Multithreading

Topics

- Introduction
- The ways to define a thread
  1. by extending Thread class
  2. By implementing Runnable (I)
- Getting & setting name of Thread
- Thread priorities
- Methods to prevent thread execution
  1. Yeild
  2. Join
  3. sleep

- Synchronization
- Inter thread communication ← wait, notify, notifyall
- Deadlock
- Deamon Threads
- Multithreading enhancements

Introduction

Executing several tasks simultaneously is the concept of multitasking

There are 2 types of multitasking

1. Process Based multitasking
2. Thread Based multitasking

Process based multitasking

Executing several task simultaneously where each task is a separate independent program (process) is called process based multitasking.

eg while typing a java program in the editor, we can listen audio songs from the same system, at same time we can download a file from net. All these tasks will be executed simultaneously & independent of each other. Hence it is process based multitasking.

Thread-based multitasking

Process based multitasking is best suitable at OS level.

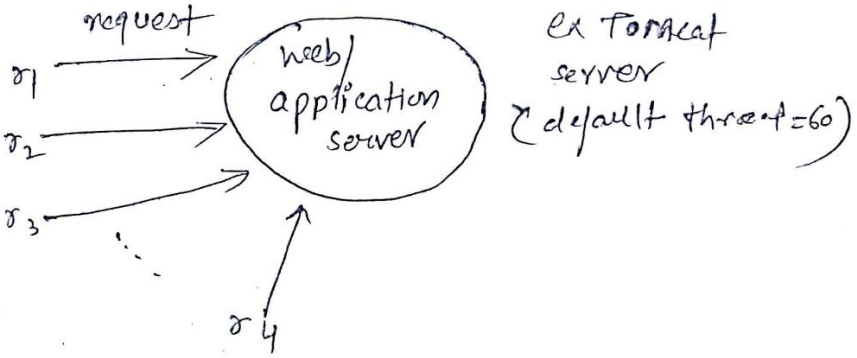
2. Thread based multitasking

Executing several tasks simultaneously where each task is a separate independent part of the same program is called thread-based multitasking & each independent part is called a thread. Best suitable at programming level.

Whether it is process based or thread based, the main objective of multitasking is to reduce response time of the system & to improve performance.

The main important application areas of multithreading are :-

1) to develop multimedia graphics
2) to develop animations
3) to develop videogames
4) to develop web servers & application servers etc.



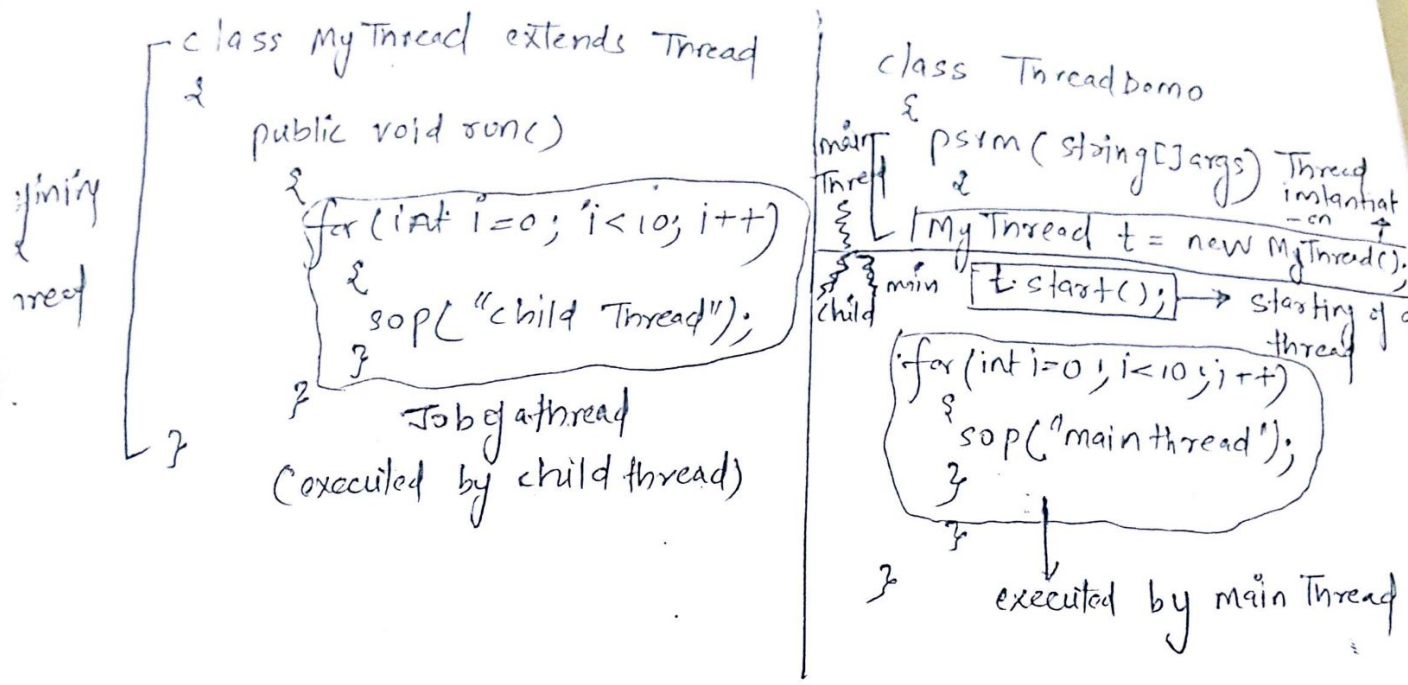
When compared with old languages, developing multithreaded applications in java is very easy because java provides inbuilt support for multithreading with rich API (Thread, Runnable, Thread group etc.)

Defining a Thread

We can define a thread in 2 ways :-

1) By extending Thread class
2) By implementing Runnable interface.

internally calls run() method of MyThread class.



output ⇒ mixed output (no dependency)
<any order threads can follow>

Case1 Thread schedular

1) It is the part of JVM
2) It is responsible to schedule threads i.e if multiple threads are waiting to get chance of execution then in which order threads will be executed is decided by Thread schedular.
3) We can't expect exact algorithm followed by thread schedular, it is varied from JVM to JVM. Hence we can't expect threads execution order & exact output.

Hence whenever situation comes to multithreading, there is no guarantee for exact output but we can provide several possible outputs

Possible outputs :-

1)
main Thread
main Thread
:
10 times
child Thread
child Thread
:
10 times

2)
child Thread
child Thread
:
10 times
main Thread
main Thread
:
10 times

3)
main Thread
child Thread
main Thread
:
:

4)
child
Main
child Thr
:
:

** Case 2 Difference between t.start() & t.run();

↓ Thread class method

```
class MyThread extends Thread
{
public void run()
  {
    sop (for(int i=0;i<10;i++)
    {
      sop("child Thread");
    }
  }
}
```

```
class ThreadDemo
{
psvm (String[] args)
{
  MyThread t = new MyThread();
    t.run();
for (int i=0; i<10; i++)
  { sop("main Thread");
  }
  }
}
```

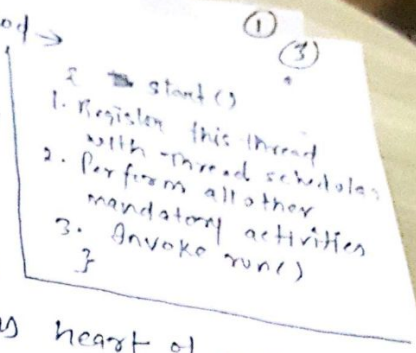
Child Thread
child Thread
:10 times
main Thred
:10 times

In the case of t.start(), a new thread will be created which is responsible for the execution of run() method

But, in the case of t.run(), a new thread wont be created & run() method will be executed just like a normal method call. by main thread. Hence in the above program, output is fixed child Thread (10 times) followed by main Thread (10 times) This total output produced by only main Thread

Importance of Thread class start() method →

Thread class start method is responsible to perform register the thread with Thread schedular & 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 multithreading



Case 4:

```
class MyThread extends Thread
{
    public void run()
    {
        sop("no-arg run");
    }
    public void run(int i)
    {
        sop("int-arg run");
    }
}
```

Overloaded methods

```
class ThreadDemo
{
    psvm(String[] args)
    {
        MyThread t = new MyThread();
        t.start();
    }
}
```

O/p → no-arg run

Case 4:-

Overloading of run() method →

Overloading of run method is possible but Thread class start method can invoke no-arg run method. The other overloaded method has to be called explicitly like a normal method call.

Case 5:- If we are not overriding run method :-

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.

```
class MyThread extends Thread
{
    // empty implementation
}
class Test
{
  psvm (String[] args)
  {
    MyThread t = new MyThread();
      t.start();
  }
}
```

output → no output

It is highly recommended to override run() method of Thread class otherwise do not go for multithreading concept.

case6:- Overriding of start() method :-

```
class MyThread extends Thread
{
  public void start()
  {
    sop ("start method");
  }
  public void run()
  {
    sop ("run method");
  }
}
```

```
class Thread
{
  psvm (String[] args)
  {
    MyThread t = new MyThread();
      t.start();
    sop ("main Thread");
  }
}
```

output → start method
main method

If we override start method then our start() method will be executed just like a normal method call & new thread won't be created. It is not recommended to override start method. otherwise do not go for multithreading concept.

Calling start method of Thread classing using super keyword & observing the possible outputs :-

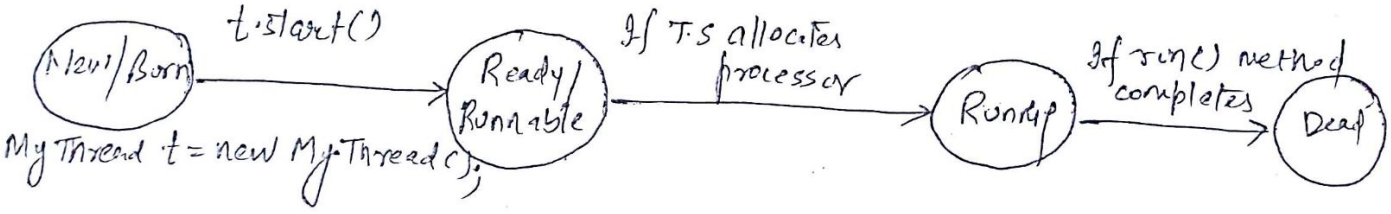
```
class MyThread extends Thread
{
public void start()
  {
    super.start();
    sop("start method");
  }
public void run()
  {
    sop("run method");
  }
```

```
class Test
{
psvm(String[] args)
 {
 MyThread t = new MyThread();
   t.start();
   sop("main method");
 }
}
```

JVM
- run method
- main
  - start method
  - main method

P1	P2	P3
run method start method main method	start method run method main method	start method run method main method

Case 8:- Thread lifecycle :-

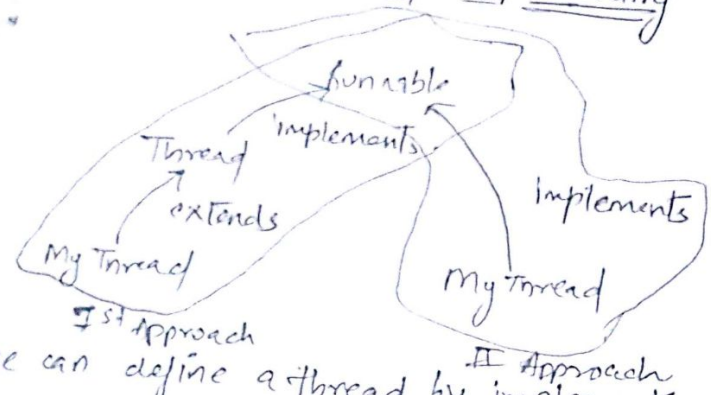


Case 9:-

After starting a thread, if we are trying to restart the same thread then we will get RuntimeException :- IllegalThreadStateException

```
Thread t = new Thread();
t.start();
:
t.start(); // IllegalThreadStateException
```

Defining a thread by implementing Runnable interface



We can define a thread by implementing Runnable interface

Runnable interface present in java.lang package & it contains only one method (run()) → public void run

Defining a thread

```
class MyThread implements Runnable
{
  public void run()
  {
    for(int i=0; i<10; i++)
    {
      sop("child Thread");
    }
  }
}
```

→ Job of a Thread executed by child Thread

```
class Test                                  → main thread
{
  psvm(String[] args)
  {
    MyThread t = new MyThread();   // main threads creates new MyThread object
    // Thread t1 = new Thread();
    //         t1.start();          // calls Thread cls. class run method (empty implementation
    Thread t1 = new Thread(t);     or case Thread(new
    t1.start();                    "target Runnable>.

    for(int i=0; i<10; i++)
    {
      sop("main thread");
    }
  }
}
```

→ executed by main Thread

will get mixed output & we can't tell exact output.

Case study :-

```
MyThread t = new MyThread();
Thread t1 = new Thread();
Thread t2 = new Thread(t);
```

Case-1 t1.start()

A new thread will be created which is responsible for the execution of Thread class run method, which has empty implementation.

Case 2 t1.run()

No new thread will be created & Thread class run method will be executed just like a normal method call.

Case 3 t2.start()

A new thread will be created which is responsible for the execution of MyThread class run method.

Case 4 t2.run()

A new thread won't be created & MyThread run method will be executed just like a normal method call.

Case 5 t.start()

We will get compile time error saying → MyThread class doesnot have start capability.

CE → cannot find symbol
symbol : method start()
location : class MyThread

Case 6 t.run()

No new thread will be created & MyThread run method will be executed like a normal method call.

Which Approach is best to define a Thread?

1)
```
class MyThread extends Thread
{
public void run()
{
}
}
```

2)
```
class MyThread implements Runnable
{
public void run()
{
}
}
```

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

In the first approach, our class always extends Thread class, there is no chance of extending any other class. Hence we are missing inheritance benifit.

But in the second approach while implementing Runnable interface we can extend any other class. Hence we won't miss any inheritance benifit.

Thread class Constructors

1) Thread t = new Thread();
2) Thread t = new Thread(Runnable r);
3) Thread t = new Thread(Runnable r, String name);
4) Thread t = new Thread(String name);
5) Thread t = new Thread(ThreadGroup g, String name);
6) Thread t = new Thread(ThreadGroup g, Runnable r);
7) Thread t = new Thread(ThreadGroup g, Runnable r, String name);
8) Thread t = new Thread(ThreadGroup g, Runnable r, String name, long stackSize);

Hybrid approach to create a Thread →

```
class MyThread extends Thread
{
public void run()
{
sop("child thread");
}
}
```

```
class Test
{
psvm(String[] args)
{
MyThread t = new MyThread();
Thread t1 = new Thread(t);
t1.start();
sop("main thread");
}
```

Runnable (I)
↑
Thread
↑
MyThread

o/p → main thread
child thread | child thread
main thread .

Getting & setting name of a Thread

ex

```
class MyThread extends Thread
{
}
class Test
{
  psvm (String[] args)
  {
    sop (Thread.currentThread().getName());   // main
    MyThread t = new MyThread();
    sop(t.getName());   // Thread-0
    Thread.currentThread().setName("java");
    sop (Thread.currentThread().getName())   // java
  }
}
```

Every thread in java has some name. It may be default name generated by JVM or customized name provided by programmer.
We can get & set name of a thread using getName() & setName() methods of Thread class respectively.

Public final String getName();
public final void setName (String name);

Note → public static native Thread currentThread();
syntax of currentThread() :-

eg:

```
class MyThread extends Thread
{
  public void run()
  {
    sop("This line is executed by:" + Thread.currentThread().getName());
  }
}
class Test
{
  psvm (String[] args)
  {
    MyThread t = new MyThread();
    t.start();
    sop("This line is executed by Thread:" + Thread.currentThread().
  }
}
```

// Thread-0 ();

// main

Note→ We can get current executing Thread object by using Thread.currentThread() method (static method)

Thread Priorities

- Every thread in Java has some priority. It may be default priority generated by JVM or customized priority provided by programmer.
- The valid range of Thread priorities is 1 to 10 where 1 is min priority & 10 is max priority

Thread class defines the following constants to represent some standard priorities:-

Thread.MAX_PRIORITY	→ 10	✓ Valid
Thread.MIN_PRIORITY	→ 1	✓ Valid
Thread.NORM_PRIORITY	→ 5	✓ Valid
Thread.HIGH_PRIORITY		✗ Invalid
Thread.LOW_PRIORITY		✗ Invalid
0	✗	Invalid
1	✓	
10		

Thread scheduler will use priorities while allocating processor. The thread which is having highest priority will get chance first. If two threads having same priority, then we can't expect exact execution order. It depends on thread schedular.

Public final int getPriority()

public final void setPriority(int p)

↓ allowed values range (1 to 10)

otherwise RE: IllegalArgumentException

ex: t.setPriority(16);

Default Priority

The default priority only for the main thread is 5. but 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 child thread.

ex:

```
class MyThread extends Thread
{
}
class Test
{
    psvm(String[] args)
    {
        sop(Thread.currentThread().getPriority()); // 5
        Thread.currentThread().setPriority(7);
        MyThread t = new MyThread(); // main thread creates child (main is parent thread)
        sop(t.getPriority()); // 7
    }
}
```

read main

rent Parent

lass Thread

ythread t = new MyThread

```
class MyThread extends Thread
{
  public void run()
  {
    for (int i=0; i<10; i++)
    {
      sop("child Thread");
    }
  }
}
```

```
class Test
{
  psvm (String[] args)
  {
    MyThread t = new MyThread();
    t.setPriority(10); ——— (1)
    t.start();
    for(int i=0; i<10; i++)
    {
      sop("main thread");
    }
  }
}
```

Note

If we are commenting line (1) then both main & child threads have the same priority 5 & hence we cant expect execution order & exact output.

If we are not commenting line (1) then main thread has a priority 5 & child thread has priority 10 Hence child thread will get the chance first followed by main thread. In this case output is

```
child Thread
:
10 times
main thread
:
5 times
```

⇒ Some platforms wont provide proper support for thread priorities

Preventing Thread from execution

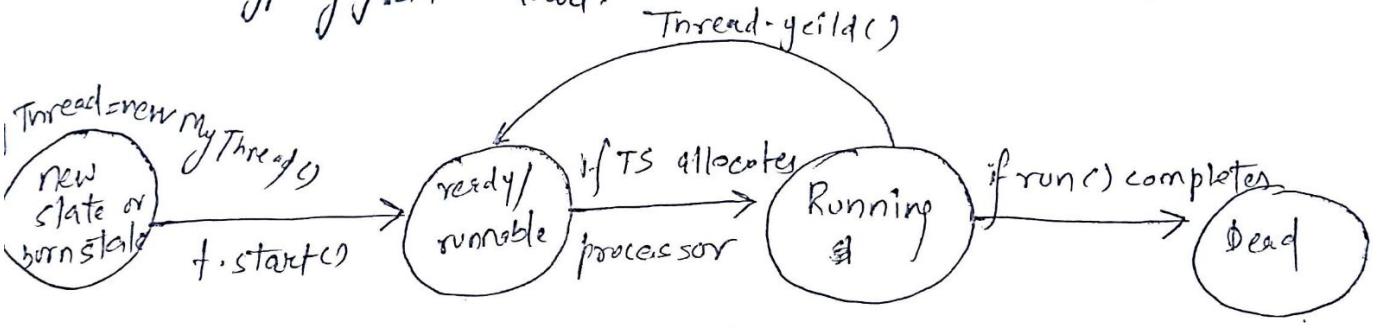
yield:- It causes to pause current executing thread to give the chance for waiting threads of same priority

- 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 schedular.
- The thread which is yeilded, when it will get the chance once again, it depends on Thread schedular.

(7) t x → t1, t2, t3, t4, t5 } 7

```
public static native void yield();
```

Prototype of yield method.




```
class MyThread extends Thread
{
  public void run()
  {
    for(int i=0; i<10; i++)
    {
      sop("child Thread");
      Thread.yield();
    }
  }
}
```

```
class Test
{
  psvm(String[] args)
  {
    MyThread t=new MyThread();
    t.start();
    for(int i=0; i<10; i++)
    {
      sop("main Thread");
    }
  }
}
```

- In the above program, if we are commenting line 1 then both threads will be executed simultaneously & 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 & the chance of completing main thread first is high.
- Some platforms won't provide proper support for yield method.

```
sop("child thread");
Thread.yield();
}
}
}
```

```
→ t.start();
for(int i=0; i<10; i++)
{
sop("main thread");
}
}
```

Join method

If a thread wants to wait until completion of some other thread then we should go for join method.

ex: If a thread t_1 wants to wait until completing t_2, then t_1 has to call `t2.join()`

If t_1 executes t_2.join() then immediately t_1 will be entered into waiting state until t_2 finishes its execution.

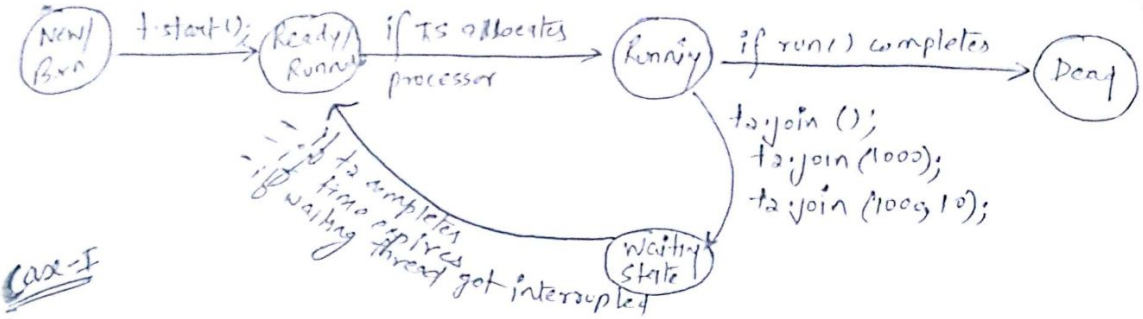
Once t_2 completes then t_1 can continue its execution.

→ checked exception

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

[not static → calling on object]

note: every join method throws InterruptedException which is checked exception. Hence, compulsory we should handle this exception either by using try-catch or by throws keyword. otherwise we will get compile time error.



Case-I

ex

```
Class MyThread extends Thread
{
    public void run()
    {
        for(int i=0; i<5; i++)
        {
            sop("child thread");
            try
            {
            Thread.sleep(4000);
            }
            catch(InterruptedException e)
            {
            }
        }
    }
}
```

```
class Test
{
    psvm(String[] args) throws Interrupted exception
    {
        MyThread t = new MyThread();
        t.start();
        t.join();  // t.join(10000);   (1)
        for(int i=0; i<10; i++)
        {
            sop("main thread");
        }
    }
}
```

- If we comment line 1 then both main & child threads will be executed simultaneously & 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 completion of child thread. In this case output is

childthread,
:
5 times
main thread
:
10 times

join() method on ... thread itself then ... program will be paused (deadlock situation)

```
class Test
{
psvm(String[] args) throws IE
{
Thread.currentThread().join();
}
}
```

main thread — executed by main thread

In this case thread has to wait infinite amount of time.

sleep() method

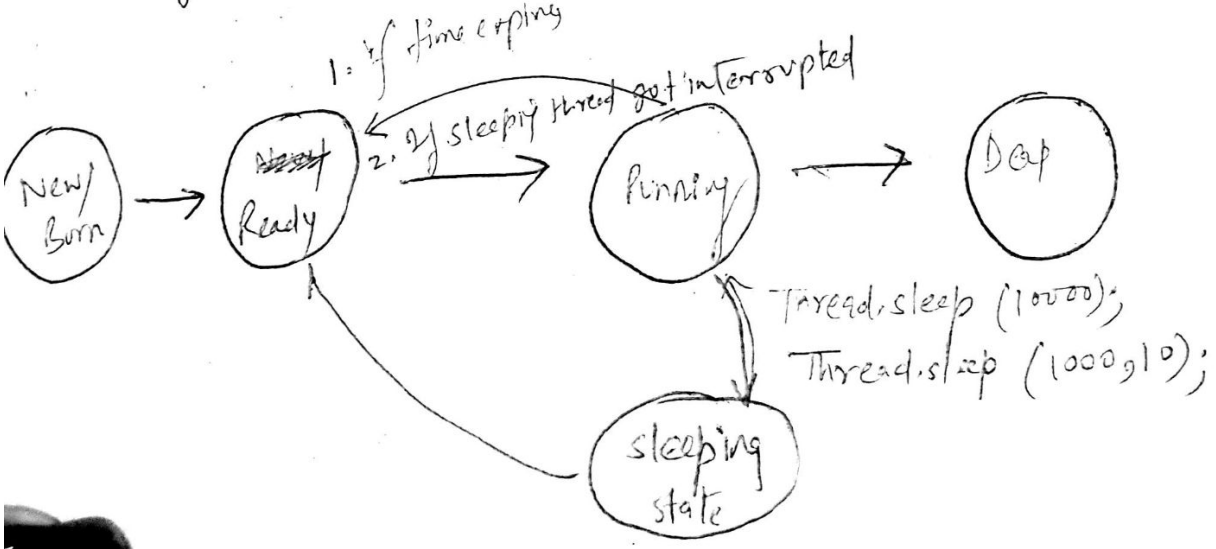
If a thread don't want to perform any operation for a particular amount of time then we should go for sleep method.

public static native void sleep(long ms) throws IE

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

Note → Every sleep method throws InterruptedException, hence (checked exception)

Hence whenever we are using sleep method we must handle InterruptedException either by try-catch or throws keyword. otherwise we will get compile time error.



Case 2

```
class MyThread extends Thread
{
 static Thread mt;
 public void run()
 {
  try {mt.join();
      }
  catch(InterruptedException e){}
  for(int i=0; i<5; i++)
  {
   sop("child Thread");
  }
 }
}
```

```
class Test
{
   psvm(String[] args) throws Interrupted
                              Exception
 {
  MyThread.mt = Thread.currentThread();
 MyThread t = new MyThread();
    t.start();
 → //t.join(); ——— (1) [Case-3]
for(int i=0; i<5; i++)
    {
      sop("main thread");
      Thread.sleep(1000);
    }
  }
}
```

o/p > main thread
:5 times
child Thread
5 times

< child thread waiting for main thread to finish its execution >

Here, child thread calls join() method on main thread object Hence child thread has to waiting until completion of main thread.

Case 3 If we remove the comment in Line (1) above :-

If main thread calls join method on child thread object & child thread calls join method on main thread object then both threads will wait forever & the program will be paused (Deadlock situation)

Case 4

static Thread a;
main() {
a = Thread.currentThread();

Test.a.join();

t.join()

```
class SlideRotator
{
psvm(String[] args) throws InterruptedException
{
for(int i=1; i<10; i++)
  {
    sop("slide" +i);
    Thread.sleep(5000);
  }
}
}
```

How a thread can interrupt another Thread

A thread can interrupt a sleeping thread or waiting thread by using interrupt method of Thread class

public void interrupt()

```
class MyThread extends Thread
{
public void run()
 {
   try {for(int i=0; i<10; i++){
        sop(" child thread");
    Thred.sleep(2000);
      }}
  catch (InterruptedException e)
  {
   sop ("I got interrupted");
  }
}
}
```

```
class Test
{
 psvm(String[] args)
  {
MyThread t= new MyThread();
  t.start();
  t.interrupt();   ← (1)
  sop("end of main");
  }
}
```

If we comment line 1 then main thread won't interrupt child thread
- In this case, child thread will execute for loop 1000 times.

If we are not commenting line 1 then main thread interrupts child thread. In this case output is :-

```
end of main
child thread
I got interrupted
```

```
class MyThread extends Thread
{
public void run()
  {
  for(int i=0; i<1000; i++)
    {
    sop ("I am not so lazy");
    }
  sop ("I want to sleep");
  try
  {
    Thread.sleep(2000);
  }
catch (InterruptedException e)
{
sop ("I got interrupted");
}
}
}
```

```
class Test
{
psvm (String[] args)
{
MyThread t = new MyThread();
   t.start();
t.interrupt();   -> waiting
  sop ("end of main");
}
}
```

the above example, interrupted call waited until child thread completes loop (1000 times)

Comparison Table of yield, join & sleep methods

Property	yield()	join()	sleep()
~~Thread~~ Purpose?	If a thread wants to pause its execution to give the chance for remaining threads of same priority then we should use yield() method	If a thread wants to wait for other thread to finish its execution then we should use join() method	If a thread don't want to perform any operation for a particular amount of time then use sleep() method
Overloaded	NO	YES (3 join methods)	YES (2 sleep methods)
is it final	NO	YES	NO
: it throws nterrupted xception	NO	YES	YES
it native	Yes	NO	sleep(long ms) → native sleep(long ms, int ns) → non-native
t static	Yes	NO	Yes