<http://javarevisited.blogspot.sg/2011/12/difference-between-wait-sleep-yield.html>

Difference between Wait and Sleep , Yield in Java

**Difference between wait and sleep** or **difference between Sleep and yield in Java** are popular [core Java interview questions](http://javarevisited.blogspot.com/2011/04/top-20-core-java-interview-questions.html) and asked on [multi-threading interviews](http://javarevisited.blogspot.com/2011/07/java-multi-threading-interview.html). Out of three Sleep () and Yield () methods are defined in thread class while wait() is defined in Object class, which is another interview question. In this Java tutorial we will learn *what is sleep in Java*, important points of sleep in java *and difference between Wait and sleep in Java*.

## Wait vs Sleep vs Yield in Java

### 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 method](http://javarevisited.blogspot.com/2011/05/wait-notify-and-notifyall-in-java.html) for more detail.

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.

### Example of Thread Sleep method in Java

Here is sample code example of Sleep [Thread in Java](http://javarevisited.blogspot.com/2011/02/how-to-implement-thread-in-java.html). In this example we have put Main thread in Sleep for 1 second.

10 points about Thread sleep() meth/\*

Example of Thread Sleep method in Java

\*/

public class SleepTest {

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

System.out.println(Thread.currentThread().getName() + " is going to sleep for 1 Second");

try {

Thread.currentThread().sleep(1000);

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

System.out.println("Main Thread is woken now");

}

}

### Output:

### main is going to sleep for 1 Second

### Main Thread is woken nowod in Java

I have listed down some important and worth to remember points about Sleep() method of Thread [Class in Java](http://javarevisited.blogspot.com/2011/10/class-in-java-programming-general.html):

1) Thread.sleep() method is used to **pause the execution, relinquish the CPU and return it to thread** scheduler.

2) Thread.sleep() method is a [static method](http://javarevisited.blogspot.com/2011/11/static-keyword-method-variable-java.html) and always *puts current thread on sleep*.

3) Java has two variants of sleep method in Thread class one with one argument which takes milliseconds as duration for sleep and other method with two arguments one is millisecond and other is nanosecond.

4) Unlike [wait() method in Java](http://javarevisited.blogspot.com/2011/05/wait-notify-and-notifyall-in-java.html), sleep() method of Thread class **doesn't relinquish the lock** it has acquired.

5) sleep() method **throws Interrupted Exception** if another thread interrupt a sleeping thread in java.

6) With sleep() in Java its not guaranteed that when sleeping thread woke up it will definitely get CPU, instead it will go to Runnable state and fight for CPU with other thread.

7) There is a **misconception** about sleep method in Java that calling t.sleep() will put Thread "t" into sleeping state, that's not true because Thread.sleep method is a static method it always put current thread into Sleeping state and not thread "t".

That’s all on **Sleep method in Java**. We have seen **difference between sleep and wait** along with **sleep and yield in Java**. In Summary just keep in mind that both sleep() and yield() operate on current thread.

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

<http://javarevisited.blogspot.com/2012/02/why-wait-notify-and-notifyall-is.html>

Why wait, notify and notifyAll is defined in Object Class and not on Thread class in Java

**Why wait, notify and notifyAll is declared in Object Class instead of Thread** is famous core java interview question which is asked during all levels of Java interview ranging from 2 years, 4years to quite senior level position on java development. Beauty of this question is that it reflect what does interviewee knows about wait notify mechanism, how does it sees whole wait and notify feature and whether his understanding is not shallow on this topic. Like [Why Multiple inheritance is not supported in Java](http://javarevisited.blogspot.com/2011/07/why-multiple-inheritances-are-not.html) or [why String is final in java](http://javarevisited.blogspot.com/2010/10/why-string-is-immutable-in-java.html) there could be multiple answers of *why wait and notify is defined in Object class* and every one could justify there reason.

In my all interview experience I found that wait and notify still remains most confusing for most of Java programmer specially up-to 2 to 3 years and if they asked to write code using wait and notify they often struggle. So if you are going for any Java interview make sure you have sound knowledge of wait and notify mechanism as well as you are comfortable writing code using wait and notify like Produce Consumer problem or implementing Blocking queue etc. by the way This article is in continuation of  my earlier article related to wait and notify e.g. [Why Wait and notify requires to be called from Synchronized block or method](http://javarevisited.blogspot.com/2011/05/wait-notify-and-notifyall-in-java.html) and  [Difference between wait, sleep and yield method in Java](http://javarevisited.blogspot.com/2011/12/difference-between-wait-sleep-yield.html) , if you haven’t read you may found interesting.

## Reason Why Wait , Notify and NotifyAll are in Object Class.

Here are some thoughts on why they should not be in Thread class which make sense to me :

1) Wait and notify is not just normal methods or synchronization utility, more than that they are **communication mechanism between two threads in Java**. And Object class is correct place to make them available for every object if this mechanism is not available via any java keyword like synchronized. Remember synchronized and wait notify are two different area and don’t confuse that they are same or related. Synchronized is to provide mutual exclusion and ensuring [thread safety of Java class](http://javarevisited.blogspot.com/2012/01/how-to-write-thread-safe-code-in-java.html) like race condition while wait and notify are communication mechanism between two thread.

2 )**Locks are made available on per Object basis**, which is another reason wait and notify is declared in Object class rather then Thread class.

3) In Java in order to enter critical section of code, Threads needs lock and they wait for lock, they don't know which threads holds lock instead they just know the lock is hold by some thread and they should wait for lock instead of knowing which thread is inside the synchronized block and asking them to release lock. this analogy fits with wait and notify being on object class rather than thread in Java.

These are just my thoughts on **why wait and notify method is declared in Object class rather than Thread in Java** and you have different version than me. In reality its another design decision made by Java designer like [not supporting Operator overloading in Java](http://javarevisited.blogspot.com/2011/08/why-java-does-not-support-operator.html). Anyway please post if you have any other convincing reason *why wait and notify method should be in Object class and not on Thread*.

Update:

@Lipido has made an insightful comment , which is worth adding here. read his comment for full text

"Java is based on Hoare's monitors idea (http://en.wikipedia.org/wiki/Monitor\_%28synchronization%29). In Java all object has a monitor. Threads waits on monitors so, to perform a wait, we need 2 parameters:

- a Thread

- a monitor (any object)

In the Java design, the thread can not be specified, it is always the current thread running the code. However, we can specify the monitor (which is the object we call wait on). This is a good design, because if we could make any other thread to wait on a desired monitor, this would lead to an "intrusion", posing difficulties on designing/programming concurrent programs. Remember that in Java all operations that are intrusive in another thread's execution are deprecated (e.g. stop())."

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

<http://java67.blogspot.com/2012/12/producer-consumer-problem-with-wait-and-notify-example.html>

Producer Consumer Problem with Wait and Notify Example

Producer Consumer Problem is a classical concurrency problem and in fact it is one of the concurrency design pattern. In last article we have seen solving [Producer Consumer problem in Java using blocking Queue](http://javarevisited.blogspot.sg/2012/02/producer-consumer-design-pattern-with.html) but one of my reader emailed me and requested code example and explanation of solving Producer Consumer problem in Java with [wait and notify method](http://javarevisited.blogspot.sg/2011/05/wait-notify-and-notifyall-in-java.html) as well, Since its often asked as one of the top [coding question in Java](http://java67.blogspot.sg/2012/08/10-java-coding-interview-questions-and.html). In this Java tutorial, I have put the code example of wait notify version of earlier producer consumer concurrency design pattern. You can see this is much longer code with explicit handling blocking conditions like when shared queue is full and when queue is empty. Since we have replaced [BlockingQueue](http://javarevisited.blogspot.sg/2012/12/blocking-queue-in-java-example-ArrayBlockingQueue-LinkedBlockingQueue.html) with Vector we need to implement blocking using [wait and notify](http://javarevisited.blogspot.sg/2012/02/why-wait-notify-and-notifyall-is.html) and that's why we have introduced produce(int i) and consume() method. If you see I have kept consumer thread little slow by allowing it to sleep for 50 Milli second to give an opportunity to producer to fill the queue, which helps to understand that Producer thread is also waiting when Queue is full.

## Java program to solve Producer Consumer Problem in Java

Here is complete Java program to solve producer consumer problem in Java programming language. In this program we have used wait and notify method from java.lang.Object class instead of using BlockingQueue for flow control.

**import** java.util.Vector;

**import** java.util.logging.Level;

**import** java.util.logging.Logger;

/\*\*

 \* **Java program to solve Producer Consumer problem using wait and notify**

 \* method in Java. Producer Consumer is also a popular concurrency design pattern.

 \*

 \* @author Javin Paul

 \*/

**public** **class** ProducerConsumerSolution {

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

**Vector** sharedQueue = **new** **Vector**();

**int** size = 4;

**Thread** prodThread = **new** **Thread**(**new** Producer(sharedQueue, size), "Producer");

**Thread** consThread = **new** **Thread**(**new** Consumer(sharedQueue, size), "Consumer");

        prodThread.start();

        consThread.start();

    }

}

**class** Producer **implements** **Runnable** {

**private** **final** **Vector** sharedQueue;

**private** **final** **int** SIZE;

**public** Producer(**Vector** sharedQueue, **int** size) {

**this**.sharedQueue = sharedQueue;

**this**.SIZE = size;

    }

    @**Override**

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

        for (**int** i = 0; i < 7; i++) {

**System**.out.println("Produced: " + i);

**try** {

                produce(i);

            } **catch** (**InterruptedException** ex) {

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

            }

        }

    }

**private** **void** produce(**int** i) **throws** **InterruptedException** {

*//wait if queue is full*

        while (sharedQueue.size() == SIZE) {

**synchronized** (sharedQueue) {

**System**.out.println("Queue is full " + **Thread**.currentThread().getName()

+ " is waiting , size: " + sharedQueue.size());

                sharedQueue.wait();

            }

        }

*//producing element and notify consumers*

**synchronized** (sharedQueue) {

            sharedQueue.add(i);

            sharedQueue.notifyAll();

        }

    }

}

**class** Consumer **implements** **Runnable** {

**private** **final** **Vector** sharedQueue;

**private** **final** **int** SIZE;

**public** Consumer(**Vector** sharedQueue, **int** size) {

**this**.sharedQueue = sharedQueue;

**this**.SIZE = size;

    }

    @**Override**

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

        while (**true**) {

**try** {

**System**.out.println("Consumed: " + consume());

**Thread**.sleep(50);

            } **catch** (**InterruptedException** ex) {

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

            }

        }

    }

**private** **int** consume() **throws** **InterruptedException** {

*//wait if queue is empty*

        while (sharedQueue.isEmpty()) {

**synchronized** (sharedQueue) {

**System**.out.println("Queue is empty " + **Thread**.currentThread().getName()

+ " is waiting , size: " + sharedQueue.size());

                sharedQueue.wait();

            }

        }

*//Otherwise consume element and notify waiting producer*

**synchronized** (sharedQueue) {

            sharedQueue.notifyAll();

**return** (**Integer**) sharedQueue.remove(0);

        }

    }

}

That’s all on **How to solve producer consumer problem in Java using wait and notify method**. I still think that using BlockingQueue to implement producer consumer design pattern is much better because of its simplicity and concise code. At the same time this problem is an excellent exercise to understand concept of wait and notify method in Java.

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

<http://javarevisited.blogspot.com/2011/05/wait-notify-and-notifyall-in-java.html>

Why wait notify and notifyAll called from synchronized block or method in Java

### Why wait (), notify () and notifyAll () must be called from synchronized block or method in Java

Most of Java developer knows that **wait() ,notify() and notifyAll() method of object class** must have to be called inside **synchronized** method or synchronized block in Java but how many times we thought why ? Recently this questions was asked to in Java interview to one of my friend, he pondered for a moment and replied that if we don't call wait () or notify () method from synchronized context we will receive IllegalMonitorStateException in java. He was right in terms of behavior of language but as per him interviewer was not completely satisfied with the answer and wanted to explain more about it. After the interview he discussed the same questions with me and I thought he might have told about race condition between **wait () and notify () in Java** that could exists if we don't call them inside synchronized method or block. Let’s see how it could happen:

We use **wait () and notify () or notifyAll () method mostly for inter-thread communication**. One thread is waiting after checking a condition e.g. In Producer Consumer example Producer Thread is waiting if buffer is full and Consumer thread notify Producer thread after he creates a space in buffer by consuming an element. calling notify() or notifyAll() 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 in 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:

   1. The Producer thread tests the condition (buffer is full or not) and confirms that it must wait (after finding buffer is full).

   2. The Consumer thread sets the condition after consuming an element from buffer.

   3. The Consumer thread calls the notify () method; this goes unheard since the Producer thread is not yet waiting.

   4. 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.

Now let's think how does this potential race condition get resolved? This race condition is resolved by using [synchronized keyword and locking provided by java](http://javarevisited.blogspot.com/2011/04/synchronization-in-java-synchronized.html). 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.

I am not sure if this is what interviewer was actually expecting but this what I thought would at least make sense, please correct me If I wrong and let us know if there is any other convincing reason of calling wait(), notify() or notifyAll method in Java.

Just to summarize we call **wait** (), **notify** () or **notifyAll** method in Java from [synchronized method or synchronized block in Java](http://javarevisited.blogspot.com/2011/04/synchronization-in-java-synchronized.html) to avoid:

1) **IllegalMonitorStateException in Java** which will occur if we don't call wait (), notify () or notifyAll () method from synchronized context.

2) **Any potential race condition between wait and notify method in Java**.

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

<http://javarevisited.blogspot.sg/2012/02/producer-consumer-design-pattern-with.html>

Producer Consumer Design Pattern with Blocking Queue Example in Java

**Producer Consumer Design pattern** is a classic concurrency or threading pattern which reduces coupling between

Producer and Consumer by separating Identification of work with Execution of Work. In producer consumer design pattern a shared queue is used to control the flow and this separation allows you to code producer and consumer separately. It also addresses the issue of different timing require to produce item or consuming item. by using **producer consumer pattern** both Producer and Consumer Thread can work with different speed. In this article we will see *What is producer consumer problem* which is very [popular multi-threading interview question](http://javarevisited.blogspot.com/2011/07/java-multi-threading-interview.html), How to solve producer consumer problem using Blocking Queue and Benefits of using Producer Consumer design pattern.

## Real World Example of Producer Consumer Design Pattern

**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.

## Benefit of Producer Consumer Pattern

Its indeed a useful [design pattern](http://javarevisited.blogspot.com/2011/12/factory-design-pattern-java-example.html) and used most commonly while writing multi-threaded or concurrent code. here

is few of its benefit:

1) Producer Consumer Pattern simple development. you can Code Producer and Consumer independently and Concurrently, they just need to know shared object.

2) Producer doesn't need to know about who is consumer or how many consumers are there. Same is true with Consumer.

3) Producer and Consumer can work with different speed. There is no risk of Consumer consuming half-baked item.

In fact by monitoring consumer speed one can introduce more consumer for better utilization.

4) Separating producer and Consumer functionality result in more clean, readable and manageable code.

## Producer Consumer Problem in Multi-threading

**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. With introduction of **BlockingQueue** Data Structure in Java 5 Its now much simpler because BlockingQueue provides this control implicitly by introducing [blocking methods](http://javarevisited.blogspot.com/2012/02/what-is-blocking-methods-in-java-and.html) put() and take(). Now you don't require to use wait and notify to communicate between Producer and Consumer. BlockingQueue put() method will block if Queue is full in case of Bounded Queue and take() will block if Queue is empty. In next section we will see a *code example of Producer Consumer design pattern*.

## Using Blocking Queue to implement Producer Consumer Pattern

*BlockingQueue* amazingly simplifies implementation of Producer-Consumer design pattern by providing outofbox support of blocking on put() and take(). Developer doesn't need to write confusing and critical piece of wait-notify code to implement communication. **BlockingQuue** is an interface and Java 5 provides different implantation like ArrayBlockingQueue and LinkedBlockingQueue , both implement FIFO order or elements, while ArrayLinkedQueue is bounded in nature LinkedBlockingQueue is optionally bounded. here is a complete **code example of Producer Consumer pattern** with BlockingQueue. Compare it with classic [wait notify](http://javarevisited.blogspot.com/2012/02/why-wait-notify-and-notifyall-is.html) code, its much simpler and easy to understand.

import java.util.concurrent.BlockingQueue;

import java.util.concurrent.LinkedBlockingQueue;

import java.util.logging.Level;

import java.util.logging.Logger;

public class **ProducerConsumerPattern** {

    public static void main(String args[]){

**//Creating shared object**

     BlockingQueue sharedQueue = new LinkedBlockingQueue();

**//Creating Producer and Consumer Thread**

     Thread prodThread = new Thread(new Producer(sharedQueue));

     Thread consThread = new Thread(new Consumer(sharedQueue));

**//Starting producer and Consumer thread**

     prodThread.start();

     consThread.start();

    }

}

**//Producer Class in java**

class **Producer** implements **Runnable** {

    private final **BlockingQueue** sharedQueue;

    public Producer(BlockingQueue sharedQueue) {

        this.sharedQueue = sharedQueue;

    }

    @Override

    public void run() {

        for(int i=0; i<10; i++){

            try {

                System.out.println("Produced: " + i);

                sharedQueue.put(i);

            } catch (InterruptedException ex) {

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

            }

        }

    }

}

**//Consumer Class in Java**

class Consumer implements Runnable{

    private final BlockingQueue sharedQueue;

    public Consumer (BlockingQueue sharedQueue) {

        this.sharedQueue = sharedQueue;

    }

    @Override

    public void run() {

        while(true){

            try {

                System.out.println("Consumed: "+ sharedQueue.take());

            } catch (InterruptedException ex) {

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

            }

        }

    }

}

You see Producer Thread  produced number and Consumer thread consumes it in FIFO order because blocking queue allows elements to be accessed in FIFO.

That’s all on **How to use Blocking Queue to solve Producer Consumer problem** or **example of Producer consumer design pattern**. I am sure its much better than wait notify example but be prepare with both if you are going for any Java Interview as Interview may ask you both way.

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

<http://javarevisited.blogspot.com/2012/01/difference-thread-vs-runnable-interface.html>

Difference between Thread vs Runnable interface in Java

**Thread vs Runnable in Java** is always been a confusing decision for beginners  in java. [Thread in Java](http://javarevisited.blogspot.com/2011/02/how-to-implement-thread-in-java.html) seems easy in comparison of Runnable because you just deal with one class **java.lang.Thread** while in case of using Runnable to implement Thread you need to deal with both Thread and Runnable two classes. though decision of using Runnable or Thread should be taken considering **differences between Runnable and Thread** and pros and cons of both approaches. This is also a very [popular thread interview questions](http://javarevisited.blogspot.com/2011/07/java-multi-threading-interview.html) and most of interviewer are really interested to know what is your point of view while choosing *Thread vs Runnable or opposite*. In this java article we will try to point out some *differences between Thread and Runnable in Java* which will help you to take an informed decision.

## Difference between Thread and Runnable interface in Java

### Thread vs Runnable in Java

Here are some of my thoughts on whether I should use **Thread or Runnable** for implementing task in Java, though you have another choice as "Callable" for implementing thread which we will discuss later.

1) [Java doesn't support multiple inheritance](http://javarevisited.blogspot.com/2011/07/why-multiple-inheritances-are-not.html), which means you can only extend one class in Java so once you extended Thread class you lost your chance and can not extend or inherit another [class in Java](http://javarevisited.blogspot.com/2011/10/class-in-java-programming-general.html).

2) In Object oriented programming extending a class generally means adding new functionality, modifying or improving behaviors. If we are not making any modification on Thread than use Runnable interface instead.

3) **Runnable** interface represent a Task which can be executed by either plain Thread or Executors or any other means. so logical separation of Task as Runnable than Thread is good design decision.

4) Separating task as **Runnable** means we can reuse the task and also has liberty to execute it from different means. since you can not restart a Thread once it completes. again **Runnable vs Thread** for task, Runnable is winner.

5) Java designer recognizes this and that's why Executors accept Runnable as Task and they have worker thread which executes those task.

6) Inheriting all Thread methods are additional overhead just for representing a Task which can can be done easily with Runnable.

These were some of notable **difference between Thread and Runnable in Java**, if you know any other differences on Thread vs Runnable than please share it via comments. I personally use Runnable over Thread for this scenario and recommends to use Runnable or Callable interface based on your requirement.

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

<http://javarevisited.blogspot.com/2011/02/how-to-implement-thread-in-java.html>

How to Implement Thread in Java with Example

**How to implement Thread in Java**

In my opinion Thread is one of the most important feature of Java programming language which helped it to become the most popular programming language. I remember, when I first started learning Java in one of the programming class on India how important Thread was portrait and how much emphasis given on clear understanding of multi-threading. It’s still popular and one of most sought after skill in Java programmer because writing concurrent and multi-threaded application in Java is challenging, despite Java providing excellent support at language level using synchronized and volatile keyword. Main problem with using multiple threads and writing multi-threaded code is issues related to concurrency e.g. [deadlock](http://javarevisited.blogspot.sg/2010/10/what-is-deadlock-in-java-how-to-fix-it.html), livelock, [race conditions](http://javarevisited.blogspot.sg/2012/02/what-is-race-condition-in.html) etc, It takes lot of effort to implement multi-threading correctly in Java application. In this core Java tutorial I will share my experience on different way of implementing Thread in Java;  By the way difference between Thread and Runnable in Java is also a very common core Java interview question and asked mostly during junior level Java interview. After reading this tutorial, you will not only able to create and start thread but also able to answer what is difference in two ways of implementing thread in Java, by implementing Runnable interface or by extending Thread class.

## How to make Thread in Java

There are two ways of implementing threading in Java

1) By extending java.lang.Thread class, or

2) By implementing java.lang.Runnable interface.

Before we go into implementation details I just like to cover when we use Thread in Java?  So we use thread if we want some part of code is executed parallel and we put that code inside run() method of either [Thread class or Runnable interface](http://javarevisited.blogspot.sg/2012/01/difference-thread-vs-runnable-interface.html).

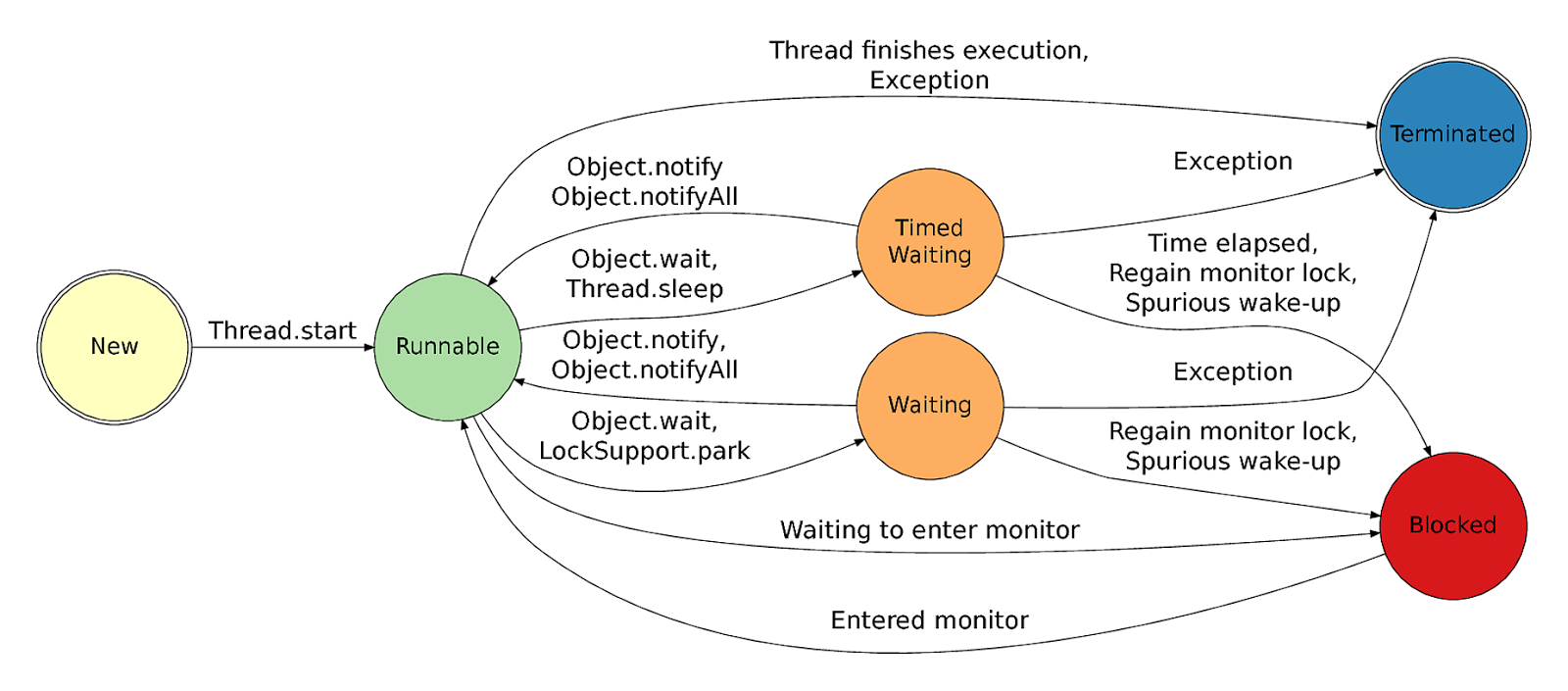
Actually public void run() method is defined in Runnable interface and since java.lang.Thread class implements Runnable interface it gets this method automatically. I remember by first Java multi threading example which was an animation program where multiple threads were used in Applet to create animation of words falling from top left, middle and top right of the page. That was pretty exciting at that time because till then I only know program which takes input from command prompt and [print output on command prompt](http://javarevisited.blogspot.sg/2011/11/run-java-program-from-command-prompt.html).

## Java Thread Tutorial and Example

So now the interview question  w*hich way of implementing Thread is better? Extending Thread class or implementing Runnable method?*

In my opinion implementing Runnable is better because in Java we can only extend one class so if we extend Thread class we can not extend any other class while by implementing Runnable interface we still have that option open with us.

Second reason which make sense to me is more on OOPS concept according to OOPS if we [extend a class](http://java67.blogspot.sg/2012/08/what-is-inheritance-in-java-oops-programming-example.html) we provide some new feature or functionality , So if the purpose is just to use the run() method to define code its better to use Runnable interface. If you are still not convince on why implementing Runnable is better than extending Thread class for creating threads in Java, I think it's time you should read [this](http://java67.blogspot.sg/2012/08/what-is-thread-and-runnable-in-java.html) article.

Thread is one of important Class in Java and multi-threading is most widely used feature,but there is no clear way to stop Thread in Java. Earlier there was a stop method exists in Thread Class but Java deprecated that method citing some safety reason. By default a Thread stops when execution of run() method finish either normally or due to any Exception.In this article we will How to Stop Thread in Java by using a boolean State variable or flag. Using flag to stop Thread is very popular way  of stopping thread and its also safe, because it doesn't do anything special rather than helping run() method to finish it self.

So first step is complete, you have implemented thread by now. Next step is to actually create object of thread class and start it. This is will create a separate path of execution parallel to main thread. Java thread is state based so it remains in predefined state at any given time and state transition occurs by calling different thread method. So, when you create object of your class which has implemented Runnable or extended Thread, you just create an object of Thread class, Thread will not start until you call the start() method of java.lang.Thread class. This is shown clearly in above thread state transition diagram in Java. It is now in NEW state, when we call start() method Java Virtual machine execute run() method of that Thread class it goes into RUNNBLE state. Now, it's upto thread scheduler to assign CPU to this thread. From here on it can either complete its execution and go to TERMINATED state or can go into WAITING, TIMED WAITING and BLOCKED state. By the way if you notice, when we call start() method, it eventually calls run() method, can anybody guess what will happen if we call the [run() method directly instead of calling start() method](http://javarevisited.blogspot.sg/2012/03/difference-between-start-and-run-method.html) ?

That another popular multi-threading interview question and answer is simple there would be no Error or Exception run() method will simply be executed in the same Thread and new Thread will not be created. Another follow up question would be what will happen if you call start() method twice in same Thread object e.g.

mythread.start();

mythread.start(); //this line will throw IllegalThreadStateException

//implementing Thread by extending Thread class

**public** **class** **MyThread** **extends** Thread{

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

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

}

}

//implementing Thread by implementing Runnable interface

**public** **class** **MyRunnable** **implements** Runnable{

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

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

}

}

//starting Thread in Java

Thread mythread = **new** MyThread(); //Thread created not started

mythread.setName("T1");

Thread myrunnable = **new** Thread(**new** MyRunnable(),"T2"); //Thread created

mythread.start(); //Thread started now but not running

myrunnable.start();

#### Bonus Tip

TIP1: It’s not guaranteed that thread mythread will start before thread myrunnable it depends upon Thread scheduler.

TIP2: Thread will be said to go on dead state once execution of run() method finished and you can not start that thread again.

Other **Java Thread tutorial** from Javarevisited Blog

1. Why wait and notify method must be called in synchronized context? ([see here](http://javarevisited.blogspot.com/2011/05/wait-notify-and-notifyall-in-java.html))
2. How Synchronization works in Java? ([read more](http://javarevisited.blogspot.com/2011/04/synchronization-in-java-synchronized.html))
3. How to write Thread-safe class in Java? ([read here](http://javarevisited.blogspot.sg/2012/01/how-to-write-thread-safe-code-in-java.html))
4. 50 Thread Questions from Java Interview for Experienced ([check here](http://javarevisited.blogspot.sg/2014/07/top-50-java-multithreading-interview-questions-answers.html))
5. How to Stop Thread in Java? ([see here](http://javarevisited.blogspot.sg/2011/10/how-to-stop-thread-java-example.html))
6. Inter thread communication in Java ([read more](http://javarevisited.blogspot.sg/2013/12/inter-thread-communication-in-java-wait-notify-example.html))
7. Difference between Daemon and User thread in Java ([read here](http://javarevisited.blogspot.sg/2012/03/what-is-daemon-thread-in-java-and.html))
8. How to create Thread Pool in Java ([read here](http://javarevisited.blogspot.sg/2013/07/how-to-create-thread-pools-in-java-executors-framework-example-tutorial.html))
9. How to check if a Thread holds an Object Lock? ([check here](http://javarevisited.blogspot.sg/2010/10/how-to-check-if-thread-has-lock-on.html))
10. Difference between wait(), sleep() and yield() in Java ([read more](http://javarevisited.blogspot.sg/2011/12/difference-between-wait-sleep-yield.html))
11. How to use ThreadLocal variable in Java? ([read here](http://javarevisited.blogspot.sg/2012/05/how-to-use-threadlocal-in-java-benefits.html))

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

<http://javarevisited.blogspot.com/2011/10/how-to-stop-thread-java-example.html>

How to Stop Thread in Java Code Example

## How to Stop Thread in Java

As I said earlier Thread in Java will stop once run() method finished. Another important point is that you can not restart a Thread which run() method has finished already , you will get an IllegalStateExceptio, here is a Sample Code for Stopping Thread in Java:

### Sample Code to Stop Thread in Java

private class Runner extends Thread{

boolean bExit = false;

public void exit(boolean bExit){

this.bExit = bExit;

}

@Override

public void run(){

while(!bExit){

System.out.println("Thread is running");

try {

Thread.sleep(500);

} catch (InterruptedException ex) {

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

}

}

}

}

**Should we make bExit Volatile**

Since every Thread has its own local memory in Java its good practice to make bExit volatile because we may alter value of bExit from any thread and making it volatile guarantees that Runner will also see any update done before making bExit.

That’s all on how to stop thread in Java , let me know if you find any other way of stopping threads in Java without using deprecated stop() method.

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

<http://java67.blogspot.sg/2012/08/what-is-thread-and-runnable-in-java.html>

What is Thread and Runnable in Java – Example

**What is Thread in Java**

Thread in Java is an independent path of execution which is used to run two task in parallel. When two Threads run in parallel that is called multi-threading in Java. Java is multi-threaded from start and excellent support of Thread at language level e.g. java.lang.Thread class, [synchronized keyword](http://javarevisited.blogspot.sg/2011/04/synchronization-in-java-synchronized.html), [volatile](http://java67.blogspot.sg/2012/08/what-is-volatile-variable-in-java-when.html) and [final keyword](http://javarevisited.blogspot.sg/2011/12/final-variable-method-class-java.html) makes writing concurrent programs easier in Java than any other programming language e.g. C++. Being multi-threaded is also a reason of Java's popularity and being number one programming language. On the other hand if your program divides a task between two threads it also brings lot of programming challenges and issues related to [synchronization](http://javarevisited.blogspot.sg/2011/04/synchronization-in-java-synchronized.html), [deadlock](http://javarevisited.blogspot.sg/2010/10/what-is-deadlock-in-java-how-to-fix-it.html), [thread-safety](http://javarevisited.blogspot.sg/2012/01/how-to-write-thread-safe-code-in-java.html) and [race conditions](http://javarevisited.blogspot.sg/2012/02/what-is-race-condition-in.html). In short answer of question *What is Thread in Java* can be given like "Thread is a class in Java but also a way to execute something in parallel independently in Java". Thread in Java requires a task which is executed by this thread independently and that task can be either Runnable or Callable which we will see in next section along with an example of  How to use multiple Thread in Java. [Difference between Thread and Runnable in Java](http://javarevisited.blogspot.sg/2012/01/difference-thread-vs-runnable-interface.html) is also a popular thread interview question in Java.

**What is Runnable in Java**

Runnable represent a task in Java which is executed by Thread. java.lang.Runnable is an [interface](http://javarevisited.blogspot.sg/2012/04/10-points-on-interface-in-java-with.html) and defines only one method called run(). When a [Thread](http://javarevisited.blogspot.sg/2011/02/how-to-implement-thread-in-java.html) is started in Java by using Thread.start() method it calls run() method of Runnable task which was passed to Thread during creation. Code written inside run() method is executed by this newly created thread. Since start() method internally calls run() method its been a doubt among Java programmers that why not directly call the run() method. This is also asked as [what is difference between start() and run() method in Java](http://javarevisited.blogspot.sg/2012/03/difference-between-start-and-run-method.html). Well when you call Runnable interface run() method directly , no new Thread will be created and task defined inside run() method is executed by calling thread.  There is another interface added in Java 1. 5 called Callable which can also be used in place of Runnable interface in Java. Callable provides additional functionality over Runnable in terms of returning result of computation. Since return type of run() method is void it can not return anything which is sometime necessary. On the other hand Callable interface defines call() method which has return type as Future which can be used to return result of computation from Thread in Java.

**Thread Example in Java.**

Here is a simple example of Thread in Java. In this Java program we create two Thread object and pass them two different Runnable instance which is implemented using Anonymous class in Java. We have also provided name to each thread as “Thread A” and “Thread B”, name is optional and if you don’t give name, Java will automatically provide default name for your Thread like “Thread 0” and “Thread 1”. When we start thread using start() method it calls run() method which has code for printing name of Thread two times for Thread A and three times for Thread B.

/\*\*

 \* Java Program to demonstrate **how to use Thread in Java with Example**

 \* Here two threads are provided Runnable interface implementation using

 \* anonymous class and when started they will print Thread's name.

 \* @author

 \*/

**public** **class** ThraedExample{

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

*//two threads in Java which runs in Parallel*

**Thread** threadA = **new** **Thread**(**new** **Runnable**(){

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

**for**(**int** i =0; i<2; i++){

**System**.out.println("This is thread : " + **Thread**.currentThread().getName());

                }

            }

        }, "Thread A");

*//Runnable interface is implemented using Anonymous Class*

**Thread** threadB = **new** **Thread**(**new** **Runnable**(){

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

**for**(**int** i =0; i<3; i++){

**System**.out.println("This is thread : " + **Thread**.currentThread().getName());

                }

            }

        }, "Thread B");

*//starting both Thread in Java*

        threadA.start(); *//start will call run method in new thread*

        threadB.start();

    }

}

That’s all on What is Thread in Java, **What is Runnable in Java** and How to use Thread in Java with Example. Thread is one of the most important concept in Java and must for every Java programmer, it also forms basis of a Java interview. Checkout these [15 Java multi-threading questions and answer](http://javarevisited.blogspot.sg/2011/07/java-multi-threading-interview.html) to improve your knowledge on Java Threads.

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

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

How to avoid deadlock in Java Threads

How to avoid deadlock in Java is one of the question which is flavor of the season for multithreading , asked more at a senior level and with lots of follow up questions , though question looks very basic but most of developer get stuck once you start going deep.

questions starts with "**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.

**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 (Integer.class) {

System.out.println("Aquired lock on Integer.class object");

synchronized (String.class) {

System.out.println("Aquired lock on String.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 thead 1 aquires 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.

now interviewer comes to final part , one of the most important in my view , **How to fix deadlock ? or How to avoid deadlock in Java ?**

if you have looked above code carefully you may have figured out that real reason for deadlock is not multiple threads but the way they access lock , if you provide an ordered access then problem will be resolved , here is

the fixed version.

public void method1() {

synchronized (Integer.class) {

System.out.println("Aquired lock on Integer.class object");

synchronized (String.class) {

System.out.println("Aquired lock on String.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");

}

}

}

Now there would not be any deadlock because both method is accessing lock on Integer and String object in same order . so if thread A acquires lock on Integer object , thread B will not proceed until thread A releases Integer lock , same way thread A will not be blocked even if thread B holds String lock because now thread B will not expect thread A to release Integer lock to proceed further.

------------------

Consistent lock acquisition ordering prevents deadlock indeed. But it is possible to have a deadlock even then. Not common scenario but plausible - [Deadlock despite consistent lock acquisition ordering](http://jaroslav-sedlacek.blogspot.com/2011/03/deadlock-despite-consistent-lock.html)

-------------------------------------------

Using this example there is a very bit chance to get deadlock! Just try this example :) You'll be running it for years to get a deadlock.

I'd like to suggest following sample that produces deadlock with 100% guarantee.

public class DeadLockTest {

static class ThreadOne implements Runnable {

public void run() {

synchronized (Integer.class) {

System.out.println(Thread.currentThread().getName() + " - Got lock on Integer.class");

synchronized (String.class) {

System.out.println(Thread.currentThread().getName() + " - Got lock on String.class");

}

}

}

}

static class ThreadTwo implements Runnable {

public void run() {

synchronized (String.class) {

System.out.println(Thread.currentThread().getName() + " - Got lock on String.class");

synchronized (Integer.class) {

System.out.println(Thread.currentThread().getName() + " - Got lock on Integer.class");

}

}

}

}

/\*\*

\* @param args

\*/

public static void main(String[] args) {

new Thread(new ThreadOne(), "ThreadOne").start();

new Thread(new ThreadTwo(), "ThreadTwo").start();

}

}You are free to use this sample in your article. Please, just mention my name - Arseny Kovalchuk.

Thanks for initial info that caused my to check this.

[October 19, 2011 at 4:41 AM](http://javarevisited.blogspot.com/2010/10/what-is-deadlock-in-java-how-to-fix-it.html?showComment=1319024504539#c1036188980134705297)

[Javin @ thread interview questions](http://javarevisited.blogspot.com/2011/07/java-multi-threading-interview.html) said...

Hi Arseny Kovalchuk, Thanks for your example but will you explain what is difference between earlier example and this for everybody's understanding , what I see run() method of your example is similar to method1, Also to get guaranteed deadlock one can use Sleep to hold one thread to hold the lock and increase deadlock chance , isn't it ?

[October 19, 2011 at 4:58 AM](http://javarevisited.blogspot.com/2010/10/what-is-deadlock-in-java-how-to-fix-it.html?showComment=1319025526299#c6335818845952851438)

Anonymous said...

@Javin, The main difference in my sample is that the synchronized block in the run methods are nested. And the major thing to get a deadlock is

1. threadOne acquires the lock on monitorOne

2. threadTwo acquires the lock on monitorTwo

3. threadOne wants to get lock on monitorTwo, but !mportant! it should not release lock on monitorOne, until it gets the lock on monitorTwo

4. at this time threadTwo wants to get lock on monitorOne, and it doesn't release lock on monitorTwo, until it gets lock on monitorOne.

That's the deadlock. In your sample there is a possibility for both threads to release the lock!

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

<http://javarevisited.blogspot.sg/2012/02/what-is-race-condition-in.html>

What is Race Condition in multithreading – 2 Examples in Java

**Race condition in Java** is a type of concurrency bug or issue which is introduced in your program because  parallel execution of your program by multiple threads at same time, Since Java is a multi-threaded programming language hence risk of Race condition is higher in Java which demands clear understanding of what causes a race condition and how to avoid that. Anyway Race conditions are just one of hazards or risk presented by  use of multi-threading in Java just like [deadlock in Java](http://javarevisited.blogspot.com/2010/10/what-is-deadlock-in-java-how-to-fix-it.html). **Race conditions** occurs when two thread operate on same object without proper synchronization and there operation interleaves on each other. Classical **example of Race condition** is incrementing a counter since increment is not an atomic operation and can be further divided into three steps like read, update and write. if two [threads](http://javarevisited.blogspot.com/2011/02/how-to-implement-thread-in-java.html) tries to increment count at same time and if they read same value because of interleaving of read operation of one thread to update operation of another thread, one count will be lost when one thread overwrite increment done by other thread. atomic operations are not subject to race conditions because those operation cannot be interleaved. This is also [a popular multi-threading interview questions](http://javarevisited.blogspot.com/2011/07/java-multi-threading-interview.html) during core java interviews. In this article we will see **how to find race condition in Java** and  two sample code patterns which often causes race conditions in Java.

## How to find Race Conditions in Java

**Finding Race conditions** in any language is most difficult job and Java is no different, though since readability of Java code is very good and synchronized constructs are well defined heaps to find race conditions by code review. finding race conditions by unit testing is not reliable due to random nature of race conditions. since race conditions surfaces only some time your unit test may passed without facing any race condition. only sure shot way to find race condition is reviewing code manually or using code review tools which can alert you on potential race conditions based on code pattern and use of synchronization in Java. I solely rely on [code review](http://javarevisited.blogspot.com/2011/09/code-review-checklist-best-practice.html) and yet to find a suitable *tool for exposing race condition in java*.

## Code Example of Race Condition in Java

Based on my experience in Java synchronization and where we use synchronized keyword I found that two code patterns namely "**check and act**" and "**read modify write**" can suffer race condition if not synchronized properly. both cases rely on natural assumption that a single line of code will be atomic and execute in one shot which is wrong e.g. ++ is not atomic.

**"Check and Act" race condition pattern**

classical example of "check and act" race condition in Java is getInstance() method of Singleton Class, remember that was one questions which we have discussed on 10 Interview questions on Singleton pattern in Java as "[How to write thread-safe Singleton in Java](http://javarevisited.blogspot.com/2011/03/10-interview-questions-on-singleton.html)". getInstace() method first check for whether instance is null and than initialized the instance and return to caller. Whole purpose of Singleton is that getInstance should always return same instance of Singleton. if you call getInstance() method from two thread simultaneously its possible that while one thread is initializing singleton after null check, another thread sees value of \_instance reference variable as null (quite possible in java) especially if your object takes longer time to initialize and enters into critical section which eventually results in getInstance() returning two separate instance of Singleton. This may not happen always because a fraction of delay may result in value of \_instance updated in main memory. here is a code example

public Singleton getInstance(){

if(\_instance == null){   //**race condition if two threads sees \_instance= null**

\_instance = new Singleton();

}

}

an easy way to fix "**check and ac**t" race conditions is to synchronized keyword and enforce locking which will make this operation atomic and guarantees that block or method will only be executed by one thread and result of operation will be visible to all threads once [synchronized blocks](http://javarevisited.blogspot.com/2011/04/synchronization-in-java-synchronized.html) completed or thread exited form synchronized block.

**read-modify-update race conditions**

This is another code pattern in Java which cause race condition, classical example is the non thread safe counter we discussed in [how to write thread safe class in Java](http://javarevisited.blogspot.com/2012/01/how-to-write-thread-safe-code-in-java.html). this is also a very popular multi-threading question where they ask you to find bugs on concurrent code. read-modify-update pattern also comes due to improper synchronization of **non-atomic operations** or combination of two individual atomic operations which is not atomic together e.g. put if absent scenario. consider below code

if(!hashtable.contains(key)){

hashtable.put(key,value);

}

here we only insert object into hashtable if its not already there. point is both contains() and put() are atomic but still this code can result in race condition since both operation together is not atomic. consider thread T1 checks for conditions and goes inside if block now CPU is switched from T1 to thread T2 which also checks condition and goes inside if block. now we have two thread inside if block which result in either T1 overwriting T2 value or vice-versa based on which thread has CPU for execution. In order to **fix this race condition in Java** you need to wrap this code inside synchronized block which makes them atomic together because no thread can go inside synchronized block if one thread is already there.

These are just some of *examples of race conditions in Java*, there will be numerous based on your business logic and code. best approach to find Race conditions is code review but its hard because thinking concurrently is not natural and we still assume code to run sequentially. Problem can become worse if JVM reorders code in absent of proper synchronization to gain performance benefit and this usually happens on production under heavily load, which is worst. I also suggest doing **load testing** in production like environment which many time helps to expose race conditions in java. Please share if you have faced any race condition in java projects.

Other **Java Thread Tutorial** you may like

[Difference between Runnable and Thread in java](http://javarevisited.blogspot.com/2012/01/difference-thread-vs-runnable-interface.html)

[How to Stop Thread in Java](http://javarevisited.blogspot.com/2011/10/how-to-stop-thread-java-example.html)

[Why wait and notify methods are declared in Object Class?](http://javarevisited.blogspot.com/2012/02/why-wait-notify-and-notifyall-is.html)

[How to Solve Producer Consumer Problem in Java](http://javarevisited.blogspot.com/2012/02/producer-consumer-design-pattern-with.html)

[Why wait and notify needs to called from Synchronized Context?](http://javarevisited.blogspot.com/2011/05/wait-notify-and-notifyall-in-java.html)

[Difference between ConcurrentHashMap and Hashtable in Java](http://javarevisited.blogspot.com/2011/04/difference-between-concurrenthashmap.html)

[Difference between invokeAndWait and InvokeLater in java Swing.](http://javarevisited.blogspot.com/2011/09/invokeandwait-invokelater-swing-example.html)

[Difference between wait and sleep in Java](http://javarevisited.blogspot.com/2011/12/difference-between-wait-sleep-yield.html)

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

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

**Java Synchronization Tutorial : What, How and Why?**

Multit-hreading and synchronization is a very important topic for any Java programmer. Good knowledge of multithreading, synchronization, and thread-safety can put you in front of other developers, at same time it's not easy to master these concept. In fact writing correct concurrent code is one of the hardest thing, even in Java, which has several inbuilt synchronization utilities. In thisJava synchronization tutorialwe will learn what is meaning of Synchronization in Java, Why do we need Synchronization in Java, What is java synchronized keyword, examples of using **Java synchronized method and blocks,** What can happen in multithreading code in absence of synchronized constructs, tips to avoid mistakes, while locking critical section in Java and some of important **points about synchronization in Java**. Since Java provides different constructs to provide synchronization and locking e.g. [volatile keyword](http://javarevisited.blogspot.sg/2011/06/volatile-keyword-java-example-tutorial.html), atomic variable, explicitly locking using java.util.concurrent.lock.Lock interface and there popular implementations e.g. ReentrantLock and ReentrantReadWriteLock, It becomes even more important to understand difference between synchronized and other constructs. Remember, clear understanding of synchronization is must to write correct concurrent code in Java, which is free of multithreading issues like deadlock, race conditions and thread-safety. I am sure, things learned in this **Java synchronization tutorial** will help. Once you gone through this article, You can further read Java Concurrency in Practice to develop your concept.  That's the one of those book which every Java developer must read.

## What is Synchronization in Java

Synchronization in Java is an important concept since Java is a multi-threaded language where multiple threads run in parallel to complete program execution. In multi-threaded environment *synchronization of Java object or synchronization of Java class becomes extremely important*. Synchronization in Java is possible by usingJava keywords ***"synchronized"*** and ***"volatile”***. Concurrent access of shared objects in Java introduces to kind of errors: thread interference and memory consistency errors and to avoid these errors you need to properly synchronize your Java object to allow mutual exclusive access of critical section to two threads. By the way This Java Synchronization tutorial is in continuation of my article [How HashMap works in Java](http://javarevisited.blogspot.com/2011/02/how-hashmap-works-in-java.html)  and [difference between HashMap and Hashtable in Java](http://javarevisited.blogspot.com/2010/10/difference-between-hashmap-and.html)  if you haven’t read already you may find some useful information based on my experience in Java Collections.

### Why do we need Synchronization in Java?

If your code is executing in multi-threaded environment, you need **syn**chronization for objects**,** which are shared among multiple threads, to avoid any corruption of state or any kind of unexpected behavior. Synchronization in Java will only be needed if shared object is mutable. if your shared object is either read only or [immutable object](http://javarevisited.blogspot.com/2013/03/how-to-create-immutable-class-object-java-example-tutorial.html), than you don't need synchronization, despite running multiple threads. Same is true with what threads are doing with object if all the threads are only reading value then you don't require synchronization in Java. JVM guarantees that ***Java synchronized code will only be executed by one thread at a time***. In Summary Java synchronized Keyword provides following functionality essential for concurrent programming :

1) synchronized keyword in Java provides locking, which ensures mutual exclusive access of shared resource and prevent data race.

2) synchronized keyword also prevent reordering of code statement by compiler which can cause subtle concurrent issue if we don't use synchronized or volatile keyword.

3) synchronized keyword involve locking and unlocking. before entering into **synchronized method or block** thread needs to acquire the lock, at this point it reads data from main memory than cache and when it release the lock, it flushes write operation into main memory which eliminates memory inconsistency errors.

### Synchronized keyword in Java

Prior to Java 1.5 synchronized keyword was only way to provide synchronization of shared object in Java. Any code written by using  **synchronized block** or enclosed inside **synchronized method** will be mutually exclusive, and can only be executed by one thread at a time. You can have both [static synchronized method and non static synchronized method](http://javarevisited.blogspot.sg/2012/03/mixing-static-and-non-static.html)and synchronized blocks in Java but we **can not** have **synchronized variable in java**. Using synchronized keyword with variable is illegal and will result in compilation error. Instead of synchronized variable in Java, you can have java volatile variable, which will instruct JVM threads to read value of volatile variable from main memory and don’t cache it locally. *Block synchronization in Java is preferred over method synchronization in Java* because by using block synchronization, you only need to lock the critical section of code instead of whole method. Since synchronization in Java comes with cost of performance, we need to synchronize only part of code which absolutely needs to be synchronized.

### Example of Synchronized Method in Java

Using **synchronized keyword** along with method is easy just apply synchronized keyword in front of method. What we need to take care is that static synchronized method locked on class object lock and non static synchronized method locks on current object (this). So it’s possible that both static and non static java synchronized method running in parallel.  This is the common mistake a naive developer do while **writing Java synchronized code**.

**public** **class** **Counter**{

**private** **static** **int** count = **0**;

**public** **static** **synchronized** **int** **getCount**(){

**return** count;

}

**public** synchoronized **setCount**(**int** count){

**this**.count = count;

}

}

In this example of *Java synchronization code is not properly synchronized* because both getCount() and setCount() are not getting locked on same object and can run in parallel which may results in incorrect count. Here getCount() will lock in Counter.class object while setCount() will lock on current object ([this](http://javarevisited.blogspot.com/2012/01/this-keyword-java-example-tutorial.html)). To make this code properly synchronized in Java *you need to either make both method static or non static or use java synchronized block instead of java synchronized method*.By the way this is one of the common mistake Java developers make while synchronizing their code.

### 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;

}

This is a classic example of [double checked locking in Singleton](http://javarevisited.blogspot.com/2014/05/double-checked-locking-on-singleton-in-java.html). In this **example of Java synchronized code,** we have made only critical section (part of code which is creating instance of singleton) synchronized and saved some performance. If you make whole method synchronized than every call of this method will be blocked, while you only need blocking to create singleton instance on first call. By the way, this is not the only way to write threadsafe singleton in Java. You can use Enum, or lazy loading to avoid thread-safety issue during instantiation. Even above code will not behave as expected because prior to Java 1.5, double checked locking was broker and even with volatile variable you can view half initialized object. Introduction of Java memory model and happens before guarantee in Java 5 solves this issue. To read more about [Singleton in Java](http://javarevisited.blogspot.com/2011/03/10-interview-questions-on-singleton.html) see that.

### Important points of synchronized keyword in Java

1. **Synchronized keyword in Java** is used to provide mutual exclusive access of a shared resource with multiple threads in Java. Synchronization in Java guarantees that, no two threads can execute a synchronized method which requires same lock simultaneously or concurrently.

2. You can use java synchronized keyword only on synchronized method or synchronized block.

3. When ever 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.

4. Java Thread acquires an **object level lock** when it enters into an instance synchronized java method and acquires a class level lock when it enters into static synchronized java method.

5. **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](http://javarevisited.blogspot.com/2011/02/how-to-implement-thread-in-java.html) which is holding lock can enter into that method without acquiring lock.

6. Java Synchronization will throw NullPointerException if object used in java synchronized block is nulle.g. synchronized (myInstance) will throws java.lang.NullPointerException if myInstance is null.

7. One Major **disadvantage of Java synchronized keyword** is that it doesn't allow concurrent read, which can potentially limit scalability. By using concept of lock stripping and using different locks for reading and writing, you can overcome this limitation of synchronized in Java. You will be glad to know that java.util.concurrent.locks.ReentrantReadWriteLock provides ready made implementation of ReadWriteLock in Java.

8. One more **limitation of java synchronized keyword** is that it can only be used to control access of shared object within the same JVM. If you have more than one JVM and need to synchronized access to a shared file system or database, the Java synchronized keyword is not at all sufficient. You need to implement a kind of global lock for that.

9. **Java synchronized keyword incurs performance cost.** Synchronized method in Java is very slow and can degrade performance. So use synchronization in java when it absolutely requires and consider using java synchronized block for synchronizing critical section only.

10. **Java synchronized block is better than java synchronized method** in Java because by using synchronized block you can only lock critical section of code and avoid locking whole method which can possibly degrade performance. A good example of java synchronization around this concept is [getInstance() method Singleton class](http://javarevisited.blogspot.com/2011/03/10-interview-questions-on-singleton.html). See here.

11. Its possible that both static synchronized and non static synchronized method can run simultaneously or concurrently because they lock on different object.

12. From java 5 after change in Java memory model **reads and writes are atomic for all variables declared using volatile keyword** (including long and double variables) and simple atomic variable access is more efficient instead of accessing these variables via synchronized java code. But it requires more care and attention from the programmer to avoid memory consistency errors.

13. **Java synchronized code could result in deadlock or starvation** while accessing by multiple thread if synchronization is not implemented correctly. To know [how to avoid deadlock in java](http://javarevisited.blogspot.com/2010/10/what-is-deadlock-in-java-how-to-fix-it.html) see here.

14. According to the Java language specification **you can not use Java synchronized keyword with constructor** it’s illegal and result in compilation error. So you can not synchronized constructor in Java which seems logical because other threads cannot see the object being created until the thread creating it has finished it.

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

16. Java.util.concurrent.locks extends capability provided by java synchronized keyword for writing more sophisticated programs since they offer more capabilities e.g. **Reentrancy** and **interruptible locks**.

17. Java synchronized keyword also synchronizes memory. In fact java synchronized synchronizes the whole of thread memory with main memory.

18. Important method related to synchronization in Java are wait(), notify() and notifyAll()which is defined in Object class. Do you know, why they are defined in java.lang.object class instead of java.lang.Thread? You can find [some reasons](http://javarevisited.blogspot.com/2012/02/why-wait-notify-and-notifyall-is.html), which make sense.

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);

}

any if you write synchronized code like above in java you may get warning **"Synchronization on non-final field"**  in IDE like Netbeans and InteliJ

20. 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.

Probably *most important point about Synchronization in Java* is that, in the absence of synchronized keyword or other construct e.g. volatile variable or atomic variable, compiler, JVM and hardware are free to make optimization, assumption, reordering or caching of code and data, which can cause subtle concurrency bugs in code. By introducing synchronization by using volatile, atomic variable or synchronized keyword, we instruct compiler and JVM to not to do that.

**Update 1**: Recently I have been reading several **Java Synchronization and Concurrency** articles in internet and I come across jeremymanson's blog which works in google and has worked on JSR 133 **Java Memory Mode**l, I would recommend some of this blog post for every java developer, he has covered certain details about concurrent programming , synchronization and volatility in simple and easy to understand language, here is the link [atomicity, visibility and ordering](http://jeremymanson.blogspot.com/2007/08/atomicity-visibility-and-ordering.html).

**Update 2:**  I am grateful to my readers, who has left some insightful comments on this post. They have shared lots of good information and experience and to provide them more exposure, I am including some of there comments on main article, to benefit new readers.

**@Vikas wrote**

Good comprehensive article about synchronized keyword in Java. to be honest I have never read all these details about synchronized block or method at one place. you may want to highlight some limitation of synchronized keyword in Java which is addressed by explicit locking using new concurrent package and Lock interface :

1. synchronized keyword doesn't allow separate locks for reading and writing. as we know that multiple thread can read without affecting [thread-safety of class](http://javarevisited.blogspot.sg/2012/01/how-to-write-thread-safe-code-in-java.html), synchronized keyword suffer performance due to contention in case of multiple reader and one or few writer.

 2. if one thread is waiting for lock then there is no way to time out, thread can wait indefinitely for lock.

3. on similar note if thread is waiting for lock to acquired there is no way to interrupt the thread.

All these limitation of synchronized keyword is addressed and resolved by using ReadWriteLock and ReentrantLock in Java 5.

**@George wrote**

Just my 2 cents on your great list of Java Synchronization facts and best practices :

1) synchronized keyword in internally implemented using two byte code instructions MonitorEnter and MonitorExit, this is generated by compiler. Compiler also ensures that there must be a MonitorExit for every MonitorEnter in different code path e.g. normal execution and abrupt execution, because of Exception.

2) java.util.concurrent package different locking mechanism than provided by synchronized keyword, they mostly used ReentrantLock, which internally use CAS operations, volatile variables and atomic variables to get better performance.

3) With synchronized keyword, you have to leave the lock, once you exist a synchronized method or block, there is no way you can take the lock to other method. java.util.concurrent.locks.ReentrantLock solves this problem by providing control for acquiring and releasing lock, which means you can acquire lock in method A and can release in method B, if they both needs to be locked in same object lock. Though this could be risky as compiler will neither check nor warn you about any accidental leak of locks. Which means, this can potentially block other threads, which are waiting for same lock.

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

5) Any thread trying to acquire lock using synchronized method will block indefinitely, until lock is available. Instead this, tryLock() method of java.util.concurrent.locks.ReentrantLock will not block if lock is not available.

Having said that, I must say, lots of good information.

### Recommend Books to learn Synchronization and Concurrency in Java

Synchronization and Concurrency is complex topic in Java and it's not easy to master them. Even more experienced Java developers struggle to write correct concurrent code in Java. I would highly recommend following Java books to master multi-threading, synchronization and Concurrency.

1. Java Concurrency in Practice By Brian Goeatz and team
2. Effective Java by Joshua Bloach
3. Java Threads By Scott Oaks and Henry Wong

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

<http://javarevisited.blogspot.sg/2012/03/mixing-static-and-non-static.html>

Mixing static and non static synchronized method - Java mistake 2

**Using static and non static synchronized method** for protecting shared resource is another Java mistake we are going to discuss in this part of  our series “learning from mistakes in Java”. In last article we have seen [why double and float should not be used for monetary calculation](http://javarevisited.blogspot.com/2012/02/java-mistake-1-using-float-and-double.html) , In this tutorial we will find out why using static and non static synchronized method together for protecting same shared resource is not advisable.

I have seen some times Java  programmer mix [static synchronized method](http://javarevisited.blogspot.com/2011/04/synchronization-in-java-synchronized.html) and instance synchronized method to protect same shared resource. They either don't know or failed to realize that **static synchronized** and **non static synchronized** method **lock on two different object** which breaks purpose of synchronizing shared resource as two thread can concurrently execute these two method breaking mutual exclusive access, which can corrupt status of mutable object or even cause subtle [race condition in Java](http://javarevisited.blogspot.com/2012/02/what-is-race-condition-in.html) or even more horrible [deadlock in java](http://javarevisited.blogspot.com/2010/10/what-is-deadlock-in-java-how-to-fix-it.html).

## Static and non static synchronized method Java

For those who are not familiar [static](http://javarevisited.blogspot.com/2011/11/static-keyword-method-variable-java.html) synchronized method locked on class object e.g. for string class its String.class while instance synchronized method locks on current instance of Object denoted by “[this” keyword in Java](http://javarevisited.blogspot.com/2012/01/this-keyword-java-example-tutorial.html). Since both of these object are different they have different lock so while one thread is executing static synchronized method , other [thread in java](http://javarevisited.blogspot.com/2011/02/how-to-implement-thread-in-java.html) doesn’t need to wait for that thread to return instead it will acquire separate lock denoted byte .class literal and enter into static synchronized method. This is even a [popular multi-threading interview questions](http://javarevisited.blogspot.com/2011/07/java-multi-threading-interview.html) where interviewer asked on which lock a particular method gets locked, some time also appear in Java test papers.

Bottom line is that  *never mix static and non static synchronized method for protecting same resource*.

## Example of Mixing instance and static synchronized methods

Here is an **example of multithreading code which is using static and non static synchronized method** to protect same shared resource:

**public** **class** SynchornizationMistakes {

**private** static int [count](http://www.php.net/count) = 0;

*//locking on this object lock*

**public** synchronized int getCount(){

        return [count](http://www.php.net/count);

    }

*//locking on .class object lock*

**public** static synchronized void increment(){

[count](http://www.php.net/count)++;

    }

}

here shared count is not accessed in mutual exclusive fashion which may result in passing incorrect count to caller of getCount() while another thread is incrementing count using static increment() method.

That’s all on this part of learning from mistakes in Java. Now we know that **static and non static synchronized method are locked on different locks** and should not be used to protect same shared object.

Other **Java thread tutorials** you may like:

[When to use Thread or Runnable interface in Java?](http://javarevisited.blogspot.com/2012/01/difference-thread-vs-runnable-interface.html)

[What does Volatile keyword do in Java](http://javarevisited.blogspot.com/2011/06/volatile-keyword-java-example-tutorial.html)

[How to Stop Thread in Java](http://javarevisited.blogspot.com/2011/10/how-to-stop-thread-java-example.html)

[How to write thread safe code in Java](http://javarevisited.blogspot.com/2012/01/how-to-write-thread-safe-code-in-java.html)

[Why wait and notify method are called from synchronized method or block](http://javarevisited.blogspot.com/2011/05/wait-notify-and-notifyall-in-java.html)

[Why wait and notify are defined in Object class](http://javarevisited.blogspot.com/2012/02/why-wait-notify-and-notifyall-is.html)

[Difference between wait and sleep in Java](http://javarevisited.blogspot.com/2011/12/difference-between-wait-sleep-yield.html)

[How to solve Producer consumer problem in Java using BlockingQueue](http://javarevisited.blogspot.com/2012/02/producer-consumer-design-pattern-with.html)

You might like:

* [java.sql.SQLException: No suitable driver found for jdbc:jtds:sqlserver - Cause and Solution](http://javarevisited.blogspot.com/2015/07/javasqlsqlexception-no-suitable-driver-found-jdbc.html)
* [How to differentiate between an Average and a Good Programmer?](http://javarevisited.blogspot.com/2015/05/how-to-differentiate-between-average.html)
* [Top 20 String Algorithm Questions from Coding Interviews](http://javarevisited.blogspot.com/2015/01/top-20-string-coding-interview-question-programming-interview.html)
* [Top 50 Java Thread Interview Questions Answers for Experienced](http://javarevisited.blogspot.com/2014/07/top-50-java-multithreading-interview-questions-answers.html)

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

<http://javarevisited.blogspot.sg/2011/07/java-multi-threading-interview.html>

Top 15 Java Multithreading, Concurrency Interview Questions Answers asked in Investment banks

**Thread interview questions Java**

Multi-threading and concurrency questions are essential part of any Java interview. If you are going for any Java interview on any Investment bank for equities front office position expect lots of **muti-threading interview questions** on your way. Multi-threading and concurrency is a favorite topics on Investment banking specially on electronic trading development and they grill candidate on many [confusing java thread interview questions](http://javarevisited.blogspot.sg/2011/04/top-20-core-java-interview-questions.html). They just want to ensure that the guy has solid knowledge of multi-threading and concurrent programming in Java  because most of them are in business of performance. High volume and low latency Electronic trading System which is used for Direct to Market (DMA) trading is usually concurrent in nature. These are my favorite thread interview questions on Java  asked on different on different time. I am not providing answer of these *thread interview questions* but I will give you hint whenever possible, some time hint is enough to answer. I will update the post further with detailed answers just like I did for [10 Singleton interview questions in Java](http://javarevisited.blogspot.sg/2011/03/10-interview-questions-on-singleton.html) recently.  With introduction of concurrency package in Java 5 questions on concurrent utility and concurrent collections are on rise as well. [ThreadLocal](http://javarevisited.blogspot.sg/2012/05/how-to-use-threadlocal-in-java-benefits.html), [BlockingQueue](http://javarevisited.blogspot.sg/2012/02/producer-consumer-design-pattern-with.html), [Counting Semaphore](http://javarevisited.blogspot.sg/2012/05/counting-semaphore-example-in-java-5.html) and [ConcurrentHashMap](http://javarevisited.blogspot.sg/2011/04/difference-between-concurrenthashmap.html) are popular among those.

## 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.

**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

**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.

**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.

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

This is my favorite java thread interview question because even though deadlock is quite common while writing multi-threaded concurrent program many candidates not able to write deadlock free code and they simply struggle. Just ask them you have n resources and n thread and to complete an operation you require all resources. 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.

This is my favorite java thread interview question because even though deadlock is quite common while writing multi-threaded concurrent program many candidates not able to write deadlock free code and they simply struggle. Just ask them you have n resources and n thread and to complete an operation you require all resources. 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 synchronization](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-ups based upon your answer and if you haven't master the particular thread topic it would be difficult to answer them*.*

**Related post:**

[Why wait, notify and notifyAll are defined in Object class in Java?](http://javarevisited.blogspot.sg/2012/02/why-wait-notify-and-notifyall-is.html)

[Difference between Runnable and Thread in Java](http://javarevisited.blogspot.sg/2012/01/difference-thread-vs-runnable-interface.html)

[Why wait and notify needs to be called from synchronized method?](http://javarevisited.blogspot.sg/2011/05/wait-notify-and-notifyall-in-java.html)

[invokeAndWait vs invokeLater in Java Swing](http://javarevisited.blogspot.sg/2011/09/invokeandwait-invokelater-swing-example.html)

[Why you should not mix static and non static synchronized method in Java](http://javarevisited.blogspot.sg/2012/03/mixing-static-and-non-static.html)

[How to stop Thread in Java](http://javarevisited.blogspot.sg/2011/10/how-to-stop-thread-java-example.html)

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

<http://javarevisited.blogspot.sg/2011/06/volatile-keyword-java-example-tutorial.html>

How Volatile in Java works? Example of volatile keyword in Java

**How to use Volatile keyword in Java**

What is volatile variable in Java and when to use Volatile variable in Java is famous multi-threading interview question in Java interviews. Though many programmer knows what is a volatile variable but they fail on second part i.e. where to use volatile variable in Java as its not common to have clear understanding and hands-on on volatile in Java. In this tutorial we will address this gap by providing simple example of volatile variable in Java and discussing some when to use Volatile variable in Java. Any way 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](http://javarevisited.blogspot.sg/2011/05/java-heap-space-memory-size-jvm.html). So if you want to share any variable in which read and write operation is atomic by implementation e.g. read and write in int or boolean variable you can declare them as volatile variable. From Java 5 along with major changes like Autoboxing, Enum, Generics and Variable arguments , Java introduces some change in Java Memory Model (JMM), Which guarantees visibility of changes made by one thread to another also as "happens-before" which solves the problem of memory writes that happen in one thread can "leak through" and be seen by another thread. Java volatile keyword cannot be used with method or class and it can only be used with variable. Java volatile keyword also guarantees visibility and ordering , after Java 5 write to any volatile variable happens before any read into volatile variable. By the way use of volatile keyword also prevents compiler or JVM from reordering of code or moving away them from [synchronization barrier](http://javarevisited.blogspot.sg/2011/04/synchronization-in-java-synchronized.html).

## Volatile variable Example in Java

To Understand example of volatile keyword in java let’s go back to [Singleton pattern in Java](http://javarevisited.blogspot.com/2011/03/10-interview-questions-on-singleton.html) and see [double checked locking in Singleton](http://javarevisited.blogspot.sg/2014/05/double-checked-locking-on-singleton-in-java.html) with Volatile and without volatile keyword in java.

/\*\*

\* Java program to demonstrate where to use Volatile keyword in Java.

\* In this example Singleton Instance is declared as volatile variable to ensure

\* every thread see updated value for \_instance.

\*

\* @author Javin Paul

\*/

**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;

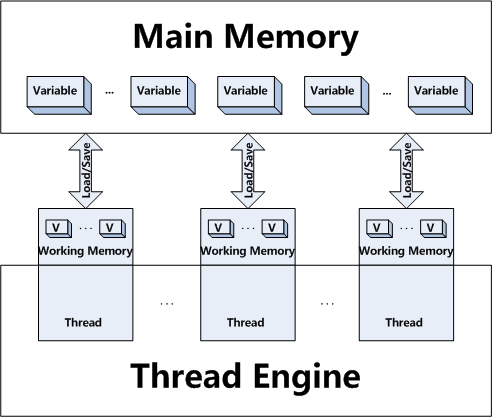
}

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

1) We are only creating instance one time

2) We are creating instance lazily at the time of first request comes.

If we do not make \_instance variable volatile then 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](http://javarevisited.blogspot.sg/2012/06/common-cause-of-javalangnullpointerexce.html).

Why? because reader threads are not doing any locking and until writer thread comes out of synchronized block, memory will not be synchronized and value of \_instance will not be updated in main memory. With Volatile keyword in Java this is handled by Java himself and such updates will be visible by all reader threads.

So in Summary apart from [synchronized keyword in Java](http://javarevisited.blogspot.com/2011/04/synchronization-in-java-synchronized.html), volatile keyword is also used to communicate content of memory between threads.

### Let’s see another example of volatile keyword in Java

most of the time while writing game we use a variable bExit to check whether user has pressed exit button or not, value of this variable is updated in [event thread](http://javarevisited.blogspot.sg/2011/09/invokeandwait-invokelater-swing-example.html) and checked in game thread , So if we don't use volatile keyword with this variable , Game Thread might miss update from event handler thread if its not synchronized in java already. volatile keyword in java guarantees that value of volatile variable will always be read from main memory and "*happens-before"* relationship in Java Memory model will ensure that content of memory will be communicated to different threads.

**private** **boolean** bExit**;**

**while(!**bExit**)** **{**

checkUserPosition**();**

updateUserPosition**();**

**}**

In this code example One Thread (Game Thread) can cache the value of "bExit" instead of getting it from [main memory](http://javarevisited.blogspot.sg/2011/05/java-heap-space-memory-size-jvm.html) every time and if in between any other thread (Event handler Thread) changes the value; it would not be visible to this thread. Making boolean variable "bExit" as volatile in java ensures this will not happen.

## When to use Volatile variable in Java

One of the most important thing in learning of volatile keyword is understanding when to use volatile variable in Java. Many [programmer](http://javarevisited.blogspot.sg/2011/06/top-programming-interview-questions.html) knows what is volatile variable and How does it work but they never really used volatile for any practical purpose. Here are couple of example to demonstrate when to use Volatile keyword in Java:

1) You can use Volatile variable if you want to read and write long and [double](http://javarevisited.blogspot.sg/2011/10/convert-double-to-string-example.html) variable atomically. long and double both are [64 bit](http://javarevisited.blogspot.sg/2012/01/find-jvm-is-32-or-64-bit-java-program.html) 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.

2) Volatile variable can be used as an alternative way of achieving [synchronization in Java](http://javarevisited.blogspot.sg/2011/04/synchronization-in-java-synchronized.html) in some cases, like Visibility. with volatile variable its guaranteed that all reader thread will see updated value of volatile variable once write operation completed, without volatile keyword different reader thread may see different values.

3) volatile variable can be used to inform compiler that a particular field is subject to be accessed by multiple threads, which will prevent compiler from doing any reordering or any kind of optimization which is not desirable in multi-threaded environment. Without volatile variable compiler can re-order code, free to cache value of volatile variable instead of always reading from main memory. like following example without volatile variable may result in [infinite loop](http://javarevisited.blogspot.sg/2011/12/how-to-traverse-or-loop-hashmap-in-java.html)

**private** **boolean** isActive **=** thread**;**

**public** **void** printMessage**(){**

**while(**isActive**){**

System**.**out**.**println**(**"Thread is Active"**);**

**}**

**}**

without *volatile modifier* its not guaranteed that one [Thread](http://javarevisited.blogspot.sg/2012/01/difference-thread-vs-runnable-interface.html) see the updated value of isActive from other thread. compiler is also free to cache value of isActive instead of reading it from main memory in every iteration. By making isActive a volatile variable you avoid these issue.

4) Another place where 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](http://javarevisited.blogspot.gr/2012/07/why-enum-singleton-are-better-in-java.html) that double checked locking was broken in Java 1.4 environment.

### Important points on Volatile keyword in Java

1. volatile keyword in Java is only application to variable and using volatile keyword with class and method is illegal.

2. volatile keyword in Java guarantees that value of volatile variable will always be read from main memory and not from Thread's local cache.

3. In Java reads and writes are [atomic](http://javarevisited.blogspot.sg/2012/02/what-is-race-condition-in.html) for all variables declared using Java volatile keyword (including long and double variables).

4. Using Volatile keyword in Java on variables reduces the risk of memory consistency errors, because any write to a volatile variable in Java establishes a happens-before relationship with subsequent reads of that same variable.

5. From Java 5 changes to a volatile variable are always visible to other threads. What’s more it also means that when a thread reads a volatile variable in Java, it sees not just the latest change to the volatile variable but also the side effects of the code that led up the change.

6. Reads and writes are atomic for reference variables are for most primitive variables (all types except long and double) even without use of volatile keyword in Java.

7. An access to a volatile variable in Java never has chance to block, since we are only doing a simple read or write, so unlike a synchronized block we will never hold on to any lock or wait for any [lock](http://javarevisited.blogspot.sg/2010/10/what-is-deadlock-in-java-how-to-fix-it.html).

8. Java volatile variable that is an object reference may be null.

9. Java volatile keyword doesn't means atomic, its common misconception that after declaring volatile ++ will be atomic, to make the operation atomic you still need to ensure exclusive access using synchronized method or block in Java.

10. If a variable is not shared between [multiple threads](http://javarevisited.blogspot.sg/2013/02/how-to-join-multiple-threads-in-java-example-tutorial.html) no need to use volatile keyword with that variable.

## Difference between synchronized and volatile keyword in Java

What is difference between volatile and synchronized is another popular core Java question asked in multi-threading and concurrency interviews. Remember volatile is not a replacement of synchronized keyword but can be used as an alternative in certain cases. Here are few differences between volatile and synchronized keyword in Java.

1. Volatile keyword in Java is a field modifier, while synchronized modifies code blocks and methods.

2. Synchronized obtains and releases lock on monitor’s Java volatile keyword doesn't require that.

3. Threads in Java can be blocked for waiting any monitor in case of synchronized, that is not the case with volatile keyword in Java.

4. Synchronized method affects performance more than volatile keyword in Java.

5. 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.

6. You can not synchronize on null object but your volatile variable in java could be null.

7. 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

In Summary volatile keyword in Java is not a replacement of synchronized block or method but in some situation is very handy and can save performance overhead which comes with [use of synchronization in Java](http://javarevisited.blogspot.com/2011/04/synchronization-in-java-synchronized.html)

if you like to know more about volatile I would also suggest to go thorough FAQ on Java Memory Model here which explains happens-before operations quite well.

Other J**ava concurrency tutorials** from Javarevisited you may like

1. [Difference between Runnable and Thread in Java](http://javarevisited.blogspot.sg/2012/01/difference-thread-vs-runnable-interface.html)
2. [How to use CyclicBarrier in Java with Example](http://javarevisited.blogspot.com/2012/07/cyclicbarrier-example-java-5-concurrency-tutorial.html)
3. [What is ConcurrentHashMap in Java](http://javarevisited.blogspot.sg/2011/04/difference-between-concurrenthashmap.html)
4. [How to use CountDownLatch in Java with Example](http://javarevisited.blogspot.com/2012/07/countdownlatch-example-in-java.html)
5. [How to solve producer consumer problem with BlockingQueue in Java](http://javarevisited.blogspot.sg/2012/02/producer-consumer-design-pattern-with.html)
6. [What is thread-safe class, How to write](http://javarevisited.blogspot.sg/2012/01/how-to-write-thread-safe-code-in-java.html)

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

<http://javarevisited.blogspot.sg/2012/03/difference-between-transient-and.html>

Difference between transient and volatile keyword in Java

Surprisingly "**Difference between transient and volatile keyword in Java**" has asked many times on [various java interview](http://javarevisited.blogspot.com/2011/04/top-20-core-java-interview-questions.html). volatile and transient are two completely different keywords from different areas of Java programming language. transient keyword is used during [serialization of Java object](http://javarevisited.blogspot.com/2011/04/top-10-java-serialization-interview.html) while volatile is related to visibility of variables modified by multiple thread during concurrent programming. Only similarity between volatile and transient is that they are less used or uncommon keywords and not as popular as public, [static](http://javarevisited.blogspot.com/2011/11/static-keyword-method-variable-java.html) or [final](http://javarevisited.blogspot.com/2011/12/final-variable-method-class-java.html). Anyway its good to know what transient keyword do in Java or how to use volatile keyword in Java. In this article we will couple of points between volatile and transient which can be used to answer this interview question.

This article is in continuation of earlier interview question on serialization like [difference between Serializable and Externaliable](http://javarevisited.blogspot.com/2012/01/serializable-externalizable-in-java.html)  and  [Top 10 Java serialization interview question](http://javarevisited.blogspot.com/2011/04/top-10-java-serialization-interview.html). If you haven’t read them already you may find them useful and interesting.

## Difference between volatile and transient keyword in Java

1) transient keyword is used along with [instance variables](http://javarevisited.blogspot.com/2012/02/difference-between-instance-class-and.html) to exclude them from serialization process. if a field  is transient its value will not be persisted. see my post [what is transient keyword in java](http://javarevisited.blogspot.com/2011/09/transient-keyword-variable-in-java.html) for more details. On the other hand volatile keyword can also be used in variables to indicate compiler and JVM that always read its value from main memory and follow happens-before relationship on visibility of volatile variable among multiple thread. see my post how and [when to use volatile keyword in Java](http://javarevisited.blogspot.com/2011/06/volatile-keyword-java-example-tutorial.html) for more details.

2) transient keyword can not be used along with static keyword but volatile can be used along with static.

3) transient variables are initialized with default value during de-serialization and there assignment or restoration of value has to be handled by application code.

That’s all on **difference between transient and volatile keyword in java**. As I said this interview question doesn’t really test you and just try to find whether you are familiar with those less known keywords in java or not. Let us know if you come across any other difference between volatile and transient keyword in java.

Other discussion on **Java interview questions** you may like

[Why Java doesn’t support multiple inheritance in Java](http://javarevisited.blogspot.com/2011/07/why-multiple-inheritances-are-not.html)

[Why wait and notify are defined in Object class](http://javarevisited.blogspot.com/2012/02/why-wait-notify-and-notifyall-is.html)

[Why character array is better than String for storing password in Java](http://javarevisited.blogspot.com/2012/03/why-character-array-is-better-than.html)

[Why implementing Runnable better than extending Thread class](http://javarevisited.blogspot.com/2012/01/difference-thread-vs-runnable-interface.html)

[Why main method is public static in Java](http://javarevisited.blogspot.com/2011/12/main-public-static-java-void-method-why.html)

[What is Marker interface and why do you used in Java](http://javarevisited.blogspot.com/2012/01/what-is-marker-interfaces-in-java-and.html)

* [20 Design pattern and Software design interview questions for Programmers](http://javarevisited.blogspot.com/2012/06/20-design-pattern-and-software-design.html)

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

<http://javarevisited.blogspot.sg/2012/01/how-to-write-thread-safe-code-in-java.html>

How to write Thread-Safe Code in Java

**thread-safety** or **thread-safe code in Java** refers to code which can safely be used or shared in concurrent or multi-threading environment and they will behave as expected. any code, class or object which can behave differently from its contract on concurrent environment is not thread-safe. thread-safety is one of the risk introduced by using [threads in Java](http://javarevisited.blogspot.com/2011/02/how-to-implement-thread-in-java.html) and I have seen java programmers and developers struggling to *write thread-safe code* or just understanding *what is thread-safe code* and what is not? This will not be very detailed article on thread-safety or low level details of [synchronization in Java](http://javarevisited.blogspot.com/2011/04/synchronization-in-java-synchronized.html) instead we will keep it simple and  focus on one example of non thread-safe code and try to understand what is thread-safety and **how to make an code thread-safe**.

## How to make Thread-Safe Code in Java

### Example of Non Thread Safe Code in Java

Here is an example of **non thread-safe code**, look at the code and find out *why this code is not thread safe* ?

// Non Thread-Safe Class in Java

public class Counter {

private int count;

/\*

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

\*/

public int getCount(){

return count++;

}

}

**Above example is not thread-safe** because ++ (increment operator) is not an **atomic operation** and can be broken down into read, update and write operation. if multiple thread call getCount() approximately same time each of these three operation may coincide or overlap with each other for example while thread 1 is updating value , thread 2 reads and still gets old value, which eventually let thread 2 override thread 1 increment and **one count is lost** because multiple thread called it concurrently.

### How to make code Thread-Safe in Java

There are multiple ways to make this code thread safe in Java:

1) Use [synchronized keyword in Java](http://javarevisited.blogspot.com/2011/04/synchronization-in-java-synchronized.html) and lock the getCount() method so that only one thread can execute it at a time which removes possibility of coinciding or interleaving.

2) use **Atomic Integer**, which makes this ++ operation atomic and since **atomic operations are thread-safe** and saves cost of external synchronization.

here is a thread-safe version of Counter class in Java:

/\*

\* Thread-Safe Example in Java

\*/

public class Counter {

private int count;

AtomicInteger atomicCount = new AtomicInteger( 0 );

/\*

\* This method thread-safe now because of locking and synchornization

\*/

public synchronized int getCount(){

return count++;

}

/\*

\* This method is thread-safe because count is incremented atomically

\*/

public int getCountAtomically(){

return atomicCount.incrementAndGet();

}

}

### 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 can not 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**.

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

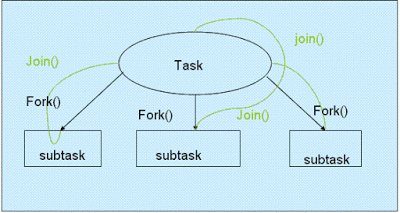
<http://javarevisited.blogspot.sg/2011/09/fork-join-task-java7-tutorial.html>

How to use Fork Join in Java 1.7 - Tutorial with Example

**What is fork Join** **framework in Java**: Already popular project coin of JDK7 release has presented lot of good feature e.g automatic resource management, string in switch case, better exception handling in JDK7 etc. On of other important feature to note is **fork join** as name implies it divide one task into several small task as a new fork means child and join all the fork when all the sub-tasks complete. Fork/join tasks is “pure” in-memory algorithms in which no I/O operations come into picture.it is based on a *work-stealing* algorithm. Concept of fork join would be much clear by following diagram.

### How fork join comes in existence:

Java’s most attractive part is it makes things easier and easier.for doing things faster java has given us concurrency concept but dealing with concurrency is not easy because we have to deal with [thread synchronization](http://javarevisited.blogspot.com/2011/04/synchronization-in-java-synchronized.html) and shared data. When we have to work with small piece of code it is easy to handle synchronization and atomicity, but it becomes for complex when code base and number of threads increased, its really challenging where several threads are working together to accomplish a large task so again java has tried to make things easy and simplifies this concurrency using **Executors and Thread Queue.**

When we compare **Executors** with old Thread it has made management of concurrent task very easy and it workon **divide and conquer** algorithm and create sub-tasks and communicate with each other to complete.But The problem with the executors framework is that a Callable is free to submit a new sub-task to its executor and wait for its result in a synchronous or asynchronous fashion. The issue is that of parallelism: When a Callable waits for the result of another Callable, it is put in a waiting state, and thus wasting an opportunity to handle another Callable queued for execution.

To solve this issue java 7 has given the concept of parallelism.New **fork-join** **framework** has been added in java.util.concurrent package.New fork-join executor framework has been created which is responsible for creating one new task object which is again responsible for creating new sub-task object and waiting for sub-task to be completed.internally it maintains a thread pool and executor assign pending task to this thread pool to complete when one task is waiting for another task to complete. whole Idea of fork-join framework is to leverage multiple processors of advanced machine.

## Fork Join framework in JDK7

**How to code using fork-join framework:**

Fork-join functionality is achieved by **ForkjoinTask object,** it has two method **fork() and join ()** Method.

* The **fork()** method allows  a new ForkJoinTask to be launched from an existing one.
* The **join()** method allows a ForkJoinTask to wait for the completion of another one.

Again **ForkjoinTask object** has been of two types**:** **RecursiveAction** and **RecursiveTask** which is more specialized form of this instance. While RecursiveAction represent executions that do not yield a return value, Instances of **RecursiveTask** yield return values.

I will try to add a suitable example for fork-join framework once I get some time, I am still looking for decent example which best suited fork join framework.

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

<http://javarevisited.blogspot.com.by/2011/09/invokeandwait-invokelater-swing-example.html>

InvokeLater and InvokeAndWait in Java Swing (an example tutorial)

Everyone who is doing programming in java swing has to come across **invokeAndWait and invokeLater** provided by SwingUtilites. In this java swing tutorial we will learn about both invokeLater() and invokeAndwait() method. In first part we will mostly focus on invokeLater and will find answers of questions like What is invokeLater, how to use **invokelater in java swing,** example of invokelater in swing etc while in second part of this invokeLater tutorial we will learn more about invokeAndWait method in java swing and will learn Why we need InvokeAndWait, how to use InvokeAndWait method in java Swing and differences between invokelater and invokeAndWait.

Finally we will see code example of both invokeLater and invokeAndWait in Swing and will be able to decide **when to use invokeLater** and **when to use invokeAndWait while doing Swing programming.**

We will also see [famous Swing interview questions](http://javarevisited.blogspot.com/2011/09/swing-interview-questions-answers-in.html) "difference between invokeLater and invokeAndWait" at end of article.

**Outline**

[Why do we need invokeLater method in Swing?](http://javarevisited.blogspot.com/2011/09/invokeandwait-invokelater-swing-example.html#_Why_do_we_need%20InvokeLater%20method%20i)

[What is invokeLater in SwingUtilities?](http://javarevisited.blogspot.com/2011/09/invokeandwait-invokelater-swing-example.html#_What_is_invokeLater_in%20Java%20Swing)

[How does invokeLater works in Swing](http://javarevisited.blogspot.com/2011/09/invokeandwait-invokelater-swing-example.html#_How_does_invokeLater_works%20in%20Java%20)

[Why do we need invokeAndWait method in Swing?](http://javarevisited.blogspot.com/2011/09/invokeandwait-invokelater-swing-example.html#_Why_invokeAndWait_Swing)

[How does invokeAndWait method works in SwingUtilities](http://javarevisited.blogspot.com/2011/09/invokeandwait-invokelater-swing-example.html#_How_does_InvokeAndWait_works%20in%20Jav)

[InvokeLater example in Swing](http://javarevisited.blogspot.com/2011/09/invokeandwait-invokelater-swing-example.html#_Example_of_InvokeLater_in%20Java%20Swin)

[InvokeAndWait example in Swing](http://javarevisited.blogspot.com/2011/09/invokeandwait-invokelater-swing-example.html#_Example_of_using_InvokeAndWait%20in%20J)

[Difference between InvokeLater and InvokeAndWait in Swing](http://javarevisited.blogspot.com/2011/09/invokeandwait-invokelater-swing-example.html#Difference_on_InvokeLater_vs%20Invoke)

### Why do we need InvokeLater method in Swing?

Before using invokelater or going deep about invokelater lets see why do we need this method in *swingutility* class? As we all know java swing is not threadsafe , you can not update swing component like JButton, JLable , JTable or JTree from any thread , they all needs to be updated from just one thread and we call it Event Dispatcher thread or EDT in short. Event Dispatcher thread is used to render graphics for java swing component and also process all events corresponding to key press, mouse click or any action. So if you want to update a particular swing component suppose label of a JButton from Yes to No you need to do this in **Event Dispatcher thread** and for doing this you need InvokeLater. invokeLater is used to perform any task asynchronously on AWT Event Dispatcher thread.

### [What is invokeLater in Java Swing](http://www.blogger.com/post-edit.g?blogID=8712770457197348465&postID=8345936409638384871&from=pencil)

Invokelater is a method in java on swing package and belongs to SwingUtilities class. Invokelater is used by java swing developer to update or perform any task on Event dispatcher thread asynchronously.invokeLater has been added into Java API from swing extension and it’s belong to SwingUtilities class.

### [How does invokeLater works in Java Swing](http://www.blogger.com/post-edit.g?blogID=8712770457197348465&postID=8345936409638384871&from=pencil)

If you see the signature of invokeLater method you will find that invokeLater takes a Runnable object and queues it to be processed by EventDispatcher thread. EDT thread will process this request only after sorting out all AWT pending events or requests. Even if invokeLater is called directly form Event dispatches thread processing of Runnable task still be done only after processing all pending AWT Events. An important point to note is that in case if run method of Runnable task throw any exception then AWT Event dispatcher thread will unwind and not the current thread.

### [Why do we need InvokeAndWait method in Swing](http://www.blogger.com/post-edit.g?blogID=8712770457197348465&postID=8345936409638384871&from=pencil)

As we know that **Swing is not thread-safe** and **we can not update the Swing component or GUI from any thread**. If you try to update GUI form any thread you will get unexpected result or exception, it could be your GUI might not be visible or simply disappered. Only method which is thread-safe in swing is **repaint() and revalidate().** On the other hand InvokeAndWait allows us to update the GUI from EDT thread synchronously. InvokeAndWait method also belongs to swingUtility class like invokeLater.

### [How does InvokeAndWait works in Java Swing](http://www.blogger.com/post-edit.g?blogID=8712770457197348465&postID=8345936409638384871&from=pencil)

If you look at the signature of **invokeAndWait** method you will see that it takes a **Runnable** object and run method of that Runnable is executed **synchronously on EDT**. This is a blocking call and wait until all pending AWT events gets processed and run() method completes. Its a preferred way of updating GUI form application thread.

Important point to note is that it should not be called from EventDispatcher thread unlike invokeLater; it’s an error because *it will result in guaranteed deadlock*. because if you cal invokeAndWait from EDT thread it will be an AWT event and caller of invokeAndWait will wait for EDT thread to complete and EDT will wait caller thread to be completed so they will be **locked in deadlock**. Its also important to remember that if run() mehtod of Runnable object throw exception then its caught in AWT EDT thread and rethrown as InvocationTargetException on caller thread.

### [Example of InvokeLater in Java Swing](http://www.blogger.com/post-edit.g?blogID=8712770457197348465&postID=8345936409638384871&from=pencil)

Here is an example of invokeLater() in Swing which will demonstrate that in case of invokeLater application thread doesn’t block.

Runnable pickHighBetaStock = **new** Runnable() {

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

System.*out*.println("High beta Stock picked by  " + Thread.*currentThread*());

}

};

SwingUtilities.*invokeLater*(pickHighBetaStock);

System.*out*.println("This might well be displayed before the other message. if Event-dispatcher thread is busy");

### [Example of using InvokeAndWait in Java Swing](http://www.blogger.com/post-edit.g?blogID=8712770457197348465&postID=8345936409638384871&from=pencil)

In this example of InvokeAndWait we will see that Application thread will block until Runnable object passed to EDT has been executed.

**final** Runnable pennyStockPicker = **new** Runnable() {

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

System.*out*.println("pick penny Stock on " + Thread.*currentThread*());

}

};

Thread stockPicker = **new** Thread() {

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

**try** {

SwingUtilities.*invokeAndWait*(pennyStockPicker);

}

**catch** (Exception e) {

e.printStackTrace();

}

System.*out*.println("This will finish after pennyStockPicker thread because InvokeAndWait is block call" + Thread.*currentThread*());

}

};

stockPicker.start();

### [Difference on InvokeLater vs InvokeAndWait in Swing](http://www.blogger.com/post-edit.g?blogID=8712770457197348465&postID=8345936409638384871&from=pencil)

Swingutilies provides us two methods for **performing any task in Event dispatcher thread**. Now let's see what *the difference between invokeLater and InvokeAndWait* is and when to use invokeLater.

1) InvokeLater is used to **perform task asynchronously in AWT Event dispatcher thread** while InvokeAndWait is used to perform task synchronously.

2) **InvokeLater is non blocking call while InvokeAndWait will block** until task is completed.

3) If run method of Runnable traget throws an Exception then *in case of invokeLater EDT threads unwinds while in case of invokeAndWait exception is caught* and rethrown as **InvocationTargetException**.

4) InvokeLater can be safely called from Event Dispatcher thread while if you call invokeAndWait from EDT thread you will get an error because as per java documentation of invokeAndWait it clearly says that "*this request will be processed only after all pending events*" and if you call this from EDT this will become one of pending event so its a deadlock because caller of InvokeAndWait is waing for completion of invokeAndWait while EDT is waiting for caller of InvokeAndWait.

5) **InvokeLater is more flexible** in terms of user interaction because it just adds the task in queue and allow user to interact with system while invokeAndWait is preffered way to update the GUI from application thread.

In Summary we can just say that since Swing is not thread-safe and we  cannot update different Swing GUI components on any thread other-than **Event dispatcher Thread** we need to use InvokeAndWait or InvokeLater to schedule any Runnable task for **AWT Event dispatcher Thread**. InvokeAndWait is synchronous and blocking call and wait until submitted Runnable completes while InvokeLater is asynchronous and non-blocking it will just submit task and exit."

That’s all on **InvokeAndWait() and InvokeLater()** method of **SwingUtilities** class. They are very important while doing GUI programming on Swing as well as this is very popular interview questions which I have discussed on my latest post on swing interview questions asked in Investment banks.

Please let me know your experience with *InvokeLater() and InvokeAndWait()* method and any issue you found while using them which would worth be mentioning here.

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