**CountDownLatch and CyclicBarrier – 2021-2022**

**CountDownLatch**

A synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads completes. A CountDownLatch is initialized with a given count. **The await methods block until the current count reaches zero due to invocations of the countDown() method, after which all waiting threads are released and any subsequent invocations of await return immediately**. **This is a one-shot phenomenon -- the count cannot be reset.** **If you need a version that resets the count, consider using a CyclicBarrier.**

**A CountDownLatch initialized with a count of one serves as a simple on/off latch, or gate**: all threads invoking await wait at the gate until it is opened by a thread invoking countDown(). A CountDownLatch initialized to N can be used to make one thread wait until N threads have completed some action, or some action has been completed N times.

A useful property of a CountDownLatch is that it doesn't require that threads calling countDown wait for the count to reach zero before proceeding, it simply prevents any thread from proceeding past an await until all threads could pass.

http://www.javamex.com/tutorials/threads/CountDownLatch.shtml

we can make several threads start at the same time;

we can wait for several threads to finish (whereas, for example, the Thread.join() method only lets you wait for a single thread).

**Introduction to CountDownLatch**

In concurrent programming, a latch is a type of "switch" or "trigger". The latch is set up with a particular count value. The count is then counted down, and at strategic moments, a thread or threads waits for the countdown to reach zero before continuing to perform some process. Note that this is a one-off process: once the latch reaches zero, there is no way to reset the count.

the CountDownLatch object is constructed with the initial count;

calling countDown() decrements the count by 1;

the await() method will wait for the count to reach zero, or proceed immediately if the count already reached zero.

The CountDownLatch class is designed to be safe to call from multiple threads without any extra synchronization. (This differs, for example, from the wait/notify mechanism, where threads must be synchronized on the given lock object before calling wait() or notify().)

**Why use CountDownLatch (rather than wait/notify, Condition etc)?**

The CountDownLatch protects you against the case of a thread missing a signal which can occur if you use these other mechanisms for coordinating jobs. Something like a Condition is useful for signaling to threads if they are waiting, but where it doesn't matter if they're not, or where a thread will explicitly check if it has to wait before waiting. With a CountDownLatch, we await a signal if it hasn't been triggered yet, but immediately continue without waiting if that signal was already triggered before we start waiting.

**How to make several threads start at the same time**

Sometimes it is useful to make a group of threads start at approximately the same time. For example, consider performance tests such as the ones conducted for this web site. If we're testing some throughput with n threads, it's only fair if the n threads start (and stop) at more or less the same time. In another situation, we might want a group of threads to start as soon as some asynchronous initialisation procedure is complete.

To coordinate the starting of several threads, we first create a CountDownLatch with an initial count of 1. Then, each thread will sit at the start of its run() method, waiting for the latch to be counted down (i.e. sitting in the await() method). The thread performing the initialisation step (or just the thread coordinating the start of the other threads in the case of our performance experiment) then calls countDown() on the latch. Because the initial count was 1, this single countdown operation triggers all the other threads to start at (approximately) the same time.

If we define a subclass of Thread to handle the concurrent tasks, then we can arrange to pass the CountDownLatch into the constructor of those threads:

public class LatchedThread extends Thread {

private final CountDownLatch startLatch;

public LatchedThread(CountDownLatch startLatch) {

this.startLatch = startLatch;

}

public void run() {

try {

**startLatch.await();**

// ... perform task

} catch (InterruptedException iex) {}

}

}

Then, to coordinate the starting of 4 of these threads:

**CountDownLatch startLatch = new CountDownLatch(10);**

for (int threadNo = 0; threadNo < 4; threadNo++) {

Thread t = new LatchedThread(startLatch);

t.start();

}

// give the threads chance to start up; we could perform

// initialisation code here as well.

Thread.sleep(200);

**startLatch.countDown();**

When we call countDown() in the main thread, we don't actually know that all of the threads have started up; we just assume that sleeping for a fraction of a second gives them a "reasonable chance" of being ready for the signal. If any of the threads "misses the signal", it won't actually matter too much: when such a thread does start up, enter its run() method await() method, it will no longer actually wait, since the latch has already reached zero.

Bear in mind that inevitably, threads will "wake up" with approximate simultaneity: how simultaneous it can actually be depends on various factors, such as whether each thread can actually be allocated to a free processor, how "busy" the system is (what other threads are running and at what priorities), what threads are doing— i.e. how quickly running threads will relinquish the CPU— and what your particular operating system's policy is on prioritising waiting threads when they are signalled to wake up. (See the section on thread scheduling for more details about these factors.)

How to wait for several threads to complete

Another common scenario is with parallel processing, where we need to wait for several threads to finish or reach a particular point. In this case, we can use a similar mechanism:

This is therefore more flexible than the join() method, which only lets us wait for a single thread. Here is an example of waiting for 10 threads to complete:

public class StopLatchedThread extends Thread {

private final CountDownLatch stopLatch;

public StopLatchedThread(CountDownLatch stopLatch) {

this.stopLatch = stopLatch;

}

public void run() {

try {

// perform interesting task

} finally {

stopLatch.countDown();

}

}

}

public void performParallelTask() throws InterruptedException {

CountDownLatch cdl = new CountDownLatch(10);

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

Thread t = new StopLatchedThread(cdl);

t.start();

}

cdl.await();

}

**Interruptions and timeouts**

A thread sitting in the await() method can be interrupted (generally by another thread calling interrupt() on it). Therefore, the await() method throws InterruptedException. Inside a run() method, the most appropriate action is usually to catch the exception around the whole logic of the method, so that interrupting the thread makes it exit. Where we are waiting for threads to complete inside a method, we can just make that method throw the exception up, and let the caller worry about what happens if the process is interrupted. For more information, see the section on thread interruption.

A version of the await() method takes a timeout (and TimeUnit in which the timeout is specified). Setting a timeout could be useful if, for example, the condition that a thread is awaiting is the initialisation of a driver, and there's a chance that the driver will not get initialised in a reasonably amount of time. In the timed case, the method returns true if the latch was actually triggered, and false if a timeout occurred. The timed method can still be interrupted and throw InterruptedException. Coordinating multi-stage/iterated parallel processes

The CountDownLatch is useful for coordination of one-off operations. In the next section, we look at the CyclicBarrier class, which allows repeated or multi-stage parallel processes to be coordinated.

An example of CountDownLatch is given below.

import java.util.concurrent.\*;

public class LatchTest {

private static final int COUNT = 10;

private static class Worker implements Runnable {

**CountDownLatch startLatch;**

**CountDownLatch stopLatch;**

String name;

Worker(CountDownLatch startLatch, CountDownLatch stopLatch, String name) {

this.startLatch = startLatch;

this.stopLatch = stopLatch;

this.name = name;

}

public void run() {

try {

**startLatch.await();** // wait until the latch has counted down to zero

} catch (InterruptedException ex) {

ex.printStackTrace();

}

System.out.println("Running: " + name);

**stopLatch.countDown();**

}

}

public static void main(String args[]) {

// CountDownLatch(int count)

// Constructs a CountDownLatch initialized with the given count.

CountDownLatch startSignal = new CountDownLatch(1);

CountDownLatch stopSignal = new CountDownLatch(COUNT);

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

new Thread(new Worker(startSignal, stopSignal, Integer.toString(i)))

.start();

}

System.out.println("Go");

startSignal.countDown();

try {

stopSignal.await();

} catch (InterruptedException ex) {

ex.printStackTrace();

}

System.out.println("Done");

}

}

CountdownLatch is used to start a series of threads and then wait until all of them are complete (or until they call countDown() a given number of times.

**Semaphore** **is used to control the number of concurrent threads that are using a resource**. That resource can be something like a file, or could be the cpu by limiting the number of threads executing. The count on a Semaphore can go up and down as different threads call acquire() and release().

**CountdownLatch makes threads wait on the await() method, until such a time as the count has reached zero.** So maybe you want all your threads to wait until 3 invocations of something, then all the threads can go. A Latch generally can not be reset.

A Semaphore allows threads to retrieve permits, which prevents too many threads from executing at once, blocking if it cannot get the permit(s) it requires to proceed. Permits can be returned to a Semaphore allowing the other waiting threads to proceed.

http://javarevisited.blogspot.in/2012/07/countdownlatch-example-in-java.html

**What is CountDownLatch in Java**

CountDownLatch in Java is a kind of synchronizer which allows one Thread to wait for one or more Threads before starts processing. This is very crucial requirement and often needed in server side core Java application and having this functionality built-in as CountDownLatch greatly simplifies the development. CountDownLatch in Java is introduced on Java 5 along with other concurrent utilities like CyclicBarrier, Semaphore, ConcurrentHashMap and BlockingQueue in java.util.concurrent package. You can also implement same functionality using wait and notify mechanism in Java but it requires lot of code and getting it write in first attempt is tricky, With CountDownLatch it can be done in just few lines. CountDownLatch also allows flexibility on number of thread for which main thread should wait, It can wait for one thread or n number of thread, there is not much change on code.

**How CountDownLatch works in Java**

Now we know What is CountDownLatch in Java, its time to find out How CountDownLatch works in Java. CountDownLatch works in latch principle, main thread will wait until Gate is open. One thread waits for n number of threads specified while creating CountDownLatch in Java. Any thread, usually main thread of application, which calls CountDownLatch.await() will wait until count reaches zero or its interrupted by another Thread. All other thread are required to do count down by calling CountDownLatch.countDown() once they are completed or ready to the job. as soon as count reaches zero, Thread awaiting starts running. One of the disadvantage of CountDownLatch is that its not reusable once count reaches to zero you can not use CountDownLatch any more, but don't worry Java concurrency API has another concurrent utility called CyclicBarrier for such requirements.

CountDownLatch Exmaple in Java

In this section we will see a full featured real world example of using CountDownLatch in Java. In following CountDownLatch example, Java program requires 3 services namely CacheService, AlertService and ValidationService to be started and ready before application can handle any request and this is achieved by using CountDownLatch in Java.

import java.util.Date;

import java.util.concurrent.CountDownLatch;

import java.util.logging.Level;

import java.util.logging.Logger;

/\*\*

\* @author Javin Paul

\*/

public class CountDownLatchDemo {

public static void main(String args[]) {

final CountDownLatch latch = new CountDownLatch(3);

Thread cacheService = new Thread(new Service("CacheService", 1000, latch));

Thread alertService = new Thread(new Service("AlertService", 1000, latch));

Thread validationService = new Thread(new Service("ValidationService", 1000, latch));

cacheService.start(); //separate thread will initialize CacheService

alertService.start(); //another thread for AlertService initialization

validationService.start();

// application should not start processing any thread until all service is up

// and ready to do there job.

// Countdown latch is idle choice here, main thread will start with count 3

// and wait until count reaches zero. each thread once up and read will do

// a count down. this will ensure that main thread is not started processing

// until all services is up.

//count is 3 since we have 3 Threads (Services)

try{

**latch.await();** //main thread is waiting on CountDownLatch to finish

System.out.println("All services are up, Application is starting now");

}catch(InterruptedException ie){

ie.printStackTrace();

}

}

}

/\*\*

\* Service class which will be executed by Thread using CountDownLatch synchronizer.

\*/

class Service implements Runnable{

private final String name;

private final int timeToStart;

private final CountDownLatch latch;

public Service(String name, int timeToStart, CountDownLatch latch){

this.name = name;

this.timeToStart = timeToStart;

this.latch = latch;

}

@Override

public void run() {

try {

Thread.sleep(timeToStart);

} catch (InterruptedException ex) {

Logger.getLogger(Service.class.getName()).log(Level.SEVERE, null, ex);

}

System.out.println( name + " is Up");

**latch.countDown();** //reduce count of CountDownLatch by 1

}

}

Output:

ValidationService is Up

AlertService is Up

CacheService is Up

All services are up, Application is starting now

By looking at output of this CountDownLatch example in Java, you can see that Application is not started until all services started by individual Threads are completed.

When should we use CountDownLatch in Java :

Use CountDownLatch when one of Thread like main thread, require to wait for one or more thread to complete, before its start doing processing. Classical example of using CountDownLatch in Java is any server side core Java application which uses services architecture, where multiple services is provided by multiple threads and application can not start processing until all services have started successfully as shown in our CountDownLatch example.

Few points about Java CountDownLatch which is worth remembering:

1) You can not reuse CountDownLatch once count reaches to zero, this is the main difference between CountDownLatch and CyclicBarrier, which is frequently asked in core Java interviews and multi-threading interviews.

2) Main Thread wait on Latch by calling CountDownLatch.await() method while other thread calls CountDownLatch.countDown() to inform that they have completed.

That’s all on What is CountDownLatch in Java, What does CountDownLatch do in Java, How CountDownLatch works in Java along with a real life CountDownLatch example in Java. This is a very useful concurrency utility and if you master when to use CountDownLatch and how to use CountDownLatch you will be able to reduce good amount of complex concurrency control code written using wait and notify in Java.

**Difference between CountDownLatch and CyclicBarrier in Java**

**Both CyclicBarrier and CountDownLatch are used to implement a scenario where one Thread waits for one or more Thread to complete there job before starts processing** but there is one Difference between CountDownLatch and CyclicBarrier in Java which separates them apart and that is, you can not reuse same CountDownLatch instance once count reaches to zero and latch is open, on the other hand CyclicBarrier can be reused by resetting Barrier, Once barrier is broken.

**A useful property of a CountDownLatch is that it doesn't require that threads calling countDown wait for the count to reach zero before proceeding, it simply prevents any thread from proceeding past an await until all threads could pass**.

A CyclicBarrier supports an optional Runnable command that is run once per barrier point, after the last thread in the party arrives, but before any threads are released. This barrier action is useful for updating shared-state before any of the parties continue.

The CyclicBarrier uses a fast-fail all-or-none breakage model for failed synchronization attempts: If a thread leaves a barrier point prematurely because of interruption, failure, or timeout, all other threads, even those that have not yet resumed from a previous await(), will also leave abnormally via BrokenBarrierException (or InterruptedException if they too were interrupted at about the same time).

One self developed example is given below.

import java.util.concurrent.CountDownLatch;

import java.util.concurrent.TimeUnit;

public class Validation2 extends Thread {

private CountDownLatch latch;

public Validation2( CountDownLatch latch ) {

this.latch = latch;

}

@Override

public void run() {

try {

System.out.println("Running Second Validation ...");

TimeUnit.SECONDS.sleep(5);

latch.countDown();

}

catch (Exception e) {

e.printStackTrace();

}

}

}

import java.util.concurrent.CountDownLatch;

import java.util.concurrent.CountDownLatch;

import java.util.concurrent.TimeUnit;

public class Validation1 extends Thread {

private CountDownLatch latch;

public Validation1( CountDownLatch latch ) {

this.latch = latch;

}

@Override

public void run() {

try {

System.out.println("Running First Validation ...");

TimeUnit.SECONDS.sleep(5);

latch.countDown();

}

catch (Exception e) {

e.printStackTrace();

}

}

}

import java.util.concurrent.TimeUnit;

public class Validation3 extends Thread {

**private CountDownLatch latch;**

public Validation3( CountDownLatch latch ) {

this.latch = latch;

}

@Override

public void run() {

try {

System.out.println("Running Third Validation ...");

TimeUnit.SECONDS.sleep(5);

latch.countDown();

}

catch (Exception e) {

e.printStackTrace();

}

}

}

import java.util.concurrent.CountDownLatch;

public class TestAllValidations {

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

**CountDownLatch startLatch = new CountDownLatch(3);**

Thread th1 = new Validation1(startLatch);

Thread th2 = new Validation2(startLatch);

Thread th3 = new Validation3(startLatch);

th1.start();

th2.start();

th3.start();

**startLatch.await(); //main thread is waiting on CountDownLatch to finish**

System.out.println("All validations completed ...");

System.out.println("Going to start a new Operation ...");

}

}

As from the above example, we have 3 validation services, we have started and we are waiting for them to complete.

After completion, we will start a new one.

Problem: **A teacher gives a problem to 5 students to compute. Once all the students have completed the computations, finally teacher has to disclose the result of all students.** Code is given below.

**import** java.util.Map;  
**import** java.util.Random;  
**import** java.util.concurrent.CountDownLatch;  
**import** java.util.concurrent.TimeUnit;  
**public class StudentThread** **implements** Runnable {  
 **private** Map<String, Integer> **dataMap**;  
 **private** String **name**;  
 **private** CountDownLatch **countDownLatch**;  
 **private int timeInSecs**;  
  
 **public** StudentThread(  
 Map<String, Integer> dataMap, String name, CountDownLatch countDownLatch, **int** timeInSecs) {  
 **this**.**dataMap** = dataMap;  
 **this**.**name** = name;  
 **this**.**countDownLatch** = countDownLatch;  
 **this**.**timeInSecs** = timeInSecs;  
 }  
  
 @Override  
 **public void** run() {  
 **try** {  
 System.***out***.println(Thread.*currentThread*().getName() + **" computing value ..."**);  
 TimeUnit.***SECONDS***.sleep(**timeInSecs**);  
 Random rand = **new** Random();  
 **int** value = rand.nextInt(100);  
 **dataMap**.put(**name**, value);  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
 System.***out***.println(Thread.*currentThread*().getName() + **" completed the computation"**);  
 **countDownLatch.countDown();**  
 }  
}

**import** java.util.Map;  
**import** java.util.concurrent.ConcurrentHashMap;  
**import** java.util.concurrent.CountDownLatch;  
  
**public class TestCountDownLatch** {  
 **public static void** main(String[] args) {  
 CountDownLatch countDownLatch = **new** CountDownLatch(5);  
 Map<String, Integer> dataMap = **new** ConcurrentHashMap<String, Integer>();  
 Thread t1 = **new** Thread(**new** StudentThread(dataMap, **"John"**, countDownLatch, 3), **"John"**);  
 Thread t2 = **new** Thread(**new** StudentThread(dataMap, **"Vidya"**, countDownLatch, 5), **"Vidya"**);  
 Thread t3 = **new** Thread(**new** StudentThread(dataMap, **"Amit"**, countDownLatch, 7), **"Amit"**);  
 Thread t4 = **new** Thread(**new** StudentThread(dataMap, **"Ronaldo"**, countDownLatch, 3), **"Ronaldo"**);  
 Thread t5 = **new** Thread(**new** StudentThread(dataMap, **"Pihu"**, countDownLatch, 9), **"Pihu"**);  
  
 t1.start();  
 t2.start();  
 t3.start();  
 t4.start();  
 t5.start();  
 **try** {  
 **countDownLatch.await();** 🡸**It is blocking**  
 System.***out***.println(**"All students completed the computations, now show the result ..."**);  
 dataMap.forEach((key, value) -> System.***out***.println(key + **"<--->"** + value));  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
}

**OUTPUT**

John computing value ...

Vidya computing value ...

Amit computing value ...

Ronaldo computing value ...

Pihu computing value ...

John completed the computation

Ronaldo completed the computation

Vidya completed the computation

Amit completed the computation

Pihu completed the computation

All students completed the computations, now show the result ...

Ronaldo<--->97

Pihu<--->84

Vidya<--->84

John<--->57

Amit<--->25

**A precise way to writing CountDownLatch using Java 8 Lambda**

**public class** TestLatch {  
  
 **public static void** task(CountDownLatch latch, String name, **int** time) {  
 **try** {  
 Thread.*currentThread*().setName(name);  
 System.***out***.println(**"Executing Task ...."**+Thread.*currentThread*().getName());  
 TimeUnit.***SECONDS***.sleep(time);  
 System.***out***.println(**"Task completed ..."**);  
 } **catch**(InterruptedException ie) {  
 ie.printStackTrace();  
 } **finally** {  
 latch.countDown();  
 }  
 }  
  
 **public static void** main(String[] args) **throws** InterruptedException {  
 CountDownLatch latch = **new** CountDownLatch(3);  
  
 Runnable r1 = () -> *task*(latch, **"Aadhar"**,7);  
 Runnable r2 = () -> *task*(latch, **"Pan"**, 5);  
 Runnable r3 = () -> *task*(latch, **"Passport"**, 3);Thread t1 = **new** Thread(r1);  
 Thread t2 = **new** Thread(r1);  
 Thread t3 = **new** Thread(r1);  
  
 t1.start();  
 t2.start();  
 t3.start();  
  
 latch.await();  
 System.***out***.println(**"All validations completed "**);  
 }  
}

**CyclicBarrier**

A synchronization aid that allows a set of threads to all wait for each other to reach a common barrier point. CyclicBarriers are useful in programs involving a fixed sized party of threads that must occasionally wait for each other. The barrier is called cyclic because it can be re-used after the waiting threads are released.

A CyclicBarrier supports an optional Runnable command that is run once per barrier point, after the last thread in the party arrives, but before any threads are released. This barrier action is useful for updating shared-state before any of the parties continue.

**http://tutorials.jenkov.com/java-util-concurrent/cyclicbarrier.html**

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.

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!

A good example is given below.

import java.util.Date;

import java.util.concurrent.BrokenBarrierException;

import java.util.concurrent.CyclicBarrier;

public class CyclicBarrier2Test {

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

CyclicBarrier barrier=new CyclicBarrier(2);

new Thread(new Worker(barrier)).start();

barrier.await();

System.out.println("Done Cycle 1\n");

new Thread(new Worker(barrier)).start();

barrier.await();

System.out.println("Done Cycle 2");

}

public static class Worker implements Runnable {

private CyclicBarrier barrier;

public Worker(CyclicBarrier barrier) {

this.barrier=barrier;

}

@Override

public void run() {

try {

System.out.println("Thread "+Thread.currentThread().getId()+" at: "+new Date());

barrier.await();

} catch (Exception e) {

System.out.println(e.getMessage());

}

}

}

}

Output

Thread 8 at: Mon Oct 03 11:04:59 CDT 2011

Done Cycle 1

Thread 9 at: Mon Oct 03 11:04:59 CDT 2011

Done Cycle 2

From the above program it is evident that you can reuse cyclicbarrier.

http://javarevisited.blogspot.in/2012/07/cyclicbarrier-example-java-5-concurrency-tutorial.html

**What is CyclicBarrier in Java**

CyclicBarrier is similar to CountDownLatch which we have seen in last article What is CountDownLatch in Java and allows multiple threads to wait for each other (barrier) before proceeding. CyclicBarrier is a natural requirement for concurrent program because it can be used to perform final part of task once individual tasks are completed. All threads which wait for each other to reach barrier are called parties, CyclicBarrier is initialized with number of parties to be wait and threads wait for each other by calling **CyclicBarrier.await() method which is a blocking method** in Java and blocks until all Thread or parties call await(). **In general calling await() is shout out that Thread is waiting on barrier**. await() is a blocking call but can be timed out or Interrupted by other thread

**CyclicBarrier in Java – Example**

Now we know what is CyclicBarrier in Java and it's time to see example of CyclicBarrier in Java. Here is a simple example of CyclicBarrier in Java on which we initialize CyclicBarrier with 3 parties, means in order to cross barrier, 3 thread needs to call await() method. each thread calls await method in short duration but they don't proceed until all 3 threads reached barrier, once all thread reach barrier, barrier gets broker and each thread started there execution from that point. Its much clear with the output of following example of CyclicBarrier in Java:

import java.util.concurrent.BrokenBarrierException;

import java.util.concurrent.CyclicBarrier;

import java.util.logging.Level;

import java.util.logging.Logger;

public class CyclicBarrierExample {

//Runnable task for each thread

private static class Task implements Runnable {

private CyclicBarrier barrier;

public Task(CyclicBarrier barrier) {

this.barrier = barrier;

}

@Override

public void run() {

try {

System.out.println(Thread.currentThread().getName() + " is waiting on barrier");

barrier.await();

System.out.println(Thread.currentThread().getName() + " has crossed the barrier");

} catch (InterruptedException ex) {

Logger.getLogger(CyclicBarrierExample.class.getName()).log(Level.SEVERE, null, ex);

} catch (BrokenBarrierException ex) {

Logger.getLogger(CyclicBarrierExample.class.getName()).log(Level.SEVERE, null, ex);

}

}

}

public static void main(String args[]) {

//creating CyclicBarrier with 3 parties i.e. 3 Threads needs to call await()

**final CyclicBarrier cb = new CyclicBarrier(3, new Runnable(){**

**@Override**

**public void run(){**

**//This task will be executed once all thread reaches barrier**

**System.out.println("All parties are arrived at barrier, lets play");**

**}**

**});**

//starting each of thread

Thread t1 = new Thread(new Task(cb), "Thread 1");

Thread t2 = new Thread(new Task(cb), "Thread 2");

Thread t3 = new Thread(new Task(cb), "Thread 3");

t1.start();

t2.start();

t3.start();

}

}

**Output**:

Thread 1 is waiting on barrier

Thread 3 is waiting on barrier

Thread 2 is waiting on barrier

All parties are arrived at barrier, lets play

Thread 3 has crossed the barrier

Thread 1 has crossed the barrier

Thread 2 has crossed the barrier

**When to use CyclicBarrier in Java**

Given the nature of CyclicBarrier it can be very handy to implement map reduce kind of task similar to fork-join framework of Java 7, where a big task is broker down into smaller pieces and to complete the task you need output from individual small task e.g. to count population of India you can have 4 threads which counts population from North, South, East and West and once complete they can wait for each other, When last thread completed there task, Main thread or any other thread can add result from each zone and print total population. You can use CyclicBarrier in Java :

1) To implement multi player game which can not begin until all player has joined.

2) Perform lengthy calculation by breaking it into smaller individual tasks, In general to implement Map reduce technique.

**Important point of CyclicBarrier in Java**

1. CyclicBarrier can perform a completion task once all thread reaches to barrier, This can be provided while creating CyclicBarrier.

2. If CyclicBarrier is initialized with 3 parties means 3 thread needs to call await method to break the barrier.

3. Thread will block on await() until all parties reaches to barrier, another thread interrupt or await timed out.

4. If another thread interrupt the thread which is waiting on barrier it will throw BrokernBarrierException as shown below:

java.util.concurrent.BrokenBarrierException

at java.util.concurrent.CyclicBarrier.dowait(CyclicBarrier.java:172)

at java.util.concurrent.CyclicBarrier.await(CyclicBarrier.java:327)

5.CyclicBarrier.reset() put Barrier on its initial state, other thread which is waiting or not yet reached barrier will terminate with java.util.concurrent.BrokenBarrierException.

CyclicBarrier makes sense in scenarios where the end user wants to ensure that multiple threads do their individual processing up-to a certain point and then wait for their colleagues to complete their processing. The class is named “cyclic” because it can be reused after the waiting threads are released. CountdownLatch on the other hand make sense for applications where the individual threads can countdown and proceed with their activities.

**Examples on CyclicBarrier in Java**

**Scenario - 1 : A person comes to a bank and wants to open an account in the bank. Our account creation process works on a multi threaded environment. When a person submits the form to the bank, customer creation process starts in the following manner.**

**1. Create a thread which will gather customer information and maintains data entry and it will halt for some time.**

**2. Create a thread which will make AML(Anti Monoey Laundry) validation and it will halt for some time.**

**3. Create a thread which will initiate card processing and it will halt for some time.**

**Finally there is one main thread which will decide whether to create the account or not based upon the status of the processes that bank has done. After the main thread's decision, all the threads which are in halting condition will continue to complete the bank process for customer creation. The program is given below.**

import java.util.List;

import java.util.concurrent.BrokenBarrierException;

import java.util.concurrent.CyclicBarrier;

import java.util.concurrent.TimeUnit;

public class DataEntryThread implements Runnable {

private CyclicBarrier barrier1;

private List<Boolean> statusDetails;

public DataEntryThread(CyclicBarrier barrier1,List<Boolean> statusDetails) {

this.barrier1 = barrier1;

this.statusDetails = statusDetails;

}

@Override

public void run() {

try {

System.out.println(Thread.currentThread().getName()+"started running ...");

System.out.println(Thread.currentThread().getName()+" Gathering customer details and maintaining data entry in the bank");

System.out.println(Thread.currentThread().getName()+" Data entry over and waiting for Passbook creation ... ");

statusDetails.add(true);

TimeUnit.SECONDS.sleep(2);

barrier1.await();

System.out.println(Thread.currentThread().getName()+" is going to perform other tasks to complete ...");

TimeUnit.SECONDS.sleep(2);

System.out.println(Thread.currentThread().getName()+" completed the task and Passbok created successfully ...");

}

catch( InterruptedException ie ) {

System.out.println("Interrupted Exception thrown ... ");

ie.printStackTrace();

}

catch( BrokenBarrierException bbe ) {

System.out.println("Broken Barrier Exception thrown ... ");

bbe.printStackTrace();

}

catch (Exception e) {

System.out.println("Exception thrown ... ");

e.printStackTrace();

}

}

}

import java.util.List;

import java.util.concurrent.BrokenBarrierException;

import java.util.concurrent.CyclicBarrier;

import java.util.concurrent.TimeUnit;

public class AmlValidationThread implements Runnable {

private CyclicBarrier barrier1;

private List<Boolean> statusDetails;

public AmlValidationThread(CyclicBarrier barrier1,List<Boolean> statusDetails) {

this.barrier1 = barrier1;

this.statusDetails = statusDetails;

}

@Override

public void run() {

try {

System.out.println(Thread.currentThread().getName()+" started running ...");

System.out.println(Thread.currentThread().getName()+" Going to make AML validation for the customer by making Web service call ... ");

TimeUnit.SECONDS.sleep(6);

statusDetails.add(false);

barrier1.await();

System.out.println(Thread.currentThread().getName()+" is going to perform other tasks to complete ...");

TimeUnit.SECONDS.sleep(4);

System.out.println(Thread.currentThread().getName()+" completed the AML validation successfully ...");

}

catch( InterruptedException ie ) {

System.out.println("Interrupted Exception thrown ... ");

ie.printStackTrace();

}

catch( BrokenBarrierException bbe ) {

System.out.println("Broken Barrier Exception thrown ... ");

bbe.printStackTrace();

}

catch (Exception e) {

System.out.println("Exception thrown ... ");

e.printStackTrace();

}

}

}

import java.util.List;

import java.util.concurrent.BrokenBarrierException;

import java.util.concurrent.CyclicBarrier;

import java.util.concurrent.TimeUnit;

public class CardsThread implements Runnable {

private CyclicBarrier barrier1;

private List<Boolean> statusDetails;

public CardsThread(CyclicBarrier barrier1,List<Boolean> statusDetails) {

this.barrier1 = barrier1;

this.statusDetails = statusDetails;

}

@Override

public void run() {

try {

System.out.println(Thread.currentThread().getName()+" started running ...");

System.out.println(Thread.currentThread().getName()+" Gathering datstomer details for creating Debit Cards");

System.out.println(Thread.currentThread().getName()+" Data entry over and waiting for Debit Card creation ... ");

TimeUnit.SECONDS.sleep(5);

statusDetails.add(true);

barrier1.await();

System.out.println(Thread.currentThread().getName()+" is going to perform other tasks to complete ...");

TimeUnit.SECONDS.sleep(3);

System.out.println(Thread.currentThread().getName()+" completed the task for card creation ...");

}

catch( InterruptedException ie ) {

System.out.println("Interrupted Exception thrown ... ");

ie.printStackTrace();

}

catch( BrokenBarrierException bbe ) {

System.out.println("Broken Barrier Exception thrown ... ");

bbe.printStackTrace();

}

catch (Exception e) {

System.out.println("Exception thrown ... ");

e.printStackTrace();

}

}

}

import java.util.List;

import java.util.concurrent.CopyOnWriteArrayList;

import java.util.concurrent.CyclicBarrier;

import java.util.concurrent.TimeUnit;

public class TestAccountCreation {

public static void main(String[] args) {

final List<Boolean> statusDetails = new CopyOnWriteArrayList<Boolean>();

CyclicBarrier barrier = new CyclicBarrier(3, new Runnable() {

@Override

public void run() {

try {

System.out.println("All the threads have completed their 50% of tasks.");

System.out.println("All the threads have come to this barrier point.");

TimeUnit.SECONDS.sleep(3);

if( statusDetails.contains(false))

System.out.println("One of the validation fails, account can not be created ... ");

else

System.out.println("Allow other threads to complete their remianing tasks and account creation is successful ...");

}

catch (Exception e) {

e.printStackTrace();

}

}

});

Runnable dateEntryRunnable = new DataEntryThread(barrier,statusDetails);

Runnable amlRunnable = new AmlValidationThread(barrier,statusDetails);

Runnable cardsRunnable = new CardsThread(barrier,statusDetails);

Thread dataEntryTh = new Thread( dateEntryRunnable );

Thread amlTh = new Thread( amlRunnable );

Thread cardsTh = new Thread( cardsRunnable );

dataEntryTh.start();

amlTh.start();

cardsTh.start();

boolean flag = true;

while( flag ) {

if( !dataEntryTh.isAlive() & !amlTh.isAlive() & !cardsTh.isAlive() )

flag = false;

}

System.out.println("All the multi threaded tasks have been completed ... ");

}

}

**Scenario - 2 : In a sports event there are 3 runners and they are running. They will run independently.**

**1. Runners will run and finally come to the barrier point where based upon the shortest time prize will be given**

**2. After the prize distribution, runners will go go home.**

import java.util.Map;

import java.util.concurrent.BrokenBarrierException;

import java.util.concurrent.CyclicBarrier;

import java.util.concurrent.TimeUnit;

public class Runner1 implements Runnable {

private CyclicBarrier barrier1;

private Map<String,Long> timeMap;

public Runner1(CyclicBarrier barrier1,Map<String,Long> timeMap) {

this.barrier1 = barrier1;

this.timeMap = timeMap;

}

@Override

public void run() {

try {

System.out.println(Thread.currentThread().getName()+" started running ...");

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

TimeUnit.SECONDS.sleep(2);

long time = System.nanoTime();

timeMap.put("Runner3", time);

barrier1.await();

System.out.println(Thread.currentThread().getName()+" is going to perform other tasks to complete ...");

TimeUnit.SECONDS.sleep(2);

System.out.println(Thread.currentThread().getName()+" completed the event and runner will go home ...");

}

catch( InterruptedException ie ) {

System.out.println("Interrupted Exception thrown ... ");

ie.printStackTrace();

}

catch( BrokenBarrierException bbe ) {

System.out.println("Broken Barrier Exception thrown ... ");

bbe.printStackTrace();

}

catch (Exception e) {

System.out.println("Exception thrown ... ");

e.printStackTrace();

}

}

}

import java.util.Map;

import java.util.concurrent.BrokenBarrierException;

import java.util.concurrent.CyclicBarrier;

import java.util.concurrent.TimeUnit;

public class Runner2 implements Runnable {

private CyclicBarrier barrier2;

private Map<String,Long> timeMap;

public Runner2(CyclicBarrier barrier1,Map<String,Long> timeMap) {

this.barrier2 = barrier1;

this.timeMap = timeMap;

}

@Override

public void run() {

try {

System.out.println(Thread.currentThread().getName()+" started running ...");

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

TimeUnit.SECONDS.sleep(4);

long time = System.nanoTime();

timeMap.put("Runner3", time);

barrier2.await();

System.out.println(Thread.currentThread().getName()+" is going to perform other tasks to complete ...");

TimeUnit.SECONDS.sleep(4);

System.out.println(Thread.currentThread().getName()+" completed the event and runner will go home ...");

}

catch( InterruptedException ie ) {

System.out.println("Interrupted Exception thrown ... ");

ie.printStackTrace();

}

catch( BrokenBarrierException bbe ) {

System.out.println("Broken Barrier Exception thrown ... ");

bbe.printStackTrace();

}

catch (Exception e) {

System.out.println("Exception thrown ... ");

e.printStackTrace();

}

}

}

import java.util.Map;

import java.util.concurrent.BrokenBarrierException;

import java.util.concurrent.CyclicBarrier;

import java.util.concurrent.TimeUnit;

public class Runner3 implements Runnable {

private CyclicBarrier barrier3;

private Map<String,Long> timeMap;

public Runner3(CyclicBarrier barrier1,Map<String,Long> timeMap) {

this.barrier3 = barrier1;

this.timeMap = timeMap;

}

@Override

public void run() {

try {

System.out.println(Thread.currentThread().getName()+" started running ...");

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

TimeUnit.SECONDS.sleep(6);

long time = System.nanoTime();

timeMap.put("Runner3", time);

barrier3.await();

System.out.println(Thread.currentThread().getName()+" is going to perform other tasks to complete ...");

TimeUnit.SECONDS.sleep(6);

System.out.println(Thread.currentThread().getName()+" completed the event and runner will go home ...");

}

catch( InterruptedException ie ) {

System.out.println("Interrupted Exception thrown ... ");

ie.printStackTrace();

}

catch( BrokenBarrierException bbe ) {

System.out.println("Broken Barrier Exception thrown ... ");

bbe.printStackTrace();

}

catch (Exception e) {

System.out.println("Exception thrown ... ");

e.printStackTrace();

}

}

}

import java.util.List;

import java.util.Map;

import java.util.concurrent.ConcurrentHashMap;

import java.util.concurrent.CopyOnWriteArrayList;

import java.util.concurrent.CyclicBarrier;

import java.util.concurrent.TimeUnit;

public class TestCyclicBarrier {

public static void main(String[] args) {

final Map<String,Long> timeMap = new ConcurrentHashMap<String, Long>();

CyclicBarrier barrier = new CyclicBarrier(3, new Runnable() {

@Override

public void run() {

try {

System.out.println("All the threads have completed their running event.");

System.out.println("All the threads have come to this barrier point.");

System.out.println("Result will be declared from the time and prize will be awarded ...");

TimeUnit.SECONDS.sleep(3);

System.out.println("Allow other threads to complete their remianing tasks ..");

}

catch (Exception e) {

e.printStackTrace();

}

}

});

Runnable runnable1 = new Runner1( barrier,timeMap);

Runnable runnable2 = new Runner1( barrier,timeMap);

Runnable runnable3 = new Runner1( barrier,timeMap);

Thread th1 = new Thread(runnable1);

Thread th2 = new Thread(runnable2);

Thread th3 = new Thread(runnable3);

th1.start();

th2.start();

th3.start();

boolean flag = true;

while( flag ) {

if( !th1.isAlive() & !th2.isAlive() & !th3.isAlive() )

flag = false;

}

System.out.println("All the multi threaded tasks have been completed ... ");

}

}

A different example on CyclicBarrier

import java.util.Date;

import java.util.concurrent.BrokenBarrierException;

import java.util.concurrent.CyclicBarrier;

public class CyclicBarrier2Test {

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

CyclicBarrier barrier=new CyclicBarrier(2);

new Thread(new Worker(barrier)).start();

barrier.await();

System.out.println("Done Cycle 1\n");

new Thread(new Worker(barrier)).start();

barrier.await();

System.out.println("Done Cycle 2");

}

public static class Worker implements Runnable {

private CyclicBarrier barrier;

public Worker(CyclicBarrier barrier) {

this.barrier=barrier;

}

@Override

public void run() {

try {

System.out.println("Thread "+Thread.currentThread().getId()+" at: "+new Date());

barrier.await();

} catch (Exception e) {

System.out.println(e.getMessage());

}

}

}

}

One More Example

import java.util.concurrent.BrokenBarrierException;

import java.util.concurrent.CyclicBarrier;

public class CyclicBarrier1Test {

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

CyclicBarrier barrier=new CyclicBarrier(2);

barrier.await();//will not return until another thread call await.

}

}

In the above case, your system will be hung.

**Example on CyclicBarrier – Nov-2017**

There are two threads and after completion of all the threads, perform the final task using CyclicBarrier.

**Task1.java**  
**import** java.util.concurrent.BrokenBarrierException;  
**import** java.util.concurrent.CyclicBarrier;  
**import** java.util.concurrent.TimeUnit;  
  
**public class** Task1 **implements** Runnable {  
  
 **private** CyclicBarrier **barrier**;  
  
 **public** Task1(CyclicBarrier barrier) {  
 **this**.**barrier** = barrier;  
 }  
  
 @Override  
 **public void** run() {  
 System.***out***.printf(**"\nStarting task1 ..."**);  
 **try** {  
 TimeUnit.***SECONDS***.sleep(3);  
 System.***out***.printf(**"\n"**+Thread.*currentThread*().getName()+**" working on task1"**);  
 **barrier**.await();  
 System.***out***.printf(**"\n"**+Thread.*currentThread*().getName()+**" waiting for others to complete ... "**);  
 }  
 **catch** (InterruptedException ie) {  
 ie.printStackTrace();  
 } **catch** (BrokenBarrierException e) {  
 e.printStackTrace();  
 }  
 System.***out***.printf(**"\n"**+Thread.*currentThread*().getName()+**" completed task1"**);  
 }  
}

**Task 2**  
**import** java.util.concurrent.BrokenBarrierException;  
**import** java.util.concurrent.CyclicBarrier;  
**import** java.util.concurrent.TimeUnit;  
  
**public class** Task2 **implements** Runnable {  
  
 **private** CyclicBarrier **barrier**;  
  
 **public** Task2(CyclicBarrier barrier) {  
 **this**.**barrier** = barrier;  
 }  
  
 @Override  
 **public void** run() {  
 System.***out***.printf(**"\nStarting task2 ..."**);  
 **try** {  
 TimeUnit.***SECONDS***.sleep(3);  
 System.***out***.printf(**"\n"**+Thread.*currentThread*().getName()+**" working on task2"**);  
 **barrier**.await();  
 System.***out***.printf(**"\n"**+Thread.*currentThread*().getName()+**" waiting for others to complete ... "**);  
 }  
 **catch** (InterruptedException ie) {  
 ie.printStackTrace();  
 } **catch** (BrokenBarrierException e) {  
 e.printStackTrace();  
 }  
 System.***out***.printf(**"\n"**+Thread.*currentThread*().getName()+**" completed task2"**);  
 }  
}

**Test1.java**  
**import** java.util.concurrent.BrokenBarrierException;  
**import** java.util.concurrent.CyclicBarrier;  
**import** java.util.concurrent.TimeUnit;  
  
**public class** Test1 {  
  
 **private static void** performFinal() {  
 **try** {  
 TimeUnit.***SECONDS***.sleep(2);  
 System.***out***.printf(**"\nPerformed the final one ... "**);  
 }  
 **catch** (InterruptedException ie) {  
 ie.printStackTrace();  
 }  
 }  
  
 **public static void** executeParallely() {  
 CyclicBarrier barrier = **new** CyclicBarrier(3);  
  
 Thread thread1 = **new** Thread(**new** Task1(barrier));  
 thread1.setName(**"Task1"**);  
  
 Thread thread2 = **new** Thread(**new** Task2(barrier));  
 thread2.setName(**"Task2"**);  
 System.***out***.printf(**"\nStarting both threads ..."**);  
 thread1.start();  
 thread2.start();  
  
 **try** {  
 barrier.await();  
 System.***out***.printf(**"\nAll tasks completed successfully ..."**);  
  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 } **catch** (BrokenBarrierException e) {  
 e.printStackTrace();  
 }  
 }  
  
 **public static void** main(String[] args) {  
 *executeParallely*();  
 *//Perform the final task  
 performFinal*();  
 }  
}

**OUTPUT**

Starting both threads ...

Starting task1 ...

Starting task2 ...

Task1 working on task1

Task2 working on task2

Task2 waiting for others to complete ...

Task2 completed task2

Task1 waiting for others to complete ...

All tasks completed successfully ...

Task1 completed task1

Performed the final one ...

Process finished with exit code 0

**Problem: You need to do 3 validations in parallel and perform a final validation after this.**

Imagine, make Aadhar, PanNo, Passport validation, after this, you have to do a fourth validation.

**import** java.util.concurrent.BrokenBarrierException;  
**import** java.util.concurrent.CyclicBarrier;  
**import** java.util.concurrent.TimeUnit;  
  
**public class AadharValidation** **implements** Runnable {  
 **private CyclicBarrier cyclicBarrier;** **private int timeInSecs**;  
  
 **public** AadharValidation(CyclicBarrier cyclicBarrier, **int** timeInSecs) {  
 **this**.**cyclicBarrier** = cyclicBarrier;  
 **this**.**timeInSecs** = timeInSecs;  
 }  
  
 @Override  
 **public void** run() {  
 **try** {  
 System.***out***.println(Thread.*currentThread*().getName() + **" started validation"**);  
 TimeUnit.***SECONDS***.sleep(**timeInSecs**);  
 System.***out***.println(Thread.*currentThread*().getName() + **" waiting ..."**);  
 **cyclicBarrier.await();** System.***out***.println(Thread.*currentThread*().getName() + **" completed validation"**);  
 } **catch** (InterruptedException | BrokenBarrierException e) {  
 e.printStackTrace();  
 }  
 }  
}

**import** java.util.concurrent.BrokenBarrierException;  
**import** java.util.concurrent.CyclicBarrier;  
**import** java.util.concurrent.TimeUnit;  
  
**public class PanNoValidation** **implements** Runnable {  
 **private CyclicBarrier cyclicBarrier;** **private int timeInSecs**;  
  
 **public** PanNoValidation(CyclicBarrier cyclicBarrier, **int** timeInSecs) {  
 **this**.**cyclicBarrier** = cyclicBarrier;  
 **this**.**timeInSecs** = timeInSecs;  
 }  
  
 @Override  
 **public void** run() {  
 **try** {  
 System.***out***.println(Thread.*currentThread*().getName() + **" started validation"**);  
 TimeUnit.***SECONDS***.sleep(**timeInSecs**);  
 System.***out***.println(Thread.*currentThread*().getName() + **" waiting ..."**);  
 **cyclicBarrier.await();** System.***out***.println(Thread.*currentThread*().getName() + **" completed validation"**);  
 } **catch** (InterruptedException | BrokenBarrierException e) {  
 e.printStackTrace();  
 }  
 }  
}

**import** java.util.concurrent.BrokenBarrierException;  
**import** java.util.concurrent.CyclicBarrier;  
**import** java.util.concurrent.TimeUnit;  
  
**public class PassportValidation** **implements** Runnable {  
 **private CyclicBarrier cyclicBarrier;** **private int timeInSecs**;  
  
 **public** PassportValidation(CyclicBarrier cyclicBarrier, **int** timeInSecs) {  
 **this**.**cyclicBarrier** = cyclicBarrier;  
 **this**.**timeInSecs** = timeInSecs;  
 }  
  
 @Override  
 **public void** run() {  
 **try** {  
 System.***out***.println(Thread.*currentThread*().getName() + **" started validation"**);  
 TimeUnit.***SECONDS***.sleep(**timeInSecs**);  
 System.***out***.println(Thread.*currentThread*().getName() + **" waiting ..."**);  
 **cyclicBarrier.await();** System.***out***.println(Thread.*currentThread*().getName() + **" completed validation"**);  
 } **catch** (InterruptedException | BrokenBarrierException e) {  
 e.printStackTrace();  
 }  
 }  
}

**import** java.util.concurrent.BrokenBarrierException;  
**import** java.util.concurrent.CyclicBarrier;  
**import** java.util.concurrent.TimeUnit;  
  
**public class EndTask** **implements** Runnable {  
 **private int timeInSecs**;  
  
 **public** EndTask(**int** timeInSecs) {  
 **this**.**timeInSecs** = timeInSecs;  
 }  
  
 @Override  
 **public void** run() {  
 **try** {  
 System.***out***.println(**"All the parties have arrived here..."**);  
 System.***out***.println(**"Final task is going to be performed now ..."**);  
 TimeUnit.***SECONDS***.sleep(**timeInSecs**);  
 System.***out***.println(**"Now other threads can complete the tasks"**);  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
 }  
}

**import** java.util.concurrent.BrokenBarrierException;  
**import** java.util.concurrent.CyclicBarrier;  
**import** java.util.concurrent.TimeUnit;  
  
**public class** TestCyclicBarrier1 {  
 **public static void** main(String[] args) **throws** Exception {  
 *// Thread name has no impact in case of CyclicBarrier Runnable task.* Thread endTask = **new** Thread(**new** EndTask(3));  
 CyclicBarrier cyclicBarrier = **new** CyclicBarrier(3, endTask);  
 *// You can also write like this  
 // CyclicBarrier cyclicBarrier = new CyclicBarrier(3);* Thread aadharThread = **new** Thread(**new** AadharValidation(cyclicBarrier, 7), **"Aadhar"**);  
 Thread passportThread = **new** Thread(**new** PassportValidation(cyclicBarrier, 5), **"Passport"**);  
 Thread panNoThread = **new** Thread(**new** PanNoValidation(cyclicBarrier, 3), **"PanNo"**);  
  
 aadharThread.start();  
 passportThread.start();  
 panNoThread.start();  
  
 aadharThread.join();  
 passportThread.join();  
 panNoThread.join();  
  
 System.***out***.println(**"All threads completed the tasks"**);  
 }  
}

OUTPUT

Passport started validation

Aadhar started validation

PanNo started validation

PanNo waiting ...

Passport waiting ...

Aadhar waiting ...

All the parties have arrived here...

Final task is going to be performed now ...

Now other threads can complete the tasks

Aadhar completed validation

Passport completed validation

PanNo completed validation

All threads completed the tasks

**Note: In the end task, thread name has no impact with CyclicBarrier.**

**How to change the thread name ?**

In case extending thread class, in the constructor use **super(“Thread-Name”)**. In case of Runnable, you have to user like this.

**Thread t1 = new Thread(new RunnableTask(),”Thread-Name”);**

**Good Example on CyclicBarrier as per Java 8**

**import** java.util.concurrent.BrokenBarrierException;  
**import** java.util.concurrent.CyclicBarrier;  
**import** java.util.concurrent.TimeUnit;  
  
**public class** TestBarrier {  
  
 **public static void** task(CyclicBarrier barrier, String name, **int** time) {  
 **try** {  
 Thread.*currentThread*().setName(name);  
 System.***out***.println(**"Executing Task ...."**+Thread.*currentThread*().getName());  
 TimeUnit.***SECONDS***.sleep(time);  
 barrier.await();  
 TimeUnit.***SECONDS***.sleep(2);  
 System.***out***.println(**"Task completed by ..."**+Thread.*currentThread*().getName());  
 } **catch**(InterruptedException ie) {  
 ie.printStackTrace();  
 } **catch** (BrokenBarrierException e) {  
 e.printStackTrace();  
 }  
 }  
  
 **public static void** endTask() {  
 System.***out***.println(**"All Task completed .."**);  
 }  
  
 **public static void** main(String[] args) {  
 Runnable runnable = () -> *endTask*();  
 CyclicBarrier barrier = **new** CyclicBarrier(3, runnable);  
  
 Runnable r1 = () -> *task*(barrier, **"John"**, 7);  
 Runnable r2 = () -> *task*(barrier, **"Vidya"**, 7);  
 Runnable r3 = () -> *task*(barrier, **"Peter"**, 7);  
  
 Thread t1 = **new** Thread(r1);  
 Thread t2 = **new** Thread(r2);  
 Thread t3 = **new** Thread(r3);  
  
 t1.start();  
 t2.start();  
 t3.start();  
  
 }  
}

**Usage of Callable Tasks**

**MyCallableTask.java**  
**import** java.util.concurrent.Callable;  
**import** java.util.concurrent.TimeUnit;  
  
**public class** MyCallableTask **implements** Callable<String> {  
  
 **private** String **name**;  
 **private int waitTimeIn**;  
  
 **public** MyCallableTask(String name, **int** waitTimeIn) {  
 **this**.**name** = name;  
 **this**.**waitTimeIn** = waitTimeIn;  
 Thread.*currentThread*().setName(name);  
 }  
  
 @Override  
 **public** String call() **throws** Exception {  
 TimeUnit.***SECONDS***.sleep(**waitTimeIn**);  
 String returnMsg = **name**+**" done ."**;  
 System.***out***.printf(**"\n "**+**name**+**" completed ..."**);  
 **return** returnMsg;  
 }  
}

Output

Task 2 completed ...

Task 1 completed ...

Task 3 completed ...

future.get = Task 2 done .

future.get = Task 1 done .

future.get = Task 3 done .

All task completed ...

**Test2.java**

**import** java.util.HashSet;  
**import** java.util.List;  
**import** java.util.Set;  
**import** java.util.concurrent.\*;  
  
**public class** Test2 {  
  
 **public static void** executeParallel() {  
 ExecutorService executorService = Executors.*newSingleThreadExecutor*();  
 Set<Callable<String>> callables = **new** HashSet<Callable<String>>();  
  
 callables.add(**new** MyCallableTask(**"Task 1"**, 3));  
 callables.add(**new** MyCallableTask(**"Task 2"**, 5));  
 callables.add(**new** MyCallableTask(**"Task 3"**, 7));  
  
 List<Future<String>> futures = **null**;  
 **try** {  
 futures = executorService.invokeAll(callables);  
 } **catch** (InterruptedException e) { e.printStackTrace(); }  
  
 **for** (Future<String> future : futures) {  
 **try** {  
 System.***out***.println(**"\nfuture.get = "** + future.get());  
 } **catch** (InterruptedException e) { e.printStackTrace();}

**catch** (ExecutionException e) { e.printStackTrace(); }  
 }  
 executorService.shutdown();  
 System.***out***.printf(**"All task completed ..."**);  
 }  
  
 **public static void** main(String[] args) {  
 *executeParallel*();  
 }  
}