

Eg: Alternative\_Approach\_For\_Thread\_Creation

// go through the code

* It is possible to change the name of the thread using setName()
* It is possible to get the name of the thread using getName()

Methods:

public final void setName(String name)

public final String getName()

Eg: Modify\_Name\_Of\_Thread

// go through the code

Eg: Modify\_Name\_Of\_Thread\_Eg2

// go through the code

Note: we can set the same name to multiple threads, but not recommended.

Thread priorities:

* For every thread in java the priority is same
* Valid range of priority is 1 to 10, not from 0 to 10
* It we try to give a different value it would result in “IllegalArgumentException”
* Thread.MIN\_PRIORITY = 1
* Thread.MAX\_PRIORITY = 10
* Thread.NORM\_PRIORITY = 5
* Thread class does not have priorities like Thread.LOW\_PRIORITY, Thread.HIGH\_PRIORITY
* Thread scheduler allocates cpu time based on “priority”.
* If both threads has same priority then which thread will be given chance can’t be predicted, and it is completely vendor dependent
* We can set and get priority values of the thread using the following methods

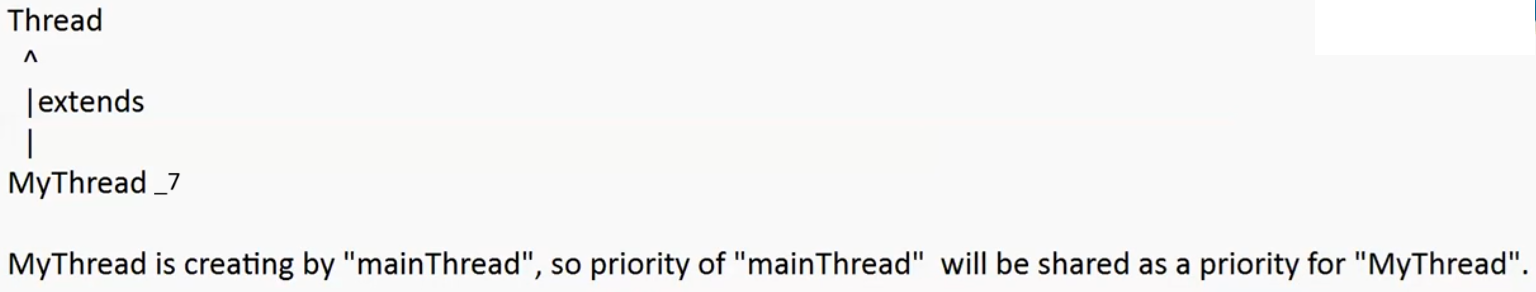
1. public final void setPriority(int PriorityNumber)
2. public final int getPriority()

Default priority:

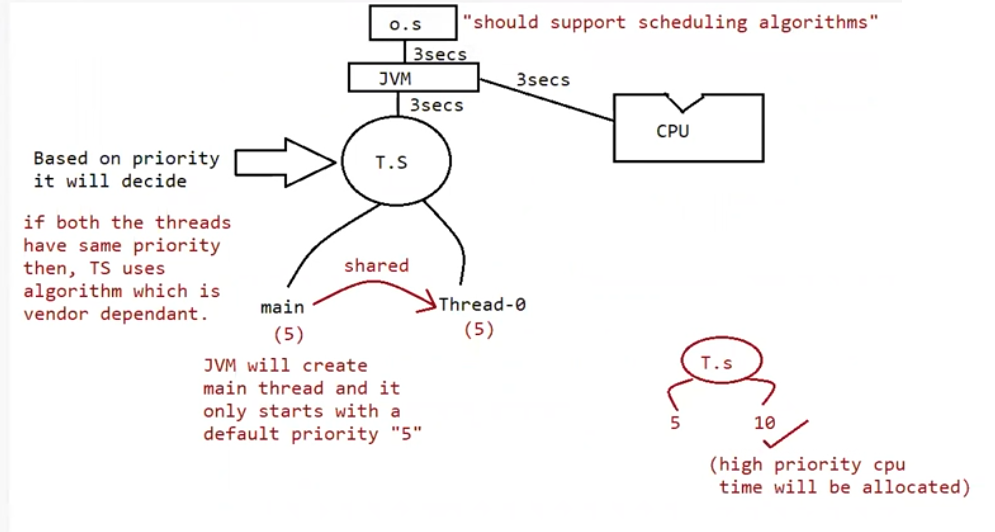
The default priority for main thread is 5, whereas for other threads priority will be inherited from parent to child.

Parent thread priority will be given as child thread priority

Eg: Default\_Priority



Eg: Thread\_Priority



yield()

it causes to pause current executing thread, for giving a chance for waiting threads of same priority

if there are no waiting threads (or) all the waiting threads have low priority then same thread can continue its execution.

If all threads have same priority and if they are waiting, then which thread will get a chance we can’t except, it depends on thread scheduler.

The thread which is yielded, when it will get chance once again depends upon the thread scheduler. We can’t except exactly.

public static native void yield()

Eg: Yeild\_Method

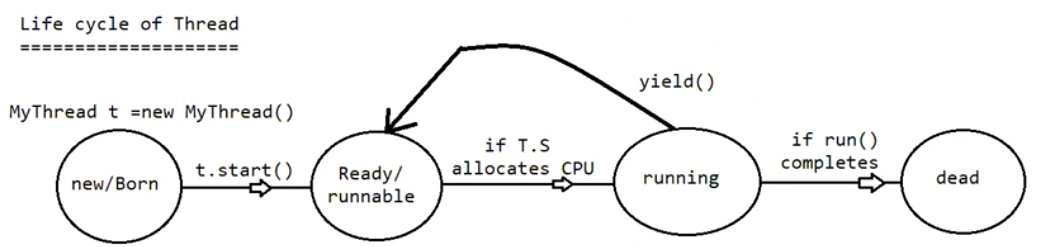
Mythread t = new Mythread(); // new state (or) born state

t.start() // enter into ready state (or) runnable state

if thread scheduler allocates processor then enters into running state

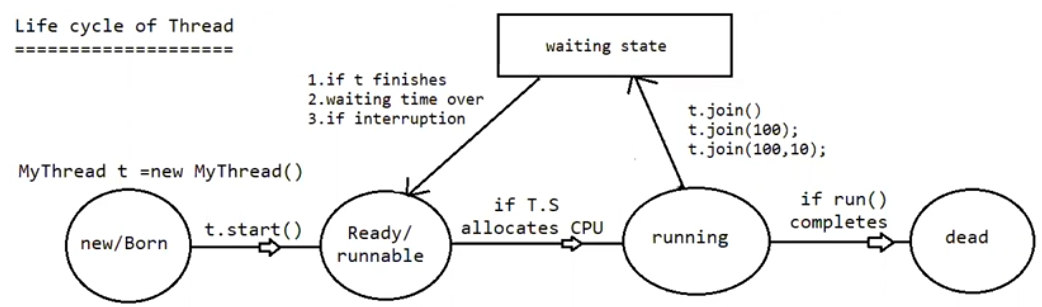
1. if running thread calls yield () then it enters into runnable state

if run() is finished with its execution then it enters into dead state



Note: some platforms won’t provide proper support for yield(), because it may be getting execution code from other language ( may be from c ) .

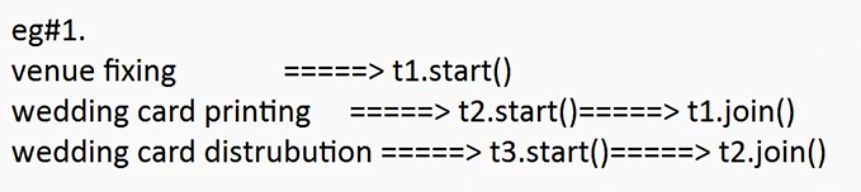
join()

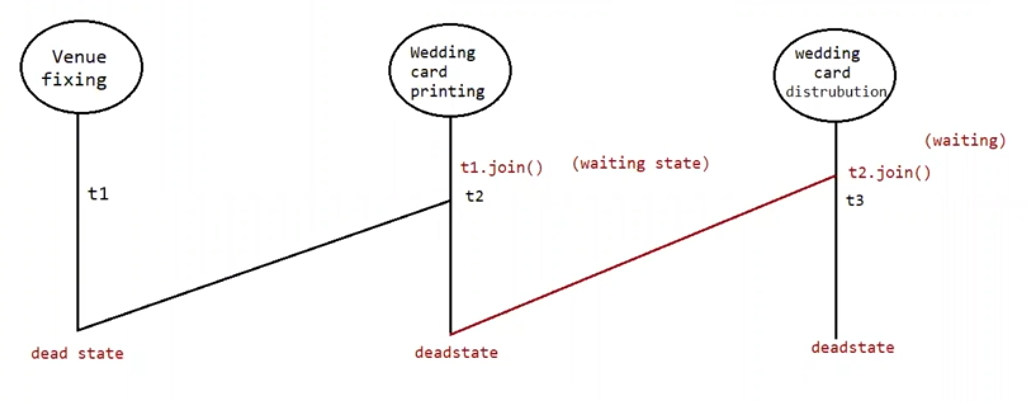


if the thread has to wait until the other thread finishes its execution then we need to go for join() method

if t1 executes t2.join then t1 should wait until t2 finishes its execution

t1 will be entered into waiting state until t2 complete its execution, once t2 completes then t1 can continue with its execution.





Prototype of join:

public final void join() throws InterruptedException

public final void join (long ms ) throws InterruptedException

public final void join (long ms, int ns) throws InterruptedException

ms -> milli-seconds

ns -> nano-seconds

Note: one thread is in waiting state and if one more thread interrupt’s then it would result in “InterruptedException”. Interrupted Exception is a checked exception which should always be handled.

Thread t = new Thread(); // new /born state

t.start() //ready / runnable state

If Thread Scheduler allocates cpu time then thread enters into running state

If currently executing thread invokes t.join / t.join(1000) , t.join(1000,100) then it would enter into waiting state

If thread finishes the execution / time expires / interrupted then it would comeback to ready state / runnable state

If run() is completed then it would enter into dead state

Eg: Join\_Method\_Eg1

// go through the code

Waiting of child thread until completing main thread

We can make child thread to wait for main thread.

Eg: Child\_Thread\_Waiting\_For\_Main\_Thread

// go through the code

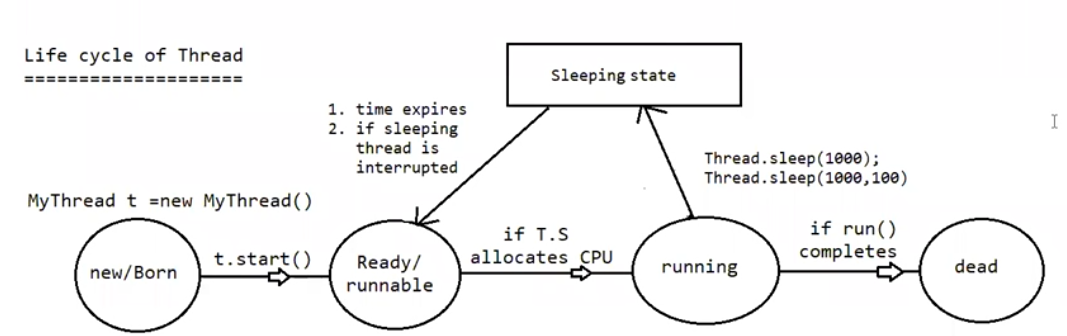
Similarly we can make main thread to wait for child thread as above example Eg: Join\_Method\_Eg1

Note: if both the threads invoke t.join and mt.join it would result in deadlock.

Eg: Main\_Thread\_Waiting\_For\_Main\_Thread

// Deadlock because main thread is waiting for main thread

sleep() :



if a thread don’t want to perform any operation for a particular amount of time then we should go for sleep() method.

Signature:

public static native void sleep(long ms) throws InterruptedException

public static void sleep(long ms , int ns) throws InterruptedException

every sleep method throws InterruptedException which is a checked exception, so we should compulsorily handle the exception using try-catch (or) throws keyword otherwise it would result in compile time error.

Thread t = new Thread(); // new / born state

t.start() // ready / runnable state

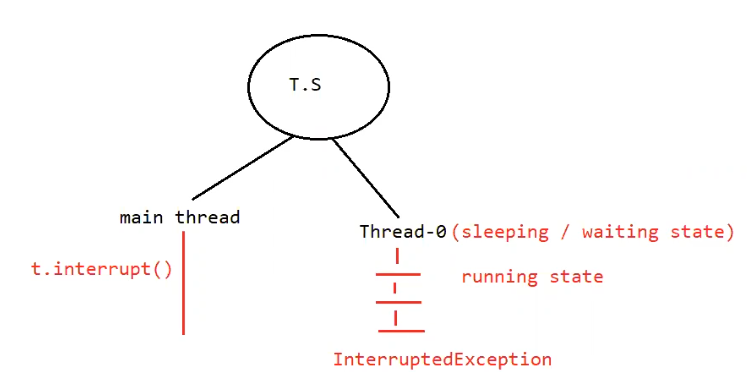
If thread scheduler allocates cpu time then it would enter into running state

If run() completes then it would enter into dead state

If running thread invokes sleep(1000) / sleep(1000,100) then it would enter into sleeping state

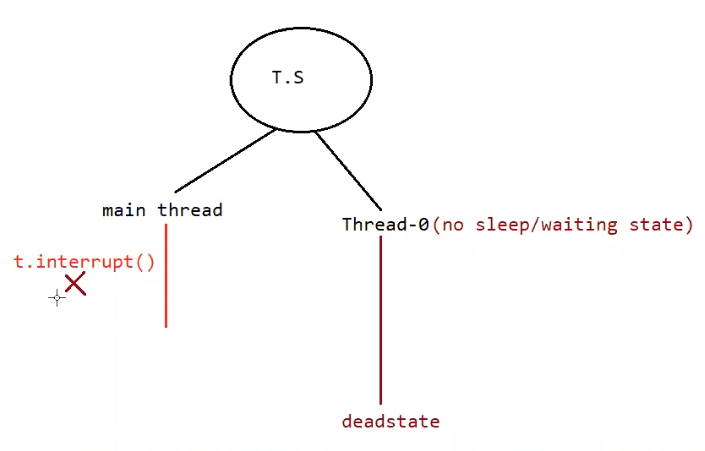
Interrupting a thread:

If a thread is in waiting state (or) sleeping state we can interrupt a thread

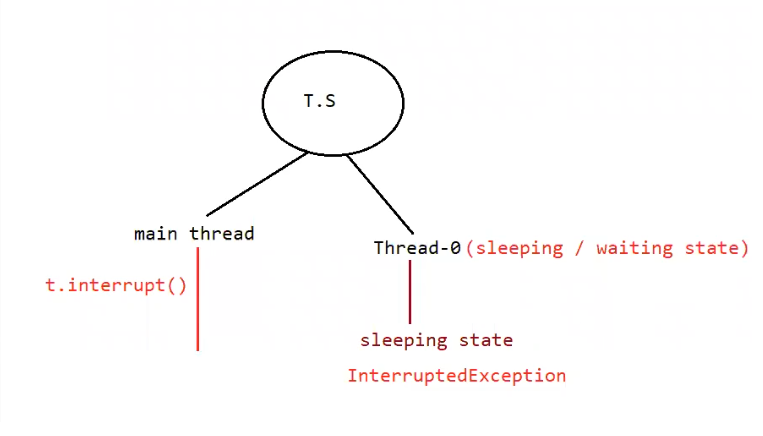


Eg: Interrupting\_Thread\_with\_Sleeping\_Or\_Waiting

Eg: Interrupting\_Thread\_Without\_Waiting\_Or\_Sleeping



Eg: Interrupting\_Thread\_Waiting\_After\_Execution



Note:

* if thread is interrupting another thread , but the target thread is not in waiting/sleeping state then there would be no exception
* interrupt() call will be waiting till the target thread enters into waiting state/ sleeping state, so this call cant be wasted.
* Once the target thread enters into waiting state / sleeping state then interrupt() will interrupt and causes its interruption
* Interrupt call be wasted only if the thread does not enters into waiting / sleeping state.

1. Purpose

yield() : to pause the current executing thread and give a chance to the remaining waiting threads of same priority

join(): if a threads needs to wait until some other thread finishes its execution then we should use join() method.

sleep(): if a thread don’t want to perform any operation for a particular amount of time then we should go for sleep() method.

1. Is it static ?

yield() : yes

join() : no

sleep() : yes

1. Is it final?

yield(): no

join():yes

sleep():no

1. Is it overloaded?

yield(): no

join(): yes

sleep():yes

1. Does it throws InterruptedException?

yield(): no

join() : yes

sleep() : yes

1. Is it native?

yield() : yes

join() : no

sleep()

sleep( long ms ) : native

sleep( long ms , int ns ) : non-native

Thread creation using lambda Expression:

Eg: Thread\_Using\_Lambda\_Expression

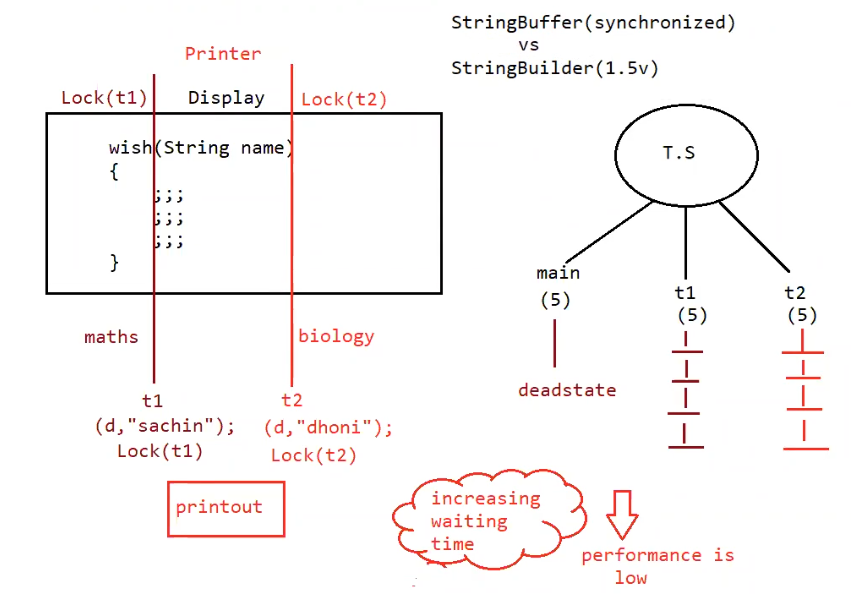
// go through the class

Thread creation using anonymous inner class

Eg: Thread\_Using\_Anonymous\_Inner\_Class

Synchronization:

* Synchronized is a keyword applied for methods and blocks
* If we declare method/block as synchronized then at a time only one thread can execute that method/block on that object
* The main advantage of synchronized keyword is we can resolve data inconsistency problems
* But the main disadvantages of the synchronized keyword is it increases the waiting time of the thread and effects performance of the system
* Hence if there is no specific requirement then never recommended to use synchronized keyword
* Internally synchronization is implemented by using lock concept.



Eg: Problem\_Without\_Synchronization

Threads t and t1 are executed simultaneously , so we cant predict the output. Here the main problem is data inconsistency .

Eg: Thread\_With\_Synchronization\_On\_Same\_Object

Here threads t and t1 will not execute simultaneously , instead if t thread is running , t1 thread will not interrupt until the t thread completes its execution. After that t1 will execute