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--**

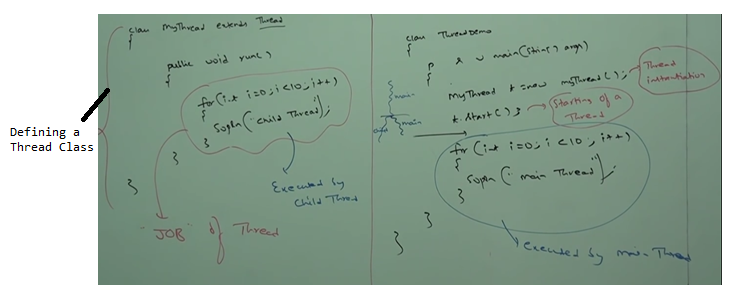
MyThread t = new MyThread ();🡪 **Thread Instantiation. Main thread creates a child thread.**

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

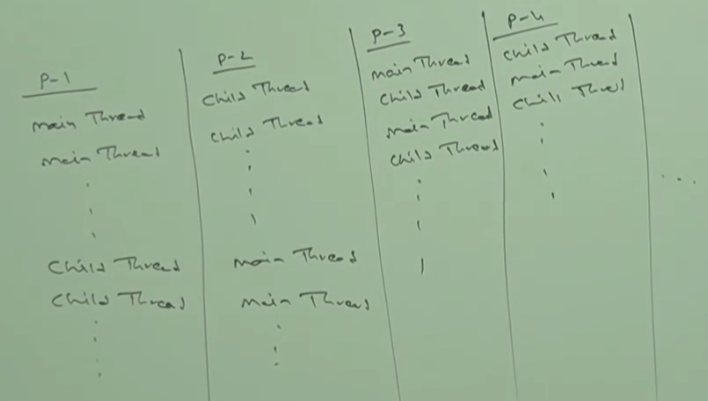
**--After t.start () In this point 2 thread---**

}

}



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.

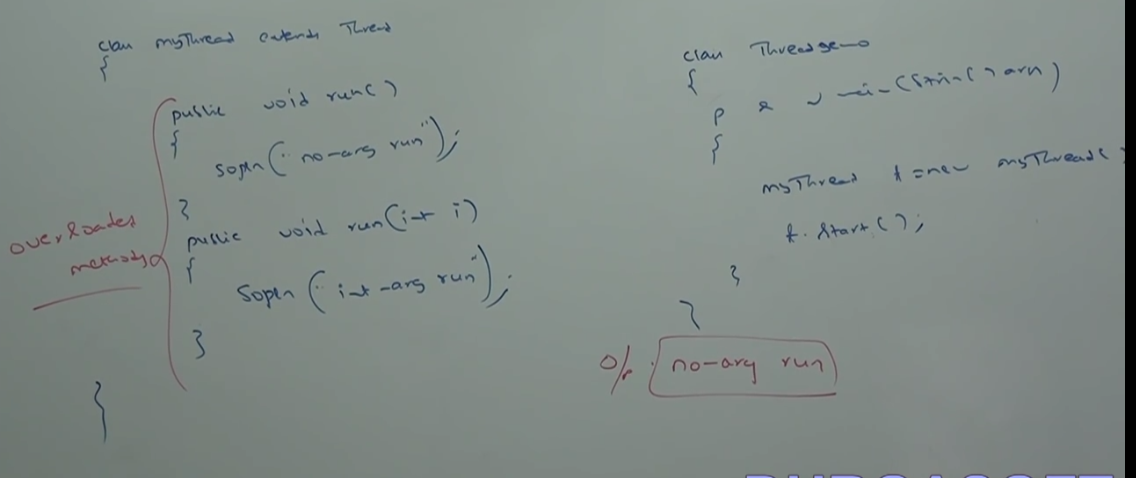
**Importance of Thread class start method:**

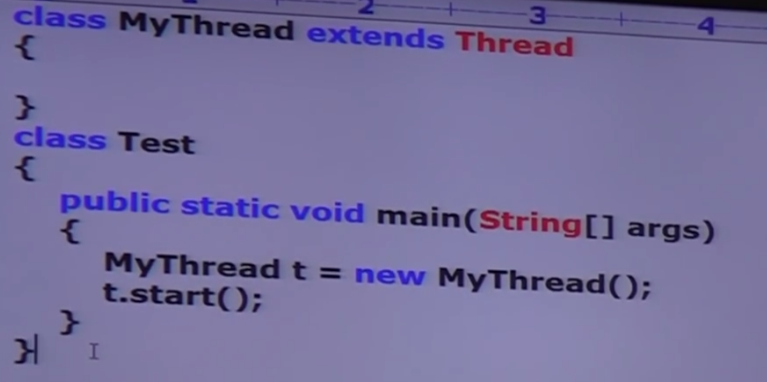
Thread class start method is responsible to register the thread with thread scheduler and all other mandatory activities. Hence, without executing thread class start method there is no chance of starting a new thread in java. Due to this, Thread class start method is considered as heart of multi-threading.



**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.

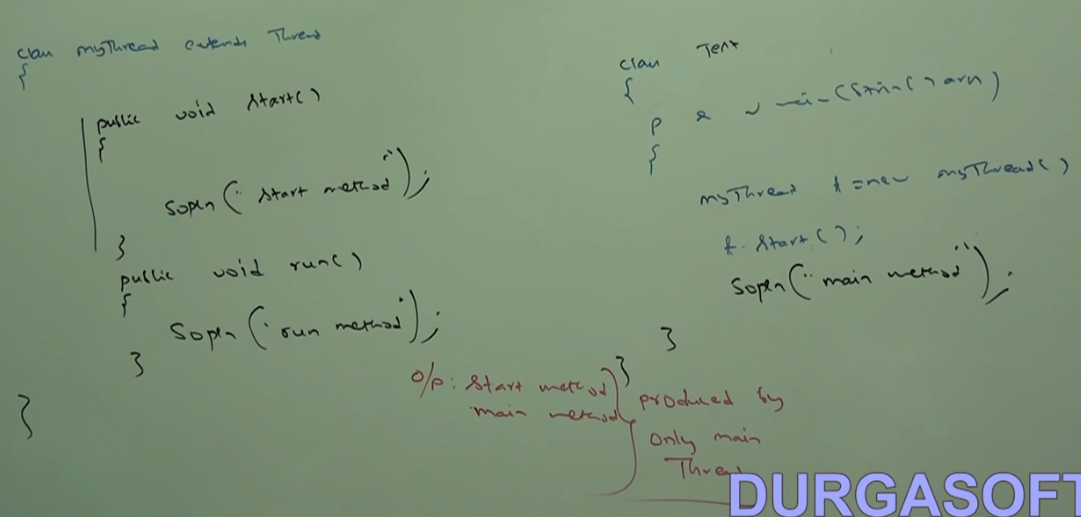
O/p: No output

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

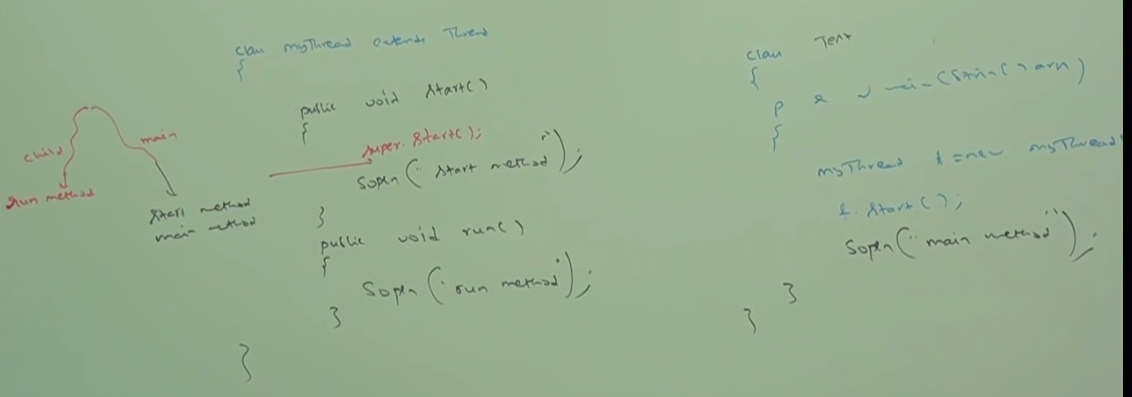
**Case 6: Overriding of start method**

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

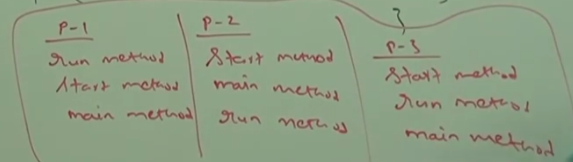
If we override start method then our start method will be executed just like a normal method call and new thread won’t be created.



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

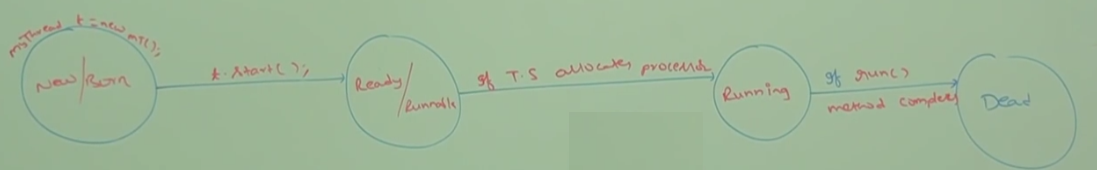


Possible output:



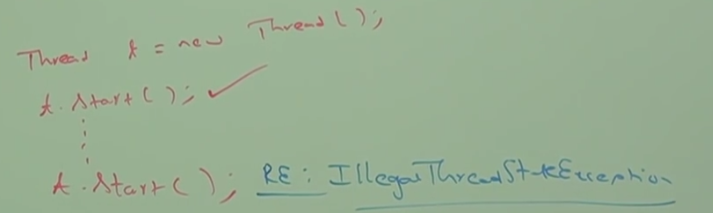
**Case 7: Thread Life Cycle**

Simple life cycle of Thread Life cycle

****

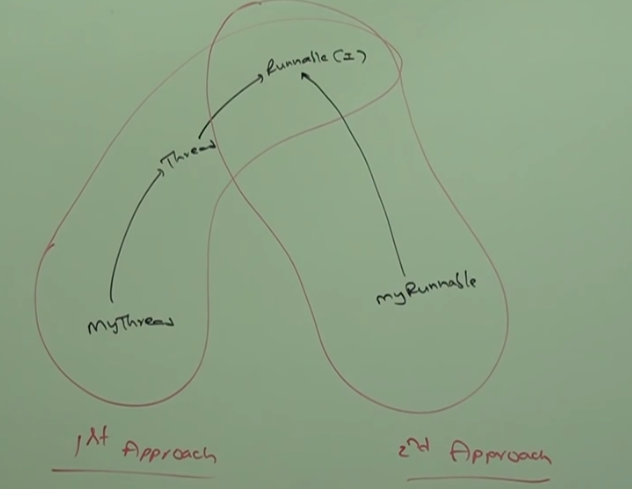
**Case 8:**

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



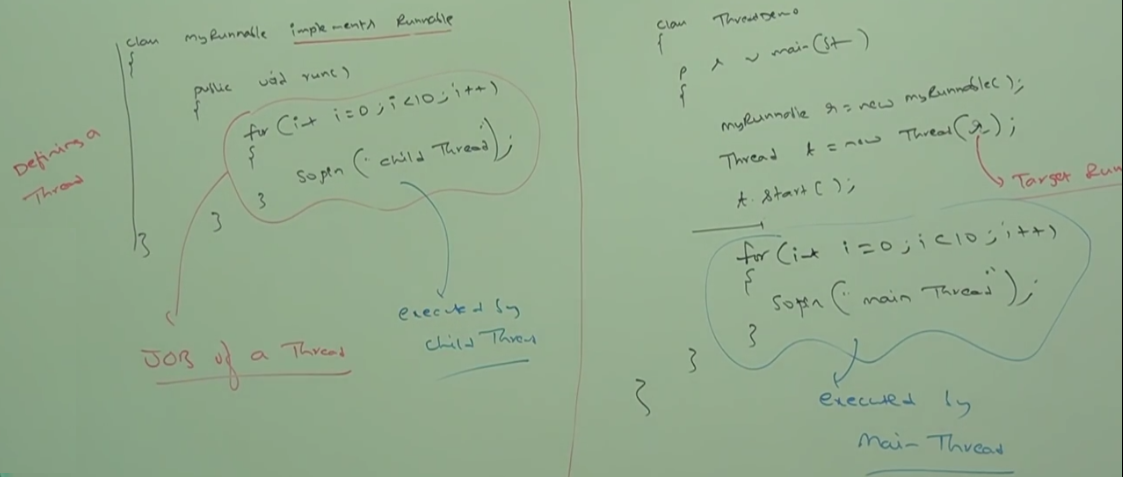
**Defining a thread by Implementing Runnable Interface:**

Thread by default implements Runnable interface. We can define a thread by implementing Runnable Interface.



Runnable Interface present in java.lang package and contains only 1 method i.e.

Public void run ();



MyRunnable does not have start method. So, we need to create a thread class and runnable instance to start the thread.

MyRunnable r = new MyRunnable();

Thread t1 = new Thread ();

Thread t2 = new Thread (r);

**Case1:** t1.start (); 🡪New Thread will be created which is responsible for the execution of thread class run method which has empty implementation.Nothing will be printed as thread run method does not have anything.

**Case2:** t1.run() 🡪 No new thread will be created and run method of Thread class will be executed just like normal method call.

**Note\*:** New thread will be created only when the start method is executed/called.

**Case3:** t2.start ()-> New Thread will be created which is responsible for the execution of MyRunnable class run method.Proper way of thread implementation.

**Case4:** t2.run (); ->t2 is created with MyRunnable class. So, MyRunnable run method will be executed just like a normal method call. No new thread will be created.

**Case5**: r.start (); 🡪we will get compile time error saying MyRunnable class does not have start capability. Runnable Interface does not have start method

Cannot find symbol: method start ()

Location class MyRunnable

**Case6**: r.run (); 🡪 No new thread will be created and MyRunnable run method will be executed just like normal method call.

**Which approach is best to define a thread.**

Among 2 ways of defining a thread, implements runnable approach is recommended.

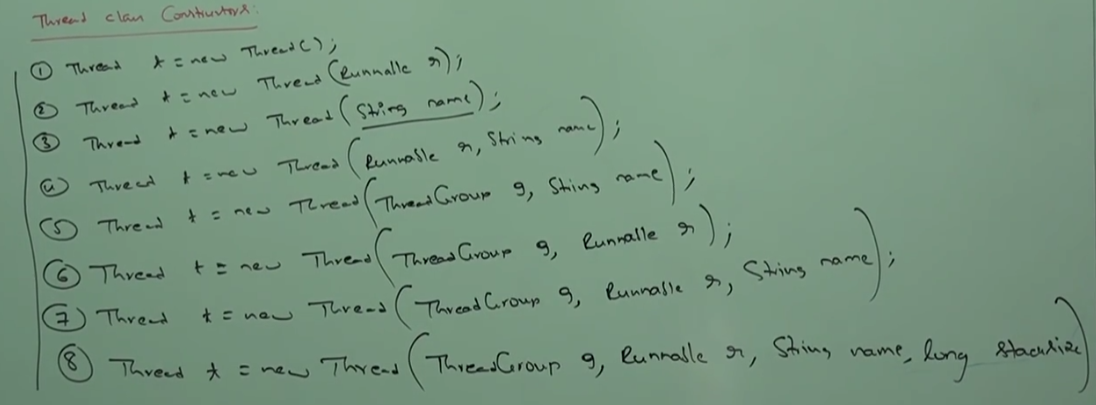
In the 1st approach, our class always extends thread class, there is no chance of extending any other class. Hence, we are missing inheritance benefit.

But in the 2nd approach, while implementing runnable interface we can extend any other class. Hence, we won’t miss any inheritance benefit.

Because of above reason implementing Runnable interface to create a thread is recommended then extending thread class.

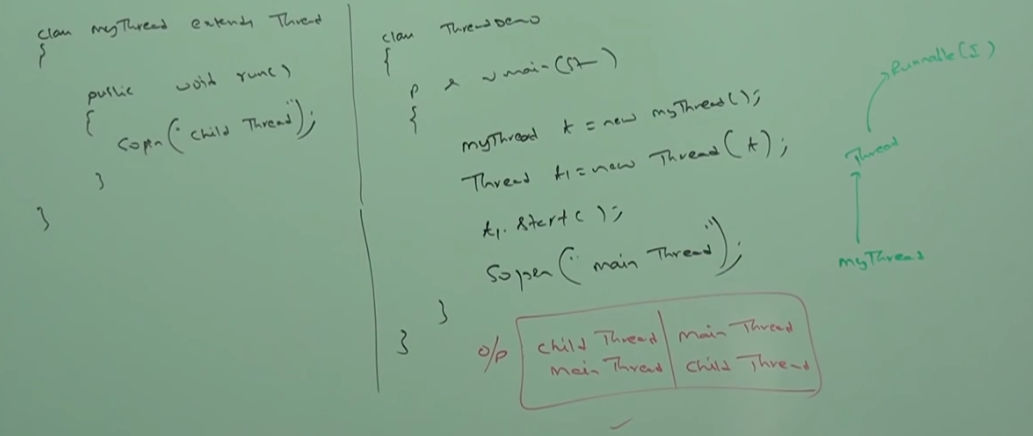
For every thread JVM creates a runnable stack.

**Thread class constructors:**



**3rd approach to create a thread:**

Instead of passing runnable instance to thread class constructor we can pass thread object itself which internally implements Runnable Interface.



**Getting and Setting name of a thread:**

Every thread in java has some name. It may be default name generated by JVM or customized name provided by programmer.

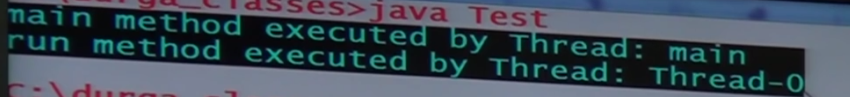
We can get and set name of a thread by using getName () and setName () methods of thread class.

Thread.currentThread().getName ->Currently executing thread name.

**Main thread is responsible to execute main method.**

We can get current executing thread object by using **Thread.currentThread** method.



o/p: 

**Thread Priorities:**

Every thread in java has some priority. It may be default priority generated by JVM or customized priority provided by programmer.

Valid range of thread priorities is 1:10. Where 1 is min priority and 10 is max priority.

Thread scheduler will use priorities while allocating processor. Thread which is having highest priority will get the chance first.

If 2 threads are having same priority then we can’t expect exact execution order. It depends on thread scheduler.

1-Min Priority—Thread.MIN\_PRIORITY 5-Thread.NORM\_PRIORITY

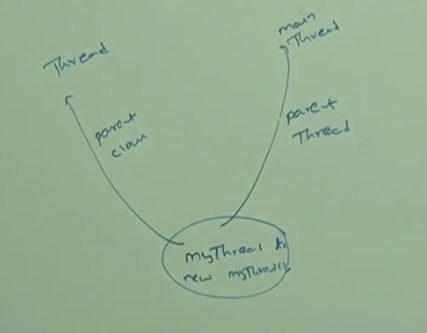
10- Max Priority—Thread.MAX\_PRIORITY

Public final int getPriority ()

Public final void setPriority (int p)—Allowed range 1 to 10. Else Illegal Argument exception.

**Default Priority:**

The default priority only for main thread is 5. For all remaining threads default priority will be inherited from parent to child i.e. whatever priority parent thread has the same priority will be there for the child thread.



Parent thread is the main thread which calls the child thread.

**Some of the Operating system won’t provide support for thread priority.**

Methods to prevent thread execution:

We can prevent a thread execution by using the following methods:

* + - 1. Yield
      2. Join
      3. Sleep

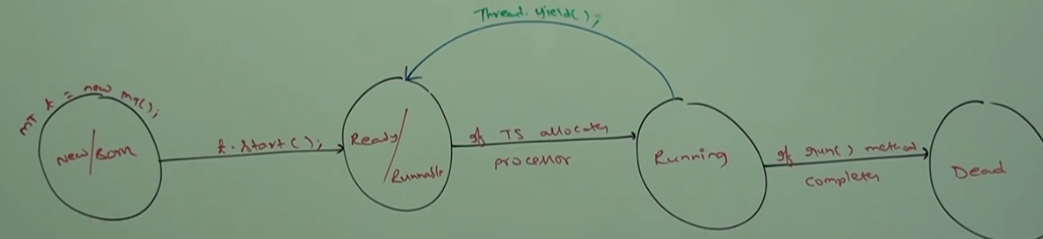
**Yield method:** Yield method causes to pause the current executing thread to give the chance to waiting threads of same priority. If there is no waiting thread or all waiting threads have low priority then same thread can continue its execution.

If multiple threads are waiting with same priority then which waiting thread will get the chance we can’t expect. It depends on thread scheduler.

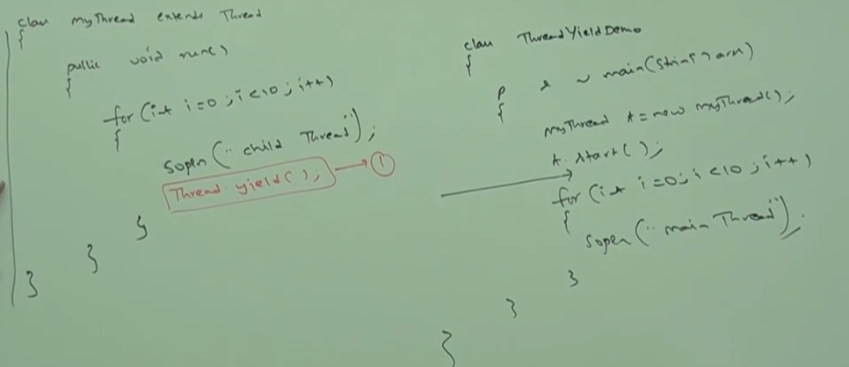
The thread which is yielded, when it will get the chance once again –It depends on Thread Scheduler. We can’t expect exactly.

**Public static native void yield ()**

Native methods are not implemented in java



Thread.yield () 🡪Pause the current executing thread and give chance to the waiting thread of same priority.



In the above program commenting the line “Thread.yield()”, then both thread will be executed simultaneously and we can’t expect which thread will complete first. If we are not commenting line 1 then child thread always calls yield method. Because of that main thread will get chance more number of times and the chance of completing main thread first is high.

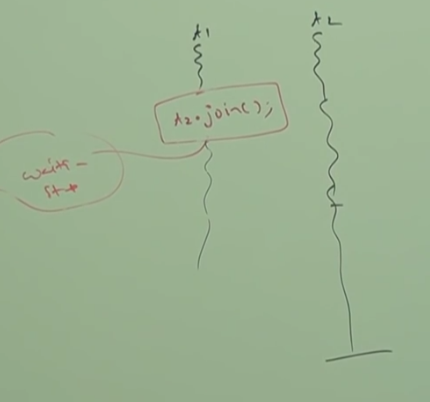
**Some platforms won’t provide proper support for yield method.**

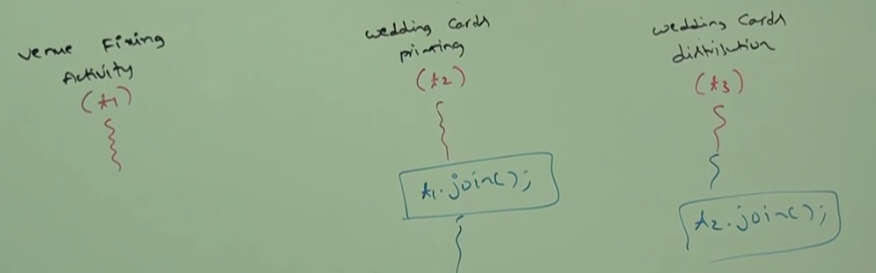
**Join method:**

If a thread wants to wait until completion of another thread then it will call the join method.

Ex1: If a thread t1 wants to wait until completion of t2 thread then t1 will have to call the join method on t2 thread i.e. t1 will call t2. Join (). If t1 executes “t2.join” then immediately t1 will be entered into waiting state until t2 completes.

Once t2 completes then t1 can continue its execution.



**Ex2:** 

In the above diagram t2 must wait until t1 completion. Hence t2 has to call t1.join.

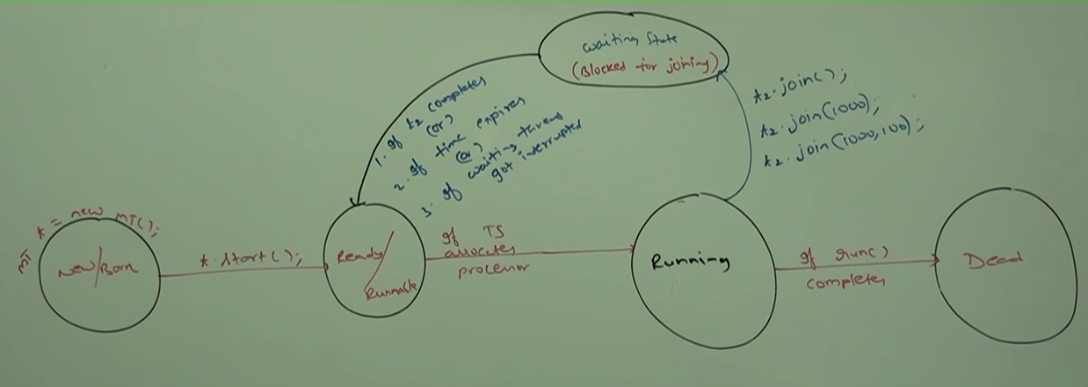
t3 must wait until t2 completion. So, t3 has to call t2.join

public final void join () throws Interrupted Exception;

public final void join (long milliseconds) throws IE;

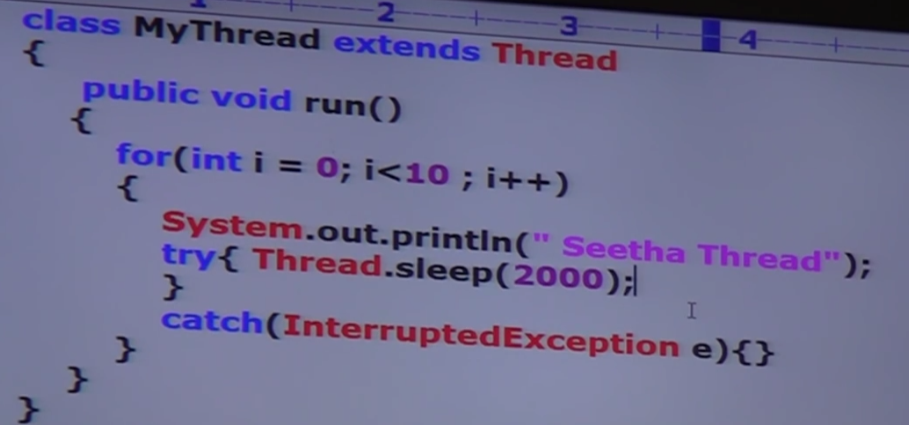
public final void join (long milliseconds, int nanoseconds) throws IE;

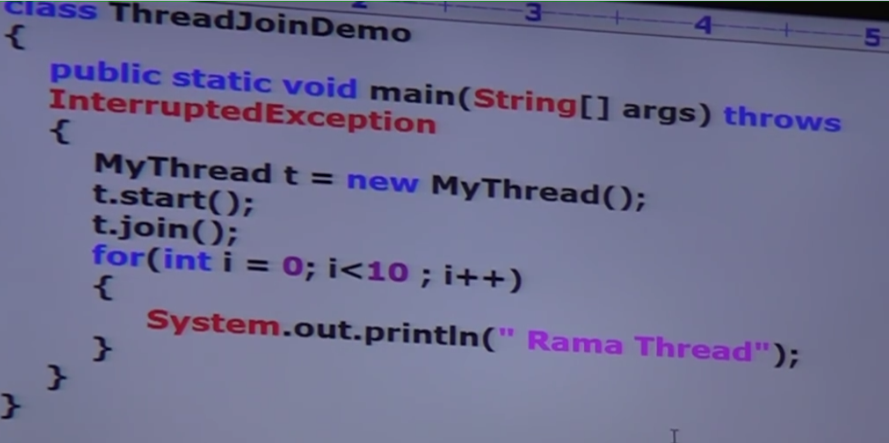
Every join method throws Interrupted exception which is checked exception. When a thread is waiting for another thread for completion then a third thread may interrupt the waiting thread.



Impact of join methods in thread life cycle.

Waiting of main thread until completing child thread:





If we comment “t.join” then both main and child thread will be executed simultaneously and we can’t expect exact output.

If we are not commenting line 1 then main thread calls join method on child thread object. Hence, main thread will wait until completing the child thread. In this case

output: “Seetha Thread” 10 times followed by “Rama Thread”