1. Ways to define a Thread
2. By extending Thread Class
3. By implementing Runnable Interface
4. Method to prevent Thread execution
5. Yield ()
6. Join ()
7. Sleep ()
8. Inter Thread Communication
9. Wait
10. Notify
11. Notify All

**User and Daemon Thread**

1. **User threads** are created by the application (**user**) to perform some specific task. Whereas

**daemon threads** are mostly created by the JVM to perform some background tasks like garbage collection.

1. JVM will wait for **user threads** to finish their tasks. JVM will not exit until all **user threads**

finish their tasks.

**Multitasking:**

Executing several tasks simultaneously is the concept of Multitasking. There are 2 types of multitasking.

1. **Process Based**: Executing several tasks simultaneously where each task is separate independent program(process) is called process based multitasking.

**Ex**: 1. Typing a Java Program in editor.

2. Listening audio songs from system.

3. Downloading a file from internet.

All these tasks executed simultaneously independent of each other. This type of multitasking is applicable in OS level.

1. **Thread Based:** Executing several tasks simultaneously, where each task is separate independent part of same program. Each independent part is called a thread.

Thread based multitasking is best suitable at programmatic level.

**Advantage of Multitasking:** Weather it is process or thread based, the main objective of multitasking is to reduce the response time of system and to improve performance.

**Where we can use Multithreading concepts:**

To develop multimedia graphics

To develop animation

To develop Video Games

To develop Web servers and application servers etc.

To develop web server and application servers etc. Tomcat has by default 60 thread in the server.

When compared with old languages developing multi-threaded applications in java is very easy. Because Java provides inbuilt support for multi-threading with rich API by (Thread, Runnable, Thread Group)

Thread is a flow of execution. For every thread, a separate independent job is there.

**Defining a thread:**

We can define a thread in following 2 ways:

1. By extending Thread Class
2. By Implementing Runnable Interface.

**By Extending Thread Class:**

Create a class by extending the Thread class and override the run method. Main thread creates child thread object. Main thread starts child thread. Job of child thread is to execute the code written inside run () method.

Class MyThread extends Thread {

Public void run () {

----------------------------Job of thread------------

}

}

Every Java program contains only one normal thread i.e. Main Thread

To start the thread:

Class ThreadDemo {

Psvm (String[] args) {

**--In this point 1 thread--**

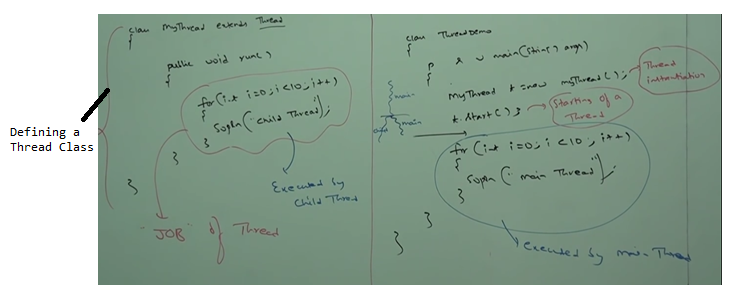
MyThread t = new MyThread ();🡪 **Thread Instantiation. Main thread creates a child thread.**

t.start();🡪**main thread starting of the child thread**

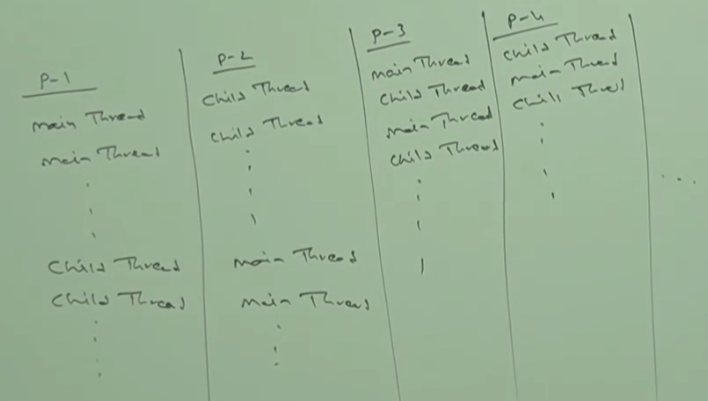
**--After t.start () In this point 2 thread---**

}

}



Possible outputs for above program:



**Case-1:**

**Thread Scheduler**: It is the part of JVM. It is responsible to schedule threads i.e. if multiple threads are waiting to get the chance of execution then in which order threads will be executed is decided by Thread Scheduler.

We can’t expect exact algorithm followed by thread scheduler. It is varied from JVM to JVM. Hence, we can’t expect thread execution order and exact output. Whenever situation comes to multi-threading there is no guarantee for exact output but we can provide several possible outputs.

**Case2:**

Whenever we are calling t.start() then start method of **Thread** class will be executed. So, thread class start method is responsible to start our thread. Thread class start method internally calls the run method.

**Difference between t.start() and t.run()**

In case of t.start🡪 2 thread A new thread will be created and that thread is responsible for execution of run method our job.

In case of t.run🡪 only 1 thread. No thread will be created and run method will be executed just like normal method call by the main thread only. There won’t be any separate thread in this case.

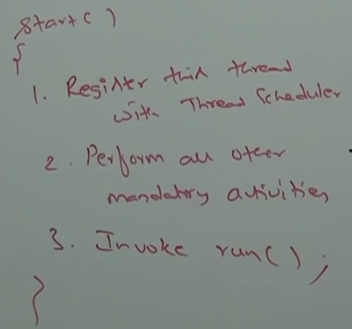
In the above program if we replace t.start with t.run then the output is :

Child Thread 10 times followed by Main Thread 10 times. This total output produced by only main thread.

**Case3:** Without executing thread class start method there is no chance of starting a new thread java.

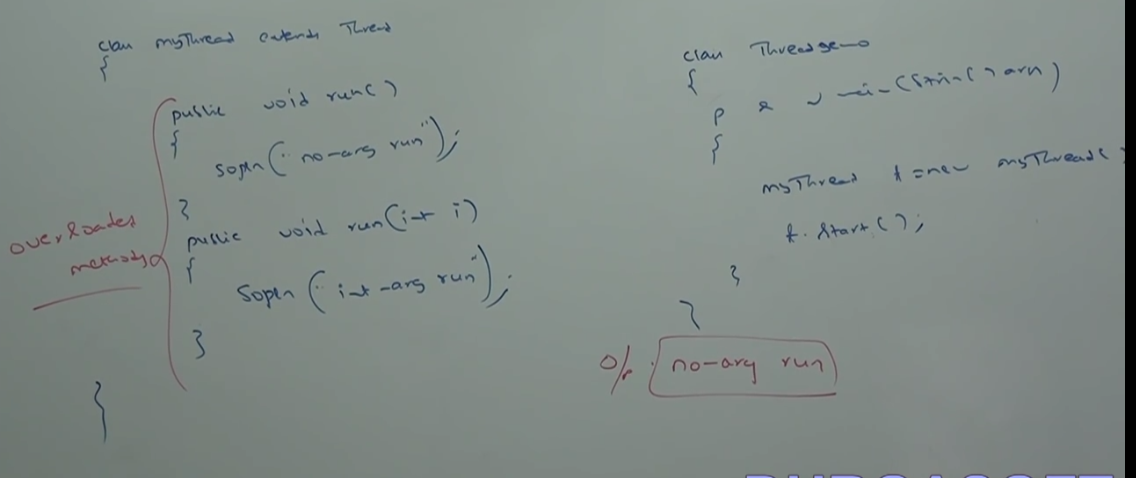
**Importance of Thread class start method:**

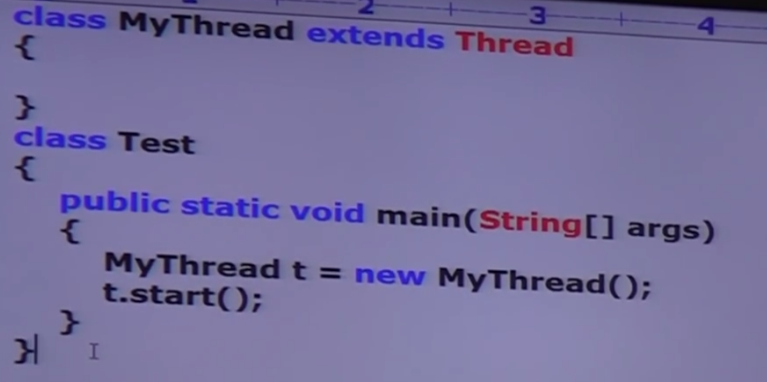
Thread class start method is responsible to register the thread with thread scheduler and all other mandatory activities. Hence, without executing thread class start method there is no chance of starting a new thread in java. Due to this, Thread class start method is considered as heart of multi-threading.



**Case4: Overloading of run method**

Overloading of run method is always possible. But thread class start method can invoke no argument run method. The other overloaded method we have to call explicitly like a normal method call.



**Case 5:** If we are not overriding run method then thread class run method will be executed which has empty implementation. Hence, we won’t get any output.

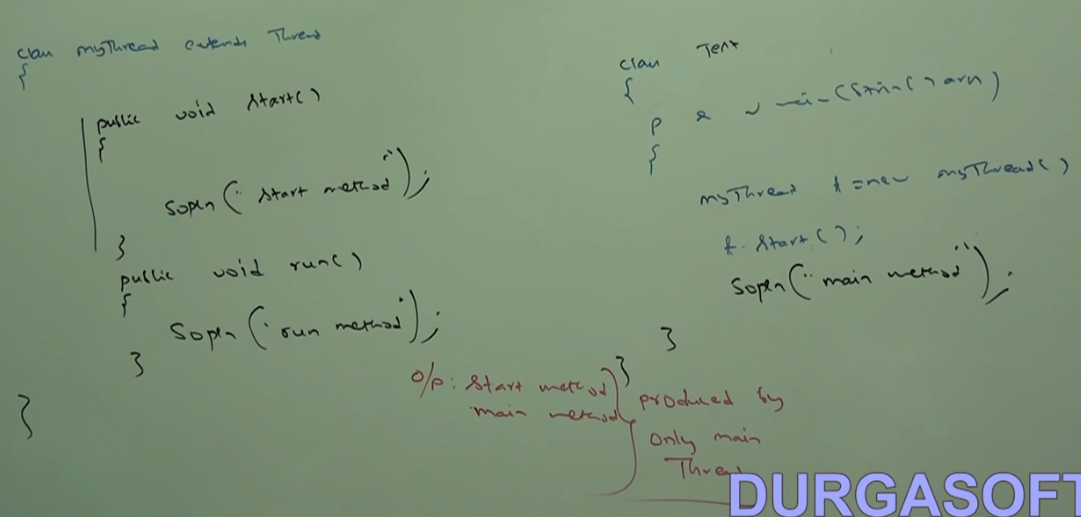
O/p: No output

It is highly recommended to override run method. Otherwise don’t go for multi-threading concept.

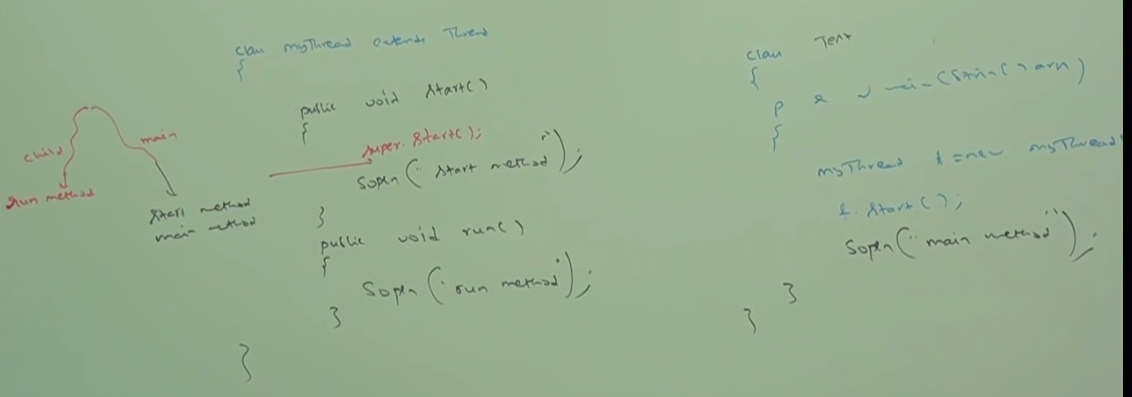
**Case 6: Overriding of start method**

Start method of child class will be executed just like normal method. No new thread will be created.

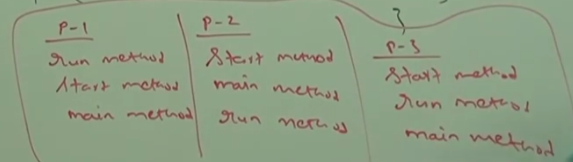
If we override start method then our start method will be executed just like a normal method call and new thread won’t be created.



It is not recommended to override start method. Otherwise don’t go for multi-threading concepts.

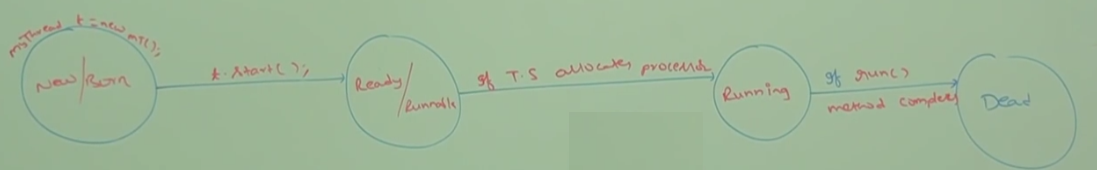


Possible output:



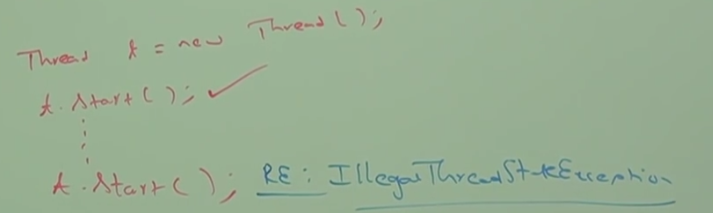
**Case 7: Thread Life Cycle**

Simple life cycle of Thread Life cycle

****

**Case 8:**

After starting a thread if we are trying to restart the same thread, then we will get run time exception saying IllegalThreadState exception**.**



**Defining a thread by Implementing Runnable Interface**