Advance Threading

3 more ways to achieve synchronization.

CountDownLatch

This is a more advance type of synchronization that can be done with concurrent package.

Consider this example where an organization needs to recruit 3 Java developers.

For this HR manager has asked 3 Tech Leads to take interview.
The HR manager only wants to distribute the offer letter only after all the 3 Java developers have been recruit.

It means if any thread has done the work then it has to wait for others to finish and all will report the HR together.
In threading terminology the HR manager should will wait till 3 developers have been recruited.

We can't use join() concept here because once we join() the main() thread, The main() won't run until
thread ends completely. if in run() method after recruiting the developers more work is available
then main() method has to wait unnecessary.

```
// concurrent package: very asked in Interview.

import java.util.concurrent.CountDownLatch;

class HRManager{

    public static void main(String...s){

// make object of CountDownLatch object. We can also define number of threads to wait.

        CountDownLatch cdL = new CountDownLatch(4);

        // give the reference id to threads and start all threads
        TechLead tL1 = new TechLead(cdL, "Sachin");
        TechLead tL2 = new TechLead(cdL, "Yuvraj");
        TechLead tL3 = new TechLead(cdL, "MS Dhoni");
        TechLead tL4 = new TechLead(cdL, "Virak Kohli");

        tL1.start();  tL2.start();  tL3.start();  tL4.start();

        try{

        System.out.println("The HR Manager is waiting for recruitment to complete");
            cdL.await();      // It will wait/pause the main thread until all threads done the required job.
```

```
        System.out.println("HR is now available and distributed the offer letter");

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

class TechLead extends Thread{

    CountDownLatch cdL;

    //Constructor
    TechLead(CountDownLatch cdL, String ThreadName){
        super(ThreadName);
        this.cdL = cdL;
    }

    @Override
    public void run(){

        // This is only to see the result slow (Nothing to do)
        try{
            Thread.sleep(2000);
        }catch(InterruptedException e){
            e.printStackTrace();
        }

        System.out.println(Thread.currentThread().getName() + ": recruited");
        cdL.countDown();  // this will --1 decrement that one thread has done job
and so on.

        // This is only to see the result slow (Nothing to do)

        try{
            Thread.sleep(3000);
        }catch(InterruptedException e){
            e.printStackTrace();
        }

        System.out.println(Thread.currentThread().getName() + " is continuing the
job and will be dead");

    }
}
```

Cyclic Barrier:

A 5the way to achieve synchronization.

Cyclic Barrier a synchronization aid that allows a set of threads to all wait for each other to reach a common barrier point. CyclicBarrier are useful in programs involving a fixed size party of threads that must occasionally wait for each other.

The barrier is called cyclic barrier because it can be re-use after the waiting threads are released. CountDownLatch we saw how a master or main thread waits till the worker thread finish their work.

CyclicBarrier class is also a flavor of CountDownLatch with single change that it can re-use.

Let's continue the same example as CountDownLatch:

An organization has to recruit 3 Java developers. And so the HR manager asks 3 Tech leads to interview the candidates. In CountDownLatch example the HR manager wanted to distribute the offer letter to all the 3 candidates, that's the reason we made him to wait. Here the HR manager wants the TechLeads to give the offer letter once they have selected the candidates.

But the TechLeads decide among themselves that they will give the offer letter to their respective candidate only when all interviews are done.

Sare threads ko kisi ek jagah pe roke rehna aur tabhi aaage badhan jab sari us jagah pe pahuch jaye.Ye aise hai jese ke sare friends ek jagah pe pahuch jayenge tabhi sare ek sath jayenge.

```java
import java.util.concurrent.CyclicBarrier;

class HRManager{

    public static void main(String...a){

        // make object of cyclic barrier with thread size and give it to all threads
        CyclicBarrier cb = new CyclicBarrier(4);

        TechLeads tL1 = new TechLeads(cb, "Sachin");
        TechLeads tL2 = new TechLeads(cb, "Yuvraj");
        TechLeads tL3 = new TechLeads(cb, "MS Dhoni");
        TechLeads tL4 = new TechLeads(cb, "V Kohli");

        tL1.start();   tL2.start();   tL3.start();   tL4.start();

        System.out.println("No work of HR here so main is dead now");
    }
}



// TechLeads Thread
class TechLeads extends Thread{

    CyclicBarrier cb;

    // Constructor
    TechLeads(CyclicBarrier cb, String ThreadName){
        super(ThreadName);
        this.cb = cb;
    }

    @Override
    public void run(){

        // To see things let's make it slow down
        try{

            Thread.sleep(2000);
            System.out.println(Thread.currentThread().getName()
            + " recruited developer and waiting for others to complete");
            cb.await();  // this pause/wait for others thread to finish their task

            // Once All done the job, the bellow line will execute.
System.out.println("All finished recruiting. " + Thread.currentThread().getName() +
            " gives offer letter to candidate");


        }catch(Exception e){

            e.printStackTrace();
        }
    }
}
```

Semaphore

Semaphore 6th way to achieve synchronization.
If we want to enter more than one object inside synchronized block then we can do this via Semaphore.

Semaphore class is in concurrent package is used as a pool that can be acquired and release very much like Lock but with a difference. When a thread acquires lock not other thread can enter synchronized block.

Here in Semaphore we can define the pool size and threads can acquire lock till there is resource left in the pool. If we want to enter more than one thread in synchronized/Critical block then we can do this via Semaphore.

Let's see how it work:
The organization needs to recruit 4 Java developers. HR manager asks 4 TechLeads to conduct test and recruit the candidates. The problem here is that TechLeads have only 2 test paper and the photo copy machine is down. That means at runtime only two candidates can give the test and other two have to wait.

Let's simulate this by Semaphore example:

```
import java.util.concurrent.Semaphore;

public class HRManager{

    public static void main(String...a){

/* make an object and give it TechLeads. We can define at a time how many threads can go to
inside critical situation. In above scenario 2 TechLeads can take test. */

        Semaphore sp = new Semaphore(2);

        TechLeads tL1 = new TechLeads(sp, "Raju TL");
        TechLeads tL2 = new TechLeads(sp, "Tahera TL");
        TechLeads tL3 = new TechLeads(sp, "Pawan TL");
        TechLeads tL4 = new TechLeads(sp, "Shayam TL");

        tL1.start();    tL2.start();    tL3.start();    tL4.start();

        System.out.println("HR Manager has no work now and main thread will dead");
    }
}



class TechLeads extends Thread{

    Semaphore sp;

    // constructor
    TechLeads(Semaphore sp, String ThreadName){
        super(ThreadName);
        this.sp = sp;
    }

    @Override
    public void run(){

        try{

    System.out.println(Thread.currentThread().getName() + " is waiting for the test paper");

        // Acquiring a question paper, one more thread can acquire it as we defined.
            sp.acquire();

System.out.println(Thread.currentThread().getName() + "acquired the question paper");

            // sleeping for 2 seconds to know that if 2nd TechLeads get the question paper

            Thread.sleep(2000);

System.out.println(Thread.currentThread().getName() + " Test done giving back the paper" );

            sp.release();   // first thread has done the work and now third thread will get
the test paper and so on.

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

Thread Pooling From JDK5

Thread Pool are useful when we need to limited number of threads in our program at the same time. There is a performance overhead associated with starting a new thread, and each thread is also allocated some memory for its stack.

Servers that processing request can spend more time and consume more system resource in creating an deploying threads that it would processing actual client requests.

Instead of starting a new thread for every tasks to execute concurrently, the task can be passed to a thread pool.

As soon as the pool has any idle threads the task is assigned to one of them and executed.

Internally the tasks are inserted into a Blocking Queue which the threads in the pool are dequeuing from. When a new task is inserted into the queue one of the idle threads will dequeue it successfully and execute it. The rest of the idle threads in the pool will be blocked waiting to dequeue tasks.

It's pool of worker threads with life cycle as follows:

1. Get a new task to execute.

2. Execute it

3. Go back waiting for next task.

ThreadPool are often used in Multi-Threaded servers. Each connection arriving at the server via the network is wrapped as a task and passed on to a thread pool. The threads in the thread pool will process the request on the connections concurrently.

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

class WorkerThread implements Runnable{

    private String command;

    //Constructor
    public WorkerThread(String s){
        this.command = s;
    }

    @Override
    public void run(){
        System.out.println(Thread.currentThread().getName() + " Start command: " + command);

        procesCommand();

        System.out.println(Thread.currentThread().getName() + " End");
    }


    private void procesCommand(){

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

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

    public String toString(){
        return this.command;
    }
}


// Class
public class SimpleThreadPool{

    public static void main(String...args){

        // this line defines how many number of threads i want in thread pool.
        //It will create 5 Thread Object and put them in ThreadPool.

        ExecutorService executor = Executors.newFixedThreadPool(5);

        // now let's give them work to execute.

        for(int i=0; i<10; i++){

            Runnable worker = new WorkerThread("Thread-" + i);
            executor.execute(worker);
        }

/* As you can see in the above loop we have assigned 10 job and threads in the threadPool is
5.So its like when a thread goes idle or finish the job, the next job given to him. Here we
are doing 10 jobs with 5 thread object. Here we are reusing the thread objects instead of
destroying and re-creating.*/

        executor.shutdown();    // When all job done it will shutdown.

        // If we don't want to dead the main thread until all jobs are done then we can set
a while loop and check if

        //all jobs are done or not.

        while(!executor.isTerminated()){    }

        System.out.println("Finished all threads");
    }
}
```

Callable Interface

Sometimes we wish that a thread return some value that we can use. Java 5 introduced "java.util.concurrent.Callable" interface
in concurrency package that is similar to "Runnable" interface but it can return any object and able to throw exception.

Callable interface use Generic to define the return type of object. Executors class provide useful methods to execute callable
in a thread pool. Since callable tasks run in a parallel; we have to wait for the returned object. Callable tasks return
java.util.concurrent.Future object. Using Future we can find out the status of the callable task and get the returned object.
It provides get() method that can wait for the callable to finish and then return the result.

Future provides cancel() method to cancel the associated callable task.
There is a overloaded version of get() method where we can specify the time to wait for the result, its useful to avoid
current thread getting blocked for longer time. There are isDown() and isCancelled() methods to find out the current status
of associated callback task.

Here is a simple example of callable task that returns the name of thread executing the task after one second. We are using
"Executor" framework to execute 10 tasks in parallel and use "Future" to get the result of the submitted tasks.

Once we execute the program, we will notice the delay in output because Future.get() method wait for the callable task to
complete.

When we submit a callable object to an Executor the framework returns an object of type java.util.concurrent.Future.
This Future object is used to check the results of a callable.

use the get() method to retrieve result of the future.

call() method is same as Runnable's run() but callable only used via ThreadPool.
Thread class does not have constructor that accepts callable object.

run() method return type is void but call() has generic return type.
We can't add try-catch in run() method but on call() method we can add try-catch block.

```
import java.util.concurrent.*;
import java.util.*;

class MyCallable implements Callable{

    @Override
    public String call() throws Exception{

        Thread.sleep(2000);

        //return name of the thread
        return ("Returned Value: " + Thread.currentThread().getName());
    }
}
```

```
public class MyCallableTest{

    public static void main(String...args){

        // Making object of 5 threads and sending them in pool.
        ExecutorService executor = Executors.newFixedThreadPool(5);

    // collection to store the result. An ArrayList that accepts Future<String> type data.
        ArrayList<Future<String>> list = new ArrayList<Future<String>> ();

        //now let's get and store returned data and add in the collection.

        for(int i=0; i<10; i++) {

            // Executor framework has method submit() that accepts our thread-callable
object and call the call().

            Future<String> future = executor.submit( new MyCallable() );
            list.add(future);
        }

        // Now let's print the result.

        for(Future<String> f: list){
            try{
                System.out.println(f.get());
            }catch(Exception e){    e.printStackTrace();    }
        }
    }
}
```

Blocking Queue

Java 5 comes with blocking queue implementation in the **java.util.concurrent.BlockingQueue** package.

BlockingQueue interface: It is used to share data from one thread to another thread. **ArrayBlockingQueue** implements BlockingQueue.

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

Why is it called BlockingQueue?
Suppose you are writing somewhere that place is full. Or You want read something from somewhere and that is empty. This is a type of case when exception can come so BlockingQueue sees if there is nothing to read then the thread that works to read will blocked. Same as if we want to write something somewhere and there is no space then it will block the thread that works to write until some space available. Just simply consider **Enqueue = Write and Dequeue = Read**.

A Blocking queue is a queue that blocks when you try to dequeue from it and the queue is empty, or if you are 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 try to enqueue an item in a full queue is blocked until some thread makes space in the queue, either by dequeuing one or more items or cleaning the queue completely.

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

The example uses the ArrayBlockingQueue implementation of the BlockingQueue interface. The BlockingQueueEx class starts a producer and a Consumer is seperate thread. The producer thread inserts string into a shared BlockingQueue while the Consumer takes them out.

```
import java.util.concurrent.*;
import java.util.*;


public class BlockingQueueEx{

    public static void main(String...args){
        try{
            ArrayBlockingQueue queue = new ArrayBlockingQueue(5);

            //Producer class that will enqueue the data
            Producer producer = new Producer(queue);

            // Consumer class that will dequeue the data
            Consumer consumer = new Consumer(queue);

            System.out.println("Starting Producer");
            new Thread(producer).start();

            System.out.println("Starting Consumer");
            new Thread(consumer).start();

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



// Producer class, It sleeps 3 seconds between each put() call.
// This will cause the consumer to block, while waiting for object in the queue.

class Producer implements Runnable{

    protected BlockingQueue queue = null;

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

    @Override
    public void run(){
        try{
            queue.put("String One");
            Thread.sleep(3000);

            queue.put("String Two");
            Thread.sleep(3000);

            queue.put("String Three");
        }catch(InterruptedException e){   e.printStackTrace();   }
    }
}
```

```
// Consumer class: It takes out the object from the queue and prints them to
System.out.

class Consumer implements Runnable{

    protected BlockingQueue queue = null;

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

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

ConcuuretLinkedQueue:

ConcurrentLinkedQueue is an unbounded thread-safe queue based on linked nodes. This queue orders elements a FIFO (First-In-First-Out).

The head of the queue is that elements that has been on the queue the longest time. The tail of the queue is that element that has been on the queue the shortest time. New elements are inserted at the tail of the queue, and the queue retrieval operations obtain elements
at the head of the queue.

A ConcurrentLinkedQueue is a good choice when many threads share access to a common collection. Like most other concurrent collection implementation, this class does not permit the use of null elements.

This Java example spawns off two threads. One puts String on the queue and the other takes strings off the queue.

```
// The producer puts strings on the queue
class Producer implements Runnable{

    ConcurrentLinkedQueue<String> queue;

    //Constructor
    Producer(ConcurrentLinkedQueue<String> queue){
        this.queue = queue;
    }

    @Override
    public void run(){
        System.out.println("Producer Started");
        try{
            for(int i=1; i<10; i++){
                queue.add("String: " + i);
                System.out.println("Added String: " + i);
                Thread.currentThread().sleep(1000);
```

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

//The Consumer removes String from the queue.
class Consumer implements Runnable{

    ConcurrentLinkedQueue<String> queue;

    //Constructor
    Consumer(ConcurrentLinkedQueue<String> queue){
        this.queue = queue;
    }

    @Override
    public void run(){
        String str;
        System.out.println("Consumer Started");

        for(int x=0; x<10; x++){

            while( (str = queue.poll()) !=null ){
                System.out.println("Removed: " + str);
            }

            try{
                Thread.currentThread().sleep(2000);
            }catch(Exception e){    e.printStackTrace();    }
        }
    }
}
```

Exchanger: can swap values between two threads

Exchanger is synchronization point at which threads can pair and swap elements within pairs.

Each thread presents some object, and receive its partners object on return.
An Exchanger may be viewed as a bi-directional form of a synchronousQueue.

This Java program creates 2 threads passing them an exchanger and a message.
Each threads print its own message, then exchange with it the other thread, then print out its new exchange message.

```
import java.util.concurrent.*;

public class ExchangerExample{

    public static void main(String...a){

        Exchanger<String> exchanger = new Exchanger<String> ();

        Thread t1 = new MyThread(exchanger, "I Like Coffee");
        Thread t2 = new MyThread(exchanger, "I Like Tea");
        t1.start();
        t2.start();
    }
}
```

```
// User Defined Thread Class
class MyThread extends Thread{

    Exchanger<String> exchanger;
    String message;

    //Constructor
    MyThread(Exchanger<String> exchanger, String message){
        this.exchanger = exchanger;
        this.message = message;
    }

    public void run(){
        try{
            System.out.println(this.getName() + " Message: " + message);

            //exchange the message
            message = exchanger.exchange(message);

            System.out.println(this.getName() + " Message: " + message);

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

ThreadLocal: It is a Generic Class

ThreadLocal class in Java enable you to create variables that can be read and write by some thread. Thus even if two threads are executing the same code, and the code has a reference to ThreadLocal variables, then the two threads can not see each others ThreadLocal variable.

Her is a code example that shows how to create a ThreadLocal variable.

```
    private ThreadLocal MyThreadLocal = new ThreadLocal();
```

As you can see, You instantiated a new ThreadLocal object. This only needs to be done once, and it does not matter which thread does that. All threads will be see the same ThreadLocal instance but the values set on the ThreadLocal via its set() method will only be visible to thread who set the value.

Even if two different thread sets different value on the same ThreadLocal object, they cannot see each others value.

```
public class ThreadLocalExample{

    static int n = 0;

    // Inner class
    private static class MyRunnable implements Runnable{

        // Creating ThreadLocal Variable
        private ThreadLocal<Integer> threadLocal = new ThreadLocal<Integer> ();

        @Override
        public void run(){

            int z = (int) (Math.random() * 100D);
            System.out.println(Thread.currentThread().getName()+ ": " + z);

            // Setting the value for the currentThread
            threadLocal.set(z);

            // Sleep, so other come and set their own value.
            try{ Thread.sleep(1000); }
            catch(Exception e){   e.printStackTrace(); }


            //Now check if value is altered.
            System.out.println("Later.............\n");
            System.out.println(Thread.currentThread().getName() + " " +
threadLocal.get());
        }
    }

    // main of outer class
    public static void main(String...args){

        MyRunnable shared = new MyRunnable();

        new Thread(shared).start();  // Thread-0
        new Thread(shared).start();  // Thread-1
        new Thread(shared).start();  // Thread-2
        new Thread(shared).start();  // Thread-3

    }
}
```

Thread Local Variable

Now requirement came to share the local variable's value between threads. Each

Thread store its local variable's value inside cache. To share the local variable, we have to make our local variable as "**volatile**" access modifier. However it most of the time does not work.

When multiple threads using the same variable, each thread will have its own copy of the local cache
for that variable. So, when its updating the value, it is actually updating in the local cache not in the main variable memory. The other thread which is using the same variable does not know anything about the values changed by the another thread. To avoid this problem.

If we declare a variable as volatile then it will not be store in the local cache. Whenever thread are updating the value, it is updated to the main memory. So, other threads can access the updated value.

If we are working with the multi-threaded programming, the volatile keyword will be more useful. The Java volatile keyword is used to mark a Java variable as being stored in a main memory. More precisely that means that every read of a volatile variable will be read from the computer's main memory, and not from the CPU cache, and that every write to a volatile will be written to main memory and not just to the CPU cache.

```
public class JavaAtomic{

    public static void main(String...a){
        // User Defined Thread
        try{
            ProcessingThread pt = new ProcessingThread();
            Thread t1 = new Thread(pt, "ThreadOne");
            t1.start();

            Thread t2 = new Thread(pt, "ThreadTwo");
            t2.start();

            // Joining the threads so main will wait
            t1.join();
            t2.join();

            System.out.println("Count: " + pt.getCount());
        }catch(Exception e){  e.printStackTrace();  }
    }
}


// User Defined Thread Class.

class ProcessingThread implements Runnable{

    private volatile int count;

    @Override
    public void run(){
        for(int i=1; i<5; i++){

// proves that loop is runs 8 times. 4 times for each thread. But it won't show counting 8 everytime.
            System.out.println("Hello");
            try{
                Thread.sleep(1000);
            }catch(Exception e){  e.printStackTrace();  }

            // counting
            count++;
        }
    }

    public int getCount(){
        return this.count;
```

```
    }
}
```

To share the local variable, we have to make our local variable as "**volatile**" access modifier. However it most of the time does not work.

If I run the above program, I will notice that count varies between 5,6,7,8. The reason is because count++ is not an atomic operation. So by the time one thread read its value and increment it by one, other threads has read the older value leading to wrong result.

To solve this issue, we will have to make sure that increment operation on count is atomic, we can do that using synchronization but Java5 java.util.concurrent.atomic provides Wrapper classes for int and long that can be used to achieve this atomically without usage of synchronization.

Here is the updated program that will always output count value as 8 because AtomicInteger incrementAndGet() increments the current value one by one.

```
import java.util.concurrent.atomic.AtomicInteger;

public class JavaAtomic{

    public static void main(String...args){
        try{

            //user defined thread class.
            ProcessingThread p = new ProcessingThread();
            Thread t1 = new Thread(p, "Thread One");
            t1.start();

            Thread t2 = new Thread(p, "Thread Two");
            t2.start();

            // Wait the main
            t1.join();
            t2.join();

            System.out.println("Count: " + p.getCount());

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


// User defined thread class.
class ProcessingThread implements Runnable{

    private AtomicInteger count = new AtomicInteger();

    @Override
    public void run(){
        for(int i=1; i<5; i++){
            try{
                Thread.sleep(1000);
                count.incrementAndGet();
            }catch(Exception e){    e.printStackTrace(); }
        }
    }
```

```
    public int getCount(){
        return this.count.get();
    }
}
```