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
  {
  }
}
```

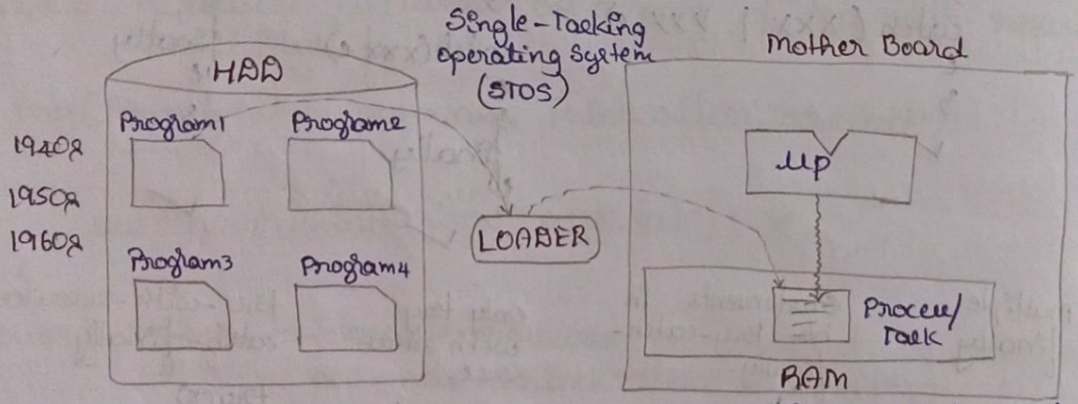
✓

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:



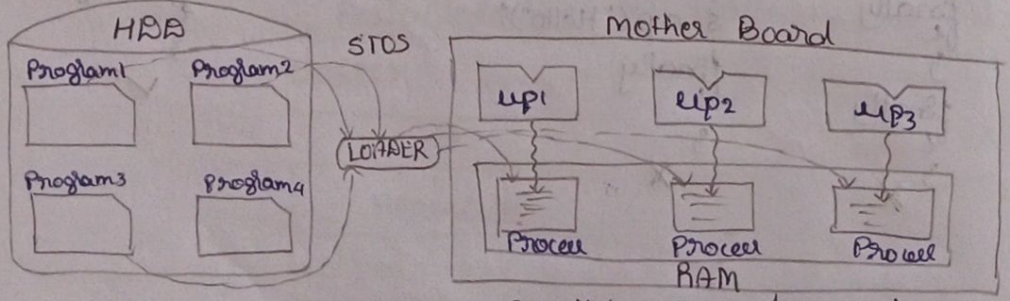
sequencial Style of Execution

[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



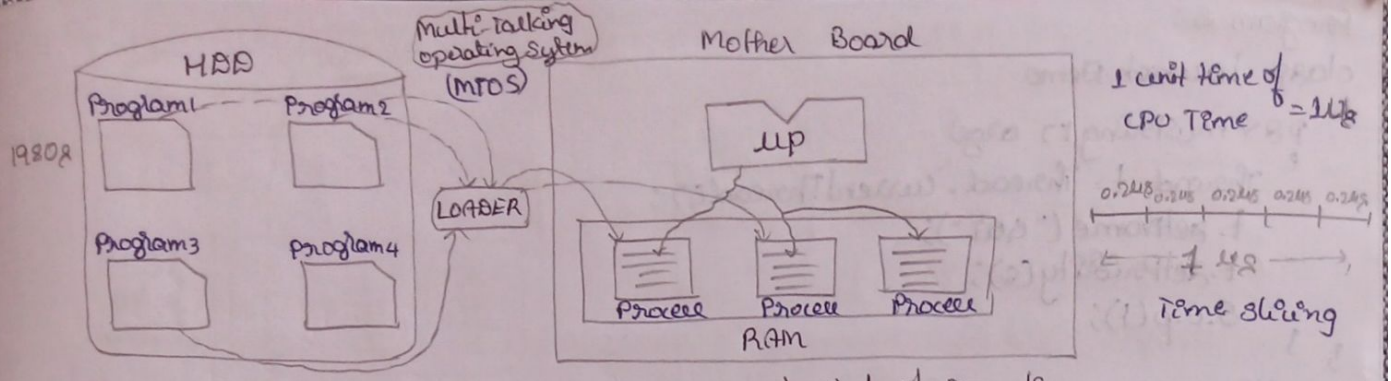
Parallel Style of Execution

[Each processor will be executing a seperate proceed]

Adv: Fast in Execution

Disad: Expensive

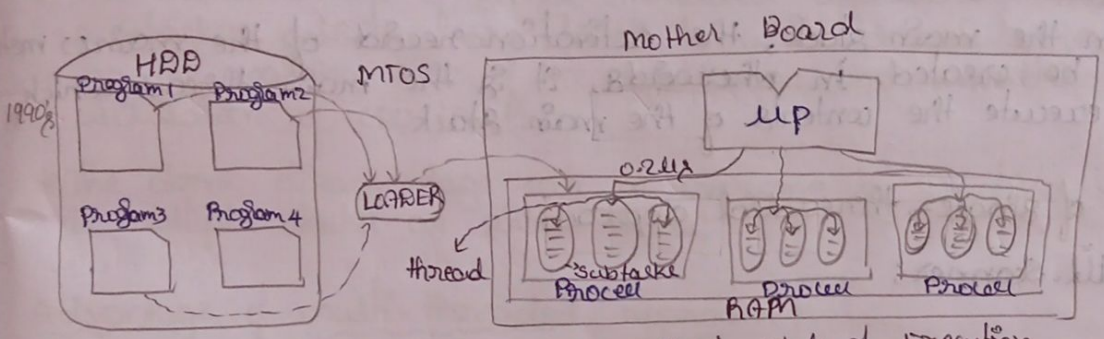
Multi-Tasking system using Single-Tasking OS & Multi-Proceed



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.

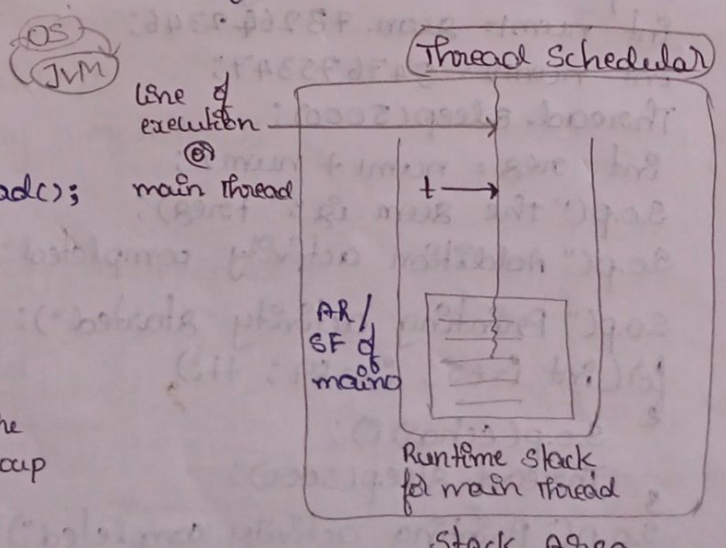
Each part of such program will be executed by a separate thread. In other words, a thread is a line of execution within a program

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.

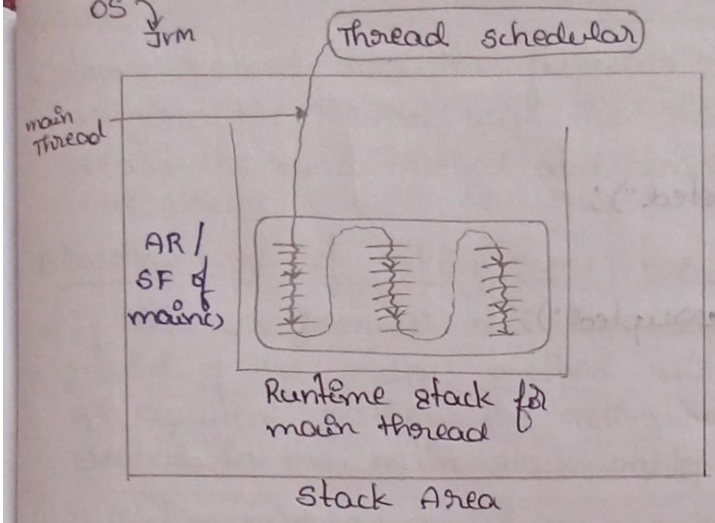
* It is on the main stack, the activation record of the main() method will be created. In otherwords, it is the main thread which would execute the contents of the main stack.

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");
} }
```



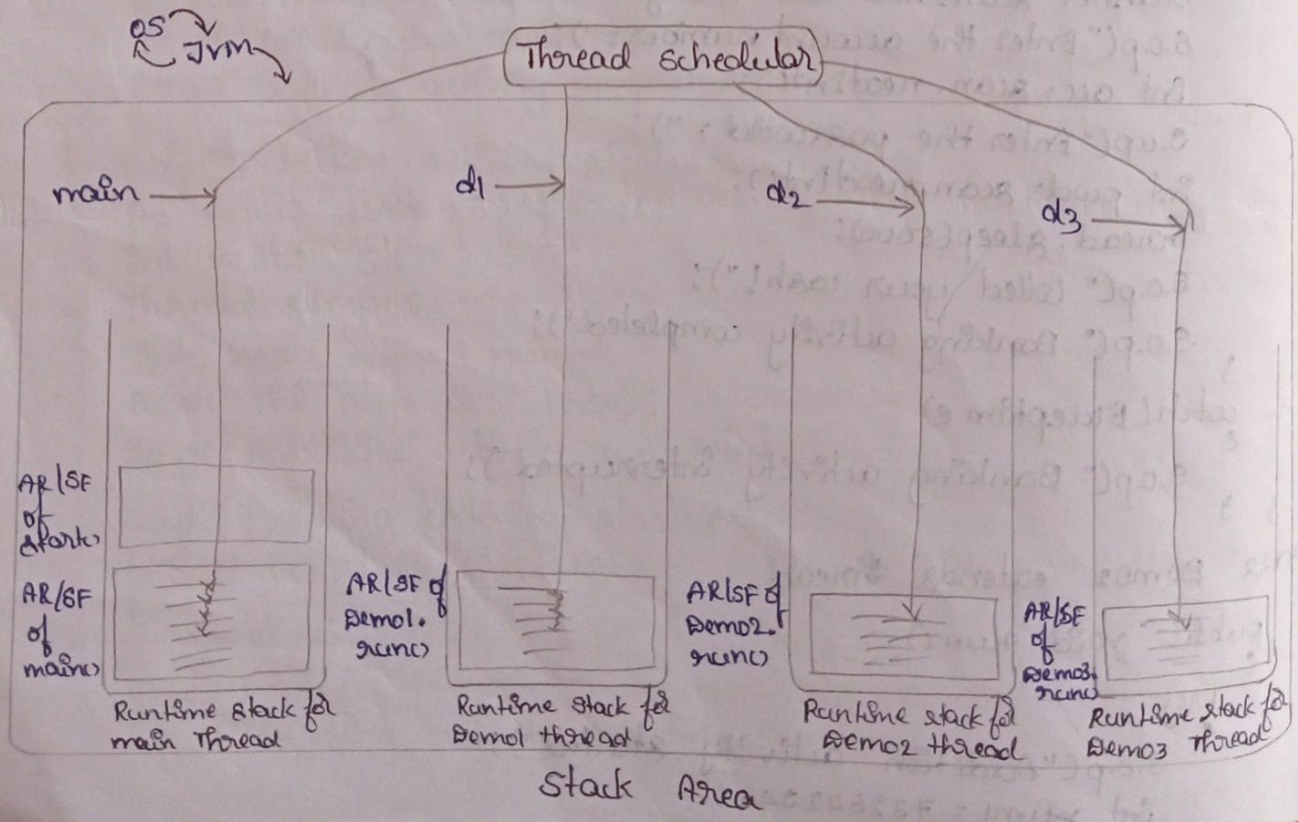
* As noticed in the above program, it is executed by a single thread called the main thread.
* The disadvantage of single threaded approach is that independed subtasks activity are forced to wait for each other. Hence increasing the waiting time of the application and resulting in in-efficient utilization 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.

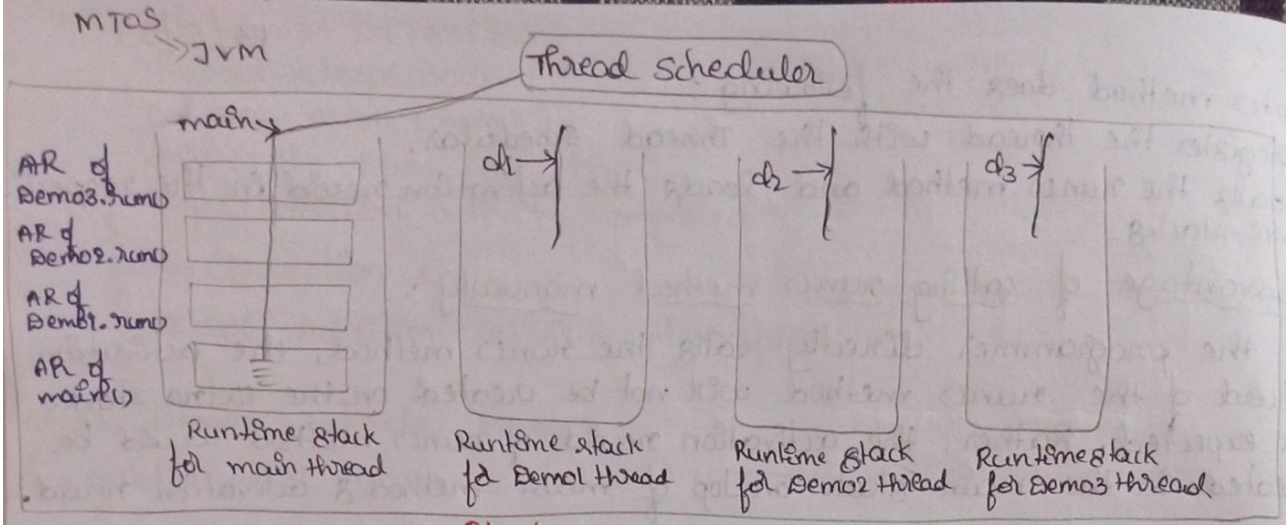
```
class Demo1 extends Thread
{
  public void run()
  {
    try
    {
    } catch(Exception e)
    {
    }
  }
}

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

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

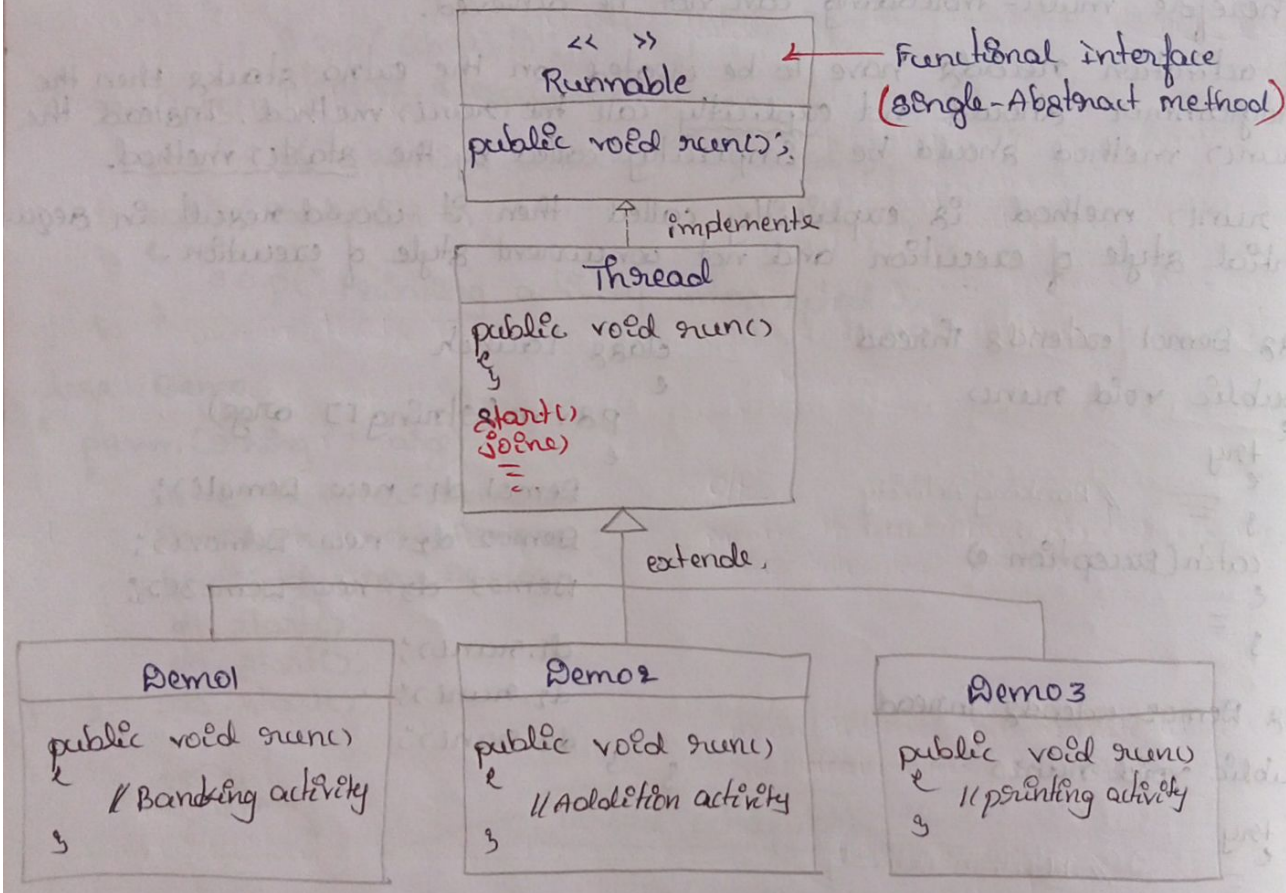
```
class Launch
{
  P s v m(String[] args)
  {
    Demo1 d1 = new Demo1();
    Demo2 d2 = new Demo2();
    Demo3 d3 = new Demo3();

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



Thread Hierarchy :

15/5/23



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

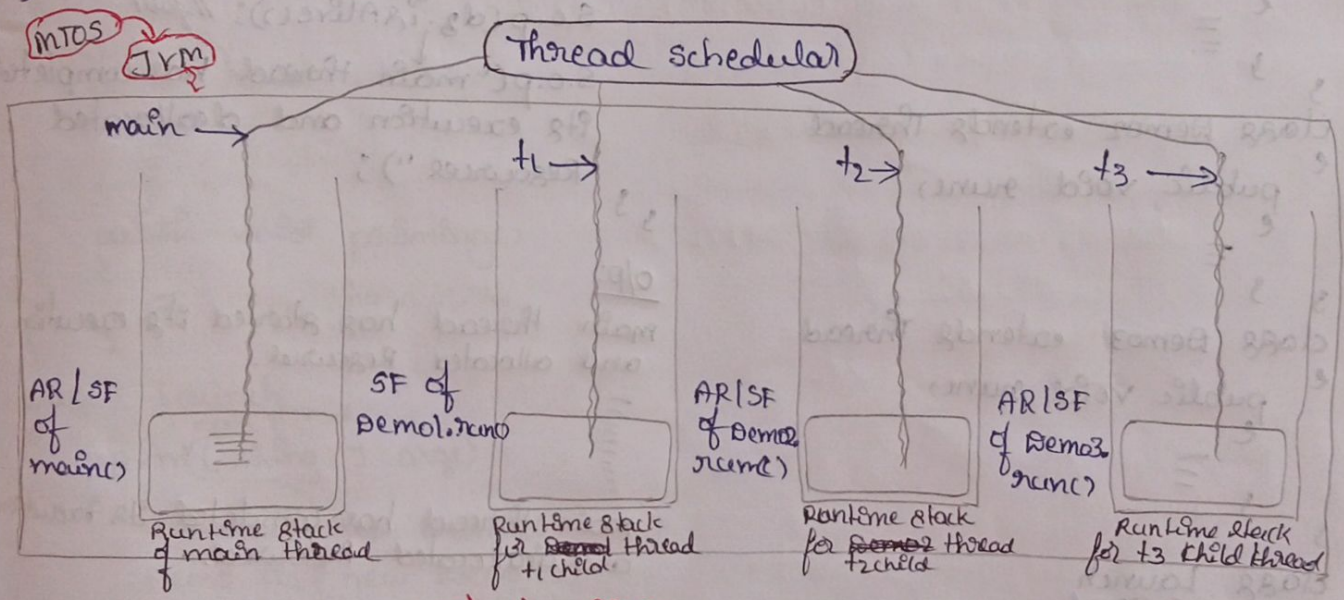
```
class Demo1 implements Runnable
{
    public void run()
    {
        ≡ // Banking activity
    }
}
class Demo2 implements Runnable
{
    public void run()
    {
        ≡ // Addition activity
    }
}
```

```
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.