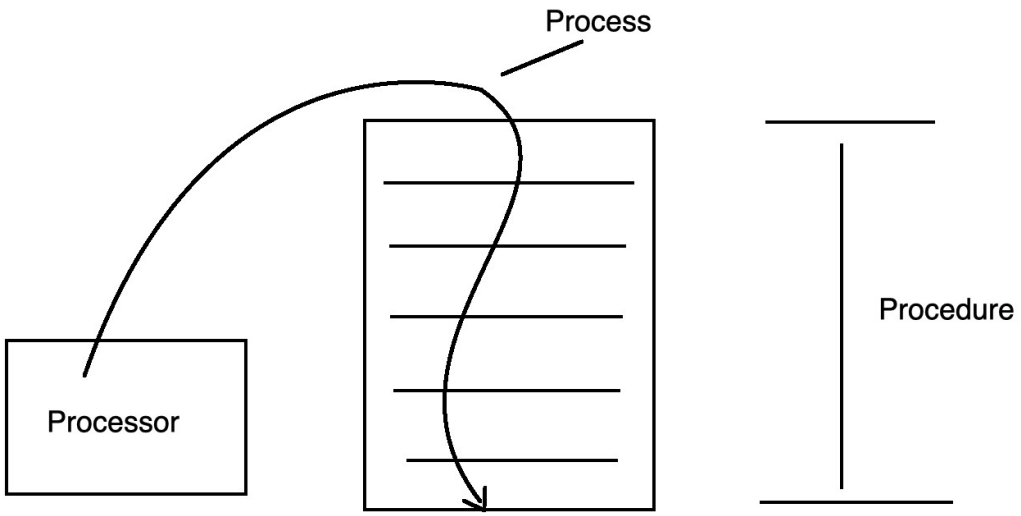
MULTI THREADING:
—---------------
Q)What is the difference between Process, Procedure and Processor?
—--
Ans:
—----
Procedure is a set of instructions that represents a particular task.

Process is a flow of execution that performs a particular task by executing a procedure.

Processor is a Hardware component to generate processes in order to perform tasks.

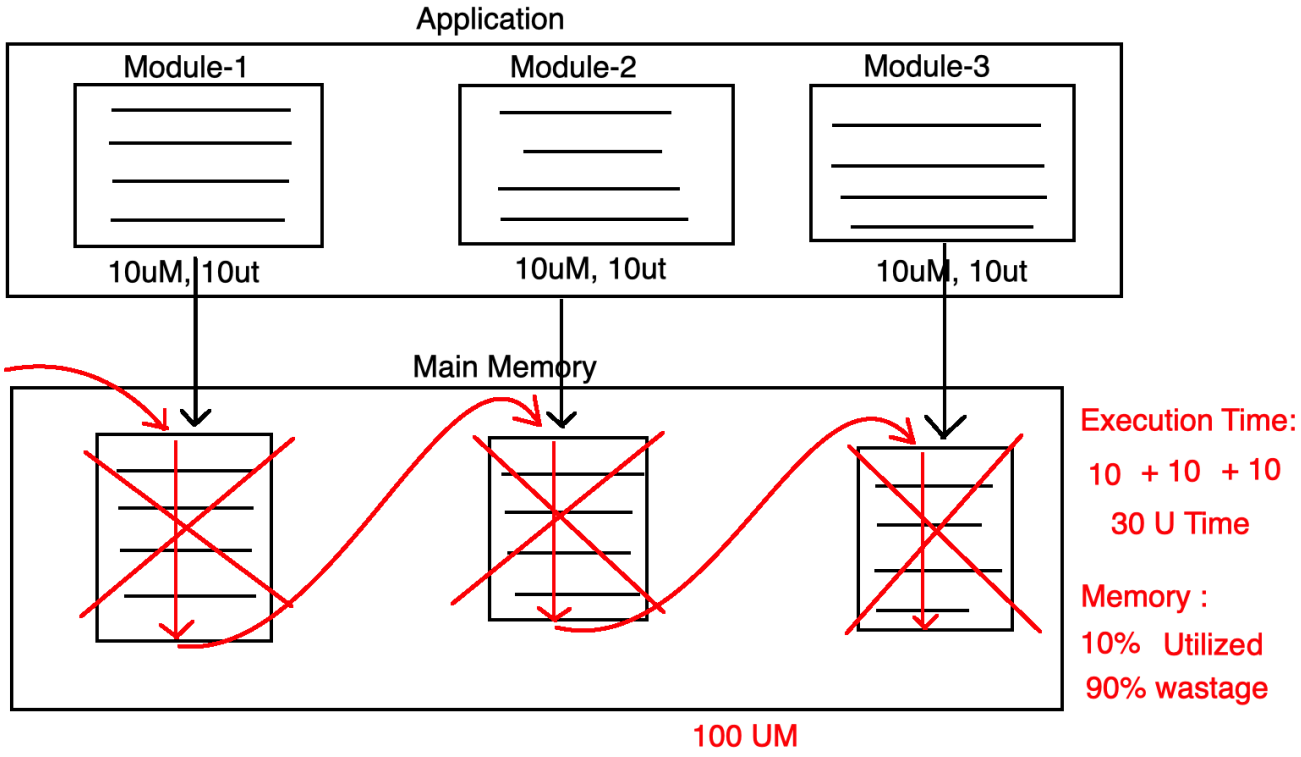


At the starting point of Computer Science, there are two mechanisms to execute the applications.

1. Single Process Mechanism.
2. Multi Process Mechanism.

Single Process Mechanism:
—-------------------------
It is also called Single Tasking.

In Single Process Mechanism, a single process will be created to execute the complete application, even if we have multiple modules or code segments a single process can execute all the modules in sequential manner.

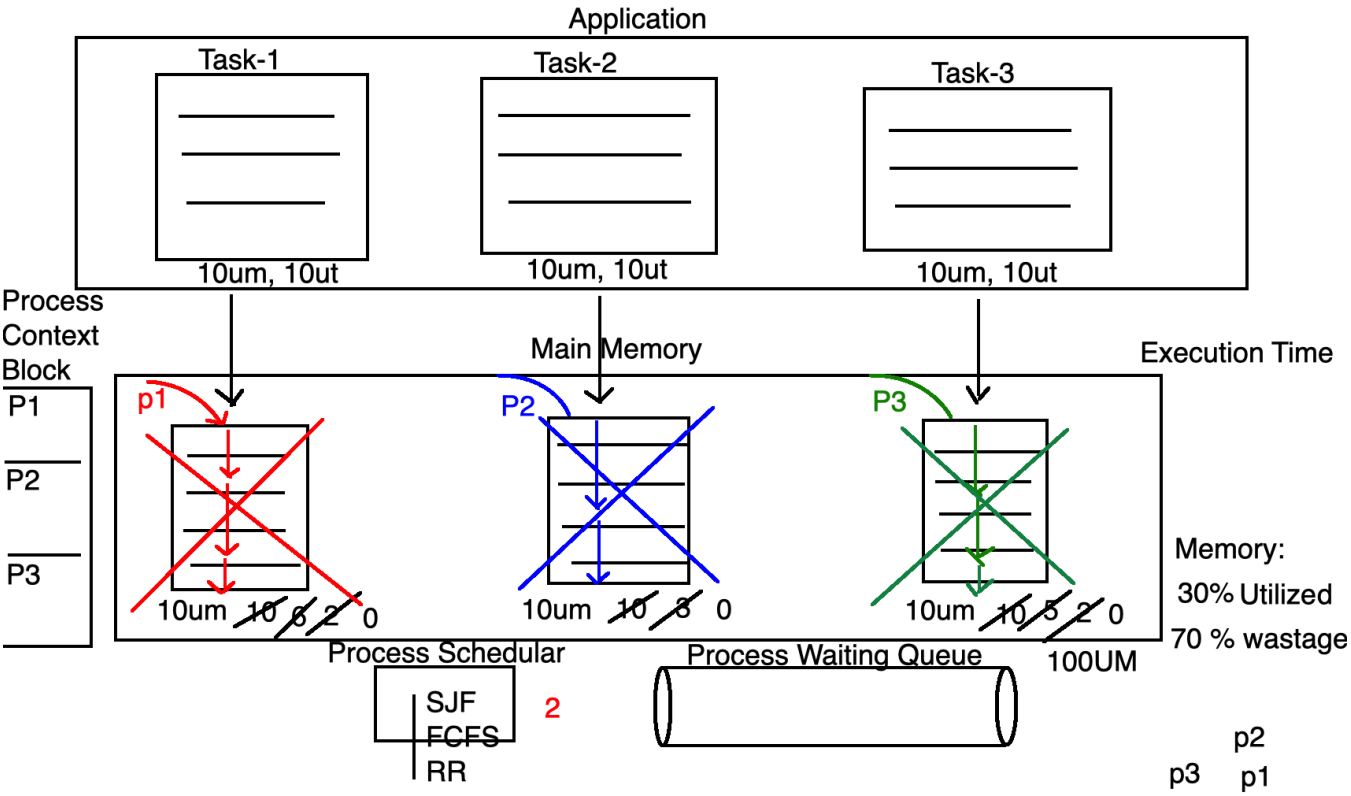


It allows only one Process at a time.
It follows sequential execution.
It increases application execution time.
It reduces the application performance.
It wastes the system resources like main memory.

Multi Process Mechanism:
—----------

It is also called Multi Tasking.
It allows more than one process at a time to perform more than one task, even if we have more than one module or more

than one code segment then the Multi Process system allows a separate process for each and every task.



1. It allows more than one process at a time to execute more than one task.
2. It follows parallel execution.
3. It reduces the application execution time.
4. It improves the application performance.
5. It utilizes the system resources like memory effectively.

In the above Multi tasking, controlling navigated from one process context to another process context, it is called Context switching.

There are two types of context switchings.

1. Heavyweight COntext switching.
2. lightweight COntext Switching

Heavyweight COntext switching:
—----------------------------
It provides the context switching between two heavyweight components.

EX: Context switching between two processes.

Lightweight Context Switching:
—----------------------------
It provides context switching between two lightweight components.

EX: Context switching between two Threads.

Q)What is the difference between process and Thread?
—---
Process is a heavyweight component, it takes more execution time and more memory, it reduces the application performance.

Thread is lightweight, it takes less execution time and less memory, it improves application performance.

There are two thread models to execute the applications.

1. Single thread Model
2. Multi Thread Model

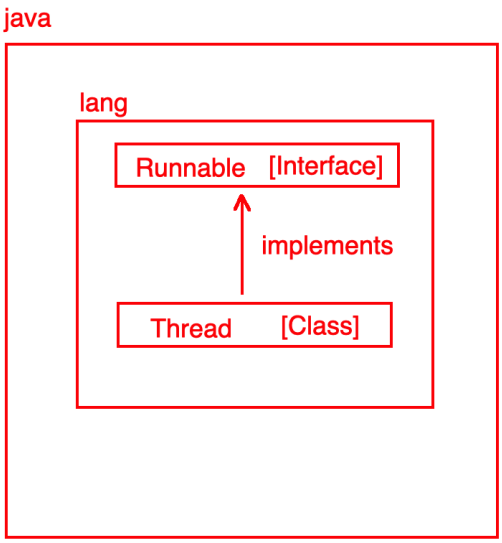
1. Single Thread Model:
   a. It allows only one thread at a time to execute the application.
   b. It follows sequential execution.
   c. It increases the application execution time.
   d. It reduces the application performance.

2. Multi Thread Model:
   a. It allows more than one thread at a time to execute the application.
   b. It follows parallel execution.
   c. It reduces the application execution time.

d.It improves the application performance.

Java is a Multi Threaded programming language, it allows more than one thread at a time to execute the code.

In Java applications, to create Threads Java has provided the following predefined library.



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

—----------

Ans:

—---

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

As per the predefined library provided by JAVA there are two ways to create Threads.

1.Extending Thread class.
2.Implementing Runnable interface.

1. Threads design by extending Thread class:

—----------

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

   public void run()

4. In the Main class and in the main() method, create an object for the user defined thread class and access start() method.

   In JAVA/J2EE applications only the start() method is able to create a new thread, no other method is able to create thread.

EX:

```
class WelcomeThread extends Thread {
   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 wt = new WelcomeThread();
       wt.start();
   }
}
```

```
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
```

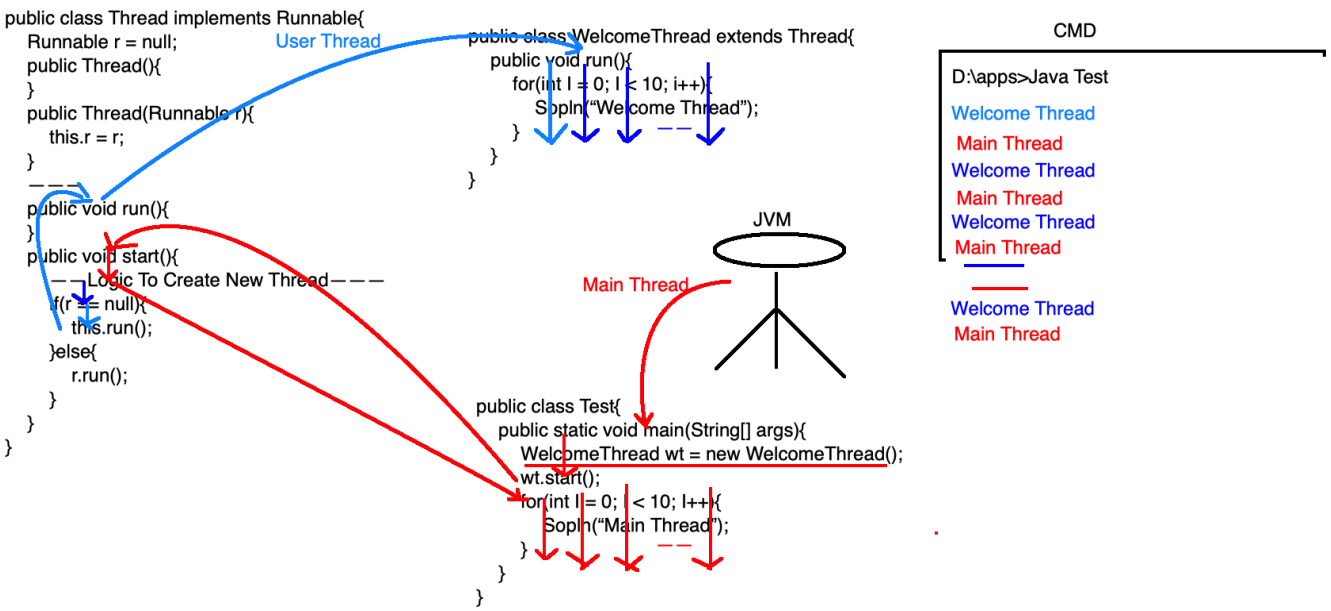
```
Welcome To Durga Software Solutions
Welcome To Durga Software Solutions
Welcome To Durga Software Solutions
```

EX:

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

```
Main Thread
Main Thread
Main Thread
Main Thread
Main Thread
Welcome Thread
Welcome Thread
Welcome Thread
Welcome Thread
Welcome Thread
Welcome Thread
Welcome Thread
Welcome Thread
Welcome Thread
Main Thread
Main Thread
Main Thread
Welcome Thread
```

Main Thread
Main Thread



Q)In Java applications, we already have the first approach of creating threads that is extending Thread class then what is the requirement to go for Second approach that is implementing Runnable interface?

Ans:

In the first approach of creating a thread , we will take a user defined class and we will extend the user defined class from java.lang.Thread class, so here there is no chance of extending other classes.

To create a Frame in GUI applications we will extend java.awt.Frame to a class.

```
public class MyClass extends Frame{
}
```

To create Threads in First approach we will extend java.lang.Thread to an user defined class.

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

If we want to provide a class with Thread and Frame capabilities then we have to extend Frame and Thread in a user defined class.

```
public class  MyClass extends Thread, Frame{
    —---
}
```

It represents Multiple Inheritance, it is not possible in Java, to overcome this problem we have to use the second approach of creating threads.

In the second approach of creating Threads, we will take an user defined class, it must implement java.lang.Runnable interface .

```
public class MyClass extends Frame implements Runnable{
    —----
}
```

Threads Design by Implementing Runnable Interface:

—---

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

```
class MyThread implements Runnable{
    public void run(){
        —--Appl Logic—---
    }
}
```
To create a new thread and to access the run() method we will use the following cases.

Case#1:
```
MyThread mt = new MyThread();
mt.start();
```

Status: Compilation Error.
Reason: start() method does not exist in MyThread class and in Runnable interface.

Case#2:
```
MyThread mt = new MyThread();
mt.run();
```
Status: No Compilation Error, only Main thread executes main() method and run() method, there is no Multi Threading.

Case#3:
```
MyThread mt = new MyThread();
Thread t = new Thread();
t.start();
```
Status: No Compilation Error, no Exception, here a new thread is created, but it executes Thread class run() method only, it does not execute MyThread class run() method.

Case#4:
```
MyThread mt = new MyThread();
Thread t = new Thread(mt);
t.start();
```
EX:
```
class WelcomeThread implements Runnable {
```

```
    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();*/

        /*WelcomeThread welcomeThread = new WelcomeThread();
        welcomeThread.run();*/

        /*WelcomeThread welcomeThread = new WelcomeThread();
        Thread thread = new Thread();
        thread.start();*/

        WelcomeThread welcomeThread = new WelcomeThread();
        Thread thread = new Thread(welcomeThread);
        thread.start();

    }
}
```

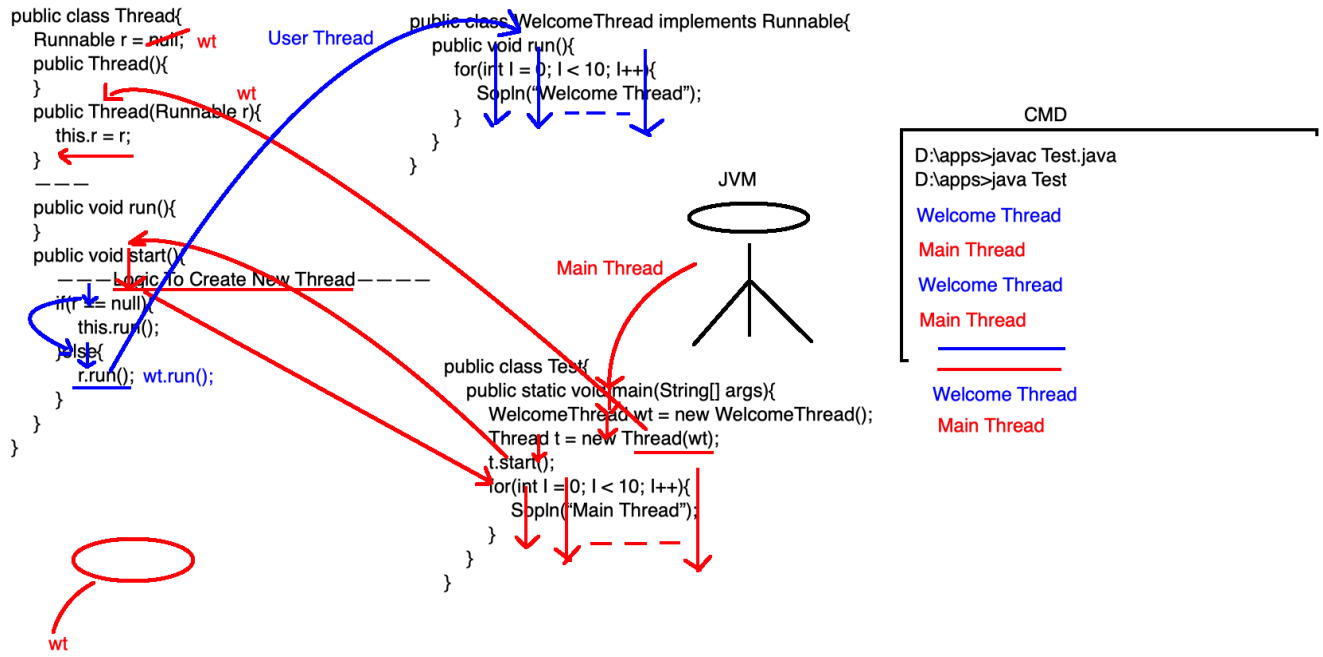
```
Welcome To Durga Software Solutions
Welcome To Durga Software Solutions
Welcome To Durga Software Solutions
Welcome To Durga Software Solutions
Welcome To Durga Software Solutions
Welcome To Durga Software Solutions
Welcome To Durga Software Solutions
Welcome To Durga Software Solutions
Welcome To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

EX:

```
class WelcomeThread implements Runnable {
```

```
    public void run() {
        for(int i = 0; i < 10; i++){
            System.out.println("User Thread");
        }
    }
}
public class Main {
    public static void main(String[] args) {

        WelcomeThread welcomeThread = new WelcomeThread();
        Thread thread = new Thread(welcomeThread);
        thread.start();

        for (int i = 0; i < 10; i++) {
            System.out.println("Main Thread");
        }

    }
}
```

```
Main Thread
Main Thread
Main Thread
Main Thread
Main Thread
User Thread
User Thread
User Thread
User Thread
User Thread
User Thread
User Thread
User Thread
User Thread
User Thread
User Thread
Main Thread
Main Thread
Main Thread
```



Thread Lifecycle:

The collective information of a Thread right from its starting point to the ending point is called Thread Lifecycle.

In Java applications, Every thread will have the following lifecycle states.

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

New / Born State:

In Java applications, when we create a Thread class object , automatically Thread will come to the New / Born State.

Runnable State:
—---------------
In Java Applications, the state when we access start() and before assigning the system resources like memory and time is called Runnable State.

Running State:
—---------------
In Java applications, the state when we access the start() method and after getting the system resources is called Running state.

Blocked State:
—---------------
In Java applications, Blocked State is possible with the following cases.

1. When we access sleep() method with sleep time automatically running thread will come to Sleep state that is Blocked State.
   When sleep time is over then the Thread will come back to the Runnable state.

2. When we access the wait() method over the running thread automatically the Running thread will come to the Waiting state that is Blocked State.

   If any Other Thread access notify() or notifyAll() methods then the waiting thread will come back to the Runnable state.

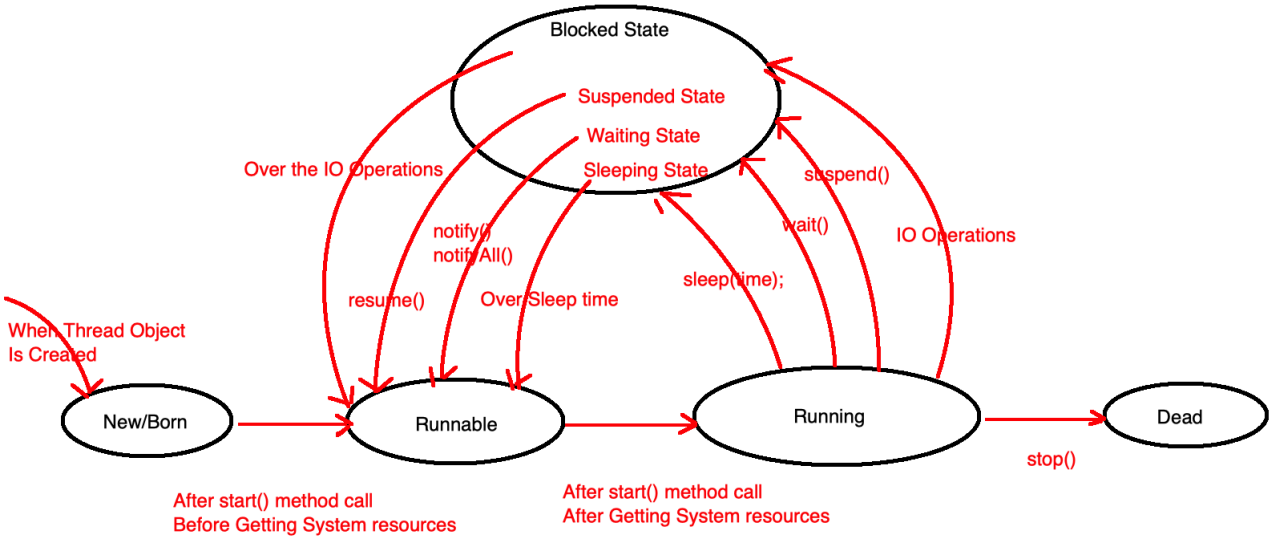
3. When we access the suspend() method over the Running thread, automatically the Running thread will come to the Suspended state that is Blocked State.

   If any other thread access resume() method then the suspended thread will come back to the Runnable state.

4. When we perform IO operations in Java applications automatically the Running thread will come to the Blocked State.

   When IO operations are completed automatically the Blocked Tread will come back to the Runnable state.

Dead State:
—----------
When we access the stop() method over the running thread automatically Thread will come to the Dead State.



Thread class Library:
—---------------------
Constructors:
1. public Thread():
It is able to create Thread class objects with the default thread properties.

Thread Name : Thread-0
Thread Priority : 5
Thread Group : main

EX:

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

```
Thread[#29,Thread-0,5,main]
#29 is newly generated name
Thread-0 is automatically generated name
5 is Thread priority
Main is Thread Group Name
```

2. Public Thread(String name):
It is able 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[#29,CORE JAVA,5,main]
```

3. public Thread(Runnable r):
It is able to create a Thread class object with the provided Runnable reference.

EX:

```
public class Main {
   public static void main(String[] args) {
       Runnable runnable = new Thread();// Thread-0
       Thread thread = new Thread(runnable);// Thread-1
```

```
        System.out.println(thread);//Thread[#29,Thread-1,5,
main]
    }
}
```

```
Thread[#30,Thread-1,5,main]
```

4. public Thread(Runnable r, String name):
It is able to create a Thread class object with the provided Runnable reference and the provided Thread name.

EX:

```
public class Main {
    public static void main(String[] args) {
        Runnable runnable = new Thread();// Thread-0
        Thread thread = new Thread(runnable, "CORE JAVA");//
Thread-1
        System.out.println(thread);//Thread[#29,Thread-1,5,
main]
    }
}
```

```
Thread[#30,CORE JAVA,5,main]
```

5. public Thread(ThreadGroup tg, String name):
It is able to create a Thread class object with the provided Thread Group name and the thread name.
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[#29,CORE JAVA,5,JAVA]
```

6. public Thread(ThreadGroup tg, Runnable r):
It is able to create Thread object with the TreadGroup name and the provided Runnable reference.

EX:

```
public class Main {
   public static void main(String[] args) {
       ThreadGroup threadGroup = new ThreadGroup("JAVA");
       Runnable runnable = new Thread();
       Thread thread = new Thread(threadGroup, runnable);
       System.out.println(thread);
   }
}
Thread[#30,Thread-1,5,JAVA]
```

7. public Thread(ThreadGroup tg, Runnable r, String name):
It is able to create Thread class objects with the ThreadGroup name, Runnable reference and the Thread name.

EX:

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

```
Thread[#30,CORE JAVA,5,JAVA]
```

Methods:
—--------
1. public void setName():

To set a particular name to the Thread.

2. public String getName():
To get the name of the Thread.

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

```
Thread-0
CORE JAVA
```

3. public void setPriority(int priorityValue)
To set a particular priority value to the Thread.

Note: There is a boundary for the priority values that is from 1 to 10 , because threads priority range is 1 to 10. If we provide any value outside of the range from 1 to 10 then JVM will raise an exception like java.lang.IllegalArgumentException.

4. public int getPriority():
It is able to get the Priority value of the Thread.

IN Java applications, to represent Priority values java.lang.Thread class has provided the following constants.

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

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());
       thread.setPriority(Thread.MAX_PRIORITY-2);
       System.out.println(thread.getPriority());

thread.setPriority(Thread.MIN_PRIORITY+Thread.NORM_PRIORITY
);
       System.out.println(thread.getPriority());
       thread.setPriority(15);
   }
}
```

```
5
7
8
6
java.lang.IllegalArgumentException
```

5. public static int activeCount():
It is able to return the number of Active threads in the present java application.

EX:

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

```
2
```

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 will return a Thread object which is running currently.

EX:

```
class A{
    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;
    }
    public void run(){
        a.m1();
    }
}
class Thread2 extends Thread{
```

```
    A a;
    Thread2(A a){
        this.a = a;
    }
    public void run(){
        a.m1();
    }
}
class Thread3 extends Thread{
    A a;
    Thread3(A a){
        this.a = a;
    }
    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-1
THREAD-1
THREAD-1
THREAD-1
THREAD-1
```

```
THREAD-1
THREAD-1
THREAD-1
THREAD-1
THREAD-2
THREAD-2
THREAD-2
THREAD-2
THREAD-2
THREAD-2
THREAD-2
THREAD-2
THREAD-3
THREAD-3
THREAD-3
THREAD-3
THREAD-3
THREAD-3
THREAD-3
THREAD-3
THREAD-3
THREAD-1
THREAD-2
THREAD-2
THREAD-2
THREAD-3
THREAD-3
```

8. public static void sleep(int time)throws InterruptedException
It is able to keep a running thread in sleeping state up to the specified sleep time.Once Sleep time is completed it will resume its execution.

EX:

```
class WelcomeThread extends Thread {
   public void run(){
       try{
           for(int i = 0; i < 10; i++) {
               Thread.sleep(1000);
               System.out.println("Welcome to Durgasoft!");
```

```
            }
        }catch (Exception 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 is able to pause the current thread execution and complete the other thread execution, after completion of the other thread execution it will continue the paused thread execution.

EX:

```
class WelcomeThread extends Thread {
    public void run(){
        for(int i = 0; i < 10; i++){
            System.out.println("Welcome Thread : "+i);
        }
    }
}
public class Main {
    public static void main(String[] args) throws
InterruptedException {
        WelcomeThread wt = new WelcomeThread();
        wt.start();
        wt.join();// Pause the Main Thread until the
WelcomeThread completion
        for(int i = 0; i < 10; i++){
            System.out.println("Main Thread : "+i);
        }
    }
}
```

```
Welcome Thread : 0
Welcome Thread : 1
Welcome Thread : 2
Welcome Thread : 3
Welcome Thread : 4
Welcome Thread : 5
Welcome Thread : 6
Welcome Thread : 7
Welcome Thread : 8
Welcome Thread : 9
Main Thread : 0
Main Thread : 1
Main Thread : 2
Main Thread : 3
Main Thread : 4
Main Thread : 5
Main Thread : 6
Main Thread : 7
Main Thread : 8
Main Thread : 9
```

Daemon Threads:
—--------------
Daemon thread is a thread which executes internally and provides services to some other threads.

In Java applications, Daemon thread will be terminated automatically when the thread which is taking services terminated.

EX: Garbage Collector is a daemon thread, it will execute internally and it will Garbage Collection service to the JVM.

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

public void setDaemon(boolean b):

If the parameter value is true then the Thread will be a daemon thread.
If the parameter value is false then the thread will not be daemon thread.

Note: If we want to make a thread as a daemon thread then we have to access setDaemon() method before accessing start() method, if we access it after accessing start() method then JVM will raise an exception like java.lang.IllegalThreadStateException.

To check whether a thread is a Daemon thread or not we have to use the following method.

public boolean isDaemon()

EX
—-

```
class GarbageCollector extends Thread {
   public void run() {
       while (true) {
           System.out.println("Garbage Collector Thread");
       }
   }
}
public class Main {
   public static void main(String[] args) throws
InterruptedException {
       GarbageCollector gc = new GarbageCollector();
       gc.setDaemon(true);
       gc.start();
       //gc.setDaemon(true);--> IllegalThreadStateException
       System.out.println(gc.isDaemon());
       for (int i = 0; i < 10; i++) {
           System.out.println("Main Thread");
       }
   }
```

```
}

Main Thread
Main Thread
Main Thread
Main Thread
Main Thread
Main Thread
Garbage Collector Thread
Garbage Collector Thread
Garbage Collector Thread
Garbage Collector Thread
Garbage Collector Thread
Garbage Collector Thread
Garbage Collector Thread
Main Thread
Main Thread
Main Thread
Garbage Collector Thread
Garbage Collector Thread
Garbage Collector Thread
Garbage Collector Thread
Garbage Collector Thread
Garbage Collector Thread
Garbage Collector Thread
Garbage Collector Thread
Garbage Collector Thread
Garbage Collector Thread

Thread safe Resources:
—----------------------
If we execute more than one thread on the same data item or
on the same program then those threads are called
Concurrent Threads and that process is called Threads
concurrency.

When more than one thread executing on the same data item
then there is a chance of getting data inconsistency,
```

because one thread execution may affect the other thread execution

In the above context, if any resource is able to allow more than one thread execution 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 in place of Mutable objects.
3. Use Synchronization

Use Local Variables instead of Class Level Variables:
—--
In Java applications, class level variables data will be stored in the Heap Memory, it is common for all the threads execution, here one thread execution may modify the data and it will impact to all other threads execution, it will show data inconsistency.

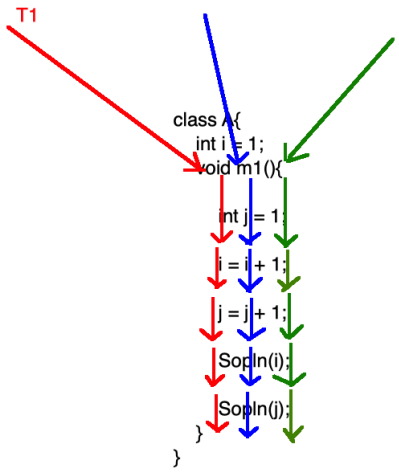
In Java applications, local variables data will be stored in the Stack memory, it is individual to all the threads, if one thread modifies the data then it will be available up to the respective threads stack only, it will not come to the other threads, it will show the Data Consistency.

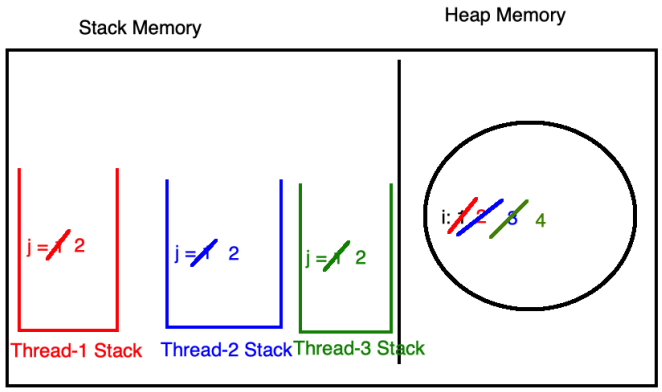
```
class A{
   int i = 1;
   void m1(){
       int j = 1;
       i = i + 1;// 2, 3, 4
       j = j + 1;// T1: 2, T2: 2, T3: 2
       System.out.println(Thread.currentThread().getName()
+" : i value is: " + i);
```

```
        System.out.println(Thread.currentThread().getName()
+" : j value is: " + j);
    }
}
class Thread1 extends Thread{
    A a;
    public Thread1(A a){
        this.a = a;
    }
    public void run(){
        a.m1();
    }
}
class Thread2 extends Thread{
    A a;
    public Thread2(A a){
        this.a = a;
    }
    public void run(){
        a.m1();
    }
}
class Thread3 extends Thread{
    A a;
    public Thread3(A a){
        this.a = a;
    }
    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 : i value is: 4
Thread-2 : i value is: 3
Thread-1 : i value is: 2
Thread-3 : j value is: 2
Thread-1 : j value is: 2
Thread-2 : j value is: 2
```





T1: i : 4
T2: i : 3
T3: i : 2

T1 : j : 2
T2 : j : 2
T3 : j : 2

2. Use Immutable objects instead of Mutable Objects:

In Java, Immutable objects are the Java objects which will not allow modifications in their content, If we perform modifications on the immutable objects content then the data is allowed for the operations but the resultant

modified data will not be stored back in the original object, where the resultant modified data will be stored by creating a new object.

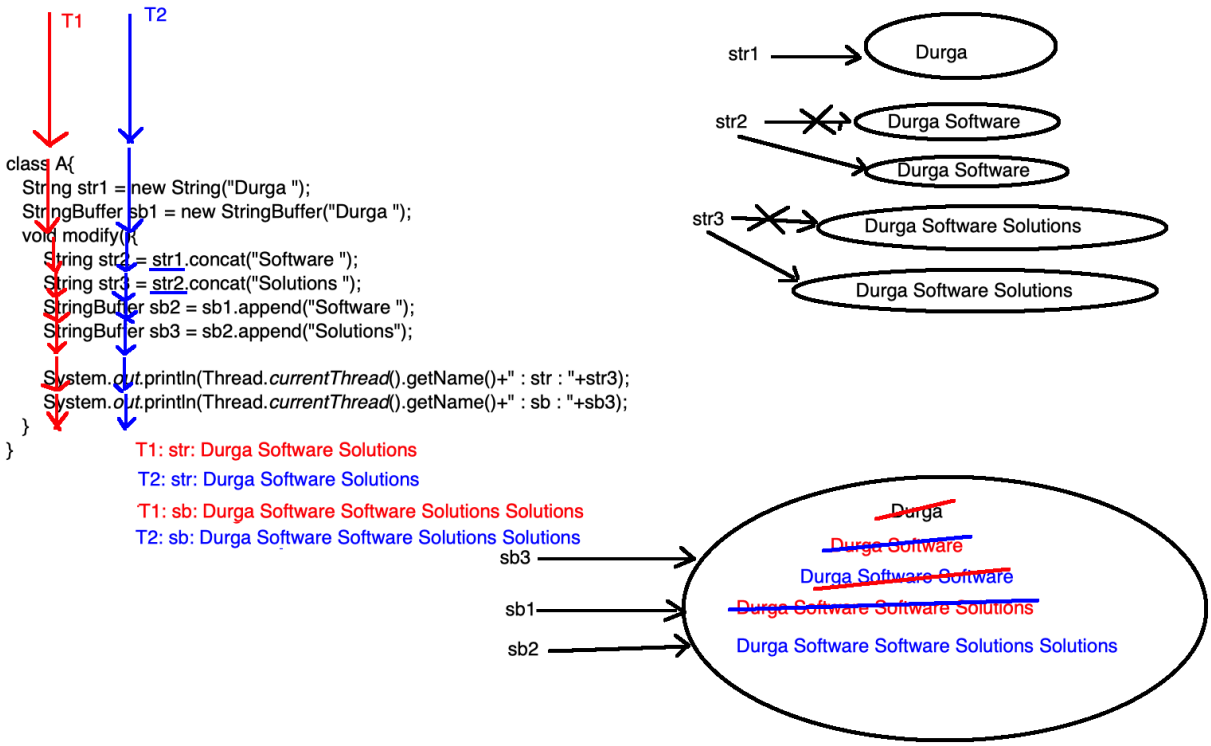
If we send multiple threads to the immutable objects and the multiple threads perform operations on the immutable object then every time a new object will be created when the data is changed, this nature of the Immutable object will provide the Data consistency and it will make the resource a Thread Safe resource.

Mutable object is able to allow the modifications in its content directly. If we send more than one thread to the mutable object and if all the threads perform modifications on the mutable object data then all the modifications are performed on the same object, this nature of the Mutable objects will provide the Data inconsistency and it will make the resource a non thread safe resource.

EX:

```
class A{
   String str1 = new String("Durga ");
   StringBuffer sb1 = new StringBuffer("Durga ");
   void modify(){
       String str2 = str1.concat("Software ");
       String str3 = str2.concat("Solutions ");
       StringBuffer sb2 = sb1.append("Software ");
       StringBuffer sb3 = sb2.append("Solutions");

       System.out.println(Thread.currentThread().getName()
+" : str : "+str3);
       System.out.println(Thread.currentThread().getName()
+" : sb : "+sb3);
   }
}
class Thread1 extends Thread{
   A a;
   Thread1(A a){
       this.a = a;
```

```
    }
    public void run(){
        a.modify();
    }
}
class Thread2 extends Thread{
    A a;
    Thread2(A a){
        this.a = a;
    }
    public void run(){
        a.modify();
    }
}
public class Main {
    public static void main(String[] args){
        A a = new A();
        Thread1 t1 = new Thread1(a);
        Thread2 t2 = new Thread2(a);
        t1.setName("THREAD-1");
        t2.setName("THREAD-2");
        t1.start();
        t2.start();
    }
}

THREAD-1 : str : Durga Software Solutions
THREAD-2 : str : Durga Software Solutions
THREAD-1 : sb : Durga Software SolutionsSoftware Solutions
THREAD-2 : sb : Durga Software SolutionsSoftware Solutions
```

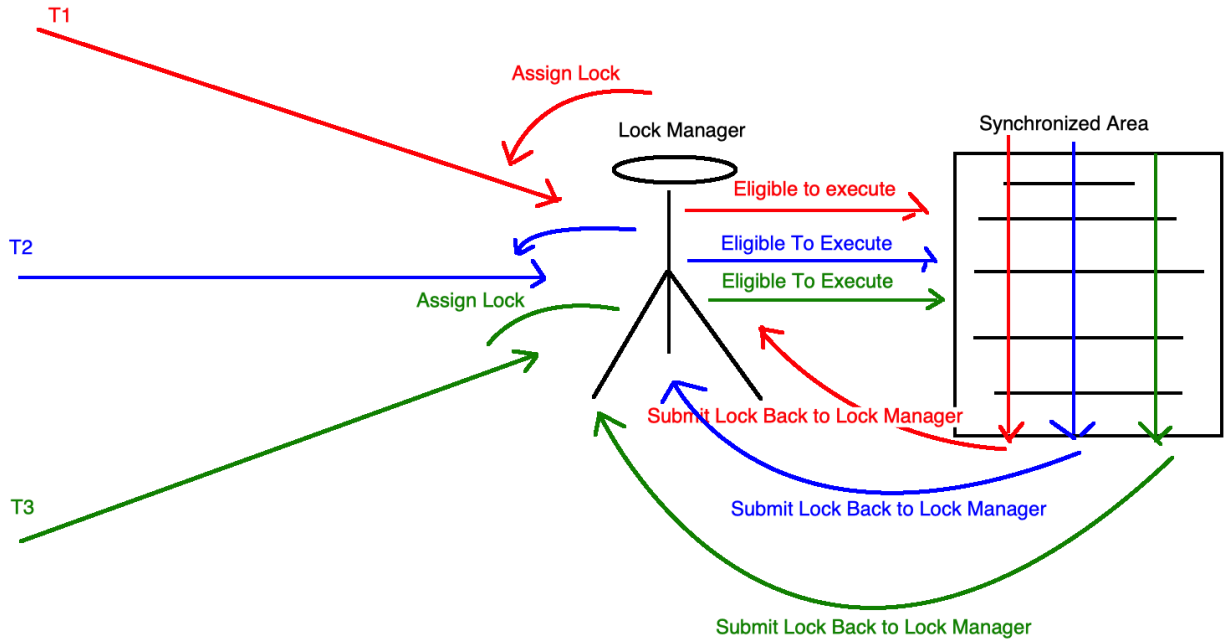


Synchronization:

—----------------

Synchronization is a mechanism, it allows only one thread at a time to execute the program, after executing the program by the present thread it will allow other threads which are waiting.

In Java applications, Synchronization is working on the basis of the locking mechanisms.



When we submit multiple Threads to the Synchronized area, Lock Manager will keep all the threads in waiting state, it will assign Lock to a thread which is getting highest priority as per the Scheduling algorithm, if the thread which is getting lock is eligible to execute the code under synchronization, once the thread completes its execution on the synchronized area automatically the it will return the Lock to the Lock Manager, here the Lock Manager will assign lock to the next highest priority thread and make eligible to execute synchronized area.

The above process will be performed until all the threads execute in the Synchronized area.

In the Synchronization, at a time only one thread is executing, there is no chance of getting data inconsistency, by default synchronization will make a program thread safe.

In Java applications, to achieve synchronization we will use the synchronized keyword.

There are two ways to achieve synchronization in Java applications.

1. Synchronized Method
2. Synchronized Block

Synchronized Method:
—--------------------
It is a set of instructions, it is able to allow only one thread at a time, it will not allow more than one thread at a time, it allows the other threads after completion of the present thread execution.

EX:

```
class A{
   synchronized void m1(){
       for(int i = 0; i < 10; i++){

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

```
    }
}
class Thread3 extends Thread{
    A a;
    Thread3(A a){
        this.a = a;
    }
    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-1 : 0
THREAD-1 : 1
THREAD-1 : 2
THREAD-1 : 3
THREAD-1 : 4
THREAD-1 : 5
THREAD-1 : 6
THREAD-1 : 7
THREAD-1 : 8
THREAD-1 : 9
THREAD-3 : 0
```

```
THREAD-3 : 1
THREAD-3 : 2
THREAD-3 : 3
THREAD-3 : 4
THREAD-3 : 5
THREAD-3 : 6
THREAD-3 : 7
THREAD-3 : 8
THREAD-3 : 9
THREAD-2 : 0
THREAD-2 : 1
THREAD-2 : 2
THREAD-2 : 3
THREAD-2 : 4
THREAD-2 : 5
THREAD-2 : 6
THREAD-2 : 7
THREAD-2 : 8
THREAD-2 : 9
```

Q) In Java applications, we have already synchronized methods to achieve synchronization then what is the requirement to use Synchronized blocks?

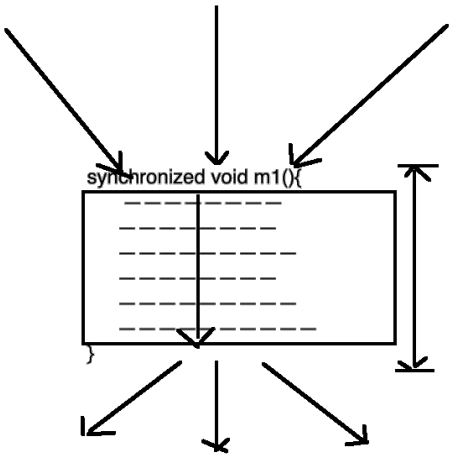
—----------

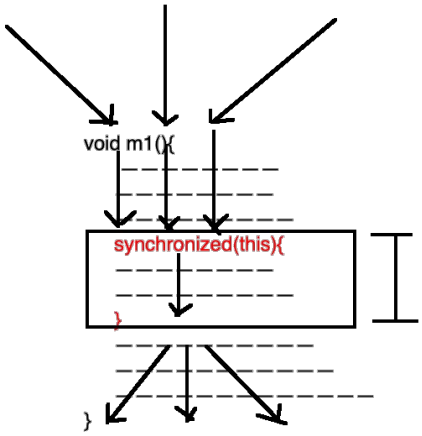
Ans:

—---

In Java applications, to achieve synchronization if we use synchronized method then it will provide synchronization throughout the method irrespective of the actual requirement, it will increase application execution time and it will reduce the a
application performance.

To overcome the above problem we have to use Sycnrhmized block, it will provide the synchronization up to the required block of instructions.





Synchronized Block:
—--------------------
It is a set of instructions, it is able to allow only one thread at a time , not to allow more than one thread at a time, it allows other threads after executing the present Thread.

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;
    }
    public void run(){
        a.m1();
    }
}
class Thread2 extends Thread{
    A a;
    Thread2(A a){
        this.a = a;
    }
    public void run(){
        a.m1();
    }
}
class Thread3 extends Thread{
    A a;
    Thread3(A a){
        this.a = a;
    }
    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();
    }
}
Before Synchronized Block : THREAD-1
Before Synchronized Block : THREAD-2
Before Synchronized Block : THREAD-3
Inside Synchronized Block : THREAD-1
Inside Synchronized Block : THREAD-1
Inside Synchronized Block : THREAD-1
Inside Synchronized Block : THREAD-1
Inside Synchronized Block : THREAD-1
Inside Synchronized Block : THREAD-1
Inside Synchronized Block : THREAD-1
Inside Synchronized Block : THREAD-1
Inside Synchronized Block : THREAD-1
Inside Synchronized Block : THREAD-1
Inside Synchronized Block : THREAD-3
Inside Synchronized Block : THREAD-3
Inside Synchronized Block : THREAD-3
Inside Synchronized Block : THREAD-3
Inside Synchronized Block : THREAD-3
Inside Synchronized Block : THREAD-3
Inside Synchronized Block : THREAD-3
Inside Synchronized Block : THREAD-3
Inside Synchronized Block : THREAD-3
Inside Synchronized Block : THREAD-3
Inside Synchronized Block : THREAD-2
Inside Synchronized Block : THREAD-2
Inside Synchronized Block : THREAD-2
Inside Synchronized Block : THREAD-2
Inside Synchronized Block : THREAD-2
Inside Synchronized Block : THREAD-2
Inside Synchronized Block : THREAD-2
Inside Synchronized Block : THREAD-2
Inside Synchronized Block : THREAD-2
Inside Synchronized Block : THREAD-2
```

Note: In Java applications, Synchronization is not suggestible , because it is able to reduce the application performance, if need the synchronization really then use it otherwise not suggestible.

Inter Thread Communication:
—----------------------------
The process of providing communication between more than one thread is called Inter Thread Communication.

In Java applications, to perform a task effectively with more than one thread then it is suggestible to provide communication between threads.

In Java applications, Inter Thread communication requires the following methods.

1. Public void wait(): To keep the thread running in the waiting state.
2. public void notify(): It will give activation state to the waiting thread.
3. Public void notifyAll(): It will provide activation state to all the threads which are in waiting state.

All the above methods require Synchronization, without synchronization they are not performing their functionalities.

IN general Inter Thread communication is providing solutions for problems like the Producer-Consumer problem.

Producer and COnsumer are two threads, where producer has to produce an item and consumer has to consume that item, it has to be repeated an infinite number of times, where producer must not produce a new item without consuming previous item by the consumer and the consumer must not consume an item without producing that item.

EX:

```
class A{
   boolean flag = true;
   int itemCount = 0;

   public synchronized void produce(){
       try{
           while (true){
               if(flag == true){
                   itemCount = itemCount + 1;
                   System.out.println("Producer Produced :
" + itemCount);
                   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 :
" + itemCount);
                   flag = true;
                   notify();
                   wait();
               }
           }
       }catch (Exception e){
           e.printStackTrace();
       }
```

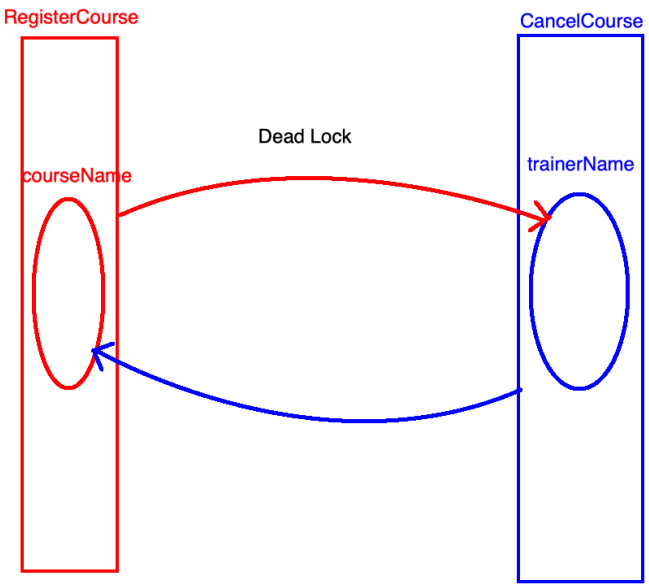
```
    }
}
class Producer extends Thread{
    A a;
    public Producer(A a){
        this.a = a;
    }
    public void run(){
        a.produce();
    }
}
class Consumer extends Thread{
    A a;
    public Consumer(A a){
        this.a = a;
    }
    public void run(){
        a.consume();
    }
}
public class Main {
    public static void main(String[] args){
        A a = new A();
        Producer p = new Producer(a);
        Consumer c = new Consumer(a);
        p.start();
        c.start();
    }
}

OP:
Producer Produced : 1
Consumer Consumed : 1
Producer Produced : 2
Consumer Consumed : 2
Producer Produced : 3
Consumer Consumed : 3
-----
-----
```

Deadlocks:
—----------
Deadlock is a situation where the program execution is struck.

In general, in java applications if we have a deadlock then there is no recovery mechanism to come back to the original state of the program, we have only precautions.

EX: If A and B are two threads, where A depends on X resources which is locked by B thread and B thread depends on the Y resource which is locked by the A thread, Thread A will not release the Y resource without getting X resource from B thread and The Thread B will not release X resource without getting Y resource from Thread A, this situation is representing Deadlock.

EX:



EX:

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

```
    public RegisterCourse(Object courseName, Object
trainerName) {
        this.courseName = courseName;
        this.trainerName = trainerName;
    }
    public void run() {
        synchronized (courseName) {
            System.out.println("RegisterCourse Holds
courseName resource and waiting for trainerName
resource......");
            synchronized (trainerName) {
                System.out.println("RegisterCourse Holds
CourseName and TrainerName resource, so Course Registration
Successful......");
            }
        }
    }
}
class CancelCourse extends Thread {
    Object courseName;
    Object trainerName;

    public CancelCourse(Object courseName, Object
trainerName) {
        this.courseName = courseName;
        this.trainerName = trainerName;
    }
    public void run() {
        synchronized (trainerName) {
            System.out.println("CancelCourse Holds
trainerName resource and waiting for courseName
resource......");
            synchronized (courseName) {
                System.out.println("CancelCourse holds
trainerName and courseName resource, so Course Cancellation
Successful......");
            }
        }
```

```
   }
}
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();
   }
}
```

```
RegisterCourse Holds courseName resource and waiting for
trainerName resource......
CancelCourse Holds trainerName resource and waiting for
courseName resource......
```

Thread Local:
—--------------
In Java applications, we are able to use the scopes like public, protected, <default> and private for the data.

In java applications, we are able to define our own scope to the data like Thread Scope, here the Thread Scope is able to provide the data up to all the resources[Methods and classes] which are accessed by the thread.

To define the Thread Scope for the data we have to use the java.lang.ThreadLocal class in order to add, remove and get data from the Thread Scope.

In Java applications, to define Thread Scopes to the data we have to use the following steps.

1. Define User defined ThreadLocal class.
   a. Declare an User defined class.
   b. Extend java.lang.ThreadLocal class to the User defined class.
   c. Override the initialValue() method in the user defined class.

      In Threadlocal class we are able to have the following methods.

      set(--): To set data to the Thread Scope.
      get(): To get the data from the Thread Scope
      initialValue(): to provide default values from the Thread
                  Scope, it will provide the values when we access the get() method without accessing set() method.

EX:

```
public class MyThreadLocal extends ThreadLocal{

}
```

2. Create Thread class :
   a. Create Thread class by extending java.lang.Thread class.
   b. Create a MyThreadLocal object with the static reference variable.
   c. Set data to the Thread Scope by using the set() method of the MyThreadLocal class.
   d. In any other method which is accessed from the Thread, get the data from the ThreadLocal by using get() method.

```
public MyThread extends Thread{
    static MyThreadLocal threadLocal = new
MyThreadLocale();
    public void run(){
        threadLocal.set(“Hello This is from MyThread
Scope”);
        a.m1();
    }
}

class A{
    void m1(){
        String msg = MyThread.threadLocal.get();
        System.out.println(msg);
    }
}
```

EX:

```
class MyThreadLocal extends ThreadLocal {
   public Object initialValue() {
       return "No Data is Defined in this Scope";
   }
}
class A{
   void m1(){
       System.out.println("m1() :
"+Thread.currentThread().getName()+" Scope :
"+Thread1.myThreadLocal.get());
       System.out.println("m1() : THREAD-2 Scope :
"+Thread2.myThreadLocal.get());
   }
   void m2(){
```

```
        System.out.println("m2() :
"+Thread.currentThread().getName()+" Scope :
"+Thread2.myThreadLocal.get());
        System.out.println("m2() : THREAD-1 Scope :
"+Thread1.myThreadLocal.get());
    }
}
class Thread1 extends Thread {
    static MyThreadLocal myThreadLocal = new
MyThreadLocal();
    A a;
    Thread1(A a) {
        this.a = a;
    }
    public void run() {
        myThreadLocal.set("Data from Thread1 Scope");
        a.m1();
    }
}
class Thread2 extends Thread {
    static MyThreadLocal myThreadLocal = new
MyThreadLocal();
    A a;
    Thread2(A a) {
        this.a = a;
    }
    public void run() {
        myThreadLocal.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();
    }
}
m1() : THREAD-1 Scope : Data from Thread1 Scope
m1() : THREAD-2 Scope : No Data is Defined in this Scope
m2() : THREAD-2 Scope : Data from Thread2 Scope
m2() : THREAD-1 Scope : No Data is Defined in this Scope


IOStreams:
—----------
```