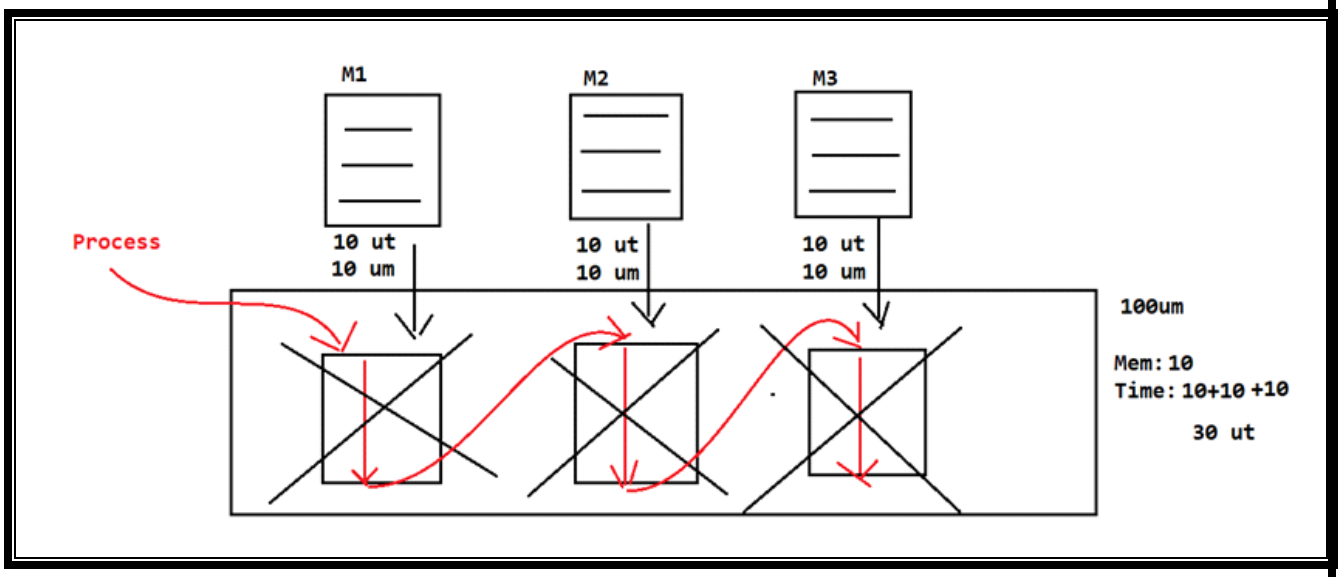
Complete Core Java In Simple Way Part - 2

[Multi Threading]

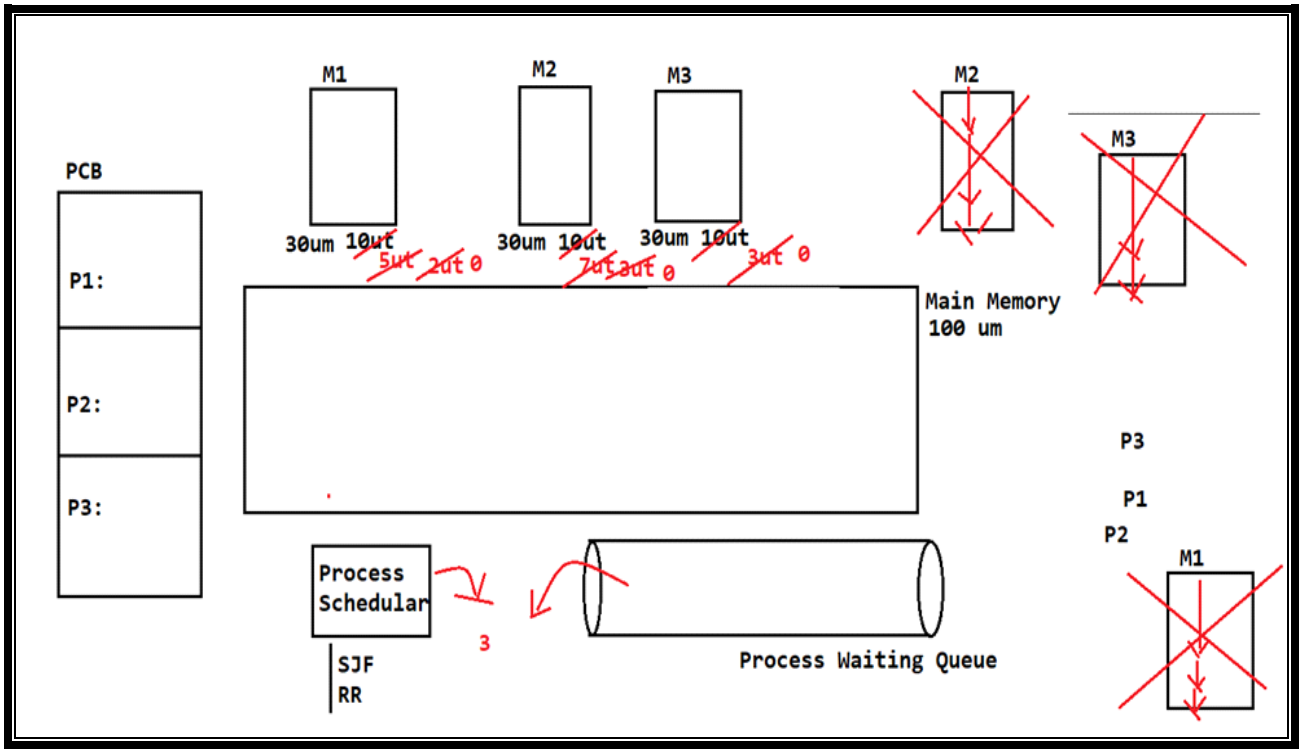
Q) What is the difference between Process, Procedure and Processor?

- Process is a flow of execution to perform a particular task.
- Procedure is a set of instructions to represent a particular task.
- Processor is an H/W component to generate no of processes in order to execute applications.
- At starting point of the computers we have Single Process Mechanism or Single Tasking to execute applications.
- In Single Process Mechanism, System is able to allow only one task at a time to load into the memory even our system main memory is capable to manage all the tasks.



- In Single Process mechanism, System is able to allow only one process to execute all the tasks which are available in our application, it will follow sequential kind of execution, it will increase application execution time and it will reduce application performance.
- To overcome the above problems we have to use Multi Process Mechanism or multi tasking. In Multi tasking system is able to allow to load more than one task at a time in main memory and it able to allow more than one process to execute application, it will follow parallel execution, it will reduce application execution time and it will improve application performance.

To execute applications by using Multi Tasking or Multi Process Mechanism we have to use the following components.



- **Main Memory:** To load all the tasks.
- **Process Waiting Queue:** To keep track of all process
- **Process Context Block:** To manage status of all the processes execution.
- **Process Scheduler:** It will take process from Process Waiting Queue and it will assign time stamps to each and every process in order to execute.

In the above multi processing system, controlling is switched from one process context to another process context called as "Context Switching".
There are two types of Context Switching's.

- **Heavy Weight Context Switching:**

It is the context switching between two heavy weight components, it will take more memory and more execution time, it will reduce application performance.

EX: Cotext switching between two Processes.

- **Light Weight Context Switching:**

It is the context switching between two light weight components, it will take less memory and less execution time and it will improve application performance.
EX: Context switching between two threads.

Q) What is the difference between Process and Thread?

Process is heavy weight, to handle it System has to consume more memory and more execution time, it will reduce application performance.

Thread is light weight, to handle it system has to consume less memory and less execution time, it will improve application performance.

There are two thread models to execute applications.

- **Single Thread Model**
- **Multi Thread Model**

• Single Thread Model:

It will allow only one thread to execute application, it will follow sequential execution, it will increase application execution time and it will reduce application performance.

• Multi Thread Model:

It will allow more than one thread to execute application, it will follow parallel execution, it will reduce application execution time and it will improve application performance.

Java is following Multi Thread Model to execute applications and it will provide very good environment to create and execute more than one Thread at a time.

In java applications, to create Threads JAVA has provided the following predefined library in the form of java.lang package.

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

- **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 in java applications.**

• Extending Thread Class:

In this approach, we have to declare a class, it must be extended from *java.lang.Thread* class.

```
class MyThread extends Thread
{
    --implementation----
}
```

- **Implementing Runnable Interface:**
  In this approach, we have to declare a class, it must implement *java.lang.Runnable* Interface.

```
class MyThread implements Runnable
{
    ---implementation----
}
```

Threads Design in Java:

There are two approaches to create threads in java applications.

- Extending Thread Class
- Implementing Runnable Interface

1) Extending Thread class

- Declare a user defined class.
- Extend java.lang.Thread class to user defined class
- Override Thread class run() method in user defined thread class with the implementation representing a particular task which we want to perform by creating a thread.
- In main class, in main() method, create object for user defined class.
- Access Thread class provided start() method on user defined thread class object reference variable.

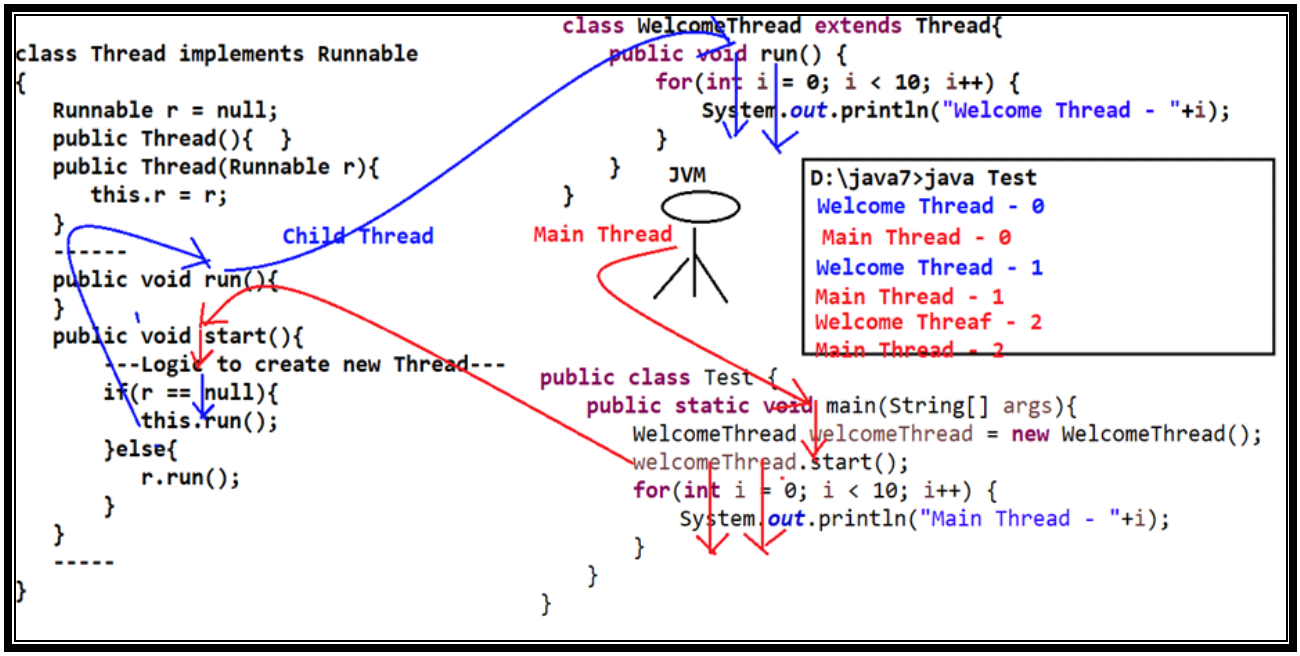
The main intention of start() method is to create new thread and to access run() method by passing the generated thread.

public void start()

EX:

```
1)  class MyThread extends Thread
2)  {
3)     public void run()
4)     {
5)        for(int i=0;i<10;i++)
6)        {
7)           System.out.println("User Thread :"+i);
8)        }
9)     }
10) }
11) class Test
12) {
13)    public static void main(String[] args)
```

```
14)  {
15)     MyThread mt=new MyThread();
16)     mt.start();
17)     for(int i=0;i<10;i++)
18)     {
19)        System.out.println("Main Thread :"+i);
20)     }
21)  }
22)}
```



Q) In Java applications, to create threads we have already first approach [Extending thread class] then what is the requirement to go for Second Approach [implementing Runnable interface]?

In java applications, to create threads if we use first approach then we have to declare an user defined class and it must be extended from java.lang.Thread class, in this context, it is not possible to extend other classes , if we extend any other class like Frame,.. along with Thread class then it will represent Multiple Inheritance, it is not possible in Java.

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

To overcome the above problem, we have to use second approach to create Thread, that is, implementing Runnable Interface.

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

2) Implementing Runnable Interface:

- Declare a user defined class.
- Implement java.lang.Runnable interface.
- Provide implementation part in run() method which we want to execute by creating a thread.
- In main class, in main() method, create a thread and access user defined thread class run() method.

To perform the above step we have to use the following cases.

```
class MyThread implements Runnable {
        public void run() {
                ---
        }
}
```

Case-1:

```
MyThread mt = new MyThread();
mt.start();
```

Status: Compilation Error.

Reason: start() method was not declared in MyThread class and in its super class java.lang.Object class,
start() method is existed in java.lang.Thread class.

Case-2:

```
MyThread mt = new MyThread();
mt.run();
```

Status: No Compilation Error, but, only Main thread access MyThread class run() method like a normal Java method, no multi threading environment.

Case-3:

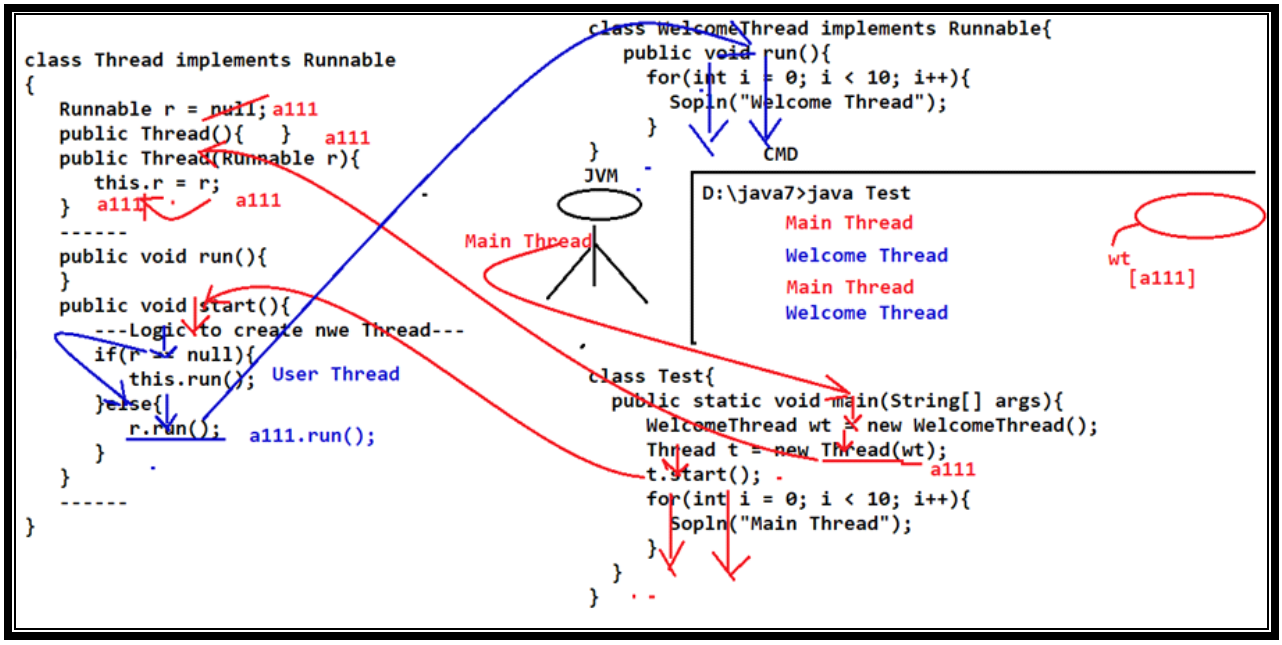
```
MyThread mt = new MyThread();
Thread t = new Thread();
t.start();
```

Status: No Compilation Error, start() method creates new thread and it access Thread class run() method, not MyThread class run() method.

Case-4:

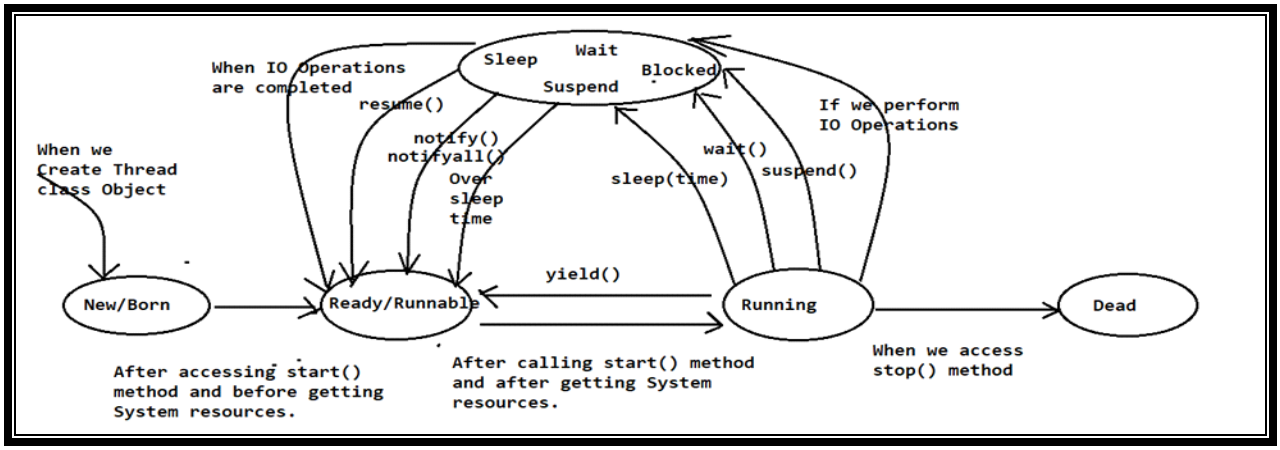
MyThread mt = new MyThread();
Thread t = new Thread(mt);
t.start();

Status: No Compilation Error, start() method creates new thread and it will bypass new thread to MyThread class run() method.



Thread Lifecycle:

- **The collective information of a thread right from its starting point to ending point is called as "Thread Life Cycle".**
- **In Java applications, Threads are able to have the following states as part of their lifecycle.**



- **New/Born State:**
  When we create Thread class object in java applications then Thread will come to New/Born state.

- **Ready/Runnable State:**
  When we access start() method Thread Scheduler has to assign system resources like memory and time, here before assigning system resources and after calling start() method is called as Ready/Runnable state.

- **Running State:**
  In Java applications, after calling start() method and after getting system resources like memory and execution time is called as "Running State".

  NOTE: We can send a thread from Running state to Runnable state directly by accessing yield() method, but, it is not supported by Windows operating system, because, it will perform its functionality on the basis of Threads priority values, priority based operations are not supported by windows operating system.

- **Dead/Destroy State:**
  In Java applications, when we access stop() method over Running thread then that thread will come to Dead/Destroy state.

- **Blocked State:**
  In Java applications, we are able to keep a thread in Blocked state from Running state in the following situations.
  a) When we access sleep(--) method with a particular sleep time.
  b) When we access wait() method.
  c) When we access suspend() method.
  d) When we perform IO Operations.

In Java applications, we are able to bring a thread from Blocked state to Ready / Runnable state in the following situations.
a) When sleep time is over.
b) If any other thread access notify() / notifyAll() methods.
c) If any other thread access resume() method.
d) When IO Operations are completed.

Thread Class Library:

Constructors:

- **public Thread()**

  This constructor can be used to create thread class object with the following properties.

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

  EX:
  ```
  Thread t = new Thread();
  System.out.println(t);
  ```

  Output: Thread[Thread-0, 5, main]

- **public Thread(String name)**

  This constructor can be used to create Thread class object with the specified name.

  EX:
  ```
  Thread t = new Thread("Core Java");
  System.out.println(t);
  ```

  Output: Thread[Core Java,5,main]

- **public Thread(Runnable r)**

  This constructor can be used to create Thread class object with the specified Runnable reference.

  EX:
  ```
  Runnable r = new Thread();
  Thread t = new Thread(r);
  System.out.println(t);
  ```

  Output: Thread[Thread-1,5,main]

- **public Thread(Runnable r, String name)**

  This constructor can be used to create Thread class object with the specified Runnable reference and with the specified name.

  EX:
```
Runnable r = new Thread();
Thread t = new Thread(r, "Core Java");
System.out.println(t);
```

  Output: Thread[Core Java,5,main]

- **public Thread(ThreadGroup tg, Runnable r)**

  This constructor can be used to create Thread class object with the specified ThreadGroup name and with the specified thread name.

NOTE: To provide ThreadGroup name we have to use a predefined class like java.lang.ThreadGroup, to create ThreadGroup class object we have to use the following Constructor.

public ThreadGroup(String name)

EX:
```
ThreadGroup tg = new ThreadGroup("Java");
Runnable r = new Thread();
Thread t = new Thread(tg, r)
System.out.println(t);
```

Output: Thread[Thread-1,5,Java]

- **public Thread(ThreadGroup tg, String name)**

  This constructor can be used to create Thread class object with the specified ThreadGroup name and with the specified Thread name.

  EX:
```
ThreadGroup tg = new ThreadGroup("Java");
Thread t = new Thread(tg, "Core Java");
System.out.println(t);
```

  Output: Thread[Core Java,5,Java]

public Thread(ThreadGroup tg, Runnable r, String name)

This constuctor constuctor can be used to create Thread class object with the specified ThreadGroup name, with the Runnable reference and with the thread name.

EX:
```
ThreadGroup tg = new ThreadGroup("Java");
Runnable r = new Thread();
Thread t = new Thread(tg, r, "Core Java");
```

Output: Thread[Core Java, 5, Java]

Methods:

- ## public void setName(String name)
  It can be used to set a particular name to the Thread explicitly.

- ## public String getName()
  It can be used to get thread name explicitly.

EX:
```
1)  class Test
2)  {
3)     public static void main(String[] args)
4)     {
5)        Thread t=new Thread();
6)        System.out.println(t.getName());
7)        t.setName("Core Java");
8)        System.out.println(t.getName());
9)     }
10) }
```

public void setPriority(int priority)

It can be used to set a particular priority value to the Thread, but, here the priority value must be provided in the range from 1 to 10, if we provide any other value then JVM will rise an exception like java.lang.IllegalArgumentException.

To represent Thread 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;
```

- **public int getPriority()**

  It can be used to get priority value of the Thread.

  EX:

```
1)  class Test
2)  {
3)     public static void main(String[] args)
4)     {
5)        Thread t=new Thread();
6)        System.out.println(t.getPriority());
7)        t.setPriority(7);
8)        System.out.println(t.getPriority());
9)        t.setPriority(Thread.MAX_PRIORITY-2);
10)       System.out.println(t.getPriority());
11)       //t.setPriority(15);-->IllegalArgumentException
12)    }
13) }
```

public static int activeCount()

It will return the no of threads which are in active.

EX:

```
1)  class Test
2)  {
3)     public static void main(String[] args)
4)     {
5)        Thread t=new Thread();
6)        t.start();
7)        System.out.println(Thread.activeCount());
8)     }
9)  }
```

public boolean isAlive()

This method can be used to check whether a thread is in live or not.

EX:

```
1)  class Test {
2)     public static void main(String[] args) {
3)        Thread t=new Thread();
4)        System.out.println(t.isAlive());
5)        t.start();
6)        System.out.println(t.isAlive());
7)     }
8)  }
```

public static thread currentThread()

It can be used to get Thread object reference which is in active at present.

EX:

```
1)  class MyThread extends Thread
2)  {
3)     public void run()
4)     {
5)        for(int i=0;i<10;i++)
6)        {
7)           System.out.println(Thread.currentThread().getName());
8)        }
9)     }
10) }
11) class Test
12) {
13)    public static void main(String[] args)
14)    {
15)       MyThread mt1=new MyThread();
16)       MyThread mt2=new MyThread();
17)       MyThread mt3=new MyThread();
18)
19)       mt1.setName("AAA");
20)       mt2.setName("BBB");
21)       mt3.setName("CCC");
22)
23)       mt1.start();
24)       mt2.start();
25)       mt3.start();
26)    }
27) }
```

public static void sleep(long time) throws InterruptedException

- This method can be used to keep a running thread into sleeping state up to the specified sleep time.
- In general, we will use sleep() method in run() method of user defined thread class, where to handle InterruptedException we must use try-catch-finally syntax only, we must not use "throws" keyword in
- run() method prototype, because, we are overriding Thread class or Runnable interface predefined run() method.

- **public void join()throws InterruptedException**
  This method will pause a thread to complete a thread on which we accessed join() method , after completion of the respective thread, paused thread will continue its execution part automatically.

Daemon Threads

These threads are running internally to provide services to some other thread and it will be terminated along with the threads which are taking services.

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

public void setDaemon(boolean b)

If 'b' value is true then thread will be daemon thread.
If 'b' value is false then thread will not be daemon thread.

EX: mt.setDaemon(true);

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

public boolean isDaemon()

EX: In Java, Garbage Collector is a thread running internally inside JVM and it will provide Garbage Collection services to JVM and it will be terminated along with JVM automatically.

Synchronization:

In Java applications, if we execute more than one thread on a single data item then there may be a chance to get data inconsistency, it may generate wrong results in java applications.

In java applications, to provide data consistency in the above situation we have to use "Synchronization".

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

In java applications, synchronization is going on the basis of Locking mechanisms, If we send multiple threads at a time to synchronized area then Lock Manager will assign lock to a thread which is having highest priority, once a thread gets loc from Lock manager then that thread is eligible to enter in synchronized area, once a thread is available in

synchronized area then Lock Manager will not assign lock to other threads, when Thread completes its execution in synchronized area then that thread has to submit lock back to Lock Manager, once Lock is given back to Lock manager then Lock Manager will assign that lock to another thread which is having next priority.

In java applications, to provide synchronization JAVA has provided a keyword in the form of "synchronized".

In java applications, we are able to achieve "synchronization" in the following two ways.

- synchronized method
- synchronized block

synchronized method:

It is a normal java method, it will allow only one thread at a time to execute instructions, it will not allow more than one thread at a time, it will allow other threads after completion of the present thread execution.

EX:

```
1)  class A
2)  {
3)     synchronized void m1()
4)     {
5)        for(int i=0;i<10;i++)
6)        {
7)           String thread_Name=Thread.currentThread().getName();
8)           System.out.println(thread_Name);
9)        }
10)    }
11) }
12) class MyThread1 extends Thread
13) {
14)    A a;
15)    MyThread1(A a)
16)    {
17)       this.a=a;
18)    }
19)    public void run()
20)    {
21)       a.m1();
22)    }
23) }
24) class MyThread2 extends Thread
25) {
26)    A a;
```

```
27)    MyThread2(A a)
28)    {
29)        this.a=a;
30)    }
31)    public void run()
32)    {
33)        a.m1();
34)    }
35) }
36) class MyThread3 extends Thread
37) {
38)    A a;
39)    MyThread3(A a)
40)    {
41)        this.a=a;
42)    }
43)    public void run()
44)    {
45)        a.m1();
46)    }
47) }
48) class Test
49) {
50)    public static void main(String[] args)
51)    {
52)        A a=new A();
53)        MyThread1 mt1=new MyThread1(a);
54)        MyThread2 mt2=new MyThread2(a);
55)        MyThread3 mt3=new MyThread3(a);
56)
57)        mt1.setName("AAA");
58)        mt2.setName("BBB");
59)        mt3.setName("CCC");
60)
61)        mt1.start();
62)        mt2.start();
63)        mt3.start();
64)    }
65) }
```

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

In java applications, if we use synchronized method to achieve synchronization then it will provide synchronization throughout the method irrespective of the actual requirement. If we need synchronization up to a block inside the synchronized method then it will provide unnecessary synchronization for the remaining part of the method, it will increase execution time and it will reduce application performance.

In the above context, to provide synchronization up to the required part then we have to use synchronized block.

- **Synchronized Block:**

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

Syntax:

```
synchronized(Object o)
{
 -----
 -----
}
```

EX:

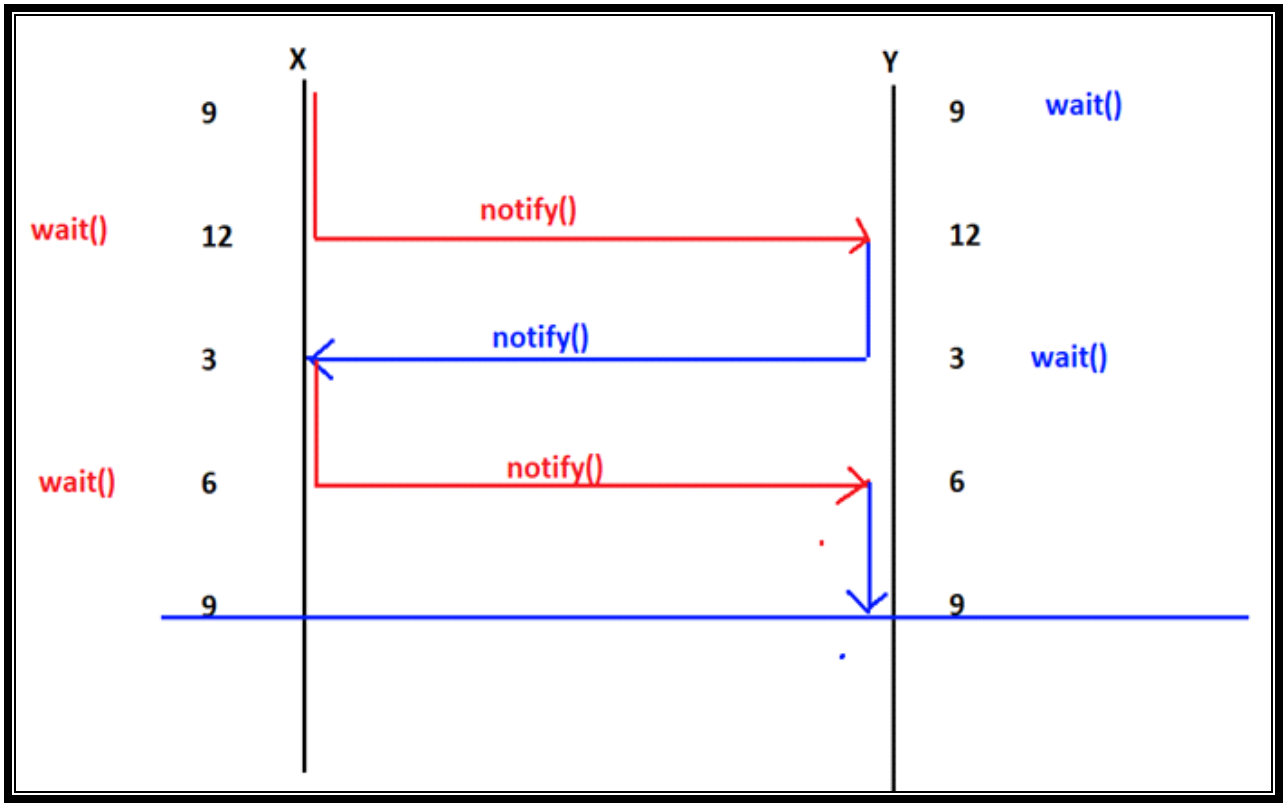
```
1)  class A
2)  {
3)     void m1()
4)     {
5)        String thread_Name=Thread.currentThread().getName();
6)        System.out.println("Before Synchronized Block :"+thread_Name);
7)        synchronized(this)
8)        {
9)           for(int i=0;i<10;i++)
10)          {
11)             String thread_Name1=Thread.currentThread().getName();
12)             System.out.println("Inside Synchronized Block :"+thread_Name1);
13)          }
14)       }
15)    }
16) }
17) class MyThread1 extends Thread
18) {
19)    A a;
20)    MyThread1(A a)
```

```
21)   {
22)      this.a=a;
23)   }
24)   public void run()
25)   {
26)      a.m1();
27)   }
28) }
29) class MyThread2 extends Thread
30) {
31)   A a;
32)   MyThread2(A a)
33)   {
34)      this.a=a;
35)   }
36)   public void run()
37)   {
38)      a.m1();
39)   }
40) }
41) class MyThread3 extends Thread
42) {
43)   A a;
44)   MyThread3(A a)
45)   {
46)      this.a=a;
47)   }
48)   public void run()
49)   {
50)      a.m1();
51)   }
52) }
53) class Test
54) {
55)   public static void main(String[] args)
56)   {
57)      A a=new A();
58)      MyThread1 mt1=new MyThread1(a);
59)      MyThread2 mt2=new MyThread2(a);
60)      MyThread3 mt3=new MyThread3(a);
61)
62)      mt1.setName("AAA");
63)      mt2.setName("BBB");
64)      mt3.setName("CCC");
65)
```

```
66)       mt1.start();
67)       mt2.start();
68)       mt3.start();
69)    }
70) }
```

Inter Thread Communication:

The process of providing communication between more than one thread is called as " Inter Thread Communication".



To perform Inter Thread Communication we have to use the following methods.

- **wait()**
- **notify()**
- **notifyAll()**

Where wait() method can be used to keep a thread in waiting state.

Where notify() method can be used to give a notification to a thread which is available in waiting state.

Where notifyAll() method can be used to give a notification to all the threads which are available i waiting state.

The above methods are provided by JAVA in java.lang.Object class.
If we want to use these methods in java applications then we must provide "Synchronization".

In general, Inter Thread Communication will provide solutions for the problems like "Producer-Consumer" problems.

In Propducer-Consumer problem, producer and consumer are two threads, where producer has to produce an item and consumer has to consume that item, the same sequence has to be provided infinite no of times, where Producer must not produce an item without consuming previous item by consumer and consumer must not consume an item without producing that item by producer.

EX:

```
1)  class A
2)  {
3)     boolean flag=true;
4)     int count=0;
5)     public synchronized void produce()
6)     {
7)        try
8)        {
9)           while(true)
10)          {
11)             if(flag == true)
12)             {
13)                count=count+1;
14)          System.out.println("Producer Produced Item"+count);
15)                flag=false;
16)                notify();
17)                wait();
18)             }
19)             else
20)             {
21)                wait();
22)             }
23)          }
24)       }
25)       catch (Exception e)
26)       {
27)          e.printStackTrace();
28)       }
29)    }
30)    public synchronized void consume()
31)    {
```

```
32)     try
33)     {
34)        while(true)
35)        {
36)           if(flag == true)
37)           {
38)              wait();
39)           }
40)           else
41)           {
42)     System.out.println("Consumer Consumed Item"+count);
43)              flag=true;
44)              notify();
45)              wait();
46)           }
47)        }
48)     }
49)     catch (Exception e)
50)     {
51)        e.printStackTrace();
52)     }
53)  }
54) }
55) class Producer extends Thread
56) {
57)    A a;
58)    Producer(A a)
59)    {
60)       this.a=a;
61)    }
62)    public void run()
63)    {
64)       a.produce();
65)    }
66) }
67) class Consumer extends Thread
68) {
69)    A a;
70)    Consumer(A a)
71)    {
72)       this.a=a;
73)    }
74)    public void run()
75)    {
76)       a.consume();
```

```
77)    }
78) }
79) class Test
80) {
81)    public static void main(String[] args)
82)    {
83)       A a=new A();
84)       Producer p=new Producer(a);
85)       Consumer c=new Consumer(a);
86)       p.start();
87)       c.start();
88)    }
89) }
```

Dead Lock:

Dead Lock is a situation, where more than one thread is depending on each other in circular dependency.

In java applications, once we are getting deadlock then program will struct in the middle, so that, it will not have any recovery mechanisms, it will have only prevention mechanisms.

EX:

```
1)  class Register_Course extends Thread
2)  {
3)     Object course_Name;
4)     Object faculty_Name;
5)     Register_Course(Object course_Name, Object faculty_Name)
6)     {
7)        this.course_Name=course_Name;
8)        this.faculty_Name=faculty_Name;
9)     }
10)    public void run()
11)    {
12)       synchronized(course_Name)
13)       {
14)          System.out.println("Register_Course Thread holds course_Name resource and
     waiting for faculty_Name resource.....");
15)          synchronized(faculty_Name)
16)          {
17)             System.out.println("Register_Course is success, because, Register_Course thr
     ead holds both course_Name and faculty_Name resources");
18)          }
19)       }
```

```
20)    }
21) }
22) class Cancel_Course extends Thread
23) {
24)    Object course_Name;
25)    Object faculty_Name;
26)    Cancel_Course(Object course_Name, Object faculty_Name)
27)    {
28)       this.course_Name=course_Name;
29)       this.faculty_Name=faculty_Name;
30)    }
31)    public void run()
32)    {
33)       synchronized(faculty_Name)
34)       {
35)          System.out.println("Cancel_Course Thread holds faculty_Name resource and
    waiting for course_Name resource.....");
36)          synchronized(course_Name)
37)          {
38)             System.out.println("Cancel_Course is success, because, Cancel_Course thread
    holds both faculty_Name and course_Name resources");
39)          }
40)       }
41)    }
42) }
43)
44) class Test
45) {
46)    public static void main(String[] args)
47)    {
48)       Object course_Name=new Object();
49)       Object faculty_Name=new Object();
50)       Register_Course rc=new Register_Course(course_Name, faculty_Name);
51)       Cancel_Course cc=new Cancel_Course(course_Name, faculty_Name);
52)       rc.start();
53)       cc.start();
54)    }
55) }
```