notifyAll() methods it releases the lock of that object but may not immediately.
- Except these (wait(),notify(),notifyAll()) methods there is no other place(method) where the lock release will be happen.

Use of Thread.sleep(ms); In Inter Thread Communication

Example

```
class ThreadA
{
    public static void main(String[] args) throws
InterruptedException
    {
        ThreadB b=new ThreadB();
        b.start();
        Thread.sleep(0,001);
        System.out.println(b.total);
    }
```

```
}
class ThreadB extends Thread
{
    int total=0;
    public void run()
    {
        for(int i=1;i<=100;i++)
        {
            total=total+i;
        }
    }
}
```

- Not Recommended to use Thread.sleep(ms); In Inter Thread Communication bcz at what time the execution of ThreadB completed we doesn't know so in this case main thread has to wait for extra time which will decrease performance

Use of join(ms); In Inter Thread Communication

Example

```
class ThreadA
{
    public static void main(String[] args) throws
InterruptedException
    {
        ThreadB b=new ThreadB();
        b.start();
        b.join(0,1);
        System.out.println(b.total);
    }
}
class ThreadB extends Thread {
    int total = 0;

    public void run() {
        for (int i = 1; i <= 100; i++) {
            total = total + i;
        }
        ;;;;;;;;;;//One lack line code
    }
}
```

- Not recommended bcz let say that in ThreadB object there are one lack Line code after For loop then main Thread Has to Wait until the execution of ThreadB completed but here main Thread required to wait only up to the execution of For Loop.

Use of wait(),notify() In Inter Thread Communication

Example

```
class ThreadA
{
    public static void main(String[] args) throws
            InterruptedException
    {
        ThreadB b=new ThreadB();
        b.start();
        b.wait();
        System.out.println(b.total);
    }
}
class ThreadB extends Thread
{
    int total=0;
    public void run()
    {
        for(int i=1;i<=100;i++)
        {
            total=total+i;
        }
        this.notify();
        ;;;;;;;;;;;//One lack line code
    }
}
```

output:

```
Exception in thread "main"
java.lang.IllegalMonitorStateException: current thread is not
owner
        at java.base/java.lang.Object.wait(Native Method)
        at java.base/java.lang.Object.wait(Object.java:338)
        at ThreadA.main(Mt1.java:8)
Exception in thread "Thread-0"
java.lang.IllegalMonitorStateException: current thread is not
owner
        at java.base/java.lang.Object.notify(Native Method)
        at ThreadB.run(Mt1.java:21)
```

- wait(),notify() can be used only in synchronized area.

Example:

```
class ThreadA
{
    public static void main(String[] args) throws InterruptedException
    {
        ThreadB b=new ThreadB();
        b.start();
        //Required lock of object b bcz want to call wait()
method on object of ThreadB
        synchronized(b)
        {
            System.out.println("Main thread trying to call wait
method"); //(1)
            b.wait();
            System.out.println("Main thread get
notification");//(4)
            System.out.println(b.total);//(5)
        }
    }
}
class ThreadB extends Thread {
    int total = 0;

    public void run() {
        synchronized (this) {
            System.out.println("child thread Start
calculation");//(2)
            for (int i = 1; i <= 100; i++) {
                total = total + i;
            }
            System.out.println("child thread trying to give
notification");//(3)
            this.notify();
        }
        ;;;;;;;;;;//One lack line code
    }
}
```

Producer Consumer Problem

Example:

```
class ProCon
{
    boolean isMarked=false;
    int data;
    synchronized void produce(int n)
    {
        if(isMarked)
        {
            try
            {
                System.out.println("Producer waiting");
                wait();
            }
            catch (Exception e)
            {
// TODO: handle exception
            }
        }
        data=n;
        System.out.println("produce= "+data);
        isMarked=true;
        notify();
    }
    synchronized void consume()
    {
        if(!isMarked)
        {
            try
            {
                System.out.println("Consumer waiting");
                wait();
            }
            catch (Exception e)
            {
// TODO: handle exception
            }
        }
        System.out.println("consume= "+data);
        isMarked=false;
        notify();
    }
}
```

```
class Producer extends Thread
{
    ProCon pc;
    Producer(ProCon pc)
    {
        this.pc=pc;
       //start();
    }
    public void run()
    {
        int i;
        for(i=1;i<=5;i++)
        {
            pc.produce(i);
        }
    }
}
class Consumer extends Thread
{
    ProCon pc;
    Consumer(ProCon pc)
    {
        this.pc=pc;
       //start();
    }
    public void run()
    {
        int i;
        for(i=1;i<=5;i++)
        {
            pc.consume();
        }
    }
}
class QB159
{
    public static void main(String[] args)
    {
        ProCon pc1=new ProCon();
        Producer p=new Producer(pc1);
        Consumer c=new Consumer(pc1);
        p.start();
        c.start();
    }
}
```