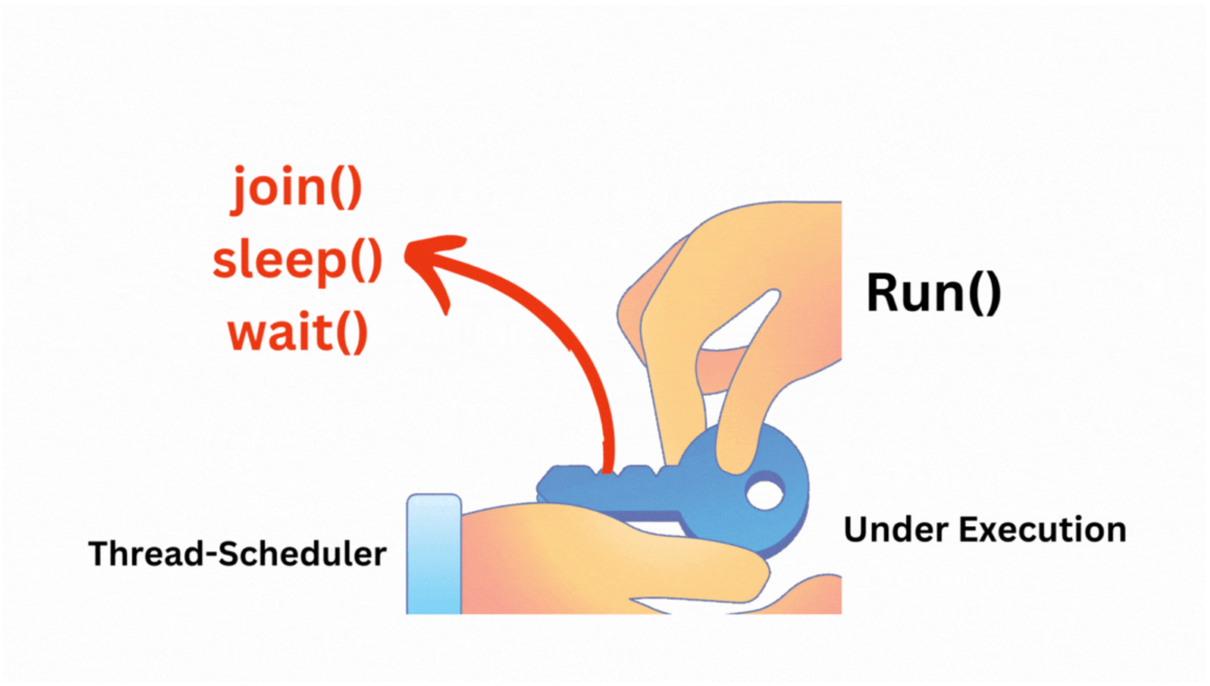
# Day 2: wait, notify, deadlock, a race condition

## Suspending a thread:



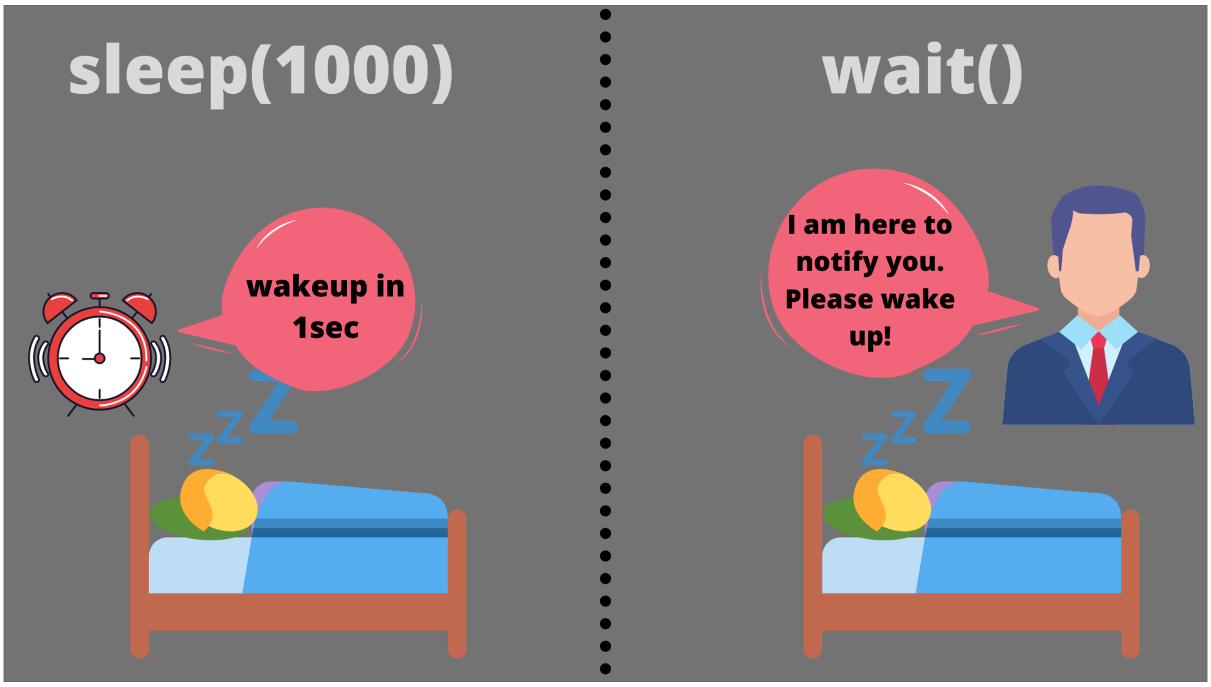
Once the run() method has been handed over to the thread-scheduler then, some time it is necessary to control the execution of the thread which is under execution.

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.

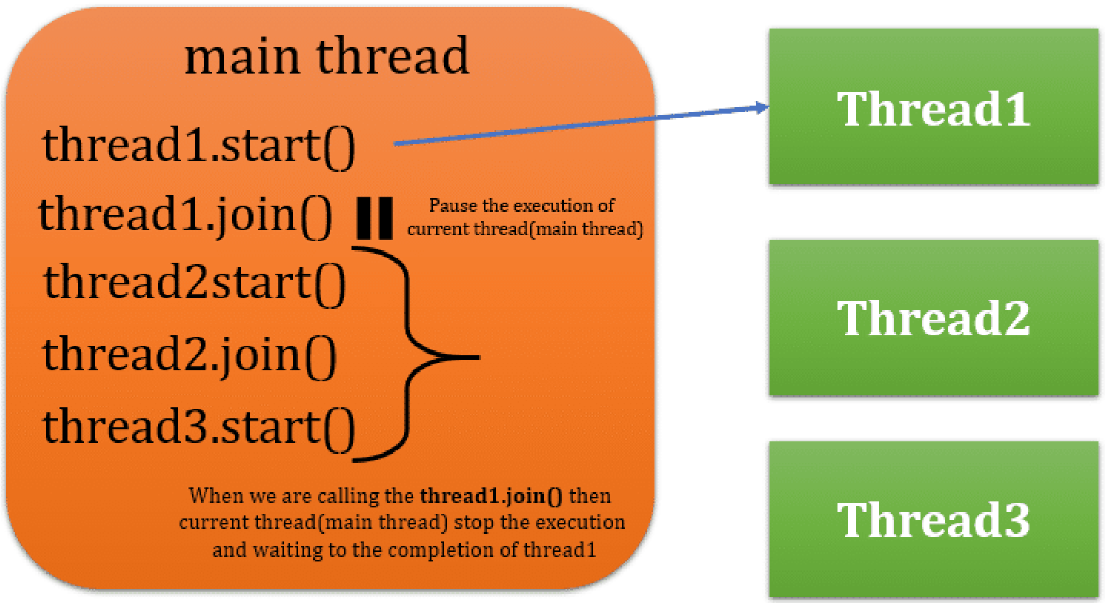
**Note : we cannot use throws with run() method, it will violate the method override rule.**

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


main thread
thread1.start()
thread1.join()
Pause the execution of current thread(main thread)
thread2start()
thread2.join()
thread3.start()
When we are calling the thread1.join() then current thread(main thread) stop the execution and waiting to the completion of thread1
Thread1
Thread2
Thread3

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

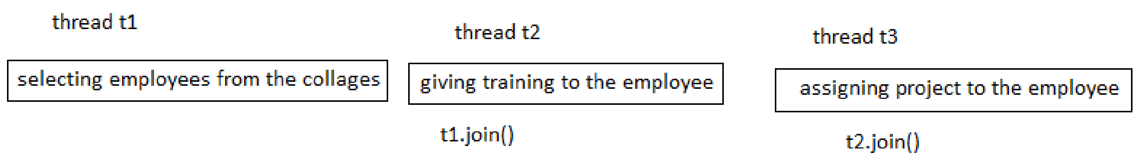
The join() method is a non-static method.

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.


thread t1
selecting employees from the collages
thread t2
giving training to the employee
t1.join()
thread t3
assigning project to the employee
t2.join()

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

Figure - 1

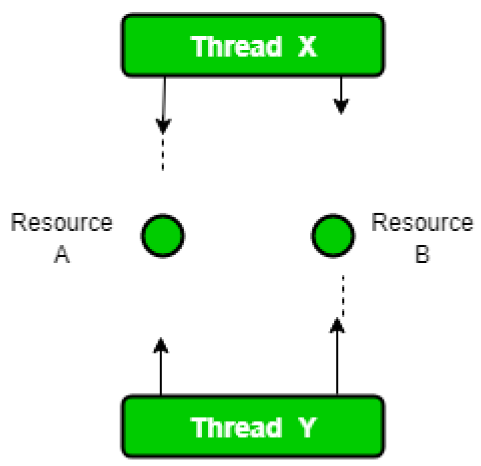
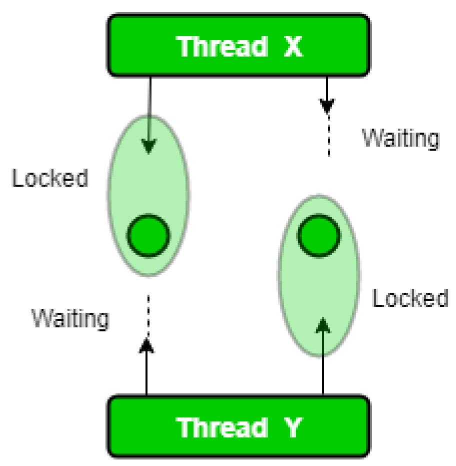


Figure - 2



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 the 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/update/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(){}

synch funB(){}

funC(){}

}
```

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.

**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:

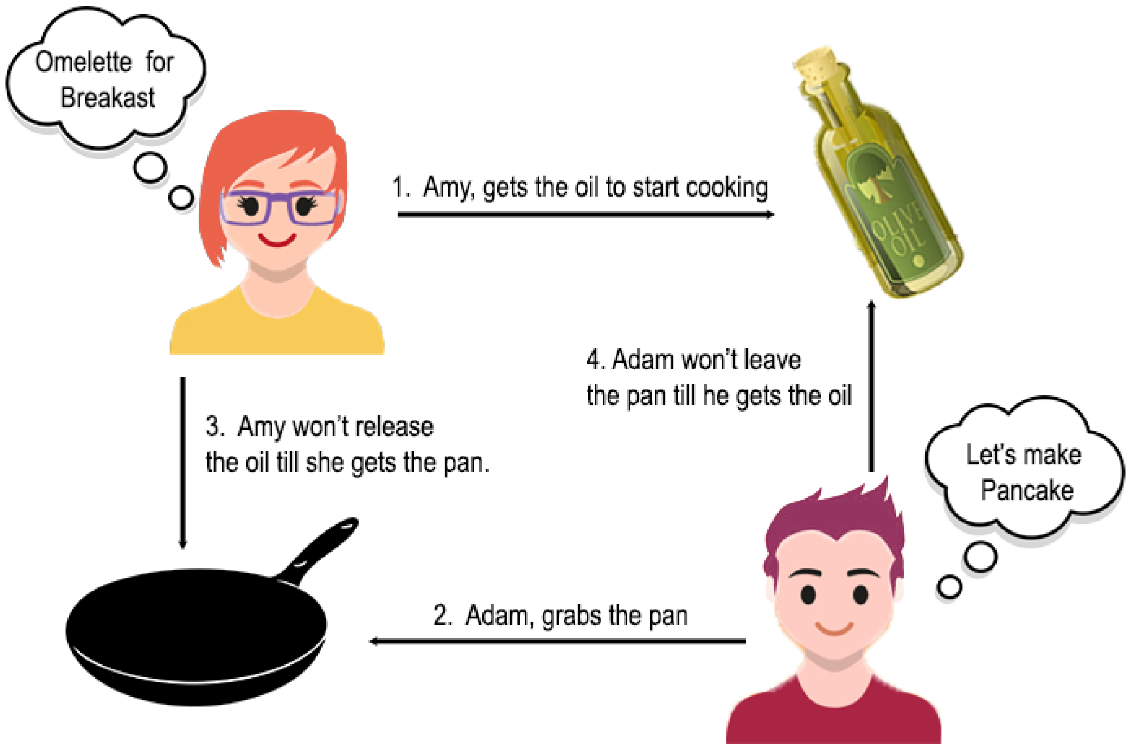
Write the fun1() method of Common class using synchronized block to get the Class level lock.

## Deadlock:

A lock without key is nothing but deadlock.

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

**Example:**


Omelette for Breakast
1. Amy, gets the oil to start cooking
4. Adam won't leave the pan till he gets the oil
3. Amy won't release the oil till she gets the pan.
Let's make Pancake
2. Adam, grabs the pan
OLIVE OIL

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.