Nested try-catch-finally

```
try
{
  try
  {
  }
  catch(xxx e)
  {
  }
  finally
  {
  }
}
catch(xxx e)
{
}
finally
{
}
```

✓

Nested try-catch-finally

```
try
{
}
catch(xxx e)
{
  try
  {
  }
  catch(xxx e)
  {
  }
  finally
  {
  }
}
finally
{
}
```

✓

Nested try-catch-finally

```
try
{
}
catch(xxx e)
{
}
finally
{
  try
  {
  }
  catch(xxx e)
  {
  }
  finally
  {
  }
}
```

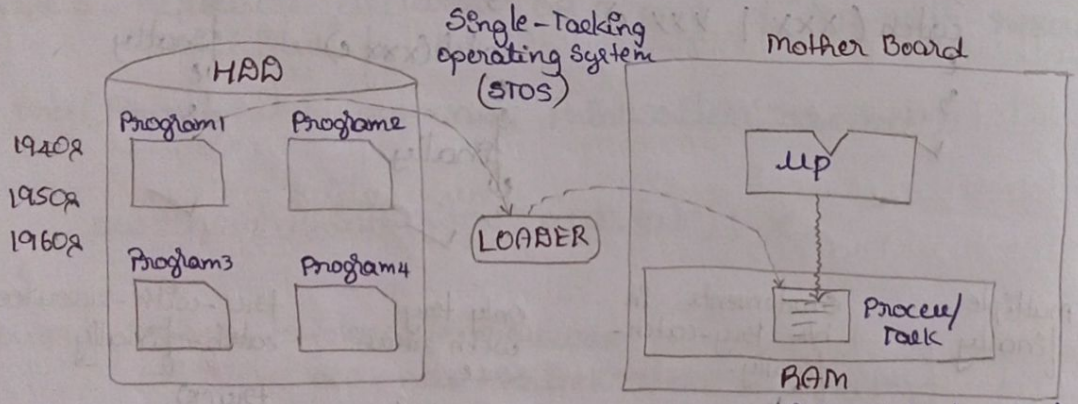
✓

Nested try-catch-finally

```
try
{
  try
  {
  }
  catch(xxx e)
  {
  }
  finally
  {
  }
}
catch(xxx e)
{
  try
  {
  }
  catch(xxx e)
  {
  }
  finally
  {
  }
}
finally
{
  try
  {
  }
  catch(xxx e)
  {
  }
  finally
  {
  }
}
```

✓

Multi-threading in Java:



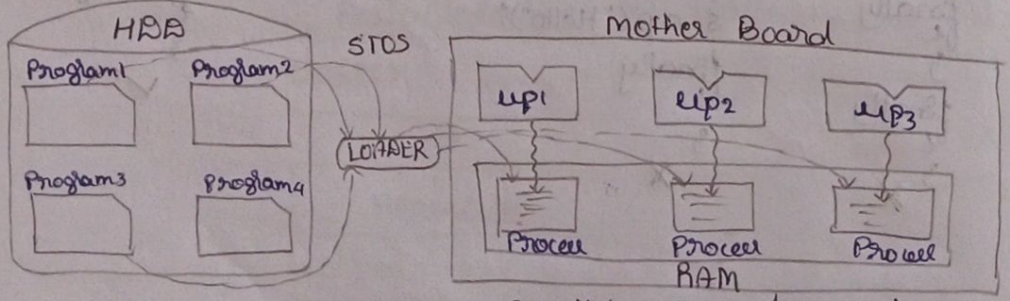
sequencial Style of Execution

[One program has to wait for another program to complete its execution.]

Adv: Inexpensive (Single Proceed)

Disadv: Slow in Execution

Single-Tasking System using STOS & Single Proceed



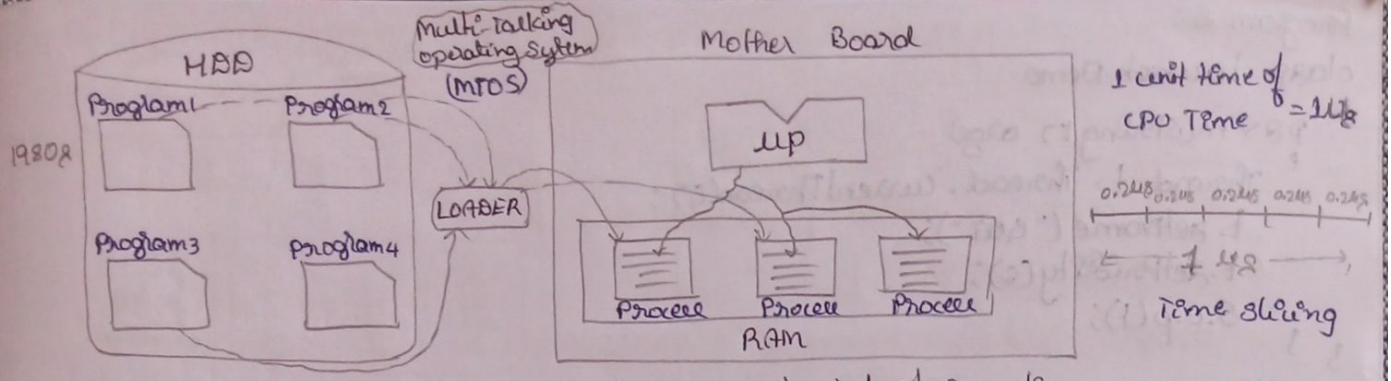
Parallel Style of Execution

[Each processor will be executing a seperate proceed]

Adv: Fast in Execution

Disad: Expensive

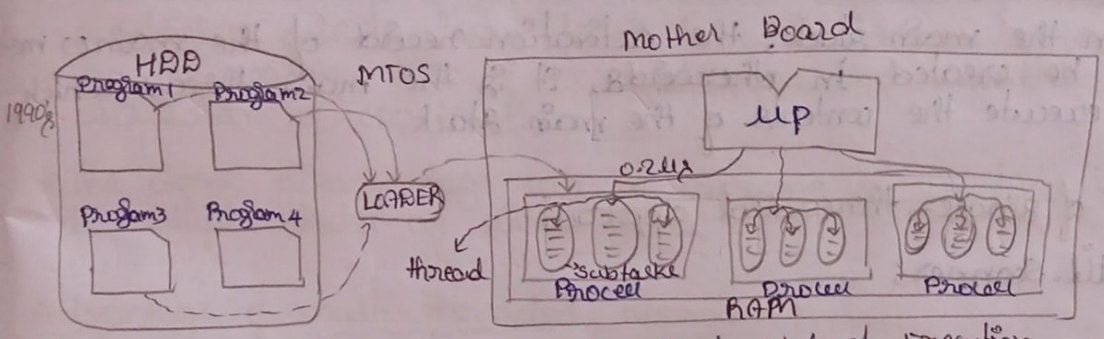
Multi-Tasking system using Single-Tasking OS & Multi-Proceed



Concurrent style of Execution
(Single processor executing multiple processes concurrently)
Multi-Tasking system using Multi-Tasking OS & Single Processor

Adv: * Inexpensive
+ Fast in execution

Dis: CPU time cannot be efficiently utilized



concurrent style of Execution
(Single processor executing multiple processes concurrently)
Multi-Tasking system with multi-threading using multi-tasking operating system & single processor

Adv: * Efficient utilization of CPU Time
* Inexpensive
+ Fast in Execution

12/05/23

Multi-threading is a feature in Java that allow the concurrent execution of 2 @ more parts (subtasks) of a program for maximum utilization of CPU time.

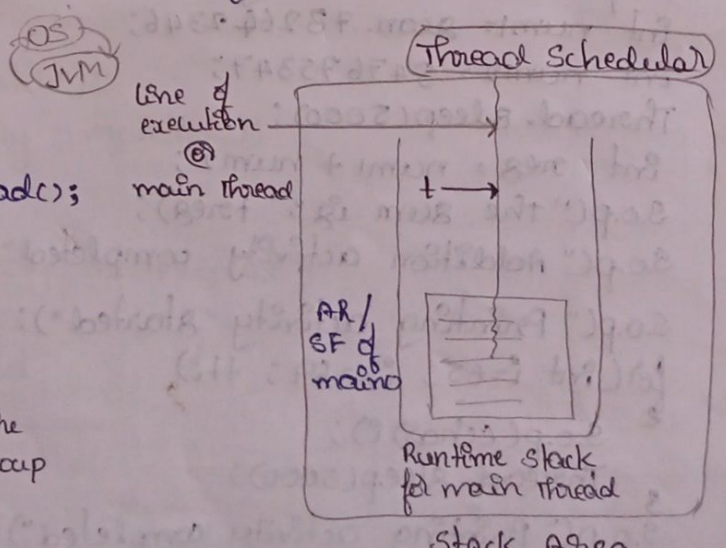
Each part of such program will be executed by a separate thread. In other words, a thread is a line of execution within a program

Program 1:

```
class Launch Demo
{
  psvm(String[] args)
  {
    Thread t = Thread.currentThread();
    S.o.p(t);
  }
}
```

O/P: Thread[main, 5, main]

name of the thread — priority of the thread — name of the ThreadGroup



Program 2:

```
class Launch Demo
{
  psvm(String[] args)
  {
    Thread t = Thread.currentThread();
    t.setName("GAT");
    t.setPriority(8);
    S.o.p(t);
  } }
```

o/p: Thread[GAT, 8, main]

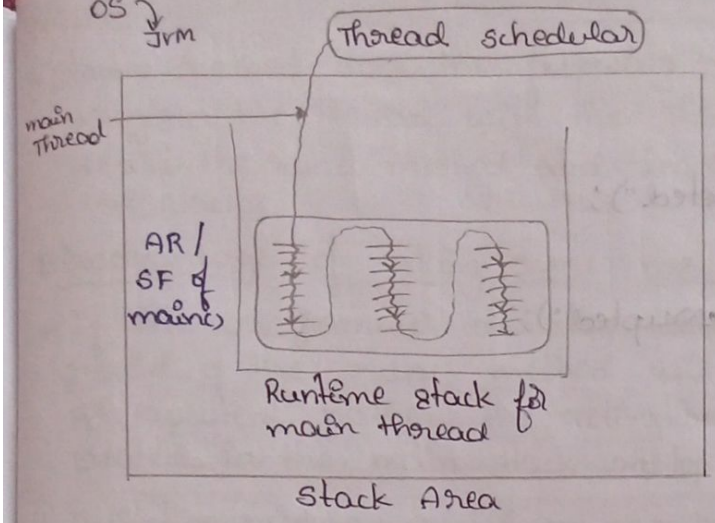
NOTE: * For a Java program to be executed, the JVM would automatically create a line of execution (thread) called as main thread.

* The JVM would then also create a runtime stack for this thread called as the main stack. It would then register that thread with the Thread scheduler.

* It is on the main stack, the activation record of the main() method will be created. In otherwords, it is the main thread which would execute the contents of the main stack.

Disadvantage of single-threaded approach:

```
import java.util.Scanner;
class Demo
{
  psvm(String[] args)
  {
    S.o.p("Banking activity started");
    Scanner scan = new Scanner(System.in);
    S.o.p("Enter the account number:");
    int acc = scan.nextInt();
    S.o.p("Enter the password: ");
    int pwd = scan.nextInt();
    Thread.sleep(5000);
    S.o.p("Collect your cash!");
    S.o.p("Banking activity completed");

    S.o.p("Addition activity started");
    int num1 = 732642346;
    int num2 = 547675347;
    Thread.sleep(5000);
    int res = num1 + num2;
    S.o.p("The sum is:" +res);
    S.o.p("Addition activity completed");

    S.o.p("Printing activity started");
    for(int i=65; i<=69; ++i)
    {
      S.o.p((char) i);
      Thread.sleep(5000);
    }
    S.o.p("Printing activity completed");
} }
```



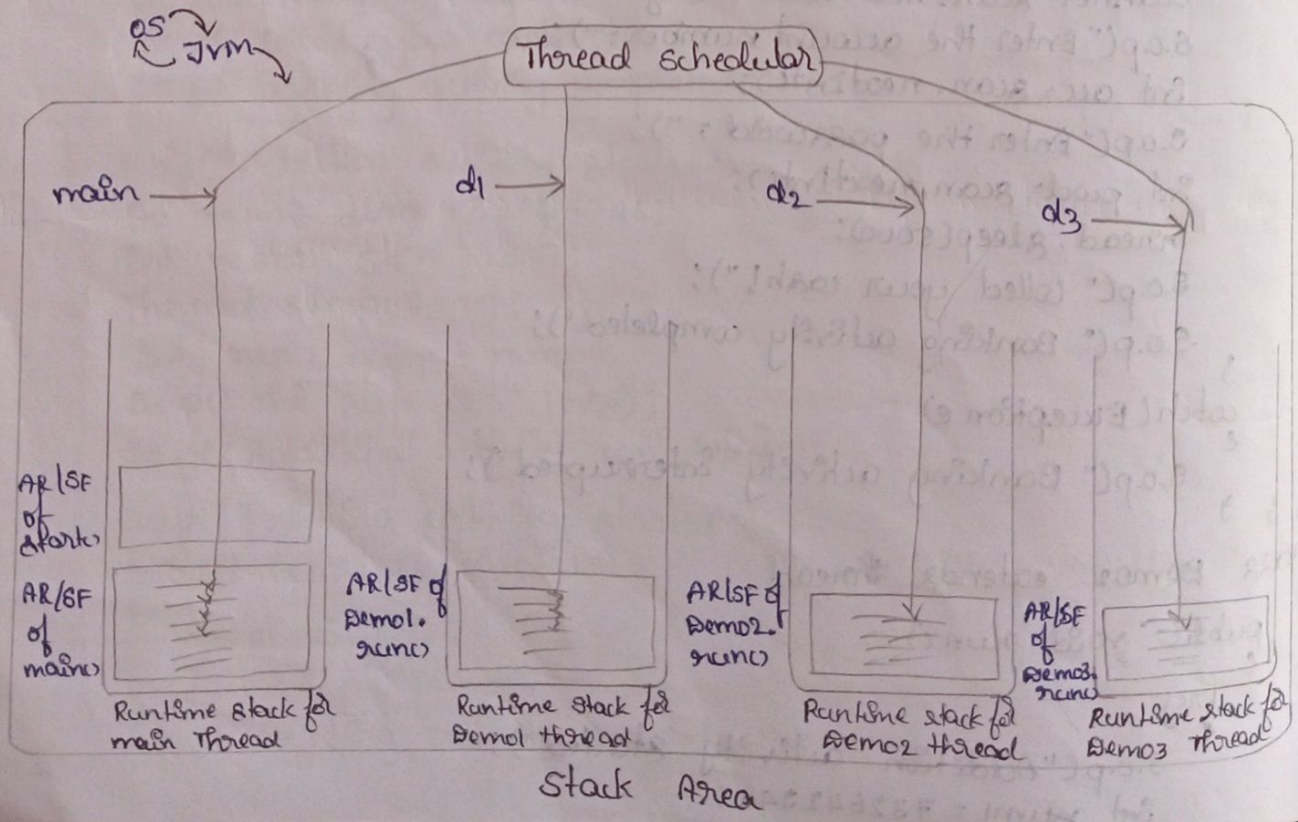
* As noticed in the above program, it is executed by a single thread called the main thread.
* The disadvantage of single threaded approach is that independed subtasks activity are forced to wait for each other. Hence increasing the waiting time of the application and resulting in in-efficient utilization of CPU time.
* The above disadvantage can be overcome by creating multiple threads & multiple stacks as shown below.

Advantage of multi-threaded approach:

Approach 1: Achieving multi-threading by extending Thread class.

```
import java.util.Scanner;
class Demo1 extends Thread
{
  public void run()
  {
    try
    {
      S.o.p("Banking activity started");
      Scanner scan=new Scanner(System.in);
      S.o.p("Enter the account number:");
      int acc=scan.nextInt();
      S.o.p("Enter the password:");
      int pwd=scan.nextInt();
      Thread.sleep(5000);
      S.o.p("Called your cash!");
      S.o.p("Banking activity completed");
    }
    catch(Exception e)
    {
      S.o.p("Banking activity interrupted");
}}}
class Demo2 extends Thread
{
  public void run()
  {
    try
    {
      S.o.p("Addition activity started");
      int num1 = 732642346;
```

```
        int num2= 547675347;
        Thread.sleep(5000);
        int res = num1 + num2;
        s.o.p("The sum is: " +res);
        s.o.p("Addition activity completed");
    }
    catch(Exception e)
    {
        s.o.p(" Addition activity interrupted");
} } }
class Demo3 extends Thread
{
    public void run()
    {
        try
        {
            s.o.p ("Printing activity started");
            for(int i=65; i<=69; i++)
            {
                s.o.p((char) i);
                Thread.sleep(5000);
            }
            S.o.p(" Printing activity completed");
        }
        catch(Exception e)
        {
            s.o.p(" Printing activity interrupted");
} } }
class Demo
{
    psvm(String[] args)
    {
        Demo1 d1= new Demo1();
        Demo2 d2 = new Demo2();
        Demo3 d3= new Demo3();
        d1.start();
        d2.start();
} }     d3.start();
```



NOTE :

13/5/23

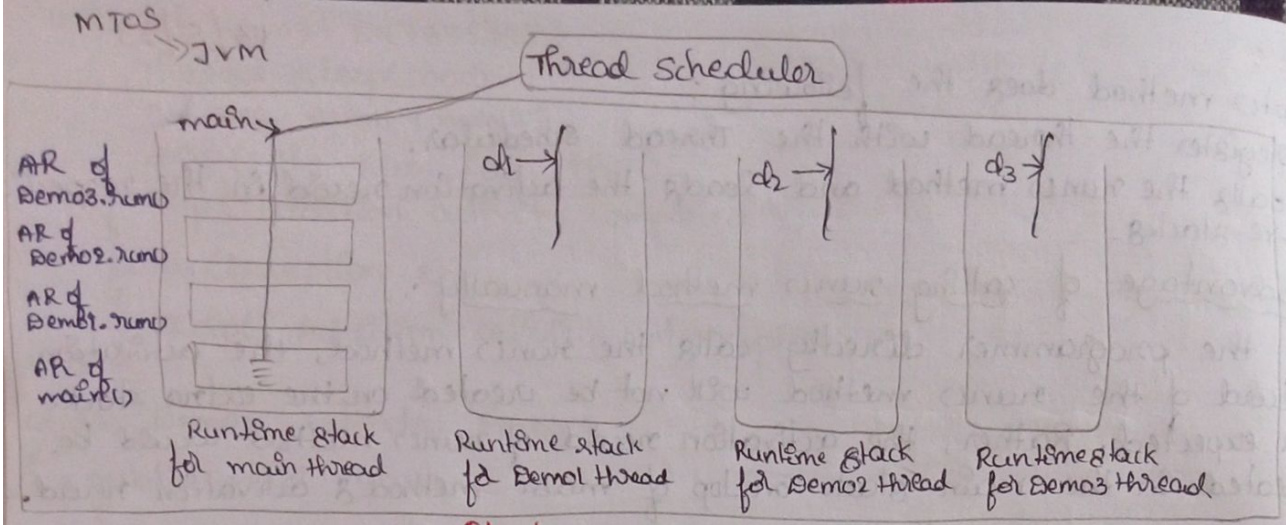
* start() method does the following :
  → register the thread with the Thread schedular.
  → calls the run() method and loads the activation record in the respective stacks.

Disadvantage of calling run() method manually :

* If the programmer directly calls the run() method, the activation record of the run() method will not be created on the extra stacks as expected. Rather, the activation record of run() method would be created in the main stack on top of main method's activation record.
* Activation records created on a specific stack can always be executed only by a single thread. Hence, in this case, all the activation records present on the main stack will be executed by the main thread. Therefore multi-threading can not be achieved.
* If activation records have to be created on the extra stacks then the programmer should not explicitly call the run() method. Instead the run() method should be implicitly called by the start() method.
* If run() method is explicitly called then it would result in sequential style of execution and not concurrent style of execution.

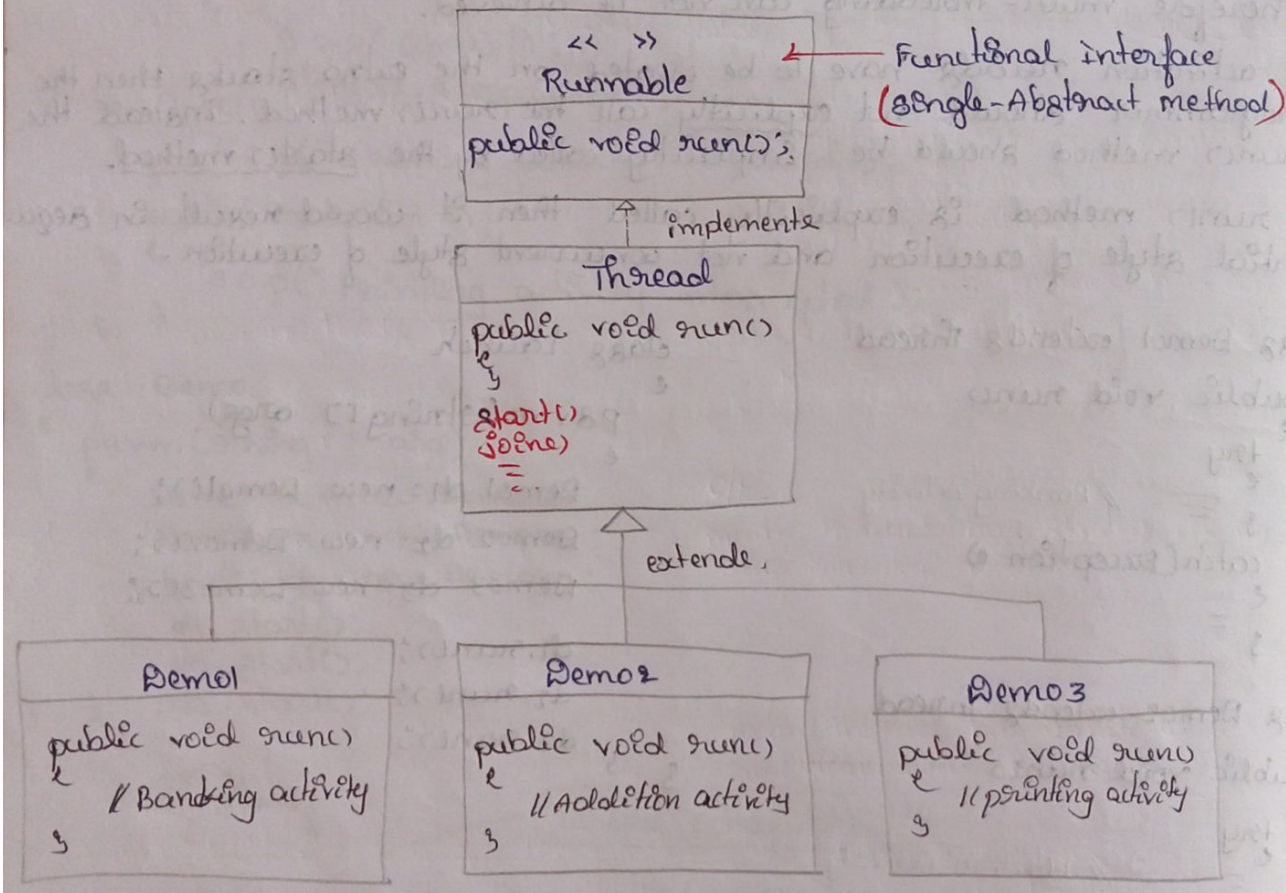
```
class Demo1 extends Thread
{
  public void run()
  {
    try
    {
    } catch(Exception e)
    {
    }
  }
}

class Demo2 extends Thread
{
  public void run()
  {
    try
    {
    } catch(Exception e)
    {
    }
  }
}

class Demo3 extends Thread
{
  public void run()
  {
    try
    {
    } catch(Exception e)
    {
    }
  }
}
```

```
class Launch
{
  P s v m(String[] args)
  {
    Demo1 d1 = new Demo1();
    Demo2 d2 = new Demo2();
    Demo3 d3 = new Demo3();

    d1.run();
    d2.run();
    d3.run();
  }
}
```



Thread Hierarchy :

15/5/23



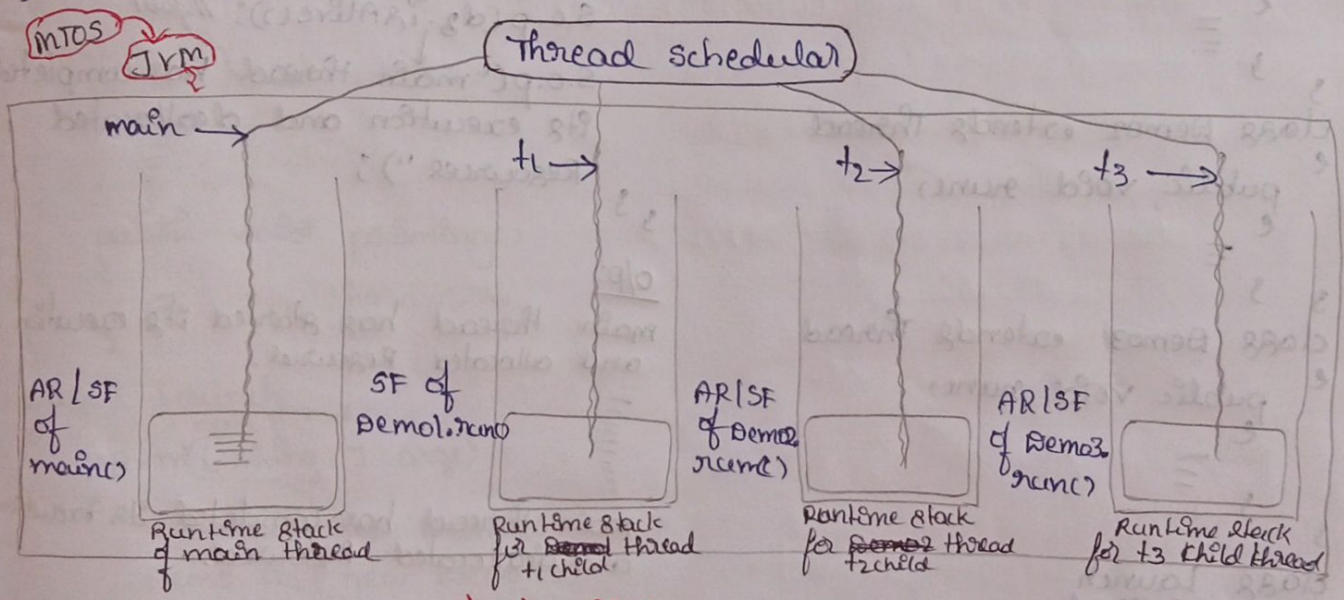
Approach 2: Achieving multi-threading by implementing the Runnable interface.

```
class Demo1 implements Runnable
{
    public void run()
    {
        ≡ // Banking activity
    }
}
class Demo2 implements Runnable
{
    public void run()
    {
        ≡ // Addition activity
    }
}
```

```
class Demo3 implements Runnable
{
  public void run()
  {
    ≡ //Printing activity
  }
}
class Launch
{
  psvm(...)
  {
    Demo1 d1=new Demo1();
    Demo2 d2=new Demo2();
    Demo3 d3=new Demo3();
                    assigning jobs
    Thread t1=new Thread(d1);
    Thread t2=new Thread(d2);
    Thread t3=new Thread(d3);

    t1.start(); //Banking job
    t2.start(); //Addition job
    t3.start(); //Printing job
  }
}
```

Jobs

workers



Stack Area.

16/5/23

NOTE: * Two ways of achieving multi-threading,
- → Extending from the Thread class.
- → Implementing from the Runnable interface.

* It is recommended to achieve multi-threading by implementing the Runnable interface. So that the implementing class can also extends from another class. This wouldn't have been possible by extending the Thread class as multiple inheritance is not permitted in Java.

NOTE: * In most of the programs, by default the main thread would be the first thread to begin its execution and the first thread to complete the execution.

* However the default behaviour of main thread is unacceptable in such programs where main thread would first begin its execution and also allocate resources using which the child threads must execute.

* If main thread is the first thread to complete its execution then all the resources allocated by main thread would also be deallocated. Hence this would create a problem for the child threads to proceed with their execution.

* We can make one thread wait for another thread to complete its execution and then proceed with its own by using a inbuilt method called as "join()" method.

* We can verify if a method thread is alive or not by using an inbuilt method called as "isAlive()" method.

* The "isAlive()" method would return true if the thread is alive and it would return "false" if the thread is not alive.

```
class Demo1 extends Thread
{
  public void run()
  {
    ==
  }
}
class Demo2 extends Thread
{
  public void run()
  {
    ==
  }
}
class Demo3 extends Thread
{
  public void run()
  {
    ==
  }
}
class Launch
{                    main thread
  psvm(String[] args) throws Exception   (due to checked exception thrown by join())
  {
    S.o.p("main thread has started its
        execution and allocated Resources");
    Demo1 d1 = new Demo1();
    Demo2 d2 = new Demo2();
    Demo3 d3 = new Demo3();
    d1.start();
    d2.start();
    d3.start();
    S.o.p(d1.isAlive()); // true
    S.o.p(d2.isAlive()); // true
    S.o.p(d2.isAlive()); // true
    d1.join(); → main thread wait till d1 thread completes its
    d2.join();          execution & joins you
    d3.join();
    S.o.p(d1.isAlive()); // false
    S.o.p(d2.isAlive()); // false
    S.o.p(d3.isAlive()); // false
    S.o.p("main thread has completed
    its execution and deallocated
    Resources");
  }
}
```

o/p:

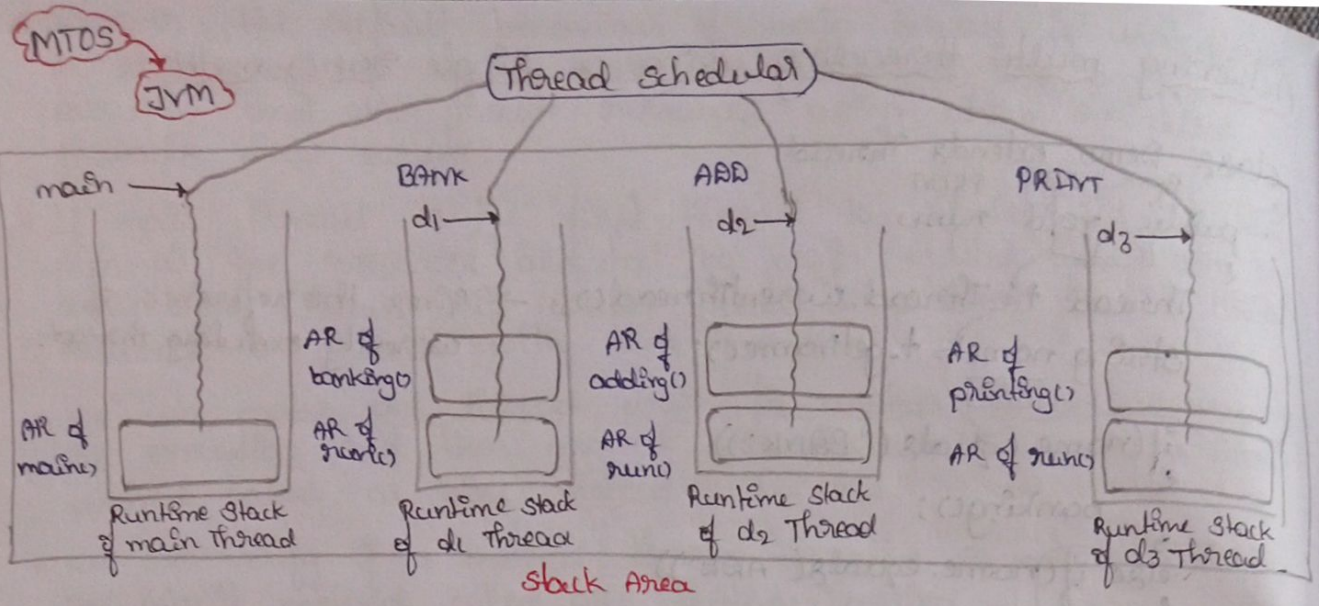
main thread has started its execution and allocated Resources.

.....

main thread has completed its execution and deallocated Resources.

Achieving multi-threading using a single run() method.

```
class Demo extends Thread
{
    BANK  ADD  PRINT
    public void run()
    {
        t
        Thread t = Thread.currentThread(); → gives the reference of currently executing thread.
        String name = t.getName();
                ADD
        if(name.equals("BANK"))
        {
            banking();
        }
        else if(name.equals("ADD"))
        {
            adding();
        }
        else
        {
            printing();
        }
    }
    public void banking()
    {
        ≡ //Banking Activity
    }
    public void adding()
    {
        ≡ //Adding Activity
    }
    public void printing()
    {
        ≡ //Printing Activity
    }
}
class Launch
{
    main Thread
    psvm(String[] args)
    {
        Demo d1 = new Demo();
        Demo d2 = new Demo();
        Demo d3 = new Demo();
        S.o.p(d1.getName()); // Thread-0
        S.o.p(d2.getName()); // Thread-1
        S.o.p(d3.getName()); // Thread-2
        d1.setName("BANK");
        d2.setName("ADD");
        d3.setName("PRINT");
        d1.start();
        d2.start();
        d3.start();
    }
}
```

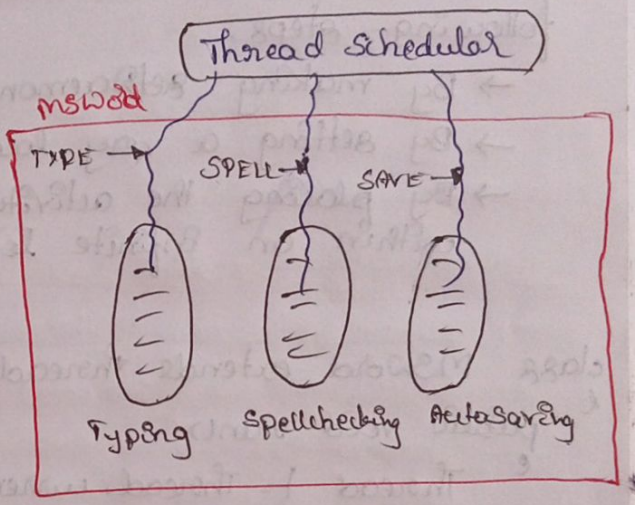


17/5/23

Demonstration of Race condition in Java:

```
class MsWord extends Thread
{
   public void run()
   {
      Thread t= Thread.currentThread();
      String name= t.getName();
      if(name.equals("TYPE"))
         typing();
      else if(name.equals("SPELL"))
         spellChecking();
      else
         autoSaving();
   }
   public void typing()
   {
      try
      {
         for(int i=1; i<=5; ++i)
         {
            S.O.P("Typing...");
            Thread.sleep(5000);
      }  }
      catch(Exception e)
      {
         S.O.P("Typing interrupted");
   }  }
   public void spellChecking()
   {
      try
      {
         for(int i=1; i<=5; ++i)
         {
            S.O.P("Spell checking....");
         }  Thread.sleep(5000);
      }
      catch(Exception e)
      {
         S.O.P("spell chaking interrupted");
   }  }
```

```
public void autoSaving()
{
    try
    {
        for(int i=1; i<=5; ++i)
        {
            S.O.p("Auto saving.....");
            Thread.sleep(5000);
        }
    }
    catch(Exception e)
    {
        S.O.p("Auto saving interrupted");
    }
}
class Launch
{
    psvm(String[] args)
    {
        MSWord w1 = new MSWord();
        MSWord w2 = new MSWord();
        MSWord w3 = new MSWord();

        w1.setName("TYPE");
        w2.setName("SPELL");
        w3.setName("SAVE");

        w1.start();
        w2.start();
        w3.start();
    }
}
```



O/P:

I
Auto saving....
Typing....
Spell checking...

II
Auto saving....
Spell checking...
Typing...

III
Typing....
Spellchecking...
Auto saving...

NOTE:

* The above program suffers from 'race condition'.
* Race condition is a phenomenon in multithreading where multiple threads fight amongst each other for CPU time and race towards completion. Hence the problem in the above program is, it produces different outputs everytime the program is executed. This makes the application inconsistent and unreliable. In the above program, typing activity is the main activity whereas spell-checking and auto-saving are subsidary activities.
* The thread which is executing the main activity should be assisted with the threads executing the subsidary activities.
* The main activity thread should be the first thread to complete its execution and only after which the helper threads should finish their execution.

Daemon Threads:

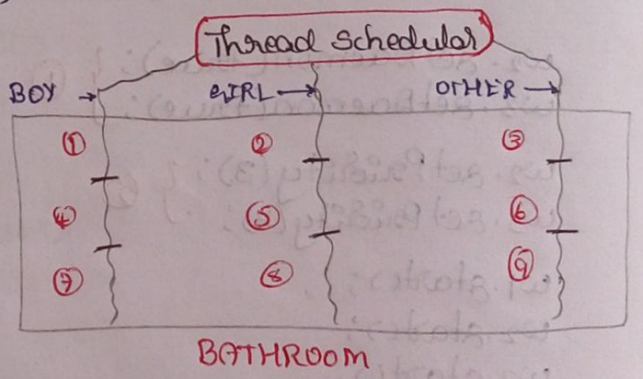
* The solution to the above problem can be provided by making use of daemon threads.
* In every application, there would be a main activity and other subsidiary activities. The subsidiary activities must only finish execution after the main activity finishes execution. This can be achieved by ensuring the subsidiary activities are executed by daemon threads.
* Daemon threads are such low-priority threads which only conclude their execution after the normal threads finish their execution.
* A thread can be converted into a daemon thread using the following steps.
  - → By making setDaemon() method as true.
  - → By setting a very low priority (<5) by using setPriority().
  - → By placing the activity which daemon thread execute within an infinite loop.

```
class MSWord extends Thread
{
  public void run()
  {
    Thread t= Thread.currentThread();
    String name= t.getName();
    if(name.equals("TYPE"))
    {
      typing();
    }
    else if(name.equals("SPELL"))
    {
      spellchecking();
    }
    else
    {
      autoSaving();
    }
  }
  public void typing()
  {
    try
    {
      for(int i=1; i<=5; ++i)
      {
        S.O.p("Typing....");
        Thread.sleep(3000);
      }
    }
    catch(Exception e)
    {
      S.O.p("Typing interrupted");
    }
  }
  public void spellchecking()
  {
    try    (3)
    {
      for(;;)
      {
        S.O.p("Spell checking....");
      }
      Thread.sleep(3000);
```

```
        catch(Exception e)
        {
            S.o.p(" Spell checking interrupted");
        }
    }
    public void autosaving()
    {
        try
        {
            for(;;)                    (3)
            {
                S.o.p(" Auto saving ....");
                Thread.sleep(3000);
            }
        }
        catch(Exception e)
        {
            S.o.p(" Auto saving interrupted");
        }
    }
}
class Launch
{
    psvm(String[] args)
    {
        MSWord w1 = new MSWord();
        MSWord w2 = new MSWord();
        MSWord w3 = new MSWord();
        w1.setName(" TYPE");
        w2.setName("SPELL");
        w3.setName(" SAVE");

        w2.setDaemon(true);
        w3.setDaemon(true);          } (1)

        w2.setPriority(3);
        w3.setPriority(3);           } (2)

        w1.start();
        w2.start();
        w3.start();
    }
}
```

Disadvantage of multi-threading:

```
class Bathroom implements Runnable
{
  public void run()
  {
    Thread t = Thread.currentThread();
    String name = t.getName();
    try
    {
      S.O.P(name + "has entered the bathroom");
      Thread.sleep(5000);
      S.O.P(name + " is using the bathroom");
      Thread.sleep(5000);
      S.O.P(name + " has exited the bathroom");
    }
    catch (Exception e)
    {
      S.O.P(" Bathroom activity interrupted");
} } }
class Launch
{
  parm (String[] args)
  {
    Bathroom b = new Bathroom(); // Job & Resource
    Thread t1 = new Thread(b);
    Thread t2 = new Thread(b);
    Thread t3 = new Thread(b);   // Workers.
    t1.setName("BOY");
    t2.setName("GIRL");
    t3.setName("OTHER");
    t1.start();
    t2.start();
    t3.start();
} }
```



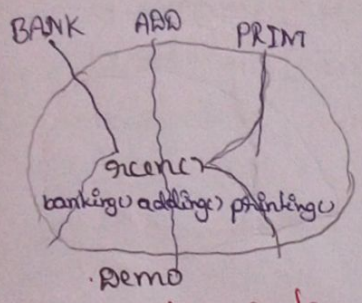
O/P:
```
BOY has entered the bathroom
GIRL has entered the bathroom
OTHER has entered the bathroom
BOY is using the bathroom
GIRL is using the bathroom
OTHR is using the bathroom
BOY has exited the bathroom
GIRL has exited the bathroom
OTHER has exited the bathroom.
```

18/5/23

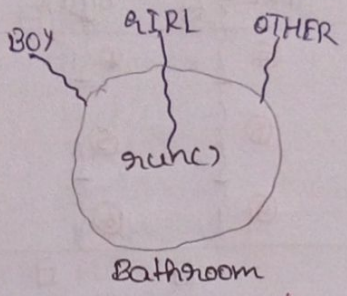
NOTE: * The problem in the above program is that a common shared resource (bathroom) is being accessed by 3 threads (boy, girl, other) and there is no synchronization b/w these threads.

* In otherwords, any one of the 3 threads starts accessing the bathroom resource, then the other 2 threads should not be

allowed to access the same resource and they must wait.



No Synchronization



Synchronization

The solution to the above problem can be provided:

Approach 1: By using join() method

```
class Launch
{
    psvm(String[] args) throws Exception
    {
        Bathroom b = new Bathroom();
        Thread t1 = new Thread(b);
        Thread t2 = new Thread(b);
        Thread t3 = new Thread(b);
        t1.setName("BOY");
        t2.setName("GIRL");
        t3.setName("OTHER");

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

Output:

```
BOY has entered the bathroom
BOY is using the bathroom
BOY has exited the bathroom
GIRL has entered the bathroom
GIRL is using the bathroom
GIRL has exited the bathroom.
OTHER has entered the bathroom
OTHER is using the bathroom
OTHER has exited the bathroom.
```

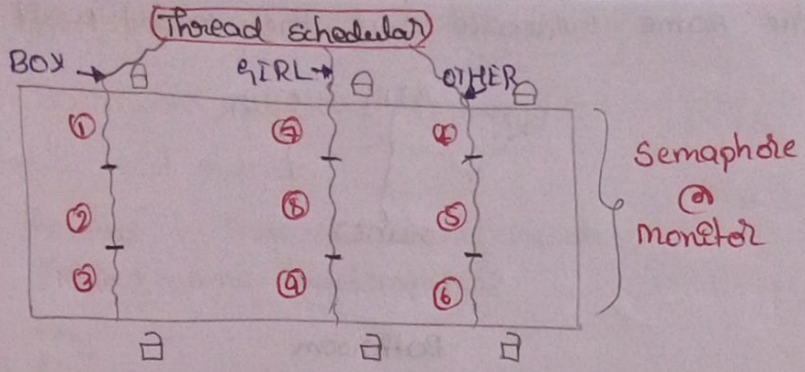
* As noticed in the output of above program, the order of execution of threads is restricted hence using join() method, an efficient solution can not be provided.

Approach 2: By using synchronized keyword

```
class Bathroom implements Runnable
{
    synchronized public void run()
    {
    }
}
class Launch
{
    psvm(String[] args)
    {
        Bathroom b = new Bathroom();
        Thread t1 = new Thread(b);
        Thread t2 = new Thread(b);
        Thread t3 = new Thread(b);
        t1.setName("BOY");
        t2.setName("GIRL");
        t3.setName("OTHER");
        t1.start();
        t2.start();
        t3.start();
    }
}
```

Output:

```
BOY has entered the bathroom
BOY is using the bathroom
BOY has exited the bathroom

OTHER has entered the bathroom
OTHER is using the bathroom
OTHER has exited the bathroom

GIRL has entered the bathroom
GIRL is using the bathroom
GIRL has exited the bathroom
```



* When multiple threads are accessing a common shared resources, synchronisation could be lacking b/w the threads.
* In order to achieve synchronization b/w the threads, the shared resource must be locked. Locks can be applied by making use of 'synchronized' keyword.
* Such statements which at any given point of time must be executed by only one thread are referred to as 'Semaphore' (or) 'monitor'.

synchronized keyword can be applied in 2 places:

case 1: When all the statements present inside the method must be locked, the method itself should be declared as synchronized.

general syntax: synchronized public void run()
{
}

Eg:
```
class Printer implements Runnable
{
    synchronized public void run()
    {
        Thread t = Thread.currentThread();
        String name = t.getName();
        try
        {
            S.O.p(name + "message printing is started");
            Thread.sleep(5000);
            S.O.p(name + "message printing is in progress");
            Thread.sleep(5000);
            S.O.p(name + "message printing is completed");
        }
        catch (Exception e)
        {
            S.O.p("Printing activity is interrupted");
        }
    }
}
class Launch
{
    P S V m(String[] args)
    {
        Printer p = new Printer();
        Thread t1 = new Thread(p);
        Thread t2 = new Thread(p);
        Thread t3 = new Thread(p);
```

```
t1.setName("Kannada");
t2.setName("Hindi");
t3.setName("English");
t1.start();
t2.start();
t3.start();
}
}
```

output:

```
Kannada - - - -
Kannada - - - -
Kannada - - - -
English - - - -
English - - - -
English - - - -
Hindi - - - -
Hindi - - - -
Hindi - - - -
```

case 2: When only few statements (not all statements) present inside a method must be locked, then such statements must be placed within a synchronized block.

General syntax:

```
public void run()
{
    ==
    synchronized (this)
    {
        ≡
    }
    ≡
}
```

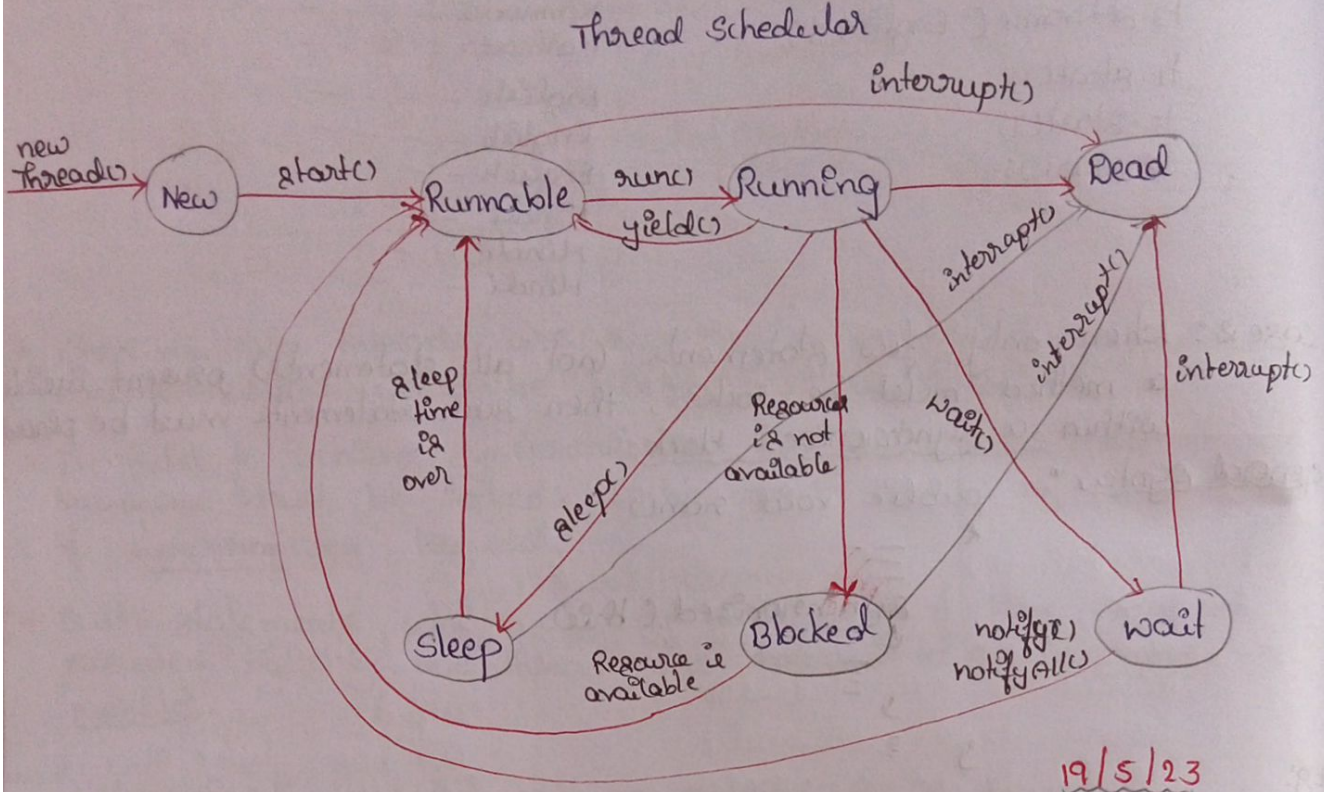
Eg:

```
class Resource implements Runnable
{
  public void run()
  {
    try
    {
      for(int i=1; i<=5; ++i)
      {
        S.O.p(i);
        Thread.sleep(5000);
      }
      synchronized(this)
      {
        for(int i=65; i<=69; ++i)
        {
          S.O.p((char)i);
          Thread.sleep(5000);
  } } }
    catch(Exception e)
    {
      e.printStackTrace();
  } } }
class Launch
{
  public static void main(String[] args)
  {
    Resource res = new Resource();
    Thread t1 = new Thread(res);
    Thread t2 = new Thread(res);
    Thread t3 = new Thread(res);
    t1.start();
    t2.start();
    t3.start();
  } }
```

output:

```
1
1
1
2
2
2
3
3
3
4
4
4
5
5
5
A
B
C
D
E
A
B
C
D
E
A
B
C
D
E
```

Thread state diagram:

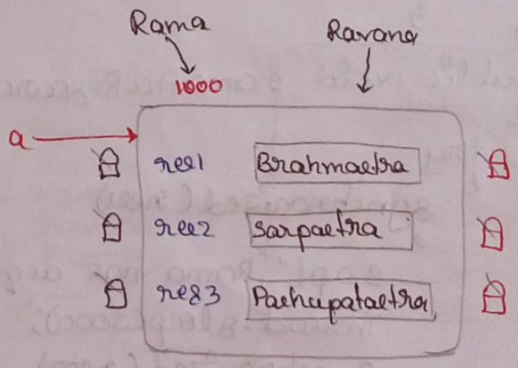


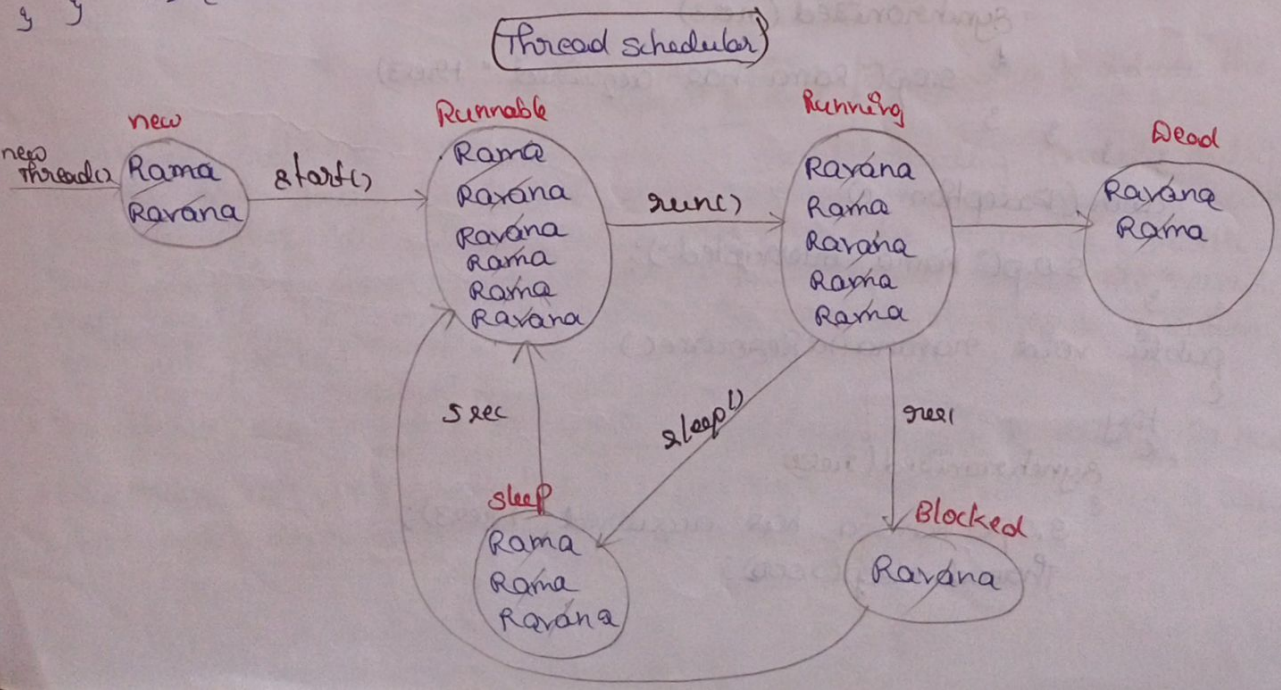
19/5/23

Program to demonstrate Thread state diagram:

```
class Astra implements Runnable
{
    String res1 = new String("Brahmastra");
    String res2 = new String("Sarpastra");
    String res3 = new String("Pashupatastra");
    public void run()                          // Rama, Ravana
    {
        Thread t = Thread.currentThread();
        String name = t.getName();
        if(name.equals("Rama"))
        {
            ramaAccResource();
        }
        else
        {
            ravaAccResource();
        }
    }
    public void ramaAccResource()
    {
        try
        {
            synchronized(res1)
            {
                S.o.p("Rama has acquired" +res1);
                Thread.sleep(5000);
                synchronized(res2)
                {
                    S.o.p("Rama has acquired" +res2);
                    Thread.sleep(500);
```

```
        synchronized (res3)
        {
            S.o.p(" Rama has acquired " +res3);
        }
    }
    catch(Exception e)
    {
        S.o.p(" Rama Interrupted ");
    }
}
public void ravanaAccResource()
{
    try
    {
        synchronized (res1)
        {
            S.o.p(" Ravana has acquired " +res1);
            Thread.sleep(5000);
            synchronized (res2)
            {
                S.o.p(" Ravana has acquired " +res2);
                Thread.sleep(5000);
                synchronized (res3)
                {
                    S.o.p(" Ravana has acquired " +res3);
                }
            }
        }
    }
    catch(Exception e)
    {
        S.o.p(" Ravana Interrupted ");
    }
}
class Ramayana
{
    psvm(String[] args)
    {
        Astra a = new Astra();
        Thread t1 = new Thread(a);
        Thread t2 = new Thread(a);
        t1.setName(" Rama ");
        t2.setName(" Ravana ");
        t1.start();
        t2.start();
    }
}
```



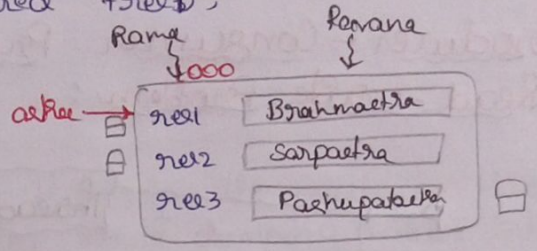


O/P:
```
Rama has acquired Brahmastra
Rama has acquired Sarpastra
Rama has acquired Pashupatastra
Ravana has acquired Brahmastra
Ravana has acquired Sarpastra
Ravana has acquired Pashupatastra
```

Program to demonstrate deadlock:

```
class Astra implements Runnable
{
    String res1 = new String("Brahmastra");
    String res2 = new String("Sarpastra");
    String res3 = new String("Pashupatastra");

    Rama                    Ravana
    public void run()
    {
        Thread t = Thread.currentThread();
        String name = t.getName();
        if(name.equals("Rama"))
        {
            ramaAccResource();
        }
        else
            ravanaAccResource();
    }

    public void ramaAccResource()
    {
        try
        {
            synchronized(res1)
            {
                S.o.p("Rama has acquired "+res1)
                Thread.sleep(5000);
                synchronized(res2)
                {
                    S.o.p("Rama has acquired "+res2)
                    Thread.sleep(5000);
                    synchronized(res3)
                    {
                        S.o.p("Rama has acquired "+res3)
                    }
                }
            }
        }
        catch(Exception e)
        {
            S.o.p("Rama interrupted");
        }
    }

    public void ravanaAccResource()
    {
        try
        {
            synchronized(res2)
            {
                S.o.p("Ravana has acquired "+res3);
                Thread.sleep(5000);
```

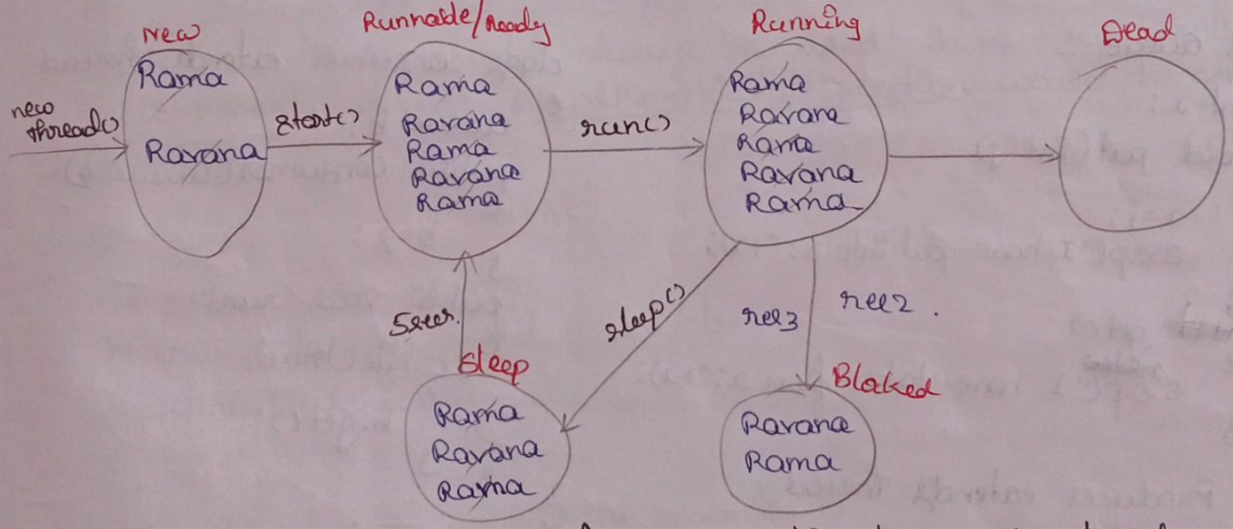
```
synchronized (res2)
{
  S.o.p("Ravana has acquired " +res2);
  Thread.sleep(5000);
  synchronized (res1)
  {
    S.o.p("Ravana has acquired " +res1);
  } } } }
  catch(Exception e)
  {
    S.o.p(" Ravana Interrupted ");
  } } }
class Ramayana
{
  psvm(String[] args)
  {
    Astra astra = new Astra();
    Thread t1 = new Thread(astra);
    Thread t2 = new Thread(astra);
    t1.setName(" Rama ");
    t2.setName(" Ravana ");
    t1.start();
    t2.start();
  } }
```



o/p:

Rama has acquired Brahmaastra
Ravana has acquired Pashupataastra
Rama has acquired sarpaastra.

↳ Program never terminates.



Threads waiting for eachother to release the resources forever

NOTE:

* Deadlocks refer to a pheromenon in multithreading wherein multiple threads are stuck in blocked state because they are mutually waiting for each other to release the resource inorder to proceed with their execution. Since none of the threads would release the resource they would be permanently stuck in blocked state and execution will not proceed.

* Deadlocks occurs due to cyclic dependencies which exist b/w threads

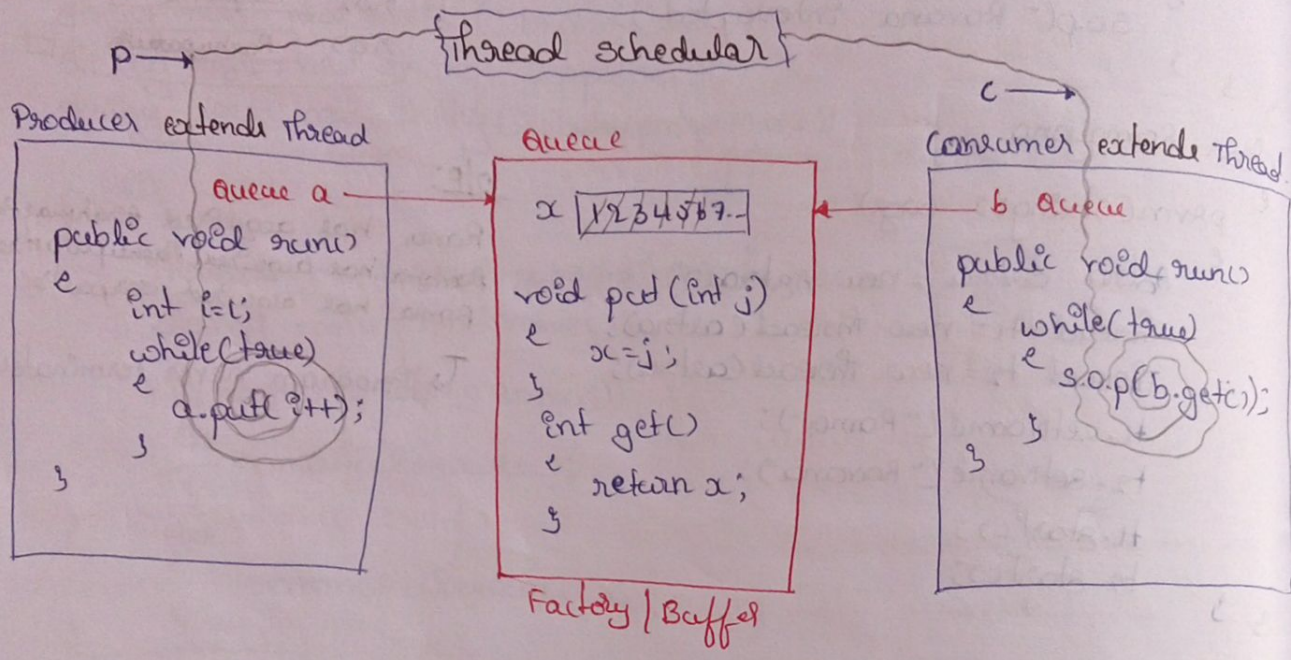
* Deadlocks can be prevented by coding application in such a way that cyclic dependancy does not exist.

Eg 1: Using thread.join() method we can make one thread wait for another thread to finish its execution.

Eg 2: Avoid unnecessary locks.

Eg 3: Avoid nested locks, etc.

Producer - Consumer Problem / Bounded Buffer Problem / Read - write Problem:




```
class Queue
{
  int x;
  void put(int j)
  {
    x=j;
    S.o.p("I have put into x: "+x);
  }
void get()
  {
    S.o.p("I have taken from x: "+x);
  }
}
class Producer extends Thread
{
  Queue a;
  public Producer(Queue q)
  {
    a=q;
  }
  public void run()
  {
    int i=1;
    while(true)
    {
      a.put(i++);
    }
  }
}
```

```
class Consumer extends Thread
{
  Queue b;
  public Consumer(Queue q)
  {
    b=q;
  }
  public void run()
  {
    while(true)
    {
      b.get();
    }
  }
}
class Launch
{
  perm(String[] args)
  {
    Queue q = new Queue();
    Producer p=new Producer(q);
    Consumer c=new Consumer(q);
    p.start();
    c.start();
  }
}
```

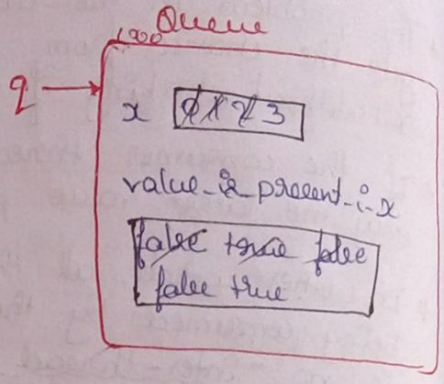
o/p:
```
I have put into x: 1
I have put into x: 2
I have put into x: 3
I have put into x: 4
I have put into x: 5
I have taken from x: 5
I have taken from x: 5
I have taken from x: 5
I hava put into x: 6
  :
```

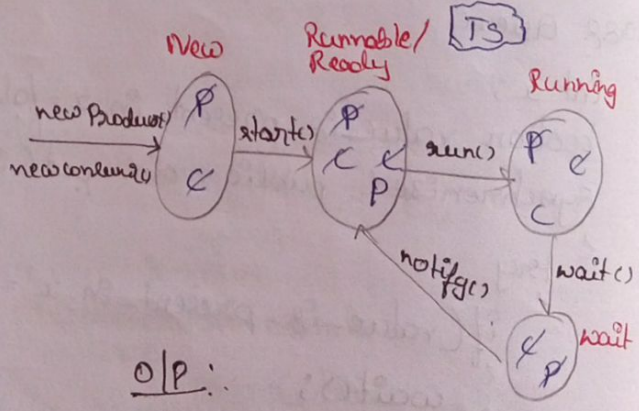
* The problem in the above program is that when the producer thread gets the chance from TS it repeatedly keeps dumping values into x without checking if the consumer has consumed the values or not.
* If the consumer thread gets a chance from TS it keeps consuming only the latest value produced by the producer.
* In other words, all the values produced by the producer are not being consumed by the consumer. This problem exist because there is no "inter-thread communication" b/w the threads where in the producer thread after producing a value should wait & notify the consumer thread and the consumer thread after consuming the value should also wait and notify to the producer thread.
* Inter-thread communication can be achieved by making use of the wait() method and notify() methods.
* wait() & notify() should always be used in a synchronized environment, if not IllegalMonitorStateException would be generated.

Solution:

```
class Queue
{
  int x;
  boolean value_is_present_in_x = false;
  synchronized public void put(int j)
  {
    try
    {
      if(value_is_present_in_x == true)
      {
        wait();
      }
      else
      {
        x=j;
        S.o.p("I have put into x: "+x);
        value_is_present_in_x = true;
      } } notify();
    catch(Exception e)
    {
      S.o.p("Producer interrupted");
    } }
```

```
synchronized void get()
{
  try
  {
    if(value_is_present_in_x == false)
    {
      wait();
    }
    else
    {
      S.O.P("I have taken from x:" +x);
      value_is_present_in_x = false;
    } } notify();
  catch(Exception e)
  {
    S.O.P("consumer interrupted");
} } }

class Producer extends Thread
{
  Queue a;
  public Producer(Queue q)
  {
    a=q;
  }
  public void run()
  {
    int i=1;
    while(true)
    {
      a.put(i++);
} } }

class Consumer extends Thread
{
  Queue b;
  public Consumer(Queue q)
  {
    b=q;
  }
  public void run()
  {
    while(true)
    {
      b.get();
} } }

class Launch
{
  psvm(String[] args)
  {
    Queue q=new Queue();
    Producer p=new Producer(q);
    Consumer c=new Consumer(q);
    p.start();
    q.start();
} }
```





O/P:

```
I have put into x: 1
I have taken from x: 1
I have put into x: 2
I have taken from x: 2
I have put into x: 3
I have taken from x: 3
        :
        :
```