DeadLock
========
If 2 Threads are waiting for each other forever(without end) such type of situation (infinite waiting) is called dead lock.
There are no resolution techniques for dead lock but several prevention(avoidance) techniques
are possible.
Synchronized keyword is the cause for deadlock hence whenever we are using synchronized keyword
we have to take special care.

eg#1.

```
class A {
      public void d1(b b){
            System.out.println("Thread-1 starts execution of d1()");
            try{
                  Thread.sleep(5000);//5sec
            }
            catch (InterruptedException e){
            }
            System.out.println("Thread-1 trying to call b last()");
            b.last();
      }
      public void last(){
            System.out.println("Inside A last() method");
      }
}
class B {
      public void d2(A a){
            System.out.println("Thread-2 starts execution of d2()");
            try{
                  Thread.sleep(5000);//5sec
            }
            catch (InterruptedException e){
            }
            System.out.println("Thread-2 trying to call A last()");
            a.last();
      }
      public void last(){
            System.out.println("Inside B last() method");
      }
}
public class Test extends Thread {
       A a=new A();
       B b=new B();

       public void m1(){
            this.start();
            a.d1(b);//executed by main thread
       }

       public void run(){
            b.d2(a);//executed by child thread
       }

      public static void main(String[] args){
            Test t=new Test();
            t.m1();
```

```
  }
}

since methods are not synchronized,lock is not requried, so no deadlock
 Output
Thread-1 starts execution of d1()
Thread-2 starts execution of d2()
Thread-1 trying to call B last()
Inside B last() method
Thread-2 trying to call A last()
Inside A last() method


eg#3.
class A extends Thread{
      public synchronized void d1(B b){
            System.out.println("Thread-1 starts execution of d1()");
            try{
                  Thread.sleep(5000);//5sec
            }
            catch (InterruptedException e){
            }
            System.out.println("Thread-1 trying to call B last()");
            b.last();
      }
      public synchronized void last(){
            System.out.println("Inside A last() method");
      }
}
class B extends Thread{
      public synchronized void d2(A a){
            System.out.println("Thread-2 starts execution of d2()");
            try{
                  Thread.sleep(5000);//5sec
            }
            catch (InterruptedException e){
            }
            System.out.println("Thread-2 trying to call A last()");
            a.last();
      }
      public synchronized void last(){
            System.out.println("Inside B last() method");
      }
}
public class Test extends Thread {
       A a=new A();
       B b=new B();

       public void m1(){
            this.start();
            a.d1(b);//line executed by main thread
       }

       public void run(){
            b.d2(a);//line executed by child thread
       }

      public static void main(String[] args){
```

```
            Test t=new Test();
            t.m1();//main thread s executing
      }
}
```

In the above program, there is a possiblity of "deadlock".

```
Output
Thread-1 starts execution of d1()
Thread-2 starts execution of d2()
Thread-1 trying to call B last()
Thread-2 trying to call A last()
 //here cursor will be waiting
```

t1 => starts d1(),since d1() is synchronized and a part of 'A' class so t1 applies lockof(A) and
starts the execution, while executing it encounters Thread.sleep().so T.S gives chance
for t2 thread.
After getting a chance again by TS, it tries to execute b.last.
but lock of b is with t2 thread, so t1 enters into waiting state.

t2=> starts d2(),since d2() is synchronized and a part of 'B' class so t2 applies lockof(B) and
starts the execution, while executing it encounter Thread.sleep(),so TS gives chance again for t1 thread.

After getting a chance again by TS, it tries to execute a.last
but lock of a is with t1 thread, so t2 enters into waiting state.

Since both the threads are in waiting state and it would be waiting for ever,so we say the above pgm would result in "DeadLock".

Note:
synchronized is the only reason why there is a deadlock,so we should be careful when we use
synchronized keyword,if we remove atleast one synchronized word then the program wont enter into
dead lock.

DeadLock vs starvation
======================
Long waiting of a thread, where waiting never ends is termed "deadlock".
Long waiting of a thread, where waiting ends at certain point is called "starvation".

eg:: Assume we have 1cr threads, where all 1cr threads have priority is 10,but one thread is there
which has priority 1,now the thread with a priority-1 has to wait for long time but still
it gets a chance,but it has to wait for long time, this scenario is called "Starvation".

Note::
Low priority thread has to wait untill completing all priority threads but ends at certain
point which is nothing but starvation.

Daemon Threads

==========
 The thread which is executing in the background is called "DaemonThread".
 eg: AttachListener,SignalDispatcher,GarbageCollector,....

remember the example of movie
   1. producer
   2. director
   3. music director
   4. ....
   5. ....
   6. ....

MainObjective of DaemonThread
  The main objective of DaemonThread, to provide support for Non-Daemon
threads(main thread).
eg:: if main threads runs with low memory then jvm will call GarbageCollector
thread, to destroy
     the useless objects,so that no of bytes of free memeory will be improved with
this free
     memory main thread can continue its execution.

Usually Daemon threads having low priority,but based on our requirement daemon
threads can run
with high priority also.
   JVM => creates 2 threads
            a. Daemon Thread(priority=1,priority=10)
            b. main (priority=5)
                  while executing the main code, if there is a shortage of memory
then immediately
                  jvm will change the priority of Daemon thread to 10, so Garbage
collector
                  activates Daemon thread and it frees the memory after doing it
immediately
                  it changes the priority to 1, so main thread it will continue.

How to check whether the Thread is Daemon or not?
  public boolean isDaemon() => To check wheter the thread is "Daemon"
  public void setDaemon(boolean b) throws IllegalThreadStateException
             b=> true,means the thread will become Daemaon,before starting the
Thread we need
                to make the thread as "Daemon" otherwise it would result in
                "IllegalThreadStateException".

What is the deafult nature of the Thread?
 Ans. By deafult the main thread is "NonDaemon".
      for all remaining thread Daemon nature is inherited from Parent to child,
that is
      if the parent thread is "Daemon" then child thread will become "Daemon" and
if the parent
      thread is "NonDaemon" then automatically child thread is also "NonDaemon".

 Is it possible to change the NonDameon nature of Main Thread?
 Ans. Not possible, becoz the main thread starting is not in our hands, it will be
started by
      "JVM".

eg::
class MyThread extends Thread{}
public class Test {

```
	public static void main(String[] args){
		System.out.println(Thread.currentThread().isDaemon());//false
		Thread.currentThread().setDaemon(true);//RE:IllegalThreadStartException

		MyThread t=new MyThread();
		System.out.println(t.isDaemon());//false
		t.setDaemon(true);
		t.start();
		System.out.println(t.isDaemon());//true
	}
}
```

Note::
Whenever last NonDaemon threads terminates,automatically all Daemon Threads will be terminated
irrespective of their position.
 eg:: makeup man in shooting is a DaemonThread
	hero is main thread
	if hero role is over, then automatically the makeup role is also over automatically.

```
eg::
class MyThread extends Thread{
	public void run(){
		for (int i=1;i<=10 ;i++ ){
			System.out.println("child thread");
			try{
				Thread.sleep(2000);//2sec
			}
			catch (InterruptedException e){
				System.out.println(e);
			}
		}
	}
}
public class Test  {
	public static void main(String[] args){
		MyThread t=new MyThread();
		t.setDaemon(true);//stmt-1
		t.start();
		System.out.println("end of main thread");
	}
}
```

Output:
if we comment stmt-1, then both the threads are NonDaemon threads it would continue with its
execution.

```
end of main thread
child thread
child thread
 ...
 ...
 ...
```

Output
If we remove comment on stmt-1, then main thread is NonDaemon thread where as userdefined thread is DaemonThread, if the main thread finishes the execution then automatically the

DaemonThread also will finish the execution.