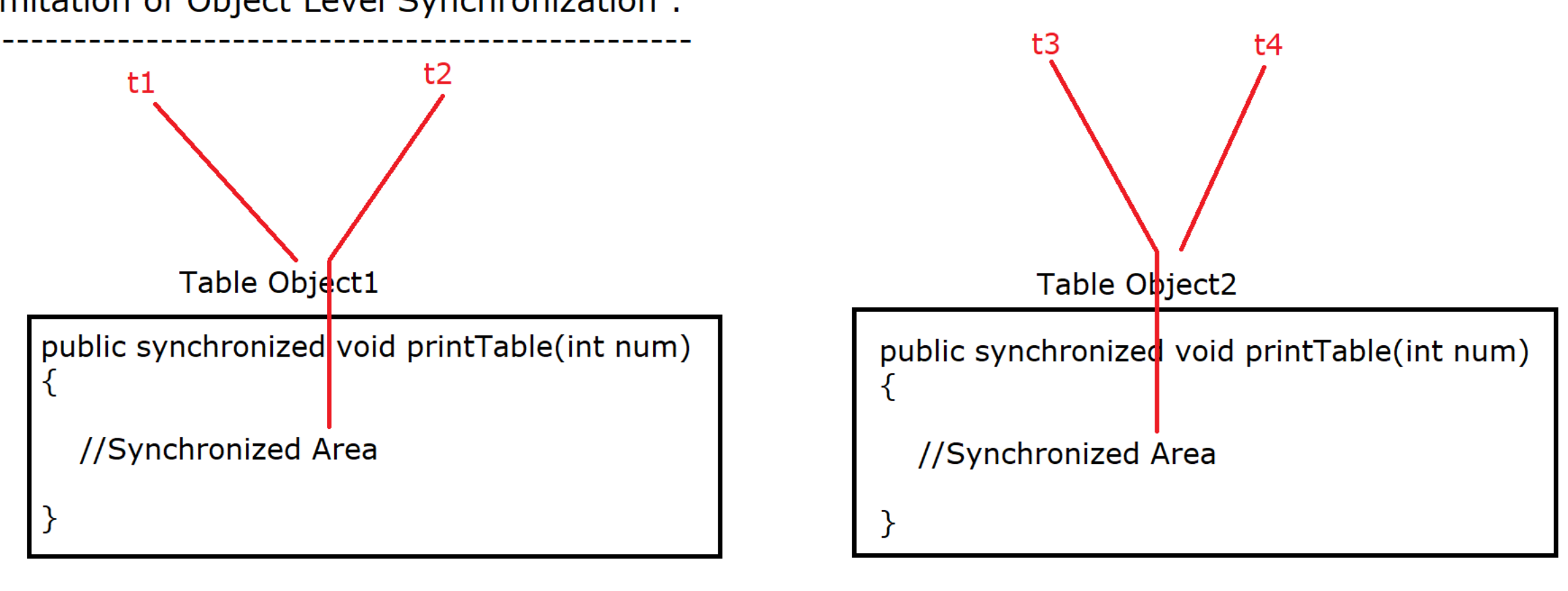
Limitation of Object Level Synchronization :



From the above diagram it is clear that there is no interference between t1 and t2 thread because they are passing throgh Object1 where as on the other hand there is no interferenec even in between t3 and t4 threads because they are also passing through Object2 (another object).

But there may be chance that with t1 Thread (object1), t3 or t4 thread can enter inside the synchronized area at the same time, simillarly it is also possible that with t2 thread, t3 or t4 thread can enter inside the synchronized area so the conclusion is, synchronization mechanism does not work with multiple Objects.

```
package com.ravi.limitation_of_object_level_syn;

public class Table
{
    public synchronized void printTable(int num)
    {
        for(int i=1; i<=10; i++)
        {
            System.out.println(num+" X "+i+" = "+(num*i));
            try
            {
                Thread.sleep(1000);
            }
            catch(InterruptedException e)
            {
                System.err.println("Thread has Interrupted!!!");
            }
        }
        System.out.println("...................");
    }
}
```

```
package com.ravi.limitation_of_object_level_syn;

public class LmitationOfObjectLevel
{
    public static void main(String[] args)
    {
        Table obj1 = new Table(); //lock1
        Table obj2 = new Table(); //lock2

        Thread t1 = new Thread()
        {
           @Override
           public void run()
           {
               obj1.printTable(5);   //lock1
           }
        };


        Thread t2 = new Thread()
        {
           @Override
           public void run()
           {
               obj1.printTable(7); //lock1
           }
        };


        Thread t3 = new Thread()
        {
           @Override
           public void run()
           {
               obj2.printTable(12); //lock2
           }
        };

        Thread t4 = new Thread()
        {
           @Override
           public void run()
           {
               obj2.printTable(15);   //lock2
           }
        };

        t1.start(); t2.start(); t3.start(); t4.start();



    }

}
```

Note : From the above program it is clear that synchronization logic will not work with multiple objects.

In order to avoid this drawback, We introduced Static Synchronization.

Static Synchronization :
----
* If we declare our **synchronized method** with **static modifier** then It is called static synchronization.

* In static synchronization, Object is not required so the thread can access static synchronized method directly with the help of class name.

* The thread will take the **lock from the class rather than Object.**

* Unlike Object, We cannot create multiple classes in the same package.

```
package com.ravi.static_syn;

public class Table
{
    public static synchronized void printTable(int num)
    {
        for(int i=1; i<=10; i++)
        {
            System.out.println(num+" X "+i+" = "+(num*i));
            try
            {
                Thread.sleep(1000);
            }
            catch(InterruptedException e)
            {
                System.err.println("Thread has Interrupted!!!");
            }
        }
        System.out.println("...................");
    }
}
```

```
package com.ravi.static_syn;

public class StaticSynchronization
{
    public static void main(String[] args)
    {

        Thread t1 = new Thread()
        {
           @Override
           public void run()
           {
               Table.printTable(5);
           }
        };


        Thread t2 = new Thread()
        {
           @Override
           public void run()
           {
              Table.printTable(7);
           }
        };


        Thread t3 = new Thread()
        {
           @Override
           public void run()
           {
               Table.printTable(12); //lock2
           }
        };

        Thread t4 = new Thread()
        {
           @Override
           public void run()
           {
               Table.printTable(15);
           }
        };

        t1.start(); t2.start(); t3.start(); t4.start();

     }

  }
```

How to work with synchronized block, If we want to take the lock from the class.

synchronized(MyClass.class) //Taking the lock from the class
{

}

Thread Priority :
----
* In java, It is possible to assign priority of the thread in numbers from 1 - 10 only, where 1 defines the minimum priority and 10 defines the maximum priority.

* Thread class has provided **final setter and getter** method to set and get the priority of the thread.

Example :
public class Thread implements Runnable
{
    int priority;

    public final void setPriority(int newPrioity)
    {

    }

    public final int getPriority()
    {

    }
}

* Thread class has also provided 3 final static variables to represent the priority of the Thread.

public static final int MIN_PRIORITY = 1

public static final int NORM_PRIORITY = 5

public static final int MAX_PRIORITY = 10

* In java, Whenever we create a thread then It contains default priority i.e 5 that means normal priority.

* The threads which are created as a part of main thread will acquire the same priority of main thread.

* We can't set the priority of the Thread beyond the limit(1-10) so if we set the priority beyond the limit (1 to 10) then it will generate an exception java.lang.IllegalArgumentException.

//Assigning Priority to the Thread :

```
package com.ravi.priority;

public class PriorityDemo1
{
    public static void main(String[] args)
    {
        Thread t1 = new Thread();
        System.out.println(t1.getPriority());

        t1.setPriority(1);
        System.out.println(t1.getPriority());


        t1.setPriority(Thread.MAX_PRIORITY);
        System.out.println(t1.getPriority());

        t1.setPriority(11); //java.lang.IllegalArgumentException
        System.out.println(t1.getPriority());

    }

}
```

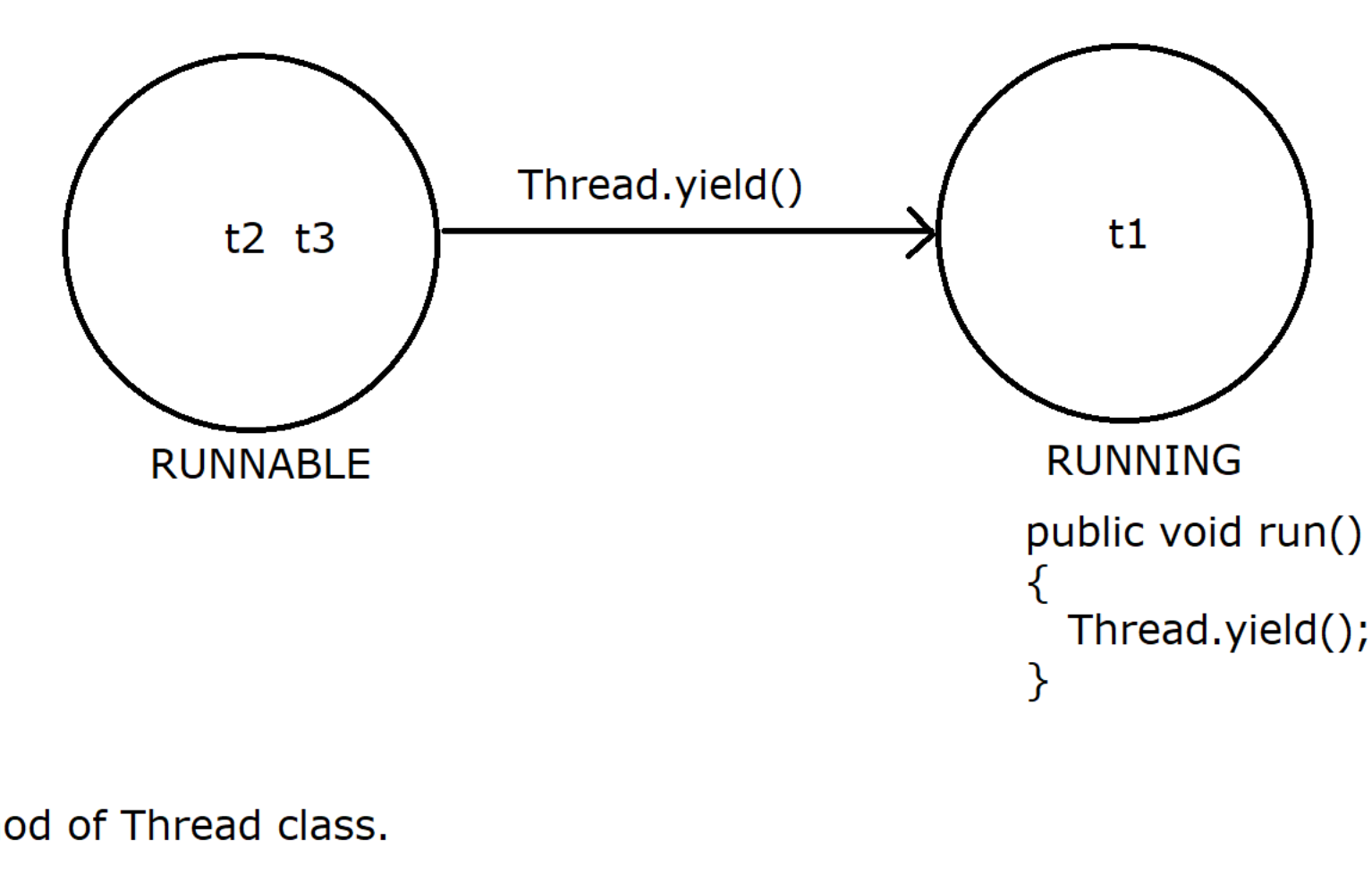
```
package com.ravi.priority;

class UserThread extends Thread
{
    @Override
    public void run()
    {
        int priority = Thread.currentThread().getPriority();
        System.out.println("Child Thread Priority is :"+priority);
    }
}

public class PriorityDemo2
{
    public static void main(String[] args)
    {
        Thread t = Thread.currentThread();
        t.setPriority(Thread.MAX_PRIORITY);
        System.out.println("Main Thread Priority is :"+t.getPriority());

         UserThread ut = new UserThread();
         ut.start();

    }

}
```

Note : UserThread created under main thread will acquire main Thread priority.

```
package com.ravi.priority;

class CustomThread implements Runnable
{
    int count = 1;

    @Override
    public void run()
    {
       for(int i=1; i<=1000000; i++)
       {
           count++;
       }

       String name = Thread.currentThread().getName();
       int priority = Thread.currentThread().getPriority();

       System.out.println("Completed thread name is :"+name+" and its priority is
:"+priority);
    }
}


public class PriorityDemo3
{
    public static void main(String[] args)
    {
        CustomThread customThread = new CustomThread();

        Thread t1 = new Thread(customThread, "Last");
        Thread t2 = new Thread(customThread, "First");

        t1.setPriority(1);
        t2.setPriority(10);

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

    }

}
```

Most of time the thread having highest priority will complete its task but we can't say that it will always complete its task first that means Thread schedular dominates Priority of the Thread.

Thread.yield() : [To Prevent a thread from over utilization of CPU]



It is a static method of Thread class.

The currently executing thread will send a notification to thread schedular to stop the currently executing Thread (In Running state) and provide a chance to Threads which are in Runnable state to enter inside the running state having same priority or higher priority than currently executing Thread.

Here The running Thread will directly move from Running state to Runnable state.

The Thread schedular may accept OR ignore this notification message given by currently executing Thread.

Here there is no guarantee that after using yield() method the running Thread will move to Runnable state and the thread which is running thread will move to Runnable state.

If the thread which is in runnable state is having low priority than the current executing thread in Running state,Running thread will continue its execution.

*It is mainly used to avoid the over-utilization a CPU by the current Thread so all the threads will get equal chance of CPU
for execution as well as In industry we can use testing and debugging purposes.

```
package com.ravi.yield;

class Test implements Runnable
{
    @Override
    public void run()
    {
        String name = Thread.currentThread().getName();

        for(int i=1; i<=10; i++)
        {
            System.out.println(i+" by "+name+" thread");

            //If Child1 is running, Give a chance to Child2
            if(name.equalsIgnoreCase("Child1"))
            {
                Thread.yield(); //Give a chance to Child2
            }
        }
    }
}
public class YieldDemo
{
    public static void main(String[] args)
    {
        Test t1 = new Test();

        Thread thread1 = new Thread(t1, "Child1");
        Thread thread2 = new Thread(t1, "Child2");

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