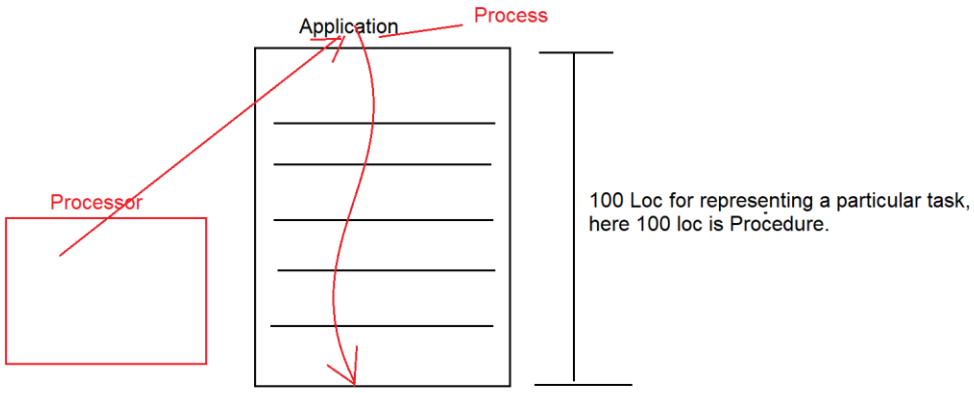
Multi Threading:
—----------------
Q)What is the difference between Process, Processor and Procedure?
—--
Ans:
—---
Procedure is a set of instructions representing a particular task.

Process is a flow of execution to perform a particular task.

Processor is a H/W component to generate a process.

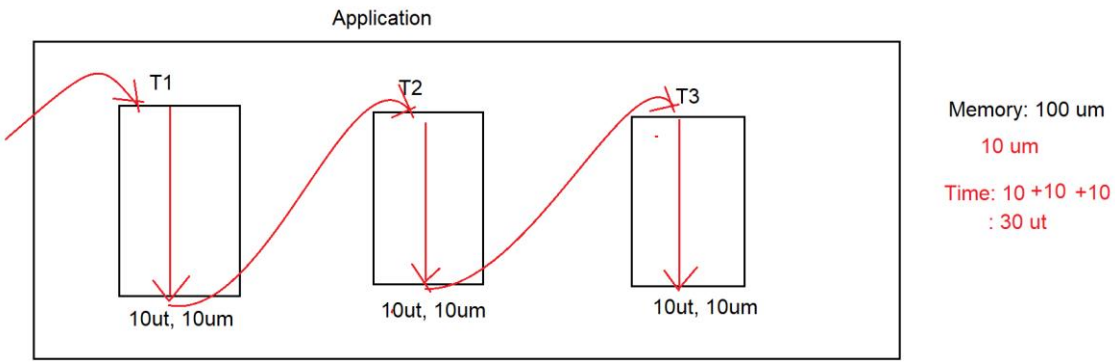


There are two process models to execute applications.

1. Single Process Model
2. Multi Process Model

Single Process Model:
—--------------------

1. It is able to allow only one process to execute the complete application.
2. It follows sequential execution.
3. It takes more execution time.
4. It will provide less performance



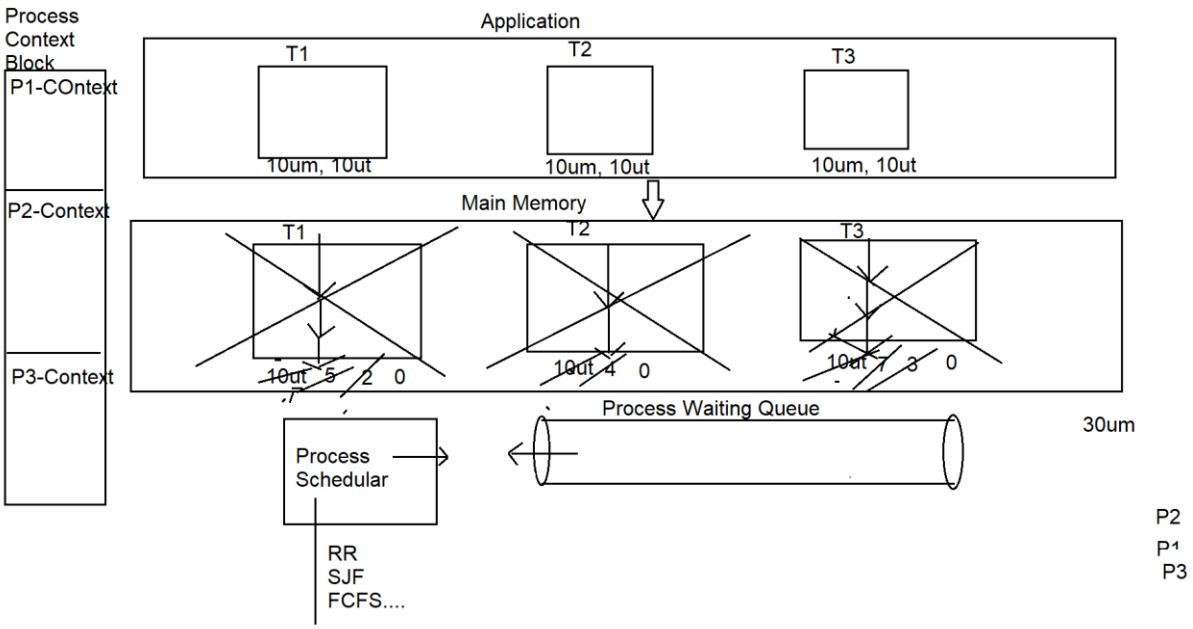
Problems:
1. Not utilizing System resources effectivily.
2. More Execution time
3. Less Performance

Multi Process Mechanism:
—-------------------------

1. It allows more than one process at a time to execute the complete application.
2. It follows parallel execution.
3. It will reduce application execution time.
4. It will increase application performance.

To implement the Multi process mechanism we need the following components internally.

1. Main Memory: To load all the tasks to execute.
2. Process Waiting Queue: To trace all the processes in order to provide timestamps.
3. Process Scheduler: To assign time stamps in order to execute tasks.
4. Process Context Block: To manage details of a process like temporary data while executing tasks, startup time of the process, ending time of the process,.....



Context Switching:
The process of switching control from one Process context to another process context is called “Context Switching”.

There are two types of Context Switchings.

1. Heavy Weight Context Switching
2. Lightweight Context Switching

Heavy Weight Context Switching: It is a context switching between two heavy weight components.
EX: Context Switching between two processes.

Lightweight Context Switching: It is the context switching between two light weight Components.
EX: Context Switching between two Threads.

Q)What is the difference between Process and thread?
—----------
Ans:
—---
Process is heavy, it requires more memory and more execution time, it will reduce applications performance.

Thread is lightweight, it requires less memory and less execution time, it will improve applications performance.

There are two thread models.

1. Single Thread Model
2. Multi Thread Model

Single Thread Model

—------------------

1. It allows only one thread at a time to execute the complete application.
2. It follows Sequential execution.
3. It will increase application execution time.
4. It will reduce application performance.

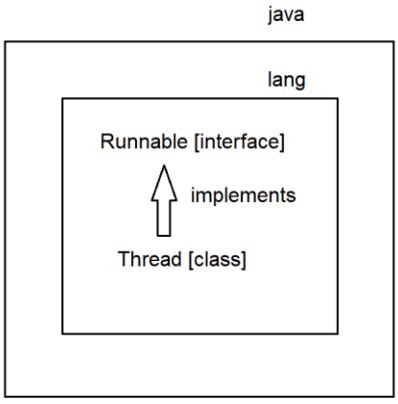
Multi Thread Model:

—-------------------

1. It allows more than one thread at a time to execute the complete application.
2. It follows Parallel execution.
3. It will reduce application execution time.
4. It will increase application performance.

Java is following the Multi Thread Model, it is able to provide a very good environment to create and execute more than one thread at a time.

To prepare Threads in Java applications we have to use the following predefined library.



Q)What is thread and in how many ways are we able to create threads in java applications?

Ans:

Thread is a flow of execution to perform a particular task.

As per the predefined library , there are two ways to create threads.

1. By Using Thread class.
2. By Using Runnable interface

Threads Design By Using java.lang.Thread class:

---

1. Declare an user defined class.
2. Extend User defined class from java.lang.Thread class.
3. Override Thread class run() method with the logic which we want to execute by creating a new thread.

   public void run()
4. In Main class, main() method, create Thread class object and access start() method inorder to create new thread and inorder to access run() method.

   public void start():

   Note: IN Java applications, only start() method is able to create new thread and it is able to access the run() method internally.

EX:

```
class WelcomeThread extends Thread{
    @Override
    public void run() {
        for(int i = 0; i < 10; i++){
            System.out.println("Welcome To Durga Software
Solutions");
        }
    }
}
public class Main {
    public static void main(String[] args) {
        WelcomeThread welcomeThread = new WelcomeThread();
        welcomeThread.start();
    }
}
```

Op:

```
Welcome To Durga Software Solutions
```

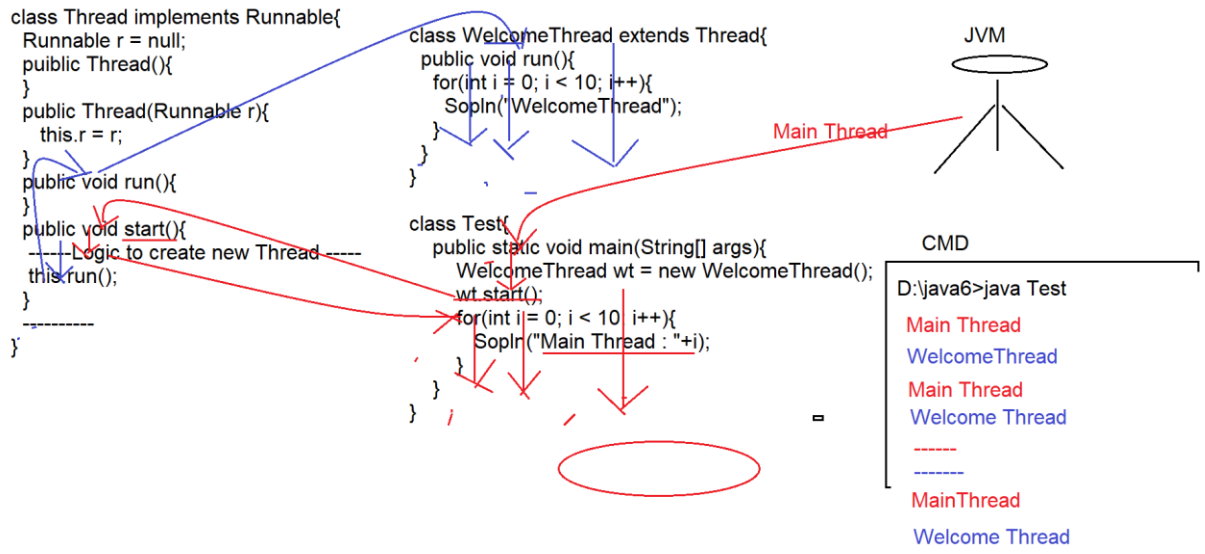
```
Welcome To Durga Software Solutions
Welcome To Durga Software Solutions
Welcome To Durga Software Solutions
Welcome To Durga Software Solutions
Welcome To Durga Software Solutions
Welcome To Durga Software Solutions
Welcome To Durga Software Solutions
Welcome To Durga Software Solutions
Welcome To Durga Software Solutions
```

EX:

```
class WelcomeThread extends Thread{
    @Override
    public void run() {
        for(int i = 0; i < 10; i++){
            System.out.println("WelcomeThread : "+i);
        }
    }
}
public class Main {
    public static void main(String[] args) {
        WelcomeThread welcomeThread = new WelcomeThread();
        welcomeThread.start();
        for(int i = 0; i < 10; i++){
            System.out.println("MainThread : "+i);
        }
    }
}
```

```
MainThread : 0
WelcomeThread : 0
MainThread : 1
WelcomeThread : 1
MainThread : 2
WelcomeThread : 2
MainThread : 3
WelcomeThread : 3
MainThread : 4
WelcomeThread : 4
MainThread : 5
WelcomeThread : 5
MainThread : 6
```

```
WelcomeThread : 6
MainThread : 7
WelcomeThread : 7
MainThread : 8
WelcomeThread : 8
MainThread : 9
WelcomeThread : 9
```



Q) To create Threads, already we have Thread class [First Approach] then what is the requirement to go for Runnable interface [Second Approach]?

Ans:

In the first approach of creating threads we have to extend java.lang.Thread class to an user defined class.

```
class MyClass extends Thread{
}
```

In this approach it is not possible to extend any other class along with Thread class.

```
class MyClass extends Thread, Frame{
}
```

Status : Invalid

In java applications, if we want to create a thread along with some other super class then we have to use the second approach that is by implementing Runnable interface.

```
class MyClass extends Frame implements Runnable{
}
```

Threads Design by Using Runnable interface:

—----------

1. Declare an user defined class.
2. Implement Runnable interface.
3. Provide implementation for run() method in the user defined class with the logic which we want to execute by creating a new thread.
4. In the main class, in the main() method, create a new thread and access run() method.

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

To create a new Thread and to access run() method we have to use the following cases.

Case#1:

```
WelcomeThread wt = new WelcomeThread();
wt.start();
```

Status: Compilation Error.

Reason: start() method was defined in Thread class, not in WelcomeThread class.

Case#2:

```
WelcomeThread wt = new WelcomeThread();
wt.run();
```

Status: No Compilation Error

OP: Sequential Output

```
Welcome Thread
Welcome Thread
—----
—----
Main Thread
```

Main Thread
—----
—----

Reason: No user defined thread is created, Only Main Thread executes both WelcomeThread class run() method and main() method.

Case#3:
WelcomeThread wt = new WelcomeThread();
Thread t = new Thread();
t.start();

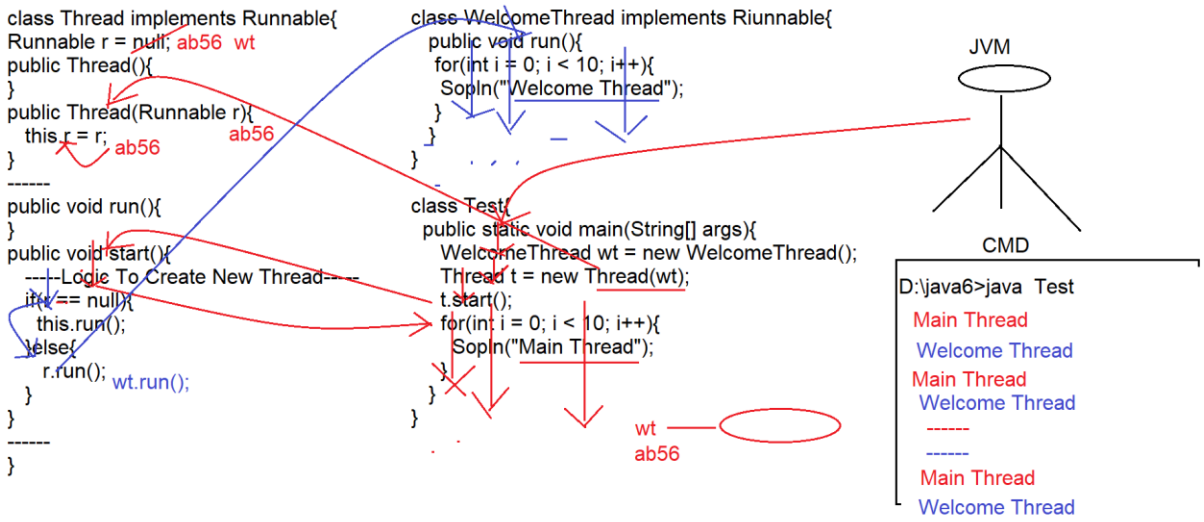
Status: No Compilation Error
Output: Output is generated from main Thread only.
Reason: When we access t.start() method a new thread is created by start() method and start() method bypasses the new thread to the Thread class run() method , but not to the WelcomeThread class run() method.

Case#4:
WelcomeThread wt = new WelcomeThread();
Thread t = new Thread(wt);
t.start();
Status: No Compilation Error
Output: Mixed output from Both Main Thread and User Thread.

EX:

```
class WelcomeThread implements Runnable{
    @Override
    public void run() {
        for (int i = 0; i < 100; i++){
            System.out.println("Welcome Thread");
        }
    }
}
public class Main {
    public static void main(String[] args) {
        WelcomeThread welcomeThread = new WelcomeThread();
        Thread t = new Thread(welcomeThread);
        t.start();
        for(int i = 0; i < 100; i++){
            System.out.println("Main Thread");
```

```
            }
        }
}
```



Thread Lifecycle:
—----------------
The collection of states from the starting point of the thread to ending of the thread is called “Thread Lifecycle”.

In Java applications, Thread Lifecycle has the following states.

1. New/Born State
2. Runnable State
3. Running State
4. Blocking State
5. Dead / Destroy State

New/Born State: When we create a thread class object, automatically thread will come to New or Born State.

Runnable State: IN Java applications, when we access start() method , a new thread will be created but it is waiting for the system resources like execution time and memory, this state is Runnable state.

Running State: After accessing start() method and after getting System resources like memory and execution time then the thread will come to Running state.

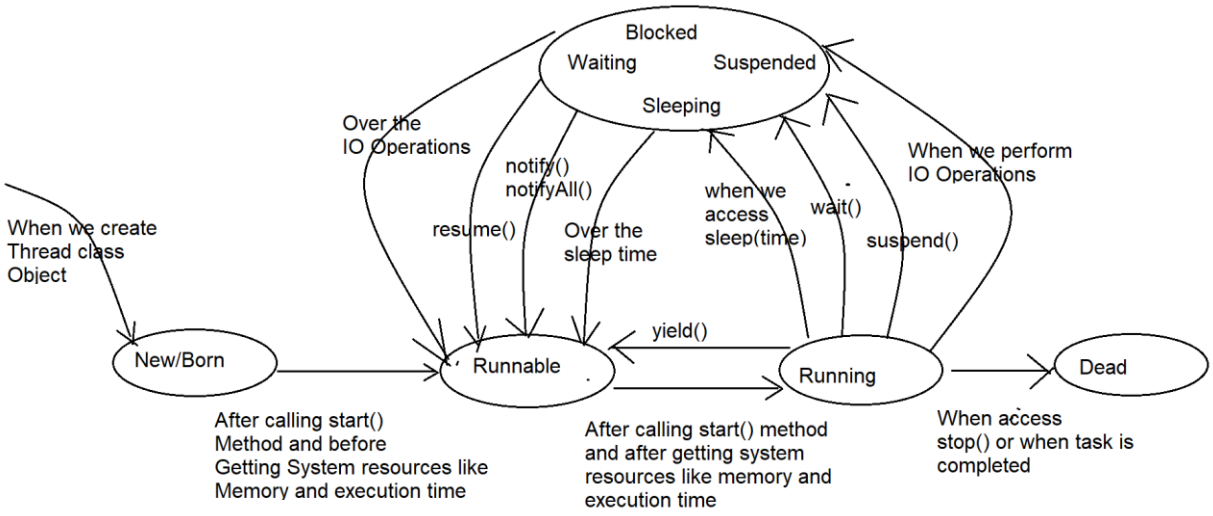
Blocked State:

a. When we access sleep() method with a particular sleep time then the running thread will come to the Blocked state, when sleep time is completed automatically the sleeping thread will come to the Runnable state.
b. When we access the wait() method then the running thread will come to the Blocked state, if any other thread access notify() or notifyAll() method then the waiting thread will come to the Runnable state.
c. When we access suspend() method over the running thread then the running will come to the Blocking state, if any other thread access resume() method then the suspended thread will come to Runnable state.
d. When we perform IO operations in java applications automatically the running thread will come to Blocked state, when IO operations are completed then the Blocked thread will come to the Runnable state.

Note: If we access the yield() method over the running thread then the thread will come from Running state to Runnable state directly.
Note: IN Java applications, yield() method is not supported in windows operating system, because yield() method is able to perform operations on the basis of Thread priorities , but Windows operating system is allowing thread priority based operations.

Dead State: When we access stop() method over the running thread automatically that thread will come to Dead state.

Note: When the Thread completes its task execution then that thread will come to Dead state automatically.



Thread Class Library:

Constructors:

1. public Thread():
   It can be used to create a Thread class object with the default thread properties.
   Thread Name : Thread-0
   Thread Priority : 5
   Thread Group Name: main

   EX:

```
public class Main {
    public static void main(String[] args) {
        Thread thread = new Thread();
         System.out.println(thread);
    }
}
```

   Thread[Thread-0,5,main]

2. public Thread(String name):
   This constructor can be used to create a Thread class object with the provided name.
   EX:

```
public class Main {
    public static void main(String[] args) {
        Thread thread = new Thread("Core Java");
```

```
        System.out.println(thread);
    }
}
```

```
Thread[Core Java,5,main]
```

3. public Thread(Runnable r):
   It can be used to create a Thread class object with the provided Runnable reference.
   EX:

```
public class Main {
    public static void main(String[] args) {
        Runnable r = new Thread();
       Thread thread = new Thread(r);
        System.out.println(thread);
    }
}
```

```
Thread[Thread-1,5,main]
```

4. public Thread(Runnable r, String name):
   It can be used to create a Thread class object with the provided Runnable reference and with the provided thread name.
   EX:

```
public class Main {
    public static void main(String[] args) {
        Runnable r = new Thread();
       Thread thread = new Thread(r, "CORE JAVA");
        System.out.println(thread);
    }
}
```

```
Thread[CORE JAVA,5,main]
```

5. public Thread(ThreadGroup tg, String name)
   It can be used to create Thread class object with the provided ThreadGroup name and with the provided Thread name.

   To represent Thread Group we will use a predefined class in the form of java.lang.ThreadGroup.

   EX:

```
public class Main {
   public static void main(String[] args) {
      ThreadGroup threadGroup = new ThreadGroup("JAVA");
      Thread thread = new Thread(threadGroup, "CORE JAVA");
       System.out.println(thread);
   }
}
```

```
Thread[CORE JAVA,5,JAVA]
```

6. public Thread(ThreadGroup tg, Runnable r):
   It can be used to create Thread class objects with the provided ThreadGroup name and with the Runnable reference.
   EX:

```
public class Main {
   public static void main(String[] args) {
       Runnable runnable = new Thread();
      ThreadGroup threadGroup = new ThreadGroup("JAVA");
      Thread thread = new Thread(threadGroup, runnable);
       System.out.println(thread);
   }
}
```

```
Thread[Thread-1,5,JAVA]
```

7. public Thread(ThreadGroup tg, Runnable r, String name):
   It can be used to create Thread class object with the provided ThreadGroup reference, Runnable reference and Thread name.
   EX:

```
public class Main {
   public static void main(String[] args) {
       Runnable runnable = new Thread();
      ThreadGroup threadGroup = new ThreadGroup("JAVA");
      Thread thread = new Thread(threadGroup, runnable,
"CORE JAVA");
       System.out.println(thread);
   }
}
```

```
Thread[CORE JAVA,5,JAVA]
```

Methods:

—--------

1. Public void setName(String name):
It can be used to set a particular name to the Thread explicitly.

2. public String getName():
It can be used to get the name of the Thread.

EX:

```
public class Main {
   public static void main(String[] args) {
        Thread thread = new Thread();
        System.out.println(thread.getName());
        thread.setName("MyThread");
        System.out.println(thread.getName());
   }
}
```

```
Thread-0
MyThread
```

3. public void setPriority(int priority):
It can be used to set a particular priority value to the Thread.

In Java, every thread should have a priority value from 1 to 10, here the default priority value for all the threads is 5.

To the setPriority() method we must provide the values from 1 to 10 only as parameter, if we provide priority value in out side range of 1 to 10 as parameter to setPriority() method then JVM will raise an exception like java.lang.IllegalArgumentException.

In Thread class, to represent Thread priority values JAVA has provided the following constant variables.

```
public static final int MIN_PRIORITY = 1;
public static final int NORM_PRIORIOTY = 5;
public static final int MAX_PRIORITY = 10;
```

4. public int getPriority():
It can be used to get the Priority value from the Thread.

EX:

```
public class Main {
```

```
    public static void main(String[] args) {
        Thread thread = new Thread();
        System.out.println(thread.getPriority());
        thread.setPriority(7);
        System.out.println(thread.getPriority());
    }
}
```

```
5
7
```

EX:

```
public class Main {
    public static void main(String[] args) {
        Thread thread = new Thread();
        System.out.println(thread.getPriority());
        thread.setPriority(15);
        System.out.println(thread.getPriority());
    }
}
```

```
5
Exception in thread "main" java.lang.IllegalArgumentException
      at java.base/java.lang.Thread.setPriority(Thread.java:1138)
      at Main.main(Main.java:5)
```

EX:

```
public class Main {
    public static void main(String[] args) {
        Thread thread = new Thread();
        System.out.println(thread.getPriority());
        thread.setPriority(Thread.MAX_PRIORITY-2);
        System.out.println(thread.getPriority());
    }
}
```

```
5
8
```

5. public static int activeCount():

It can be used to get the number of threads which are running in the present application.
EX:

```
class MyThread1 extends Thread{
    @Override
    public void run() {
        for(int i = 0; i < 2; i++){

        }
    }
}
class MyThread2 extends Thread{
    @Override
    public void run() {
        for(int i = 0; i < 10; i++){

        }
    }
}
public class Test {
    public static void main(String[] args) {
        MyThread1 mt1 = new MyThread1();
        MyThread2 mt2 = new MyThread2();
        System.out.println(Thread.activeCount());
        mt1.start();
        mt2.start();
        System.out.println(Thread.activeCount());
    }
}
```

```
D:\java6>javac Test.java

D:\java6>java Test
1
3
```

6. public boolean isAlive():
It can be used to check whether a thread is in live or not.
EX:

```
public class Main {
   public static void main(String[] args) {
        Thread thread = new Thread();
```

```
        System.out.println(thread.isAlive());
        thread.start();
        System.out.println(thread.isAlive());
    }
}
```

```
false
true
```

7. public static Thread currentThread():
It can be used to get a Thread object reference which is executing the current instruction.
EX:

```
class A{
    public void m1(){
        for(int i = 0; i < 10; i++) {
            System.out.println(Thread.currentThread().getName());
        }
    }
}
class MyThread1 extends Thread{
    A a;
    MyThread1(A a){
        this.a = a;
    }
    @Override
    public void run() {
        a.m1();
    }
}
class MyThread2 extends Thread{
    A a;
    MyThread2(A a){
        this.a = a;
    }
    @Override
    public void run() {
        a.m1();
    }
}
class MyThread3 extends Thread{
```

```
    A a;
    MyThread3(A a){
        this.a = a;
    }
    @Override
    public void run() {
        a.m1();
    }
}
public class Main {
    public static void main(String[] args) {
        A a = new A();
        MyThread1 mt1 = new MyThread1(a);
        MyThread2 mt2 = new MyThread2(a);
        MyThread3 mt3 = new MyThread3(a);

        mt1.setName("First Thread");
        mt2.setName("Second Thread");
        mt3.setName("Third Thread");

        mt1.start();
        mt2.start();
        mt3.start();
    }
}
```

```
First Thread
First Thread
First Thread
First Thread
First Thread
First Thread
First Thread
First Thread
First Thread
First Thread
Second Thread
Third Thread
Third Thread
Third Thread
Third Thread
```

```
Third Thread
Second Thread
Second Thread
Second Thread
Second Thread
Second Thread
Second Thread
Second Thread
Second Thread
Second Thread
Third Thread
Third Thread
Third Thread
Third Thread
Third Thread
```

8. public static void sleep(long time)throws InterruptedException:
It can be used to keep a running thread into sleeping state up to the specified sleep time. When sleep time is completed , automatically, the thread will continue its execution.
EX:

```
class WelcomeThread extends Thread{
   @Override
   public void run() {
       for (int i = 0; i < 10; i++){
           try {
               Thread.sleep(1000);
               System.out.println("Welcome To Durga Software
Solutions");
           } catch (InterruptedException e) {
               e.printStackTrace();
           }

       }
   }
}
public class Main {
   public static void main(String[] args) {
       WelcomeThread welcomeThread = new WelcomeThread();
       welcomeThread.start();
   }
```

```
}
```

9. public void join()throws InterruptedException:
It can be used to pause a thread to complete another thread, after completion of the other thread the paused thread will continue its execution automatically.
EX:

```
class WelcomeThread extends Thread{
   @Override
   public void run() {
       for (int i = 0; i < 10; i++){
           System.out.println("WelcomeThread : "+i);
       }
   }
}
public class Main {
   public static void main(String[] args) throws
InterruptedException {
       WelcomeThread welcomeThread = new WelcomeThread();
       welcomeThread.start();
       welcomeThread.join();
       for(int i = 0; i < 10; i++){
           System.out.println("MainThread : "+i);
       }
   }
}
```

```
WelcomeThread : 0
WelcomeThread : 1
WelcomeThread : 2
WelcomeThread : 3
WelcomeThread : 4
WelcomeThread : 5
WelcomeThread : 6
WelcomeThread : 7
WelcomeThread : 8
WelcomeThread : 9
MainThread : 0
MainThread : 1
MainThread : 2
MainThread : 3
```

```
MainThread : 4
MainThread : 5
MainThread : 6
MainThread : 7
MainThread : 8
MainThread : 9
```

Daemon Threads:
—---------------
Daemon thread is a thread running internally and providing services to some other thread and it will be terminated automatically when the thread which is taking services is terminated.
EX: IN JVM, Garbage Collector is a Daemon Thread, it will run internally and it will provide Garbage Collection service to the JVM and it will be terminated automatically along with JVM.

To make a Thread as Daemon Thread we have to use the following method.

public void setDaemon(boolean b)
If we provide true value as a parameter then the thread will be a Daemon thread.
If we provide a false value as a parameter then the thread will not be a daemon thread.

Note: In the case of Daemon threads, we must access setDaemon() method before starting the thread, if we access setDaemon() method after starting the thread then JVM will raise an exception like java.lang.IllegalThreadStateException.

To check whether a thread is a daemon thread or not we will use the following method.

public boolean isDaemon()

EX:

```java
class GarbageCollector extends Thread{
   @Override
   public void run() {
        while(true){
            System.out.println("Garbage Collector Thread.....");
        }
   }
}
```

```
public class Main {
   public static void main(String[] args)  {
       GarbageCollector garbageCollector = new
GarbageCollector();
       garbageCollector.setDaemon(true);
       garbageCollector.start();
       //garbageCollector.setDaemon(true);---
>IllegalThreadSTateException
       for (int i = 0; i < 10; i++){
           System.out.println("JVM Thread....");
       }
   }
}
```

Threadsafe Resources:
—---------------------
If any resource is able to allow more than one thread at a time and if it is able to process more than one thread without having data inconsistency then that resource is called “Thread Safe Resource”.

To make a resource as a Thread Safe resource then we have to use the following guidelines.

1. Use Local variables instead of Class level Variables.
2. Use Immutable Objects instead of Mutable Objects.
3. Use Synchronization

Use Local variables instead of Class level Variables:
—---
If we provide data in the form of instance variables that are class level variables then that data will be stored inside the objects in heap memory, heap memory objects provided data is sharable to all the threads which are executing in the present java application.

In the above context, one thread modified data will be used by the other threads, where there's a chance of getting data inconsistency.

IN the above situation, to provide data consistency we have to use local variables instead of Instance variables that are class level variables.
In Java applications, local variables data will not be shared to all the threads which are running in the java applications. Local variables data will

be stored inside a stack in the Stack memory which is specific to a particular thread, if the thread perform modifications on a local variable then that modification is available up to the thread respective stack only, it will not be available to other threads, this approach will not provide data inconsistency, it will make the resource as Thread Safe resource.

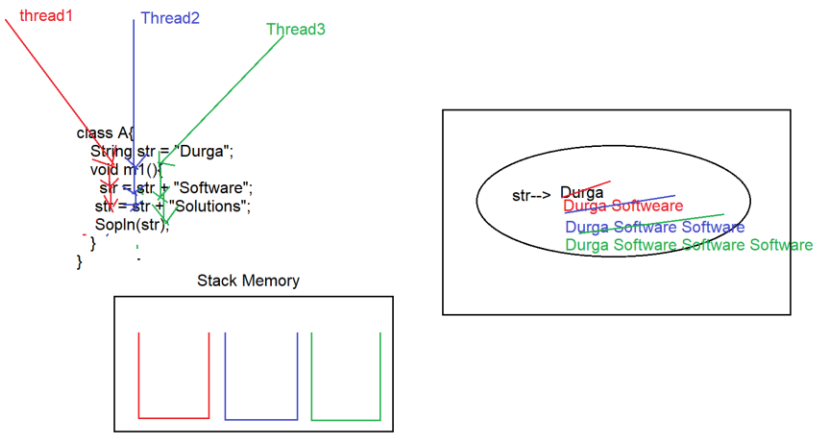
EX:

```
class A{
    String str = "Durga";
    void m1(){

        str=str+" Software";
        str = str+ " Solutions";
        System.out.println(Thread.currentThread().getName()+" :
"+str);
    }
}
class Thread1 extends Thread{
    A a;
    Thread1(A a){
        this.a = a;
    }
    @Override
    public void run() {
        a.m1();
    }
}
class Thread2 extends Thread{
    A a;
    Thread2(A a){
        this.a = a;
    }
    @Override
    public void run() {
        a.m1();
    }
}
class Thread3 extends Thread{
    A a;
    Thread3(A a){
        this.a = a;
```

```
    }
    @Override
    public void run() {
        a.m1();
    }
}
public class Main {
    public static void main(String[] args) {
        A a = new A();
        Thread1 t1 = new Thread1(a);
        Thread2 t2 = new Thread2(a);
        Thread3 t3 = new Thread3(a);

        t1.setName("Thread-1");
        t2.setName("Thread-2");
        t3.setName("Thread-3");

        t1.start();
        t2.start();
        t3.start();
    }
}
```

Output:

```
Thread-1 : Durga Software Solutions
Thread-3 : Durga Software Solutions Software Solutions
Thread-2 : Durga Software Solutions
```

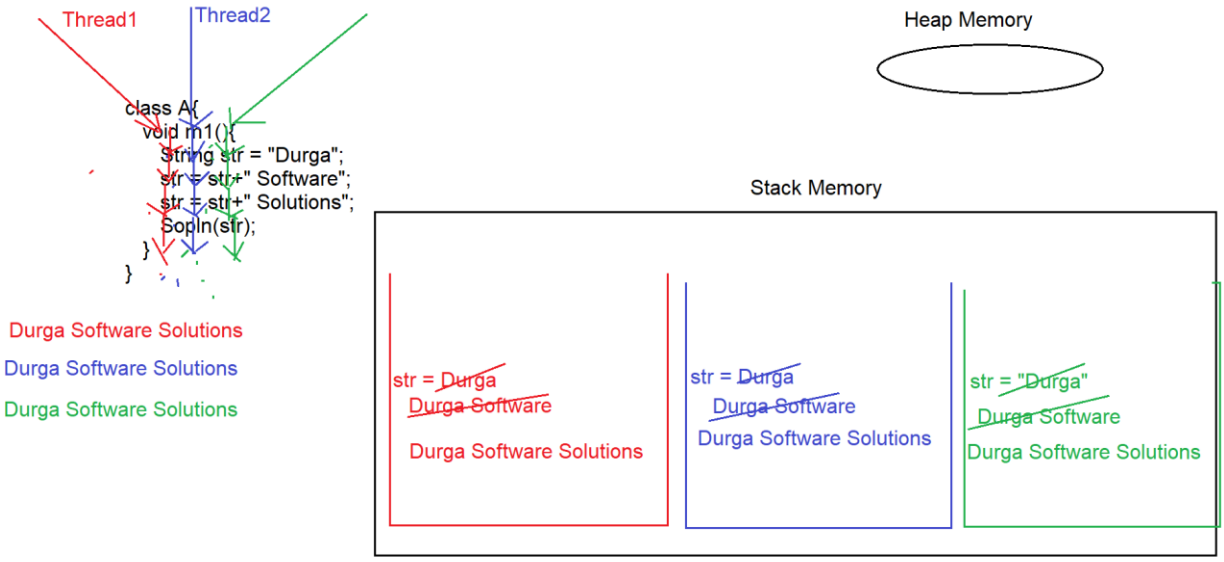


EX:

```
class A{

   void m1(){
       String str = "Durga";
       str=str+" Software";
       str = str+ " Solutions";
       System.out.println(Thread.currentThread().getName()+" :
"+str);
   }
}
class Thread1 extends Thread{
   A a;
   Thread1(A a){
       this.a = a;
   }
   @Override
   public void run() {
       a.m1();
   }
}
class Thread2 extends Thread{
   A a;
   Thread2(A a){
       this.a = a;
   }
   @Override
   public void run() {
       a.m1();
   }
}
class Thread3 extends Thread{
   A a;
   Thread3(A a){
       this.a = a;
   }
   @Override
   public void run() {
       a.m1();
   }
}
public class Main {
```

```
    public static void main(String[] args) {
        A a = new A();
        Thread1 t1 = new Thread1(a);
        Thread2 t2 = new Thread2(a);
        Thread3 t3 = new Thread3(a);

        t1.setName("Thread-1");
        t2.setName("Thread-2");
        t3.setName("Thread-3");

        t1.start();
        t2.start();
        t3.start();
    }
}
```

```
Thread-3 : Durga Software Solutions
Thread-1 : Durga Software Solutions
Thread-2 : Durga Software Solutions
```



2. Use Immutable Objects Over Mutable Objects:

Mutable object is a java object, it is able to allow modifications on its content directly.
EX: StringBuffer

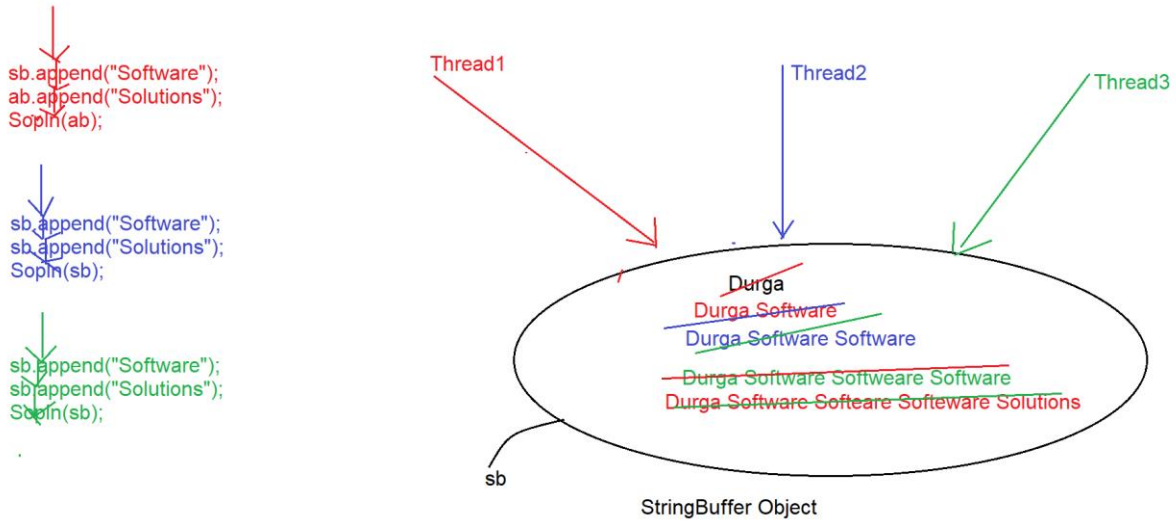
If we provide more than one thread on the Mutable object and more than one thread is performing operations on the Mutable object then all the threads modified data will be stored in the same Mutable object, it will provide data inconsistency, it will not make the resource as Thread safe resource.

Immutable object is a Java object, it will not allow modifications on its content directly, if we are trying to perform modifications on its content then data is allowed for modifications but the modified resultant data will not be stored back in the original object, where the modified resultant data will be stored by creating a new object, this nature of the immutable objects will provide data consistency, it will make the resource as Thread safe Resource.

EX:

```
class Thread1 extends Thread{
    StringBuffer sb;
    Thread1(StringBuffer sb){
        this.sb = sb;
    }
    @Override
    public void run() {
        sb.append("Software ");
        sb.append("Solutions");
        System.out.println(sb);
    }
}
class Thread2 extends Thread{
    StringBuffer sb;
    Thread2(StringBuffer sb){
        this.sb = sb;
    }
    @Override
    public void run() {
        sb.append("Software ");
        sb.append("Solutions");
        System.out.println(sb);
```

```
    }
}
class Thread3 extends Thread{
    StringBuffer sb;
    Thread3(StringBuffer sb){
        this.sb = sb;
    }
    @Override
    public void run() {
        sb.append("Software ");
        sb.append("Solutions");
        System.out.println(sb);
    }
}
public class Main {
    public static void main(String[] args) {
       StringBuffer sb = new StringBuffer("Durga ");
        Thread1 t1 = new Thread1(sb);
        Thread2 t2 = new Thread2(sb);
        Thread3 t3 = new Thread3(sb);

        t1.setName("Thread-1");
        t2.setName("Thread-2");
        t3.setName("Thread-3");

        t1.start();
        t2.start();
        t3.start();
    }
}
```

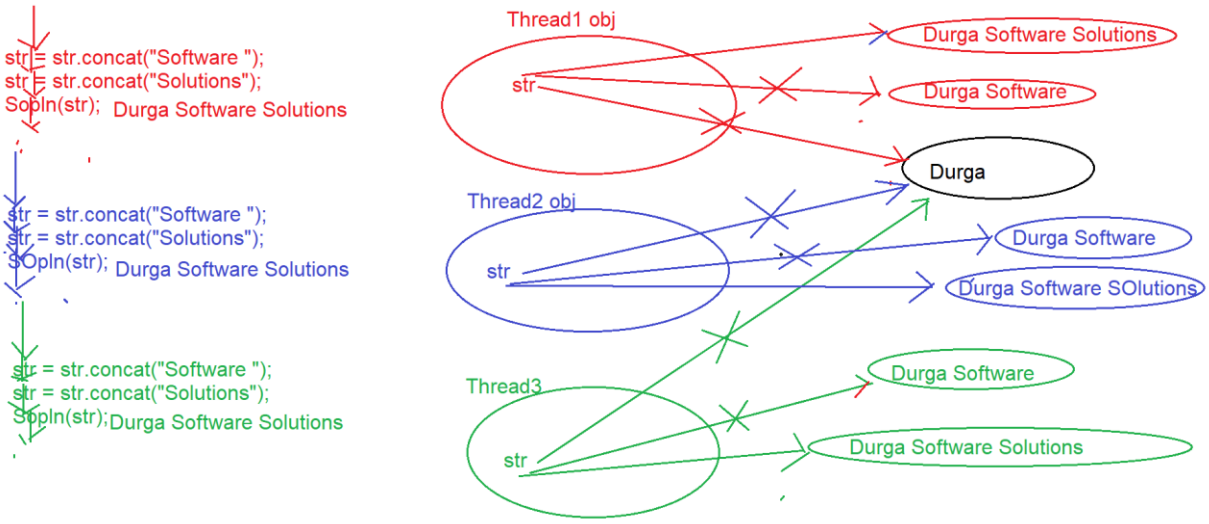


EX:

```
package pack1;

class Thread1 extends Thread{
  String str;
   Thread1(String str){
       this.str = str;
   }
   @Override
   public void run() {
       str = str.concat("Software ");
       str = str.concat("Solutions");
       System.out.println(str);
   }
}
class Thread2 extends Thread{
   String str;
   Thread2(String str){
       this.str = str;
   }
   @Override
   public void run() {
       str = str.concat("Software ");
       str = str.concat("Solutions");
       System.out.println(str);
   }
```

```
}
class Thread3 extends Thread{
   String str;
   Thread3(String str){
       this.str = str;
   }
   @Override
   public void run() {
       str = str.concat("Software ");
       str = str.concat("Solutions");
       System.out.println(str);
   }
}
public class Test {
   public static void main(String[] args) {
      String str = new String("Durga ");
       Thread1 t1 = new Thread1(str);
       Thread2 t2 = new Thread2(str);
       Thread3 t3 = new Thread3(str);

       t1.setName("Thread-1");
       t2.setName("Thread-2");
       t3.setName("Thread-3");

       t1.start();
       t2.start();
       t3.start();
   }
}
```

```
Durga Software Solutions
Durga Software Solutions
Durga Software Solutions
```

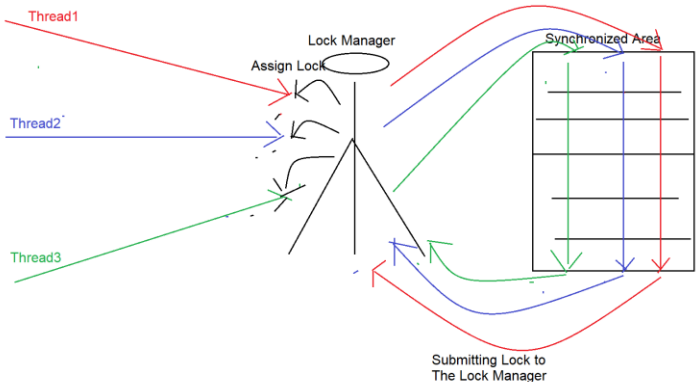


Synchronization:

—----------

Synchronization is a mechanism, Where it is able to allow only one thread at a time , not allowing more than one thread at a time, allowing other threads after completion of the present thread execution.

In Java, Synchronization was implemented on the basis of Locking mechanisms, where in Locking Mechanism there is a Lock Manager, it will assign a lock to the thread out of multiple threads which is coming to the Synchronized Area , if any thread acquire a lock then that thread is eligible to execute synchronized area code , after executing synchronized area code, at the end of the Synchronized Area code execution the respective thread must submit lock back to the Lock Manager, where Lock Manager will assign lock to the another thread.



In Java , to provide synchronization Java has provided a predefined keyword that is “Synchronized”.

There are two ways to provide Synchronization in java applications.

1. Synchronized Method.
2. Synchronized Block.

Synchronized Method:

—--------------------

Synchronized method is a normal java method, it is able to allow only one thread at a time, it is unable to allow more than one thread at a time, it is able to allow other threads after completion of the present thread.

EX:

```
synchronized void m1(){
      —-----
}
```

EX:

—---

```
class A{
    synchronized void m1(){
        for(int i = 0; i < 10; i++){
            System.out.println(Thread.currentThread().getName());
        }
    }
}
class Thread1 extends Thread{
    A a;
    Thread1(A a){
        this.a = a;
    }
    @Override
    public void run() {
        a.m1();
    }
}
class Thread2 extends Thread{
    A a;
    Thread2(A a){
        this.a = a;
```

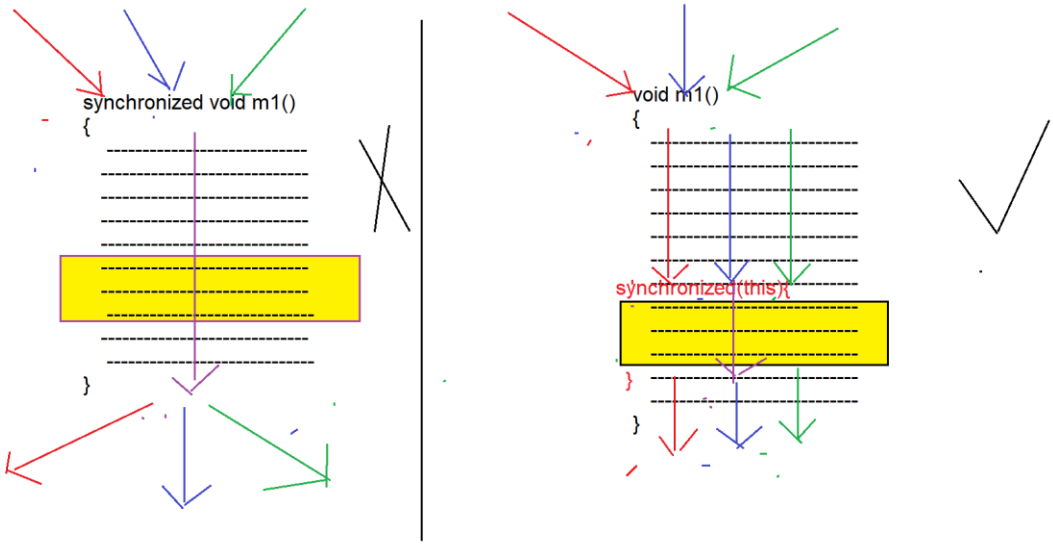
```
    }
    @Override
    public void run() {
        a.m1();
    }
}
class Thread3 extends Thread{
    A a;
    Thread3(A a){
        this.a = a;
    }
    @Override
    public void run() {
        a.m1();
    }
}
public class Main {
    public static void main(String[] args) {
        A a = new A();

        Thread1 t1 = new Thread1(a);
        Thread2 t2 = new Thread2(a);
        Thread3 t3 = new Thread3(a);

        t1.setName("AAA");
        t2.setName("BBB");
        t3.setName("CCC");

        t1.start();
        t2.start();
        t3.start();


    }
}
```

Q)IN Java applications, to provide Synchronization we already have a Synchronized method then what is the requirement to use Synchronized Block?

ANs:

In Java applications, if we provide synchronization by using synchronized method then Synchronization will be provided throughout the method irrespective of the actual requirement, it will increase application execution time and it will reduce application performance.

In the above context, if we want to provide synchronization to the required block of instructions , not to the complete method then we must use a Synchronized block inside the method, it will reduce application execution time and it will improve application performance.



Synchronized Block:

It is a set of instructions , which are able to allow only one thread at a time, not able to allow more than one thread at a time, it is able to allow other threads after completing the present thread execution.

Syntax:

```
synchronized(Object){
}
```

EX:

```
class A{
   void m1(){
```

```
        System.out.println("Before Synchronized Block :
"+Thread.currentThread().getName());
        synchronized (this) {
            for (int i = 0; i < 10; i++) {
                System.out.println("Inside Synchronized Block :
"+Thread.currentThread().getName());
            }
        }
    }
}
class Thread1 extends Thread{
    A a;
    Thread1(A a){
        this.a = a;
    }
    @Override
    public void run() {
        a.m1();
    }
}
class Thread2 extends Thread{
    A a;
    Thread2(A a){
        this.a = a;
    }
    @Override
    public void run() {
        a.m1();
    }
}
class Thread3 extends Thread{
    A a;
    Thread3(A a){
        this.a = a;
    }
    @Override
    public void run() {
        a.m1();
    }
}
public class Main {
```

```
    public static void main(String[] args) {
        A a = new A();

        Thread1 t1 = new Thread1(a);
        Thread2 t2 = new Thread2(a);
        Thread3 t3 = new Thread3(a);

        t1.setName("AAA");
        t2.setName("BBB");
        t3.setName("CCC");

        t1.start();
        t2.start();
        t3.start();


    }
}
```

```
Before Synchronized Block : CCC
Before Synchronized Block : BBB
Before Synchronized Block : AAA
Inside Synchronized Block : CCC
Inside Synchronized Block : CCC
Inside Synchronized Block : CCC
Inside Synchronized Block : CCC
Inside Synchronized Block : CCC
Inside Synchronized Block : CCC
Inside Synchronized Block : CCC
Inside Synchronized Block : CCC
Inside Synchronized Block : CCC
Inside Synchronized Block : CCC
Inside Synchronized Block : CCC
Inside Synchronized Block : AAA
Inside Synchronized Block : AAA
Inside Synchronized Block : AAA
Inside Synchronized Block : AAA
Inside Synchronized Block : AAA
Inside Synchronized Block : AAA
Inside Synchronized Block : AAA
Inside Synchronized Block : AAA
Inside Synchronized Block : AAA
```

```
Inside Synchronized Block : AAA
Inside Synchronized Block : BBB
Inside Synchronized Block : BBB
Inside Synchronized Block : BBB
Inside Synchronized Block : BBB
Inside Synchronized Block : BBB
Inside Synchronized Block : BBB
Inside Synchronized Block : BBB
Inside Synchronized Block : BBB
Inside Synchronized Block : BBB
Inside Synchronized Block : BBB
```

Note: IN Java applications, synchronization is not suggestible, it is suggestible to avoid synchronization in java applications, if it is mandatory to use Synchronization then only use Synchronization otherwise Synchronization is not suggestible, because it will reduce applications performance.

Inter Thread Communication:

The process of providing communication between more than one thread in order to complete a particular task is called Inter thread communication.

To implement Inter Thread communication in java applications we have to use the following methods from java.lang.Object class.

wait(): It will make a thread to wait until we get a notification from another thread.

notify(): To make a waiting thread to activate and to execute its relation task.

notifyAll(): To give activation to all the threads which are in waiting state.

If we want to use all the above methods in java applications we must provide synchronization otherwise we may get some exceptions.

In general, Inter Thread Communication is able to provide the solutions for the problems like Producer-Consumers

In the Producer-Consumer problem, Both producer and consumer are two threads, where producer has to produce an item and consumer has to consume that item, where producer would not produce an item without consuming the previous item by the COnsumer, where Consumer should not consume an item without producing that item.

EX:

—--

```
class A{
   boolean flag = true;
   int item = 0;
   public synchronized void produce(){
       try{
           while(true){
               if(flag == true){
                   item = item + 1;
                   System.out.println("Producer Produced  :
Item-"+item);
                   flag = false;
                   notify();
                   wait();
               }else{
                   wait();
               }
           }
       }catch(Exception e){
           e.printStackTrace();
       }
   }
   public synchronized void consume(){
       try{
           while(true){
               if(flag == true){
                   wait();
               }else{
                   System.out.println("Consumer Consumed : Item-
"+item);
                   flag = true;
```

```
                    notify();
                    wait();
                }
            }
        }catch(Exception e){
            e.printStackTrace();
        }
    }
}
class Producer extends Thread{
    A a;
    Producer(A a){
        this.a = a;
    }

    @Override
    public void run() {
        a.produce();
    }
}
class Consumer extends Thread{
    A a;
    Consumer(A a){
        this.a = a;
    }

    @Override
    public void run() {
        a.consume();
    }
}
public class Main {
    public static void main(String[] args) {
        A a = new A();
        Producer producer = new Producer(a);
        Consumer consumer = new Consumer(a);
        producer.start();
        consumer.start();
    }
}
```

ThreadLocale:
—----------
IN Java , we are able to provide the following four types of scopes for the data.

1. Public
2. Protected
3. <default>
4. private

Where public members are having scope throughout the application.

Where protected members will have scope up to the current package and up to the child classes which are available in other packages.

Where <default> members will have scope up to the present package.

Where Private members will have scope up to the present class.

In Java , there is one more scope unofficially to define for data that is “Thread Scope”.

If we define thread scope to the data then that data will be available to all the methods which are accessed by the present thread, where the methods are available in a single class or in multiple classes.

To provide Thread Scope for the data in Java applications we have to use a predefined class in the form of java.lang.ThreadLocal.

To keep data in Thread Scope and to get data from Thread Scope ThreadLocal class has provided the following methods.

public void set(String data)

public String get()

public void remove()

If we access get() method to get data but we have not set data in thread scope then get() method will return null value, in this context if we want to provide some default data instead of null value then we have to override the following method from threadLocal class.

Public void initialValue()

Ex:

```
----
class ThreadScope extends ThreadLocal{
    @Override
    protected Object initialValue() {
        return "No Data Found in this Thread Scope";
    }
}
class A{
    void m1(){
        System.out.println(Thread.currentThread().getName()+" :
m1() Thread-1 Scope : "+Thread1.threadScope.get());
        System.out.println(Thread.currentThread().getName()+" :
m1() Thread-2 Scope : "+Thread2.threadScope.get());
    }
    void m2(){
        System.out.println(Thread.currentThread().getName()+" :
m2() Thread-2 Scope : "+Thread2.threadScope.get());
        System.out.println(Thread.currentThread().getName()+" :
m2() Thread-1 Scope : "+Thread1.threadScope.get());
    }
}
class Thread1 extends Thread{
    static ThreadScope threadScope = new ThreadScope();
    A a;
    Thread1(A a){
        this.a = a;
    }
    @Override
    public void run() {
        threadScope.set("Data From Thread1 Scope");
        a.m1();
    }
}
```

```
class Thread2 extends Thread{
   static ThreadScope threadScope = new ThreadScope();
   A a;
   Thread2(A a){
       this.a = a;
   }
   @Override
   public void run() {
       threadScope.set("Data from Thread2 Scope");
       a.m2();
   }
}
public class Main {
   public static void main(String[] args) {
       A a = new A();
       Thread1 thread1 = new Thread1(a);
       Thread2 thread2 = new Thread2(a);

       thread1.setName("Thread-1");
       thread2.setName("Thread-2");

       thread1.start();
       thread2.start();
   }
}
```

```
Thread-1 : m1() Thread-1 Scope : Data From Thread1 Scope
Thread-2 : m2() Thread-2 Scope : Data from Thread2 Scope
Thread-1 : m1() Thread-2 Scope : No Data Found in this Thread
Scope
Thread-2 : m2() Thread-1 Scope : No Data Found in this Thread
Scope
```

Deadlocks:

—----------

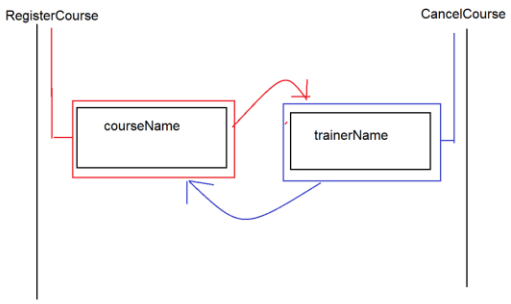
In Multi Threading, more than one thread is depending on each other in circular dependency, in this case Java program execution is struck, this situation is called Deadlock situation.

IN Java applications, we are unable to recover a program from Deadlock once it has occurred, only we are able to avoid deadlock situations while writing the program.

EX:
IN a Java applications, we will use the threads like RegisterCourse and CancelCourse , both the tasks required two resources like CourseName and TrainerName. Let us consider the below situation to represent a deadlock.

1. RegisterCourse Thread and CancelCourse Thread are started at a time.
2. At a time, RegisterCourse thread holds courseName resource and CancelCourse thread holds trainerName resource.
3. To complete the RegisterCourse thread execution it must require trainerName resource from CancelCourse Thread, to complete CancelCourse thread execution it must require courseName resource from RegisterCourse thread.
4. RegisterCourse thread will not release courseName resource to the CancelCourse thread without getting trainerName resource from CancelCourse thread, CancelCourse thread will not release trainerName resource to the RegisterCourse thread without getting courseName resource from RegisterCourse thread.

   The above situation is able to provide deadlock situations in java applications.



EX:

```
class RegisterCourse extends Thread{
    Object courseName;
    Object trainerName;

    RegisterCourse(Object courseName, Object trainerName){
        this.courseName = courseName;
        this.trainerName = trainerName;
    }

    @Override
    public void run() {
        synchronized (courseName){
            System.out.println("Register Course Thread Holds
courseName resource and waiting for trainerName
resource......");
            synchronized (trainerName){
                System.out.println("RegisterCourse thread holds
both courseName and trainerName , so Course Registration is
Success ");
            }
        }
    }
}
class CancelCourse extends Thread{
    Object courseName;
    Object trainerName;

    CancelCourse(Object courseName, Object trainerName){
        this.courseName = courseName;
        this.trainerName = trainerName;
    }

    @Override
    public void run() {
        synchronized (trainerName){
            System.out.println("CancelCourse Thread holds
trainerName resource and waiting for courseName
resource......");
            synchronized (courseName){
```

```
                System.out.println("CancelCourse Thread holds
both trainerName and courseName resources, so Course
Cancellation is Success");
            }
        }
    }
}
public class Main {
    public static void main(String[] args) {
        Object courseName = new Object();
        Object trainerName = new Object();

        RegisterCourse registerCourse = new
RegisterCourse(courseName, trainerName);
        CancelCourse cancelCourse = new CancelCourse(courseName,
trainerName);

        registerCourse.start();
        cancelCourse.start();
    }
}
```

```
CancelCourse Thread holds trainerName resource and waiting for
courseName resource......
Register Course Thread Holds courseName resource and waiting for
trainerName resource......

================================================================
```