**package** main.java.threads;

**import** java.util.concurrent.\*;

**import** java.util.List;

**import** java.util.ArrayList;

**public** **class** Future\_Callable {

**private** **static** **final** **int** *nThreads* = 3;

**public** **static** ExecutorService *e* = Executors.*newFixedThreadPool*(*nThreads*);

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

ExecutionException {

List<String> outputs = **new** ArrayList<String>();

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

MyCallable myCallable = **new** MyCallable();

myCallable.setNum(i);

// On the executor you can use the method submit to "submit"

// (instead of .execute(..)) a Callable and to get a future.

// To retrieve the result of the future use the get()

Future<String> future = *e*.submit(myCallable);

outputs.add(future.get());

}

**for** (String o : outputs) {

System.*out*.println(o);

}

*e*.shutdown();

}

}

// For threads to return a computed result/ per thread- use

// java.util.concurrent.Callable.

// Callable uses generic to define the type of object which is returned.

// It implements Callable<T> instead of Runnable in normal thread

// it should provide an implementation of call() function instead of run()

// It should return a value

**class** MyCallable **implements** Callable<String> {

**private** **int** num;

**public** **void** setNum(**int** n) {

**this**.num = n;

}

**public** String call() {

**int** sum = 0;

**for** (**int** i = 1; i <= num; i++) {

sum += i;

}

**return** "Sum of " + **this**.num + " numbers is: " + sum;

}

}

OUtPUT : Sum of 0 numbers is: 0

Sum of 1 numbers is: 1

Sum of 2 numbers is: 3

Sum of 3 numbers is: 6

Sum of 4 numbers is: 10

Sum of 5 numbers is: 15

Sum of 6 numbers is: 21

Sum of 7 numbers is: 28

Sum of 8 numbers is: 36

Sum of 9 numbers is: 45

package main.java.threads;

// HIGH LEVEL CONCURRENCY - through java.util.concurrent

import java.util.Random;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.TimeUnit;

// Executor Interface:

// The Executor interface provides a single method, execute, to replace (new Thread(r)).start(); with e.execute(r);

// It creates & launches new thread immediately. It uses an existing worker thread to run r, or to place r in a queue to wait for a worker thread to become available.

// ExecutorService Interface: Subinterface of Executor

// It provides more versatile submit method. Like execute, submit accepts Runnable objects, but ALSO accepts Callable objects,

// which allow the task to return a value. The submit method returns a Future object, which is used to retrieve the Callable return.

// It provides a number of methods for managing the shutdown of the executor.

// The ScheduledExecutorService interface : Subinterface of ExecutorService

// It executes a Runnable or Callable task after a specified delay or repeated intervals (scheduleAtFixedRate and scheduleWithFixedDelay)

# public class Executor\_ThreadPools {

public static final int nThreads = 10;

// Note - There is no "new" instantiation here

// It creates a thread pool of nThreads and allocate these worker threads to execute Runnable passed into .execute(..)

// When thread finishes - it comes back to the pool and again is allocated to another waiting runnable in a queue

public static ExecutorService e = Executors.newFixedThreadPool(nThreads);

public static Random random = new Random();

public static void main(String[] args) {

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

MyTask runnableObj = new MyTask();

runnableObj.setNum(i);

// assigning new runnable objects to threads in a pool

// note that the task that we submit has to be a thread class (implementing runnabe or extending thread)

e.execute(runnableObj);

}

// initiates an orderly shutdown in which previously submitted tasks are executed, but no new tasks will be accepted.

// Invocation has no additional effect if already shut down.

e.shutdown();

try {

// Blocks until all tasks have completed execution after a shutdown request, or the timeout occurs, or the current thread is interrupted, whichever happens first.

e.awaitTermination(1000, TimeUnit.SECONDS);

} catch (InterruptedException e1) {

e1.printStackTrace();

}

}

}

class MyTask implements Runnable {

private int num;

public void setNum(int n) {

this.num = n;

}

public void run() {

//doToRandomSleep();

System.out.println("This is the num - " + this.num);

}

private void doRandomSleep() {

try {

Thread.sleep(Executor\_ThreadPools.random.nextInt(5) \* 1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

# java.util.concurrent.CountDownLatch Example

CountDownLatch is used in synchronization to allow one or more threads to wait until a set of operations being performed in other threads completes.

For Example, it can be used when a Thread has to wait until few dependent threads have started. CountDownLatch is initialized with a given count. This count is decremented by calls to the countDown() method. Each Thread on whose completion the main Thread is waiting will call countDown() method and decrease the count. Once all such threads are executed the count will become 0 and then the waiting main thread can start its execution.

JavaCountDownLatchExample.java

package com.jcg.example;

import java.util.concurrent.CountDownLatch;

public class JavaCountDownLatchExample {

public static void main(String[] args) {

// intialising count down latch by 2, as it will wait for 2 threads to

// finish execution

final CountDownLatch latch = new CountDownLatch(2);

// making two threads for 2 services

Thread serviceOneThread = new Thread(new ServiceOne(latch));

Thread serviceTwoThread = new Thread(new ServiceTwo(latch));

serviceOneThread.start();

serviceTwoThread.start();

// latch waits till the count becomes 0

// this way we can make sure that the execution of main thread only

// finishes ones 2 services have executed

try {

latch.await();

System.out.println("Starting main Thread!!!");

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

}

Output:

started service Two

Started service One

Starting main Thread!!!

As we can see in the output that the main thread’s execution waited until service One and Service Two Threads have been completed.

Let us also see the code for ServiceOne and ServiceTwo classes :

ServiceOne.java

package com.jcg.example;

import java.util.concurrent.CountDownLatch;

public class ServiceOne implements Runnable{

private final CountDownLatch latch;

public ServiceOne(CountDownLatch latch) {

this.latch = latch;

}

public void run() {

System.out.println("Started service One");

//reduce count of Count Down Latch by 1.

latch.countDown();

}

}

ServiceTwo.java

**package** com.jcg.example;

**import** java.util.concurrent.CountDownLatch;

**public** **class** ServiceTwo **implements** Runnable{

**private** **final** CountDownLatch latch;

**public** ServiceTwo(CountDownLatch latch) {

**this**.latch = latch;

}

@Override

**public** **void** run() {

System.*out*.println("started service Two");

latch.countDown();

}

}

In the example above we saw how we can use CountDownLatch to make sure that a thread stays in wait state until other threads have finished execution.

Download Source Code

So, In this example we saw how we can use CountDownLatch in Java.

# CyclicBarrier Concept

CyclicBarrier is another concurrency utility introduced in Java 5 which is used when a number of threads (also known as parties) wants to wait for each other at a common point, also known as barrier before starting processing again.

Its similar to CountDownLatch but instead of calling countDown() each thread calls await() and when last thread calls await() which signals that it has reached barrier, all thread started processing again, also known as barrier is broken.

You can use CyclicBarrier wherever you want to use CountDownLatch, but opposite is not possible because you can not reuse the latch once count reaches to zero.

Some of the common usage of CyclicBarrier is in writing unit test for concurrent program, to simulate concurrency in test class or calculating final result after individual task has completed.   
 [Java Concurrency in Practice by Brian Goetz](http://www.amazon.com/dp/0321349601/?tag=javamysqlanta-20), I strongly recommend this book to anyone seriously wants to master threading and concurrency in Java

## CyclicBarrier Example in Java

In this program, we have four worker threads and one main thread, which is running your main method. We have an object of CyclicBarrier, which is initialized with parties = 4, the argument we passed in CyclicBarrier constructor is nothing but number of party, which is actually number of threads to stop at barrier. The barrier will not be broken until all parties are arrived. A party (thread) is said to be arrived with it call barrier.await() method.

If you look at their [Runnable implementation](http://java67.blogspot.sg/2012/08/what-is-thread-and-runnable-in-java.html), you will find that each party sleep for some seconds and then call await() method on barrier. The sleep is introduced so that every thread calls barrier method after some time.  Sleep time is also on increasing order, which means PARTY-4 should be the last one to call await. So as per our theory, every thread (party) should wait after calling await()until the last thread (PARTY-4) calls the await() method, after that every thread should wake up and start processing. Of-course they need to compete for CPU and they will start running once they got the CPU from thread scheduler, but what is more important is that once barrier is broken, each thread (party) becomes eligible for scheduling. By the way, you can reuse the barrier even after its broken this is where [CyclicBarrier is different than CountDownLatch](http://java67.blogspot.sg/2012/08/difference-between-countdownlatch-and-cyclicbarrier-java.html).

import java.util.concurrent.BrokenBarrierException;

import java.util.concurrent.CyclicBarrier;

/\*\*

\* Java Program to demonstrate how to use CyclicBarrier, Its used when number of threads

\* needs to wait for each other before starting again.

\*

\* @author Javin Paul

\*/

public class HelloHP {

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

CyclicBarrier barrier = new CyclicBarrier(4);

Party first = new Party(1000, barrier, "PARTY-1");

Party second = new Party(2000, barrier, "PARTY-2");

Party third = new Party(3000, barrier, "PARTY-3");

Party fourth = new Party(4000, barrier, "PARTY-4");

first.start();

second.start();

third.start();

fourth.start();

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

}

}

class Party extends *Thread* {

private int duration;

private CyclicBarrier barrier;

public Party(int duration, CyclicBarrier barrier, String name) {

super(name);

this.duration = duration;

this.barrier = barrier;

}

@Override

public void run() {

try {

Thread.sleep(duration);

System.out.println(Thread.currentThread().getName() + " is calling await()");

barrier.await();

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

} catch (InterruptedException | BrokenBarrierException e) {

e.printStackTrace();

}

}

}

Output

main has finished

PARTY-1 is calling await()

PARTY-2 is calling await()

PARTY-3 is calling await()

PARTY-4 is calling await()

PARTY-4 has started running again

PARTY-1 has started running again

PARTY-2 has started running again

PARTY-3 has started running again

If you look at the output is exactly matches with our theory. Each worker thread (PARTY 1 - 3) calls the await() method and then they stop processing until PARTY-4 comes and call await() method, after that every thread gets a wakeup call and started execution again, depending upon when they are scheduled by Java thread scheduler. This is how CyclicBarrier class works. You can still reuse the barrier object and if a thread calls barrier.await() again, it will wait for four worker thread before it gets wake up call. By the way, If barrier is broken before a thread calls await() then this method will throw BrokenBarrierException.

## When to use CyclicBarrier in Java Program

It is a very useful class and have several practical uses. You can use this to perform final task once individual task are completed. You can use it to write some unit tests to check some variants as well. Remember you can reuse the barrier as opposed to latch.  One a side note, this CyclicBarrier example is also a good example of how to catch multiple exception in one catch block in Java, [a feature introduced in JDK 1.7](http://javarevisited.blogspot.sg/2014/04/10-jdk-7-features-to-revisit-before-you.html). You can see that we have two unrelated exceptions InterruptedException and BrokenBarrierException, but we have caught then in same catch block, because of this feature, this code requires Java 7 to run. If you are not using JDK 7 then just use two catch block instead of one.

You should use it when one thread needs to wait for a fixed number of thread before starting an event e.g. Party. You can use CyclicBarrier to write concurrency unit test and implement generic algorithms in Java.

## Some real time examples of CyclicBarrier

In real time, a Cyclic Barrier can be used in download managers. Download managers, as we know, download the data in packets i.e. they split a file into parts (say 8 parts) and then download each part individually giving each part to a separate thread (i.e. 8 parts, 8 threads). Once after all the threads finish their work (i.e. all the parts are downloaded), the parts are combined. In this process, any part can be downloaded first. Consider, here, that a Cyclic barrier is created with 8 parties (threads), so await() is called in all the 8 threads. And that the barrier thread's job is to combine those downloaded parts. The thread that completes downloading its part, calls the await() method and when the last await() method is called (i.e. the last thread to complete the download calls the await() method), then the barrier thread is executed thereby combining all the downloaded parts into a single file. Here, as you can see, the barrier thread cannot execute until all the threads call await() method because, it to combine the parts, they must be downloaded first.

A simple thing to remember here, is that there is possibly no work, that 8 download threads can do after they reached the barrier. Because, they completed downloading their parts and after the barrier thread is executed, the parts are combined. The downloading is completed.   
  
Recommended Java concurrency resources for further learning:

* Java documentation of CountDownLatch ([documentation](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CountDownLatch.html))
* The Art of Multiprocessor Programming by Maurice Herlihy  ([the book](http://www.amazon.com/The-Multiprocessor-Programming-Revised-Reprint/dp/0123973376?tag=javamysqlanta-20))
* Java Concurrency in Practice by Brian Goetz ([the book](http://www.amazon.com/dp/0321349601/?tag=javamysqlanta-20))

# ****Common questions****

Be prepared to face these questions related to CountDownLatch in your next interview.

* What are main methods CountDownLatch class has?

To Download the sourcecode of above example application follow given link.