**Multithreading Interview Questions**

Contents

[1. Difference between semaphore and mutex? 2](#_Toc486274187)

[2. Print odd and even number in sequence with using two threads in java? 6](#_Toc486274192)

[3. Explain Fork Join Framework? 9](#_Toc486274193)

[4. Explain Blocking queue in Java? 18](#_Toc486274201)

[5. Explain Coundownlatch in Java? 24](#_Toc486274202)

[6. Explain CyclicBarrier in Java? 28](#_Toc486274204)

[7. What’s volatile variable in Java? 32](#_Toc486274205)

[8. What’s Reentrant Lock in Java? 36](#_Toc486274207)

[9. Explain Java Inter-thread Communication using Piped Streams? 45](#_Toc486274208)

[10. Explain Java ThreadLocal Variables? 48](#_Toc486274209)

[11. Object level Locking vs. Class level Locking in Java 53](#_Toc486274210)

[12. Can we make an Array or ArrayList volatile in Java? 57](#_Toc486274211)

[13. Why wait(), notify() and notifyAll() must be called from synchronized block or method in Java? 59](#_Toc486274212)

[14. How to Implement own Thread pool in java? 61](#_Toc486274213)

1. **Difference between semaphore and mutex?**

## Semaphore:

* + - A semaphore restricts the number of **simultaneous users** of a shared resource up to a maximum number. Threads can request access to the resource (decrementing the semaphore), and can signal that they have finished using the resource (incrementing the semaphore).
    - Counting Semaphore in java maintains specified number of pass or permits
    - In order to access the shared resource, current thread must acquire a permit
    - If permit is already exhausted by other thread then it can wait until a permit is available due to release of permit from different thread
    - **java.util.Semaphore** class represent a Counting semaphore which is initialized with number of permits.
    - Semaphore provides two main method **acquire()** and **release()** for getting permits and releasing permits.
    - A Counting semaphore with one permit is known as binary semaphore because it has only two state permit available or permit unavailable.
    - They are able to make threads wait when counter value is zero i.e. they act as Locks with counter functionality.
    - When a thread has finished the use of the shared resource, it must release the semaphore so that the other threads can access the shared resource. That operation increases the internal counter of the semaphore.

## Example

import java.util.concurrent.Semaphore;  
 public class SemaphoreTest {  
  
    [Semaphore](http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/Semaphore.html) binary = new Semaphore(1);    
    public static void main(String args[]) {  
        final SemaphoreTest test = new SemaphoreTest();  
        new Thread(){  
            @Override  
            public void run(){  
              test.mutualExclusion();   
            }  
        }.start();        
        new Thread(){  
            @Override  
            public void run(){  
              test.mutualExclusion();   
            }  
        }.start();        
    }    
    private void mutualExclusion() {  
        try {  
            binary.acquire();  
            //mutual exclusive region  
            System.out.println(Thread.currentThread().getName() + " inside mutual exclusive region");  
            Thread.sleep(1000);  
        } catch (InterruptedException i.e.) {  
            ie.printStackTrace();  
        } finally {  
            binary.release();  
            System.out.println(Thread.currentThread().getName() + " outside of mutual exclusive region");  
        }  
    }     
}

Output:  
 Thread-0 inside mutual exclusive region  
 Thread-0 outside of mutual exclusive region  
 Thread-1 inside mutual exclusive region  
 Thread-1 outside of mutual exclusive region

## Mutex:

* + - The word mutex is shorthand for a primitive object that provides **MUTual EXclusion between threads.**
    - A mutual exclusion (mutex) is used cooperatively between threads to ensure that **only one of the cooperating threads is allowed to access the data** or run certain application code at a time.
    - Any thread that successfully locks the mutex is the owner until it unlocks the mutex.
    - Any thread that attempts to lock the mutex waits until the owner unlocks the mutex.
    - When the owner unlocks the mutex, control is returned to one waiting thread with that thread becoming the owner of the mutex.
    - There can be only one owner of a mutex.

## Example

import com.sun.corba.se.impl.orbutil.concurrent.Mutex;

public class MutexTest {

Mutex binary = new Mutex();

public static void main(String args[]) {

final MutexTest test = new MutexTest();

new Thread(){

@Override

public void run(){

test.mutualExclusion();

}

}.start();

new Thread(){

@Override

public void run(){

test.mutualExclusion();

}

}.start();

}

private void mutualExclusion() {

try {

binary.acquire();

//mutual exclusive region

System.out.println(Thread.currentThread().getName() + " inside mutual exclusive region");

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

} finally {

binary.release();

System.out.println(Thread.currentThread().getName() + " outside of mutual exclusive region");

}

}

}

Output:

Thread-0 inside mutual exclusive region

Thread-0 outside of mutual exclusive region

Thread-1 inside mutual exclusive region

Thread-1 outside of mutual exclusive region

1. **Print odd and even number in sequence with using two threads in java?**

public class EvenOddThreadExample {

public static void main(String[] args) {

SharedPrinter sp = new SharedPrinter();

Thread t1 = new Thread(new OddNumProducer(sp, 10));

Thread t2 = new Thread(new EvenNumProducer(sp,10));

t1.start();

t2.start();

}

}

class SharedPrinter {

boolean evenFlag = false;

public void printOddNumber(int number) {

synchronized(this) {

while (evenFlag) {

try {

wait();

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

System.out.println("Odd Number : " + number);

evenFlag = true;

notify();

}

}

public void printEvenNumber(int number) {

synchronized(this) {

while (!evenFlag) {

try {

wait();

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

System.out.println("Even Number : " + number);

evenFlag = false;

notify();

}

}

}

class OddNumProducer implements Runnable {

SharedPrinter sp;

int number;

public OddNumProducer(SharedPrinter s, int num) {

this.sp = s;

this.number = num;

}

@Override

public void run () {

for (int i=1; i <= number; i = i+2) {

sp.printOddNumber(i);

}

}

}

class EvenNumProducer implements Runnable {

SharedPrinter sp;

int number;

public EvenNumProducer(SharedPrinter s, int num) {

this.sp = s;

this.number = num;

}

@Override

public void run() {

for (int i = 2; i <= number; i=i+2) {

sp.printEvenNumber(i);

}

}

}

1. **Explain Fork Join Framework?**
   * + Basically the Fork-Join breaks the task at hand into mini-tasks until the mini-task is simple enough that it can be solved without further breakups.
     + It’s like a divide-and-conquer algorithm. One important concept to note in this framework is that ideally no worker thread is idle.
     + They implement a work-stealing algorithm in that idle workers steal the work from those workers who are busy.

## Core Classes used in Fork/Join Framework

* + - The core classes supporting the Fork-Join mechanism are ForkJoinPool and ForkJoinTask.

## ForkJoinPool

* + - The ForkJoinPool is basically a specialized implementation of ExecutorService implementing the work-stealing algorithm we talked about above. We create an instance of ForkJoinPool by providing the target parallelism level i.e. the number of processors as shown below:

ForkJoinPool pool = new ForkJoinPool(numberOfProcessors);

Where numberOfProcessors = Runtime.getRunTime().availableProcessors();

* + - If you use a no-argument constructor, by default, it creates a pool of size that equals the number of available processors obtained using above technique.
    - Although you specify any initial pool size, the pool adjusts its size dynamically in an attempt to maintain enough active threads at any given point in time. Another important difference compared to other ExecutorService's is that this pool need not be explicitly shutdown upon program exit because all its threads are in daemon mode.
    - There are three different ways of submitting a task to the ForkJoinPool.
      * execute() method //Desired asynchronous execution; call its fork method to split the work between multiple threads.
      * invoke() method: //Await to obtain the result; call the invoke method on the pool.
      * submit() method: //Returns a Future object that you can use for checking status and obtaining the result on its completion.

## ForkJoinTask

* + - This is an abstract class for creating tasks that run within a ForkJoinPool. The RecursiveAction and RecursiveTask are the only two direct, known subclasses of ForkJoinTask. The only difference between these two classes is that the RecursiveAction does not return a value while RecursiveTask does have a return value and returns an object of specified type.
    - In both cases, you would need to implement the ***compute method*** in your subclass that performs the main computation desired by the task.
    - The ForkJoinTask class provides several methods for checking the execution status of a task. The isDone() method returns true if a task completes in any way. The isCompletedNormally() method returns true if a task completes without cancellation or encountering an exception, and isCancelled() returns true if the task was cancelled. Lastly, isCompletedabnormally() returns true if the task was either cancelled or encountered an exception.

## Example

* + - In this example, you will learn how to use the asynchronous methods provided by the ForkJoinPool and ForkJoinTask classes for the management of tasks.
    - You are going to implement a program that will search for files with a determined extension inside a folder and its subfolders.
    - The ForkJoinTask class you’re going to implement will process the content of a folder. For each subfolder inside that folder, it will send a new task to the ForkJoinPool class in an asynchronous way. For each file inside that folder, the task will check the extension of the file and add it to the result list if it proceeds.

FolderProcessor.java

package forkJoinDemoAsyncExample;

import java.io.File;

import java.util.ArrayList;

import java.util.List;

import java.util.concurrent.RecursiveTask;

public class FolderProcessor extends RecursiveTask<List<String>> {

private static final long serialVersionUID = 1L; //This attribute will store the full path of the folder this task is going to process.

private final String path; //This attribute will store the name of the extension of the files this task is going to look for.

private final String extension;

//Implement the constructor of the class to initialize its attributes

public FolderProcessor(String path, String extension) {

this.path = path;

this.extension = extension;

}

//Implement the compute() method. As you parameterized the RecursiveTask class with the List<String> type, this method has to return an object of that type.

@Override

protected List<String> compute() {

//List to store the names of the files stored in the folder.

List<String> list = new ArrayList<String>();

//FolderProcessor tasks to store the subtasks that are going to process the subfolders stored in the folder

List<FolderProcessor> tasks = new ArrayList<FolderProcessor>();

//Get the content of the folder.

File file = new File(path);

File content[] = file.listFiles();

//For each element in the folder, if there is a subfolder, create a new FolderProcessor object

//and execute it asynchronously using the fork() method.

if (content != null) {

for (int i = 0; i < content.length; i++) {

if (content[i].isDirectory()) {

FolderProcessor task = new FolderProcessor(content[i].getAbsolutePath(), extension);

task.fork();

tasks.add(task);

}

//Otherwise, compare the extension of the file with the extension you are looking for using the checkFile() method

//and, if they are equal, store the full path of the file in the list of strings declared earlier.

else {

if (checkFile(content[i].getName())) {

list.add(content[i].getAbsolutePath());

}

}

}

}

//If the list of the FolderProcessor subtasks has more than 50 elements, rite a message to the console to indicate this circumstance.

if (tasks.size() > 50) {

System.out.printf("%s: %d tasks ran.\n", file.getAbsolutePath(), tasks.size());

}

//add to the list of files the results returned by the subtasks launched by this task.

addResultsFromTasks(list, tasks);

//Return the list of strings

return list;

}

//For each task stored in the list of tasks, call the join() method that will wait for its finalization and then will return the result of the task.

//Add that result to the list of strings using the addAll() method.

private void addResultsFromTasks(List<String> list, List<FolderProcessor> tasks) {

for (FolderProcessor item : tasks) {

list.addAll(item.join());

}

}

//This method compares if the name of a file passed as a parameter ends with the extension you are looking for.

private boolean checkFile(String name) {

return name.endsWith(extension);

}

}

Main.java

package forkJoinDemoAsyncExample;

import java.util.List;

import java.util.concurrent.ForkJoinPool;

import java.util.concurrent.TimeUnit;

public class Main {

public static void main(String[] args) {

//Create ForkJoinPool using the default constructor.

ForkJoinPool pool = new ForkJoinPool();

//Create three FolderProcessor tasks. Initialize each one with a different folder path.

FolderProcessor system = new FolderProcessor("C:\\Windows", "log");

FolderProcessor apps = new FolderProcessor("C:\\Program Files", "log");

FolderProcessor documents = new FolderProcessor("C:\\Documents And Settings", "log");

//Execute the three tasks in the pool using the execute() method.

pool.execute(system);

pool.execute(apps);

pool.execute(documents);

//Write to the console information about the status of the pool every second

//until the three tasks have finished their execution.

do {

System.out.printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

System.out.printf("Main: Parallelism: %d\n", pool.getParallelism());

System.out.printf("Main: Active Threads: %d\n", pool.getActiveThreadCount());

System.out.printf("Main: Task Count: %d\n", pool.getQueuedTaskCount());

System.out.printf("Main: Steal Count: %d\n", pool.getStealCount());

System.out.printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

try {

TimeUnit.SECONDS.sleep(1);

} catch (InterruptedException e) {

e.printStackTrace();

}

} while ((!system.isDone()) || (!apps.isDone()) || (!documents.isDone()));

//Shut down ForkJoinPool using the shutdown() method.

pool.shutdown();

//Write the number of results generated by each task to the console.

List<String> results;

results = system.join();

System.out.printf("System: %d files found.\n", results.size());

results = apps.join();

System.out.printf("Apps: %d files found.\n", results.size());

results = documents.join();

System.out.printf("Documents: %d files found.\n", results.size());

}

}

Output of above program will look like this:

Main: Parallelism: 2

Main: Active Threads: 3

Main: Task Count: 1403

Main: Steal Count: 5551

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Main: Parallelism: 2

Main: Active Threads: 3

Main: Task Count: 586

Main: Steal Count: 5551

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

System: 337 files found.

Apps: 10 files found.

Documents: 0 files found.

## How it works?

* + - In the FolderProcessor class, Each task processes the content of a folder. As you know, this content has the following two kinds of elements:
      * Files
      * Other folders
    - If the task finds a folder, it creates another Task object to process that folder and sends it to the pool using the fork() method. This method sends the task to the pool that will execute it if it has a free worker-thread or it can create a new one. The method returns immediately, so the task can continue processing the content of the folder. For every file, a task compares its extension with the one it’s looking for and, if they are equal, adds the name of the file to the list of results.
    - Once the task has processed all the content of the assigned folder, it waits for the finalization of all the tasks it sent to the pool using the join() method. This method called in a task waits for the finalization of its execution and returns the value returned by the compute() method. The task groups the results of all the tasks it sent with its own results and returns that list as a return value of the compute() method.

## Difference between Fork/Join Framework And ExecutorService

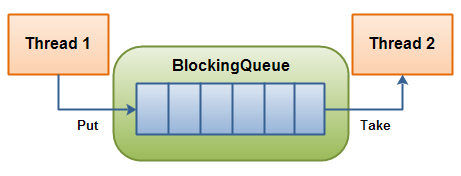
* + - The main difference between the Fork/Join and the Executor frameworks is the work-stealing algorithm. Unlike the Executor framework, when a task is waiting for the finalization of the sub-tasks it has created using the join operation, the thread that is executing that task (called worker thread ) looks for other tasks that have not been executed yet and begins its execution. By this way, the threads take full advantage of their running time, thereby improving the performance of the application.

## Existing Implementations in JDK

* + - There are some generally useful features in Java SE which are already implemented using the fork/join framework.
      * One such implementation, introduced in Java SE 8, is used by the java.util.Arrays class for its parallelSort() methods. These methods are similar to sort(), but leverage concurrency via the fork/join framework. Parallel sorting of large arrays is faster than sequential sorting when run on multiprocessor systems.
      * Parallelism used in Stream.parallel(). Read more about this parallel stream operation in java 8.

1. **Explain Blocking queue in Java?**

* BlockingQueue in Java is added in Java 1.5 along with various other concurrent Utility classes like ConcurrentHashMap, Counting Semaphore, CopyOnWriteArrrayList etc.
* BlockingQueue is a unique collection type which not only store elements but also supports flow control by introducing blocking if either BlockingQueue is full or empty. take() method of BlockingQueue will block if Queue is empty and put() method of BlockingQueue will block if Queue is full.
* This property makes BlockingQueue an ideal choice for implementing Producer consumer design pattern where one thread insert element into BlockingQueue and other thread consumes it.



* BlockingQueue in Java ***doesn't allow null elements***, various implementation of BlockingQueue like ***ArrayBlockingQueue***, ***LinkedBlockingQueue*** throws NullPointerException when you try to add null on queue.

*BlockingQueue<String> bQueue = new ArrayBlockingQueue<String>(10);  
//bQueue.put(null); //NullPointerException - BlockingQueue in Java doesn't allow null        
bQueue = new LinkedBlockingQueue<String>();  
bQueue.put(null);  
Exception in thread "main" java.lang.NullPointerException  
        at java.util.concurrent.LinkedBlockingQueue.put(LinkedBlockingQueue.java:288)*

* BlockingQueue can be bounded or unbounded. A bounded BlockingQueue is one which is initialized with initial capacity and call to put() will be blocked if BlockingQueue is full and size is equal to capacity. This bounding nature makes it ideal to use a shared queue between multiple threads like in most common Producer consumer solutions in Java. An unbounded Queue is one which is initialized without capacity, actually by default it initialized with Integer.MAX\_VALUE. Most common example of BlockingQueue uses bounded BlockingQueue as shown in below example.

*BlockingQueue<String> bQueue = new ArrayBlockingQueue<String>(2);  
bQueue.put("Java");  
System.out.println("Item 1 inserted into BlockingQueue");  
bQueue.put("JDK");  
System.out.println("Item 2 is inserted on BlockingQueue");  
bQueue.put("J2SE");  
System.out.println("Done");  
  
Output:  
Item 1 inserted into BlockingQueue  
Item 2 is inserted on BlockingQueue*

* BlockingQueue implementations like ***ArrayBlockingQueue***, ***LinkedBlockingQueue*** and ***PriorityBlockingQueue*** are thread-safe. All queuing method uses concurrency control and internal locks to perform operation atomically. Since BlockingQueue also extend Collection, bulk Collection operations like addAll(), containsAll() are not performed atomically until any BlockingQueue implementation specifically supports it. So call to addAll() may fail after inserting couple of elements.
* Common methods of BlockingQueue is are put() and take() which are blocking methods in Java and used to insert and retrieve elements from BlockingQueue in Java. put() will block if BlockingQueue is full and take() will block if BlockingQueue is empty, call to take() removes element from head of Queue
* BlockingQueue interface extends Collection, Queue and Iterable interface which provides it all Collection and Queue related methods like poll(), and peak(), unlike take(), peak() method returns head of the queue without removing it, poll() also retrieves and removes elements from head but can wait till specified time if Queue is empty.
* Other important methods from BlockingQueue in Java is remainingCapacity() and offer(), former returns number remaining space in BlockingQueue, which can be filled without blocking while later insert object into queue if possible and return true if success and false if fail unlike add() method which throws IllegalStateException if it fails to insert object into BlockingQueue. Use offer() over add() wherever possible.

import java.util.concurrent.BlockingQueue;

public class Producer implements Runnable{

BlockingQueue<Object> queue = null;

Producer(BlockingQueue<Object> theQueue) {

this.queue = theQueue;

}

@Override

public void run() {

while (true) {

try {

Object obj = getResource();

queue.put(obj);

System.out.println("Produced Resource - Queue size now : " + queue.size());

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

public Object getResource() {

try {

Thread.sleep(100);

} catch (InterruptedException e) {

System.out.println("Thread Interrupted");

}

return new Object();

}

}

import java.util.concurrent.BlockingQueue;

public class Consumer implements Runnable{

BlockingQueue<Object> queue = null;

public Consumer(BlockingQueue<Object> theQueue) {

this.queue = theQueue;

}

@Override

public void run() {

try {

Object obj = queue.take();

System.out.println("Consumed Object - Queue size now : " + queue.size());

take(obj);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

public static void take(Object obj) {

try {

Thread.sleep(100);

} catch (InterruptedException e) {

System.out.println("Thread Interrupted");

}

System.out.println("Consumed Resource : " + obj);

}

}

import java.util.concurrent.BlockingQueue;

import java.util.concurrent.LinkedBlockingQueue;

public class ProducerConsumerExample {

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

BlockingQueue<Object> queue = new LinkedBlockingQueue<>(20);

int numProducer = 4;

int numConsumer = 4;

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

new Thread(new Producer(queue)).start();;

}

for (int j=0; j<numConsumer; j++) {

new Thread(new Consumer(queue)).start();

}

Thread.sleep(10\*100);

System.exit(0);

}

}

# **Implementations of BlockingQueue:**

* [**ArrayBlockingQueue**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ArrayBlockingQueue.html)
  + A bounded blocking queue backed by an array. This queue orders elements FIFO (first-in-first-out). The head of the queue is that element 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.
  + This is a classic "bounded buffer", in which a fixed-sized array holds elements inserted by producers and extracted by consumers. Once created, the capacity cannot be changed. Attempts to put an element into a full queue will result in the operation blocking; attempts to take an element from an empty queue will similarly block.
  + This class supports an optional fairness policy for ordering waiting producer and consumer threads. By default, this ordering is not guaranteed. However, a queue constructed with fairness set to true grants threads access in FIFO order. Fairness generally decreases throughput but reduces variability and avoids starvation.
* [**DelayQueue**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/DelayQueue.html)
  + An unbounded [blocking queue](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html) of Delayed elements, in which an element can only be taken when its delay has expired. The *head* of the queue is that Delayed element whose delay expired furthest in the past. If no delay has expired there is no head and poll will return null. Expiration occurs when an element's getDelay(TimeUnit.NANOSECONDS) method returns a value less than or equal to zero. Even though unexpired elements cannot be removed using take or poll, they are otherwise treated as normal elements. For example, the size method returns the count of both expired and unexpired elements. This queue does not permit null elements.
* [**LinkedBlockingDeque**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/LinkedBlockingDeque.html)
  + An optionally-bounded blocking deque based on linked nodes.
  + The optional capacity bound constructor argument serves as a way to prevent excessive expansion. The capacity, if unspecified, is equal to Integer.MAX\_VALUE. Linked nodes are dynamically created upon each insertion unless this would bring the deque above capacity.
  + Most operations run in constant time (ignoring time spent blocking). Exceptions include remove, removeFirstOccurrence, removeLastOccurrence, contains, iterator.remove(), and the bulk operations, all of which run in linear time.
* [**LinkedBlockingQueue**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/LinkedBlockingQueue.html)
  + An optionally-bounded blocking queue based on linked nodes. This queue orders elements FIFO (first-in-first-out). The head of the queue is that element 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. Linked queues typically have higher throughput than array-based queues but less predictable performance in most concurrent applications.
  + The optional capacity bound constructor argument serves as a way to prevent excessive queue expansion. The capacity, if unspecified, is equal to Integer.MAX\_VALUE. Linked nodes are dynamically created upon each insertion unless this would bring the queue above capacity.
* [**LinkedTransferQueue**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/LinkedTransferQueue.html)
* [**PriorityBlockingQueue**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/PriorityBlockingQueue.html)
* [**SynchronousQueue**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/SynchronousQueue.html)
  + A blocking queue in which each insert operation must wait for a corresponding remove operation by another thread, and vice versa. A synchronous queue does not have any internal capacity, not even a capacity of one. You cannot peek at a synchronous queue because an element is only present when you try to remove it; you cannot insert an element (using any method) unless another thread is trying to remove it; you cannot iterate as there is nothing to iterate. The head of the queue is the element that the first queued inserting thread is trying to add to the queue; if there is no such queued thread then no element is available for removal and poll() will return null. For purposes of other Collection methods (for example contains), a SynchronousQueue acts as an empty collection. This queue does not permit null elements.

# **ArrayBlockingQueue:**

* ArrayBlockingQueue is a BlockingQueue implementation with the following characteristics:
* **Internal data structure:**
  + it is based on a circular array to store elements. The first and the last elements of the array are treated logically adjacent, hence circular array.
  + The queue advances the indices of the head and tail elements whenever an element is added/removed from the queue. When either index advances the last element in the array, it is restarted from 0.
  + This mechanism is better than using a regular array, as the queue doesn’t have to shift all the elements whenever the head is removed. However, if you remove an element in the middle (via Iterator.remove), it must shift the elements.
* **Capacity:**
  + There’s a limit on the number of elements an ArrayBlockingQueue can hold, as you must specify a capacity when constructing a new object of this class. Hence ArrayBlockingQueueis a bounded BlockingQueue.
* **Order:**
  + since it is based on an array, elements are ordered by FIFO (First-In First-Out).
* **Operations:**
  + queue insertion and removal are executed in constant time (very fast).
* **Iterator:**
  + the iterators returned by are ArrayBlockingQueue weakly consistent which means the iterator may be used concurrently with other operations. It will never throw ConcurrentModificationException and support remove operation.
* You also need to pay attention when creating a new object of ArrayBlockingQueueas it provides 3 constructors:

ArrayBlockingQueue(int capacity)

ArrayBlockingQueue(int capacity, boolean fair)

ArrayBlockingQueue(int capacity, boolean fair, Collection<? extends E> c)

* Notice the parameter fair specifies the access policy for threads blocked on insertion or removal. Consider this scenario: several threads blocked because the queue is either full or empty, so when space becomes available or elements available, which thread is granted access first? which is granted next?
* So ArrayBlockingQueueallows you to choose one of two policies according to the value of the parameter fair:
  + if fair is set to true:
    - threads are granted access in FIFO order, which means the longest waiting thread will have access first.
  + if fair is set to false:
    - the access order is unspecified.
* By default (as using the first constructor), the access order is unspecified.
* Now, let’s see how to use the ArrayBlockingQueuein details with code examples.

# Creating a new ArrayBlockingQueue

* The following line creates an ArrayBlockingQueueobject with a fixed capacity of 100 elements of type String:

|  |  |
| --- | --- |
|  | BlockingQueue<String> queue1 = new ArrayBlockingQueue<>(100); |

* This line creates an ArrayBlockingQueue object with a fixed capacity of 50 Integer elements and the access policy is fair:

|  |  |
| --- | --- |
|  | BlockingQueue<Integer> queue2 = new ArrayBlockingQueue<>(50, true); |

* And the following line creates an ArrayBlockingQueue object with a fixed capacity, fair access policy and initially elements from a given collection:

|  |  |
| --- | --- |
|  | List<String> list = Arrays.asList("One", "Two", "Three");  BlockingQueue<String> queue3 = new ArrayBlockingQueue<>(50, true, list); |

# Inserting an element to the tail of the queue

* The following code snippet uses the put() method to insert an element to the tail of the queue:

|  |  |
| --- | --- |
|  | BlockingQueue<String> queue = new ArrayBlockingQueue<>(100);  try {      queue.put("Four");  } catch (InterruptedException ie) {      ie.printStackTrace();  } |

* Remember the put() method will block if the queue is full, waiting for space becomes available. And while waiting, it will throw InterruptedException if the current thread is interrupted.
* In case you want to wait only a specified amount of time when the queue is full, use the offer(element, timeout, timeunit) method. For example:

|  |  |
| --- | --- |
|  | try {      long timeout = 1000;      boolean success = queue.offer("Five", timeout, TimeUnit.MILLISECONDS);        if (!success) {          System.out.println("Queue is full and timeout elapsed");      }  } catch (InterruptedException ie) {      ie.printStackTrace();  } |

* In this code, the offer()method will return false if the queue is full and the current thread has been waiting for more than the specified time.

# Retrieving and removing an element from the head of the queue

* The following code snippet uses the take() method to retrieve and remove an element at the head of the queue, waiting if necessary until an element becomes available in case the queue is empty:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | try {      String head = queue.take();      System.out.println("Head element: " + head);    } catch (InterruptedException ie) {      ie.printStackTrace();  } |

* And use the poll(timeout, timeunit) method if you want to wait up to a specified time if the queue is empty, for an element to become available. Here’s an example:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | try {      long timeout = 1000;      String head = queue.poll(timeout, TimeUnit.MILLISECONDS);      System.out.println("Head element: " + head);    } catch (InterruptedException ie) {      ie.printStackTrace();  } |

* Note that the poll(timeout, timeunit) method will return null if the timeout expires.

# Draining the queue

* When the queue is no longer used with insertion and removal operations, and if you want to get all the remaining elements in the queue, use the drainTo(collection) method that removes all available elements in the queue and adds them to the specified collection. Here’s an example:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | BlockingQueue<String> queue = new ArrayBlockingQueue<>(100);    //...  // insertion and removal operations by threads  //...    List<String> list = new ArrayList<>();    queue.drainTo(list); |

* Using this method is more efficient than repeated polling the queue.

# Java ArrayBlockingQueue Methods

* ArrayBlockingQueue class has below given important methods, you should know.
* **void put(Object o)** : Inserts the specified element at the tail of this queue, waiting for space to become available if the queue is full.**boolean add(object)** : Inserts the specified element at the tail of this queue if it is possible to do so immediately without exceeding the queue’s capacity, returning true upon success and throwing an IllegalStateException if this queue is full.
* **boolean offer(object)** : Inserts the specified element at the tail of this queue if it is possible to do so immediately without exceeding the queue’s capacity, returning true upon success and throwing an IllegalStateException if this queue is full.
* **boolean remove(object)** : Removes a single instance of the specified element from this queue, if it is present.
* **Object peek()** : Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.
* **Object poll()** : Retrieves and removes the head of this queue, or returns null if this queue is empty.
* **Object poll(timeout, timeUnit)** : Retrieves and removes the head of this queue, waiting up to the specified wait time if necessary for an element to become available.
* **Object take()** : Retrieves and removes the head of this queue, waiting if necessary until an element becomes available.
* **void clear()** : Removes all of the elements from this the queue.
* **boolean contains(Object o)** : Returns true if this queue contains the specified element.
* **Iterator iterator()** : Returns an iterator over the elements in this queue in proper sequence.
* **int size()** : Returns the number of elements in this queue.
* **int drainTo(Collection c)** : Removes all available elements from this queue and adds them to the given collection.
* **int drainTo(Collection c, int maxElements)** : Removes at most the given number of available elements from this queue and adds them to the given collection.
* **int remainingCapacity()** : Returns the number of additional elements that this queue can ideally (in the absence of memory or resource constraints) accept without blocking.
* **Object[] toArray()** : Returns an array containing all of the elements in this queue, in proper sequence.

# A Producer - Consumer Example using ArrayBlockingQueue

* Let’s see a full example that makes use of ArrayBlockingQueue in a file text search program that searches all text files in a directory for a given keyword. The program will print which file contains the keyword and at which line.
* The producer class is a thread that recursively lists the content of a given directory for files that match the given extension, and then it puts each file on the queue. The following code is of the producer class:

|  |
| --- |
| import java.io.\*;  import java.util.\*;  import java.util.concurrent.\*;    /\*\*   \* This producer enumerates files of a specific type in a given directory   \* and then put the files on the queue.   \*   \* @author www.codejava.net   \*/  public class DirectoryLister extends Thread {      private BlockingQueue<File> queue;      private File directory;      private String extension;        public DirectoryLister(BlockingQueue<File> queue,              File directory, String extension) {          this.queue = queue;          this.directory = directory;          this.extension = extension;      }        public void run() {            try {                listDirectory(directory);              queue.put(new File("END"));            } catch (InterruptedException ie) {              ie.printStackTrace();          }      }        private void listDirectory(File dir) throws InterruptedException {          File[] files = dir.listFiles(new FileFilter() {              public boolean accept(File file) {                  return file.getName().endsWith(extension);              }          });            if (files != null && files.length > 0) {              for (File aFile : files) {                  if (aFile.isDirectory()) {                      listDirectory(aFile);                  } else {                      queue.put(aFile);                  }              }          }      }  } |

You can notice that the producer, after reading directory’s content, put a special element on the queue:

|  |  |
| --- | --- |
| 1 | queue.put(new File("END")); |

This acts as a “signal” element that tells the consumer that there’s no more elements added to the queue. You can see how the consumer class processes elements from the queue in the following code:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43 | import java.io.\*;  import java.nio.file.\*;  import java.util.\*;  import java.util.concurrent.\*;    /\*\*   \* This consumer takes a File from the queue, and parses the file   \* to search for the given keyword.   \*   \* @author www.codejava.net   \*/  public class FileParser extends Thread {      private BlockingQueue<File> queue;      private String keyword;        public FileParser(BlockingQueue<File> queue, String keyword) {          this.queue = queue;          this.keyword = keyword;      }        public void run() {          while (true) {              try {                  File file = queue.peek();                    if (file != null && !file.getName().equals("END")) {                      file = queue.take();                      parseFile(file);                  } else {                      break;                  }              } catch (InterruptedException | IOException ex) {                  ex.printStackTrace();              }          }      }        private void parseFile(File file) throws IOException {          List<String> lines = Files.readAllLines(file.toPath());          int lineCount = 0;            for (String aLine : lines) {              lineCount++;              if (aLine.contains(keyword)) {                  String result = "Found in %s at line %d\n";                  System.out.printf(result, file.getAbsolutePath(), lineCount);                  break;              }          }        }  } |

First, it examines the head of the queue:

|  |  |
| --- | --- |
| 1 | File file = queue.peek(); |

If the head is not the “signal element”, it removes the head from the queue and parses the file:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | if (file != null && !file.getName().equals("END")) {      file = queue.take();      parseFile(file);  } else {      break;  } |

If the head is the “signal element”, exits the while loop and the thread terminates. Note that we must keep the “signal element” always at the head of the queue so that other consumers can examine and act accordingly, hence the peek()method is used first, then take() is used if the element is actually a file.

And here’s code of the main program:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import java.io.\*;  import java.util.concurrent.\*;    /\*\*   \* This program demonstrates using ArrayBlockingQueue   \* in a producer-consumer application.   \*   \* @author www.codejava.net   \*/  public class FileTextSearch {      public static void main(String[] args) {          String dirPath = args[0];          String extension = args[1];          String keyword = args[2];            BlockingQueue<File> queue = new ArrayBlockingQueue<>(100);            DirectoryLister lister = new DirectoryLister(queue, new File(dirPath), extension);          lister.start();            for (int i = 0; i < 10; i++) {              FileParser parser = new FileParser(queue, keyword);              parser.start();          }      }  } |

As you can see, the program creates one producer thread and 10 consumer threads, sharing an ArrayBlockingQueuewhich contains a maximum of 100 elements. Run this program from command like this:

|  |  |
| --- | --- |
| 1 | java FileTextSearch “D:\JDKSource\java\util\concurrent” “.java” “BlockingQueue” |

The program will search all .java files in the D:\JDKSource\java\util\concurrent directory to find the files that contain the text “BlockingQueue”. The output looks something like this:

|  |  |
| --- | --- |
| 1  2  3 | Found in D:\JDKSource\java\util\concurrent\ArrayBlockingQueue.java at line 82  Found in D:\JDKSource\java\util\concurrent\package-info.java at line 114  Found in D:\JDKSource\java\util\concurrent\ThreadPoolExecutor.java at line 183 |

As you can see, an ArrayBlockingQueueis shared among multiple threads but we don’t have to write any code for synchronization or locking, as the ArrayBlockingQueue handles concurrent threads itself.

Now, try to run this program on your computer with different inputs and check the results

1. **Explain Coundownlatch 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.

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 it’s not reusable once count reaches to zero you cannot use CountDownLatch anymore, but don't worry Java concurrency API has another concurrent utility called CyclicBarrier for such requirements.

## **Example**

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;

/\*\*

\* Java program to demonstrate How to use CountDownLatch in Java. CountDownLatch is

\* useful if you want to start main processing thread once its dependency is completed

\* as illustrated in this CountDownLatch Example

\*

\*/

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.

1. **Explain CyclicBarrier in Java?**

CyclicBarrier in Java is a synchronizer introduced in JDK 5 on java.util.Concurrent package along with other concurrent utility like Counting Semaphore, BlockingQueue, ConcurrentHashMap etc.

CyclicBarrier is similar to CountDownLatch which allows multiple threads to wait for each other (barrier) before proceeding.

CyclicBarrier is a natural requirement for a concurrent program because it can be used to perform final part of the task once individual tasks are completed.

All threads which wait for each other to reach barrier are called parties, CyclicBarrier is initialized with a number of parties to 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 the barrier.

await() is a blocking call but can be timed out or Interrupted by other thread.

If you look at CyclicBarrier, it also the does the same thing but there is different you cannot reuse CountDownLatch once the count reaches zero while you can reuse CyclicBarrier by calling reset () method which resets Barrier to its initial State.

What it implies that CountDownLatch is a good for one-time events like application start-up time and CyclicBarrier can be used to in case of the recurrent event e.g. concurrently calculating a solution of the big problem etc.

## **Example**

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 the barrier, once all thread reach the barrier, barrier gets broker and each thread started their execution from that point.

It’s 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;  
  
/\*\*  
 \* Java program to demonstrate how to use CyclicBarrier in Java. CyclicBarrier is a

 \* new Concurrency Utility added in Java 5 Concurrent package.

 \*  
 \*/  
**public** **class** CyclicBarrierExample {  
  
    *//Runnable task for each thread*  
    **private** **static** **class** Task **implements** [**Runnable**](http://javarevisited.blogspot.sg/2012/01/difference-thread-vs-runnable-interface.html) {  
  
        **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 have 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 broken 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 count population from North, South, East, and West and once complete they can wait for each other, When last thread completed their 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 cannot 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 the 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. The thread will block on await() until all parties reach to the barrier, another thread interrupt or await timed out.
4. If another thread interrupts 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)

1. 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.
2. **What’s volatile variable in Java?**

The volatile keyword in Java is used as an indicator to Java compiler and Thread that do not cache value of this variable and always read it from main memory.

So if you want to share any variable in which read and write operation is atomic by implementation e.g. read and write in an int or a boolean variable then you can declare them as volatile variable.

The Java volatile keyword cannot be used with method or class and it can only be used with a variable.

Java volatile keyword also guarantees visibility and ordering, after Java 5 write to any volatile variable happens before any read into the volatile variable. By the way use of volatile keyword also prevents compiler or JVM from the reordering of code or moving away them from synchronization barrier.

## **The Volatile variable Example**

To Understand example of volatile keyword in java let’s go back to Singleton pattern in Java and see double checked locking in Singleton with Volatile and without the volatile keyword in java.

**public** **class** **Singleton**{

**private** **static** **volatile** Singleton \_instance; //volatile variable

**public** **static** Singleton **getInstance**(){

**if**(\_instance == **null**){

**synchronized**(Singleton.class){

**if**(\_instance == **null**)

\_instance = **new** Singleton();

}

}

**return** \_instance;

}

If you look at the code carefully you will be able to figure out:

* We are only creating instance one time
* We are creating instance lazily at the time of the first request comes.

If we do not make the \_instance variable volatile than the Thread which is creating instance of Singleton is not able to communicate other thread, that instance has been created until it comes out of the Singleton block, so if Thread A is creating Singleton instance and just after creation lost the CPU, all other thread will not be able to see value of \_instance as not null and they will believe its still null.

So in Summary apart from synchronized keyword in Java, volatile keyword is also used to communicate the content of memory between threads.

## **When to use Volatile variable in Java**

You can use Volatile variable if you want to read and write long and double variable atomically. long and double both are 64 bit data type and by default writing of long and double is not atomic and platform dependence. Many platform perform write in long and double variable 2 step, writing 32 bit in each step, due to this its possible for a Thread to see 32 bit from two different write. You can avoid this issue by making long and double variable volatile in Java.

A volatile variable can be used as an alternative way of achieving synchronization in Java in some cases, like Visibility. With volatile variable, it's guaranteed that all reader thread will see updated value of the volatile variable once write operation completed, without volatile keyword different reader thread may see different values.

volatile variable can be used to inform the compiler that a particular field is subject to be accessed by multiple threads, which will prevent the compiler from doing any reordering or any kind of optimization which is not desirable in a multi-threaded environment. Without volatile variable compiler can re-order the code, free to cache value of volatile variable instead of always reading from main memory.

Another place where a volatile variable can be used is to fixing double checked locking in Singleton pattern. As we discussed in Why should you use Enum as Singleton that double checked locking was broken in Java 1.4 environment.

1. **Difference between synchronized and volatile keyword in Java?**
   * + The volatile keyword in Java is a field modifier while synchronized modifies code blocks and methods.
     + Synchronized obtains and releases the lock on monitor’s Java volatile keyword doesn't require that.
     + Threads in Java can be blocked for waiting for any monitor in case of synchronized, that is not the case with the volatile keyword in Java.
     + Synchronized method affects performance more than a volatile keyword in Java.
     + Since volatile keyword in Java only synchronizes the value of one variable between Thread memory and "main" memory while synchronized synchronizes the value of all variable between thread memory and "main" memory and locks and releases a monitor to boot. Due to this reason synchronized keyword in Java is likely to have more overhead than volatile.
     + You cannot synchronize on the null object but your volatile variable in Java could be null.
     + From Java 5 writing into a volatile field has the same memory effect as a monitor release, and reading from a volatile field has the same memory effect as a monitor acquire
2. **What’s Reentrant Lock in Java?**

In Java 5.0, a new addition called Reentrant Lock was made to enhance intrinsic locking capabilities. Prior to this, "synchronized" and "volatile" were the means for achieving concurrency.

public synchronized void doAtomicTransfer(){

//enter synchronized block , acquire lock over this object.

operation1()

operation2();

} // exiting synchronized block, release lock over this object.

Synchronized uses intrinsic locks or monitors. Every object in Java has an intrinsic lock associated with it. Whenever a thread tries to access a synchronized block or method, it acquires the intrinsic lock or the monitor on that object.

In case of static methods, the thread acquires the lock over the class object. An intrinsic locking mechanism is a clean approach in terms of writing code, and is pretty good for most of the use-cases.

An intrinsic locking mechanism can have some functional limitations, such as:

* It is not possible to interrupt a thread waiting to acquire a lock (lock Interruptibly).
* It is not possible to attempt to acquire a lock without being willing to wait for it forever (try lock).
* Cannot implement non-block-structured locking disciplines, as intrinsic locks must be released in the same block in which they are acquired.

Lets see a few of the methods implemented by ReentrantLock class (which implements Lock):

void lock();

void lockInterruptibly() throws InterruptedException;

boolean tryLock();

boolean tryLock(long time, TimeUnit unit) throws InterruptedException;

.....

Lets try and understand the use of these and see what benefits we can get.

## **Polled and Timed Lock Acquisition**

Let's see some example code:

public void transferMoneyWithSync(Account fromAccount, Account toAccount,

float amount) throws InsufficientAmountException {

synchronized (fromAccount) {

// acquired lock on fromAccount Object

synchronized (toAccount) {

// acquired lock on toAccount Object

if (amount > fromAccount.getCurrentAmount()) {

throw new InsufficientAmountException(

"Insufficient Balance");

} else {

fromAccount.debit(amount);

toAccount.credit(amount);

}

}

}

}

In the transferMoney() method above, there is a possibility of deadlock when two threads A and B are trying to transfer money at almost the same time.

A: transferMoney(acc1, acc2, 20);

B: transferMoney(acc2, acc1 ,25);

It is possible that thread A has acquired a lock on the acc1 object and is waiting to acquire a lock on the acc2 object. Meanwhile, thread B has acquired a lock on the acc2 object and is waiting for a lock on acc1. This will lead to deadlock, and the system would have to be restarted! There is, however, a way to avoid this, which is called "lock ordering." Personally, I find this a bit complex.

A cleaner approach is implemented by ReentrantLock with the use of tryLock() method. This approach is called the "timed and polled lock-acquisition." It lets you regain control if you cannot acquire all the required locks, release the ones you have acquired and retry. So, using tryLock, we will attempt to acquire both locks. If we cannot attain both, we will release if one of these has been acquired, then retry.

public boolean transferMoneyWithTryLock(Account fromAccount,

Account toAccount, float amount) throws InsufficientAmountException, InterruptedException {

// we are defining a stopTime

long stopTime = System.nanoTime() + 5000;

while (true) {

if (fromAccount.lock.tryLock()) {

try {

if (toAccount.lock.tryLock()) {

try {

if (amount > fromAccount.getCurrentAmount()) {

throw new InsufficientAmountException(

"Insufficient Balance");

} else {

fromAccount.debit(amount);

toAccount.credit(amount);

}

} finally {

toAccount.lock.unlock();

}

}

} finally {

fromAccount.lock.unlock();

}

}

if(System.nanoTime() < stopTime)

return false;

Thread.sleep(100);

}//while

}

Here we implemented a timed lock, so if the locks cannot be acquired within the specified time, the transferMoney method will return a failure notice and exit gracefully. We can also maintain time budget activities using this concept.

## **Interruptible Lock Acquisition**

Interruptible lock acquisition allows locking to be used within cancellable activities.

The lockInterruptibly() method allows us to try and acquire a lock while being available for interruption. So, basically it allows the thread to immediately react to the interrupt signal sent to it from another thread.

This can be helpful when we want to send a KILL signal to all the waiting locks.

Let's see one example: Suppose we have a shared line to send messages. We would want to design it in such a way that if another thread comes and interrupts the current thread, the lock should release and perform the exit or shut down operations to cancel the current task.

public boolean sendOnSharedLine(String message) throws InterruptedException{

lock.lockInterruptibly();

try{

return cancellableSendOnSharedLine(message);

} finally {

lock.unlock();

}

}

private boolean cancellableSendOnSharedLine(String message){

.......

The timed tryLock is also responsive to interruption.

## **Non-block Structured Locking:**

In intrinsic locks, acquire-release pairs are block-structured.

In other words, a lock is always released in the same basic block in which it was acquired, regardless of how control exits the block.

Extrinsic locks allow the facility to have more explicit control. Some concepts, like Lock Strapping, can be achieved more easily using extrinsic locks. Some use cases are seen in hash-bashed collections and linked lists.

## **Fairness:**

The ReentrantLock constructor offers a choice of two fairness options: create a non-fair lock or a fair lock. With fair locking, threads can acquire locks only in the order in which they were requested, whereas an unfair lock allows a lock to acquire it out of its turn. This is called barging (breaking the queue and acquiring the lock when it became available).

Fair locking has a significant performance cost because of the overhead of suspending and resuming threads. There could be cases where there is a significant delay between when a suspended thread is resumed and when it actually runs. Let's see a situation:

A -> holds a lock.

B -> has requested and is in a suspended state waiting for A to release the lock.

C -> requests the lock at the same time that A releases the lock, and has not yet gone to a suspended state.

As C has not yet gone to a suspended state, there is a chance that it can acquire the lock released by A, use it, and release it before B even finishes waking up. So, in this context, unfair lock has a significant performance advantage.

Intrinsic locks and extrinsic locks have the same mechanism inside for locking, so the performance improvement is purely subjective. It depends on the use cases we discussed above. Extrinsic locks give a more explicit control mechanism for better handling of deadlocks, starvation, and so on.

import java.util.concurrent.locks.ReentrantLock;

import java.util.logging.Level;

import java.util.logging.Logger;

/\*\*

 \* Java program to show, how to use ReentrantLock in Java.

 \* Reentrant lock is an alternative way of locking

 \* apart from implicit locking provided by synchronized keyword in Java.

 \*/

public class ReentrantLockHowto {

    private final ReentrantLock lock = new ReentrantLock();

    private int count = 0;

     //Locking using Lock and ReentrantLock

     public int getCount() {

        lock.lock();

        try {

            System.out.println(Thread.currentThread().getName() + " gets Count: " + count);

            return count++;

        } finally {

            lock.unlock();

        }

     }

     //Implicit locking using synchronized keyword

     public synchronized int getCountTwo() {

            return count++;

     }

public static void main(String args[]) {

final ThreadTest counter = new ThreadTest();

Thread t1 = new Thread() {

@Override

public void run() {

while (counter.getCount() &lt; 6) {

try {

Thread.sleep(100);

} catch (InterruptedException ex) {

ex.printStackTrace(); }

}

}

};

Thread t2 = new Thread() {

@Override

public void run() {

while (counter.getCount() &lt; 6) {

try {

Thread.sleep(100);

} catch (InterruptedException ex) {

ex.printStackTrace();

}

}

}

};

t1.start();

t2.start();

public void run() {

while (counter.getCount() &lt; 6) {

try {

Thread.sleep(100);

} catch (InterruptedException ex) {

ex.printStackTrace();

}

}

}

};

t1.start();

t2.start();

}

}

Output:

Thread-0 gets Count: 0

Thread-1 gets Count: 1

Thread-1 gets Count: 2

Thread-0 gets Count: 3

Thread-1 gets Count: 4

Thread-0 gets Count: 5

Thread-0 gets Count: 6

Thread-1 gets Count: 7

1. **Explain Java Inter-thread Communication using Piped Streams?**

## **What are piped streams**

Pipe streams are just like real plumbing pipes. You put things into to a pipe at one end using some methods. Then you receive the same things back from the pipe stream at the other end using some other methods. They come out in FIFO order, first-in first-out, just like from real plumbing pipes.

## **PipedReader and PipedWriter**

PipedReader is an extension of Reader class which is used for reading character streams. Its read() method reads the connected PipedWriter’s stream. Similarly, PipedWriter is an extension of Writer class and does all the things which Reader class contracts.

A writer can be connected to a reader by following two methods:

* + - * Using constructor PipedWriter(PipedReader pr)
      * Using connect(PipedReader pr) method

Once connected through any of above ways, any thread can write data in stream using write(….) methods, and data will be available to reader and can be read using read() method.

## **Example**

Below given sample program creates two threads. One thread is responsible for writing into stream and second one is only reading the data to print them in console.

import java.io.\*;

public class PipedCommunicationTest {

public static void main(String[] args) {

new PipedCommunicationTest();

}

public PipedCommunicationTest() {

try {

// Create writer and reader instances

PipedReader pr = new PipedReader();

PipedWriter pw = new PipedWriter();

// Connect the writer with reader

pw.connect(pr);

// Create one writer thread and one reader thread

Thread thread1 = new Thread(

new PipeReaderThread("ReaderThread", pr));

Thread thread2 = new Thread(

new PipeWriterThread("WriterThread", pw));

// start both threads

thread1.start();

thread2.start();

} catch (Exception e) {

System.out.println("PipeThread Exception: " + e);

}

}

}

class PipeReaderThread implements Runnable {

PipedReader pr;

String name = null;

public PipeReaderThread(String name, PipedReader pr) {

this.name = name;

this.pr = pr;

}

public void run() {

try {

// continuously read data from stream and print it in console

while (true) {

char c = (char) pr.read(); // read a char

if (c != -1) { // check for -1 indicating end of file

System.out.print(c);

}

}

} catch (Exception e) {

System.out.println(" PipeThread Exception: " + e);

}

}

}

class PipeWriterThread implements Runnable {

PipedWriter pw;

String name = null;

public PipeWriterThread(String name, PipedWriter pw) {

this.name = name;

this.pw = pw;

}

public void run() {

try {

while (true) {

// Write some data after every two seconds

pw.write("Testing data written...n");

pw.flush();

Thread.sleep(2000);

}

} catch (Exception e) {

System.out.println(" PipeThread Exception: " + e);

}

}

}

Output:

Testing data written...

Testing data written...

Testing data written...

## **Important notes**

* + - You cannot write to a pipe without having some sort of reader created and connected to it. In other words, both ends must be present and already connected for the writing end to work.
    - You cannot switch to another reader, to which the pipe was not originally connected, once you are done writing to a pipe.
    - You cannot read back from the pipe if you close the reader. You can close the writing end successfully, however, and still read from the pipe.

You cannot read back from the pipe if the thread which wrote to it ends.

1. **Explain Java ThreadLocal Variables?**

Today, one of the most critical aspects of a concurrent application is shared data. When you create thread that implements the Runnable interface and then start various Thread objects using the same Runnable object, all the threads share the same attributes that are defined inside the runnable object.

This essentially means that if you change any attribute in a thread, all the threads will be affected by this change and will see the modified value by first thread.

Sometimes it is desired behavior e.g. multiple threads increasing / decreasing the same counter variable; but sometimes you want to ensure that every thread MUST work on its own copy of thread instance and does not affect others data.

## **When to use ThreadLocal?**

For example, consider you are working on an eCommerce application. You have a requirement to generate a unique transaction id for each and every customer request this controller process and you need to pass this transaction id to the business methods in manager/DAO classes for logging purpose. One solution could be passing this transaction id as a parameter to all the business methods. But this is not a good solution as the code is redundant and unnecessary.

To solve that, here you can use ThreadLocal variable. You can generate a transaction id in controller OR any pre-processor interceptor; and set this transaction id in the ThreadLocal. After this, whatever the methods, that this controller calls, they all can access this transaction id from the threadlocal. Also note that application controller will be servicing more that one request at a time and since each request is processed in separate thread at framework level, the transaction id will be unique to each thread and will be accessible from all over the thread’s execution path.

## **Inside ThreadLocal Class?**

The Java Concurrency API provides a clean mechanism for thread-local variables using ThreadLocal class with a very good performance.

public class ThreadLocal<T> extends Object {...}

This class provides thread-local variables. These variables differ from their normal counterparts in that each thread that accesses one (via its get or set method) has its own, independently initialized copy of the variable. ThreadLocal instances are typically private static fields in classes that wish to associate state with a thread (e.g., a user ID or Transaction ID).

This class has following methods:

* + - * get() : Returns the value in the current thread’s copy of this thread-local variable.
      * initialValue() : Returns the current thread’s “initial value” for this thread-local variable.
      * remove() : Removes the current thread’s value for this thread-local variable.
      * set(T value) : Sets the current thread’s copy of this thread-local variable to the specified value.

## **How to use ThreadLocal?**

Below example uses two thread local variables i.e. threadId and startDate. Both have been defined as “private static” fields as recommended. ‘threadId‘ will be used to identify the thread which is currently running and ‘startDate‘ will be used to get the time when thread started it’s execution. Above information will be printed in console to verify that each thread has maintained it’s own copy of variables.

**class** DemoTask **implements** Runnable {

// Atomic integer containing the next thread ID to be assigned

**private** **static** **final** AtomicInteger ***nextId*** = **new** AtomicInteger(0);

// Thread local variable containing each thread's ID

**private** **static** **final** ThreadLocal<Integer> ***threadId*** = **new** ThreadLocal<Integer>() {

@Override

**protected** Integer initialValue() {

**return** ***nextId***.getAndIncrement();

}

};

// Returns the current thread's unique ID, assigning it if necessary

**public** **int** getThreadId() {

**return** ***threadId***.get();

}

// Returns the current thread's starting timestamp

**private** **static** **final** ThreadLocal<Date> ***startDate*** = **new** ThreadLocal<Date>() {

**protected** Date initialValue() {

**return** **new** Date();

}

};

@Override

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

System.***out***.printf("Starting Thread: %s : %s\n", getThreadId(), ***startDate***.get());

**try** {

TimeUnit.SECONDS.sleep((**int**) Math.*rint*(Math.*random*() \* 10));

} **catch** (InterruptedException e) {

e.printStackTrace();

}

System.***out***.printf("Thread Finished: %s : %s\n", getThreadId(), ***startDate***.get());

}

}

Now to verify that variables essentially are able to maintain their state irrespective of multiple initializations for multiple threads, let’s create three instances of this task; start the threads; and then verify the information they print in console.

Starting Thread: 0 : Wed Dec 24 15:04:40 IST 2014

Thread Finished: 0 : Wed Dec 24 15:04:40 IST 2014

Starting Thread: 1 : Wed Dec 24 15:04:42 IST 2014

Thread Finished: 1 : Wed Dec 24 15:04:42 IST 2014

Starting Thread: 2 : Wed Dec 24 15:04:44 IST 2014

Thread Finished: 2 : Wed Dec 24 15:04:44 IST 2014

In above output, sequence of printed statement will vary every time. I have put them in sequence so that we can clearly identify that thread local values are kept safe for each thread instance; and never intermixed.

Most common use of thread local is when you have some object that is not thread-safe, but you want to avoid synchronizing access to that object using synchronized keyword/block. Instead, give each thread its own instance of the object to work with.

A good alternative to synchronization or threadlocal is to make the variable a local variable. Local variables are always thread safe. The only thing which may prevent you to do this is your application design constraints.

1. **Object level Locking vs. Class level Locking in Java**
   * + Synchronization refers to multi-threading. A synchronized block of code can only be executed by one thread at a time.
     + Java supports multiple threads to be executed. This may cause two or more threads to access the same fields or objects.
     + Synchronization is a process which keeps all concurrent threads execution to be in synch.
     + Synchronization avoids memory consistence errors caused due to inconsistent view of shared memory.
     + When a method is declared as synchronized; the thread holds the monitor for that method’s object, if another thread is executing the synchronized method, your thread is blocked until that thread releases the monitor.
     + Synchronization in java is achieved using synchronized keyword. You can use synchronized keyword in your class on defined methods or blocks. Keyword cannot be used with variables or attributes in class definition.
   * ***Object level locking***
     + Object level locking is mechanism when you want to synchronize a non-static method or non-static code block such that only one thread will be able to execute the code block on given instance of the class.
     + This should always be done to make instance level data thread safe. This can be done as below:

public class DemoClass

{

public synchronized void demoMethod(){}

}

or

public class DemoClass

{

public void demoMethod(){

synchronized (this)

{

//other thread safe code

}

}

}

or

public class DemoClass

{

private final Object lock = new Object();

public void demoMethod(){

synchronized (lock)

{

//other thread safe code

}

}

}

* + ***Class level locking***
    - Class level locking prevents multiple threads to enter in synchronized block in any of all available instances on runtime.
    - This means if in runtime there are 100 instances of DemoClass, then only one thread will be able to execute demoMethod() in any one of instance at a time, and all other instances will be locked for other threads. This should always be done to make static data thread safe.

public class DemoClass

{

public synchronized static void demoMethod(){}

}

or

public class DemoClass

{

public void demoMethod(){

synchronized (DemoClass.class)

{

//other thread safe code

}

}

}

or

public class DemoClass

{

private final static Object lock = new Object();

public void demoMethod(){

synchronized (lock)

{

//other thread safe code

}

}

}

* + ***Some Important Points***
    - Synchronization in java guarantees that no two threads can execute a synchronized method which requires same lock simultaneously or concurrently.
    - Synchronized keyword can be used only with methods and code blocks. These methods or blocks can be static or non-static both.
    - Whenever a thread enters into java synchronized method or block it acquires a lock and whenever it leaves java synchronized method or block it releases the lock. Lock is released even if thread leaves synchronized method after completion or due to any Error or Exception.
    - java synchronized keyword is re-entrant in nature it means if a java synchronized method calls another synchronized method which requires same lock then current thread which is holding lock can enter into that method without acquiring lock.
    - Java Synchronization will throw NullPointerException if object used in java synchronized block is null. For example, in above code sample if lock is initialized as null, the synchronized (lock) will throw NullPointerException.
    - Synchronized methods in Java put a performance cost on your application. So use synchronization when it is absolutely required. Also, consider using synchronized code blocks for synchronizing only critical section of your code.
    - It’s possible that both static synchronized and non-static synchronized method can run simultaneously or concurrently because they lock on different object.
    - According to the Java language specification you cannot use java synchronized keyword with constructor it’s illegal and result in compilation error.
    - Do not synchronize on non-final field on synchronized block in Java. Because reference of non-final field may change any time and then different thread might synchronizing on different objects i.e. no synchronization at all. Best is to use String class, which is already immutable and declared final.

1. **Can we make an Array or ArrayList as volatile in Java?**
   * ***volatile modifier***
     + The volatile is a modifier in Java which only applies to member variables, both instance and class variables and both primitive and reference type. It provides the happens-before guarantee which ensures that a write to a volatile variable will happen before any reading. This ensures that any modification to volatile object or primitive type will be visible to all threads i.e. it provides the visibility guarantee.
     + The volatile modifier also provides ordering guarantee because the compiler cannot re-order any code or operation which involves volatile variables (primitive and objects), but what is perhaps more important to know and remember is that volatile variable doesn't provide atomicity (except for write to the volatile double variable) and mutual exclusion, which is also the main difference between volatile and synchronized keyword.
     + There are certain restrictions with volatile keyword e.g. you cannot make a member variable both final and volatile at the same time, but you can make a static variable volatile in Java.
   * ***Can we make array volatile in Java?***
     + The answer is, Yes, you can make an array (both primitive and reference type array e.g. an int array and String array) volatile in Java but only changes to reference pointing to an array will be visible to all threads, not the whole array. What this means is that suppose you have a reference variable called primes as shown below:

protected volatile int[] primes = new int[10];

* + - then if you assign a new array to primes variable, change will be visible to all threads, but changes to individual indices will not be covered under volatile guarantee i.e.

primes = new int[20];

* + - will follow the "happens-before" rule and cause memory barrier refresh, but following code will not do so

primes[0] = 10;

primes[1] = 20;

primes[2] = 30;

primes[3] = 40;

* + - This means, if multiple threads are changing individual array elements e.g. storing updates, there won’t be any happens-before guarantee provided by the volatile modifier for such modification. So, if your use-case is to provide memory visibility guarantee for individual array elements than volatile is not the right choice. You must rely on other synchronization and a thread-safety mechanism to cover this case e.g. synchronized keyword, atomic variables, or ReentrantLock.
  + ***Can we make ArrayList or HashMap volatile in Java?***
    - The answer is same, of course, you can make a reference variable pointing to a Collection volatile in Java, but the happens-before guarantee will only be provided if the value of that reference variable is changed e.g. you assign a new collection to it.
    - Any modification done on actual collection object e.g. adding or removing elements from ArrayList will not invoke happens-before guarantee or memory barrier refresh.

1. **Why wait(), notify() and notifyAll() must be called from synchronized block or method in Java?**
   * + We use wait(), notify(), or notifyAll() method mostly for inter-thread communication in Java. One thread is waiting after checking a condition
     + e.g. In the classic Producer-Consumer problem, the Producer thread waits if the buffer is full and Consumer thread notify Producer thread after it creates a space in the buffer by consuming an element.
     + Calling notify() or notifyAll() methods issues a notification to a single or multiple thread that a condition has changed and once notification thread leaves synchronized block, all the threads which are waiting fight for object lock on which they are waiting and lucky thread returns from wait() method after reacquiring the lock and proceed further.
     + Let’s divide this whole operation into steps to see a possibility of race condition between wait() and notify() method in Java, we will use Produce Consumer thread example to understand the scenario better:
       - The Producer thread tests the condition (buffer is full or not) and confirms that it must wait (after finding buffer is full).
       - The Consumer thread sets the condition after consuming an element from a buffer.
       - The Consumer thread calls the notify () method; this goes unheard since the Producer thread is not yet waiting.
       - The Producer thread calls the wait () method and goes into waiting state.
     + So due to race condition here we potential lost a notification and if we use buffer or just one element Produce thread will be waiting forever and your program will hang.
     + This race condition is resolved by using synchronized keyword and locking provided by Java. In order to call the wait (), notify () or notifyAll () methods in Java, we must have obtained the lock for the object on which we're calling the method.
     + Since the wait() method in Java also releases the lock prior to waiting and reacquires the lock prior to returning from the wait() method, we must use this lock to ensure that checking the condition (buffer is full or not) and setting the condition (taking element from buffer) is atomic which can be achieved by using synchronized method or block in Java.
   * ***Summary*** 
     + IllegalMonitorStateException in Java which will occur if we don't call wait (), notify () or notifyAll () method from synchronized context.
     + Any potential race condition between wait and notify method in Java.
2. **How to Implement own Thread pool in java?**
   * + Thread Pool is a pool of threads which **reuses a fixed number of threads** to execute tasks.
     + At any point, at most nThreads threads will be active processing tasks. If additional tasks are submitted when all threads are active, **they will wait in the queue** until a thread is available.
     + ThreadPool implementation internally uses [**LinkedBlockingQueue**](http://www.javamadesoeasy.com/2015/03/custom-implementation-of.html)for adding and removing tasks.

## Advantage of ThreadPool?

* + - Instead of creating new thread every time for executing tasks, we can create ThreadPool which**reuses a fixed number of threads** for executing tasks.
    - As threads are reused, performance of our application improves drastically.

## [Life cycle of threads](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html) in ThreadPool

|  |
| --- |
| *new ThreadPoolsThread(taskQueue,this);* |

* + - When threads are created in constructor of ThreadPool they are in [**New state**](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html)***.***

|  |
| --- |
| *threadPoolsThread.start();* |

* + - When threads are started in constructor of ThreadPool they enter **Runnable state**.

|  |
| --- |
| *class ThreadPoolsThread extends Thread {*  *. . .*  *public void run() {*  *. . .*  *}*  *. . .*  *}* |

* + - When threads enter run() method of ThreadPoolsThread class they enter **Running state**.

* + - Thread can go from **running** to **waiting state** when taskQueue.take() is called and taskQueue’s size is 0. Thread will wait for tasks to become available.
    - **How can task become available/ Threads could go from waiting to runnable state?**
      * When execute() method of ThreadPool is called, it internally calls put() method on taskQueue to add tasks.
    - Once task is available thread can go from **waiting** to **runnable state**. And later thread scheduler puts thread from runnable to running state at discretion of implementation.
    - Once shutdown of ThreadPool is initiated, previously submitted tasks are executed by threads and then threads enter **dead state**.

## How ThreadPool works?

|  |
| --- |
| *ThreadPool threadPool=new ThreadPool(2);* |

* + - We will instantiate ThreadPool, in ThreadPool’s constructor *nThreads* number of threads are created and started.
    - Here 2 threads will be created and started in ThreadPool.
    - Then, threads will enter run() method of **ThreadPoolsThread** class and will call take() method on taskQueue.
    - If tasks are available thread will execute task by entering run() method of task (As tasks executed always implements Runnable). Else waits for tasks to become available.

|  |
| --- |
| *public void run() {*  *. . .*  *while (true) {*  *. . .*  *Runnable runnable = taskQueue.take();*  *runnable.run();*  *. . .*  *}*  *. . .*  *}* |

* + - **When tasks are added?**
      * When execute() method of **ThreadPool** is called, it internally calls put() method on taskQueue to add tasks.

|  |
| --- |
| *taskQueue.put(task);* |

* + - Once tasks are available all waiting threads are notified that task is available.
    - **How threads in ThreadPool can be stopped?**
      * shutDown() method can be used to stop threads executing in threadPool, once shutdown of ThreadPool is initiated, previously submitted tasks are executed, but no new tasks could be accepted.
    - After thread has executed task
      1. Check whether pool shutDown has been initiated or not, if pool shutDown has been initiated and
      2. taskQueue does not contain any unExecuted task (i.e. taskQueue's size is 0 ) than interrupt() the thread.

|  |
| --- |
| *public void run() {*  *. . .*  *while (true) {*  *. . .*  *runnable.run();*  *//task EXECUTED*  *. . .*  *if(this.threadPool.isPoolShutDownInitiated() &&*  *this.taskQueue.size()==0)*  *this.interrupt();*    *}*  *. . .*  *}* |

## Program to implement ThreadPool in java

|  |
| --- |
| **package** ThreadPool;  **import** java.util.concurrent.BlockingQueue;  **import** java.util.concurrent.LinkedBlockingQueue;  /\*\*  \* ThreadPool is a class which creates a thread pool that reuses a fixed  \* number of threads to execute tasks.  \* At any point, at most nThreads threads will be active processing tasks.  \* If additional tasks are submitted when all threads are active,  \* they will wait in the queue until a thread is available.  \*  \* Once shutdown of ThreadPool is initiated, previously submitted tasks are  \* executed, but no new tasks will be accepted.  \*  \*/  **class** ThreadPool {  **private** BlockingQueue<Runnable> taskQueue;       /\*      \* Once pool shutDown will be initiated, poolShutDownInitiated will become true.      \*/  **private** **boolean** poolShutDownInitiated = **false**;     /\* Constructor of ThreadPool      \* nThreads= is a number of threads that exist in ThreadPool.      \* nThreads number of threads are created and started. \*      \*/  **public** ThreadPool(**int** nThreads){         taskQueue = **new** LinkedBlockingQueue<Runnable>(nThreads);         //Create and start nThreads number of threads.  **for**(**int** i=1; i<=nThreads; i++){          ThreadPoolsThread threadPoolsThread=**new** ThreadPoolsThread(taskQueue,**this**);          threadPoolsThread.setName("Thread-"+i);          System.*out*.println("Thread-"+i +" created in ThreadPool.");          threadPoolsThread.start();   //start thread         }       }       /\*\*      \* Execute the task, task must be of Runnable type.      \*/  **public** **synchronized** **void**  execute(Runnable task) **throws** Exception{  **if**(**this**.poolShutDownInitiated)  **throw** **new** Exception("ThreadPool has been shutDown, no further tasks can be added");         /\*       \* Add task in sharedQueue,       \* and notify all waiting threads that task is available.            \*/         System.*out*.println("task has been added.");  **this**.taskQueue.put(task);     }  **public** **boolean** isPoolShutDownInitiated() {  **return** poolShutDownInitiated;     }     /\*\*      \* Initiates shutdown of ThreadPool, previously submitted tasks      \* are executed, but no new tasks will be accepted.      \*/  **public** **synchronized** **void** shutdown(){  **this**.poolShutDownInitiated = **true**;         System.*out*.println("ThreadPool SHUTDOWN initiated.");     }  }  /\*\*  \* These threads are created and started from constructor of ThreadPool class.  \*/  **class** ThreadPoolsThread **extends** Thread {  **private** BlockingQueue<Runnable> taskQueue;  **private** ThreadPool threadPool;  **public** ThreadPoolsThread(BlockingQueue<Runnable> queue,                   ThreadPool threadPool){         taskQueue = queue;  **this**.threadPool=threadPool;       }  **public** **void** run() {  **try** {                   /\*                   \* ThreadPool's threads will keep on running                   \* until ThreadPool is not shutDown (shutDown will interrupt thread) and                   \* taskQueue contains some unExecuted tasks.                   \*/  **while** (**true**) {                         System.*out*.println(Thread.*currentThread*().getName()                                       +" is READY to execute task.");                         /\*ThreadPool's thread will take() task from sharedQueue                         \* only if tasks are available else                         \* waits for tasks to become available.                         \*/                         Runnable runnable = taskQueue.take();                         System.*out*.println(Thread.*currentThread*().getName()                                       +" has taken task.");                         //Now, execute task with current thread.                         runnable.run();                           System.*out*.println(Thread.*currentThread*().getName()                                       +" has EXECUTED task.");                           /\*                         \* 1) Check whether pool shutDown has been initiated or not,                         \* if pool shutDown has been initiated and                         \* 2) taskQueue does not contain any                         \*    unExecuted task (i.e. taskQueue's size is 0 )                         \* than  interrupt() the thread.                         \*/  **if**(**this**.threadPool.isPoolShutDownInitiated()                                       &&  **this**.taskQueue.size()==0){  **this**.interrupt();                             /\*                                \*  Interrupting basically sends a message to the thread                                \*  indicating it has been interrupted but it doesn't cause                                \*  a thread to stop immediately,                                \*                                \*  if sleep is called, thread immediately throws InterruptedException                                \*/                                Thread.*sleep*(1);                         }                     }            } **catch** (InterruptedException e) {                   System.*out*.println(Thread.*currentThread*().getName()+" has been STOPPED.");            }     }  }  /\*\*  \* Task class which implements Runnable.  \*/  **class** Task **implements** Runnable{     @Override  **public** **void** run() {  **try** {                   Thread.*sleep*(2000);                   System.*out*.println(Thread.*currentThread*().getName()                                +" is executing task.");            } **catch** (InterruptedException e) {                   e.printStackTrace();            }     }  };  /\*\*  \* Test ThreadPool.  \*/  **public** **class** ThreadPoolTest{  **public** **static** **void** main(String[] args) **throws** Exception {            ThreadPool threadPool=**new** ThreadPool(2); //create 2 threads in ThreadPool            Runnable task=**new** Task();            threadPool.execute(task);            threadPool.execute(task);              threadPool.shutdown();     }    }  /\*OUTPUT  Thread-1 created in ThreadPool.  Thread-2 created in ThreadPool.  Thread-1 is READY to execute task.  Thread-2 is READY to execute task.  task has been added.  task has been added.  Thread-1 has taken task.  Thread-2 has taken task.  ThreadPool SHUTDOWN initiated.  Thread-1 is executing task.  Thread-1 has EXECUTED task.  Thread-1 has been STOPPED.  Thread-2 is executing task.  Thread-2 has EXECUTED task.  Thread-2 has been STOPPED.  \*/ |

## Let’s discuss output in detail, to get better understanding of ThreadPool program

* + - Note: I have mentioned output in green text.
    - Total number of thread created in ThreadPool was 2.

Thread-1 **created** in ThreadPool.

* + - Till now Thread-1 have been created.

Thread-2 **created** in ThreadPool.

* + - Till now Thread-2 have been created.

Thread-1 is **READY** to execute task.

* + - Thread-1 have entered run() method and taskQueue’s size is 0. So its waiting for task to become available.

Thread-2 is **READY** to execute task.

* + - Thread-2 have entered run() method and taskQueue’s size is 0. So its waiting for task to become available.

task has been **added**.

* + - execute() method of ThreadPool is called by main thread, it internally calls put() method on taskQueue to add tasks. Once tasks is available all waiting threads are notified that task is available.

task has been **added**.

* + - execute() method of ThreadPool is called by main thread, it internally calls put() method on taskQueue to add tasks. Once tasks is available all waiting threads are notified that task is available.

Thread-1 has taken task.

* + - As waiting Thread-1 has been notified it takes task.

Thread-2 has taken task.

* + - As waiting Thread-2 has been notified it takes task.

ThreadPool SHUTDOWN initiated.

* + - threadPool.shutdown() is called by main thread, previously submitted tasks are executed, but no new tasks will be accepted.

Thread-1 is executing task.

* + - Thread-1 is executing task, it’s in run() method of Task class (shutdown was initiated, but  previously submitted tasks are executed ).

Thread-1 has EXECUTED task.

* + - Thread-1 has executed task.

Thread-1 has been STOPPED.

* + - Thread-1 has been stopped.

Thread-2 is executing task.

* + - Thread-2 is executing task, it’s in run() method of Task class.

Thread-2 has EXECUTED task.

* + - Thread-2 has executed task.

Thread-2 has been STOPPED.

* + - Thread-2 has been stopped.

## How performance of applications is improved by reusing threads?

* + - So, after constructor and before shutdown is called on ThreadPool, threads will remain either in Running, Runnable or Waiting state. Therefore excluding overhead of being in New and Dead state.
    - Therefore, for every task executed by thread it would never go in new and dead state hence saving the time and will improve applications performance.

1. **Difference between deadlock, livelock and starvation?**

## 1. Understanding Deadlock

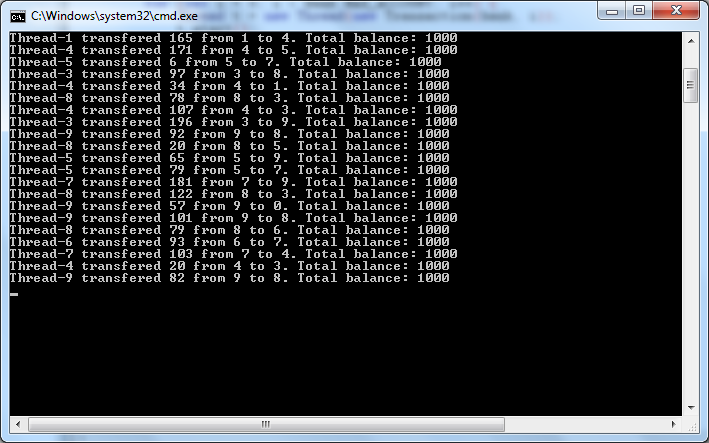
* + - **Deadlock describes a situation where two more threads are blocked because of waiting for each other forever**. When deadlock occurs, the program hangs forever and the only thing you can do is to kill the program.
    - Let’s consider the account transaction example in [this tutorial](https://www.codejava.net/java-core/concurrency/java-synchronization-tutorial-part-3-using-synchronized-keyword-intrinsic-locking). Modify the maximum amount can be transferred from 10 to 200 in the Bank class as follows:

public static final int MAX\_AMOUNT = 200;

* + - Look at the Transaction class you see the amount is chosen randomly by this statement:

int amount = (int) (Math.random() \* Bank.MAX\_AMOUNT);

* + - Now, recompile the Bank and Transaction classes, and then run the TransactionTest program. Guess what will happen?
    - You will see that the program runs for a few transactions and hangs forever, as shown in the following screenshot:



* + - The program encounters a deadlock and cannot continue. Why can deadlock happen when we increase the maximum amount of money can be transferred among accounts?
    - Let’s analyze the code to understand why.
    - In the Bank class you will each account is initialized with an amount of 100. Now the maximum amount can be transferred is 200, so there will be some threads trying to transfer an amount which is greater than the account’s balance, for example:
    - Thread 1 tries to transfer 150 from account 1 to account 2
    - Thread 2 tries to transfer 170 from account 3 to account 1
    - Account 1 has only 100 in balance so thread 1 has to wait for other threads to deposit more funds to this account. Similarly, thread 2 also has to wait because account 3 doesn’t have sufficient fund. Other threads may add funds to accounts 1 and 3, but if all threads are trying to transfer an amount greater than the account’s balance, they are waiting for each other forever. Hence deadlock occurs.
    - That’s why you see the program quickly runs into deadlock after few transactions have been done. It hangs and you have to press Ctrl + C to terminate the program.
    - You can ask why the previous version of the example runs fine. It’s because the maximum account is smaller (10) than the balance (100), so all accounts have enough fund to transfer.

## **Another Deadlock Example**

* + - Another common reason for deadlock problem is two or more threads attempt to acquire two locks simultaneously, but in different order. Consider the following class:

/\*\*

 \* Business.java

 \* This class is used to illustrate a deadlock situation.

 \*/

public class Business {

    private Object lock1 = new Object();

    private Object lock2 = new Object();

    public void foo() {

        synchronized (lock1) {

            synchronized (lock2) {

                System.out.println("foo");

            }

        }

    }

    public void bar() {

        synchronized (lock2) {

            synchronized (lock1) {

                System.out.println("bar");

            }

        }

    }

}

* + - As you can see, both the methods foo() and bar() try to acquire two lock objects lock1 and lock2 but in different order.
    - And consider the following test program:

/\*\*

 \* BusinessTest1.java

 \* This program tests for deadlock situation.

 \*/

public class BusinessTest1 {

    public static void main(String[] args) {

        Business business = new Business();

        Thread t1 = new Thread(new Runnable() {

            public void run() {

                business.foo();

            }

        });

        t1.start();

        Thread t2 = new Thread(new Runnable() {

            public void run() {

                business.bar();

            }

        });

        t2.start();

    }

}

* + - This program creates two threads, one executes the foo() method and another executes the bar() method on a shared instance of the Business class. But deadlock is likely never to occur because one thread can execute and exit a method very quickly so the other thread have chance to acquire the locks.
    - Let’s modify this test program in order to create 10 threads for executing foo() and other 10 threads for executing bar() as follows:

/\*\*

 \* BusinessTest2.java

 \* This program tests for deadlock situtation.

  \*/

public class BusinessTest2 {

    public static void main(String[] args) {

        Business business = new Business();

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

            new Thread(new Runnable() {

                public void run() {

                    business.foo();

                }

            }).start();

        }

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

            new Thread(new Runnable() {

                public void run() {

                    business.bar();

                }

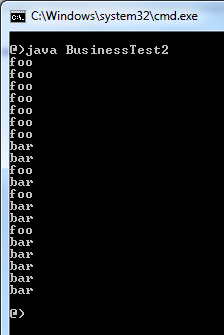
            }).start();

        }

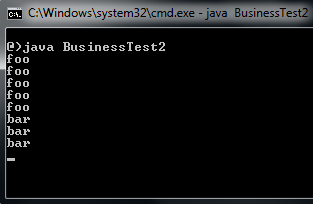
    }

}

* Run this program several times (4-10 times), you will see that sometimes the program runs fine:



* But sometimes it hangs like this:



* Why? It’s because deadlock happens. Let me explain how:
  + Thread 1 enters foo() method and it acquires lock1. At the same time, thread 2 enters bar() method and it acquires lock2.
  + Thread 1 tries to acquire lock2 which is currently held by thread 2, hence thread 1 blocks.
  + Thread 2 tries to acquire lock1 which is currently held by thread 1, hence thread 2 blocks.
* Both threads block each other forever, deadlock occurs and the program hangs.

## **So how to avoid deadlock?**

* Java doesn’t have anything to escape deadlock state when it occurs, so you have to design your program to avoid deadlock situation. Avoid acquiring more than one lock at a time. If not, make sure that you acquire multiple locks in consistent order. In the above example, you can avoid deadlock by synchronize two locks in the same order in both methods:

public void foo() {

    synchronized (lock1) {

        synchronized (lock2) {

            System.out.println("foo");

        }

    }

}

public void bar() {

    synchronized (lock1) {

        synchronized (lock2) {

            System.out.println("bar");

        }

    }

}

* Also try to shrink the synchronized blocks as small as possible to avoid unnecessary locking on code that doesn’t need to be synchronized.

# Understanding Livelock

* **Livelock describes situation where two threads are busy responding to actions of each other**. They keep repeating a particular code so the program is unable to make further progress:
  + Thread 1 acts as a response to action of thread 2
  + Thread 2 acts as a response to action of thread 1
* Unlike deadlock, threads are not blocked when livelock occurs. They are simply too busy responding to each other to resume work. In other words, the program runs into an infinite loop and cannot proceed further

## **A Livelock Example:**

* Let’s see an example: a criminal kidnaps a hostage and he asks for ransom in order to release the hostage. A police agrees to give the criminal the money he wants once the hostage is released. The criminal releases the hostage only when he gets the money. Both are waiting for each other to act first, hence livelock.
* Here’s the code of this example.
* Criminal class:.

/\*\*

 \* Criminal.java

 \* This class is used to demonstrate livelock situation

 \*/

public class Criminal {

    private boolean hostageReleased = false;

    public void releaseHostage(Police police) {

        while (!police.isMoneySent()) {

            System.out.println("Criminal: waiting police to give ransom");

            try {

                Thread.sleep(1000);

            } catch (InterruptedException ex) {

                ex.printStackTrace();

            }

        }

        System.out.println("Criminal: released hostage");

        this.hostageReleased = true;

    }

    public boolean isHostageReleased() {

        return this.hostageReleased;

    }

}

/\*\*

 \* Police.java

 \* This class is used to demonstrate livelock situation

 \*/

public class Police {

    private boolean moneySent = false;

    public void giveRansom(Criminal criminal) {

        while (!criminal.isHostageReleased()) {

            System.out.println("Police: waiting criminal to release hostage");

            try {

                Thread.sleep(1000);

            } catch (InterruptedException ex) {

                ex.printStackTrace();

            }

        }

        System.out.println("Police: sent money");

        this.moneySent = true;

    }

    public boolean isMoneySent() {

        return this.moneySent;

    }

}

Test class:

/\*\*

 \* HostageRescueLivelock.java

 \* This class is used to demonstrate livelock situation

 \*/

public class HostageRescueLivelock {

    static final Police police = new Police();

    static final Criminal criminal = new Criminal();

    public static void main(String[] args) {

        Thread t1 = new Thread(new Runnable() {

            public void run() {

                police.giveRansom(criminal);

            }

        });

        t1.start();

        Thread t2 = new Thread(new Runnable() {

            public void run() {

                criminal.releaseHostage(police);

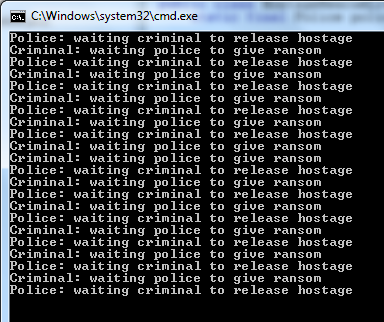
            }

        });

        t2.start();

    }

}

Run this program and you will see that it runs into a loop which never terminates:

* So how to avoid livelock? There’s no general guideline, you have to design your program to avoid livelock situation.

# 3. Understanding Starvation

* **Starvation describes a situation where a greedy thread holds a resource for a long time so other threads are blocked forever**. The blocked threads are waiting to acquire the resource but they never get a chance. Thus they starve to death.
* Starvation can occur due to the following reasons:
  + Threads are blocked infinitely because a thread takes long time to execute some synchronized code (e.g. heavy I/O operations or infinite loop).
  + A thread doesn’t get CPU’s time for execution because it has low priority as compared to other threads which have higher priority.
  + Threads are waiting on a resource forever but they remain waiting forever because other threads are constantly notified instead of the hungry ones.
* When a starvation situation occurs, the program is still running but doesn’t run to completion because some threads are not executed.

## **A Starvation Example:**

* Let’s see an example. Suppose we have a Worker class like this:

import java.io.\*;

/\*\*

 \* Worker.java

 \* This class is used to demonstrate starvation situation.

 \*/

public class Worker {

    public synchronized void work() {

        String name = Thread.currentThread().getName();

        String fileName = name + ".txt";

        try (

            BufferedWriter writer = new BufferedWriter(new FileWriter(fileName));

        ) {

            writer.write("Thread " + name + " wrote this mesasge");

        } catch (IOException ex) {

            ex.printStackTrace();

        }

        while (true) {

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

        }

    }

}

* This class has a synchronized method work() that creates a text file <thread-name>.txt and writes a message to it. Then it repeatedly prints a message:

<thread-name> is working

* And the following program creates 10 threads that call the work() method on a shared instance of the Worker class:

/\*\*

 \* StarvationExample.java

 \* This class is used to demonstrate starvation situation.

 \*/

public class StarvationExample {

    public static void main(String[] args) {

        Worker worker = new Worker();

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

            new Thread(new Runnable() {

                public void run() {

                    worker.work();

                }

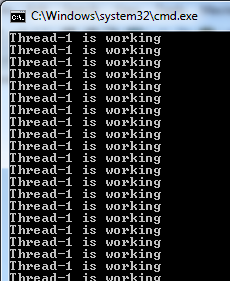
            }).start();

        }

    }

}

* Compile and run this program and you will see that there’s only one thread gets executed:



* According to the code logic, each thread should create a text file with the name of <thread-name>.txt but you see only one gets created, e.g. thread-1.txt. That means other threads are unable to execute the work() method.
* Why does this happen? It’s because the while loop runs forever so that the first executed thread never release the lock, causing other threads to wait forever.
* A solution to solve this starvation problem is to make the current thread waits for a specified amount of time so other threads have chance to acquire the lock on the Worker object:

while (true) {

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

    try {

        wait(1000);

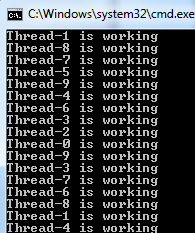
    } catch (InterruptedException ex) {

        ex.printStackTrace();

    }

}

* Recompile and run this program again and you will see that all threads get executed, proven by 10 text files created and in the output:



* In general, you should design your program to avoid starvation situation.

# 4. Conclusion

* So far I have helped you identify the 3 problems which can happen in multi-threading Java programs: deadlock, livelock and starvation.  Livelock and starvation are less common than deadlock but they still can occur. To summarize, the following points help you understand the key differences of these problems:
  + **Deadlock**:  All threads are blocked, the program hangs forever.
  + **Livelock**: No threads blocked but they run into infinite loops. The program is still running but unable to make further progress.
  + **Starvation**: Only one thread is running, and other threads are waiting forever.
* You should be aware of these problems which can occur with multiple threads and synchronization, and design your programs to avoid them.