Day14: Multithreading, Executor Framework

Multithreading In Java:

Multithreading is a programming concept in which the application can create a small unit of tasks to execute in parallel. If you are working on a computer, it runs multiple applications and allocates processing power to them. A simple program runs in sequence and the code statements execute one by one. This is a single-threaded application. But, if the programming language supports creating multiple threads and passes them to the operating system to run in parallel, it's called multithreading.

Multithreading in Java is a process of executing multiple threads simultaneously.

A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory areas so saves memory, and context-switching between the threads takes less time than process.

Advantages of Java Multithreading:

1. It **doesn't block the user** because threads are independent and you can perform multiple operations at the same time.
2. You **can perform many operations together, so it saves time**.
3. Threads are **independent**, so it doesn't affect other threads if an exception occurs in a single thread.

Some of application areas to apply multithreading:

- To develop multimedia graphics
- To develop animations
- To develop video games
- To develop Web Servers

Multitasking:

Multitasking is the process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved in two ways:

- Process-based Multitasking (Multiprocessing)
- Thread-based Multitasking (Multithreading)

Process-based Multitasking (Multiprocessing):

- Each process has an address in memory. In other words, each process allocates a separate memory area.
- A process is heavyweight.

- The cost of communication between the process is high.
- Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists, etc.

Thread-based Multitasking (Multithreading):

- Threads share the same address space.(a thread is a part of a process)
- A thread is lightweight.
- The cost of communication between the thread is low.

Note: At least one process is required for each thread.

2 Ways to create threads

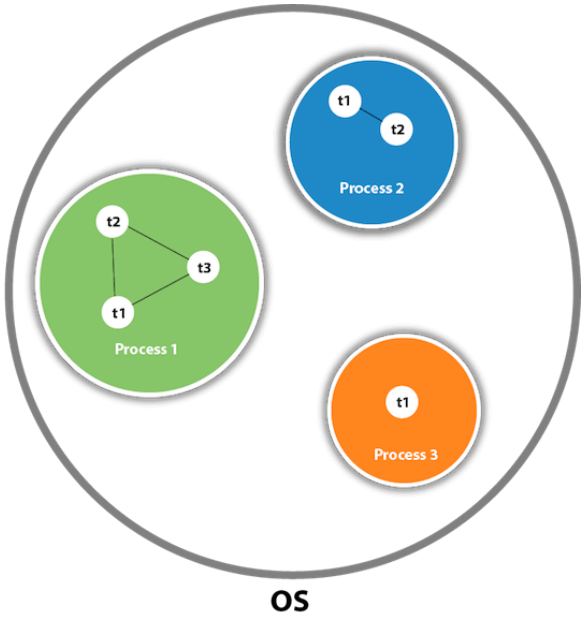
Thread in java:

An application when it is under execution is called a process.

A thread is a part or sub-process of an application.

A thread is a lightweight sub-process, the smallest unit of processing. It is a separate path of execution.

A thread is a separate flow of execution that execute some functionality of a program with the other part of the program simultaneously.



Note: In Java, every program/application has a default flow of execution, a default thread, it is called the main thread. if we can start another flow of execution(another thread) along with the main thread simultaneously then it is called a multithreaded application or program.

Implementing thread in java:

Implementing thread in java is a two-step process:

1. first of all, we have to define a functionality that can be executed as a thread along with the main thread.
2. Start that functionality as a thread.

Thread class and Runnable interface are the two structures using which we implement Thread based multitasking in java.

The signature of a function using which we implement a thread is defined in an interface by the name **Runnable.**

This **Runnable** interface belongs to **java.lang** package. it has only one abstract method:

```
public void run();
```

Inside this run() method we need to define the functionality, that we want to execute as a thread along with the main thread. after providing the body we need to execute this functionality as a thread (i.e. simultaneously with the other part of the program)

There is a class by the name **Thread** present in **java.lang** package, which has a method called **start()**, this start() method is used to execute a given functionality defined inside the run() method of **Runnable** interface as a separate thread.

This start() method of the Thread class recognize the run() method of the Runnable interface and then the run() method is executed as a separate individual thread.

We implement threads either of the following two ways:

1. By implementing Runnable Interface
2. By extending Thread class

Example:

```
class A implements Runnable{

  @Override
  public void run(){
    //define the taks which we want to execute as a thread
  }
}

//or

class A extends Thread
{
  @Override
  public void run(){
    //define the taks which we want to execute as a thread
  }

}
```

Note: Internally the Thread class implements the Runnable interface and override run() method with empty implementation.

Example:

```
public class Thread implements Runnable{

  public void run(){
    //it is empty body overriden from Runnable interface
  }

  public void start(){//this is thread class own method....
  }

  //other methods of the Thread class

}
```

Note:-weather we extends Thread class or implement Runnable interface directly, we have to use run() method of the Runnable interface.

Example : Creating a new thread by extending the Thread class:

```
class X extends Thread{

  @Override
  public void run(){
    for(int i=0;i<30;i++){
      System.out.println("inside run mehod "+i);
    }
    System.out.println("end of run() method");
  }

  public static void main(String[] args){

    //------here one thread (main)

    X x1=new X();

    //x1.run();//it will be called as a normal method.
    x1.start();//here second thread will start

    for(int i=25;i<60;i++){
        System.out.println("inside main mehod"+i);
    }

    System.out.println("end of main()..");
  }
}
```

Here functionality of the start() method is to pick the run() method present in the object on which the start() method is called and to handover this run() method to the thread-scheduler for the Scheduling.
Control will be in the main() method and other statements of main() will be executed simultaneously along with run() method.

Since both the threads are getting executed simultaneously, the start/end execution of a thread completely depends on the time slice allocated for each method thread

Because of scheduling we can't guess the output of the above application.

Responsibility of start() method of Thread class:

The start() method present in the Thread class is responsible to perform all the mandatory activity which are required for our thread. (like, registering our thread with the Thread-scheduler, performing all the low level task to start a separate flow of execution. and then calling the run() method).

Hence without executing thread class start() method there is no chance of starting new thread in Java.

After starting a thread we are not allowed to restart the same thread once again otherwise we will get a runtime exception called **IllegalThreadStateException**.

Defining multiple Thread simultaneously:

```
class ThreadA extends Thread {
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println("inside run mehod of ThreadA" + i);
        }
        System.out.println("end of ThreadA");
    }
}

    class ThreadB extends Thread {
    @Override
    public void run() {
        for (int i = 50; i < 60; i++) {
            System.out.println("inside run mehod of ThreadB" + i);
        }
        System.out.println("end of ThreadB");
    }
    }

    class ThreadC extends Thread {
    @Override
    public void run() {
            for (int i = 20; i < 30; i++) {
                System.out.println("inside run mehod of ThreadC" + i);
            }
            System.out.println("end of ThreadC");
        }
    }

  class Main {

        public static void main(String[] args) {

            ThreadA t1 = new ThreadA();
            ThreadB t2 = new ThreadB();
            ThreadC t3 = new ThreadC();



            t1.start();
            t2.start();
            t3.start(); //4

            for (int i = 70; i < 80; i++) {
                System.out.println("inside main of Test:" + i);
            }
            System.out.println("end of main");
        }
    }
```

If we extends Thread class to create a new thread then we loose the chance of Object Orientation biggest advantage of inheritance i.e. we cannot extends another class simultaneously.

To solve the above problem, we use the Runnable interface, so we take a separate class by implementing Runnable interface and overrides the run() method and inside the run() method define the functionality which we want to execute as a separate thread,

then supply the object of that class to the Thread class constructor and create Thread class object and start that thread.

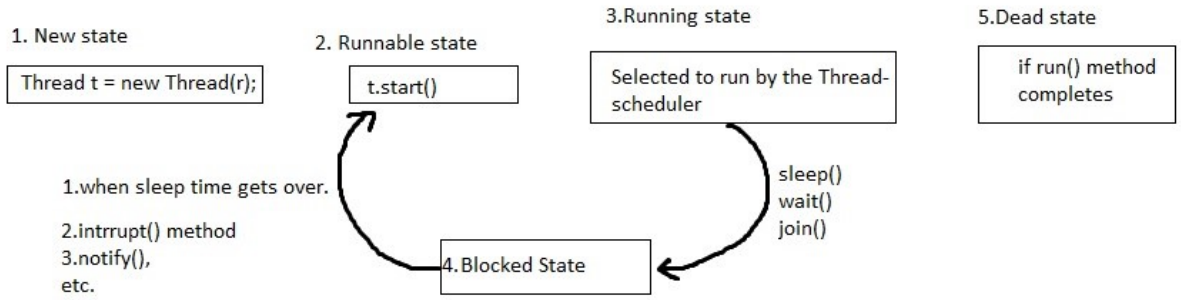
Example: Starting a thread using **Runnable** interface

```
class ThreadA implements Runnable{
  @Override
  public void run(){
    for(int i=20;i<40;i++){
      System.out.println("inside run() of ThreadA"+i);
    }
  }
  public static void main(String[] args){

    ThreadA t1=new ThreadA();
    Thread t=new Thread(t1); //passing Runnable object to the constructor of Thread class
    t.start();

    for(int i=20;i<40;i++){
      System.out.println("inside main of ThreadA:"+i);
    }
  }
}
```

Life-cycle of a Thread (State of a thread):

In Java, a thread always exists in any one of the following states. These states are:

1. New state
2. Runnable state
3. Running state
4. Blocked state
5. Dead state



Getting and setting name of a thread:

Every thread in java has some name, it may be default name generated by the JVM or explicitly provided by the programmer.

Thread class has provides following getter and setter methods to get and set name of a thread.

public final String getName();

public final void setName(String name);

Example:

```
class ThreadA implements Runnable{

  @Override
  public void run() {

    for(int i=0;i<20;i++){
      String tname=Thread.currentThread().getName();
      System.out.println(tname +" is running ");
    }
  }
}

class ThreadB implements Runnable{

  @Override
  public void run() {

    for(int i=0;i<20;i++){
      String tname=Thread.currentThread().getName();
      System.out.println(tname +" is running ");
    }
  }
}


public class Main {

  public static void main(String[] args) {

    ThreadA ta=new ThreadA();
    ThreadB tb=new ThreadB();
```

```
    Thread t1=new Thread(ta);
    Thread t2=new Thread(tb);

    t1.setName("Raj");
    t2.setName("simran");

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

  }
}
```

Note:- in the above example, two separate thread executes on the two different objects simultaneously.

Running two or more thread on a same/single object:

We can run multiple threads on a single object also, that means one single functionality (run() method) of one object will be executed by two separate thread separately and individually in their own call-stack.
It's like asking two persons to build a wall.

```
public class RunThread implements Runnable{

  public void run(){

    for(int i=0;i<25;i++){
      String tname=Thread.currentThread().getName();
      System.out.println(tname +": is running");


    }
  }

  public static void main(String[] args){

    RunThread job=new RunThread();

    Thread one=new Thread(job);
    Thread two=new Thread(job);

    one.setName("Dhoni thread..");
    two.setName("Kohli thread..");

    one.start();
    two.start();
  }
}
```

Here also, if we run this program multiple time, we can't predict the output.

Thread Priorities:

Every thread in java has some priority. it may be default priority generated by the JVM or explicitly provided by programmer.

The valid range of thread priority is 1 to 10. but not 0 to 10.

1 is least and 10 is highest.

Note :- priority should be assigned to the corresponding thread before calling the start() method.

Thread class defines the following constant to represent standard priorities:-

1. Thread.MAX_PRIORITY---->10
2. Thread.MIN_PRIORITY---->1
3. Thread.NORM_PRIORITY--->5

Thread scheduler will use these priorities while allocating processer to our thread.

The thread which is having highest priority will get chance first.

If two threads having the same priority then we can't expect the exact execution order.it depends on the thread scheduler.

Thread class defines the following setter and getter methods to get and set priority of a thread.

1. **public final int getPriority();**
2. **public final void setPriority(int priority);** it allows values from 1 to 10 otherwise we will get a runtime exception.

Note:-the default priority for main thread is 5 and for all remaining thread it will be inherited from parent to child.

A high priority thread does not run faster than lower priority thread.

Thread priority is not a rule for thread-scheduler, it is just a hint. so no guarantee for the execution (it will work only if thread is in the waiting state or if there is a limited CPU time).

• If we are using thread priority for thread scheduling then we should always keep in mind that the underlying platform should provide support for scheduling based on thread priority.

Example:

```
class Main extends Thread {

    public void run(){
        System.out.println("Inside run method");
    }

    public static void main(String[] args){

         Main t1 = new Main();
         Main t2 = new Main();
         Main t3 = new Main();

        System.out.println("t1 thread priority : " + t1.getPriority());

        System.out.println("t2 thread priority : " + t2.getPriority());

        System.out.println("t3 thread priority : " + t3.getPriority());

        t1.setPriority(2);
        t2.setPriority(5);
        t3.setPriority(8);

        // t3.setPriority(21); will throw IllegalArgumentException

        System.out.println("t1 thread priority : " + t1.getPriority());

        System.out.println("t2 thread priority : " + t2.getPriority());

        System.out.println("t3 thread priority : " + t3.getPriority());
```

```
        System.out.println("Currently Executing Thread : " + Thread.currentThread().getName());

        System.out.println("Main thread priority : "+ Thread.currentThread().getPriority());

        Thread.currentThread().setPriority(10);

         System.out.println("Main thread priority : "+Thread.currentThread().getPriority());
    }
}
```

Suspending a thread:

Once the run() method has been handed over to the thread-scheduler then,
some time it is necessary to control the execution of the thread which is under execution.

For that purpose, we have some functionality(methods) inside the Thread class which helps us in gaining partial control over the execution of run() method which are already scheduled by the thread-scheduler.

A thread which is already under execution can be suspended (prevented from getting executed further) or we can control their execution based on three criteria:

1. **Time**
2. **Conditionally**
3. **Unconditionally.**

Suspending a thread based on time:

There is a static method by name **sleep(long ms)** inside the Thread class which takes time in milliseconds as an argument
Example:

Thread.sleep(1000);

This method will suspend the current thread which is under execution with those many millisecond passed as argument(1000ms=1sec).
This sleep method is proven to generate checked exception hence it must be called inside try and catch block.

Note : we cannot use throws with run() method, it will violate the method override rule.

The main reason is that there's no-one to catch the exception, except for any catch-all handlers you might register with your `Thread` or the containing `ThreadGroup`

Example:

```
class Main implements Runnable {

    @Override
    public void run() {


        for (int i = 0; i < 5; i++) {
            System.out.println("inside run " + i);
            try {
```

```
                Thread.sleep(1000);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }

        System.out.println("end of run()...");
    }

    public static void main(String[] args) {

        Main job = new Main();
        Thread t1 = new Thread(job);

        t1.start();

        System.out.println("end of main()...");
    }
}
```

Suspending a thread conditionally:

join() method

If we want to suspend a running thread conditionally then we should use join() method of the Thread class.

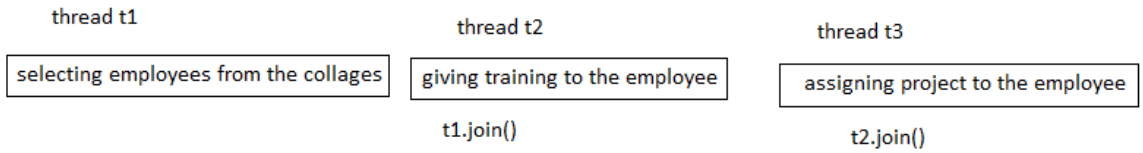
The join() method is a non-static method.

If a thread wants to wait until completion of other thread then we should use join() method.

If a thread t1 calls join() method on another thread t2, like t2.join() then t1 thread will be enter into waiting state until t2 thread completes.

Let's assume that we have two threads available t1 and t2.

now if we have a condition that inside run() method of t1 we need to use some of the values calculated in run() method of t2,then in this case we have to stop the execution of run() method of t1 until the run() method of t2 is completely executed. in such situation we have to make use of join() method.



Here t2 thread will wait until completion of t1 thread and t3 thread will wait until the completion of t2 thread.

I Problem:

Let’s take a thread that will calculate the sum of 1 to 10 number, and another thread (main thread) will print the calculated sum value of first thread.

```
class A implements Runnable {

    int sum;

    public void run() {

        for (int i = 0; i < 10; i++) {
            System.out.println("inside A thread");
            sum = sum + i;
        }
    }
}

class Main {

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

        A a1 = new A();

        Thread t = new Thread(a1);

        t.start();

        t.join();//Here main thread will wait until the t thread completes
                 //if we comment t.join then we will get incorrect value.
        int result = a1.sum;

        for (int i = 0; i < 10; i++) {
            System.out.println("inside main thread....");
            System.out.println(result);

        }
    }
}
```

Thread-safety in Java:

The concept of avoiding multiple threads acting upon the same functionality simultaneously is known as Thread-safety.

If multiple threads are trying to operate simultaneously on the same functionality then there may be a chance of data inconsistency problem.

Concurrency issue lead to the **race-condition**. and **race-condition** lead to the data inconsistency.

Race-condition:

Java is a multi-threaded programming language and there is a higher risk to occur race conditions. Because the same resource may be accessed by multiple threads at the same time and may change the data.

A race-condition is a condition in which the critical section (a part of the program where shared memory is accessed) is concurrently executed by two or more threads. It leads to incorrect behavior of a program.

In layman terms, a **race condition** can be defined as, a condition in which two or more threads compete together to get certain shared resources.

For example, if thread A is reading data from the linked list and another thread B is trying to delete the same data. This process leads to a race condition that may result in run time error

To solve the data inconsistency problem in java **synchronized** keyword is used.
So the thread-safety is achieved and race condition is avoided by the help of **synchronized** keyword.

Example:

```
class Common{
  public void fun1(String name){

  System.out.print("Welcome");
  try{
    Thread.sleep(1000);
  }
  catch(Exception ee){
  }
  System.out.println(name);

}
```

the above method fun1() is supposed to give the output as welcome and after one second print the supplied name. Now what will happen if two threads acts on this fun simultaneously.

```
 class ThreadA extends Thread{

  Common c;
  String name;

  public ThreadA(Common c,String name) {
    this.c=c;
    this.name=name;
  }

  @Override
  public void run() {
    c.fun1(name);
  }
}

 class ThreadB extends Thread{

  Common c;
  String name;

  public ThreadB(Common c,String name) {
    this.c=c;
    this.name=name;
  }

  @Override
  public void run() {
    c.fun1(name);
```

```
  }
}

class Main{

  public static void main(String[] args){

    Common c=new Common();

    //sharing same Common object to two thread
    ThreadA t1=new ThreadA(c,"Ram");
    ThreadB t2=new ThreadB(c,"Shyam");

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

Now the output will be **Welcome Welcome Ram Shyam** which is not expected.

We can get the desired output if we avoid two thread acting on fun1() simultaneously.

To achieve this requirement we need to make fun1() as **synchronized**.

Note: The **synchronized** keyword applicable only for methods and blocks but not for variables and classes.

If a method or block is declared as **synchronized** then at a time only one thread is allowed to execute that method or block on a given object so that data inconsistency problem will be resolved.

The main advantage of the **synchronized** keyword is we can resolve data inconsistency problem. but the main disadvantage of the **synchronized** keyword is it increases waiting time of the threads and creates performance problem on it. hence if there is no specific requirement then it is never recommended to use the **synchronized** keyword.

Example:

> checking seat availability method should be non-synchronized, where as book seat method should be synchronized
>
> Any method that changes the state of an object. i.e. add/updaxte/delete/replace method we should use as synchronized.

Synchronization concept:

- Internally synchronization concept is implemented by using lock concept.
- Every object in a java has a unique lock. most of the time the lock is unlocked.
- When an object has one or more synchronized methods ,a thread can enter into a synchronized method or block only when If that thread have the lock of that object.
- The locks are not per methods basis, instead they are per object basis.
- The thread won't release the lock until it completes the synchronized methods, so while that thread is holding the lock of that object. once a synchronized method execution completed then thread releases the lock automatically.

- Until the lock is released (completion of synchronized method.)no other threads can enter any of the synchronized methods or blocks of that object.
- So if an object has synchronized methods or blocks, a threads can enter any one of the synchronized methods or block only if the lock of that object is available.
- Acquiring and releasing the lock internally taken care by JVM, programmer are not responsible for this activity.

Note: while one thread is executing any one synchronized method on the given object then the remaining threads are not allowed to execute any other synchronized method simultaneously. but remaining threads are allowed to execute any non-synchronized method simultaneously.

Example:

```
class A{

synch funA(){}//ATM1

synch funB(){}//ATM2

funC(){}//Backery Shop

}
```

Here if one thread try to execute funA on A class object then it needs the lock of that object, once it acquires the lock it starts the execution of that method funA, while executing funA by one thread, other threads are not allowed to execute funA and even funB() also, but other threads can executes funC() simultaneously.

Class Level lock:

In our previous Common class example

if there are two threads try to operate on "two different object" of **Common.java** class, then we will get the irregular output even though fun1() is **synchronized,** because both threads operates on two different objects, and they are holding the locks of two different objects.

Example:

```
class Test{

public static void main(String[] args){

  Common c1=new Common();
  Common c2=new Common();

  ThreadA t1=new ThreadA(c1,"Ram");
  ThreadB t2=new ThreadB(c2,"Shyam");  wel wel

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

```
  }
}
```

But if we mark the **synchronized fun1()** method of class Common as "**static**" then we will get the regular output irrespective of multiple objects also.

The reason behind this is because :

In java as there is a unique lock for each object of a class, similarly there is a unique lock for each class also.

So there are two types of lock in java:

1. object level lock(it is unique for each object of a class)
2. class level lock(it is unique for each class)

If a thread try to execute a static synchronized method then it required class level lock.

object lock and class level lock both are independent and there is no link between them.

While a thread is executing a static synchronized method, the remaining threads are not allowed to execute any other static synchronized method of that class simultaneously (**even on the multiple object of that class also)
but remaining threads are allowed to execute normal static and synchronized non-static methods and normal non-static methods simultaneously.

.

Synchronized block:

If very few lines of the code requires synchronization then it is not recommended to declare entire method as synchronized. we have to enclose those few lines of the code in synchronized block.

In a method, assume 10000 lines code is there, and in the middle somewhere few line need some Database operation like(update/delete/add) then declaring entire method as synchronized is a worst kind of programming. here it degrades the performance.

The main advantage of synchronized block over synchronized method is ,it reduces the waiting time of the threads and improves the performance of the application.

- [ ] Resume from here

There are following syntax of the synchronized block:

1. synchronized block to get a object level lock of the same class:

   Example

```
void fun1(){

System.out.println("do something...");

//1 thousand lines of codes are there

synchronized(this){--//here if a thread gets the lock of current obj then it is allowed to execute
      //this block
```

```
  System.out.println("do some more thing1");
}

//1 thousand lines of codes r there

}
```

2. synchronized block to get a object level lock of different object

   Example:

```
A a1=new A();
A a2=new A();

void fun1(){

System.out.println("do something...");

synchronized(a1){//if a thread gets the lock of a1 object of A class
                  //(not a2 obj of A class) then only it is allowed
                  //to execute  the following block of code

System.out.println("do some more thing1");
System.out.println("do some more thing2");

}
```

3. To get a class level lock:

   Example

```
synchronized(A.class){//if a thread gets the class level lock of class A
                      //then only it is allowed to execute following block

  System.out.println("do some more thing1");
}
```

I Problem:

Race Condition Demo

Another Example (We problem)

Lets write the fun1() method of Common class using synchronized block to get the object level lock.

```
class Common {

  public void fun1(String name){

    synchronized(this){

    System.out.print("Welcome :");

    try {
      Thread.sleep(1000);
    } catch (InterruptedException e) {
      e.printStackTrace();
```

```
    }

    System.out.println(name);
    }

  }
}
```

Inter-thread Communication or Thread-synchronization:

It means two synchronized threads communicate each other.

Two synchronized thread can communicate each other by using some methods present in Object class, those methods are wait(), notify(), notifyAll().

By using above methods we can gain partial control on the scheduling mechanism which is supervised by the thread-scheduler.

To gain this partial control the threads should have a sign of mutual understanding between them .they should be able to communicate with each other.

Whenever we need to suspend a synchronized thread unconditionally then we use wait() method.

Whenever we need to resume a suspended(waiting) thread then we use notify() method.

this is known as thread-synchronization or inter-thread communication.

In the inter-thread communication the thread which require updation it has to call wait() method.
The thread which performing updation it will call notify() method, so that waiting thread will gets the notification and it continues its execution with those updation.

Note:- we can call wait(), notify(),notifyAll() only in the synchronized block or synchronized methods. otherwise we will get a runtime exception.

To call wait() or notify() method on any object we must have that particular object lock otherwise we will get a runtime exception called IllegalMonitorStateException.

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

Once a thread calls notify() method on any object it also releases the lock of that object but not immediately.

Wait and notify or notifyAll method belongs from Object class because "a thread" can call these methods on any java object.

Example:

```
class MyThread extends Thread
{

  int total=0;

  public void run(){

    for(int i=0;i<=100;i++){

        total=total+i;
    }
//10000

  }
}



class Main{

public static void main(String[] args){

  MyThread mt=new MyThread();

  mt.start();

  System.out.println(mt.total);
}
}
```

Here the main method of class Main is requires updation and here MyThread class run method performing updation,

Here if we run the above application normally then there may be a chance of:

a. getting the total value
b. getting the value as 0
c. getting some partial value.

because its up to the thread scheduler which thread is scheduled.

to get the correct output we have 3 ways:-

1. either main thread calls the sleep() method, so our main thread will enter into sleep state and other thread perform complete operation.
   but it is not recommended because if updation is completed in nano second then extra time will be wasted. or if the updation will take too much time then we will get intermediate value.
2. main thread calls mt.join() method here until the run() method of MyThread completely finished main thread will be in waiting state. and suppose if 10 thousand lines are there after that for loop then main has to wait till all the lines of run method completely executes.
3. main thread calls the wait() method on the mt obj ,but remember if main thread try to call wait method then main thread must have the lock of mt obj. otherwise we gets an exception.

Solution :

```
class MyThread extends Thread{

 int num=0;

 @Override
 public void run() {

   synchronized (this) {

     System.out.println("child thread performing calculation");
     for(int i=0;i<=100;i++){
       num=num+i;
     }

     System.out.println("child thread giving the notification");
     this.notify();

   }

  }
}

 class Test{

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

    MyThread mt=new MyThread();

    mt.start();
    //Thread.sleep(5000);
    synchronized (mt) { //getting the lock of mt object
      System.out.println("main thread calls the wait method");

      mt.wait();

      System.out.println("main thread got the notification");

      System.out.println(mt.num);
    }
  }
}
```

Note:-In the above example most of the time main thread gets the chance first but if we call Thread.sleep() method after the mt.start() then child thread gets the chance and the main thread will go to the waiting state forever, to solve this problem we can use wait(5000) i.e.wait till 5sec.

You Problem:

Write the fun1() method of Common class using synchronized block to get the Class level lock.

Deadlock:

A lock without key is nothing but deadlock.

If two synchronized threads are waiting for each other forever(for infinite time).

The **synchronized** keyword is the only reason for deadlock.

There is no any solution for the deadlock but there are several prevention is there.

Example:

```
 class A  {

  public synchronized void funA(B b1){

    System.out.println("funA of A starts");

    b1.fun2();

    System.out.println("funA of A ends");

  }

  public synchronized void fun1(){
    System.out.println("inside fun1 of A");
  }
}


 class B {

  public synchronized void funB(A a1){

    System.out.println("funB of B starts");

    a1.fun1();

    System.out.println("funB of B ends");
  }

  public synchronized void fun2(){
    System.out.println("inside fun2 of B");
  }

}

 class ThreadA extends Thread{

  A a1;
  B b1;

  public ThreadA(A a1,B b1) {
    this.b1=b1;
    this.a1=a1;
  }


  @Override
  public void run() {

    a1.funA(b1);
  }
}

 class ThreadB extends Thread{


  A a1;
  B b1;

  public ThreadB(A a1,B b1) {
    this.b1=b1;
    this.a1=a1;
```

```
  }


  @Override
  public void run() {

    b1.funB(a1);
  }
}


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

    A a1 = new A();
    B b1 = new B();

    ThreadA t1 = new ThreadA(a1, b1);

    ThreadB t2 = new ThreadB(a1, b1);

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

  }

}
```

Here if any method of class A or class B we remove the synchronized keyword then it will not become deadlock.

Executor Framework or ThreadPool:

Consider the following example:

```
Runnable task = () -> System.out.println(“Hello World “ );

Thread thread= new Thread(task);

thread.start();
```

In the above example:

1. A task is an instance of Runnable
2. The task is passed to a new instance of Thread
3. The Thread is launched
4. The thread is created on demand ,by user
5. Once the task is done, thread dies.
6. Thread are expensive resource.

How can we improve the use of Threads ,as a resources ?

By creating pools of ready to use threads ,and using them.

Creating a new thread for every task may creates performance and memory problems, to overcome this we should go for Thread pool.

Thread pool is a pool of already created threads ready to do our tasks.

Thread pool framework is also known as executor framework. this concept is introduced in java5.

Thread pool related API comes in the form of **java.util.concurrent** package.

Without Thread pool, if we have 10 different independent tasks are there then we need to create 10 separate threads.

But using Thread pool concept ,we create a Thread pool of 5 threads and submit all the 10 tasks to this Thread pool.
Here a single thread can perform multiple independent tasks. so that performance will be improved.

We can create Thread Pool as follows:

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

Here we have created the pool of 3 threads.

After creating the pool we need to submit the tasks to this pool.

```
service.submit(task); // here task is the object of Runnable.
```

We Problem:

Execute 6 task by using 3 different threads.

```
class PrintJob implements Runnable{

    String name;
    PrintJob(Sting name){
      this.name=name;
    }

    public void run(){
      System.out.println(name +" job started by Thread :"+Thread.currentThread().getName());

    try{
        Thread.sleep(5000);
      }catch(InturreptedException e){
          e.printStackTrace();
      }

      System.out.println(name +"..job completed by Thread :"+Thread.currentThread().getName());
    }
}


class Main{
```

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

    PrintJob[] jobs={
           new PrintJob("Ram"),
           new PrintJob("Ravi"),
           new PrintJob("Anil"),
           new PrintJob("Shiva"),
           new PrintJob("Pawan"),
           new PrintJob("Suresh")
     };

ExecutorService service = Executors.newFixedThreadPool(3);

for(PrintJob job:jobs){

    service.submit(job);
}

  service.shutdown();//to shutdown the executorService.
}
}
```

[Further Reading]

Callable and Future:

In the case of Runnable tasks ,Threads won't return anything after completing the task.

If a thread is required to return some result after execution, then we should use the Callable Interface instead of Runnable.

Callable interface belongs from **java.util.concurrent** package.

Callable interface also contains only one method:

```
public Object call()throws Exception
```

Note:- the object of Callable we can't pass to the normal Thread class constructor unlike Runnable object, here we need to use the ExecutorService class help.

if we submit a Callable object to **ExecutorService** object, then after completing the task, thread returns an object of the type **Future**.

The **Future** object can be used to retrieved the result from the Callable tasks.

Example:

```
import java.util.concurrent.*;
 class MyCallable implements Callable{

  int num;

  public MyCallable(int num) {
    this.num = num;
  }
```

```
  @Override
  public Object call() throws Exception {

    System.out.println(Thread.currentThread().getName()+" .. is responsible to find the sum of first "+num+" numbers");

    int sum=0;

    for(int i=0;i<=num;i++){
      sum = sum+i;
    }
    return sum;
  }

}


class Main{

  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();
  }
}
```

Difference Between Runnable and Callable:

Runnable	**Callable**
If a thread won't returns anything.	If a Thread returns anything
only one method **public void run()**	only one method **public Object call() throws Exception**
return type void	return type is Object
if any exception raise compulsory we need to handle within try catch.	not required to use try-catch
Belongs to java.lang package	Belongs to java.util.concurrent package
from java 1.0 version	from java 1.5 version

Suspending a thread unconditionally:

yield() method:

This yield() method is a static method defined inside the Thread class, it is used to pause the current executing thread for giving the chance to remaining waiting thread of same priority, if there is no any waiting thread or all waiting threads have low priority then same thread will get the chance once again for the execution.

suspend() method:

In order to suspend a thread unconditionally, we make use of non-static method suspend() present in a Thread class.

resume() method:

In order to resume a thread which has been suspended long back we use resume() method, it is also a non-static method defined inside the Thread class.

Note: suspend() and resume() method are deprecated methods. we should not use those methods. we should not use any deprecated methods.

The alternate methods are **wait()** and **notify()**. these methods are the non-static method defined inside the **Object** class.

Difference between yield() and wait() method:

The wait() method will suspend a thread till notify() method is called, whereas yield() method says right now it is not important please give the chance to another thread of same priority.