1. [Threads\_use\_Exam](#Threads_use_Exam)
2. [What is dead lock & example](#Dead_example)
3. [ThreadLocal](#ThreadLocal)
4. [Diff\_Runnable\_Callable](#Diff_Runnable_Callable)
5. [Thread States](#Thread_states)
6. [Advatange\_Lock\_over\_synchronized](#Advatange_Lock_over_synchronized)
7. [deadlock\_Racecondition](#deadlock_Racecondition)
8. [Thread Class Methods](#Thread_Methods)
9. [Example of Syncronized Block](#Example_of_SynchronizedBlock)
10. [Synchronization rules](#synchronization_rules)
11. [executors\_submit\_execute](#executors_submit_execute)
12. [Wait & notify example](#wait_notify_Exmaple)
13. [Diff wait and sleep in java](#_Difference_between_Wait)
14. [Atomic operations](#atomic_operations_concur)

Java threads:

Java threads share the same memory /address space created by process from where these threads are created.

For example: Main method is the process,

MS WORD AND MS EXCEL are two different process.

MS WORD can save while editing the document. It means in the MS WORD process there are two threads, to the point , saving task is handled through a thread for the specified time.

if our computer has 4 CPUs with hyper-threading technologies, your code could potentially handle 8 Threads at the same time.

Thread Local in java

<http://javarevisited.blogspot.com/2012/05/how-to-use-threadlocal-in-java-benefits.html>

ThreadLocal in Java is another way to achieve [thread-safety](http://javarevisited.blogspot.com/2012/01/how-to-write-thread-safe-code-in-java.html) apart from writing immutable classes.

## ThreadLocal in Java is a different way to achieve thread-safety, it doesn't address synchronization requirement, instead it eliminates sharing by providing explicitly copy of Object to each thread. Since Object is no more shared there is no requirement of Synchronization which can improve scalability and performance of application. When to use ThreadLocal in Java

below are some well know usage of ThreadLocal class in Java:

1) ThreadLocal are fantastic to implement Per Thread Singleton classes or per thread context information like transaction id.

2) You can wrap any non Thread Safe object in ThreadLocal and suddenly its uses becomes Thread-safe, as its only being used by Thread Safe. One of the classic example of ThreadLocal is sharing SimpleDateForamt. Since [SimpleDateFormat is not thread safe](http://javarevisited.blogspot.com/2012/03/simpledateformat-in-java-is-not-thread.html), having a global formatter may not work but having per Thread formatter will certainly work.

3) ThreadLocal provides another way to extend Thread. If you want to preserve or carry information from one method call to another you can carry it by using ThreadLocal. This can provide immense flexibility as you don't need to modify any method.

On basic level ThreadLocal provides Thread Confinement which is extension of [local variable](http://javarevisited.blogspot.com/2012/02/difference-between-instance-class-and.html). while local variable only accessible on block they are declared, ThreadLocal are visible only in Single Thread. No two Thread can see each other’s ThreadLocal variable. Real Life example of ThreadLocal are in [J2EE application servers](http://javarevisited.blogspot.com/2012/05/5-difference-between-application-server.html) which uses java ThreadLocal variable to keep track of transaction and security Context. It makes lot of sense to share heavy object like Database Connection as ThreadLocal in order to avoid excessive creation and cost of locking in case of sharing global instance.

Example code

<http://veerasundar.com/blog/2010/11/java-thread-local-how-to-use-and-code-sample/>

## How to use Thread Local?

Java provides an [ThreadLocal](http://download.oracle.com/javase/6/docs/api/java/lang/ThreadLocal.html) object using which you can set/get thread scoped variables. Below is a code example demonstrating what I'd explained above.

Lets first have the Context.java file which will hold the transactionId field.

package com.veerasundar;

public class Context {

private String transactionId = null;

/\* getters and setters here \*/

}

Now create the MyThreadLocal.java file which will act as a container to hold our context object.

package com.veerasundar;

/\*\*

\* this class acts as a container to our thread local variables.

\* @author vsundar

\*

\*/

public class MyThreadLocal {

public static final ThreadLocal userThreadLocal = new ThreadLocal();

public static void set(Context user) {

userThreadLocal.set(user);

}

public static void unset() {

userThreadLocal.remove();

}

public static Context get() {

return userThreadLocal.get();

}

}

In the above code, you are creating a ThreadLocal object as a static field which can be used by rest of the code to set/get thread local variables.

Let's create our main class file which will generate and set the transaction ID in thread local and then call the business method.

package com.veerasundar;

public class ThreadLocalDemo extends Thread {

public static void main(String args[]) {

Thread threadOne = new ThreadLocalDemo();

threadOne.start();

Thread threadTwo = new ThreadLocalDemo();

threadTwo.start();

}

@Override

public void run() {

// sample code to simulate transaction id

Context context = new Context();

context.setTransactionId(getName());

// set the context object in thread local to access it somewhere else

MyThreadLocal.set(context);

/\* note that we are not explicitly passing the transaction id \*/

new BusinessService().businessMethod();

MyThreadLocal.unset();

}

}

Finally, here's the code for the BusinessService.java which will read from thread local and use the value.

package com.veerasundar;

public class BusinessService {

public void businessMethod() {

// get the context from thread local

Context context = MyThreadLocal.get();

System.out.println(context.getTransactionId());

}

}

When you run the ThreadLocalDemo file, you'll get the below output:

Thread-0

Thread-1

As you might see, even though we are not explicitly passing the transaction id, the value can be accessed from the business method and printed on the console. Adding to it, the transaction ID differs for each thread (0 and 1).

Well, that's it. I hope I'd explained it in a simple possible way. Please let me know what do you think about this article in comments. Do leave a comment if you want to add anything to this topic.

### Important points about Thread-Safety in Java

Here is some points worth remembering to **write thread safe code in Java**, these knowledge also helps you to avoid some serious concurrency issues in Java like race condition or [deadlock in Java](http://javarevisited.blogspot.com/2010/10/what-is-deadlock-in-java-how-to-fix-it.html):

1) Immutable objects are by default thread-safe because there state cannot be modified once created. Since String is immutable in Java, its inherently thread-safe.

2) Read only or [final variables in Java](http://javarevisited.blogspot.com/2011/12/final-variable-method-class-java.html) are also thread-safe in Java.

3) Locking is one way of achieving thread-safety in Java.

4) [Static variables](http://javarevisited.blogspot.com/2011/11/static-keyword-method-variable-java.html) if not synchronized properly becomes major cause of thread-safety issues.

5) Example of thread-safe class in Java: Vector, Hashtable, ConcurrentHashMap, String etc.

6) Atomic operations in Java are thread-safe e.g. reading a 32 bit int from memory because its an atomic operation it can't interleave with other thread.

7) local variables are also thread-safe because each thread has there own copy and using local variables is good way to writing thread-safe code in Java.

8) In order to avoid thread-safety issue minimize sharing of objects between multiple thread.

9) [Volatile keyword in Java](http://javarevisited.blogspot.com/2011/06/volatile-keyword-java-example-tutorial.html) can also be used to instruct thread not to cache variables and read from main memory and can also instruct JVM not to reorder or optimize code from threading perspective.

That’s all on **how to write thread safe class or code in Java** and avoid serious concurrency issues in Java. To be frank thread-safety is a little tricky concept to grasp, you need to think concurrently in order to catch whether a code is thread-safe or not. Also [JVM](http://javarevisited.blogspot.com/2011/12/jre-jvm-jdk-jit-in-java-programming.html) plays a spoiler since it can **reorder code** for optimization, so the code which looks sequential and runs fine in development environment not guaranteed to run similarly in production environment because JVM may ergonomically adjust itself as server JVM and perform more optimization and reorder which cause **thread-safety issues**.

Read more: <http://javarevisited.blogspot.com/2012/01/how-to-write-thread-safe-code-in-java.html#ixzz30V3tYUCI>

Difference between Callable and Runnable in Java

The Callable interface is similar to Runnable, in that both are designed for classes whose instances are potentially executed by another thread. A Runnable, however, does not return a result and cannot throw a checked exception.

Callable introduced in java 1.5 only.

* A Callable needs to implement call() method while a Runnable needs to implement run() method.
* A Callable can return a value but a Runnable cannot.
* A Callable can throw checked exception but a Runnable cannot.
* A Callable can be used with ExecutorService#invokeXXX methods but a Runnable cannot be.
* public interface Runnable {
* void run();
* }

public interfa

**Ruannable vs Callable<T>**

"*The Callable interface is similar to Runnable, in that both are designed for classes whose instances are potentially executed by another thread. A Runnable, however, does not return a result and cannot throw a checked exception.*"

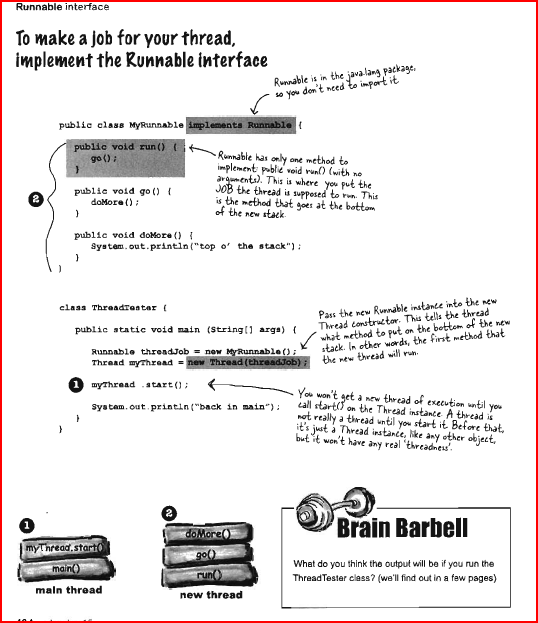
|  |  |
| --- | --- |
| **Runnable** | **Callable<T**> |
| Introduced in Java 1.0 | Introduced in Java 1.5 as part of java.util.concurrent library |
| Runnable cannot be parametrized | Callable is a parametrized type whose type parameter indicates the return type of its run method |
| Classes implementing Runnable needs to implement run() method | Classes implementing Callable needs to implement call() method |
| Runnable.run() returns no Value | Callable.call() returns a value of Type T |
| Can not throw Checked Exceptions | Can throw Checked Exceptions |
| public class RunnableTest implements Runnable {               @Override               public void run() {                          //any processing               } } | import java.util.concurrent.Callable;   public class CallableTest implements Callable {            @Override             public String call() throws Exception {                        // any processing                     return new String("I am Callable and can return value and throw checked exception");                }  } |

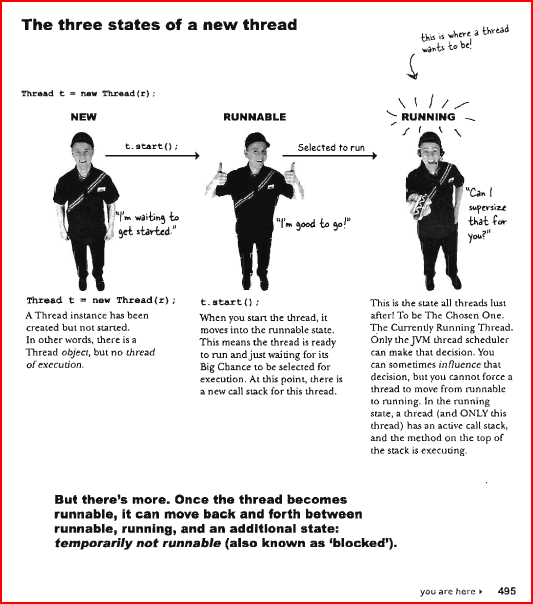
Each thread t.start() need a job to run. we need the code that you want to have run by a separate thread.

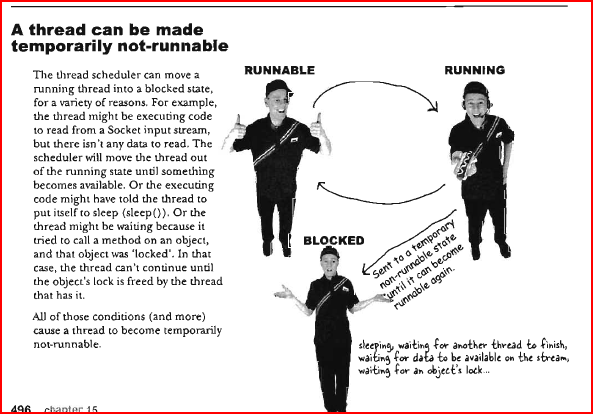
A Thread object represents a *thread of execution;* you'll create an instance of

class Thread each time you want to start up a new *thread* of execution.

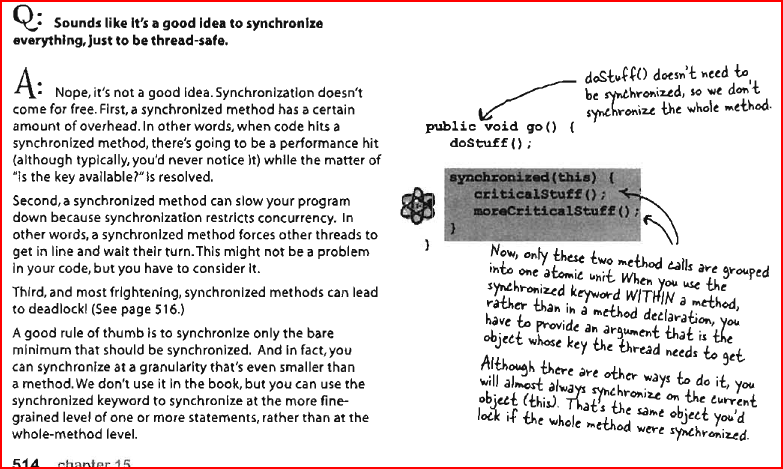
\*Runnable is to a Thread what job is to is to a worker. A Runnable is the job a thread is supposed to run.







Dead lock



Thread class methods:

## Thread Methods:

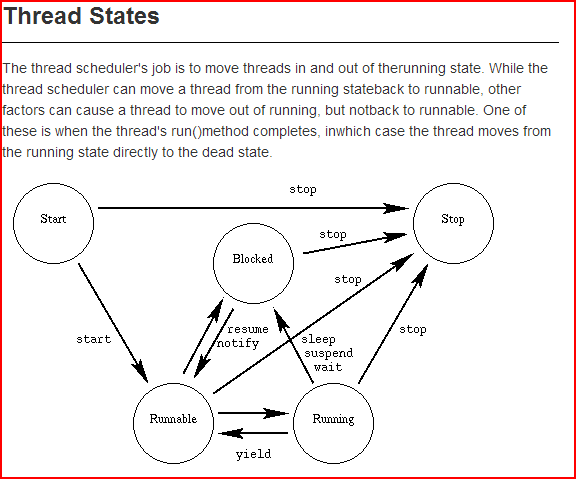
Following is the list of important methods available in the Thread class.

|  |  |
| --- | --- |
| **SN** | **Methods with Description** |
| 1 | **public void start()** Starts the thread in a separate path of execution, then invokes the run() method on this Thread object. |
| 2 | **public void run()** If this Thread object was instantiated using a separate Runnable target, the run() method is invoked on that Runnable object. |
| 3 | **public final void setName(String name)** Changes the name of the Thread object. There is also a getName() method for retrieving the name. |
| 4 | **public final void setPriority(int priority)** Sets the priority of this Thread object. The possible values are between 1 and 10. |
| 5 | **public final void setDaemon(boolean on)** A parameter of true denotes this Thread as a daemon thread. |
| 6 | **public final void join(long millisec)** The current thread invokes this method on a second thread, causing the current thread to block until the second thread terminates or the specified number of milliseconds passes. |
| 7 | **public void interrupt()** Interrupts this thread, causing it to continue execution if it was blocked for any reason. |
| 8 | **public final boolean isAlive()** Returns true if the thread is alive, which is any time after the thread has been started but before it runs to completion. |

The previous methods are invoked on a particular Thread object. The following methods in the Thread class are static. **Invoking one of the static methods performs the operation on the currently running thread.**

|  |  |
| --- | --- |
| **SN** | **Methods with Description** |
| 1 | **public static void yield()** Causes the currently running thread to yield to any other threads of the same priority that are waiting to be scheduled. |
| 2 | **public static void sleep(long millisec)** Causes the currently running thread to block for at least the specified number of milliseconds. |
| 3 | **public static boolean holdsLock(Object x)** Returns true if the current thread holds the lock on the given Object. |
| 4 | **public static Thread currentThread()** Returns a reference to the currently running thread, which is the thread that invokes this method. |
| 5 | **public static void dumpStack()** Prints the stack trace for the currently running thread, which is useful when debugging a multithreaded application. |

Thread states



## Thread Priorities:

Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled.

Java priorities are in the range between MIN\_PRIORITY (a constant of 1) and MAX\_PRIORITY (a constant of 10). **By default, every thread is given priority NORM\_PRIORITY (a constant of 5).**

"What is deadlock ?"  
answer is simple , when two or more threads waiting for each other to release lock and get stuck for infinite time , situation is called deadlock . it will only happen in case of multitasking.

though this could have many answers , my version is first I would look the code if I see nested synchronized block or calling one synchronized method from other or trying to get lock on different object then there is good chance of deadlock if developer is not very careful.

## 15 Java Thread Interview Questions and answers

**1) You have thread T1, T2 and T3, how will you ensure that thread T2 run after T1 and thread T3 run after T2?**

This thread interview questions is mostly asked in first round or phone screening round of interview and purpose of this multi-threading question is to check whether candidate is familiar with concept of *"join"* method or not. Answer of this multi-threading questions is simple it can be achieved by using **join** method of Thread class.

**2)** **What is the advantage of new Lock interface over synchronized block in Java? You need to implement a high performance cache which allows multiple reader but single writer to keep the integrity how will you implement it?**

The major advantage of lock interfaces on multi-threaded and concurrent programming is they provide two separate lock for reading and writing which enables you to write high performance data structure like [ConcurrentHashMap](http://javarevisited.posterous.com/difference-between-concurrenthashmap-and-coll) and [conditional blocking](http://javarevisited.blogspot.sg/2012/02/what-is-blocking-methods-in-java-and.html). This java threads interview question is getting increasingly popular and more and more follow-up questions come based upon answer of interviewee. I would strongly suggest reading **Locks** before appearing for any *java multi-threading interview* because now days Its  heavily used to build cache for electronic trading system on client and exchange connectivity space.

ABOUT LOCKS:

<http://tutorials.jenkov.com/java-concurrency/locks.html>

From Java 5 the package java.util.concurrent.locks contains several lock implementations, so you may not have to implement your own locks. B

<http://www.journaldev.com/2377/java-lock-example-and-concurrency-lock-vs-synchronized>

We use lock() and unlock() methods as shown below instead of synchronized keyword

public class Counter {

private Lock lock = new Lock();

private int count = 0;

public int inc(){

lock.lock();

int newCount = ++count;

lock.unlock();

return newCount;

}}

**3) What are differences between wait and sleep method in java?**

Another frequently asked thread interview question in Java mostly appear in phone interview. **Only major difference is** *wait* **release the lock or monitor while sleep doesn't release any lock or monitor while waiting**. **Wait is used for inter-thread communication while sleep is used to introduce pause on execution**. See my post [wait vs sleep in Java](http://javarevisited.blogspot.sg/2011/12/difference-between-wait-sleep-yield.html) for more differences

1. wait is called from synchronized context only while sleep can be called without synchronized block.
2. wait is called on Object while sleep is called on Thread

**4) Write code to implement blocking queue in Java?**

This is relatively tough java multi-threading interview question which servers many purpose, it checks whether candidate can actually write Java code using [thread](http://javarevisited.blogspot.sg/2011/02/how-to-implement-thread-in-java.html) or not, it sees how good candidate is on understanding concurrent scenarios and you can ask lot of follow-up question based upon his code. If he uses [wait() and notify() method](http://javarevisited.blogspot.sg/2011/05/wait-notify-and-notifyall-in-java.html) to implement blocking queue, Once interviewee successfully writes it  you can ask him to write it again using new java 5 concurrent classes etc.

**Where to use wait and sleep method in Java**

By reading properties and behavior of wait and sleep method it's clear that wait() method should be used in conjunction with notify() or notifyAll() method and intended for communication between two threads in Java while Thread.sleep() method is a utility method to introduce short pauses during program or thread execution. Given the requirement of synchronization for wait, it should not be used just to introduce pause or sleep in Java.

**5) Write code to solve the Produce consumer problem in Java?**

Similar to above questions on thread but more classic in nature, some time interviewer ask follow up questions How do you solve producer consumer problem in Java, well it can be solved in multiple way, I have shared one way to solve [producer consumer problem using BlockingQueue in Java](http://javarevisited.blogspot.sg/2012/02/producer-consumer-design-pattern-with.html) , so be prepare for surprises. Some time they even ask to implement solution of dining philosopher problem as well.

<http://javarevisited.blogspot.sg/2012/02/producer-consumer-design-pattern-with.html>  
**Producer consumer pattern** is every where in real life and depict coordination and collaboration. Like one person is preparing food (Producer) while other one is serving food (Consumer), both will use shared table for putting food plates and taking food plates. Producer which is the person preparing food will wait if table is full and Consumer (Person who is serving food) will wait if table is empty. table is a shared object here. On Java library **Executor framework** itself implement Producer Consumer design pattern be separating responsibility of addition and execution of task.  
  
**Producer-Consumer Problem** is also a [popular java interview question](http://javarevisited.blogspot.com/2011/04/top-20-core-java-interview-questions.html) where interviewer ask to implement producer consumer design pattern so that Producer should wait if Queue or bucket is full and Consumer should wait if queue or

bucket is empty. This problem can be implemented or solved by different ways in Java, classical way is using [wait and notify method](http://javarevisited.blogspot.com/2011/05/wait-notify-and-notifyall-in-java.html) to communicate between **Producer and Consumer thread** and blocking each of them on individual condition like full queue and empty queue.

import java.util.concurrent.BlockingQueue;

That’s all on **How to use Blocking Queue to solve Producer Consumer problem** or **example of Producer consumer design pattern**.

**6) Write a program which will result in deadlock? How will you fix deadlock in Java?**

Here n can be replace with 2 for simplest case and higher number to make question more intimidating. see  [How to avoid deadlock in java](http://javarevisited.blogspot.com/2010/10/what-is-deadlock-in-java-how-to-fix-it.html)  for more information on deadlock in Java.

**7) What is atomic operation? What are atomic operations in Java?**

Simple java thread interview questions, another follow-up is do you need to synchronized an atomic operation? :) You can read more about [java synchroniz**How do you detect deadlock in Java ?**ation](http://javarevisited.blogspot.com/2011/04/synchronization-in-java-synchronized.html) here.

**8) What is volatile keyword in Java? How to use it? How is it different from synchronized method in Java?**

Thread questions based on [volatile keyword in Java](http://javarevisited.blogspot.com/2011/06/volatile-keyword-java-example-tutorial.html) has become more popular after changes made on it on Java 5 and Java memory model. It’s good to prepare well about how volatile variables ensures visibility, ordering and consistency in concurrent environment.

**9) What is race condition? How will you find and solve race condition?**

Another multi-threading question in Java which appear mostly on senior level interviews. Most interviewer grill on recent race condition you have faced and how did you solve it and some time they will write sample code and ask you detect race condition. See my post on [Race condition in Java](http://javarevisited.blogspot.sg/2012/02/what-is-race-condition-in.html) for more information. In my opinion this is one of the best java thread interview question and can really test the candidate's experience on solving race condition or writing code which is free of data race or any other race condition. Best book to get mastery of this topic is "Concurrency practices in Java'".

**10) How will you take thread dump in Java? How will you analyze Thread dump?**

In UNIX you can use**kill -3** and then thread dump will print on log on windows you can use **"CTRL+Break".** Rather simple and focus thread interview question but can get tricky if he ask how you analyze it. Thread dump can be useful to analyze deadlock situations as well.

**11) Why we call start() method which in turns calls run() method, why not we directly call run() method ?**

Another classic java multi-threading interview question This was my original doubt when I started programming in thread. Now days mostly asked in phone interview or first round of interview at mid and junior level java interviews. Answer to this question is that, when you call start() method it creates new Thread and execute code declared **in run() while directly calling run() method doesn’t create any new thread and execute code on same calling** thread. Read my post [Difference between start and run method in Thread](http://javarevisited.blogspot.sg/2012/03/difference-between-start-and-run-method.html) for more details.

**12) How will you awake a blocked thread in java?**

This is tricky question on threading, blocking can result on many ways, if thread is blocked on IO then I don't think there is a way to interrupt the thread, let me know if there is any, on the other hand if thread is blocked due to result of calling wait(), sleep() or join() method you can interrupt the thread and it will awake by throwing InterruptedException. See my post [How to deal with blocking methods in Java](http://javarevisited.blogspot.sg/2012/02/what-is-blocking-methods-in-java-and.html) for more information on handling blocked thread.

**13) What is difference between CyclicBarriar and CountdownLatch in Java ?**

New java thread interview questions mostly to check familiarity with JDK 5 concurrent packages. One difference is that you can reuse CyclicBarrier once barrier is broken but you can not reuse ContdownLatch.

**14) What is immutable object? How does it help on writing concurrent application?**

Another classic interview questions on multi-threading, not directly related to thread but indirectly helps a lot. This java interview question can become more tricky if ask you to write an immutable class or ask you [Why String is immutable in Java](http://javarevisited.blogspot.com/2010/10/why-string-is-immutable-in-java.html) as follow-up.

**15) What are some common problems you have faced in multi-threading environment? How did you resolve it?**

Memory-interference, race conditions, [deadlock](http://javarevisited.blogspot.sg/2010/10/what-is-deadlock-in-java-how-to-fix-it.html), live lock and starvation are example of some problems comes in multi-threading and concurrent programming. There is no end of problem if you get it wrong and they will be hard to detect and debug. This is mostly experienced based interview question on java thread instead of fact based.

These were my favorite Java thread interview questions and mostly asked on Investment banks. This list is by no means complete so please contribute some of interesting java thread questions you have faced during interview. Purpose of this article is to collect and share great interview questions on multi-threading concept which not only helps on interview but opens door for learning new threading concept.

**Update:**

One of Javarevisited reader, Hemant has contributed some more thread interview questions in Java, though he hasn’t provide answer and left that job for me, I will certainly do when time allows, just like I have recently updated 10 Singleton interview question in Java with answers. If you guys know answers of this java concurrency questions than please post as comment:

Here is his comment “Good questions on multi-threading though you may need to prepare more in order to clear any multi-threading interview, you need to be familiar with concept of [immutability](http://avdheshsemwal.blogspot.sg/2012/02/why-string-is-immutable-or-final-in.html), [thread-safety](http://javarevisited.blogspot.sg/2011/07/java-multi-threading-interview.html), [race condition](http://javarevisited.blogspot.sg/2012/02/what-is-race-condition-in.html) and many more. 10 or 15 question is good for quick recap but you at-least need to prepare more than 50 questions on threading and concurrency to perform better on Java interview. You can find some interesting thread question below which is no doubt highly popular –

1)  Difference between green thread and native thread in Java?

2)  Difference between thread and process?

3)  What is context switching in multi-threading?

4)  Difference between deadlock and livelock, deadlock and starvation?

5)  What thread-scheduling algorithm is used in Java?

6)  What is thread-scheduler in Java?

7)  How do you handle un-handled exception in thread?

8)  What is thread-group, why its advised not to use thread-group in Java?

9)  Why Executor framework is better than creating and managing thread by application ?

10) Difference between Executor and Executors in Java?

10) How to find which thread is taking maximum cpu in windows and Linux server?

Apart from practicing these question answers, more important is to understand the concept behind these **multi-threading questions** simply mugging the answers of these *thread interview questions* is not going to help because there would be a lot of follow-up.

**Wait() and notify() example**

**public** **class** ThreadA {

**public** **static** **void** main(String[] args){

ThreadB b = **new** ThreadB();

b.start();

**synchronized**(b){

**try**{

System.out.println("Waiting for b to complete...");

b.wait();

}**catch**(InterruptedException e){

e.printStackTrace();

}

System.out.println("Total is: " + b.total);

}

}

}

**class** ThreadB **extends** Thread{

**int** total;

@Override

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

**synchronized**(**this**){

**for**(**int** i=0; i<100 ; i++){

total += i;

}

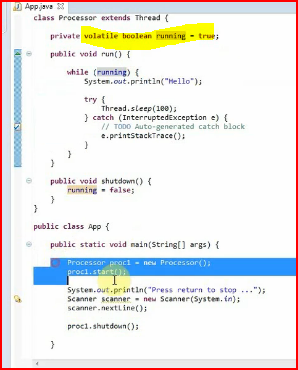
notify();

}

}

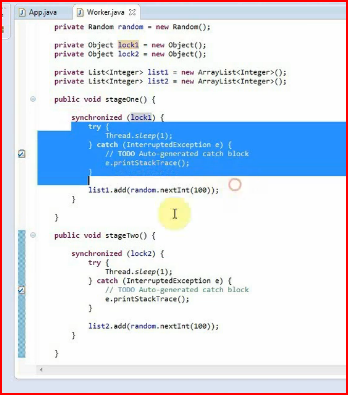
}

When using one threads value in another, use volatile because threads cache the value if we don’t reset them within the thread.

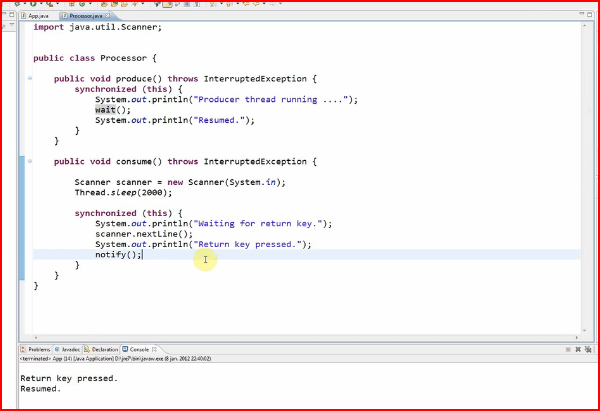


Multiple locks using synchronized blocks

https://www.udemy.com/java-multithreading/#/lecture/108987



Same lock on object use wait() and notify()



use of volatile keyword:

when multiple thread have shared variable, we should use it, because thread can have local copy of the thread variable value. To make sure the thread does not maintain its own copy and it should be in sync with threads, use *volatile* key word. It is applicable to only variables.

<http://stackoverflow.com/questions/3603157/volatile-keyword-in-java-clarification>

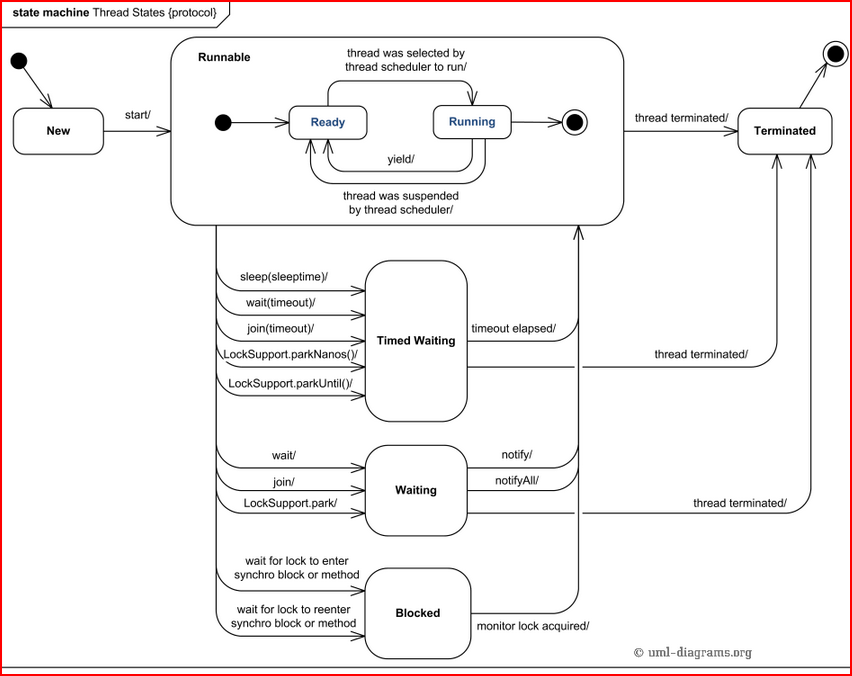
public static enum ****Thread.State****

extends [Enum](http://docs.oracle.com/javase/7/docs/api/java/lang/Enum.html)<[Thread.State](http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.State.html" \o "enum in java.lang)>

A thread state. A thread can be in one of the following states:

* [NEW](http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.State.html#NEW)  
  A thread that has not yet started is in this state.
* [RUNNABLE](http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.State.html#RUNNABLE)  
  A thread executing in the Java virtual machine is in this state.
* [BLOCKED](http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.State.html#BLOCKED)  
  A thread that is blocked waiting for a monitor lock is in this state.
* [WAITING](http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.State.html#WAITING)  
  A thread that is waiting indefinitely for another thread to perform a particular action is in this state.
* [TIMED\_WAITING](http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.State.html#TIMED_WAITING)  
  A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state.
* [TERMINATED](http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.State.html#TERMINATED)  
  A thread that has exited is in this state.

A thread can be in only one state at a given point in time. These states are virtual machine states which do not reflect any operating system thread states.



JAVA DEAD LOCK:

<http://javarevisited.blogspot.sg/2010/10/what-is-deadlock-in-java-how-to-fix-it.html>

**How do you detect deadlock in Java ?**  
though this could have many answers , my version is first I would look the code if I see nested synchronized block or calling one synchronized method from other or trying to get lock on different object then there is good chance of deadlock if developer is not very careful.  
  
other way is to find it when you actually get locked while running the application , try to take thread dump , in Linux you can do this by command "kill -3" , this will print status of all the thread in application log file and you can see which thread is locked on which object.  
  
other way is to use jconsole , jconsole will show you exactly which threads are get locked and on which object.  
  
once you answer this , they may ask you to **write code which will result in deadlock ?**  
here is one of my version  
  
public void method1(){  
synchronized(String.class){  
System.out.println("Aquired lock on String.class object");  
  
synchronized (Integer.class) {  
System.out.println("Aquired lock on Integer.class object");  
}  
}  
}  
  
public void method2(){  
synchronized(Integer.class){  
System.out.println("Aquired lock on Integer.class object");  
  
synchronized (String.class) {  
System.out.println("Aquired lock on String.class object");  
}  
}  
}  
  
If method1() and method2() both will be called by two or many threads , there is a good chance of deadlock because if thread 1 acquires lock on Sting object while executing method1() and thread 2 acquires lock on Integer object while executing method2() both will be waiting for each other to release lock on Integer and String to proceed further which will never happen.  
  
Read more: <http://javarevisited.blogspot.com/2010/10/what-is-deadlock-in-java-how-to-fix-it.html#ixzz306PcrKZl>

In Java 5 Arrays.toString(arr) or Arrays.deepToString(arr) for arrays within arrays. and add Note that Object[] version calls .toString() of each object in array. If my memory serves me correct, the output is even decorated in the exact way you're asking.

Edit: Don't forget to add import java.util.Arrays; like this:

Thread Local:

<http://java.dzone.com/articles/java-thread-local-%E2%80%93-how-use>

## What is Thread Local?

Thread Local can be considered as a scope of access, like a request scopeor session scope. It’s a thread scope. You can set any object in Thread Local and this object will be global and local to the specific thread which is accessing this object. Global **and**local!!? Let me explain:

* Values stored in Thread Local are globalto the thread, meaning that they can be accessed from anywhere inside that thread. If a thread calls methods from several classes, then all the methods can see the Thread Local variable set by other methods (because they are executing in same thread). The value need not be passed explicitly. It’s like how you use global variables.
* Values stored in Thread Local are local to the thread, meaning that each thread will have it’s own Thread Local variable. One thread can not access/modify other thread’s Thread Local variables.

Well, that’s the concept of Thread Local. I hope you understood it (if not, please leave a comment).

Lock vs synchronized

Based on above details and program, we can easily conclude following differences between Lock and synchronization.

1. Lock provides more visibility and options for locking, unlike synchronized where a thread might end up waiting indefinitely for the lock, we can use **tryLock**() to make sure thread waits for specific time only.
2. Synchronization code is much cleaner and easy to maintain whereas with Lock we are forced to have try-finally block to make sure Lock is released even if some exception is thrown between lock() and unlock() method calls.
3. synchronization blocks or methods can cover only one method whereas we can acquire the lock in one method and release it in another method with Lock API.
4. synchronized keyword doesn’t provide fairness whereas we can set fairness to true while creating ReentrantLock object so that longest waiting thread gets the lock first.
5. We can create different conditions for Lock and different thread can await() for different conditions.

That’s all for Java Lock example and a comparative analysis with synchronized keyword.

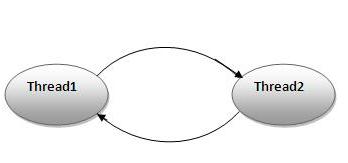
Busy Spinning?

<http://java67.blogspot.com/2012/08/5-thread-interview-questions-answers-in.html>

 It's a wait strategy, where one thread wait for a condition to become true, but instead of calling wait or sleep method and releasing CPU, it just spin. This is particularly useful if condition is going to be true quite quickly i.e. in millisecond or micro second. Advantage of not releasing CPU is that, all cached data and instruction are remained unaffected, which may be lost, had this thread is suspended on one core and brought back to another thread.

# Deadlock:

|  |
| --- |
| Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread. Since, both threads are waiting for each other to release the lock, the condition is called deadlock. |



### Example of Deadlock in java:

1. **public** **class** DeadlockExample {
2. **public** **static** **void** main(String[] args) {
3. **final** String resource1 = "ratan jaiswal";
4. **final** String resource2 = "vimal jaiswal";
5. // t1 tries to lock resource1 then resource2
6. Thread t1 = **new** Thread() {
7. **public** **void** run() {
8. **synchronized** (resource1) {
9. System.out.println("Thread 1: locked resource 1");
11. **try** { Thread.sleep(100);} **catch** (Exception e) {}
13. **synchronized** (resource2) {
14. System.out.println("Thread 1: locked resource 2");
15. }
16. }
17. }
18. };
20. // t2 tries to lock resource2 then resource1
21. Thread t2 = **new** Thread() {
22. **public** **void** run() {
23. **synchronized** (resource2) {
24. System.out.println("Thread 2: locked resource 2");
26. **try** { Thread.sleep(100);} **catch** (Exception e) {}
28. **synchronized** (resource1) {
29. System.out.println("Thread 2: locked resource 1");
30. }
31. }
32. }
33. };

36. t1.start();
37. t2.start();
38. }
39. }

Differences and the common points (if any) between a **race** and a **dead lock** ? An detailed answer would be appreciated ;).

**Race Conditions**

A race condition occurs when two threads access a shared variable at the same time. The first thread reads the variable, and the second thread reads the same value from the variable. Then the first thread and second thread perform their operations on the value, and they race to see which thread can write the value last to the shared variable. The value of the thread that writes its value last is preserved, because the thread is writing over the value that the previous thread wrote.

**Deadlocks**

A deadlock occurs when two threads each lock a different variable at the same time and then try to lock the variable that the other thread already locked. As a result, each thread stops executing and waits for the other thread to release the variable. Because each thread is holding the variable that the other thread wants, nothing occurs, and the threads remain deadlocked.

### Example of Synchronized Block in Java

Using **synchronized block in java** is also similar to using **synchronized keyword in methods**. Only important thing to note here is that if object used to lock synchronized block of code, Singleton.class in below example is null then Java synchronized block will throw a [NullPointerException](http://javarevisited.blogspot.com/2012/06/common-cause-of-javalangnullpointerexce.html).

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

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

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

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

**synchronized**(Singleton.class){

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

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

}

}

**return** \_instance;

}

Read more: <http://javarevisited.blogspot.com/2011/04/synchronization-in-java-synchronized.html#ixzz35Q0PyTEu>

**JAVA SYNCHNORIZATION RULES**

<http://javarevisited.blogspot.com/2011/04/synchronization-in-java-synchronized.html>

15. **You cannot apply java synchronized keyword with variables** and can not use java volatile keyword with method.

19. 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. example of synchronizing on non final field :

private String lock = new String("lock");

synchronized(lock){

System.out.println("locking on :" + lock);

}

Its **not recommended to use String object as lock in java synchronized block** because [string is immutable object](http://javarevisited.blogspot.com/2010/10/why-string-is-immutable-in-java.html) and literal string and interned string gets stored in String pool. so by any chance if any other part of code or any third party library used same String as there lock then they both will be locked on same object despite being completely unrelated which could result in unexpected behavior and bad performance. instead of String object its advised to use new Object() for **Synchronization in Java on synchronized block**.  
  
**private static final String LOCK = "lock";   //not recommended**  
private static final Object OBJ\_LOCK = new Object(); //better  
  
public void process() {  
   **synchronized(LOCK) {**  
      ........  
   }  
}  
  
21. From Java library Calendar and [SimpleDateFormat classes are not thread-safe](http://javarevisited.blogspot.sg/2012/03/simpledateformat-in-java-is-not-thread.html) and requires **external synchronization in Java** to be used in multi-threaded environment.  

Prefer ReentrantLock over synchronized keyword, it provides more control on lock acquisition, lock release and better performance compared to synchronized keyword.

# [Choose between ExecutorService's submit and ExecutorService's execute](http://stackoverflow.com/questions/3929342/choose-between-executorservices-submit-and-executorservices-execute)

There is a difference concerning exception/error handling.

A task queued with execute() that generates some Throwable will cause the UncaughtExceptionHandlerfor the Thread running the task to be invoked. The default UncaughtExceptionHandler, which typically prints the Throwable stack trace to System.err, will be invoked if no custom handler has been installed.

On the other hand, a Throwable generated by a task queued with submit() will bind the Throwable to the Future that was produced from the call to submit(). Calling get() on that Future will throw an ExecutionException with the original Throwable as its cause (accessible by calling getCause() on the ExecutionException).

The submit(...) method is executor framework extension introduced in ExecutorService interface.  
It's main difference from execute(Runnable) is that it can accept Callable<V> (whereas execute()accepts only Runnable) and returns instance of Future<V>, which you can use later in caller to retrieve result of asynchronous computation (potentially blocking until computation is completed).

1. The java.util.concurrent.**ThreadPoolExecutor** is an implementation of the ExecutorService interface. The **ThreadPoolExecutor** executes the given task ( Callable or Runnable ) using one of its internally pooled threads. The thread pool contained inside the **ThreadPoolExecutor** can contain a varying amount of threads.

# [Java Thread: notify() and wait() examples](http://www.programcreek.com/2009/02/notify-and-wait-example/)

1. synchronized keyword is used for exclusive accessing.
2. To make a method synchronized, simply add the synchronized keyword to its declaration. Then no two invocations of synchronized methods on the same object can interleave with each other.
3. Synchronized statements must specify the object that provides the intrinsic lock. When synchronized(this) is used, you have to avoid to synchronizing invocations of other objects' methods.
4. wait() tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify( ).
5. notify() wakes up the first thread that called wait() on the same object.

2. notify() and wait() - example 1

|  |
| --- |
| **public** **class** ThreadA {  **public** **static** **void** main(String[] args){  ThreadB b = **new** ThreadB();  b.start();    **synchronized**(b){  **try**{  System.out.println("Waiting for b to complete...");  b.wait();  }**catch**(InterruptedException e){  e.printStackTrace();  }    System.out.println("Total is: " + b.total);  }  }  }    **class** ThreadB **extends** Thread{  **int** total;  @Override  **public** **void** run(){  **synchronized**(**this**){  **for**(**int** i=0; i<100 ; i++){  total += i;  }  notify();  }  }  } |

In the example above, an object, b, is synchronized. b completes the calculation before Main thread outputs its total value.

Output:

Waiting for b to complete...

Total is: 4950

### Difference between Wait and Sleep in Java

Main *difference between wait and sleep is that wait()* method **release the acquired monitor** when thread is waiting while Thread.sleep() method **keeps the lock** or monitor even if thread is waiting. Also wait method in java should be called from synchronized method or block while there is no such requirement for sleep() method. Another difference is **Thread.sleep()** method is a static method and **applies on current thread**, while wait() is an instance specific method and only got wake up if some other thread calls notify method on same object. also in case of sleep, sleeping thread immediately goes to Runnable state after waking up while in case of wait, waiting thread first acquires the lock and then goes into Runnable state. So based upon your need if you require a specified second of pause use sleep() method or if you want to implement inter-thread communication use wait method.

here is list of difference between wait and sleep in Java :  
  
1) wait is called from synchronized context only while sleep can be called without synchronized block. see [Why wait and notify needs to call from synchronized](http://javarevisited.blogspot.com/2011/05/wait-notify-and-notifyall-in-java.html)

2) wait is called on Object while sleep is called on Thread. see [Why wait and notify are defined in object class instead of Thread.](http://javarevisited.blogspot.com/2012/02/why-wait-notify-and-notifyall-is.html)  
  
3) waiting thread can be awake by calling notify and notifyAll while sleeping thread can not be awaken by calling notify method.  
  
4) wait is normally done on condition, Thread wait until a condition is true while sleep is just to put your thread on sleep.

5) wait release lock on object while waiting while sleep doesn’t release lock while waiting.

### Difference between yield and sleep in java

*Major difference between yield and sleep in Java* is that yield() method **pauses the currently executing** thread temporarily for giving a chance to the remaining waiting threads of the same priority to execute. If there is no waiting thread or all the waiting threads have a lower priority then the same thread will continue its execution. The yielded thread when it will get the chance for execution is decided by the thread scheduler whose behavior is vendor dependent. Yield method doesn’t guarantee  that current thread will pause or stop but it guarantee that CPU will be relinquish by current Thread as a result of call to Thread.yield() method in java.

Sleep method in Java has two variants one which takes millisecond as sleeping time while other which takes both mill and nano second for sleeping duration.

**sleep(long millis)**

or

**sleep(long millis,int nanos)**

Cause the **currently executing thread to sleep** for the specified number of milliseconds plus the specified number of nanoseconds.

Read more: <http://javarevisited.blogspot.com/2011/12/difference-between-wait-sleep-yield.html#ixzz3R7fIgpw3>

Read more: <http://javarevisited.blogspot.com/2011/12/difference-between-wait-sleep-yield.html#ixzz3R7exKptK>

<http://www.javatechtipssharedbygaurav.com/2013/11/about-atomic-operation-in-java.html>

About Atomic Operation in Java

**What is Atomic Operation in java?**

Atomic means each action take place in one step without interruption or we can justify that operation is performed as a single unit of work without the possibility of interference from other operations.

An Atomic operation can't stop in the middle, either it happened completely or doesn't happen at all. No side effects of an atomic operation is visible until the action/operation is complete.

According to java language specification, it  guarantees that

      Reads and writes are atomic for reference variables and for most primitive variables (for all primitive data types except long and double).

      Read and write are atomic for all variable declared volatile including long and double variables.

The operation like below is not an atomic operation:-

            int i++;

The upper operation is having 3 steps to complete.

1).    Reading the current value of i;

2).    Incrementing the current value of i;

3).    Writing the modified value of i;

Example of non thread safe code in java:-

**package** com.gaurav.java.atomictest;

**public** **class** Counter {

**private** **int** incrementCounter;

      /\*

       \* This method is not a thread safe method because ++ is not an atomic

       \* operation

       \*/

**public** **int** getIncrementCounter() {

**return** incrementCounter++;

      }

}

In the above example, inside the Counter class the  *getIncrementCounter*() method is not a thread safe operation because ++(increment operator) is not atomic operation and I mentioned earlier that this can be broken down into three different steps. So if multiple threads call this *getIncrementCounter*() method simultaneously then each of these three operation may overlap with each other. For example while thread 1 is updating value , thread 2 reads and but still gets old value, which eventually let thread 2 override thread 1 increment and one count is lost because multiple threads are working concurrently.

***Writing thread safe code in java for above scenario's***

***There are many ways to make the above code as thread safe in Java:***

**First Approach**

With the use of synchronized keyword in Java by providing locking to the getIncrementCounter() method, we can assure that only one thread can execute it at a time which removes possibility of coinciding or overlapping.

**package** com.gaurav.java.atomictest;

**public** **class** SynchronizeCounter {

**private** **int** incrementCounter;

      /\* This method is thread safe because of locking provided by synchronization \*/

**public** **synchronized** **int** getIncrementCounter() {

**return** incrementCounter++;

      }

}

**Second Approach**

With the use of Atomic Integer which is available in java.util.concurrent.atomic package with jdk1.5 api, which helps to make this ++ operation atomic and since atomic operations are thread-safe and saves cost of external synchronization.

**package** com.gaurav.java.atomictest;

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

**public** **class** AutomicIntegerCounter {

      AtomicInteger atomicCounterIncrement = **new** AtomicInteger(0);

      /\*

       \* This method is thread safe because the counter is incremented

       \* automatically using the AtomicInteger class methods

       \*/

**public** **int** getCountIncrementAutomatically() {

**return** atomicCounterIncrement.incrementAndGet();

      }

}

**Facts about thread-safety**

* Immutable objects are by default thread-safe because there state can not be changed once created.
* Read only or Final variables are useful in writing of thread safe programs.
* With the use of already available thread safe classes like-StringBuffer, HashTable, Vector e.t.c
* With the use of local variables because each thread has there own copy of local variables.
* By minimizing sharing of objects between multiple thread we can also avaiod the issue of thread safely.
* Volatile keyword in Java can also be used to instruct thread not to cache variables and read from main memory and can also instruct JVM not to reorder or optimize code from threading perspective