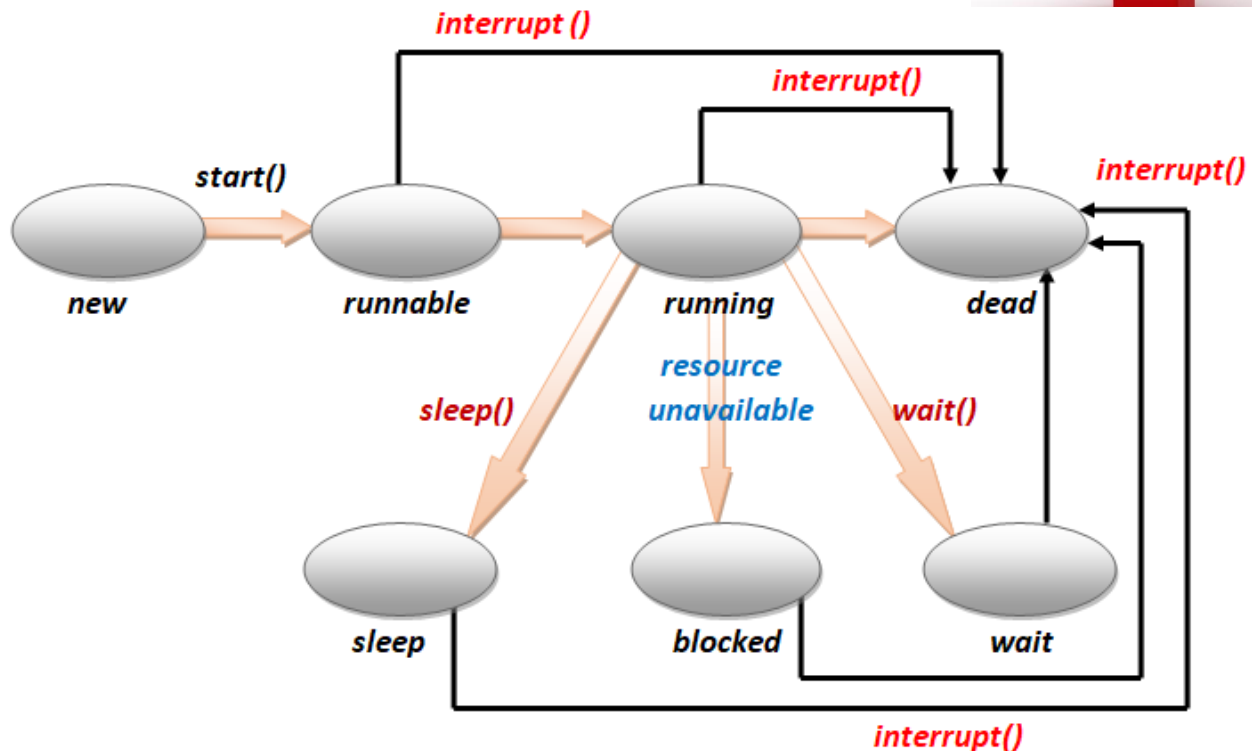


Multithreading Continued

Life Cycle of a thread (Thread State diagram)

The moment we create a thread till the moment the thread finishes its execution, a thread goes through different phases or states.



- **New:** As soon as, you create a thread, it's in *New state*. It remains in this state until the program starts the thread using its **start()** method.
- **Runnable:** A thread that is ready to run is moved to the runnable state after invocation of `start()` method, but the thread scheduler has not selected it to be the running thread.
- **Running:** The thread is in running state if the thread scheduler has selected it and given time to run.
- **Blocked:** A *Runnable thread* transitions to the *blocked state* when it attempts to perform a task that cannot be completed immediately and it must temporarily wait until that task completes.

Which States?

- **Waiting:** Usually program put a thread in *wait state* by calling **wait()** method because something else needs to be done prior to what current thread is doing.
- **Dead:** A thread enters this state when it successfully completes its task or otherwise terminated due to any error or even if it is forcefully killed.

Threads from runnable, running, sleep, blocked and wait state can be moved to dead state by calling **interrupt()** method.

Now let us understand single thread program using thread state diagram.

```
class Demo
{
    public static void main(String[] args) throws InterruptedException
    {
        System.out.println("main thread started execution");

        System.out.println("main thread is executing");
        Thread.sleep(5000);
        System.out.println("main thread is executing");
        Thread.sleep(5000);
        System.out.println("main thread is executing");
        Thread.currentThread().interrupt();
        Thread.sleep(5000);

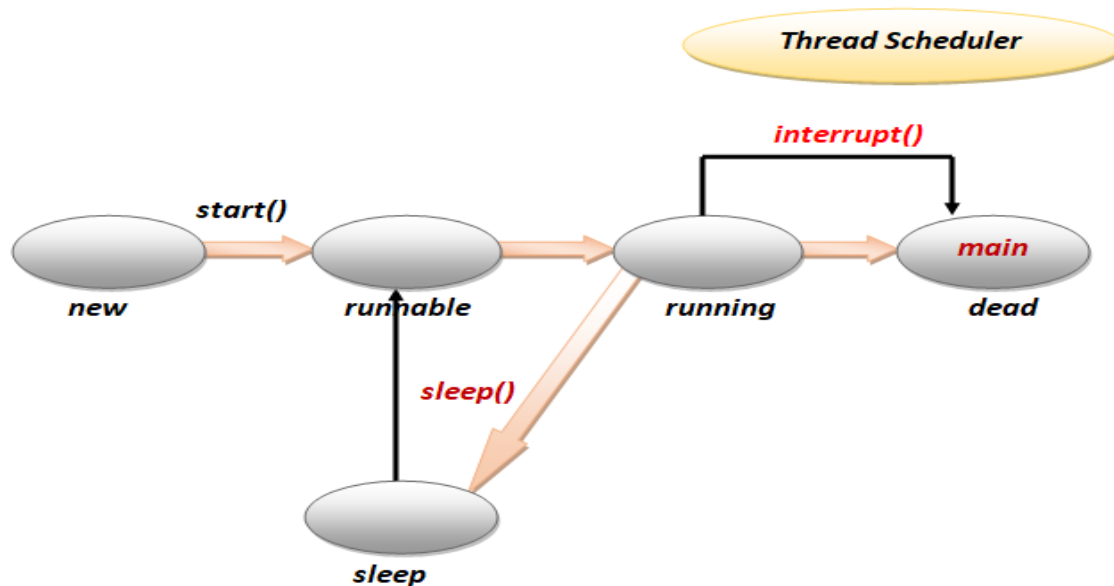
        System.out.println("main thread completed execution");
    }
}
```

The first thread that is created in every Java program is the main thread. As soon as, the thread is created it exists in the **new state**. Before the thread start executing it should be brought to the **runnable state** by calling **start()** method making the thread ready to run.

The duty of the thread scheduler is to ensure whether the CPU time is efficiently utilized or not. If the running state is empty and no thread is executing it will take the main thread from the runnable state to the **running state**.

Whenever the thread is executing is called by the **sleep()** method, then the running thread stop execution and moves to the **sleep state**.

When the sleep time is finished it again moves back to the runnable state and the flow repeats till it completes its execution. If an ***interrupt()*** method is called then the main thread which is alive is killed and moved to the ***dead state***. And when this happens Java creates an exception.



Since this is a single threaded program when the running state is empty there would be no other thread to execute and **therefore the CPU time is not been used efficiently.**

Output:

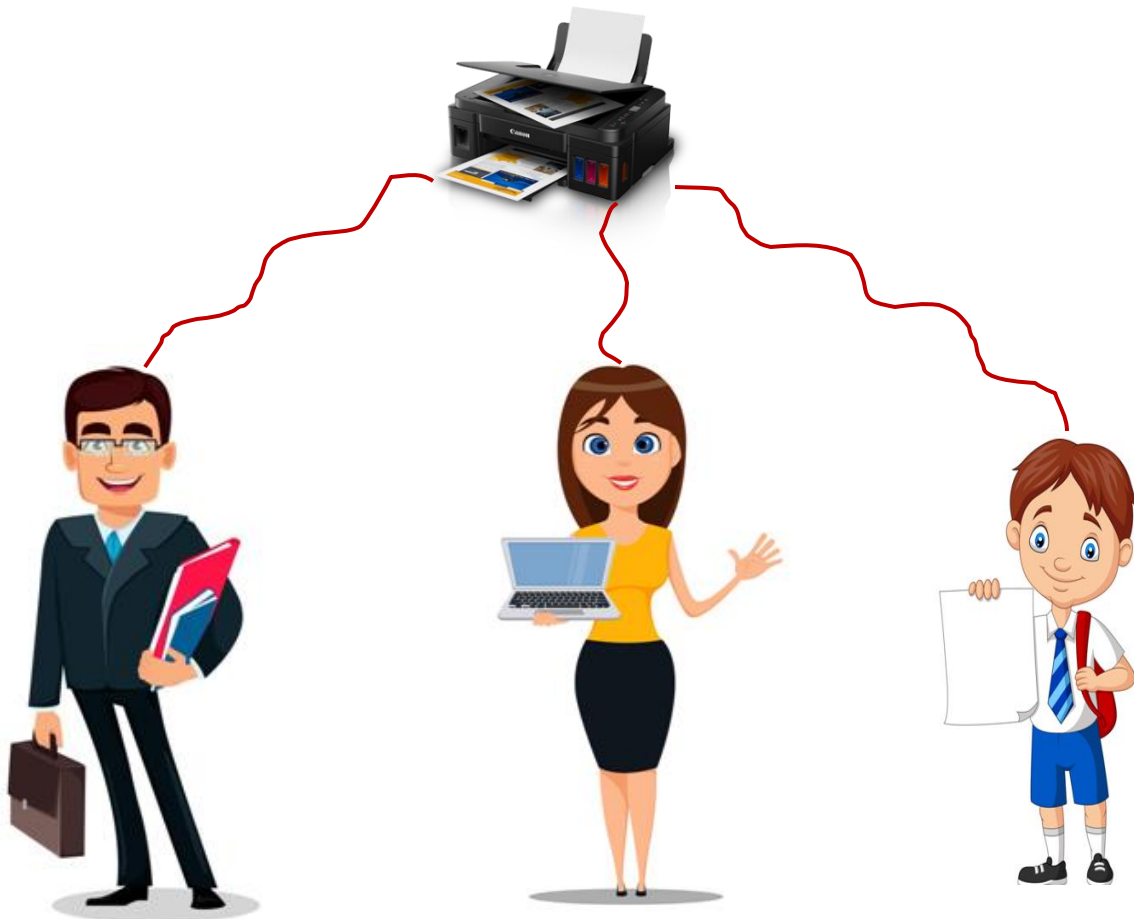
```
main thread started execution
main thread is executing
main thread is executing
main thread is executing
Exception in thread "main" java.lang.InterruptedException: sleep interrupted
    at java.lang.Thread.sleep(Native Method)
    at Demo.main(Demo.java:13)
```

Disadvantage of Multithreading.

Applying the approach of multithreading at all cases is not a good practice. In some cases we have to use Single threaded approach.

Let's understand this with a scenario.

Now imagine we have a single printer and there are three people as Men, Women and a Child and they want to a print of a document. Now should this one resource be used by all these people at th same time?



If this was the case, then when print is given, the printer takes the input from all these people and concurrent execution happens.

The printer at any given time should take the input from one person for a proper document to be printed.

Let us code this Scenario

```
class Printer implements Runnable
{
    public void run()
    {
        String name = Thread.currentThread().getName();

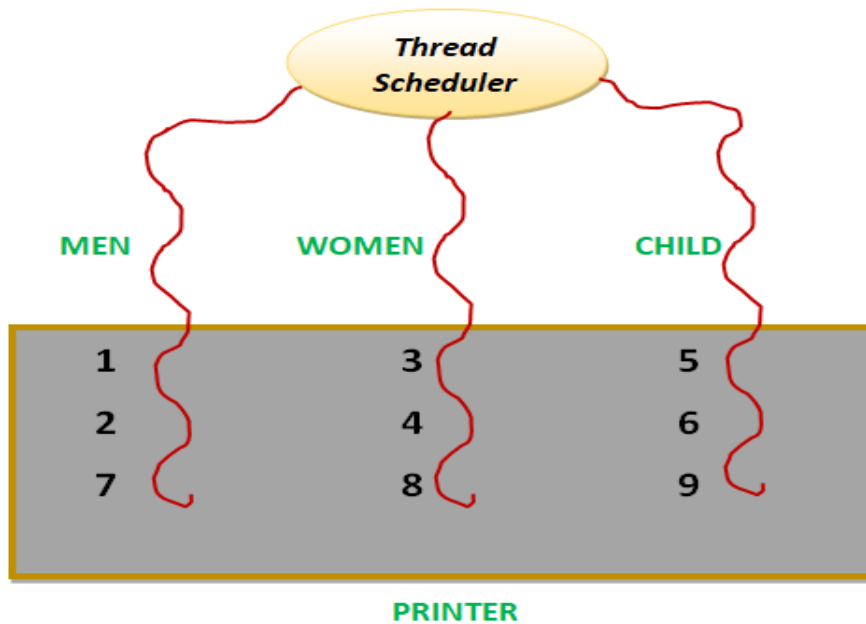
        System.out.println(name+ " started printing");
        for(int i=1;i<=3;i++)
        {
            System.out.println(name+ " is printing");
            try
            {
                Thread.sleep(4000);
            }
            catch (Exception e)
            {
                System.out.println("Some problem occurred");
            }
        }
        System.out.println(name+ " completed printing");
    }
}

class Demo
{
    public static void main(String[] args) throws InterruptedException
    {
        Printer p = new Printer();

        Thread t1 = new Thread(p);
        Thread t2 = new Thread(p);
        Thread t3 = new Thread(p);
        t1.setName("MEN");
        t2.setName("WOMEN");
        t3.setName("CHILD");

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

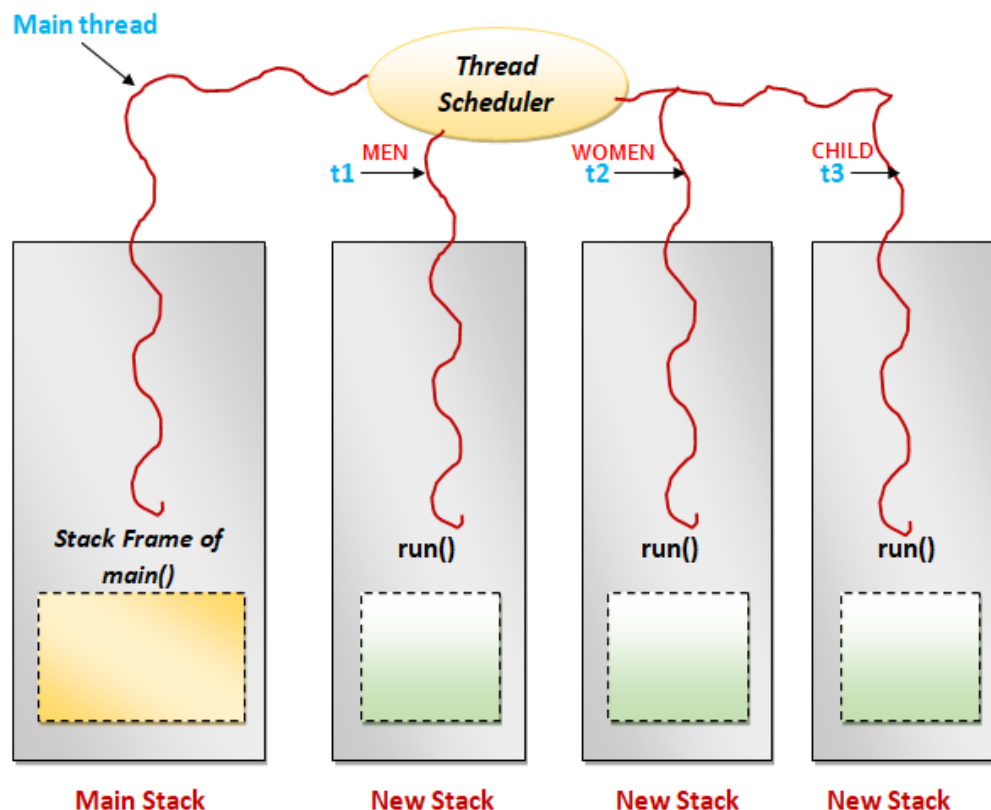




In this program, in the Stack segment the stack frame of main method gets created and the Thread scheduler will create the main thread.

When we create an object of a class which is implemented a runnable interface then a new Stack will not get created. If we want to create a new thread, then we should create an object of Thread class and we have to mention which run method should it refer to. Similarly we create three new stacks for Men, Women and Child.

Then we start the execution by calling the start() method with their corresponding references.



Output:

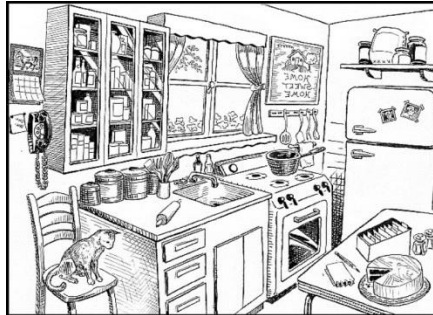
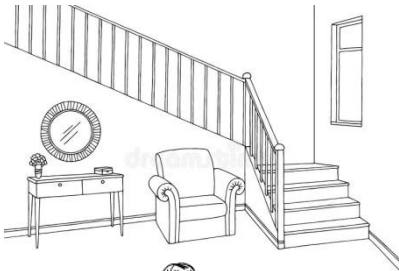
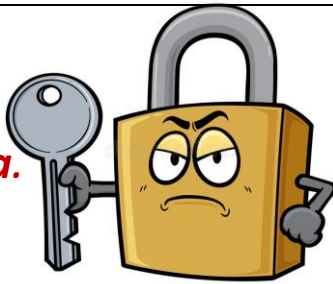
```
WOMEN started printing  
CHILD started printing  
MEN started printing  
CHILD is printing  
WOMEN is printing  
MEN is printing  
WOMEN is printing  
CHILD is printing  
MEN is printing  
CHILD is printing  
MEN is printing  
WOMEN is printing  
CHILD completed printing  
MEN completed printing  
WOMEN completed printing
```

Here in the above program the problem was all the three threads were accessing the same resource at the same time and therefore the final result we got was in the above manner.

Let us understand how to resolve this problem? What is the solution?



Concept of Lock in Java.



Now in these two cases, a single resource (such as the hall or kitchen) can be used by multiple people (threads). This is allowed/permitted/acceptable.

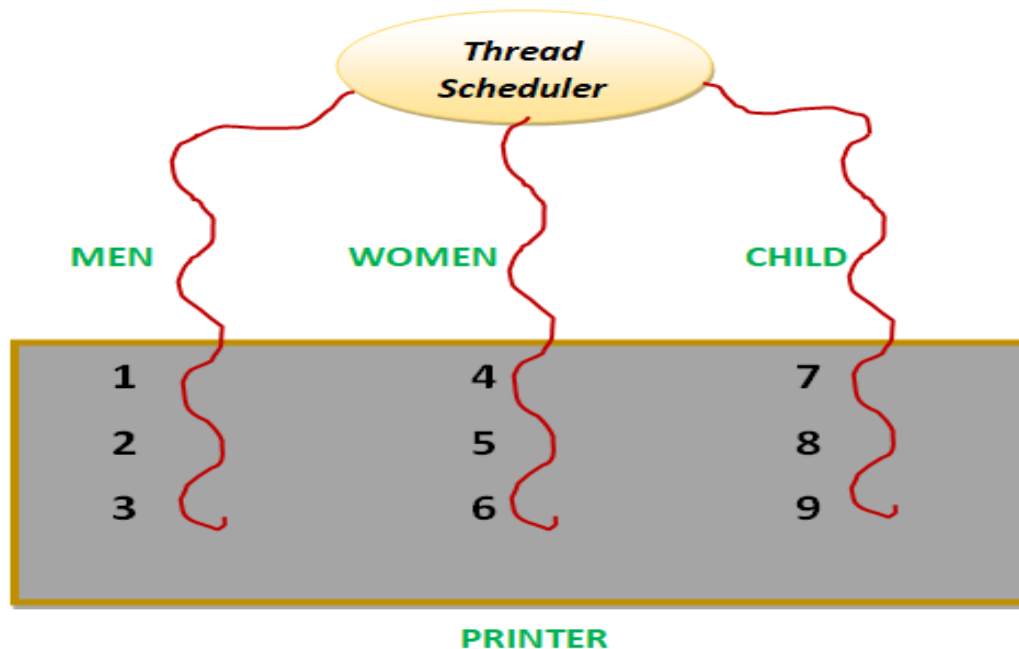
But the same approach is not preferred here. Therefore, we have to restrict people (threads) by applying lock to the resource.

Similarly in Java program, a common resource is accessed by multiple threads at the same time. This will result in unexpected output and hence must be prevented. Such statements which only a single thread must access at any given point of time are referred to as **Monitor or Semaphore**.

A semaphore can be achieved in Java by using the facility of locks. A lock can be implemented by using the **synchronized keyword**.

Multithreading with Synchronization.

In this case, the thread scheduler should not give access to any other thread until the current thread completes its execution. This can be done by Sequential execution with applying locks to the threads.



Case-1:

```
class Printer implements Runnable
{
    synchronized public void run()
    {
        String name = Thread.currentThread().getName();

        System.out.println(name+ " started printing");
        for(int i=1;i<=3;i++)
        {
            System.out.println(name+ " is printing");
            try
            {
                Thread.sleep(4000);
            }
            catch (Exception e)
            {
                System.out.println("Some problem occurred");
            }
        }
        System.out.println(name+ " completed printing");
    }
}
```

```

class Demo
{
    public static void main(String[] args) throws InterruptedException
    {
        Printer p = new Printer();

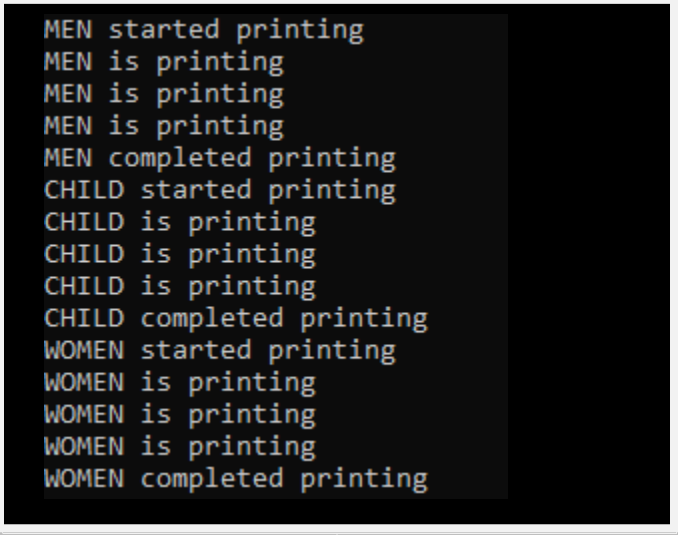
        Thread t1 = new Thread(p);
        Thread t2 = new Thread(p);
        Thread t3 = new Thread(p);
        t1.setName("MEN");
        t2.setName("WOMEN");
        t3.setName("CHILD");

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

```

So now by applying lock to **run()** method, no other threads will be able to access it.

Output:



```

MEN started printing
MEN is printing
MEN is printing
MEN is printing
MEN completed printing
CHILD started printing
CHILD is printing
CHILD is printing
CHILD is printing
CHILD completed printing
WOMEN started printing
WOMEN is printing
WOMEN is printing
WOMEN is printing
WOMEN completed printing

```

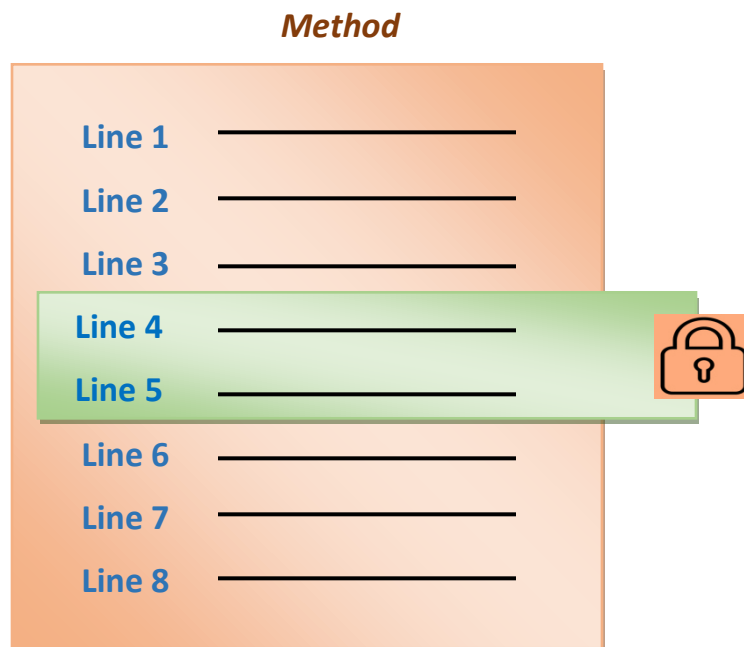
Why Synchronization?

- To prevent thread interference.
- To prevent consistency problem.

Case-2:

If all the lines in method should be locked then use “synchronized” to the entire method.

Whereas, if only certain lines of the method should be locked then use “synchronized” only to those lines by putting it in a block. And name the block as synchronized.



Code:

```
class Demo
{
    public static void main(String[] args) throws InterruptedException
    {
        Test t = new Test();

        Thread t1 = new Thread(t);
        Thread t2 = new Thread(t);

        t1.setName("ONE");
        t2.setName("TWO");

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

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

        try
        {
            System.out.println(name+ " is executing first line");
            Thread.sleep(3000);
            System.out.println(name+ " is executing second line");
            Thread.sleep(3000);
            System.out.println(name+ " is executing third line");
            Thread.sleep(3000);

            synchronized(this)
            {
                System.out.println(name+ " is executing fourth line");
                Thread.sleep(3000);
                System.out.println(name+ " is executing fifth line");
                Thread.sleep(3000);
            }

            System.out.println(name+ " is executing sixth line");
            Thread.sleep(3000);
            System.out.println(name+ " is executing seventh line");
            Thread.sleep(3000);
            System.out.println(name+ " is executing eighth line");
            Thread.sleep(3000);
        }

        catch (Exception e)
        {
            System.out.println("Some problem occurred");
        }
    }
}
```

Output:

```
ONE is executing first line  
TWO is executing first line  
TWO is executing second line  
ONE is executing second line  
TWO is executing third line  
ONE is executing third line  
TWO is executing fourth line  
TWO is executing fifth line  
TWO is executing sixth line  
ONE is executing fourth line  
ONE is executing fifth line  
TWO is executing seventh line  
ONE is executing sixth line  
TWO is executing eighth line  
ONE is executing seventh line  
ONE is executing eighth line
```

