
Thread

1 dec

03 December 2022 00:21

Syllabus

========

1.Introduction.

2. The ways to define, instantiate and start a new Thread.
    1. By extending Thread class
    2. By implementing Runnable interface
3. Thread class constructors
4. Thread priority
5. Getting and setting name of a Thread.
6. The methods to prevent(stop) Thread execution.
    1. yield()
    2. join()
    3. sleep()
7. Synchronization.
8. Inter Thread communication.
9. Deadlock
10. Daemon Threads.
11. Various Conclusion
    1. To stop a Thread
    2 . Suspend + resume of a thread
    3. Thread group
    4. Green Thread
    5. Thread Local
12. Life cycle of a Thread

MulitTasking

==========

Executing several task simultaneously is the concept of multitasking.

There are 2 types of Multitasking.

a. Process based multitasking

b. Thread based multitasking.

Process based multitasking

==========

Executing several tasks simultaneously where each task is a seperate independent process such type of multitasking is called "process based multitasking".

eg:: typing a java pgm

listening to a song

downloading the file from internet

Process based multitasking is best suited at "os level".

Thread based multitasking

==========

=>Executing several tasks simulatenously where each task is a seperate independent part of the same Program, is called "Thread based MultiTasking". Each independent part is called "Thread".

1. This type of multitasking is best suited at "Programatic level".
The main advantages of multitasking is to reduce the response time of the system and to improve the performance.

2. The main important application areas of multithreading are
a. To implement mulitmedia graphics
b. To develop web application servers(will learn in JEE)
c. To develop video games
d. To develop animations

3. Java provides inbuilt support to work with threads through API called
Thread,Runnable,ThreadGroup,ThreadLocal,...

4. To work with multithreading, java developers will code only for 10% remaining
90% java API will take care..

What is thread?
A. Seperate flow of execution is called "Thread".
if there is only one flow then it is called "SingleThread" programming.
For every thread there would be a seperate job.

B. In java we can define a thread in 2 ways
a. Implementing Runnable interface
b. extending Thread class

1. Extending Thread class
=> we can create a Thread by extending a Thread.

```
class MyThread extends Thread{
    @Override
    public void run(){
    for(int i=0;i<10;i++)
    System.out.println("child thread");
    }
    }
```
} task

defining a thread(writing a class and extending a Thread) job a thread(code written inside run())

```
class ThreadDemo{
    public static void main(String... args){
    MyThread t =new MyThread();//Thread instantiation
    t.start();//starting a thread
    ;;;; // At this line 2 threads are there
    for(int i=1;i<=5;i++)
    System.out.println("Main Thread");
    }
```

```
}}
```

Behind the scenes

1. Main thread is created automatically by JVM.
2. Main thread creates child thread and starts the child thread.

ThreadScheduler

==========

If multiple threads are waiting to execute,then which thread will execute 1st is decided by ThreadScheduler which is part of JVM.

In case of MultiThreading we can't predict the exact output only possible output we can expect.

Since jobs of threads are important,we are not interested in the order of execution it should just execute such that performance should be improved.

CPU utilize efficiently in given time.

case2: diff b/w t.start() and t.run()

if we call t.start() and seperate thread will be created which is responsible to execute run() method.

if we call t.run(), no seperate thread will be created rather the method will be called just like normal method by main thread.

if we replace t.start() with t.run() then the output of the program would be

child thread	main thread
child thread	main thread
child thread	main thread
child thread	main thread
child thread	main thread
child thread	main thread
child thread	main thread
child thread	main thread
child thread	main thread
child thread	main thread

case3:: Importance of Thread class start() method

For every thread, required mandatory activities like registering the thread with Threadscheduler will be taken care by Thread class start() method and programmer is responsible of just doing the job of the Thread inside run() method.

start() acts like an assistance to programmer.

```
public void start()
{
```

```
register thread with ThreadScheduler
All other mandatory low level activities
invoke or calling run() method.
}
```

We can conclude that without executing Thread class start() method there is no chance of starting a new Thread in java.

Due to this start() is considered as heart of MultiThreading.

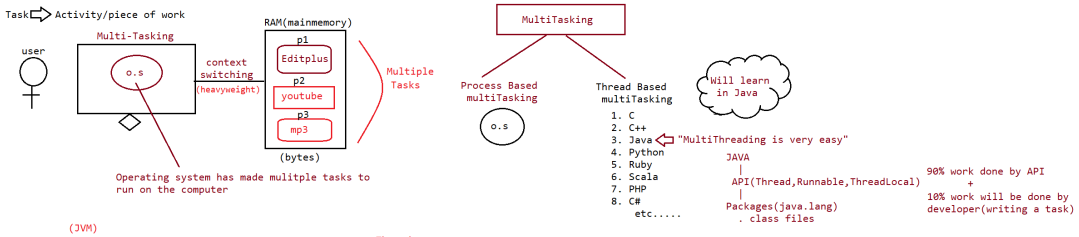
case4:: If we are not overriding run() method

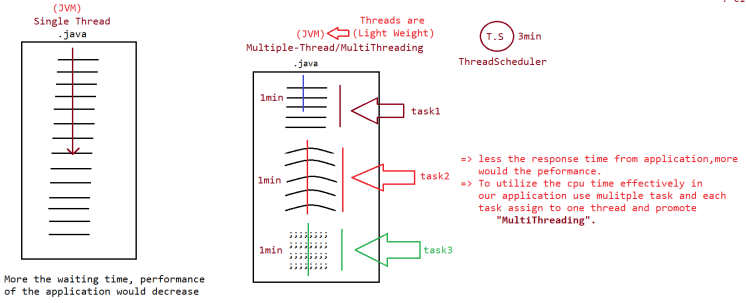
If we are not Overriding run() method then Thread class run() method will be executed which has empty implementation and hence we wont get any output.

eg::

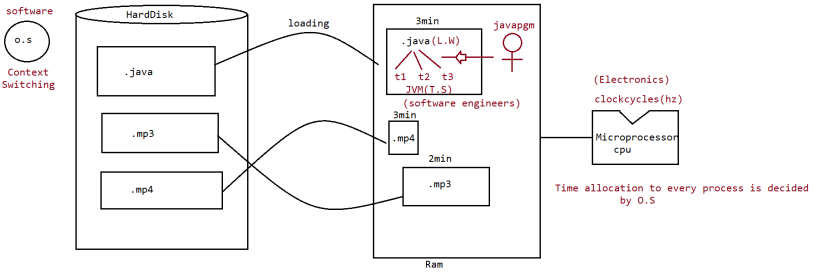
```
class MyThread extends Thread{}
class ThreadDemo{
public static void main(String... args){
MyThread t=new MyThread();
t.start();
}
}
```

It is highly recomended to override run() method, otherwise don't go for MultiThreading concept.





Agenda of MultiTasking
==========
1. To use CPU time effectively,so that performance can be improved




```
class MyThread extends Thread
{
    @Override
    public void run()
    {
        for (int i=1;i<=10;i++)
        {
            System.out.println("Child thread");
        }
    }
}
```

override run() to define a job for a thread

Task of a Thread

Defining a Thread

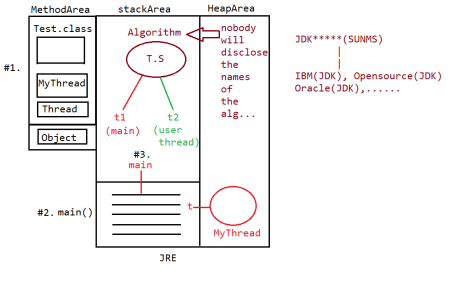
```
public class Thread{
    public void run(){

    }
    public void start(){
      //logic internally available
    }
}
```

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

        MyThread t = new MyThread();
        t.start();
        mainthread and userdefined thread
        for (int i = 1;i<=10;i++)
        {
            System.out.println("Main thread");
        }
    }
}
```

```
javac Test.java => Test.class,MyThread.class
java  Test
      |=> contain main() so load and start
          the execution
```



2 dec

04 December 2022 11:40

case5:Overloading of run() method

we can overload run() method but Thread class start() will always call run() with zero argument.

if we overload run method with arguments, then we need to explicitly call argument based run method and it will be executed just like normal method.

eg::
```
class MyThread extends Thread{
    public void run(){
    System.out.println("no arg method");
    }
    public void run(int i){
    System.out.println("zero arg method");
    }
    }
    class ThreadDemo{
    public static void main(String... args){
    MyThread t=new MyThread();
    t.start();
    }
    }
```
Output:: NO arg method.

Case6::Overriding of start() method

If we override start() then our start() method will be executed just like normal method,but no new Thread will be created and no new Thread will be started.

eg#1.
```
class MyThread extends Thread{
    public void run(){
    System.out.println("no arg method");
    }
    public void start(){
    System.out.println("start arg method");
    }
    }
    class ThreadDemo{
    public static void main(String... args){
    MyThread t=new MyThread();
    t.start();
    }
    }
```
Output:: start arg method

It is never recomended to override start() method.

eg#2.

```
class MyThread extends Thread{
```

case7::

```
class MyThread extends Thread{
    public void run(){
    System.out.println("run method");
    }
    public void start(){
    System.out.println("start method");
    }
    }
    class ThreadDemo{
    public static void main(String... args){
    MyThread t=new MyThread();
    t.start();
    System.out.println("Main method");
    }
    }
```

Output:: MainThread

a. Main method

b. start method.

```
    public void start(){
    super.start();
    System.out.println("start method");
    }
    public void run(){
    System.out.println("run method");
    }
    }
    class ThreadDemo{
    public static void main(String... args){
    MyThread t=new MyThread();
    t.start();
    System.out.println("Main method");
    }
    }
```

Output::

MainThread

a. Main method

b. start method

UserDefinedThread

a. run method

case8:: Life cycle of a Thread

```
MyThread t=new MyThread(); // Thread is in born state
 t.start(); //Thread is in ready/runnable state
```

if Thread scheduler allocates CPU time then we say thread entered into Running state.

if run() is completed by thread then we say thread entered into dead state.

=> Once we created a Thread object then the Thread is said to be in new state or born state.

=> Once we call start() method then the Thread will be entered into Ready or Runnable state.

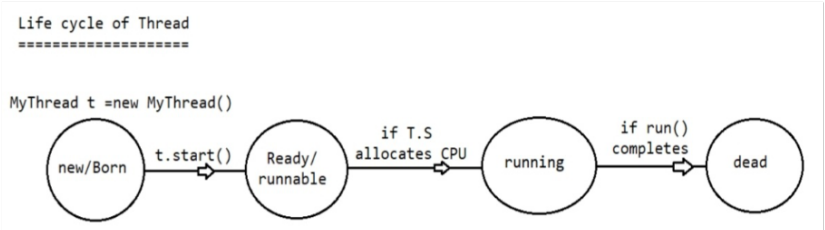
=> If Thread Scheduler allocates CPU then the Thread will be entered into running state.

=> Once run() method completes then the Thread will entered into dead state.

case9::

After starting the Thread, we are not supposed to start the same Thread again,then we say Thread is in "IllegalThreadStateException".

```
MyThread t=new MyThread(); // Thread is in born state
t.start(); //Thread is in ready state
....
....
t.start(); //IllegalThreadStateException
```



Creation of Thread using Runnable interface

==========

1. Creating a Thread using java.lang.Thread class

a. use start() from Thread class

b. override run() and define the job of the Thread.

2. Creation of a Thread requirement to SUNMS is an SRS

```
interface Runnable{
    void run();
    }
    class Thread implements Runnable{ // Adapter class
    public void start(){
    1. Register the thread with ThreadScheduler
    2. All other mandatory low level activities(memory level)
    3. invoke or call run() method
    }
    public void run(){
    //job for a thread
    }
```

} functional interface.

shortcuts of eclipse

ctrl+shift+T => To open a definition of any class

ctrl + o => To list all the methods of the class

Note:

public java.lang.Thread();

|=> thread class start(), followed by thread class run()

public java.lang.Thread(java.lang.Runnable);

|=> thread class start(), followed by implementation class of Runnable run()

Defining a Thread by implementing Runnable Interface

==========

```
public interface Runnable{
    public abstract void run();
   }
   public class Thread implements Runnable{
   public void start(){
   1. register Thread with ThreadScheduler
   2. All other mandatory low level activites
   3. invoke run()
   }
   public void run(){
   //empty implementation
   }
   }
```

```
eg::1
 class MyRunnable implements Runnable{
@Override
public void run(){
for(int i=1;i<=10;i++)
System.out.println("child thread");
}
 }
public class ThreadDemo{
public static void main(String... args){
MyRunnable r=new MyRunnable();
Thread t=new Thread(r);//call MyRunnable run()
t.start();
for(int i=1;i<=10;i++)
System.out.println("main thread");
}
}
Output::
 MainThread
 a. main thread
 ....
 ....
 ChildThread
 a. child thread
...
...
 ...
```

Case study

==========

```
MyRunnable r=new MyRunnable();
Thread t1=new Thread();
Thread t2=new Thread(r);
```

case1: t1.start()

A new thread will be created,which is responsible for executing Thread class run()

|

no o/p

output

mainthread

main thread

main thread

main thread

main thread

main thread

case2: t2.start()

A new thread will be created,which is responsible for executing MyRunnable run()

output
mainthread
main thread
main thread
main thread
main thread
main thread
userdefinedthread
child thread
child thread
child thread
child thread
child thread

case3: t1.run()

No new thread will be created,but Thread class run() will be executed just like normal method call.

output
mainthread
main thread
main thread
main thread
main thread
main thread

case4: t2.run()

No new thread will be created,but MyRunnable class run() will be executed just like normal method call.

output
mainthread
child thread
child thread
child thread
child thread
child thread
main thread
main thread
main thread
main thread
main thread

case5: r.start()

— start() not inside Runnable.

It results in CompileTime Error

case6. r.run()

No new thread will be created,but MyRunnable class run() will be executed just like normal method call.

```
MyRunnable r=new MyRunnable();
 Thread t1=new Thread();
 Thread t2=new Thread(r);
case1: t1.start()
case2: t2.start()
case3: t2.run()
case4: t1.run()
case5: r.start()
case6: r.run()
```

output
mainthread
child thread
child thread
child thread
child thread
child thread
main thread
main thread
main thread
main thread
main thread

In which of the above cases a new Thread will be created which is responsible for the execution of MyRunnable run() method ?

t2.start();

In which of the above cases a new Thread will be created ?

t1.start();

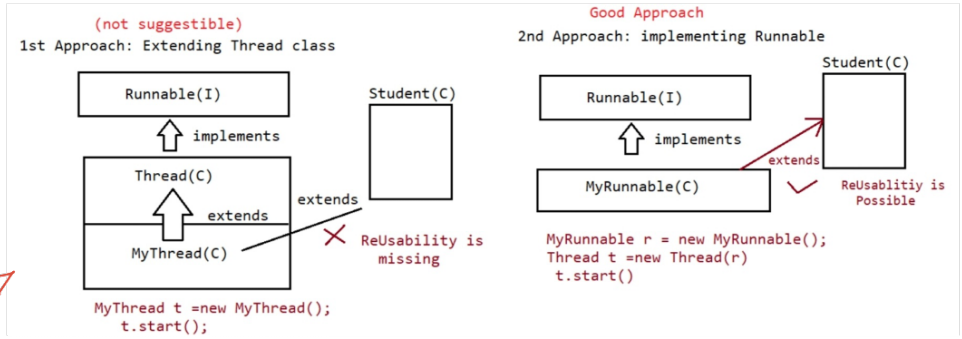
t2.start();

In which of the above cases MyRunnable class run() will be executed?

t2.start();

t2.run();

r.run();



Different approach for creating a Thread?

A. extending Thread class

B. implementing Runnable interface

Which approach is the best approach?

a. implements Runnable interface is recomended becoz our class can extend other class through which inheritance benift can brought in to our class.

Internally performance and memory level is also good when we work with interface.

b. if we work with extends feature then we will miss out inheritance benefit becoz already our class has inherited the feature from "Thread class", so we normally don't prefere extends approach rather implements approach is used in real time for working with "MultiThreading".

Various Constructors available in Thread class

==========

a. Thread t=new Thread()

b. Thread t=new Thread(Runnable r)

c. Thread t=new Thread(String name)

d. Thread t=new Thread(Runnable r,String name)

e. Thread t=new Thread(ThreadGroup g, String name);

f. Thread t=new Thread(ThreadGroup g, Runnable r);

g. Thread t=new Thread(ThreadGroup g, Runnable r,String name);

h. Thread t=new Thread(ThreadGroup g, Runnable r,String name,long stackSize);

Alternate approach to define a Thread(not recomended)

==========

```
class MyThread extends Thread{
    public void run(){
    System.out.println("child thread");
    }
    }
    class ThreadDemo {
    public static void main(String... args){
    MyThread t=new MyThread();
    Thread t1=new Thread(t);
    t1.start();
    System.out.println("main thread");
    }
    }
```

'implements Runnable

Output::2 threads are created

MainThread

main thread

ChildThread

child thread

internally related

==========

Runnable

^

|

Thread

^

|

MyThread

Names of the Thread

==========

Internally for every thread,there would be a name for the thread.

a. name given by jvm

b. name given by the user.

eg::

```
class MyThread extends Thread{
}
public class TestApp{
public static void main(String... args){
System.out.println(Thread.currentThread().getName());//main
MyThread t=new MyThread();
t.start();
System.out.println(t.getName());//Thread-0
Thread.currentThread().setName("Yash");//Yash
System.out.println(Thread.currentThread().getName());//Yash
System.out.println(10/0);
 //Exception in thread "yash" java.lang.ArithmeticException:/by zero
 TestApp.main()
}
}
```

=> It is also 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 String getName();
public final void setName(String name);
```

eg#2.

```
class MyThread extends Thread{
    @Override
    public void run(){
     System.out.println("run() executed by Thread ::
    "+Thread.currentThread().getName());
    }
    }
    public class TestApp{
    public static void main(String... args){
    MyThread t=new MyThread();
    t.start();
     System.out.println("main() executed by Thread ::
    "+Thread.currentThread().getName());
    }
    }
```

Output:: run() executed by Thread:: Thread-0
main() executed by Thread:: main

5 dec

06 December 2022 12:33

ThreadPriorities

==========

For every Thread in java has some priority.

valid range of priority is 1 to 10,it is not 0 to 10.

if we try to give a differnt value the it would result in "IllegalArgumentException".

Thread.MIN_PRIORITY = 1

Thread.MAX_PRIORITY = 10

Thread.NORM_PRIORITY = 5

Thread class does not have priorities is

Thread.LOW_PRIORITY,Thread.HIGH_PRIORITY.

Thread scheduler allocates cpu time based on "Priority".

If both the threads have the same priority then which thread will get a chance as a pgm we can't

predict becoz it is vendor dependent.

We can set and get priority values of the thread using the following methods

a. public final void setPriority(int priorityNumber)

b. public final int getPriority()

The allowed priorityNumber is from 1 to 10,if we try to give other values it woudl result in

"IllegalArgumentException".

```
System.out.println(Thread.currentThread().setPriority(100);//IllegalArgumentException.
```

DefaultPriority

==========

The default priority for only main thread is "5",where as for other threads priority will be inherited from parent to child.

Parent Thread priority will be given as Child Thread Priority.

eg#1.

```
class MyThread extends Thread{}
public class TestApp{
    public static void main(String... args){
        System.out.println(Thread.currentThread().getPriority());//5
        Thread.currentThread().setPriority(7);
        MyThread t= new MyThread();
        System.out.println(Thread.currentThread().getPriority());//7
    }
}
```

MyThread is creating by "mainThread", so priority of "mainThread" will be shared as a

priority for "MyThread".

eg#2.

```
class MyThread extends Thread{
    @Override
    public void run(){
        for (int i=1;i<=5 ;i++ ){
            System.out.println("child thread");
        }
    }
}
public class TestApp{
    public static void main(String... args){

        MyThread t= new MyThread();
        t.setPriority(7);//line -1
        t.start();
        for (int i=1; i<=5; i++){
            System.out.println("main thread");
        }

    }
}
```

O/P { main main = child child =

Since priority of child thread is more than main thread, jvm will execute child thread first whereas for the parent thread priority is 5 so it will get last chance.
if we comment line-1, then we can't predict the order of execution becoz both the threads have same priority.

Some platform won't provide proper support for Thread priorities. (cracked version of windows) that's why it won't have scheduling algorithms.
eg:: windows7,windows10,...

We can prevent Threads from Execution
a. yield()
b. sleep()
c. join()

yield() => It causes to pause current executing Thread for giving chance for waiting Threads of same priority.
If there is no waiting Threads or all waiting Threads have low priority then same Thread can continue its execution.
If all the threads have same priority and if they are waiting then which thread will
get chance we can't expect, it depends on ThreadScheduler.
The Thread which is yielded, when it will get the chance once again depends on the mercy on "ThreadScheduler" and can't expect exactly.
public static native void yield()

MyThread t= new MyThread() //new state or born state
t.start() // enter into ready state/runnable state
if ThreadScheduler allocates processor then enters into running state.

a. if running Thread calls yield() then it enters into runnable state.
if run() is finished with execution then it enters into dead state.

eg#1.

```
class MyThread extends Thread{

    @Override
    public void run(){
        for (int i=1;i<=5 ;i++ ){
            System.out.println("child thread");
            Thread.yield();//line-1
        }
    }
}
public class TestApp{
    public static void main(String... args){
        MyThread t= new MyThread();
        t.start();
        for (int i=1;i<=5 ;i++ ){
            System.out.println("Parent Thread");
        }
    }
}
```

Note::

If we comment line-1, then we can't expect the output becoz both the threads have same priority then which
thread the ThreadScheduler will schedule is not in the hands of programmer but if we don't comment line-1,
then there is a possibility of main thread getting more no of times, so main thread execution is faster than
child thread will get chance.

Note: Some platforms wont provide proper support for yield(),becuase it is getting the execution
code from other language prefereably from 'C'.

b. join()

If the thread has to wait untill the other thread finishes its execution then we need to go for join().
if t1 executes t2.join() then t1 should should wait till t2 finishes its execution.
t1 will be entered into waiting state untill t2 completes, once t2 completes then t1 can continue with its execution.

eg#1.

venue fixing ======> t1.start()
wedding card printing ======> t2.start()======> t1.join()
wedding card distrubution ======> t3.start()======> t2.join()

Prototype of join()

==========

wait

```
public final void join() throws InterruptedException ——→ until finish
public final void join(long ms)throws InterruptedException → till limited time.
public final void join(long ms,int ns)throws  → till limited time
InterruptedException
```

Note: While one thread is in waiting state and if one more thread interupts then it would result
in "InteruptedException".InteruptedException is checkedException which should always be
handled.

Thread t =new Thread();//new/born state
t.start();//ready/runnable state
-> If T.S 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 the thread finishes the execution/time expires/interupted then it would come back to ready state/runnable state.
-> If run() is completed then it would enter into dead state.

eg#1.

```
class MyThread extends Thread{
    @Override
    public void run(){
        for (int i=1;i<=10 ;i++ ){
            System.out.println("Sita Thread");
            try{
                Thread.sleep(2000);
            }
            catch (InterruptedException e){
            }
        }
    }
}
public class Test3 {
    public static void main(String... args)throws InterruptedException{
        MyThread t=new MyThread();
        t.start();
        t.join(10000);//line-n1    Rama waits until sita joins (executed)
        for (int i=1;i<=10;i++ ){
            System.out.println("rama thread");
        }
    }
}
```

```
Output
 2 Threads
 a. Child Thread
    sita thread
    sita thread
         ....
 b. Main Thread
    rama thread
    rama thread
        ....
```

=> If line-n1 is commented then we can't predict the output becoz it is the duty of the T.S to assign C.P.U time
=> If line-n1 is not commented, then rama thread(main thread) will enter into waiting state till

sita thread(child thread) finishes its execution.

Waiting of Child Thread untill Completing Main Thread

==========

we can make main thread to wait for child thread as well as we can make child thread also to wait for main thread.

eg#1.

```
class MyThread extends Thread{
    static Thread mt;
    @Override
    public void run(){
            try{
                mt.join();
            }
            catch (InterruptedException e){
            }

            for (int i=1;i<=10 ;i++ ){
                System.out.println("child thread");
            }

    }
}
public class Test3 {
    public static void main(String... args)throws InterruptedException{
        MyThread.mt=Thread.currentThread();
        MyThread t=new MyThread();
        t.start();

        for (int i=1;i<=10;i++ ){
            System.out.println("main thread");
            Thread.sleep(2000);//20sec sleep
        }

    }
}
```

deadlock

```
Output
 2 Threads(MainThread,ChildThread)
MainThread
 a. main thread
     ....
     ....
ChildThread
 a. child thread
     ....
     ....
```

eg#2.

```
class MyThread extends Thread{
    static Thread mt;
    @Override
    public void run(){
            try{
                mt.join();
            }
            catch (InterruptedException e){
            }

            for (int i=1;i<=10 ;i++ ){
                System.out.println("child thread");
            }

    }
}
public class Test3 {
    public static void main(String... args)throws
InterruptedException{
        MyThread.mt=Thread.currentThread();
        MyThread t=new MyThread();
        t.start();
        t.join();

        for (int i=1;i<=10;i++ ){
            System.out.println("main thread");
            Thread.sleep(2000);//2sec sleep
        }

    }
}
```

```
output:
2 threads(Main,child thread)
main thread
    ;;;;
    ;;;;
childthread
```

if mt.join() //removed

child
child
=
main
main
=

2sec

Note::

If both the threads invoke t.join(),mt.join() then the program would result in "deadlock".

eg#3.

```
public class Test3 {
    public static void main(String... args)throws
```

```
InterruptedException{
        Thread.currentThread().join();
    }
}
```

Output:: Deadlock, becoz main thread is waiting for the main thread itself.

sleep()

======

If a thread dont' want to perform any operation for a particular amount of time then we should go for sleep().

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 checkedexception so we should compulsorily handle the exception using try catch or by throws keyword otherwise it would result in compile time error.

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

t.start() // ready/runnable state sleep(); // ∞ sleep.

=> If T.S 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

=> If time expires/ if sleeping thread got interrupted then thread would come back to "ready/runnable state".

eg#1.

```
public class SlideRotator {
    public static void main(String... args)throws InterruptedException{
        for (int i=1;i<=10 ;i++ ){
            System.out.println("Slide: "+i);
            Thread.sleep(5000);
        }
    }
}
```

Output::

Slide:: 1

Slide:: 2

Slide:: 3

Slide:: 4

Slide:: 5

Slide:: 6

Slide:: 7

Slide:: 8

Slide:: 9

Slide:: 10

6th dec

07 December 2022 00:43

Interupting a Thread

==========

public void interrupt()

=> If thread is in sleeping state (sleep()) or in waiting state (join()) we can interupt a thread.

eg#1.

```
class MyThread extends Thread{
    @Override
    public void run(){
        try{
            for (int i=1;i<=10;i++ ){
                System.out.println("I am lazy thread");
                Thread.sleep(2000);
            }
        }
        catch (InterruptedException e){
            System.out.println("I got interrupted");
        }
    }
}
public class Test3 {
    public static void main(String... args)throws InterruptedException{

        MyThread t=new MyThread();
        t.start();
        t.interrupt();//line-n1
        System.out.println("End of Main...");

    }
}
```

Scneario:: If t.interrupt() then
2 thread
a. Main Thread
main thread
b. Child Thread
I am lazy thread
I got interrupted

Scenario:: If a comment line-n1
2 thread
a. Main Thread
End of Main...
b. Child Thread
I am lazy thread
.....
.....

eg#2.

```
class MyThread extends Thread{
    @Override
    public void run(){
        for (int i=1;i<=10000 ;i++ ){
            System.out.println("I am lazy thread : "+i);
        }
        System.out.println("I am entering into sleeping state");
        try
        {
            Thread.sleep(2000);
        }
        catch (InterruptedException ie){
            ie.printStackTrace();
        }
    }
}
public class TestApp {
    public static void main(String[] args)throws InterruptedException {
```

```
        MyThread t=new MyThread();
        t.start();
        t.interrupt();//line-n1
        System.out.println("main thread");

    }
}
```

line-n1 is commented then no problem

line-n1 is not commented, then interrupt() will wait till the Thread enters into waiting state/sleeping state.

Note::

If thread is interrupting another thread, but target thread is not in waiting state/sleeping state then there would be no exception.

interrupt() call be waiting till the target thread enters into waiting state/sleeping state so this call wont be wasted.

once the target thread enters into waiting state/sleeping state then interrupt() will interrupt and it causes the exception.

interrupt() call will be wasted only if the Thread does not enters into waiting state/sleeping state.

yield() join() sleep()

==========

1) Purpose

yield()

To pause current executing Thread for giving the chance of remaining waiting Threads of same priority.

join()

If a Thread wants to wait until completing some other Thread then we should go for join.

sleep()

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

2) Is it static

yield() yes
join() no
sleep() yes

3) Is it final?

yield() no
join() yes
sleep() no

4) Is it overloaded?

yield() no
join() yes
sleep() yes

5) Is it throws IE?

yield() no
join() yes
sleep() yes

```
6) Is it native method?
        yield() yes
    join()  no
    sleep()
      sleep(long ms) -->native
              sleep(long ms,int ns) -->non-native
```

Note::using lambda expression

```
Runnable r = ()-> {
    for (int i = 1;i<=5 ; i++)
    {
        System.out.println("child thread");
    }
  };
Thread t = new Thread(r);
t.start();
```

using annonmyous inner class

==========

```
new Thread(new Runnable(){
    @Override
    public void run(){
            for (int i = 1;i<=5 ;i++ )
            {
                    System.out.println("child thread");
            }
        }
    }
).start();
```

synchronization — 2 levels — 1. object, 2. class

==========

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

```
class X{
    synchronized void m1(){}
    synchronized void m2(){}
    void m3(){}
 }
```

KeyPoints

==========

1. if t1 thread invokes m1() then on the Object X lock will applied.
2. if t2 thread invokes m2() then m2() can't be called because lock of X object is with m1.
3. if t3 thread invokes m3() then execution will happen becoz m3() is non-synchronized.

Lock concept is applied at the Object level not at the method level.

7. Every object in java has a unique lock. Whenever we are using synchronized keyword then only lock concept will come into the picture.

8. If a Thread wants to execute any synchronized method on the given object 1st it has to get the lock of that object.

Once a Thread got the lock of that object then it's allow to execute any synchronized method on that object.

If the synchronized method execution completes then automatically Thread releases lock.

9. While a Thread executing any synchronized method the remaining Threads are not allowed execute any synchronized method on that object simultaneously. But remaining Threads are allowed to execute any non-synchronized method simultaneously. [lock concept is implemented based on object but not based on method].

Note::

Every object will have 2 area[Synchronized area and NonSynchronized area]

Synchronized Area => write the code only to perform update,insert,delete

NonSynchronized Area => write the code only to perform select operation

```
class ReservationApp{
    checkAvailablity(){
        //perform read operation
```

```
        }
    synchronized bookTicket(){
        //peform update operation
    }
}
```

eg#1.

```
class Display{
    public void wish(String name){
        for (int i=1;i<=10 ;i++ )
        {
            System.out.print("Good Morning: ");
            try{
                Thread.sleep(2000);
            }
            catch (InterruptedException e){
            }
            System.out.println(name);
        }
    }
}
class MyThread extends Thread{
    Display d;
    String name;

    MyThread(Display d,String name){
        this.d=d;
        this.name=name;
    }

    @Override
    public void run(){
        d.wish(name);
    }
}
public class Test3 {
    public static void main(String... args){
        Display d=new Display();
        MyThread t1= new MyThread(d,"dhoni");
        MyThread t2= new MyThread(d,"yuvi");
        t1.start();
        t2.start();
    }
}
```

Ouput:: As noticed below the output is irregular becoz at a time on a resource called *wish*()

2 threads are acting simulataneously.

3 Threads

a. *Main* Thread

b. *Child* Thread-1

c. *Child* Thread-2

```
GoodMorning :GoodMorning : ..
....
....
....
....
....
```

eg#2.

```
class Display{
    public synchronized void wish(String name){
        for (int i=1;i<=10 ;i++ )
        {
            System.out.print("Good Morning: ");
```

```
            try{
                Thread.sleep(2000);
            }
            catch (InterruptedException e){
            }
            System.out.println(name);
        }
    }
}
class MyThread extends Thread{
    Display d;
    String name;

    MyThread(Display d,String name){
        this.d=d;
        this.name=name;
    }

    @Override
    public void run(){
            d.wish(name);
    }
}
public class Test3 {
    public static void main(String... args)throws InterruptedException{
        Display d=new Display();
        MyThread t1= new MyThread(d,"dhoni");
        MyThread t2= new MyThread(d,"yuvi");
        t1.start();
        t2.start();
    }
}
```

```
Ouput::
3 Threads
 a. Main Thread
 b. Child Thread-1
    GoodMorning:dhoni
    GoodMorning:dhoni
    .....
    .....
    .....
 c. Child Thread-2
    GoodMorning:yuvi
    GoodMorning:yuvi
    ....
    ....
....
```

Note::

As noticed above there are 2 threads which are trying to operate on single object called "Display" we need synchronization to resolve the problem of "Datainconsistency".

casestudy::

```
Display d1=new Display();
Display d2=new Display();
MyThread t1=new MyThread(d1,"yuvraj");
MyThread t2=new MyThread(d2,"dhoni");
 t1.start();
 t2.start();
```

In the above case we get irregular output, because two different object and since the method is synchronized lock is applied w.r.t object and both the threads will start simulataneously on different java objects
due to which the output is "irregular". - no use of sync. even after applied.

Conclusion :

If multiple threads are operating on multiple objects then there is no impact of Syncronization.

If multiple threads are operating on same java objects then syncronized concept is required(applicable).

classlevel lock

==========

1. Every class in java has a unique level lock.
2. If a thread wants to execute static synchronized method then the thread requires "class level lock".
3. While a Thread executing any static synchronized method the remaining Threads are not allow to execute any static synchronized method of that class simultaneously.
4. But remaining Threads are allowed to execute normal synchronized methods, normal static methods, and normal instance methods simultaneously.
5. Class level lock and object lock both are different and there is no relationship between these two.

eg::

```
class X{
    static synchronized m1(){}//class level lock
        static synchronized m2(){}
           static m3(){}//no lock required
           synchronized m4(){}//object level lock
        m5(){}//no lock required
}
 t1=> m1() => class level lock applied and chance is given
 t2=> m2() => enter into waiting state
 t3=> m3() => gets a chance for execution without any lock
 t4=> m4() => object level lock applied and chance is given
 t5=> m5() => gets a chance for execution without any lock

eg#1.
class Display{
    public synchronized void displayNumbers(){
        for (int i=1;i<=10 ;i++ )
        {
            System.out.print(i);
            try{
                Thread.sleep(2000);
            }
            catch (InterruptedException e){
            }
```

```
        }
    }
    public synchronized void displayCharacters(){
        for (int i=65;i<=75 ;i++ )
        {
            System.out.print((char)i);
            try{
                Thread.sleep(2000);
            }
            catch (InterruptedException e){
            }

        }
    }
}


class MyThread1 extends Thread{
    Display d;
    MyThread1(Display d){
        this.d=d;
    }
    @Override
    public void run(){
        d.displayNumbers();
    }
}
class MyThread2 extends Thread{
    Display d;
    MyThread2(Display d){
        this.d=d;
    }
    @Override
    public void run(){
        d.displayCharacters();
    }
}
public class Test3 {
    public static void main(String... args){
        Display d1=new Display();
        MyThread1 t1= new MyThread1(d1);
        MyThread2 t2= new MyThread2(d1);
        t1.start();
        t2.start();
    }
}
```

```
Output::
3 Threads
  a.MainThread
  b.userdefinedThread
    displayCharacters()
  c.userdefinedThread
displayNumbers()
```

Synchronized block — method level / block level

==========

lock — class level / obj level

```
synchronized void m1(){
    ...
    ...
    ======
```

```
    ======
    ...
    ...
}
```

if few lines of code is required to get synchronized then it is not recomeneded to make method only as synchronized.

If we do this then for threads performance will be low, to resolve this problem we use "synchronized block", due to synchronized block performance will be improved.

Case Study

==========

If a thread got a lock of current object, then it is allowed to execute that block

a.

```
synchronized(this){
    .....
}
```

To get a lock of particular object:: B

b.

```
synchronized(B){
    .....
}
```

If a thread got a lock of particular object B, then it is allowed to execute that block.

c. To get class level lock we have to declare synchronized block as follow

```
synchronized(Display.class){
    ....
 }
```

If a thread gets class level lock, then it is allowed to execute that block

7th dec

08 December 2022 00:31

synchronized block

==========

eg#1.

```
class Display{
    public void wish(String name){
        ;;;;;;;;;;;;; //1-Lakh lines of code
    synchronized(this){
        for (int i=1;i<=10;i++ )
        {
            System.out.print("Good morning:");
            try{
                Thread.sleep(2000);
            }
            catch (InterruptedException e){}
            System.out.println(name);
        }
    }
        ;;;;;;;;;;;;;//1-Lakh lines of code
    }
}
class MyThread extends Thread{
    Display d;
    String name;

    MyThread(Display d,String name){
        this.d=d;
        this.name=name;
    }
    public void run(){
        d.wish(name);
    }
}
class Test {
    public static void main(String[] args) {
        Display d=new Display();
        MyThread t1=new MyThread(d,"dhoni");
        MyThread t2=new MyThread(d,"yuvi");
        t1.start();
        t2.start();
    }
}
```

obj level

Good morning: dhoni
: dhoni
≡
: yuvi
≡

o/p: same obj, regular. "

eg#3.

```
class Display{
    public void wish(String name){
        ;;;;;;;;;;;;; //1-Lakh lines of code
    synchronized(Display.class){
        for (int i=1;i<=10;i++ )
        {
            System.out.print("Good morning:");
            try{
                Thread.sleep(2000);
            }
            catch (InterruptedException e){}
            System.out.println(name);
        }
    }
        ;;;;;;;;;;;;;//1-Lakh lines of code
    }
}
```

Class level

eg#2.

```
class Display{
    public void wish(String name){
        ;;;;;;;;;;;;; //1-Lakh lines of code
    synchronized(this){
        for (int i=1;i<=10;i++ )
        {
            System.out.print("Good morning:");
            try{
                Thread.sleep(2000);
            }
            catch (InterruptedException e){}
            System.out.println(name);
        }
    }
        ;;;;;;;;;;;;;//1-Lakh lines of code
    }
}
class MyThread extends Thread{
    Display d;
    String name;

    MyThread(Display d,String name){
        this.d=d;
        this.name=name;
    }
    public void run(){
        d.wish(name);
    }
}
public class Test {
    public static void main(String[] args) {
        Display d1=new Display();
        Display d2=new Display();
        MyThread t1=new MyThread(d1,"dhoni");
        MyThread t2=new MyThread(d2,"yuvi");
        t1.start();
        t2.start();
    }
}
```

obj level

Diff obj

Output::Irregular output becoz two object and two threads acting on two different objects

```
class MyThread extends Thread{
    Display d;
    String name;

    MyThread(Display d,String name){
        this.d=d;
        this.name=name;
    }
    public void run(){
        d.wish(name);
    }
}
public class Test {
    public static void main(String[] args) {
        Display d1=new Display();
        Display d2=new Display();
        MyThread t1=new MyThread(d1,"dhoni");
        MyThread t2=new MyThread(d2,"yuvi");
        t1.start();
        t2.start();
    }
}
```

Note:: 2 object, 2 thread, but the thread which gets a chance applied class level lock so output is regular.

Note:: lock concept applicable only for objects and class types, but not for primitive types. if we try to do it would result in compile time error saying "unexpected type".

eg:: int x=10;

synchronized(x){//CE: unexpected type found:int required:reference

}

InterThreadCommunication(remember postbox example)

==========

Two threads can communicate each other with the help of

a. notify()
b. notifyAll()
c. wait()

notify()=> Thread which is performing updation should call notify(),so the waiting thread will get notification so it will continue with its execution with the updated items.

wait() => Thread which is expecting notification/updation should call wait(), immediately the Thread will enter into waiting state.

If a thread wants to call wait(),notify()/notifyall() then compulsoily the thread should be the owner of the object otherwise it would result in "IllegalMonitorStateException".

We say thread to be owner of that object if thread has lock of that object.

It means these methods are part of synchronized block or synchronized method, if we try to use outside synchronized area then it would result in RunTimeException called "IllegalMonitorStateException".

if a thread calls wait() on any object, then first it immediately releases the lock on that object and it enters into waiting state.

if a thread calls notify() on any object,then he may or may not release the lock on that object immediately.

Except wait(),notify(),notifyAll() lock can't be relased by other methods.

Note::

yield(),sleep(),join() => can't release the lock.

wait(),notify(),notifyAll() => will release the lock,otherwise interthread communication can't happen.

Once a Thread calls wait(), notify(), notifyAll() methods on any object then it releases the lock of that particular object but

not all locks it has.

Method prototype of wait(),notify(),notifyAll()

1. public final void wait()throws InterruptedException
2. public final native void wait(long ms)throws InterruptedException
3. public final void wait(long ms,int ns)throws InterruptedException
4. public final native void notify()
5. public final void notifyAll()

Interview Question

==========

Method like wait(),notify(),notifyAll() are present inside Object class, y not in Thread class?

Thread will call wait(),notify(),notifyAll() on Objects like PostBox,Stack,Customer,Student,....

=> obj.wait(),obj.notify(),obj.notifyAll()

These methods should be available for every object in java,if the method has to available for every object in java then those

methods should come from "Object" class.

Program

=======

eg#1.

```
class ThreadB extends Thread{
    int total =0;
    @Override
    public void run(){
        for (int i=0;i<=100 ; i++){
            total+=i;
        }
    }
}
public class Test {
    public static void main(String[] args)throws InterruptedException {
        ThreadB b=new ThreadB();
        b.start();
        stmt-1;
        System.out.println(b.total);
    }
```

}

A. stmt-1

if i replace with Thread.sleep(10000) then thread will enter into waiting statement

but within 1ns only the updation value is ready.

with in 10 sec if the updation is not ready, then we should not use Thread.sleep(10000)

B. stmt-2

if i replace with b.join(), then main thread will enter into waiting state,then child will execute for loop,till then main thread has to wait.

main thread is waiting for updation result.

```
for (int i=0;i<=100 ; i++){
        total+=i;
}
//1cr lines of code is available
```

drawback "why we should not use sleep() & join()"

main thread has to wait till 1 cr lines of code,y main thread should wait for the complete code to finish.

eg#2.

```
class ThreadB extends Thread{
    int total =0;
  @Override
  public void run(){
    synchronized(this){
        System.out.println("Child thread started calculation");
        for (int i=0;i<=100 ; i++){
            total+=i;
        }
        System.out.println("Child thread trying to give notification");
        this.notify();
    }
  }
}
public class Test {
    public static void main(String[] args)throws InterruptedException {
        ThreadB b=new ThreadB();
        b.start();
        Thread.sleep(10000);//10sec
    synchronized(b){
        System.out.println("Main thread is calling wait on B object");
        b.wait();
        System.out.println("Main thread got notification");
        System.out.println(b.total);
    }
   }
}
```

Output

```
======
Child thread started calculation
Child thread trying to give notification
Main thread is calling wait on B object
```

becoz of Thread.sleep(10000) main thread will never get notification.

eg#3.

```
class ThreadB extends Thread{
    int total =0;
  @Override
  public void run(){
    synchronized(this){
        System.out.println("Child thread started calculation");
        for (int i=0;i<=100 ; i++){
            total+=i;
        }
        System.out.println("Child thread trying to give notification");
        this.notify();
    }
  }
}
public class Test {
    public static void main(String[] args)throws InterruptedException {
        ThreadB b=new ThreadB();
        b.start();
    synchronized(b){
        System.out.println("Main thread is calling wait on B object");
        b.wait(10000);//10sec
        System.out.println("Main thread got notification");
        System.out.println(b.total);
    }
  }
}
```

Output

```
======
Child thread started calculation
Child thread trying to give notification
Main thread is calling wait on B object for 10sec
Main thread got notification
5050
```

ProducerConsumer Problem

==========

Producer => produce the item and update in the Queue

Consumer => consume the item from the Queue

```
class Producer extends Thread{
    Producer(){
        synchronized(q){
            produce the item and update it to queue
            q.notify();
        }
    }
}
class Consumer extends Thread{
    Consumer(){
        synchronized(q){
            if(q is empty){
                q.wait();
            }else{
                consume the item from the queue
            }
        }
    }
}
```

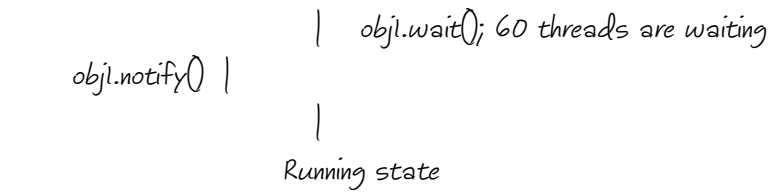
Difference b/w notify and notifyAll()

notify() => To give notification only for one waiting thread

notifyAll() => To give notification for many waiting thread

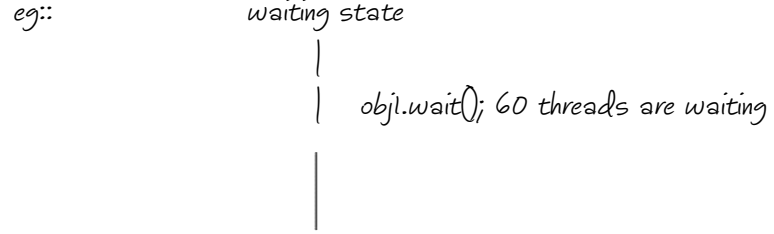
=> We can use notify() method to give notification for only one Thread. If multiple Threads are waiting then only one Thread will get the chance and remaining Threads has to wait for further notification.

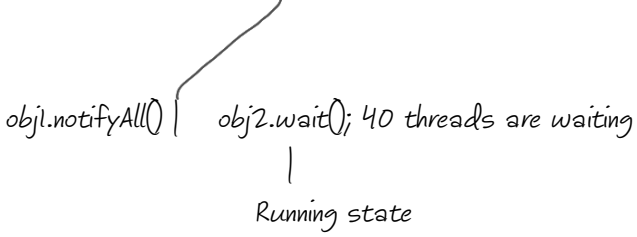
But which Thread will be notify(inform) we can't expect exactly it depends on JVM.



Among 60 threads which thread will get a chance we don't have control over that it is decided by JVM(threadscheduler).

=> We can use notifyAll() method to give the notification for all waiting Threads of particular object. All waiting Threads will be notified and will be executed one by one, because they required lock.





Note: On which object we are calling wait(), notify() and notifyAll() methods that corresponding object lock we have to get but not other object locks.

```
eg:: Stack s1=new Stack();
     Stack s2=new Stack();

synchronized(s1){
    s2.wait();//RE: IllegalMonitorStateException
}
synchronized(s2){
    s2.wait();(valid)
}
```

Question based on lock

==========

1. If a thread calls wait() immediately it will enter into waiting state without releasing any lock(false)
2. If a thread calls wait() it releases the lock of that object but may not immediately (false)
3. If a thread calls wait() on any object,it releases all locks acquired by that thread and enters into waiting state(false)
4. If a thread calls wait() on any object,it immediately releases the lock of that particular object and entered into
   waiting state(true).
5. If a thread calls notify() on any object,it immediately releases the lock of that particular object(invalid)
6. If a thread calls notify() on any object,it releases the lock of that object but may not immediately(true)

8th notes

09 December 2022 12:45

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".

live lock: lock be live after some time.

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.

8th Summary

09 December 2022 15:33

```java
interface Computer{
 void compile();
}
class Laptop implements Computer{
    public void compile() {
        System.out.println("lapy run");
    }
}
class Desktop implements Computer{
    public void compile() {
        System.out.println("desktop run");
    }
}
class Developer {
    public void code(Computer lp) {
        lp.compile();
    }
}
public class Summary {
    public static void main(String[] args) {
        Computer lp = new Laptop();
        Developer d = new Developer();

        d.code(lp);// lapy run

        // Now the company say we'll only provide desktop.So you should be able to work
        // with it also...(for that we use loose Coupling)

        Computer c = new Desktop();
        d.code(c);//desktop run

        //After getting whichever machine i want to customize it with my Compatibility..
        //It's for my purpose only (Anonymous class)\

        Computer c1 = new Computer(){
            @Override
            public void compile(){
                System.out.println("Customizing...");
            }
        };

        d.code(c1);//Customizing…

        Computer c2 = ()->System.out.println("Customized");

        d.code(c2);//Customized
    }
}
```