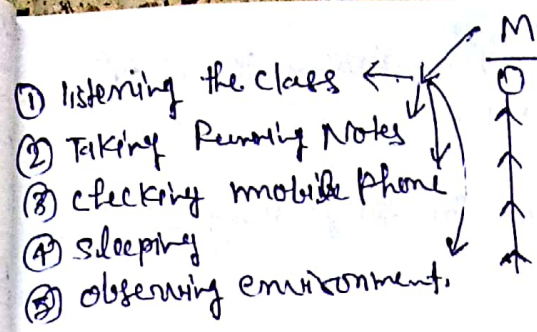


executing several task simultaneously is a concept of multitasking. there are 2 types of multitasking.

Multitasking Divided in two Parts

(1) Process Based (OS Level) (2) Thread Based (Programing level)

(a) Typing a Java Program in editor
(b) Listen audio songs from same System
(c) Download a file from Net

Process Based Multitasking :- executing several task simultaneously where each task is seprate independent program (Process) is called process based multitasking

Example -> while typing a Java program in the editor we can listen audio song from the same system at the same time we can download a file from net, all these task will be executed simultaneously and independent of each other hence it is Process based multitasking. Process based multitasking is best suitable at OS Level.

Thread Based Multitasking :- executing several task simultaneously where Each task is seprate independent Path of same program is called thread based multitasking and Each independent path is called a thread. thread based multitasking is best suitable at programatical level.

whether it is process based or thread based the main objective of multitask is to reduce response time of the system and to improve perfermence.

The main important application areas of multithreading are

(1) To Develop multimedia graphics
(2) To Develop animations
(3) To Develop video games
(4) To Develop Web servers & Application servers etc

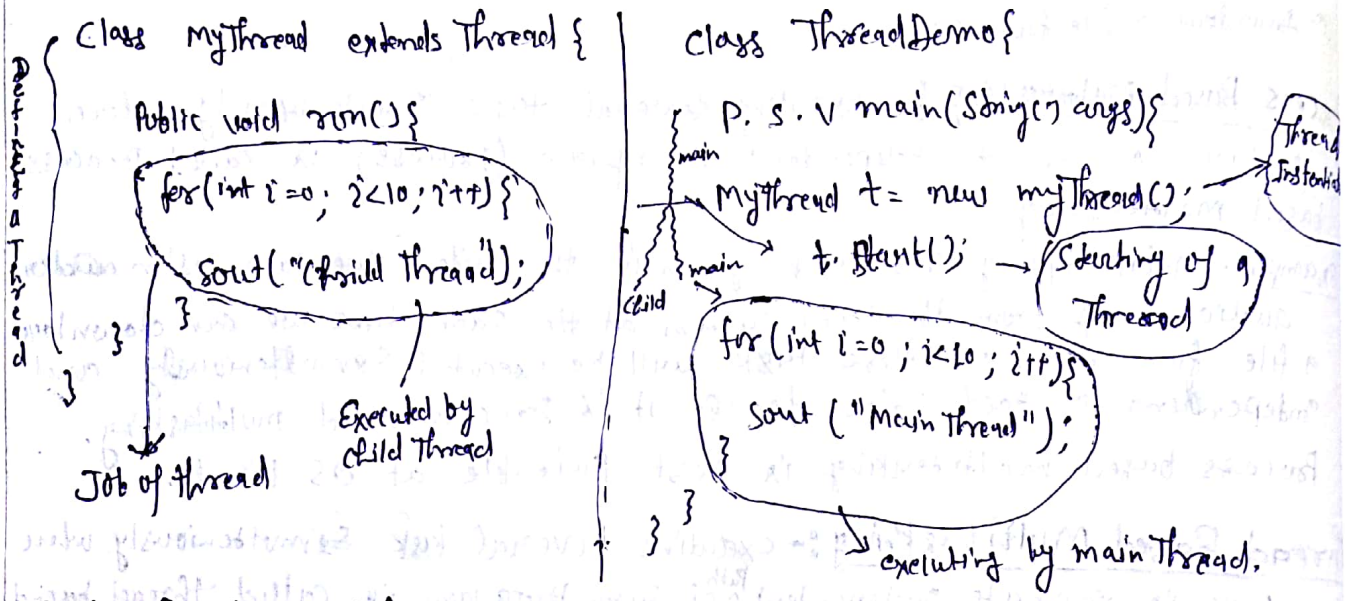
When compared with old languages Developing multithreaded Applic in Java is very Easy because Java Provide inbuild support for multithreading with Rich API [Thread, Runnable, Thread Group....]

Thread :- Thread is a flow of Execution. Every thread is a Separate independent Job is there.

Defining A Thread :- we can Define a thread in the following 2 ways.

(1) By Extending thread class
(2) By implementing Runnable interface

(1) By Extending Thread Class :-



Note:- Thread Scheduler :-

- It is the Part of JVM, It is responsible to Schedule threads that is if multiple threads are waiting to get ~~change~~ chance of execution then in which order threads will be executed is decided by thread Scheduler.
- we can't expect exact algorithm by thread Scheduler it is varied from JVM To JVM hence we can't expect threads execution order and exact output.

hence whenever situation comes Multithreading there is no guranties for exact output but we can provide several possible outputs.

The following are various possible outputs for the above program.

P-1	P-2	P-3	P-4
Main Thread Main thread : : Child thread Child thread :	Child Thread Child Thread : : main Thread main Thread {	main Thread child Thread main Thread child Thread : : : :	child Thread Child Thread Main Thread main Thread : : : :

Case 2:- Difference between t.start(); & t.run(); method :-

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 t.run() a new thread will not be created and run method will be executed Just like a normal method called by main thread.

Hence in the above program if I replace t.start() with t.run() then the output is

```
child thread
child thread
   :
   :
main thread
main thread
```

This total output produced by only main thread.

Case 3:- Importance of Thread class start() method :-

Thread class start() method is responsible to registered the thread with Thread scheduler and all other mandatory activities hence without executing thread class start() method there is no chance of starting a new thread in Java Due to this thread class start() method considered as heart of multithreading.

```
start () {
  1. Register this thread with thread scheduler
  2. Perform all other mandatory activities
  3. Invoke run(); method.
```

Case 4:- Overloading of run() method :-

```
class myThread extends Thread {
  Public void run(){
    Sout ("no args run");
  }
  Public void run(int i){
    Sout ("int arg run");
  }
}
```

(overloading run methods)

```
class ThreadDemo {
  P S V m (String[] args) {
    myThread t = new myThread();
    t.run(); t.start();
  }
}
```

Output:- no args run

- Overloading of run() method is always possible but thread class start() method can invoke no args run() method the other overloaded method we have to call explicitly like a normal method call.

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

If we are not overriding run() method then thread class run method will be executed which has empty implimentation hence we will not get any output.

```
- Class MyThread extends Thread {
  }
- Class Test {
    P S V m (String[] args) {
      MyThread t = new MyThread();
        t.start();
    }
  }
```

output :- no output.

Note : It is highly recommended to override run method otherwise don't go for multithreading concept.

Case 6 :- Overriding of start() method.

```
- Class MyThread extends Thread {
    Public void start(){
    SOut ("start method");
  }
  Public void run(){
    SOUT ("run method");
  }
}
- Class Test {
    Public S V m (String[] args) {
      MyThread t = new MyThread();
        t.start();
      SOUT ("main thread");
  }
}
```

Total output produced by only main thread.

output :- start method
main thread

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

Case:-7 super.start():

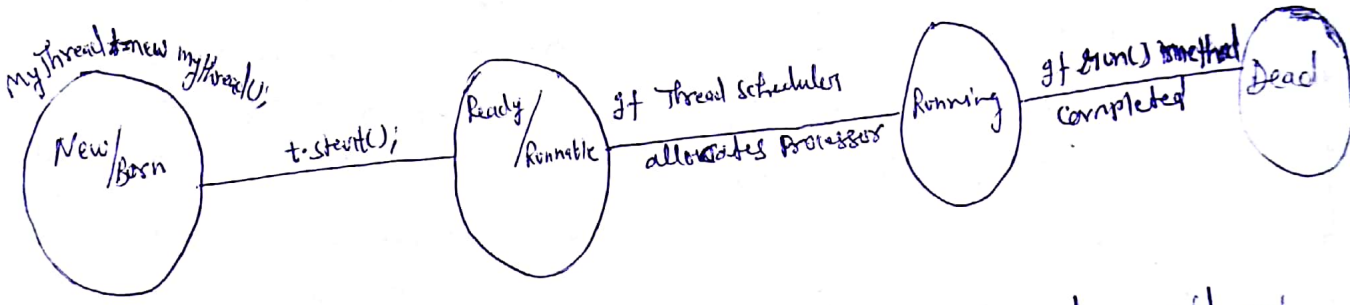
```
class myThread extends Thread{
    Public void start(){
        super.start();
        SOUT("start method");
    }
    Public void run(){
        Sout("run method");
    }
}
```

```
Class Test{
    P.S.V m(String[] args){
        myThread t = new myThread();
        t.start();
        sout("main method");
    }
```

output:-

P-1	P-2	P-3
run method	start method	start method
start method	main method	run method
main method	run method	main method

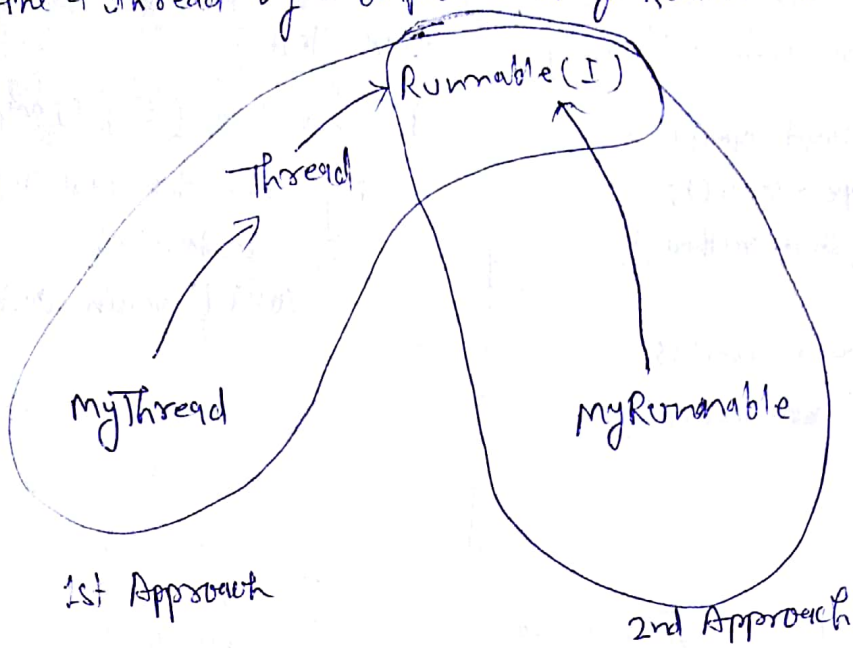
Case-8 Thread life cycle



Case-9 after starting a thread if we are trying to restart the same thread then we will get run time Exception saying IllegalThreadStateException.

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

We can define a thread by implementing Runnable interface



- Runnable Interface present in java.lang package and it contain only method, (Run Method)

Defining a Thread

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

Job of a Thread — executed by child thread

```
class ThreadDemo
    P S V M (String[] args) {
MyRunnable r = new MyRunnable();
Thread t = new Thread(r);
t.start();
for(int i=0; i<10; i++){
    SOUT("main thread");
}
}
```

Target Runnable — executed by main Thread

- After completing this code we will get mixed output and we can't tell exact output.
- **Case study :-**

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

Case 1: t1.start(); A new thread will be created and which is responsible for execution of thread class run method which has Empty implementation.

Case-2: t1.run(); No new thread will be created and 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 MyRunnable class run method.

Case-4: t2.run(); A New thread will not be created & MyRunnable class run method will be executed just like a normal method call.

Case-5: r.start(); We will get compile time error saying MyRunnable class does'nt have start capability

```
CE: can'nt find Symbol
Symbol  method start()
location class MyRunnable
```

Case-6: r.run(); No new thread will be created & MyRunnable run method will be executed like normal method call

Which Approach is best to define a thread:-

- Among two ways of defining a thread implements Runnable approach is recommended.
- In the first approach our class always extends Thread class, there is chance of extending other class hence we are missing inheritence benifit.
- But in the second approace while implementing Runnable interface we can extend any other class. hence we will not miss any inheritence benifite.
- Because of above Reason implementing Runnable Interface approach is recommended then extending thread class.

Thread class Constructors :-

1. Thread t = new Thread();
2. Thread t = new Thread(Runnable r);
3. Thread t = new Thread(String name);
4. Thread t = new Thread(Runnable r, 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);

Another Approach to define a Thread (Not Recommended to use)

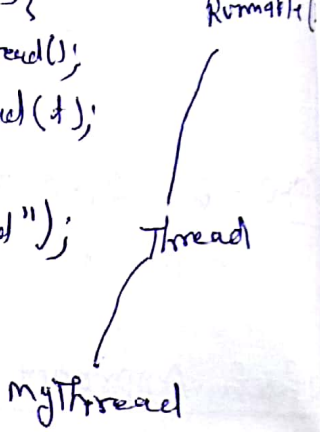
```
class MyThread extends Thread{
    public void run(){
        sout("child Thread");
    }
}
```

```
class ThreadDemo{
    P S V main(String[] args){
        MyThread t = new MyThread();
        Thread t1 = new Thread(t);
        t1.start();
        sout("main thread");
    }
```

It is hybrid approach

output;

P1	P2
child Thread	main Thread
main Thread	child Thread



getting & setting Name of a Thread :-

- Every Thread in Java has some name it may be default name generated by JVM OR Customised name provided by Programmer.
- We can get & set name of a thread by using following two methods of Thread class
  - Public final String getName();
  - Public final void setName();

```
class MyThread extends Thread {
}
class Test {
    P. S. V main(String[] args) {
        Sout(Thread.currentThread().getName());
        MyThread t = new MyThread();
        Sout(t.getName());
        Thread.currentThread().setName("Bawan Kumar");
        Sout(Thread.currentThread().getName());
    }
}
```

output :-

```
main
Thread-0
Bawan Kumar
```

- We can get current executing thread obj by using Thread.currentThread() method.

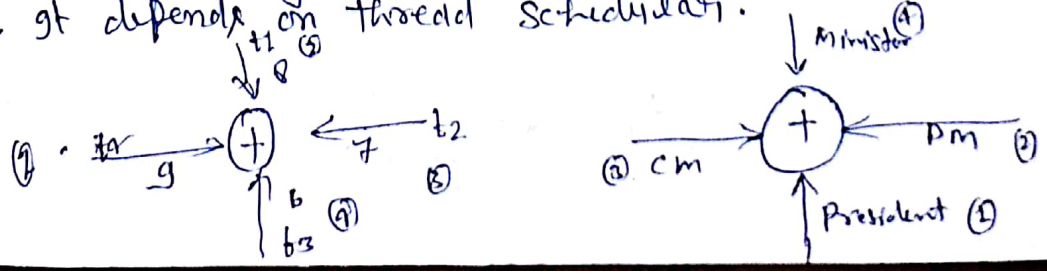
```
Class MyThread extends Thread {
    Public void run(){
      SOUT("run method executed by Thread "+Thread.currentThread().getName());
    }
}

Class Test {
    Ps v main(String[] args){
      myThread t = new MyThread();
        t.start();
    SOUT("main method executed by Thread "+ Thread.currentThread().getName());
    }
}
```

Output :- main thread executed by Thread : main
run method executed by thread : Thread-0

* 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 priority is 1 to 10 where 1 is Min priority & 10 is Max priority.
- Thread class define following constants to re-present some standard priorities.
  1. Thread.MIN_PRIORITY ⟶ 1
  2. Thread.NORM_PRIORITY ⟶ 5
  3. Thread.MAX_PRIORITY ⟶ 10

* Thread Scheduler will use priorities while allocating Processor.
* The Thread with is having highest priorities will get chance first.
* If Two Threads having same priorites then we can't expect execution order. It depends on thread Scheduler.



* Thread class defines the following methods to get & set priorities of a thread.

Public final int getPriority()

Public final void setPriority(int p)

Allowed values
Range: 1 to 10
otherwise RE: IllegalArgumentException.

Example:- t.setPriority(7); ✓ it is valid

t.setPriority(17); ✗ → RE: IllegalArgumentException.

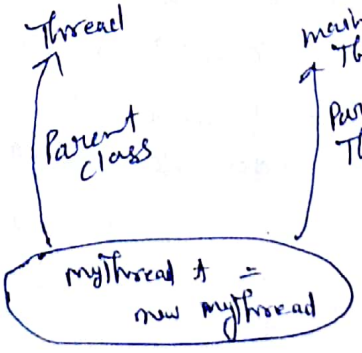
Default Priority :-

- The default priority only for the main thread is 5 but for all remaining thread default priority will be inherited from Parent to child that is whatever priority Parent thread has the same priority will be there for the child thread.

Example

```
class MyThread extends Thread{
}
class Test{
P S V main(String[] args){
SOut(Thread.currentThread().getPriorities()); //5
//Thread.currentThread().setPriorities(15); //RE: IllegalArgument
Thread.currentThread().setPriorities(7); ... (1)
MyThread t = new MyThread();
SOUT t = new myThread();
S.OUT(t.getPriorities); //7
}
}
```

* If we will comment line (1) then child Thread priorities will become 5.



Example

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

class ThreadPrioritiesDemo {
    p s v main(String[] args){
        MyThread t = new MyThread();
        t.setPriorities(10);  ——— (1)
        t.start();
        for(int i=0; i<10; i++){
            SOUT("main Thread");
        }
    }
}
```

* If we are commenting line (1) then both main & child thread have the same priority 5 and hence we can't expect execution order & exact output.
* If we are not commenting line (1) then main thread has a priority 5 and child thread has a priority 10 hence child thread will get the chance first followed by main thread. In this case output is

Output:-
```
Child Thread
Child Thread
    :
    : 10 times
Main Thread
Main Thread
    :
    : 10 times
```

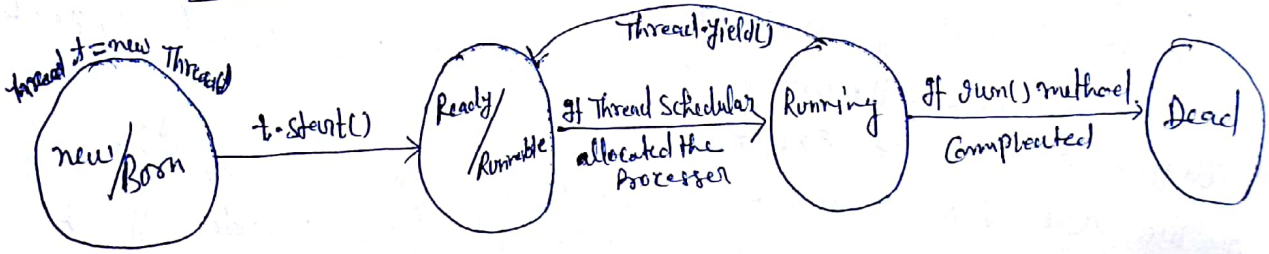
Note:- Some platforms don't provide proper support for thread priorities. for getting the proper support we should send the mail to microsoft to get the patch. but getting this patch we should use the licensed operating system.

* We can prevent a Thread execution by using the following methods
  1. yield method
  2. Join method
  3. sleep method

① **yield** :- This method causes to pause current executing thread to give the chance for waiting thread of same priority

* If there is no waiting thread OR All waiting Thread have Low priority Then same thread can continue its execution.
* If multiple thread are waiting with same priority then which waiting thread will get the chance we can't expect. It depends of thread scheduler.
* The thread which is yielded, when it will get the chance once again. It depends on thread scheduler and we can't expect exactly.

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




```
Class myThread extends Thread
{
  Public void run(){
    for(int i=0; i<10; i++){
      SOUT("Child Thread");
      Thread.yield(); ---> ①
    }
  }
}
```

```
Class ThreadYieldDemo{
  P S. V main(String[] args){
    myThread t = new myThread();
    t.start();
    for(int i=0; i<10; i++){
      SOUT("main Thread");
    }
  }
}
```

* In the above program if we are commenting line 1 then both thread will be executed simultaneously and we can't expect which thread will complete first.
* If we are not commenting line 1 then child thread allways call yield() method because of that main thread will get the chance more no. of times and the chance of completed first is high.

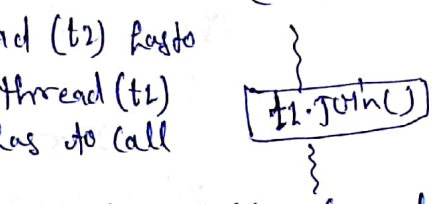
Note! Some platforms don't provide proper support for yield method.

Join Method :→ If a thread wants to wait untill completing some other thread then we should go for Join method

- for example if a thread t_1 wants to wait untill completing t_2 then t_1 has to call (t_2.Join).
- If it execute t_2.Join() then immidiately t_1 will be entered in to waiting states. untill t_2 complete.
- once t_2 completes then t_1 can continue its execution.

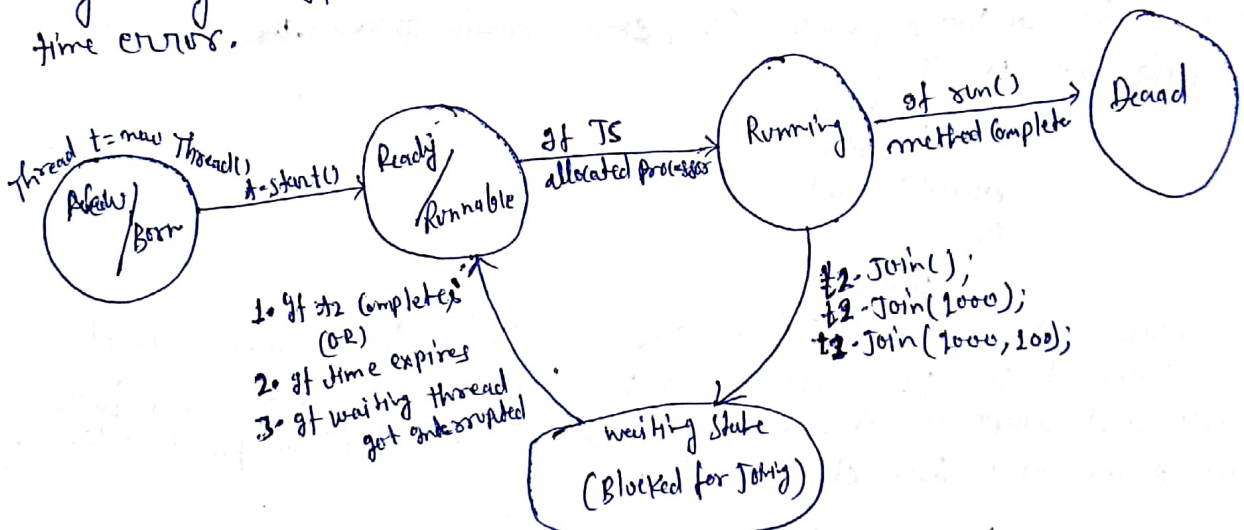
Example :→ venue fixing Activity (t_1) | wedding cards Printing (t_2) | Wedding cards distribution (t_3)

- wedding cards printing thread (t_2) has to wait untill venue fixing thread (t_1) Completion hence t_2 has to call t_1.Join()
- wedding cards distribution thread (t_3) has to wait untill weddings cards printing thread (t_2) Completion hence t_3 has to call t_2.Join()



- Public final void Join() throws InterruptedException
- Public final void Join(long ms) throws InterruptedException
- Public final void Join(long ms, int ns) throws InterruptedException

Note: Every Join method throws InterruptedException which is checked exception hence compulsory we should handle this exception either by using try/catch or throws keyword otherwise we will get Compile time error.




```
Class MyThread extends Thread
{
    Public void run(){
    for(int i=0; i<10; i++){
      SOUT("Child thread");
        try{ Thread.sleep(1000);
        }catch(IE e){ }
    }
  }
}
```

```
Class ThreadJoinDemo {
  P s v main(String[] args) throws IE {
    MyThread t = new MyThread();
      t.start();
      t.Join();  ------> (1)
    for(int i=0; i<10; i++){
      SOUT("Main Thread");
    }
  }
}
```

* If we Comment line-1 then both main & child thread will be executed Simultenously and we can't expect exact output.
* If we are not commenting line-1 then main thread calls Join method on child thread object hence main thread will wait untill completing child thread so. this case output is

```
Seetha Thread
Seetha Thread
    : 10 times
Rama thread
Rama thread
    : 10 times
```

Exp:2 Waiting of child thread until competing main thread. (Case-2)

```
class MyThread extends Thread {
    Static Thread mt;
    Public void run() {
        try {
            mt.Join();
        }
        catch (InterruptedException e) {}
        for (int i = 0; i<10; i++) {
        SOUT ("child Thread");
    }
}
}
```

```
Class ThreadJoinDemo1{
    Public static void main(String[] args) throws
        InterruptedException {
    MyThread.mt = Thread.currentThread();
    MyThread t = new MyThread();
        t.start();
    for (int i=0; i<10; i++) {
        SOUT ("Main Thread");
        Thread.sleep (2000);
    }
}
```

In the above Exp. child thread calls on main thread object hence child thread has to wait untill completing main thread. so in this case output is:-

```
main Thread
main Thread
    : 10 times
child Thread
child Thread
    : 10 times
```

Case 3:- If main thread calls Join method on child thread object and child thread calls Join method on main thread object then both threads will wait forever and the program will be paused (stuck) (this is something like dead lock).

Case 4:- If a thread calls Join method on the same thread itself then the program will be stucked (This is something like deadlock) in this case thread has to wait infinite amount of time.

Exp-

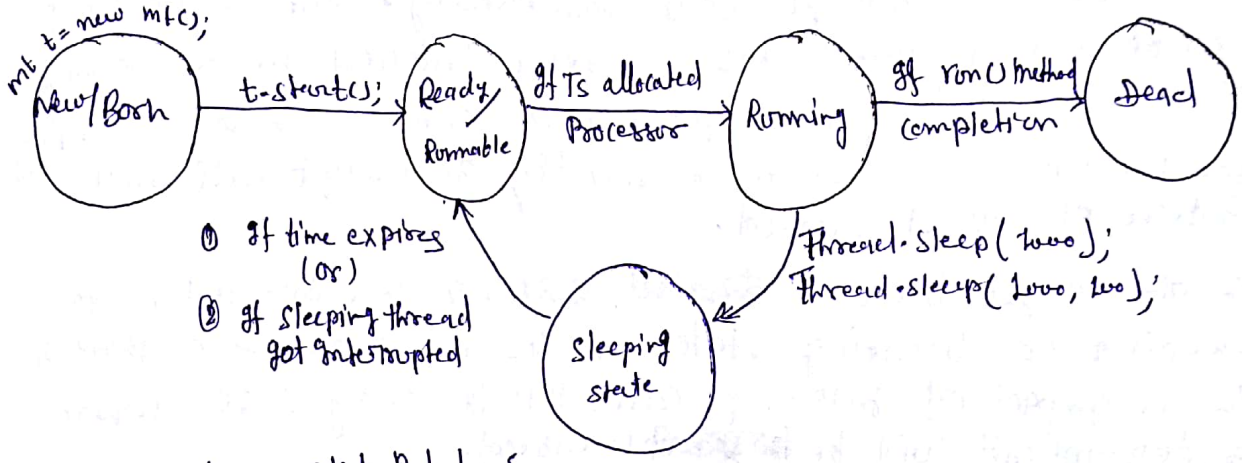
```
Class Test {
    Public static void main (String [] args) throws
        InterruptedException {
        Thread.currentThread().Join();
    }
}
```

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

InterruptedException

- Public static native void sleep(long ms) throws IE
- Public static void sleep(long ms, int ns) throws IE

Note:- Every sleep method throws Interrupted Exception, which is checked Exception hence whenever we are using sleep method compulsory we should handle Interrupted exception either by try/catch or throws keyword otherwise we will get Compiletime Error.



Ex:-

```
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);
    }
  }
}
```

- **How a thread can interrupt another thread?**

A Thread can interrupted 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++){
        System.out.println("I am Lazy Thread");
        Thread.sleep(2000);
      }
    }
    catch(InterruptedException e){
      S.O.p("I got Interrupted");
    }
  }
}
```

```
Class ThreadSleepDemo{
  Public static void main(String[] args)
  {
    MyThread t = new MyThread();
    t.start();
    t.interrupt();  ——— (1)
    System.out.println("End of main Thread");
  }
}
```

* If we comment line 1 (t.interrupt()) then main thread will not interrupt child thread. In this case child thread will execute for loop 10 times.
* If we will not comment line 1 (t.interrupt) then main thread will interrupt child thread. In this case output is.

End of main thread
I am lazy thread
I got interrupted.

* Whenever we are calling interrupt method if the target thread not in sleep or waiting state then there is no impact of interrupt call immediately. Interrupt call will be waited unfill target thread entered in to sleeping or waiting state. If target thread entered in to sleeping or waiting state then immediately interrupt call will interrupt target thread.
* If the target thread never entered in to sleeping or waiting state in its life time then there is no impact of interrupt call. this is only case where interrupt call will be waisted.

```
Class MyThread extends Thread
{
Public void run(){
 for(int i=0; i<=10000; i++){
  S.O.P("I am Lazy Thread-"+i);
 }
 S.o.p("I am entering into sleeping
                    state.");
 try{
    Thread.sleep(10000);
  }
  Cateh(InterruptedException e){
    S.o.p("I got Interrupted");
  }
}}
  Class ThreadSleepDemo1{
  P.S.V.m(String[] args){
    MyThread t= new MyThread();
      t.start();
      t.Interrupted();
  }}  S.O.P("End of mainthread");}
```

* In this Example interrupted call waited untill child thread complete for loop 10000 times.

Property	yield()	Join()	Sleep()
Purpose ?	If a thread wants to pause its execution to give the chances for remaing thread of same priority then we should go for yield() method	If a thread wants do wait until Completing some other thread then we should go for join method.	If a thread don't want to perform any operation for particular amount of time then we go for Sleep method.
1) Is it overloaded?	NO	Yes	Yes
2) Is it final ?	NO	Yes	NO
3) Is it throws IE?	NO	Yes	Yes
4) Is it native?	Yes	NO	sleep(long ms) → native sleep(long ms, int ns) → non-native
5) Is it static ?	Yes	NO	Yes

synchronization

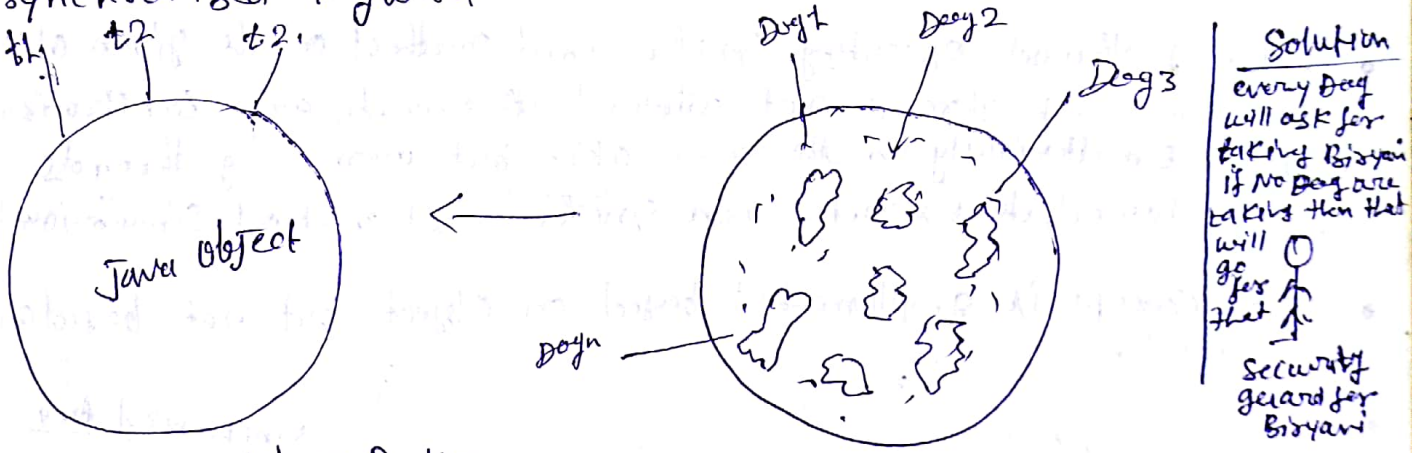
synchronized is the modifier & it is only aplicable on Methods & Block but not for classes & variables.

If multiple threads are trying to operates symultaniously on the same Java object then there may be chanched a Data Inconsistency Problem.

To overcome this problem we should go for synchronized keyword.

If a Method & Block declared as synchronized then at a time only one thread allow to execut that method or Block on the given object. So that Data Inconsistency problem will be resolved.

The main Advantage of Synchronized Keyword is we can resolved Data Inconsistency problem. but main Dis-advantage of synchronized keyword is it increases waiting times of thread and it creates performance problem hence if there is no specifice requirement then it is not recommended to use synchronized keyword.

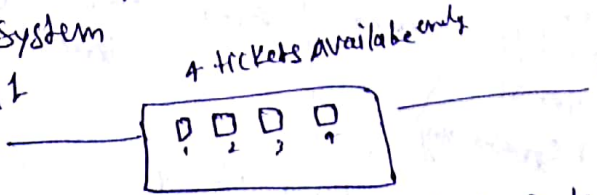


: Data Inconsistency Problem (in Java Programming)

Biryani - Inconsistency Problem. (in Real World)

Examples

① Railway Reservation System



- 2 customers are trying to book seats at same time. then there will be Data inconsistency Problem, to resolve this Problem we overcome for synchronized keyword.

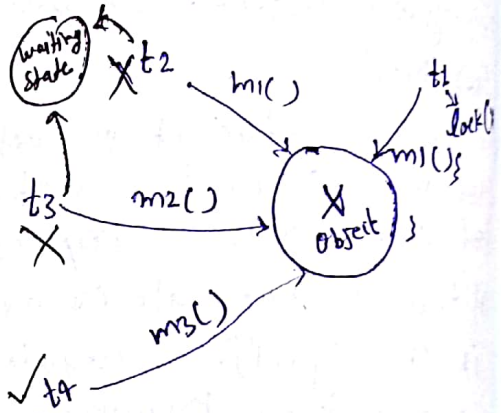
② Joints A/C

wife trying to withdraw money by using ATM & Husband trying to pay bill online on same bank A/C object on same time (simultaniously). So there may be Data Inconsistency Problem, to resolve this Problem we overcome for synchronized keyword.

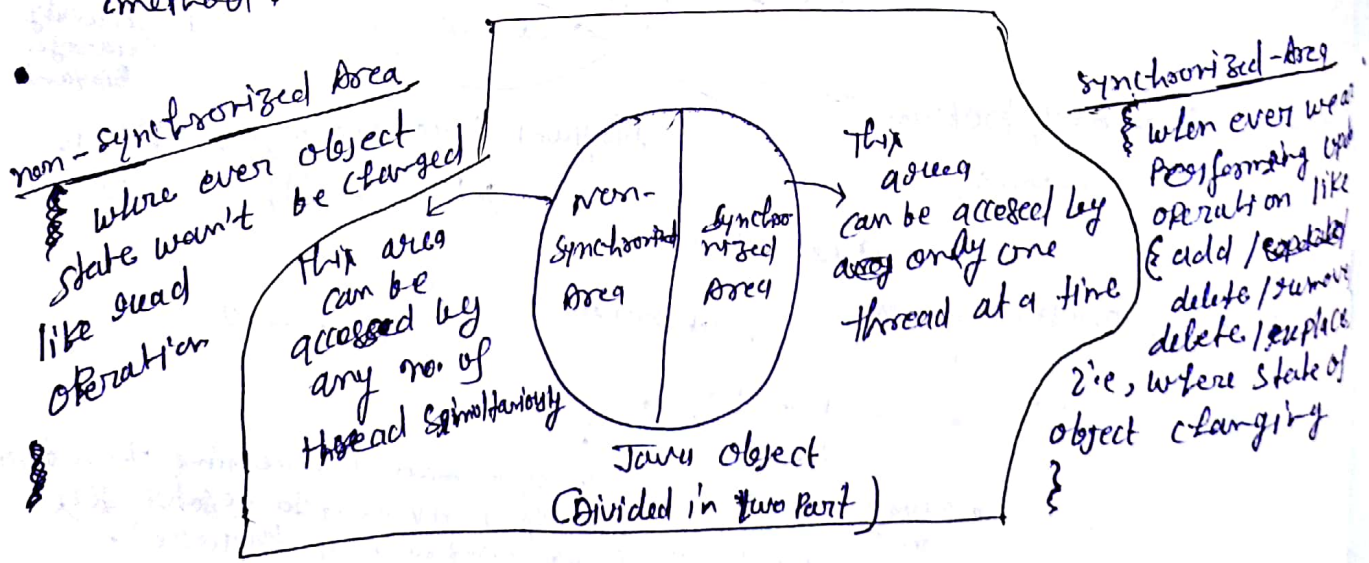
- Internally Synchronization concept is implemented by using lock, every obj in Java has a unique lock.
- Whenever we are using synchronized keyword then only lock concept will come in to the picture.
- If a thread wants to execute synchronized method on the given obj first it has to get lock of that object, once thread gets the lock then it is allowed to execute any synchronized method on that obj. once method execution complete. Automatically thread releases the lock.
- Aquiring & releasing the lock internally takes care by JVM & Programmer is not responsible for this activity.

Ex→

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



- While a thread executing synchronized method on the given obj the remaining thread not allowed to execute any synchronized method simultaniously on the same obj. but remaining threads are allowed to execute non synchronized method simultaniously.
- lock concept is implemented based on object but not based on method.
-



Example :-

```
class ReservationSystem {
Non synchronized  check Availability of tickets()
        {
            ≡ Just Read operation
        }
Synchronized  bookTickets()
        {
            ≡ Update Operation
        }
}
```

(3)

- if you are not declaring wish() method as synchronized then both thread will be executed simultaneously & hence we will get irregular output:-

Good Morning : Good Morning : YuvRaj
Good Morning : Dhoni
Good Morning : YuvRaj
Good Morning : Dhoni
Good Morning ; ,,

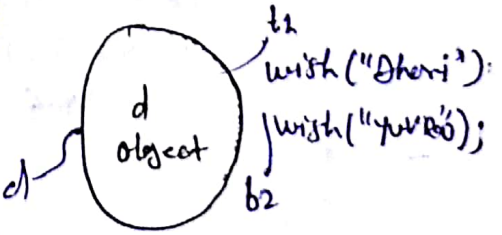
- if we declare wish method as synchronized then at a time only one thread is allowed executed wish method on the given display object hence we will get regular output:

Good Morning : Dhoni
Good Morning : Dhoni
- - - - - - - - - -
10 times
Good Morning : YuvRaj
Good Morning : YuvRaj
- - - - - - - - - -
10 times

```
class Display {
Public synchronized void wish(String name){
    for(int i=0; i<10; i++){
      System.out.Print("Good Morning");
    try {
        Thread.Sleep(2000);
      } Catch (InterruptedException e) { }
      System.out.Println(name);
    }
} }
```

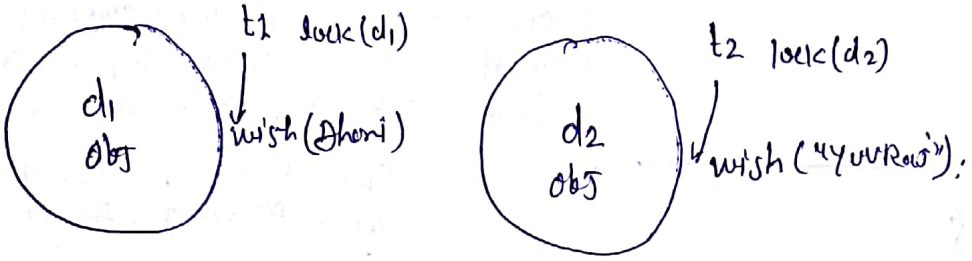
```
class MyThread extends Thread {
    Display d;
    String name;
    MyThread (Display d, String name) {
        this.d = d;
        this.name = name;
    }
    Public void run(){
        d.wish(name);
    }
}
```

```
class SynchronizedDemo {
Public Static void main(String[] arg){
    Display d = new Display();
    MyThread t1 = new myThread(d, "Dhoni");
    MyThread t2 = new myThread(d, "YuvRaj");
        t1.Start();
        t2.Start();
    }
}
```



Case Study :-

```
Display d1 = new Display();
Display d2 = new Display();
MyThread t1 = new MyThread(d1, "Dhoni");
MyThread t2 = new MyThread(d2, "YuvRaj");

t1.start();
t2.start();
```



- even though wish method is synchronized we will get irregular output because threads are operating on different Java objects.

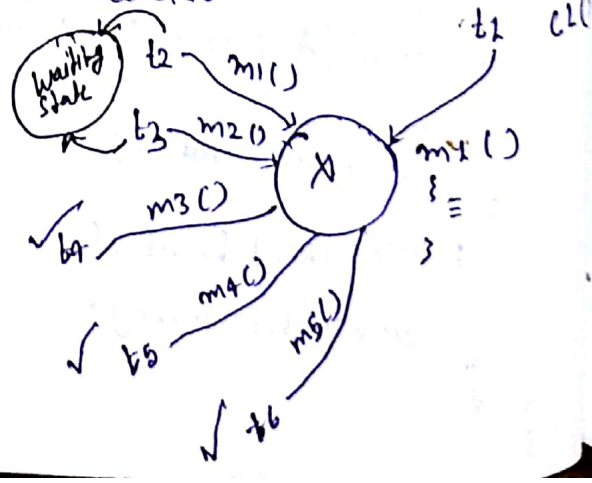
Conclusion:- If multiple threads are operating on same Java obj then synchronization is required.

- If multiple threads are operating on multiple Java object then synchronization is not required.

Class level lock:- every class in Java has a unique Lock which is nothing but class level lock.

- If a thread wants to execute static synchronized method then thread required class level lock. once threads get class level lock then it is allowed to execute any static synchronized method of that class. once method execution completes then automatically thread releases the lock.

```
class X {
    static synchronized m1(){ }
    static synchronized m2(){ }
    static m3(){ }
    synchronized m4(){ }
    m5(){ }
}
```



while a thread executing static synchronized method the remaining threads are not allowed to execute any static synchronized of that class simultaneously. but remaining threads are allowed to execute the following method simultaneously.

① normal static method
② synchronized Instance method
③ normal Instance method.

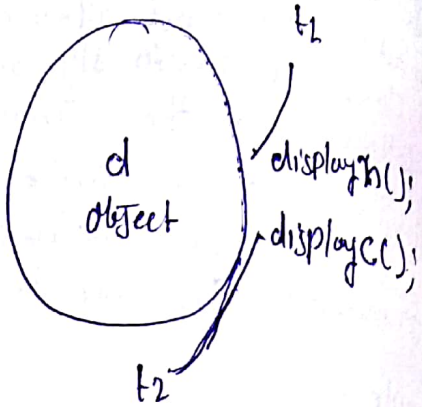
Example :-

```
class Display {
 public synchronized void displayn() {
    for (int i=1; i<=10; i++){
        System.out.print(i);
       try {
           Thread.sleep(2000);
        } catch (InterruptedException e){ }
       }
    }
 public synchronized void displayc() {
    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 (Disply d) {
       this.d=d;
    }
    public void run() {
        d.displayn();
    }
}
```

```
class MyThread2 extends Thread {
    Display d;
    MyThread2 (Display d) {
       this.d = d;
    }
    public void run() {
       d.displayc();
    }
}
```

```
Class SynchronizedDemo1 {
    Public static void main(String[] args) {
        Display d= new Display();
        MyThread1 t1 = new MyThread1(d);
        MyThread2 t2 = new MyThread2(d);
            t1.Start();
            t2.Start();
    }
}
```



Synchronized block :-

- If very few lines of the code required synchronization then it is not recomended to declare entire method as synchronized, but have to enclose those few lines of the code by using synchronized block. The main advantage of Synchronized block over Synchronized method is it reduce the waiting time of thread and improve the performances of the system or Application.

We can declare Synchronized block as follows :

① To get lock of Current Obj.

```
synchronized (this){
    ≡
}
```

→ If a thread get lock of current object then only it is allowed to execute this area.

② To get lock of Particuller Obj b.

```
Synchronized (b){
    ≡
}
```

→ If a thread get lock of Particuller Object 'b' then only it is allowed to execute this area.

③ To get lock of class

```
Synchronized (Display.
    ≡
}
```

→ If a thread get class level lock of Display class, then only it allowed to execute this area.

```
class Display {
 Public void wish(String name){
   ;;;;;;;// 1 Lakh lines of code.
  Synchronized (this) {
   for (int i=0; i<10; i++){
    System.out.Println("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 SynchronizedDemo{
  Public static void main(String[] args){
     Display d = new Display();
     MyThread t1 = new MyThread(d,"Dhoni");
     MyThread t2 = new MyThread(d,"YUVRaj");
         t1.start();
         t2.start();
}
```

Lock concept applicable for object types & class type. but not for Primitive hence we can't pass Premitives type as arguments to synchronized block otherwise we will get compile type Error saying "Unexpected type found int required reference.

```
int x = 10;
Synchronized (x){
    ⋮
}
```

CE: Unexpected type
found : int
required: reference

If multiple threads operating simultaneously on same java object then there may be a chance of Data in Consistency Problem. this is called Race Condition.

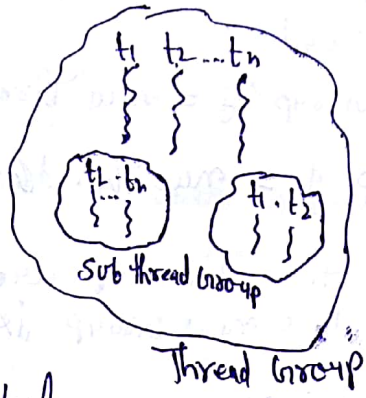
we can over come this problem by using Synchronized keyword

Thread Group

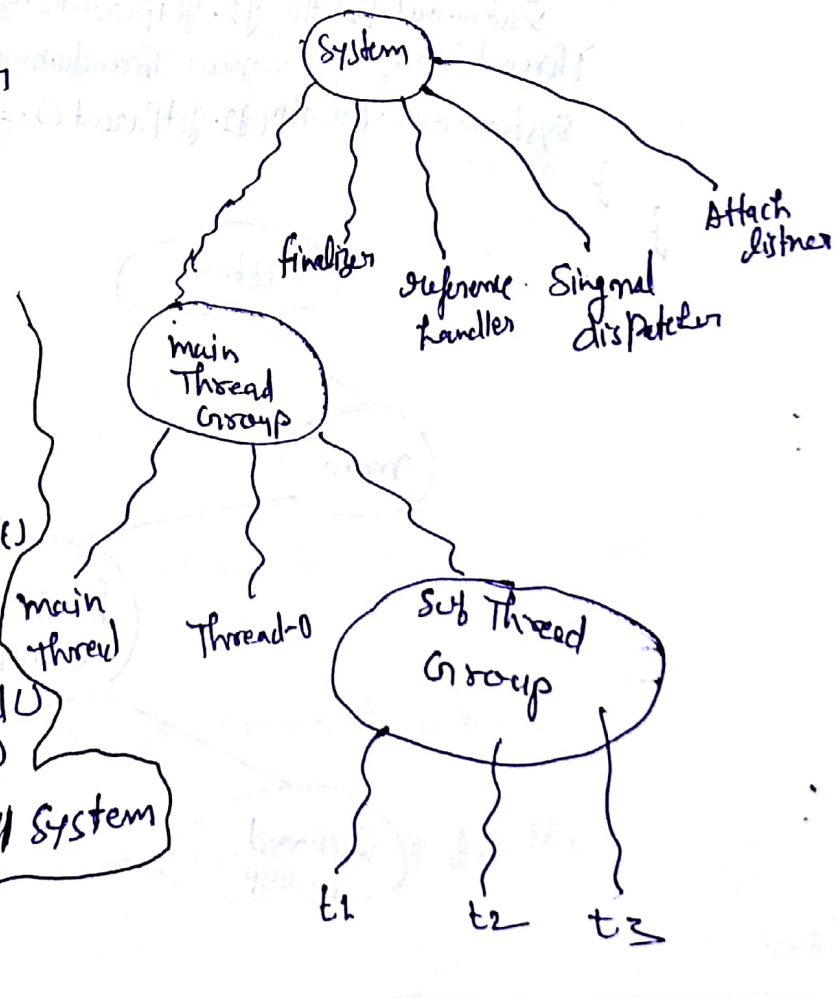
Java.Lang Package

- Based on functionality we can group thread in to a single unit which is nothing but thread Group. that is thread Group Contains a Group of threads.
- In addition to threads threadGroup can also contains Sub thread Groups.

- The main advantage of maintaining threads in the form of Thread Group is we can Perform Common Operation very Eatily.



- Every Thread in Java belongs to Some Group.
- main thread belongs to "main" Group.
- Every thread Group in Java is the child Group of "System" Group. Either directly or indirectly. Hence System group acts root for all thread Groups in Java.
- System Group Contains several system level threads like
  - → finalizer
  - → reference Handler
  - → Signal dispatcher
  - → Attach listner
  - ⋮



Example:-

```
Class Test {
P.S.V. main(String[] args){
S.O.pln(Thread.CurrentThread()
        .getThreadGroup()
        .getName());
        // main
SOPln(Thread.CurrentThread()
        .getThreadGroup()
        .getParent()
        .getName()); // System
```

main thread ← main Thread Group ← System Thread Group ← System

• ThreadGroup is a Java Class present in java.lang Package and it is the direct child class of Object.

Constructors:-

① ThreadGroup g = new ThreadGroup(String groupName);

• Creates a new thread Group with the specified Group name. the parrent of this new group is the Thread Group of currently Executing Thread.

Ex:- ThreadGroup g = new ThreadGroup("First Group");

② ThreadGroup g = new ThreadGroup(ThreadGroup pg, String gName);

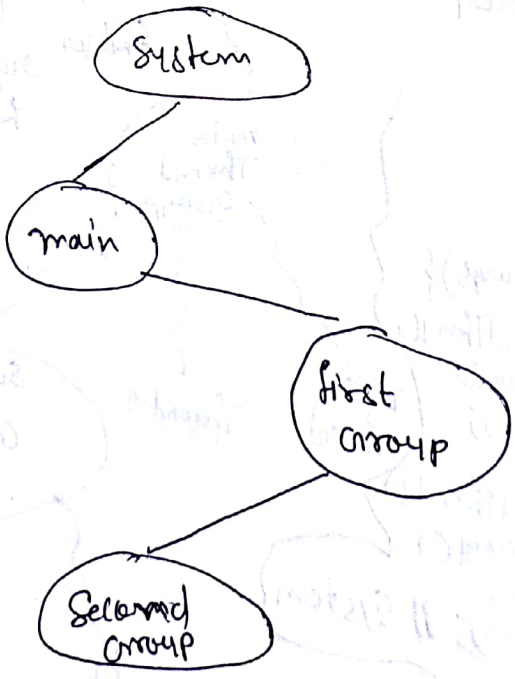
Creates a new thread Group with the specified Group name. the parrent of this new Group is specified parrent group.

Ex:-

ThreadGroup g1 = new ThreadGroup(g, "Second Group");

Example:-

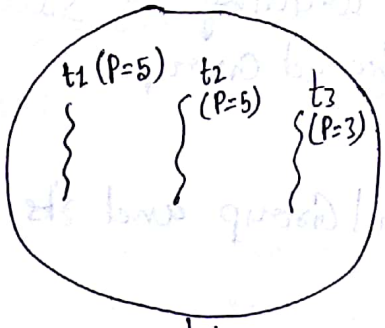
```
Class Test {
    Public static void main(String[] args) {
ThreadGroup g1 = new ThreadGroup("First Group");
System.out.println(g1.getParent().getName()); //main
ThreadGroup g2 = new ThreadGroup(g1, "Second Group");
System.out.println(g2.getParent().getName()); //First Group
    }
}
```



Importants of methods of ThreadGroup Class:-

1. String getName();
   - return name of the ThreadGroup.
2. int getMaxPriority();
   - return max priority of threadGroup.
3. void SetMaxPriority(int p);
   - To Set Maximum priority of threadGroup.
   - The Default max priority is 10.
   - Threads in the thread Group that already have higher priority don't be affected but for newly added this max priority is applicable.

Ex:-
```
Class ThreadGroupDemo2{
    Public Static Void main(String[] args){
        ThreadGroup g1 = new ThreadGroup("tg");
        Thread t1 = new Thread(g1, "Thread1");
        Thread t2 = new Thread(g1, "Thread2");
        g1.SetMaxPriority(3);
        Thread t3 = new Thread(g1, "Thread3");
        System.out.println(t1.getPriority()); // 5
        System.out.println(t2.getPriority()); // 5
        System.out.println(t3.getPriority()); // 3
```



max priority = ~~10~~ 3

4. ThreadGroup getParent();
   - returns Parent Group of Current Thread.
5. Void list();
   - It Prints information about thread Group to the Console.

6. Int activeCount();
- returns number of active threads present in the thread Group.

7. int activeGroupCount();
- it returns number of active groups present in the current thread Group.

8. int enumerate(Thread[] t);
- To copy all active thread of this thread group in to provided thread array.
- In this case sub thread group thread also will be Considered.

9. int enumerate(ThreadGroup[] g);
- To copy all active sub thread Groups in to Thread-Group array

10. boolean isDaemon();
- To check whether the threadGroup is Daemon or not.

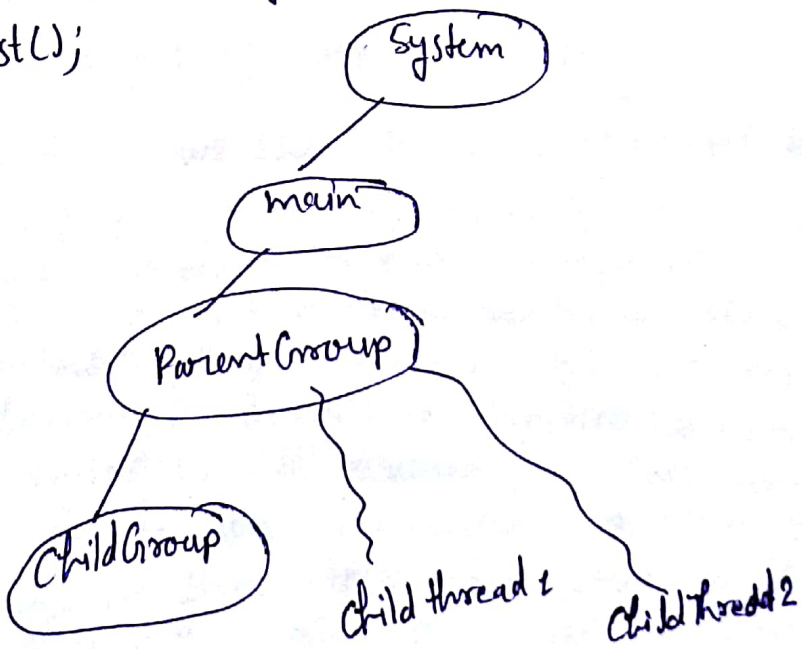
11. Void setDaemon(boolean b);
- To change the daemon instead of threadGroup.

12. Void interrupt();
- To interrupt all waiting or sleeping thread Present in the thread Group.

13. Void destroy();
- To destroy thread Group and its Sub thread Groups.

Example :-
```
Class MyThread extends Thread {
    MyThread(ThreadGroup g, String name){
        Super(g, name);
    }
    Public void run() {
        System.out.println("Child Thread");
```

```
try {
    Thread.sleep(5000);
  }
  catch (InterruptedException e) { }
}
}

class ThreadGroupDemo3 {

public static void main(String[] args) throws Exception
 {
   ThreadGroup pg = new ThreadGroup("ParrentGroup");
   ThreadGroup cg = new ThreadGroup(pg, "ChildGroup");
   MyThread t1 = new MyThread(pg, "ChildThread1");
   MyThred t2 = new MyThread(pg, "ChildThread2");
            t1.start();
            t2.start();
   System.out.println(pg.activeCount()); //2
   System.out.println(pg.activeGroupCount()); //1
   pg.list();
   Thread.sleep(10000);
   System.out.println(pg.activeCount()); //0
   System.out.println(pg.activeGroupCount); //1
     pg.list();
}
}
```



- Write the program to display all active thread names belong to SystemGroup and its child Groups.

```
Class ThreadGroupDemo4 {
  Public static void main(String [] args){
   ThreadGroup system =
    ~~ThreadGroup~~.Thread.currentThread().getThreadGroup()
                  .getParent();
   Thread [] t = new Thread[system.activeCount()];
    system.enumerate(t);
    for (Thread t1:t) {
      System.out.println(t1.getName()+"....."+
                         t1.isDaemon());
    }
  }
}
```

Output:- Reference Handler ..... true.
Finalizer ..... true
Signal Dispatcher ..... true
Attach Listener ..... true
main ..... false

The Problem with traditional synchronized Keyword :-

1. We are not having any flexibility to try for a lock without waiting.
2. There is no way to specify maximum waiting time for a thread to get lock. so that thread will wait untill getting the lock, which may create Performance Problem which may cause deadlock.
3. If a thread releases the lock then which waiting thread will get that lock we are not having any Controll on this
4. There is no API To List out all waiting thread for a lock.
5. The synchronized Keyword Compulsory we have to use either at method level or with in the method and it is not possible to use accross multiple methods.

→ To over come These Problem SUN Pepole introduce Java.Util.concurrent.Lock package in 1.5 version
→ It is also Provide several Enchancement to the Programmer to Provide more Controll on concurrency.

LOCK INTERFACE :- Lock object is similar to implicit Lock acquired by a thread to execute synchronized method or synchronized block.

- Lock implementation Provide more extensive operation then tradition implicit locks

Important methods of Lock interface :-

① void lock(); we can use this method to acquired a lock. If lock is already available the imediatly current thread will get the lock. If the lock is not already available then it will wait untill getting the lock. It is exactly same behaviour of traditional synchronized keyword.

② boolean tryLock(); To acquired the lock without waiting. If the lock is available then the Thread acquired that lock and returns true. If the lock is not available then this method returns false and can continue its execution without waiting

In this case thread never be entered in to waiting state.

```
if(l.tryLock())
{ performs safe operations
}else{
   Performs alternative operations
}
```

③ boolean tryLock(long time, TimeUnit unit);

If lock is available then the thread will get the lock and continue its execution.

If lock is not available then the Thread will wait until specify amount of time. Still if the lock is not available then thread can continue its execution.

TimeUnit → Nanoseconds, microseconds, milliseconds, Seconds, minutes, Hours, Days.

⇒
```
enum TimeUnit{
   NANOSECONDS,
   MICROSECONDS,
   MILLISECONDS,
   SECONDS,
   MINUTES,
   HOURS,
   DAYS.
}
```
} It is present in Java.util.Concurrent Package.

Example:-
```
if(l.tryLock(2000, TimeUnit.MILLISECONDS))
{
}
```

④ void lockInterruptibly();

acquired the lock if it is available and the returns immediately. If the lock is not available then it will wait. While waiting if the thread is interrupted then thread will not get the lock.

⑤ void unlock();

- To release the lock.
- To call this method compulsory current thread should be owner of the lock other wise we will get runtime Exception saying IllegalMonitorStateException.

ReentrantLock (C) :-

- It is the Implementation class of ReentrantLock Interface. and it is the direct child class of object.
- Reentrant means a thread can acquired same lock multiple times without any issue
- Internally Reentrant lock increments threads Personal counts whenever we call lock methods and decrement counts value whenever thread calls Unlock methods and lock will be released whenever counts reaches zero.

Constructor :-

① ReentrantLock l = new RenentrantLock();

- Create the instance of ReentrantLock.

② ReentrantLock l = new ReentrantLock(boolean fairness);

- Creates the reentrantlock with the given fairness Policy. If the fairness is true then longest waiting thread can get the lock if it is aviable. that is it follows FCFS (first come first serve) Policy.
- If fairness is false then which waiting thread will get the chance we can't expect.

Note:- the default value for fairness is false.

Which are declaration are equals.

ReentrantLock l = new ReentrantLock();
ReentrantLock l = new ReentrantLock(true);
ReentrantLock l = new ReentrantLock(false);
all the above

Importants methods of ReentrantLock :-

① void Lock();
② boolean tryLock();
③ boolean tryLock(long l, TimeUnit t);
④ void lockInterruptibly();
⑤ void unlock();
⑥ Int getHoldCount();
Returns no. of Hold on this lock by Current thread.

(7) boolean isHeldByCurrentThread()

Returns true if and only if lock is hold by current Thread.

(8) Int getQueueLength()

Returns numbers of the threads waiting for the lock.

(9) Collection getQueuedThreads()

It Returns a Collection of threads which are waiting to get the lock.

(10) boolean hasQueuedThreads()

Returns true if any thread waiting to get the lock

(11) boolean isLocked()

Returns true if the lock is acquired by some Thread.

(12) boolean isFair();

Returns true if the fairness policy is set with true value.

(13) Thread getOwner()

Returns Thread which acquired the lock.

Example;-

```
import java.util.concurrent.locks.*;
class ReentrantLock2{
  Public static void main(String[] args){
    ReentrantLock l = new ReentrantLock();
      l.lock();
      l.lock();
    System.out.println(l.isLocked()); // true
    System.out.println(l.isHeldByCurrentThread()); // true
    System.out.println(l.getQueueLength()); //0
    l.unLock();
    System.out.println(l.getHoldCount()); //1
    System.out.println(l.isLocked()); // true
    l.unlock();
    System.out.println(l.isLocked()); // false
    system.out.println(l.isFair()); // false
  }
}
```

example:-

```
import java.util.concurrent.locks.*;
Class Display{
ReentrantLock l = new ReentrantLock();
Public void wish(String name){
  l.lock();  ——————→ 1
 for(int i=0; i<10; i++){
  System.out.println("Good Morning:");
  try{
   Thread.sleep(2000);
  }
  catch (InterruptedException e){
  }
  System.out.println(name);
 }l.unlock(); ——————→ 2
}
}
```

⇒ Class MyThread extends Thread {

```
Display d;
String name;
MyThread(Display d, String name){
    this.d = d;
    this.name = name;
}
Public void run(){
  d.wish();
}
}
```

⇒ Class ReentrantLockDemo1 {

```
Public static void main(String[] args){
    Display d = new Display();
    MyThread t1 = new MyThread(d, "Dhoni");
    MyThread t2 = new MyThread(d, "yuvRaj");
    MyThread t3 = new MyThread(d, "Kohli");
      t1.start();
      t2.start();
  } t3.start();
}
```

- If we comment lines 1 and 2 then the thread will be executed simultaneously and we will get irregular output.
- if we are not commenting lines 1 and 2 then the thread will be executed one by one and we will get regular output.

Demo program for tryLock method

```
import java.util.concurrent.locks.*;
⇒ class MyThread extends Thread {
    static ReentrantLock l = new ReentrantLock();
    MyThread(String name){
      super(name);
    }
    public void run(){
      if(l.tryLock()){
        System.out.println(Thread.currentThread().getName()
            + "....get lock and performing safe
              operations");
        try{
          Thread.sleep(2000);
        }
        catch(InterruptedException e){}
        l.unlock();
      }
      else{
        System.out.println(Thread.currentThread().getName()
            +"....unable to get lock and hence
          Performing alternative operations");
      }
    }
⇒ class ReentrantLockDemo3 {
    public static void main(String[] args){
      MyThread t1 = new MyThread("first thread");
      MyThread t2 = new MyThread("second Thread");
      t1.start();
      t2.start();
    }
}
```

output:- First Thread:....got lock and performing safe operations
second Thread:....unable to get lock and hence performing alternate operations

Example:-

```
Import Java.util.Concurrent.lock.*;
Import Java.Util.Concurrent.*;
⇒ Class MyThread extends Thread {
   Static ReentrantLock l = new ReentrantLock();
   MyThread(String name){
      super(name);
   }
   Public Void run(){
      do{
         try {
            if(l.tryLock(5000,TimeUnit.MILLISECONDS)){
            System.out.println(Thread.CurrentThread().getName()
                              +".....got lock");

            Thread.sleep(30000);
            l.unlock();
         System.out.println(Thread.currentThread().getName()
                              +"---- releases the lock");

            break;
         }
         else{
            System.out.println(Thread.currentThread().getName()
                              +"... unable to get the lock and
                     +" will try again");
         }
      }
      Catch(Exception e){ }
   }while(true);
}}

⇒ Class ReentrantLockDemo4{
      Public static void main(String[] args){
         MyThread t1 = new MyThread("first Thread");
         MyThread t2 = new MyThread("second Thread");
            t1.start();
            t2.start();
   }}
```

· Output :

First Thread ---- got lock
Second Thread ---- Unable to get the lock and will try again
Second Thread ---- Unable to get the lock and will try again
" " " " " " " " " "
" " " " " " " " "
" " " " " " " " "

First Thread ..... Release lock
Second Thread .... get lock
Second Thread .... Release lock

Thread Pools {Executor framework}

- Creating a new thread for every job may creating Performance & memory Problem to overcome this we should go for thread pool.
- Thread pool is a pool of already created thread ready to do our job.
- Java 1.5 version introduces thread pool framework to implement Thread pool.
- Thread Pool framework also known as Executor framework.
- we can create Thread Pool as follows.

```
ExecutorService service = Executors.newFixedThreadPool(3);
```

- we can submit a Runnable Job by using submit method.

```
service.submit(job);
```

- we can shutdown executor service by using shutdown method.

```
service.shutdown();
```

Example

```
Import java.util.concurrent.*;
class PrintJob implements Runnable{
    String name
    PrintJob(String name){
      this.name=name;
    }
    Public void run(){
     System.out.println(name+"...Job Started by
       Thread:" + Thread.currentThread().getName());
    try{
      Thread.sleep(5000);
    }
    Catch(InterruptedException e){}
    System.out.println(name+"....Job Completed by
      Thread:"+Thread.currentThread().getName());
}}
```

```
Class ExecutorDemo {
  Public Static void main (String[] args){
    PrintJob[] Jobs = { new PrintJob("durga");
                        new PrintJob("Ravi");
                        new PrintJob("Shiva");
                        new PrintJob("Pawan");
                        new PrintJob("Suresh");
                        new PrintJob("Anil") };
    ExecutorService service = Executors.newFixedThreadPool(3);
      for (PrintJob Job : Jobs){
          service.submit(Job);
      }
      service.shutdown();
    }
}
```

- In the above example 3 threads are responsible to execute 6 Jobs so that a single thread can be reused of multiple Jobs.

Note
- While designing webserver and application server we can use thread pool concept.

Callable and Future:

- In the case of runnable Job thread won't return anything after completing the Job.
- If a thread is required to return after execution then we should go Callable.
- Callable interface contain only one method call. Public object call() throws Exception
- If we submit Callable object to executor then after completing the Job thread return object of the type "Future".
- That is Future object can be used to retrived the result from Callable Job.

Example:- Callable future

```
import java.util.concurrent.*;
class MyCallable implements Callable{
    int num;
    MyCallable(int num){
        this.num = num;
    }
    Public Object call() throws Exception{
    System.out.println(Thread.currentThread.getName() +"
        is... responsible to find sum of first " +num+"
        numbers");
    int sum=0;
    for(int i=1; i<=num; i++){
        sum = sum+i;
    }
    return sum;
    }
}
Class CallableFutureDemo{
Public Static void main(String[] args) throws Exception{
MyCallable[] Jobs = { new MyCallable(10),
                      new MyCallable(20),
                      new MyCallable(30),
                      new MyCallable(40),
                      new MyCallable(50),
                      new MyCallable(60)};

ExecutorService service = Executors.newFixedThreadPool(3);
    for(MyCallable Job: Jobs){
        Future f = service.submit(Job);
        System.out.println(f.get());
    }
    service.shutdown();
}
}
```

output:

```
55
210
465
820
1275
1830
```

Difference between Runnable and Callable

Runnable	Callable
• If a Thread is not required to return anything after completing the Job. then we should go for Runnable	• If a Thread is required to return something after completing the Job then we should go for Callable
• Runnable Interface Contains only one method "run()"	• Callable Interface Contains only one method "Call()".
• Runnable Job is not required to return anything and hence return type of run method is Void.	• Callable Job is required to return something and hence return type of Call method is Object.
• With in the run() method if there is any chance of raising Checked Exception Compulsory we should handle by using try/catch because we can't use throws keyword for run method	• Inside Call method if there is any chance of raising checked Exception we are not required to handle by using try/catch because Call method already throws Exception
• Runnable Interface present in "Java.lang" Package.	• Callable Interface Present in "Java.util.Concurrent" Package.
• Introduce in 1.0 version	• Introduce in 1.5 version

Available Thread-Pools :-

Many type of thread-pools available out-of-the-box

- Fixed Thread pool
- Cached Thread pool
- Single Thread Executer (technically not a 'pool')
- Scheduled Thread pool
- Single Thread Scheduled Executor (technically not a 'pool')

Fixed Thread pool :-

ExecutorService

The `java.util.concurrent.ExecutorService` interface represents an asynchronous execution mechanism which is capable of executing tasks in the background. An `ExecutorService` is thus very similar to a thread pool. In fact, the implementation of `ExecutorService` present in the `java.util.concurrent` package is a thread pool implementation

Here is a simple Java `ExectorService` example:

```
ExecutorService executorService = Executors.newFixedThreadPool(10);

executorService.execute(new Runnable() {
    public void run() {
        System.out.println("Asynchronous task");
    }
});
executorService.shutdown();
```

First an `ExecutorService` is created using the `newFixedThreadPool()` factory method. This creates a thread pool with 10 threads executing tasks.

Second, an anonymous implementation of the `Runnable` interface is passed to the `execute()` method. This causes the `Runnable` to be executed by one of the threads in the `ExecutorService`.

ExecutorService Implementations

Since `ExecutorService` is an interface, you need to its implementations in order to make any use of it. The `ExecutorService` has the following implementation in the `java.util.concurrent` package:

- ThreadPoolExecutor
- ScheduledThreadPoolExecutor

Creating an ExecutorService

How you create an `ExecutorService` depends on the implementation you use. However, you can use the `Executors` factory class to create `ExecutorService` instances too. Here are a few examples of creating an `ExecutorService`:

```
ExecutorService executorService1 = Executors.newSingleThreadExecutor();
ExecutorService executorService2 = Executors.newFixedThreadPool(10);
ExecutorService executorService3 = Executors.newScheduledThreadPool(10);
```

ExecutorService Usage

There are a few different ways to delegate tasks for execution to an `ExecutorService`:

- **execute(Runnable)**
- **submit(Runnable)**
- **submit(Callable)**
- **invokeAny(...)**
- **invokeAll(...)**

I will take a look at each of these methods in the following sections.

➢ execute(Runnable)

The `execute(Runnable)` method takes a `java.lang.Runnable` object, and executes it asynchronously. Here is an example of executing a `Runnable` with an `ExecutorService`:

```
ExecutorService executorService = Executors.newSingleThreadExecutor();

executorService.execute(new Runnable() {
    public void run() {
        System.out.println("Asynchronous task");
    }
});

executorService.shutdown();
```

There is no way of obtaining the result of the executed `Runnable`, if necessary. You will have to use a `Callable` for that (explained in the following sections).

➢ submit(Runnable)

The `submit(Runnable)` method also takes a `Runnable` implementation, but returns a `Future` object. This `Future` object can be used to check if the `Runnable` as finished executing.
Here is a `ExecutorService submit()` example:

```
Future future = executorService.submit(new Runnable() {
    public void run() {
        System.out.println("Asynchronous task");
    }
});

future.get();  //returns null if the task has finished correctly.
```

➢ submit(Callable)

The `submit(Callable)` method is similar to the `submit(Runnable)` method except for the type of parameter it takes. The `Callable` instance is very similar to a `Runnable` except that its `call()` method can return a result.
The `Runnable.run()` method cannot return a result.
The `Callable`'s result can be obtained via the `Future` object returned by the `submit(Callable)` method. Here is an `ExecutorService Callable` example:

```
Future future = executorService.submit(new Callable(){
    public Object call() throws Exception {
        System.out.println("Asynchronous Callable");
        return "Callable Result";
    }
});

System.out.println("future.get() = " + future.get());
```

The above code example will output this:

```
Asynchronous Callable
future.get() = Callable Result
```

- **invokeAny()**

The `invokeAny()` method takes a collection of `Callable` objects, or subinterfaces of `Callable`. Invoking this method does not return a `Future`, but returns the result of one of the `Callable` objects. You have no guarantee about which of the `Callable`'s results you get. Just one of the ones that finish.

If one of the tasks complete (or throws an exception), the rest of the `Callable`'s are cancelled.

Here is a code example:

```
ExecutorService executorService = Executors.newSingleThreadExecutor();

Set<Callable<String>> callables = new HashSet<Callable<String>>();

callables.add(new Callable<String>() {
    public String call() throws Exception {
        return "Task 1";
    }
});
callables.add(new Callable<String>() {
    public String call() throws Exception {
        return "Task 2";
    }
});
callables.add(new Callable<String>() {
    public String call() throws Exception {
        return "Task 3";
    }
});

String result = executorService.invokeAny(callables);
System.out.println("result = " + result);
executorService.shutdown();
```

This code example will print out the object returned by one of the `Callable`'s in the given collection. I have tried running it a few times, and the result changes. Sometimes it is "Task 1", sometimes "Task 2" etc.

- **invokeAll()**

The `invokeAll()` method invokes all of the `Callable` objects you pass to it in the collection passed as parameter. The `invokeAll()` returns a list of `Future` objects via which you can obtain the results of the executions of each `Callable`.

Keep in mind that a task might finish due to an exception, so it may not have "succeeded". There is no way on a `Future` to tell the difference.
Here is a code example:

```
ExecutorService executorService = Executors.newSingleThreadExecutor();
Set<Callable<String>> callables = new HashSet<Callable<String>>();

callables.add(new Callable<String>() {
    public String call() throws Exception {
        return "Task 1";
    }
});
callables.add(new Callable<String>() {
    public String call() throws Exception {
        return "Task 2";
    }
});
callables.add(new Callable<String>() {
    public String call() throws Exception {
        return "Task 3";
    }
});

List<Future<String>> futures = executorService.invokeAll(callables);
for(Future<String> future : futures){
    System.out.println("future.get = " + future.get());
}
executorService.shutdown();
```

ExecutorService Shutdown

When you are done using the `ExecutorService` you should shut it down, so the threads do not keep running.

For instance, if your application is started via a `main()` method and your main thread exits your application, the application will keep running if you have an active `ExexutorService` in your application. The active threads inside this `ExecutorService` prevents the JVM from shutting down.

To terminate the threads inside the `ExecutorService` you call its `shutdown()` method.
The `ExecutorService` will not shut down immediately, but it will no longer accept new tasks, and once all threads have finished current tasks, the `ExecutorService` shuts down. All tasks submitted to the `ExecutorService` before `shutdown()` is called, are executed.

If you want to shut down the `ExecutorService` immediately, you can call the `shutdownNow()` method. This will attempt to stop all executing tasks right away, and skips all submitted but non-processed tasks. There are no guarantees given about the executing tasks. Perhaps they stop, perhaps the execute until the end. It is a best effort attempt.

ThreadPoolExecutor

The `java.util.concurrent.ThreadPoolExecutor` is an implementation of the `ExecutorService` interface. The `ThreadPoolExecutor` executes the given task (`Callable` or `Runnable`) using one of its internally pooled threads.
The thread pool contained inside the `ThreadPoolExecutor` can contain a varying amount of threads. The number of threads in the pool is determined by these variables:

- `corePoolSize`
- `maximumPoolSize`

If less than `corePoolSize` threads are created in the the thread pool when a task is delegated to the thread pool, then a new thread is created, even if idle threads exist in the pool.

If the internal queue of tasks is full, and `corePoolSize` threads or more are running, but less than `maximumPoolSize` threads are running, then a new thread is created to execute the task.

Creating a ThreadPoolExecutor

The `ThreadPoolExecutor` has several constructors available. For instance:

```
int  corePoolSize  =    5;
int  maxPoolSize   =   10;
long keepAliveTime = 5000;

ExecutorService threadPoolExecutor =
        new ThreadPoolExecutor(
                corePoolSize,
                maxPoolSize,
                keepAliveTime,
                TimeUnit.MILLISECONDS,
                new LinkedBlockingQueue<Runnable>()
                );
```

However, unless you need to specify all these parameters explicitly for your `ThreadPoolExecutor`, it is often easier to use one of the factory methods in the `java.util.concurrent.Executors` class, as shown in the **ExecutorService** text.

ScheduledExecutorService

The `java.util.concurrent.ScheduledExecutorService` is an `ExecutorService` which can schedule tasks to run after a delay, or to execute repeatedly with a fixed interval of time in between each execution.
Tasks are executed asynchronously by a worker thread, and not by the thread handing the task to the `ScheduledExecutorService`.

ScheduledExecutorService Example

Here is a simple `ScheduledExecutorService` example:

```
ScheduledExecutorService scheduledExecutorService =
        Executors.newScheduledThreadPool(5);

ScheduledFuture scheduledFuture =
    scheduledExecutorService.schedule(new Callable() {
        public Object call() throws Exception {
            System.out.println("Executed!");
            return "Called!";
        }
    },
    5,
    TimeUnit.SECONDS);
```

First a `ScheduledExecutorService` is created with 5 threads in. Then an anonymous implementation of the `Callable` interface is created and passed to the `schedule()` method. The two last parameters specify that the `Callable` should be executed after 5 seconds.

ScheduledExecutorService Implementations

Since `ScheduledExecutorService` is an interface, you will have to use its implementation in the `java.util.concurrent` package, in order to use it. `ScheduledExecutorService` as the following implementation:

- **ScheduledThreadPoolExecutor**

Creating a ScheduledExecutorService

How you create an `ScheduledExecutorService` depends on the implementation you use. However, you can use the `Executors` factory class to create `ScheduledExecutorService` instances too. Here is an example:

```
ScheduledExecutorService scheduledExecutorService =
        Executors.newScheduledThreadPool(5);
```

ScheduledExecutorService Usage

Once you have created a `ScheduledExecutorService` you use it by calling one of its methods:

- schedule (Callable task, long delay, TimeUnit timeunit)
- schedule (Runnable task, long delay, TimeUnit timeunit)
- scheduleAtFixedRate (Runnable, long initialDelay, long period, TimeUnit timeunit)
- scheduleWithFixedDelay (Runnable, long initialDelay, long period, TimeUnit timeunit)

I will briefly cover each of these methods below.

- **schedule (Callable task, long delay, TimeUnit timeunit)**

This method schedules the given `Callable` for execution after the given delay. The method returns a `ScheduledFuture` which you can use to either cancel the task before it has started executing, or obtain the result once it is executed.

Here is an example:

```
ScheduledExecutorService scheduledExecutorService =
        Executors.newScheduledThreadPool(5);

ScheduledFuture scheduledFuture =
    scheduledExecutorService.schedule(new Callable() {
        public Object call() throws Exception {
            System.out.println("Executed!");
            return "Called!";
        }
    },
    5,
    TimeUnit.SECONDS);

System.out.println("result = " + scheduledFuture.get());
scheduledExecutorService.shutdown();
```

This example outputs:

```
Executed!
result = Called!
```

- **schedule (Runnable task, long delay, TimeUnit timeunit)**

This method works like the method version taking a `Callable` as parameter, except a `Runnable` cannot return a value, so the `ScheduledFuture.get()` method returns null when the task is finished.

- **scheduleAtFixedRate (Runnable, long initialDelay, long period, TimeUnit timeunit)**

This method schedules a task to be executed periodically. The task is executed the first time after the `initialDelay`, and then recurringly every time the `period` expires.

If any execution of the given task throws an exception, the task is no longer executed. If no exceptions are thrown, the task will continue to be executed until the `ScheduledExecutorService` is shut down.

If a task takes longer to execute than the period between its scheduled executions, the next execution will start after the current execution finishes. The scheduled task will not be executed by more than one thread at a time.

- **scheduleWithFixedDelay (Runnable, long initialDelay, long period, TimeUnit timeunit)**

This method works very much like `scheduleAtFixedRate()` except that the `period` is interpreted differently.

In the `scheduleAtFixedRate()` method the `period` is interpreted as a delay between the start of the previous execution, until the start of the next execution.

In this method, however, the `period` is interpreted as the delay between the **end** of the previous execution, until the start of the next. The delay is thus between finished executions, not between the beginning of executions.

ScheduledExecutorService Shutdown

Just like an `ExecutorService`, the `ScheduledExecutorService` needs to be shut down when you are finished using it. If not, it will keep the JVM running, even when all other threads have been shut down.

You shut down a `ScheduledExecutorService` using the `shutdown()` or `shutdownNow()` methods which are inherited from the `ExecutorService` interface.

There are many static methods available with the *Executors* class (Note that *Executor* is an interface and *Executors* is a class. Both included in the package *java.util.concurrent*). A few of the commonly used are as follows:

- **newCachedThreadPool():** It creates new thread as needed, but is quick to reuse any available thread previously constructed. The thread pool can shrink and expand, depending upon the workload.
- **newFixedThreadPool(int nThreads):** The thread pool created by this method has a fixed size set by the parameter passed. At any given time, there can be maximum of nThread (number of threads) in the pool.
- **newSingleThreadExecutor():** This method ensures that there will be only one thread to execute all the tasks. If this thread dies unexpectedly, a new one is created. But, there is a guarantee that there will be a single thread at any given time.

All of these methods return an *ExecutorService* object.

* Thread Local *

* Thread Local Class provides ThreadLocal variable
* ThreadLocal class maintains values Per Thread basis.
* Each ThreadLocal Object maintains a seprate value like UserID Transaction Id Extra for Each thread that Access that Object
* Thread Can access its Local value, Can manipulate its value and even Can remove its value. In Every Part Of the Code which is Exicuted by the thread we can access its Local variable

Example:- Consider A servlets which Invokes some business methods we have a requirements to generate a unique Transaction-Id for Each an Every request. and we have to pass this Transaction Id to the Buisness methods for this requirement We can use thread Local to maintain a seprate transaction Id for Every request that is for every thread.

Note*→ ① Thread Local Class Introduced in 1.2 Version and Enhanced in 1.5 version.
② Thread Local can be associated with Thread Scope.
③ Totale code which is Executed by the thread has access to corresponding thread Local variable.
④ A Thread Can Access its own Local Variable and can't Access othrer threads Local variables.
⑤ once Thread entered in to Dead State all its local variables are by Default available for garbage collection.

Constructor for ThreadLocal

```
ThreadLocal tl = new ThreadLocal();
```

* Creates a Thread Local variables.

Methods:-

① Object get();
returns the value of ThreadLocal Variable associated with Corrent thread.

② Object initialValue();
returns initial value of ThreadLocal variable associated with Corrent thread. The default implementaion of this method returns null To customize our own initial value we have to override this method.

③ Void Set(Object newValue);
To set a new value

④ Void remove();
To remove the value of ThreadLocal Variable associated with (Corrent Thread)

* It is newly added method in 1.5 version.
* After removing if we are trying to access it will reinitialize once again by invoking its initialValue method.

Example:-

```
Class ThreadLocalDemo1
{
 Public static void main(String[] args)
 {
  ThreadLocal tl = new ThreadLocal();
 System.out.println(tl.get()); // null
 tl.set("durga");
 System.out.println(tl.get()); // durga
 tl.remove();
 System.out.println(tl.get()); // null;
 }
}
```

Example 2:- overriding of Initial value method.

```
Class ThreadLocalDemo1A
{
 Public static void main(String[] args)
  {
   ThreadLocal tl = new ThreadLocal()
     {
       Public Object initialValue()
       {
         return "abc";
       }
     };
  System.out.println(tl.get()); // abc
  tl.set("durga");
  System.out.println(tl.get()); // durga;
  tl.remove();
  System.out.println(tl.get()); // abc;
 }
}
```

Example:-

```
Class CustomerThread extends Thread
{
  Static Integer custId = 0;
Private Static ThreadLocal td = new ThreadLocal();
{
  Protected Integer initialValue(){
     return ++custId;
  }
};
CustomerThread(String name)
{
   super(name);
}
Public Void run()
{
 System.out.println(Thread.currentThread.getName()+"
   executing with Customer Id :"+ td.get());
}
}
Class ThreadLocalDemo2
{
  Public Static Void main(String[] args)
  {
   CustomerThread C1 = new CustomerThread("
                       CustomerThread-1");
   CustomerThread C2 = new CustomerThread("
                       CustomerThread-2");
   CustomerThread C3 = new CustomerThread("
                       CustomerThread-3");
   CustomerThread C4 = new CustomerThread("
                       CustomerThread-4");
   C1.start();
   C2.start();
   C3.start();
   C4.start();
  }
}
```

Output:- order of thread can't be gaurantee.

```
Customer thread-1 execution with Customer id :1
"   Thread-3   "   "   "   :4
"   Thread-4   "   "   "   :3
"   Thread-2   "   "   "   :2
```

The above Program the Every Customer thread a seperate Customer Id will be maintained by ThreadLocal object.

ThreadLocal with respect to Inheritence :-

```
Class ParrentThread extends Thread
{
   static ThreadLocal tl = new ThreadLocal();

      Public Stoid run()
      {
          tl.set("pp");
          System.out.println("Parent value:" + tl.get()); // PP
          ChildThread ct = new ChildThread();
            ct.start();
      }
}

Class ChildThread extends Thread
{
    Public void run()
     {
      System.out.println("Child value:" + ParrentThread.tl.get());
                                                    // null
    }
}

Class ThreadLocalDemo3
{
    Public static void main(String[] args)
    {
      ParrentThread Pt = new ParentThread();
        Pt.start();
    }
}
```

Thread local vs Inheritence

- Parrent Threads threadLocal variable by default not available To the child thread. If we want to make Parrent Threads Thread local variable value available to the child Thread. Then we should go for "InheritableThreadLocal" class.
- By Default child Thread value is exactly same as Parrent threads value But we can provide customized value for child Thread by overriding childValue() method.
- Constructor :-

```
InheritableThreadLocal tl = new InheritableThreadLocal();
```

Methods :-

InheritableThreadLocal is the child class of ThreadLocal and hence all methods present in thread Local By Default available to InheritableThreadLocal. In addition to these methods It Contains only one method.

```
Public Object childValue(Object ParrentValue);
```

Example :-

```
class ParentThread Extends Thread
{
Public Static InheritableThreadLocal tl = new
  InheritableThreadLocal(){
    return "cc";
  }
};

Public void run(){
   tl.set("pp");
  System.out.println("Parrent Thread Value - "+tl.get());
   ChildThread ct = new ChildThread();
   ct.start();
  }
}

class ChildThread extends Thread {
    Public void run() {
        System.out.println("Child Thread value..."+
```

```
        ParrentThread.td.get());
    }
}
Class ThreadLocalDemo3
  {
    Public Static void main(String[] args)
    {
      ParentThread pt = new ParrentThread();
        Pt.start();
    }
}
```

Output :- ParentThread value :--- pp
Child Thread value: -- cc

* In the above program if we replace InheritableThreadLocal with the ThreadLocal and if we are not overriding childValue method then the output is

Parent Thread value -- pp
Child Thread value -- null

* In the above program if we are maintaining InheritableThread Local and if we are not overriding childValue Method Then output is:

Parent Thread value -- pp
Child Thread value -- pp

Q- Print Even and Odd using two Threads in Java ;-

```
Public class PrintEvenOddTester {
  Public static void main (String .... args) {
    Printer print = new printer();
  Thread t1 = new Thread (new TaskEvenOdd (print, 10, false));
  Thread t2 = new Thread (new TaskEvenOdd (print, 10, true));
        t1.start();
        t2.start();
  }
}

Class TaskEvenOdd implements Runnable {
  Private int max;
  Private Printer print;
  Private boolean isEvenNumber;
   TaskEvenOdd (Printer print, int max, boolean isEvenNumber) {
              this.print = print;
              this.max = max;
   }          this.isEvenNumber = isEvenNumber;

  Public void run () {
    int number = isEvenNumber == true ? 2 : 1;
    While (number <= max) {
       if (isEvenNumber) {
         Print.PrintEven (number);
       } else {
       } Print.PrintOdd (number);
       number += 2;
  }
}
}
```

```
class Printer {
   boolean isOdd = false;
   Synchronized void PrintEven (int number) {
       While (isOdd == false) {
         try {
            wait();
         } catch (InterruptedException e) {
            e.printStackTrace();
       }
   }
  System.out.println ("Even:" + number);
      isOdd = false;
    notifyAll();
}

   Synchronized void printOdd (int number) {
       while ( isOdd == true) {
         try {
            wait();
         } catch ( InterruptedException e) {
            e.printStackTrace();
         }
       }
      System.out.println ( "odd:" + number);
        isOdd = true;
      notifyAll();
   }
}
```

output :-

```
odd : 1
Even : 2
odd : 3
Even : 4
Odd : 5
Even : 6
Odd : 7
Even : 8
odd : 9
Even : 10
```

Q:- Print the natural sequence of the integers starting from 1 from a set of 3 (or more) threads that take turns in a round robin fashion; such that, thread 1 prints 1, thread 2 prints 2, thread 3 prints 3 and then thread 1 prints 4 and so on.

```
Class Worker implements Runnable {
  BlockingQueue<Integer> q = new LinkedBlockingQueue<Integer>();
   Worker next = null; // next worker in the chain
  Public void setNext(Worker t) {
        this.next = t;
  }

  Public void accept(int i) {
        q.add(i);
   }

  @Override
  Public void run() {
     while (true) {
        try {
          int i = q.take(); // blocks till it receives a number
          System.out.println(Thread.currentThread().getName()+i);
          Thread.sleep(1000); // delay to slow the printing
          if (next != null) {
           next.accept(i+1); // pass the next number to the
                                                next worker
          }
        } catch (InterruptedException e) {
          System.err.println(Thread.currentThread().getName()
                                + "interrupted.");
        }
     }
  }
}
```

```
Public class SequencePrinter {
  Pubic static void main(String[] args) {
     // Create the workers
    Worker w1 = new Worker();
    Worker w2 = new Worker();
    Worker w3 = new Worker();
    // chain them in a round robin fashion
      w1.setNext(w2);
      w2.setNext(w3);
      w3.setNext(w1);
    // Create named threads for the workers
     Thread t1 = new Thread(w1, "Thread-1-");
     Thread t2 = new Thread(w2, "Thread-2-");
     Thread t3 = new Thread(w3, "Thread-3-");
     // Start the threads
        t1.start();
        t2.start();
        t3.start();

      // Seed the first worker
        w1.accept(1);
  }
```

Output:-

```
Thread-1 - 1
Thread-2 - 2
Thread-3 - 3
Thread-1 - 4
Thread-2 - 5
Thread-3 - 6
Thread-1 - 7
Thread-2 - 8
Thread-3 - 9
Thread-1 - 10
Thread-2 - 11
Thread-3 - 12
.....
```

Java Thread – Mutex and Semaphore example

Java multi threads example to show you how to use Semaphore and Mutex to limit the number of threads to access resources.

1. **Semaphores** – Restrict the number of threads that can access a resource. Example, limit max 10 connections to access a file simultaneously.
2. **Mutex** – Only one thread to access a resource at once. Example, when a client is accessing a file, no one else should have access the same file at the same time.

1. Semaphore

Consider an ATM cubicle with 4 ATMs, Semaphore can make sure only 4 people can access simultaneously.

SemaphoreTest.java

```
package com.mkyong;
import java.util.concurrent.Semaphore;
public class SemaphoreTest {
	// max 4 people
	static Semaphore semaphore = new Semaphore(4);

	static class MyATMThread extends Thread {
		String name = "";
		MyATMThread(String name) {
			this.name = name;
		}

	public void run() {
			try {
				System.out.println(name + " : acquiring lock...");
				System.out.println(name + " : available Semaphore permits now: "
						+ semaphore.availablePermits());

				semaphore.acquire();
				System.out.println(name + " : got the permit!");

				try {
					for (int i = 1; i <= 5; i++) {
						System.out.println(name + " : is performing operation "+i
							+ ", available Semaphore permits : "
							+ semaphore.availablePermits());
						// sleep 1 second
						Thread.sleep(1000);
					}

				} finally {
					// calling release() after a successful acquire()
					System.out.println(name + " : releasing lock...");
					semaphore.release();
					System.out.println(name + " : available Semaphore permits now: "
						+ semaphore.availablePermits());
				}
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
		}
	}

	public static void main(String[] args) {
		System.out.println("Total available Semaphore permits : "
				+ semaphore.availablePermits());
```

```
            MyATMThread t1 = new MyATMThread("A");
            t1.start();

            MyATMThread t2 = new MyATMThread("B");
            t2.start();

            MyATMThread t3 = new MyATMThread("C");
            t3.start();

            MyATMThread t4 = new MyATMThread("D");
            t4.start();

            MyATMThread t5 = new MyATMThread("E");
            t5.start();

            MyATMThread t6 = new MyATMThread("F");
            t6.start();

        }
}
```

Output may vary, but the flow of locking and releasing should be more or less same.

```
Total available Semaphore permits : 4
A : acquiring lock...
D : acquiring lock...
C : acquiring lock...
B : acquiring lock...
B : available Semaphore permits now: 4
C : available Semaphore permits now: 4
E : acquiring lock...
F : acquiring lock...
F : available Semaphore permits now: 2
F : got the permit!
F : is performing operation 1, available Semaphore permits : 1
D : available Semaphore permits now: 4
A : available Semaphore permits now: 4
D : got the permit!
D : is performing operation 1, available Semaphore permits : 0
E : available Semaphore permits now: 2
C : got the permit!
B : got the permit!
C : is performing operation 1, available Semaphore permits : 0
B : is performing operation 1, available Semaphore permits : 0
F : is performing operation 2, available Semaphore permits : 0
D : is performing operation 2, available Semaphore permits : 0
C : is performing operation 2, available Semaphore permits : 0
B : is performing operation 2, available Semaphore permits : 0
F : is performing operation 3, available Semaphore permits : 0
D : is performing operation 3, available Semaphore permits : 0
C : is performing operation 3, available Semaphore permits : 0
B : is performing operation 3, available Semaphore permits : 0
F : is performing operation 4, available Semaphore permits : 0
D : is performing operation 4, available Semaphore permits : 0
C : is performing operation 4, available Semaphore permits : 0
B : is performing operation 4, available Semaphore permits : 0
D : is performing operation 5, available Semaphore permits : 0
F : is performing operation 5, available Semaphore permits : 0
B : is performing operation 5, available Semaphore permits : 0
C : is performing operation 5, available Semaphore permits : 0
```

```
D : releasing lock...
F : releasing lock...
D : available Semaphore permits now: 1
A : got the permit!
A : is performing operation 1, available Semaphore permits : 0
F : available Semaphore permits now: 1
E : got the permit!


E : is performing operation 1, available Semaphore permits : 0
B : releasing lock...
B : available Semaphore permits now: 1
C : releasing lock...
C : available Semaphore permits now: 2
A : is performing operation 2, available Semaphore permits : 2
E : is performing operation 2, available Semaphore permits : 2
A : is performing operation 3, available Semaphore permits : 2
E : is performing operation 3, available Semaphore permits : 2
A : is performing operation 4, available Semaphore permits : 2
E : is performing operation 4, available Semaphore permits : 2
A : is performing operation 5, available Semaphore permits : 2
E : is performing operation 5, available Semaphore permits : 2
A : releasing lock...
A : available Semaphore permits now: 3
E : releasing lock...
E : available Semaphore permits now: 4
```

Observe the above output carefully, you will see that there are maximum 4 people (C, B, F, D) to perform an operation at a time, the people A and E are waiting. As soon as one of them release the lock (D and F), A and E will acquire it and resumes immediately.

2. Mutex

Mutex is the Semaphore with an access count of 1. Consider a situation of using lockers in the bank. Usually the rule is that only one person is allowed to enter the locker room.

MutexTest.java

```
package com.mkyong;
import java.util.concurrent.Semaphore;

public class SemaphoreTest {
	// max 1 people
	static Semaphore semaphore = new Semaphore(1);
	static class MyLockerThread extends Thread {
		String name = "";
		MyLockerThread(String name) {
			this.name = name;
		}

		public void run() {
			try {
				System.out.println(name + " : acquiring lock...");
				System.out.println(name + " : available Semaphore permits now: "
					+ semaphore.availablePermits());

				semaphore.acquire();
```

```
                        System.out.println(name + " : got the permit!");
                        try {
                                for (int i = 1; i <= 5; i++) {
                                        System.out.println(name + " : is performing operation
i
                                                + ", available Semaphore permits : "
                                                + semaphore.availablePermits());
                                        // sleep 1 second
                                        Thread.sleep(1000);
                                }
                        } finally {
                                // calling release() after a successful acquire()
                                System.out.println(name + " : releasing lock...");
                                semaphore.release();
                                System.out.println(name + " : available Semaphore permits now
                                                + semaphore.availablePermits());
                        }

                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
        }
    }

    public static void main(String[] args) {
            System.out.println("Total available Semaphore permits : "
                            + semaphore.availablePermits());
            MyLockerThread t1 = new MyLockerThread("A");
            t1.start();
            MyLockerThread t2 = new MyLockerThread("B");
            t2.start();
            MyLockerThread t3 = new MyLockerThread("C");
            t3.start();
            MyLockerThread t4 = new MyLockerThread("D");
            t4.start();
            MyLockerThread t5 = new MyLockerThread("E");
            t5.start();
            MyLockerThread t6 = new MyLockerThread("F");
            t6.start();
    }
}
```

Output may vary, but the flow of locking and releasing should be same.

```
Total available Semaphore permits : 1
A : acquiring lock...
B : acquiring lock...
A : available Semaphore permits now: 1
C : acquiring lock...
B : available Semaphore permits now: 1
C : available Semaphore permits now: 0
A : got the permit!
D : acquiring lock...
E : acquiring lock...
A : is performing operation 1, available Semaphore permits : 0
E : available Semaphore permits now: 0
D : available Semaphore permits now: 0
F : acquiring lock...
F : available Semaphore permits now: 0
A : is performing operation 2, available Semaphore permits : 0
A : is performing operation 3, available Semaphore permits : 0
A : is performing operation 4, available Semaphore permits : 0
A : is performing operation 5, available Semaphore permits : 0
```

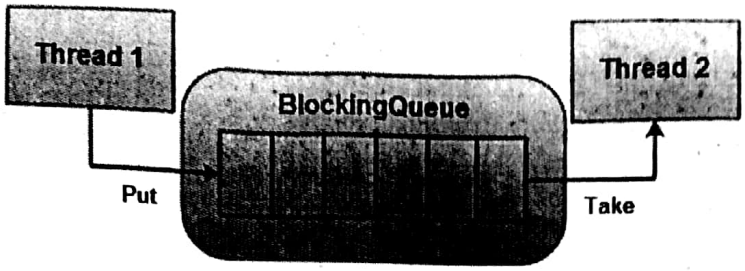
```
A : releasing lock...
A : available Semaphore permits now: 1
B : got the permit!
B : is performing operation 1, available Semaphore permits : 0
B : is performing operation 2, available Semaphore permits : 0
B : is performing operation 3, available Semaphore permits : 0
B : is performing operation 4, available Semaphore permits : 0
B : is performing operation 5, available Semaphore permits : 0
B : releasing lock...
B : available Semaphore permits now: 1
C : got the permit!
C : is performing operation 1, available Semaphore permits : 0
C : is performing operation 2, available Semaphore permits : 0
C : is performing operation 3, available Semaphore permits : 0
C : is performing operation 4, available Semaphore permits : 0
C : is performing operation 5, available Semaphore permits : 0
C : releasing lock...
C : available Semaphore permits now: 1
E : got the permit!
E : is performing operation 1, available Semaphore permits : 0
E : is performing operation 2, available Semaphore permits : 0
E : is performing operation 3, available Semaphore permits : 0
E : is performing operation 4, available Semaphore permits : 0
E : is performing operation 5, available Semaphore permits : 0
E : releasing lock...
E : available Semaphore permits now: 1
D : got the permit!
D : is performing operation 1, available Semaphore permits : 0
D : is performing operation 2, available Semaphore permits : 0
D : is performing operation 3, available Semaphore permits : 0
D : is performing operation 4, available Semaphore permits : 0
D : is performing operation 5, available Semaphore permits : 0
D : releasing lock...
D : available Semaphore permits now: 1
F : got the permit!
F : is performing operation 1, available Semaphore permits : 0
F : is performing operation 2, available Semaphore permits : 0
F : is performing operation 3, available Semaphore permits : 0
F : is performing operation 4, available Semaphore permits : 0
F : is performing operation 5, available Semaphore permits : 0
F : releasing lock...
F : available Semaphore permits now: 1
```

As it can be seen, only one thread executes at a time here. This is the role of a Mutex.

Blocking Queues

A blocking queue is a queue that blocks when you try to dequeue from it and the queue is empty, or if you try to enqueue items to it and the queue is already full. A thread trying to dequeue from an empty queue is blocked until some other thread inserts an item into the queue. A thread trying to enqueue an item in a full queue is blocked until some other thread makes space in the queue, either by dequeuing one or more items or clearing the queue completely.

Here is a diagram showing two threads cooperating via a blocking queue:



A BlockingQueue with one thread putting into it, and another thread taking from it.

The producing thread will keep producing new objects and insert them into the queue, un the queue reaches some upper bound on what it can contain. It's limit, in other words. If the blocking queue reaches its upper limit, the producing thread is blocked while trying to insert the new object. It remains blocked until a consuming thread takes an object out of the queue.

The consuming thread keeps taking objects out of the blocking queue, and processes them If the consuming thread tries to take an object out of an empty queue, the consuming thread is blocked until a producing thread puts an object into the queue.

Java 5 comes with blocking queue implementations in the `java.util.concurrent` package.

Blocking Queue Implementation

The implementation of a blocking queue looks similar to a **Bounded Semaphore**. Here is a simple implementation of a blocking queue:

```
public class BlockingQueue {

  private List queue = new LinkedList();
  private int  limit = 10;

  public BlockingQueue(int limit){
    this.limit = limit;
  }


  public synchronized void enqueue(Object item)
  throws InterruptedException  {
    while(this.queue.size() == this.limit) {
      wait();
    }
    if(this.queue.size() == 0) {
      notifyAll();
    }
    this.queue.add(item);
  }


  public synchronized Object dequeue()
  throws InterruptedException{
    while(this.queue.size() == 0){
```

```
        wait();
      }
      if(this.queue.size() == this.limit){
        notifyAll();
      }

      return this.queue.remove(0);
    }

}
```

Notice how notifyAll() is only called from enqueue() and dequeue() if the queue size is equal to the size bounds (0 or limit). If the queue size is not equal to either bound when enqueue() or dequeue() is called, there can be no threads waiting to either enqueue or dequeue items.

BlockingQueue Methods

A BlockingQueue has 4 different sets of methods for inserting, removing and examining the elements in the queue. Each set of methods behaves differently in case the requested operation cannot be carried out immediately. Here is a table of the methods:

	Throws Exception	Special Value	Blocks	Times Out
Insert	add(o)	offer(o)	put(o)	offer(o, timeout, timeunit)
Remove	remove(o)	poll()	take()	poll(timeout, timeunit)
Examine	element()	peek()		

The 4 different sets of behaviour means this:

1. **Throws Exception**:
   If the attempted operation is not possible immediately, an exception is thrown.
2. **Special Value**:
   If the attempted operation is not possible immediately, a special value is returned (often true / false).
3. **Blocks**:
   If the attempted operation is not possible immedidately, the method call blocks until it is.
4. **Times Out**:
   If the attempted operation is not possible immedidately, the method call blocks until it is, but waits no longer than the given timeout. Returns a special value telling whether the operation succeeded or not (typically true / false).

It is not possible to insert null into a BlockingQueue. If you try to insert null, the BlockingQueue will throw a NullPointerException.
It is also possible to access all the elements inside a BlockingQueue, and not just the elements at the start and end. For instance, say you have queued an object for

processing, but your application decides to cancel it. You can then call e.g. remove(o) to remove a specific object in the queue. However, this is not done very efficiently, so you should not use these Collection methods unless you really have to.

BlockingQueue Implementations

Since `BlockingQueue` is an interface, you need to use one of its implementations to use it. The `java.util.concurrent` package has the following implementations of the `BlockingQueue` interface (in Java 6):

- **ArrayBlockingQueue**
- **DelayQueue**
- **LinkedBlockingQueue**
- **PriorityBlockingQueue**
- **SynchronousQueue**

Click the links in the list to read more about each implementation. If a link cannot be clicked, that implementation has not yet been described. Check back again in the future, or check out the JavaDoc's for more detail.

Java BlockingQueue Example

Here is a Java BlockingQueue example. The example uses the ArrayBlockingQueue implementation of the BlockingQueue interface.
First, the BlockingQueueExample class which starts a Producer and a Consumer in separate threads. The Producer inserts strings into a shared BlockingQueue, and the Consumer takes them out.

```
public class BlockingQueueExample {

    public static void main(String[] args) throws Exception {

        BlockingQueue queue = new ArrayBlockingQueue(1024);

        Producer producer = new Producer(queue);
        Consumer consumer = new Consumer(queue);

        new Thread(producer).start();
        new Thread(consumer).start();

        Thread.sleep(4000);
    }
}
```

Here is the Producer class. Notice how it sleeps a second between each put() call. This will cause the Consumer to block, while waiting for objects in the queue.

```
public class Producer implements Runnable{

    protected BlockingQueue queue = null;
```

```
    public Producer(BlockingQueue queue) {
        this.queue = queue;
    }

    public void run() {
        try {
            queue.put("1");
            Thread.sleep(1000);
            queue.put("2");
            Thread.sleep(1000);
            queue.put("3");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Here is the `Consumer` class. It just takes out the objects from the queue, and prints them to `System.out`.

```
public class Consumer implements Runnable{

    protected BlockingQueue queue = null;

    public Consumer(BlockingQueue queue) {
        this.queue = queue;
    }

    public void run() {
        try {
            System.out.println(queue.take());
            System.out.println(queue.take());
            System.out.println(queue.take());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

ArrayBlockingQueue

`ArrayBlockingQueue` is a bounded, blocking queue that stores the elements internally in an array. That it is bounded means that it cannot store unlimited amounts of elements. There is an upper bound on the number of elements it can store at the same time. You set the upper bound at instantiation time, and after that it cannot be changed.

The `ArrayBlockingQueue` stores the elements internally in FIFO (First In, First Out) order. The `head` of the queue is the element which has been in queue the longest time, and the `tail` of the queue is the element which has been in the queue the shortest time.

Here is how to instantiate and use an `ArrayBlockingQueue`:

```
BlockingQueue queue = new ArrayBlockingQueue(1024);
queue.put("1");

Object object = queue.take();
```

Here is a BlockingQueue example that uses Java Generics. Notice how you can put and take String's instead of :

```
BlockingQueue<String> queue = new ArrayBlockingQueue<String>(1024);

queue.put("1");

String string = queue.take();
```

DelayQueue

The DelayQueue blocks the elements internally until a certain delay has expired. The elements mus implement the interface java.util.concurrent.Delayed. Here is how the interface looks:

```
public interface Delayed extends Comparable<Delayed< {

 public long getDelay(TimeUnit timeUnit);

}
```

The value returned by the getDelay() method should be the delay remaining before this element can be released. If 0 or a negative value is returned, the delay will be considered expired, and the element released at the next take() etc. call on the DelayQueue.

The TimeUnit instance passed to the getDelay() method is an Enum that tells which time unit th delay should be returned in. The TimeUnit enum can take these values:

```
DAYS
HOURS
MINUTES
SECONDS
MILLISECONDS
MICROSECONDS
NANOSECONDS
```

The Delayed interface also extends the java.lang.Comparable interface, as you can see, which means that Delayed objects can be compared to each other. This is probably used internally in the DelayQueue to order the elements in the queue, so they are released ordered by their expiration time.

Here is an example of how to use the DelayQueue:

```
public class DelayQueueExample {

    public static void main(String[] args) {
        DelayQueue queue = new DelayQueue();

        Delayed element1 = new DelayedElement();

        queue.put(element1);

        Delayed element2 = queue.take();
    }
}
```

The DelayedElement is an implementation of the Delayed interface that I have created. It is not part of the java.util.concurrent package. You will have to create your own implementation of the Delayed interface to use the DelayQueue class.

LinkedBlockingQueue

The LinkedBlockingQueue keeps the elements internally in a linked structure (linked nodes). This linked structure can optionally have an upper bound if desired. If no upper bound is specified, Integer.MAX_VALUEis used as the upper bound.

The LinkedBlockingQueue stores the elements internally in FIFO (First In, First Out) order. The head of the queue is the element which has been in queue the longest time, and the tail of the queue is the element which has been in the queue the shortest time.

Here is how to instantiate and use a LinkedBlockingQueue:

```
BlockingQueue<String> unbounded = new LinkedBlockingQueue<String>();
BlockingQueue<String> bounded   = new LinkedBlockingQueue<String>(1024);

bounded.put("Value");

String value = bounded.take();
```

PriorityBlockingQueue

The PriorityBlockingQueue is an unbounded concurrent queue. It uses the same ordering rules as thejava.util.PriorityQueue class. You cannot insert null into this queue.

All elements inserted into the PriorityBlockingQueue must implement the java.lang.Comparable interface. The elements thus order themselves according to whatever priority you decide in your Comparableimplementation.

Notice that the PriorityBlockingQueue does not enforce any specific behaviour for elements t have equal priority (compare() == 0).

Also notice, that in case you obtain an Iterator from a PriorityBlockingQueue, the Iterator does no guarantee to iterate the elements in priority order.

Here is an example of how to use the PriorityBlockingQueue:

```
BlockingQueue queue   = new PriorityBlockingQueue();

    //String implements java.lang.Comparable
    queue.put("Value");

    String value = queue.take();
```

SynchronousQueue

The SynchronousQueue is a queue that can only contain a single element internally. A thread inseting an element into the queue is blocked until another thread takes that element fro the queue. Likewise, if a thread tries to take an element and no element is currently present, that thread is blocked until a thread insert an element into the queue.

Calling this class a queue is a bit of an overstatement. It's more of a rendesvouz point.

CountDownLatch

CountDownLatch is used to make sure that a task waits for other threads before it starts. To understand its application, let us consider a server where the main task can only start when all th required services have started.

Working of CountDownLatch:
When we create an object of CountDownLatch, we specify the number if threads it should wait fo all such thread are required to do count down by calling CountDownLatch.countDown() once the are completed or ready to the job. As soon as count reaches zero, the waiting task starts running

Example of CountDownLatch in JAVA:

```
/* Java Program to demonstrate how to use CountDownLatch, Its used when a thread
needs to wait for other threads before starting its work. */
import java.util.concurrent.CountDownLatch;

public class CountDownLatchDemo
{
    public static void main(String args[]) throws InterruptedException
```

```
    {
        // Let us create task that is going to wait for four
        // threads before it starts
        CountDownLatch latch = new CountDownLatch(4);

        // Let us create four worker threads and start them.
        Worker first = new Worker(1000, latch, "WORKER-1");
        Worker second = new Worker(2000, latch, "WORKER-2");
        Worker third = new Worker(3000, latch, "WORKER-3");
        Worker fourth = new Worker(4000, latch, "WORKER-4");
        first.start();
        second.start();
        third.start();
        fourth.start();

        // The main task waits for four threads
        latch.await();

        // Main thread has started
        System.out.println(Thread.currentThread().getName() + " has finished");
    }
}

// A class to represent threads for which the main thread waits.
class Worker extends Thread
{
    private int delay;
    private CountDownLatch latch;
    public Worker(int delay, CountDownLatch latch, String name)
    {
        super(name);
        this.delay = delay;
        this.latch = latch;
    }

    @Override
    public void run()
    {
        try
        {
            Thread.sleep(delay);
            latch.countDown();
            System.out.println(Thread.currentThread().getName()
                               + " finished");

        }
        catch (InterruptedException e)
        {
            e.printStackTrace();
        }
    }
}
```

Output:

```
WORKER-1 finished
WORKER-2 finished
WORKER-3 finished
WORKER-4 finished
main has finished
```

Facts about CountDownLatch:

1. Creating an object of CountDownLatch by passing an int to its constructor (the count), is actually num of invited parties (threads) for an event.
2. The thread, which is dependent on other threads to start processing, waits on until every other thread called count down. All threads, which are waiting on await() proceed together once count down reache zero.
3. countDown() method decrements the count and await() method blocks until count == 0

A java.util.concurrent.CountDownLatch is a concurrency construct that allows one or more threads to w for a given set of operations to complete.

A CountDownLatch is initialized with a given count. This count is decremented by calls to the countDown()method. Threads waiting for this count to reach zero can call one of the await() methods. Calling await()blocks the thread until the count reaches zero.

Below is a simple example. After the Decrementer has called countDown() 3 times on the CountDownLa the waiting Waiter is released from the await() call.

```
CountDownLatch latch = new CountDownLatch(3);

Waiter      waiter      = new Waiter(latch);
Decrementer decrementer = new Decrementer(latch);

new Thread(waiter)      .start();
new Thread(decrementer).start();

Thread.sleep(4000);
public class Waiter implements Runnable{

    CountDownLatch latch = null;

    public Waiter(CountDownLatch latch) {
        this.latch = latch;
    }

    public void run() {
        try {
            latch.await();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Waiter Released");
    }
}

public class Decrementer implements Runnable {

    CountDownLatch latch = null;

    public Decrementer(CountDownLatch latch) {
        this.latch = latch;
    }

    public void run() {

        try {
            Thread.sleep(1000);
```

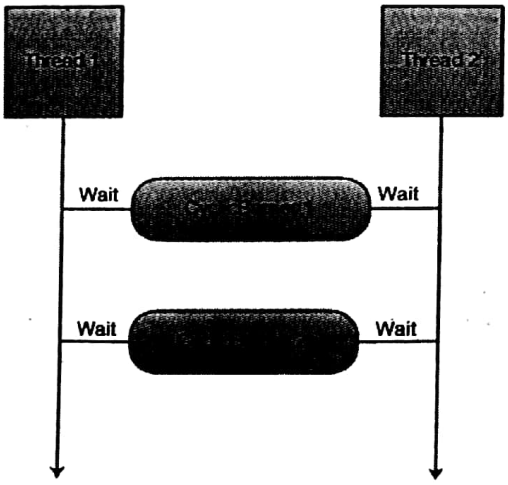
```
            this.latch.countDown();

            Thread.sleep(1000);
            this.latch.countDown();

            Thread.sleep(1000);
            this.latch.countDown();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

CyclicBarrier

The java.util.concurrent.CyclicBarrier class is a synchronization mechanism that can synchronize threads progressing through some algorithm. In other words, it is a barrier that all threads must wait at, until all threads reach it, before any of the threads can continue. Here is a diagram illustrating that:

The threads wait for each other by calling the await() method on the CyclicBarrier. Once N threads are waiting at the CyclicBarrier, all threads are released and can continue running.

Two threads waiting for each other at CyclicBarriers.



Creating a CyclicBarrier

When you create a CyclicBarrier you specify how many threads are to wait at it, before releasing them. Here is how you create a CyclicBarrier:

```
CyclicBarrier barrier = new CyclicBarrier(2);
```

Waiting at a CyclicBarrier

Here is how a thread waits at a `CyclicBarrier`:

```
barrier.await();
```

You can also specify a timeout for the waiting thread. When the timeout has passed the thread is also released, even if not all N threads are waiting at the CyclicBarrier. Here is how you specify a timeout:

```
barrier.await(10, TimeUnit.SECONDS);
```

The waiting threads waits at the CyclicBarrier until either:

- The last thread arrives (calls await())
- The thread is interrupted by another thread (another thread calls its interrupt() method)
- Another waiting thread is interrupted
- Another waiting thread times out while waiting at the CyclicBarrier
- The CyclicBarrier.reset() method is called by some external thread.

CyclicBarrier Action

The CyclicBarrier supports a barrier action, which is a Runnable that is executed once the last thread arrives. You pass the Runnable barrier action to the CyclicBarrier in its constructor, like this:

```
Runnable       barrierAction = ... ;
CyclicBarrier barrier        = new CyclicBarrier(2, barrierAction);
```

CyclicBarrier Example

Here is a code example that shows you how to use a `CyclicBarrier`:

```
Runnable barrier1Action = new Runnable() {
    public void run() {
        System.out.println("BarrierAction 1 executed ");
    }
};
Runnable barrier2Action = new Runnable() {
    public void run() {
        System.out.println("BarrierAction 2 executed ");
    }
};
```

```
CyclicBarrier barrier1 = new CyclicBarrier(2, barrier1Action);
CyclicBarrier barrier2 = new CyclicBarrier(2, barrier2Action);

CyclicBarrierRunnable barrierRunnable1 =
        new CyclicBarrierRunnable(barrier1, barrier2);

CyclicBarrierRunnable barrierRunnable2 =
        new CyclicBarrierRunnable(barrier1, barrier2);

new Thread(barrierRunnable1).start();
new Thread(barrierRunnable2).start();
```

Here is the `CyclicBarrierRunnable` class:

```
public class CyclicBarrierRunnable implements Runnable{

    CyclicBarrier barrier1 = null;
    CyclicBarrier barrier2 = null;

    public CyclicBarrierRunnable(
            CyclicBarrier barrier1,
            CyclicBarrier barrier2) {

        this.barrier1 = barrier1;
        this.barrier2 = barrier2;
    }

    public void run() {
        try {
            Thread.sleep(1000);
            System.out.println(Thread.currentThread().getName() +
                               " waiting at barrier 1");
            this.barrier1.await();

            Thread.sleep(1000);
            System.out.println(Thread.currentThread().getName() +
                               " waiting at barrier 2");
            this.barrier2.await();

            System.out.println(Thread.currentThread().getName() +
                               " done!");

        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (BrokenBarrierException e) {
            e.printStackTrace();
        }
    }
}
```

Here is the console output for an execution of the above code. Note that the sequence in which the threads gets to write to the console may vary from execution to execution. Sometimes Thread-0 prints first, sometimes Thread-1 prints first etc.

```
Thread-0 waiting at barrier 1
Thread-1 waiting at barrier 1
```

```
BarrierAction 1 executed
Thread-1 waiting at barrier 2
Thread-0 waiting at barrier 2
BarrierAction 2 executed
Thread-0 done!
Thread-1 done!
```