[1. What is Thread in Java? 3](#_Toc510797103)

[2. What is the difference between Thread and Process in Java? 3](#_Toc510797104)

[3. How do you implement Thread in Java? 3](#_Toc510797105)

[4. When to use Runnable vs Thread in Java? ( 5](#_Toc510797106)

[5. What is the difference between start() and run() method of Thread class?  ( 5](#_Toc510797107)

[6. What is the difference between Runnable and Callable in Java? ( 5](#_Toc510797108)

[7. What is the difference between CyclicBarrier and CountDownLatch in Java?  ( 6](#_Toc510797109)

[8. What is Java Memory model? 6](#_Toc510797110)

[9. What is volatile variable in Java? ( 6](#_Toc510797111)

[10. What is thread-safety? is Vector a thread-safe class? (Yes, see 7](#_Toc510797112)

[11. What is race condition in Java? Given one example? 7](#_Toc510797113)

[12. How to stop a thread in Java? ( 9](#_Toc510797114)

[13. What happens when an Exception occurs in a thread? 9](#_Toc510797115)

[14. How do you share data between two thread in Java? 9](#_Toc510797116)

[15. What is the difference between notify and notifyAll in Java? 13](#_Toc510797117)

[16. Why wait, notify and notifyAll are not inside thread class?  ( 17](#_Toc510797118)

[17. What is Thread Local variable in Java? 18](#_Toc510797119)

[18. What is FutureTask in Java? 18](#_Toc510797120)

[19. What is the difference between the interrupted() and isInterrupted() method in Java? 18](#_Toc510797121)

[20. Why wait and notify method are called from synchronized block? ( 18](#_Toc510797122)

[21. Why should you check condition for waiting in a loop? ( 19](#_Toc510797123)

[22. What is the difference between synchronized and concurrent collection in Java? 20](#_Toc510797124)

[23. What is the difference between Stack and Heap in Java? ( 20](#_Toc510797125)

[24. What is thread pool? Why should you thread pool in Java? 20](#_Toc510797126)

[25. Write code to solve Producer Consumer problem in Java? ( 21](#_Toc510797127)

[26. How do you avoid deadlock in Java? 22](#_Toc510797128)

[27. What is the difference between livelock and deadlock in Java? 24](#_Toc510797129)

[28. How do you check if a Thread holds a lock or not? 24](#_Toc510797130)

[29. How do you take thread dump in Java? ( 25](#_Toc510797131)

[30. Which JVM parameter is used to control stack size of a thread? 25](#_Toc510797132)

[31. What is the difference between synchronized and ReentrantLock in Java? 25](#_Toc510797133)

[32. How do you ensure sequence T1, T2, T3 threads in Java? 27](#_Toc510797134)

[33. Explain yield(), wait() and sleep() methods? 28](#_Toc510797135)

[34. What is the concurrency level of ConcurrentHashMap in Java? 30](#_Toc510797136)

[35. What is Semaphore in Java? 30](#_Toc510797137)

[36. What happens if you submit a task when the queue of the thread pool is already filled? 32](#_Toc510797138)

[37. What is the difference between the submit() and execute() method thread pool? 32](#_Toc510797139)

[38. What is blocking method in Java? 32](#_Toc510797140)

[39. What is the difference between invokeAndWait and invokeLater in Java? 33](#_Toc510797141)

[40. What is ReadWriteLock in Java? 33](#_Toc510797142)

[41. What is busy spin in multi-threading? ( 35](#_Toc510797143)

[42. What is the difference between the volatile and atomic variable in Java? 35](#_Toc510797144)

[43. What happens if a thread throws an Exception inside synchronized block? 36](#_Toc510797145)

[44. List down 3 multi-threading best practice you follow? ( 37](#_Toc510797146)

[45. How do you force to start a Thread in Java? 37](#_Toc510797147)

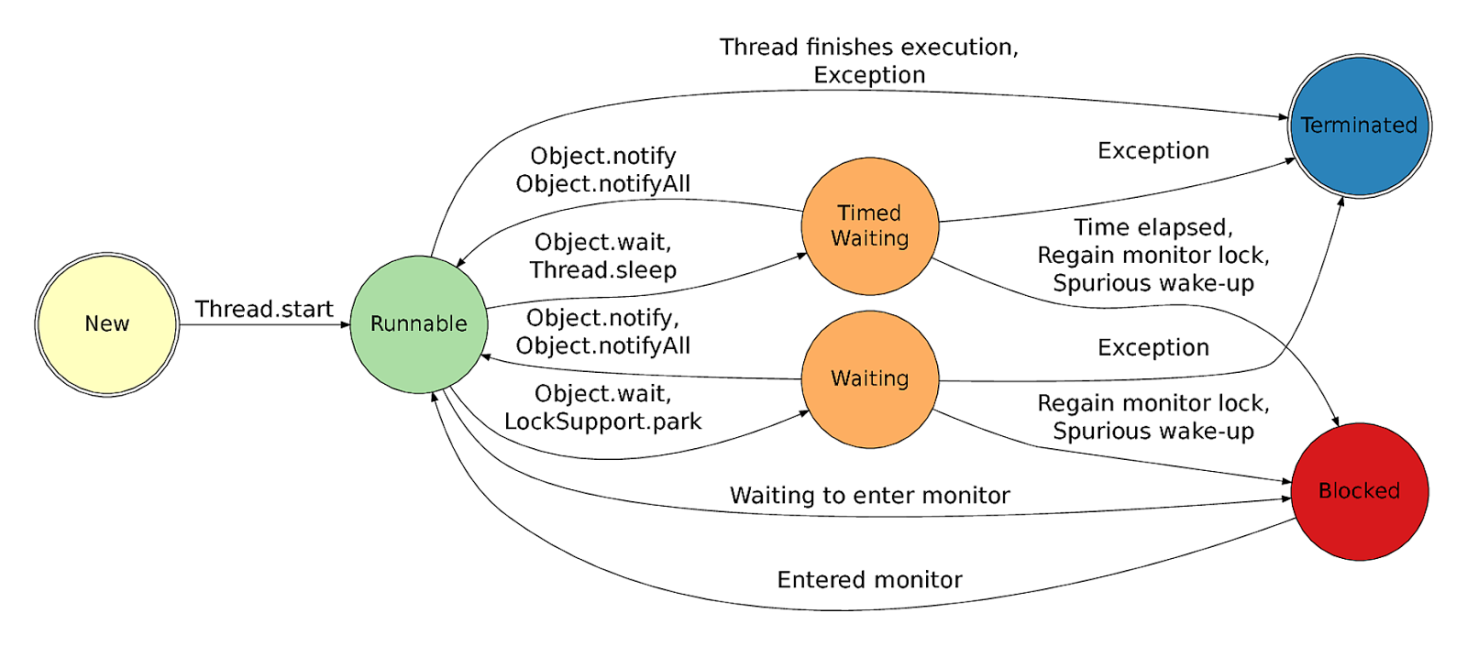
[46. What is the fork-join framework in Java? 37](#_Toc510797148)

1. What is Thread in Java?   
   The thread is an independent path of execution. It's way to take advantage of multiple CPU available in a machine. By employing multiple threads you can speed up CPU bound task. For example, if one thread takes 100 milliseconds to do a job, you can use 10 threads to reduce that task into 10 milliseconds. Java provides excellent support for multithreading at the language level, and it's also one of the strong selling points.
2. What is the difference between Thread and Process in Java?   
   The thread is a subset of Process, in other words, one process can contain multiple threads. Two process runs on different memory space, but all threads share same memory space. Don't confuse this with stack memory, which is different for the different thread and used to store local data to that thread. For more detail see the answer.

* Both process and [Thread](http://javarevisited.blogspot.de/2011/02/how-to-implement-thread-in-java.html) is independent path of execution but one process can have multiple Threads.
* Every process has its own memory space, executable code and a unique process identifier (PID) while every thread has its own stack in Java but it uses process main memory and shares it with other threads.
* Threads are also refereed as task or light weight process (LWP) in operating system.
* Threads from same process can communicate with each other by using Programming language construct like [wait and notify in Java](http://javarevisited.blogspot.sg/2011/05/wait-notify-and-notifyall-in-java.html) and much simpler than inter process communication.
* Another difference between Process and Thread in Java is that it's How Thread and process is created. It's easy to create Thread as compared to Process which requires duplication of parent process.
* All Threads which is part of same process share system resource like file descriptors , [Heap Memory](http://javarevisited.blogspot.de/2011/05/java-heap-space-memory-size-jvm.html) and other resource but each Thread has its own Exception handler and own stack in Java.

1. How do you implement Thread in Java?([answer](http://javarevisited.blogspot.com/2011/02/how-to-implement-thread-in-java.html))  
   At the language level, there are two ways to implement Thread in Java. An instance of java.lang.Thread represent a thread but it needs a task to execute, which is an instance of interface java.lang.Runnable. Since Thread class itself implement Runnable, you can override run () method either by extending Thread class or just implementing Runnable interface.

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. So now the interview question which *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 cannot 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 it’s 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.

[](http://4.bp.blogspot.com/-y4elKO9qfAU/VJ4uKPCV6tI/AAAAAAAACSA/UMMsH0VZpIo/s1600/Thread+in+Java.png)

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 up to thread scheduler to assign CPU to this thread. From here on it can either complete its execution or go to TERMINATED state or can go into WAITING, TIMED WAITING and BLOCKED state.

Main difference is that when program calls start() method a new Thread is created and code inside run() method is executed in new Thread while if you call run() method directly no new Thread is created and code inside run() will execute on current Thread. Most of the time calling run() is bug or [programming mistake](http://javarevisited.blogspot.com/2012/02/java-mistake-1-using-float-and-double.html) because caller has intention of calling start() to create new thread and this error can be detect by many static code coverage tools like findbugs. If you want to perform time consuming task than always call start() method otherwise your [main thread](http://javarevisited.blogspot.com/2011/12/main-public-static-java-void-method-why.html) will stuck while performing time consuming task if you call run() method directly. Another difference between start vs run in Java thread is that you cannot call start() method twice on thread object. once started, second call of start() will throw IllegalStateException in Java while you can call run() method twice.

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();

1. When to use Runnable vs Thread in Java? ([answer](http://javarevisited.blogspot.com/2012/01/difference-thread-vs-runnable-interface.html" \t "_blank))  
   This is a follow-up of previous multi-threading interview question. As we know we can implement thread either by extending Thread class or implementing Runnable interface, the question arise, which one is better and when to use one? This question will be easy to answer if you know that Java programming language doesn't support multiple inheritances of class, but it allows you to implement multiple interfaces. Which means, it’s better to implement Runnable then extends Thread if you also want to extend another class e.g. Canvas or Command Listener? For more points and discussion you can also refer this post.
2. What is the difference between start() and run() method of Thread class?  ([answer](http://javarevisited.blogspot.com/2012/03/difference-between-start-and-run-method.html" \t "_blank))  
   One of trick Java question from early days, but still good enough to differentiate between shallow understanding of Java threading model start() method is used to start newly created thread, while start() internally calls run() method, there is difference calling run() method directly. When you invoke run() as normal method, it’s called in the same thread, no new thread is started, which is the case when you call start() method. Read this answer for much more detailed discussion.
3. What is the difference between Runnable and Callable in Java? ([answer](http://java67.blogspot.com/2013/01/difference-between-callable-and-runnable-java.html" \t "_blank))  
   Both Runnable and Callable represent task which is intended to be executed in a separate thread. Runnable is there from JDK 1.0 while Callable was added on JDK 1.5. Main difference between these two is that Callable's call() method can return value and throw Exception, which was not possible with Runnable's run() method. Callable return Future object, which can hold the result of computation. See my [blog post](http://java67.blogspot.com/2013/01/difference-between-callable-and-runnable-java.html) on the same topic for a more in-depth answer to this question.

* Runnable interface is older than Callable, there from JDK 1.0, while Callable is added on Java 5.0.
* Runnable interface has run() method to define task while Callable interface uses call() method for task definition.
* run() method does not return any value, it's return type is void while call method returns value. Callable interface is a [generic parameterized interface](http://javarevisited.blogspot.sg/2012/08/how-to-write-parametrized-class-method-Generic-example.html) and Type of value is provided, when instance of Callable implementation is created.
* Another difference on run and call method is that run method cannot [throw](http://java67.blogspot.sg/2012/10/difference-between-throw-vs-throws-in.html) checked exception, while call method can throw checked exception in Java.

1. What is the difference between CyclicBarrier and CountDownLatch in Java?  ([answer](http://javarevisited.blogspot.com/2012/07/cyclicbarrier-example-java-5-concurrency-tutorial.html" \t "_blank))  
   Though both CyclicBarrier and CountDownLatch wait for number of threads on one or more events, the main difference between them is that you can not re-use CountDownLatch once count reaches to zero, but you can reuse same CyclicBarrier even after barrier is broken.  See this [answer](http://javarevisited.blogspot.com/2012/07/cyclicbarrier-example-java-5-concurrency-tutorial.html) for few more points and sample code example.
2. What is Java Memory model?   
   Java Memory model is set of rules and guidelines which allow Java programs to behave deterministically across multiple memory architecture, CPU, and operating system. It's particularly important in case of multi-threading. Java Memory Model provides some guarantee on which changes made by one thread should be visible to others, one of them is happens-before relationship. This relationship defines several rules which allow programmers to anticipate and reason behaviour of concurrent Java programs. For example, happens-before relationship guarantees :

Each action in a thread happens-before every action in that thread that comes later in the program order; this is known as program order rule.

An unlock on a monitor lock happens-before every subsequent lock on that same monitor lock, also known as Monitor lock rule.

A write to a volatile field happens-before every subsequent read of that same field, known as Volatile variable rule.

A call to Thread.start on a thread happens-before any other thread detects that thread has terminated, either by successfully return from Thread.join() or by Thread.isAlive() returning false, also known as Thread start rule.

A thread calling interrupt on another thread happens-before the interrupted thread detects the interrupt (either by having InterruptedException thrown, or invoking isInterrupted or interrupted), popularly known as Thread Interruption rule.

The end of a constructor for an object happens-before the start of the finalizer for that object, known as Finalizer rule.

If A happens-before B, and B happens-before C, then A happens-before C, which means happens-before guarantees Transitivity.

I strongly suggest reading [Chapter 16 of Java Concurrency in Practice](http://www.amazon.com/dp/0321349601/?tag=javamysqlanta-20) to understand Java Memory model in more detail.

1. What is volatile variable in Java? ([answer](http://java67.blogspot.com/2012/08/what-is-volatile-variable-in-java-when.html))  
   volatile is a special modifier, which can only be used with instance variables. In concurrent Java programs, changes made by multiple threads on instance variables are not visible to other in absence of any synchronizers e.g. synchronized keyword or locks. Volatile variable guarantees that a write will happen before any subsequent read: as stated: *"volatile variable rule"* in previous question. Read this answer to learn more about volatile variable and when to use them.

**What is Volatile variable in Java**  
volatile variable in Java is a special variable which is used to signal threads, compiler that this particular variables values is going to be updated by multiple thread inside Java application. By making a variable volatile using volatile keyword in Java, application programmer ensures that its value should always been read from [main memory](http://javarevisited.blogspot.sg/2011/05/java-heap-space-memory-size-jvm.html) and thread should not use cached value of that variable from their own stack. With the introduction of Java memory model from Java 5 onwards along with introduction of [CountDownLatch](http://javarevisited.blogspot.sg/2012/07/countdownlatch-example-in-java.html), [CyclicBarrier](http://javarevisited.blogspot.sg/2012/07/cyclicbarrier-example-java-5-concurrency-tutorial.html), [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), volatile variable also guarantees "happens-before" relationship, which means not only other thread has visibility of latest value of volatile variable but also all the variable seen by the thread which has updated value of volatile variable before this threads sees it. What is volatile variable and when to use it is always a [popular Java threading question](http://javarevisited.blogspot.sg/2011/07/java-multi-threading-interview.html).  
  
**Important point related to volatile keyword in Java**  
Since volatile keyword is used to make any variable volatile in Java environment, its good to know more about What is volatile keyword, what is its limitation and How to use volatile keyword in Java.  
  
1) Volatile keyword can only be applied to variable, it cannot be applied to class or method. using volatile keyword along with class and method is compiler error.  
  
2) volatile is also refereed as modifier in Java.  
  
**When to use Volatile variable in Java**  
This is the most important thing to learn while learning about volatile variable in Java. When to use volatile variable in Java is also a famous multi-threading interview question in Java. here are some of the scenario where you can use volatile variable in Java :  
  
1) Any variable which is shared between multiple threads should be made variable, in order to ensure that all thread must see latest value of volatile variable.  
  
2) A signal to compiler and JIT to ensure that compiler does not change ordering or volatile variable and moves them out of synchronized context.  
  
3) You want to save cost of synchronization as volatile variables are less expensive than synchronization.

1. What is thread-safety? is Vector a thread-safe class? (Yes, see [details](http://javarevisited.blogspot.sg/2011/09/difference-vector-vs-arraylist-in-java.html))  
   Thread-safety is a property of an object or code which guarantees that if executed or used by multiple threads in any manner e.g. read vs write it will behave as expected. For example, a thread-safe counter object will not miss any count if same instance of that counter is shared among multiple threads. Apparently, you can also divide collection classes in two categories, thread-safe and non-thread-safe. Vector is indeed a thread-safe class and it achieves thread-safety by synchronizing methods which modify state of Vector, on the other hand, its counterpart ArrayList is not thread-safe.
2. What is race condition in Java? Given one example?    
   Race conditions occur when two threads 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 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

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 then 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 it’s 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 causes 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);

}

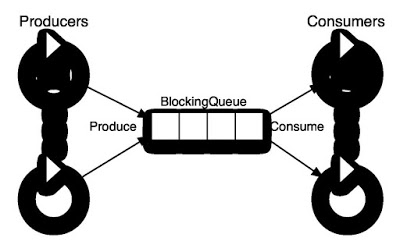
Here we only insert object into hashtable if its not already there. Point is both contains () and put () are atomic but still this code can result in race condition since both operation together is not atomic. Consider thread T1 checks for conditions and goes inside if block now CPU is switched from 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 it’s 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.

1. How to stop a thread in Java? ([answer](http://java67.blogspot.com/2015/07/how-to-stop-thread-in-java-example.html))  
   I always said that Java provides rich APIs for everything but ironically Java doesn't provide a sure shot way of stopping thread. There was some control methods in JDK 1.0 e.g. stop(), suspend() and resume() which was deprecated in later releases due to potential deadlock threats, from then Java API designers has not made any effort to provide a consistent, thread-safe and elegant way to stop threads. Programmers mainly rely on the fact that thread stops automatically as soon as they finish execution of run() or call() method. To manually stop, programmers either take advantage of volatile boolean variable and check in every iteration if run method has loops or interrupt threads to abruptly cancel tasks. See this [tutorial](http://javarevisited.blogspot.com/2011/10/how-to-stop-thread-java-example.html) for sample code of stopping thread in Java.
2. What happens when an Exception occurs in a thread?   
   This is one of the good [tricky Java question](http://java67.blogspot.sg/2012/09/top-10-tricky-java-interview-questions-answers.html) I have seen in interviews. In simple words, If not caught thread will die, if an uncaught exception handler is registered then it will get a call back. Thread.UncaughtExceptionHandler is an interface, defined as nested interface for handlers invoked when a Thread abruptly terminates due to an uncaught exception. When a thread is about to terminate due to an uncaught exception the Java Virtual Machine will query the thread for its UncaughtExceptionHandler using Thread.getUncaughtExceptionHandler() and will invoke the handler's uncaughtException() method, passing the thread and the exception as arguments.

An uncaught exception will cause the thread to exit. When it bubbles to the top of Thread.run() it will be handled by the Thread's [**UncaughtExceptionHandler**](http://download.oracle.com/javase/6/docs/api/java/lang/Thread.UncaughtExceptionHandler.html). By default, this will merely print the stack trace to the console. The thread itself will exit at this point - it couldn't really continue anyway, because its run() method has finished. So if you want the exception to be reraised in your main thread, you can define an UncaughtExceptionHandler that will do this (it's a very simple interface), and then call Thread.setUncaughtExceptionHandler on the spawned thread after it's created, passing in your custom exception handler.

1. How do you share data between two thread in Java?   
   You can share data between threads by using shared object, or concurrent data structure like BlockingQueue. See this tutorial to learn [inter-thread communication in Java](http://javarevisited.blogspot.sg/2013/12/inter-thread-communication-in-java-wait-notify-example.html). It implements Producer consumer pattern using wait and notify methods, which involves sharing objects between two threads.

[](https://2.bp.blogspot.com/-E9psXxo8Pjs/VuBAQ39X8JI/AAAAAAAAFCY/c__uJ2d3jwc/s1600/Inter+thread+Communication+in+Java.jpg)

Java Blocking Queue Example implementing Producer Consumer Problem

**java.util.concurrent.BlockingQueue** is a Queue that supports operations that wait for the queue to become non-empty when retrieving and removing an element, and wait for space to become available in the queue when adding an element.

Blocking Queue doesn’t accept null values and throw NullPointerException if you try to store null value in the queue.

Blocking Queue implementations are **thread-safe**. All queuing methods are atomic in nature and use internal locks or other forms of concurrency control.

Blocking Queue interface is part of java collections framework and it’s primarily used for implementing producer consumer problem. We don’t need to worry about waiting for the space to be available for producer or object to be available for consumer in BlockingQueue as it’s handled by implementation classes of BlockingQueue.

Java provides several BlockingQueue implementations such as ArrayBlockingQueue, LinkedBlockingQueue, PriorityBlockingQueue, SynchronousQueue etc.

While implementing producer consumer problem, we will use ArrayBlockingQueue implementation and following methods are important to know.

**put(E e)**: This method is used to insert elements to the queue, if the queue is full it waits for the space to be available.

**E take()**: This method retrieves and remove the element from the head of the queue, if queue is empty it waits for the element to be available.

Let’s implement producer consumer problem using BlockingQueue now.

Message

Just a normal java object that will be produced by Producer and added to the queue.

|  |  |
| --- | --- |
| Message.java | |
| 1  2  3  4  5  6  7  8  9  10 | package com.journaldev.concurrency;  public class Message {  private String msg;  public Message(String str){  this.msg=str;  }  public String getMsg() {  return msg;  }  } |

Producer

Producer class that will create messages and put it in the queue.

|  |  |
| --- | --- |
| Producer.java | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | package com.journaldev.concurrency;  import java.util.concurrent.BlockingQueue;  public class Producer implements Runnable {  private BlockingQueue<Message> queue;  public Producer(BlockingQueue<Message> q){  this.queue=q;  }  @Override  public void run() {  //produce messages  for(int i=0; i<100; i++){  Message msg = new Message(""+i);  try {  Thread.sleep(i);  queue.put(msg);  System.out.println("Produced "+msg.getMsg());  } catch (InterruptedException e) {  e.printStackTrace();  }  }  //adding exit message  Message msg = new Message("exit");  try {  queue.put(msg);  } catch (InterruptedException e) {  e.printStackTrace();  }  }  } |

Consumer

Consumer class that will process on the messages from the queue and terminates when exit message is received.

|  |  |
| --- | --- |
| Consumer.java | |
| 5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | package com.journaldev.concurrency;  import java.util.concurrent.BlockingQueue;  public class Consumer implements Runnable{  private BlockingQueue<Message> queue;  public Consumer(BlockingQueue<Message> q){  this.queue=q;  }  @Override  public void run() {  try{  Message msg;  //consuming messages until exit message is received  while((msg = queue.take()).getMsg() !="exit"){  Thread.sleep(10);  System.out.println("Consumed "+msg.getMsg());  }  }catch(InterruptedException e) {  e.printStackTrace();  }  }  } |

ProducerConsumerService

The producer consumer service will create the BlockingQueue with fixed size and it will be shared by both producers and consumers. This service will start producer and consumer threads and exit.

|  |  |
| --- | --- |
| ProducerConsumerService.java | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | package com.journaldev.concurrency;  import java.util.concurrent.ArrayBlockingQueue;  import java.util.concurrent.BlockingQueue;  public class ProducerConsumerService {  public static void main(String[] args) {  //Creating BlockingQueue of size 10  BlockingQueue<Message> queue = new ArrayBlockingQueue<>(10);  Producer producer = new Producer(queue);  Consumer consumer = new Consumer(queue);  //starting producer to produce messages in queue  new Thread(producer).start();  //starting consumer to consume messages from queue  new Thread(consumer).start();  System.out.println("Producer and Consumer has been started");  }  } |

Output of the above program is:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | Producer and Consumer has been started  Produced 0  Produced 1  Produced 2  Produced 3  Produced 4  Consumed 0  Produced 5  Consumed 1  Produced 6  Produced 7  Consumed 2  Produced 8  ... |

When you create a thread, you need an instance of Runnable. The easiest way to pass in a parameter would be to pass it in as an argument to the constructor:

class MyRunnableThread implements Runnable{

public String myParam;

public MyRunnableThread(String myParam){

this.myParam = myParam;

}

public void run() {

try{

System.out.println("Expl Thread: "+ myParam);

Thread.sleep(100);

} catch (InterruptedException iex) {

System.out.println("Exception in thread: "+iex.getMessage());

}

}

}

public class RunMyThread {

public static void main(String a[]){

System.out.println("Starting Main Thread...");

MyRunnableThread mrt = new MyRunnableThread("Gaurav");

Thread t = new Thread(mrt);

t.start();

}

}

OutPut : Starting Main Thread...

Expl Thread: **Gaurav**

1. What is the difference between notify and notifyAll in Java?

What is difference between notify and notifyAll method is one of the tricky Java question, which is easy to answer but once Interviewer ask followup questions, you either got confused or not able to provide clear cut and to the point answers. Main difference between notify and notifyAll is that notify method will only notify one Thread and notifyAll method will notify all Threads which are waiting on that monitor or lock. By the way this is something you have been reading in all over places and to be frank, this statement despite being correct is not complete and its very difficult to understand difference between notify vs notifyAll by just reading this statement. Lot of questions comes in mind like

Which thread will be notified if I use notify()?

How do I know how many threads are waiting, so that I can use notifyAll() ?

How to call notify()?

What are these thread waiting for being notified etc.

Actually discussion of notify and notifyAll is incomplete without discussing wait method in Java and I had touched based on this on my earlier article why wait and notify must be called from synchronized context. In order to get answer of those questions and understand difference between notify and notifyAll we will use a simple Java Thread example using wait and notify code :

**Difference between notify and notifyAll in Java**

Java provides two methods notify and notifyAll for waking up threads waiting on some condition and you can use any of them but there is subtle difference between notify and notifyAll in Java which makes it one of the popular multi-threading interview question in Java. When you call notify only one of waiting thread will be woken and its not guaranteed which thread will be woken, it depends upon Thread scheduler. While if you call notifyAll method, all threads waiting on that lock will be woken up, but again all woken thread will fight for lock before executing remaining code and that's why wait is called on loop because if multiple threads are woken up, the thread which will get lock will first execute and it may reset waiting condition, which will force subsequent threads to wait. So key difference between notify and notifyAll is that notify() will cause only one thread to wake up while notifyAll method will make all thread to wake up.

**When to use notify and notifyAll in Java**

This is the follow-up question if you get pass the earlier one Difference between notifyAll and notify in Java. If you understand notify vs notifyAll then you can answer this by applying little common sense. You can use notify over notifyAll if all thread are waiting for same condition and only one Thread at a time can benefit from condition becoming true. In this case notify is optimized call over notifyAll because waking up all of them because we know that only one thread will benefit and all other will wait again, so calling notifyAll method is just waste of cpu cycles. Though this looks quite reasonable there is still a caveat that unintended recipient swallowing critical notification. by using notifyAll we ensure that all recipient will get notify. Josh bloach has explained this in good detail in his book Effective Java , I highly recommend this book if you haven't read them already. Another one you can try is Concurrency Practice in Java and Java Thread, which discusses wait and notify methods in good details.

**Example of notify and notifyAll method in Java**

I have put together an example to show how all threads gets notified when we call notifyAll method in Java and just one Thread will wake up when we call notify method in Java. In this example three threads will wait if boolean variable go is false, remember boolean go isa volatile variable, so that all threads will see its updated value. Initially three threads WT1, WT2, WT3 will wait because variable go is false than one thread NT1 will make go true and notify all threads by calling notifyAll method or notify just one thread by calling notify() method. In case of notify() call there is no guarantee which thread will woke up and you can see it by running this Java program multiple times. In case of notifyAll all thread will woke up but they will compete for monitor or lock and the Thread which will get the lock first will finish its execution and resetting go to false which will force other two threads still waiting. At the end of this program you will have two threads waiting and two threads including notification thread finished. Program will not terminate because other two threads are still waiting and they are not daemon threads. Purpose of this notify and notifyAll example is to show you How to use them and How notify and notifyAll method works in Java.

**Code Example of notify and notifyAll**

Here is complete code example of How to use notify and notifyAll method in Java. We have already explained when to use notify vs notifyAll method and this example will clarify effect of calling notify and notifyAll method in Java.

**import** java.util.logging.Level;  
**import** java.util.logging.Logger;  
/\*\*  
\* **Java program to demonstrate How to use notify and notifyAll method in Java** and

\* How notify and notifyAll method notifies thread, which thread gets woke up etc.  
\*/  
**public** **class** NotificationTest {  
**private** **volatile** **boolean**go = **false**;  
**public** **static** **void**main(**String** args[]) **throwsInterruptedException** {  
**final** NotificationTest test = **new** NotificationTest();  
  
**Runnable** waitTask = **new** **Runnable**(){  
@**Override**  
**publicvoid** run(){  
**try**{  
test.shouldGo();  
}

**catch** (**InterruptedException** ex) {  
**Logger**.getLogger(NotificationTest.**class**.getName()).

log(**Level**.SEVERE, **null**, ex);  
}  
**System**.out.println(**Thread**.currentThread() + " finished Execution");  
}  
};  
  
**Runnable** notifyTask = **new** **Runnable**(){  
@**Override**  
**publicvoid** run(){  
test.go();  
**System**.out.println(**Thread**.currentThread() + " finished Execution");  
}  
};  
  
**Thread** t1 = **newThread**(waitTask, "WT1"); *//will wait*  
**Thread** t2 = **newThread**(waitTask, "WT2"); *//will wait*  
**Thread** t3 = **newThread**(waitTask, "WT3"); *//will wait*  
**Thread** t4 = **newThread**(notifyTask,"NT1"); *//will notify*  
  
*//starting all waiting thread*  
t1.start();  
t2.start();  
t3.start();  
  
*//pause to ensure all waiting thread started successfully*  
**Thread**.sleep(200);  
*//starting notifying thread*  
t4.start();  
}  
*/\*  
\* wait and notify can only be called from synchronized method or bock  
\*/*  
**private** **synchronized** **void**shouldGo() **throwsInterruptedException** {  
while(go != **true**){  
**System**.out.println(**Thread**.currentThread()

+ " is going to wait on this object");  
wait(); *//release lock and reacquires on wakeup*  
**System**.out.println(**Thread**.currentThread() + " is woken up");  
}  
go = **false**;*//resetting condition*  
}  
*/\*  
\* both shouldGo() and go() are locked on current object referenced by "this" keyword  
\*/*  
**privatesynchronized** **void**go() {  
while (go == **false**){  
**System**.out.println(**Thread**.currentThread()

+ " is going to notify all or one thread waiting on this object");  
go = **true**; *//making condition true for waiting thread*  
*//notify(); // only one out of three waiting thread WT1, WT2,WT3 will woke up*  
notifyAll(); *// all waiting thread WT1, WT2,WT3 will woke up*  
}  
}  
}  
  
**Output in caseof using notify**  
**Thread**[WT1,5,main] is going to wait on **this**object  
**Thread**[WT3,5,main] is going to wait on **this**object  
**Thread**[WT2,5,main] is going to wait on **this**object  
**Thread**[NT1,5,main] is going to notify all or one thread waiting on **this** object  
**Thread**[WT1,5,main] is woken up  
**Thread**[NT1,5,main] finished Execution  
**Thread**[WT1,5,main] finished Execution  
  
**Output in caseof calling notifyAll**  
**Thread**[WT1,5,main] is going to wait on **this**object  
**Thread**[WT3,5,main] is going to wait on **this**object  
**Thread**[WT2,5,main] is going to wait on **this**object  
**Thread**[NT1,5,main] is going to notify all or one thread waiting on **this** object  
**Thread**[WT2,5,main] is woken up  
**Thread**[NT1,5,main] finished Execution  
**Thread**[WT3,5,main] is woken up  
**Thread**[WT3,5,main] is going to wait on **this**object  
**Thread**[WT2,5,main] finished Execution  
**Thread**[WT1,5,main] is woken up  
**Thread**[WT1,5,main] is going to wait on **this**object

1. Why wait, notify and notifyAll are not inside thread class?  ([answer](http://java67.blogspot.com/2013/03/difference-between-wait-vs-notify-vs-notifyAll-java-thread.html))  
   This is a design related question, which checks what candidate thinks about existing system or does he ever thought of something which is so common but looks in-appropriate at first. In order to answer this question, you have to give some reasons why it makes sense for these three methods to be in Object class, and why not on Thread class. One reason which is obvious is that Java provides lock at object level not at thread level. Every object has lock, which is acquired by thread. Now if thread needs to wait for certain lock it make sense to call wait() on that object rather than on that thread. Had wait() method declared on Thread class, it was not clear that for which lock thread was waiting. In short, since wait, notify and notifyAll operate at lock level, it makes sense to defined it on object class because lock belongs to object. You can also see this [article](http://javarevisited.blogspot.sg/2012/02/why-wait-notify-and-notifyall-is.html) for more elaborate answer of this question.
2. What is Thread Local variable in Java?    
   ThreadLocal variables are special kind of variable available to Java programmer. Just like instance variable is per instance, ThreadLocal variable is per thread. It's a nice way to achieve thread-safety of expensive-to-create objects, for example you can make SimpleDateFormat thread-safe using ThreadLocal. Since that class is expensive, its not good to use it in local scope, which requires separate instance on each invocation. By providing each thread their own copy, you shoot two birds with one arrow. First, you reduce number of instance of expensive object by reusing fixed number of instances, and Second, you achieve thread-safety without paying cost of synchronization or immutability. Another good example of thread local variable is ThreadLocalRandom class, which reduces number of instances of expensive-to-create Random object in multi-threading environment. See this [answer](http://javarevisited.blogspot.sg/2012/05/how-to-use-threadlocal-in-java-benefits.html) to learn more about thread local variables in Java.
3. What is FutureTask in Java?   
   FutureTask represents a cancellable asynchronous computation in concurrent Java application. This class provides a base implementation of Future, with methods to start and cancel a computation, query to see if the computation is complete, and retrieve the result of the computation. The result can only be retrieved when the computation has completed; the get methods will block if the computation has not yet completed. A FutureTask object can be used to wrap a Callable or Runnable object. Since FutureTask also implements Runnable, it can be submitted to an Executor for execution.
4. What is the difference between the interrupted() and isInterrupted() method in Java?   
   Main difference between interrupted() and isInterrupted() is that former clears the interrupt status while later does not. The interrupt mechanism in Java multi-threading is implemented using an internal flag known as the interrupt status. Interrupting a thread by calling Thread.interrupt() sets this flag. When interrupted thread checks for an interrupt by invoking the [static method](http://java67.blogspot.com/2012/11/what-is-static-class-variable-method.html)Thread.interrupted(), interrupt status is cleared. The non-static isInterrupted() method, which is used by one thread to query the interrupt status of another, does not change the interrupt status flag. By convention, any method that exits by throwing an InterruptedException clears interrupt status when it does so. However, it's always possible that interrupt status will immediately be set again, by another thread invoking interrupt
5. Why wait and notify method are called from synchronized block? ([answer](http://javarevisited.blogspot.com/2011/05/wait-notify-and-notifyall-in-java.html))  
   Main reasons for calling wait and notify method from either synchronized block or method is that it made mandatory by Java API. If you don't call them from synchronized context, your code will throw IllegalMonitorStateException. A more subtle reason is to avoid the race condition between wait and notify calls. To learn more about this, check my similarly titled post [here](http://javarevisited.blogspot.com/2011/05/wait-notify-and-notifyall-in-java.html).

We use wait(), notify(), or notifyAll() method mostly for [inter-thread communication](http://javarevisited.blogspot.sg/2013/12/inter-thread-communication-in-java-wait-notify-example.html) in Java. One thread is waiting after checking a condition e.g. In the classic Producer-Consumer problem, the Producer thread waits if the buffer is full and Consumer thread notify Producer thread after it creates a space in the buffer by consuming an element.

Calling [notify() or notifyAll()](http://java67.blogspot.com/2013/03/difference-between-wait-vs-notify-vs-notifyAll-java-thread.html) methods issues a notification to a single or multiple thread that a condition has changed and once notification thread leaves synchronized block, all the threads which are waiting fight for object lock on which they are waiting and lucky **thread returns from wait() method after reacquiring the lock** and proceed further.  
  
Let’s divide this whole operation into steps to see a possibility of *race condition between wait() and notify() method in Java*, we will use [Produce Consumer thread example](http://javarevisited.blogspot.com/2015/06/java-lock-and-condition-example-producer-consumer.html) to understand the scenario better:

* The Producer thread tests the condition (buffer is full or not) and confirms that it must wait (after finding buffer is full).
* The Consumer thread sets the condition after consuming an element from a buffer.
* The Consumer thread calls the notify () method; this goes unheard since the Producer thread is not yet waiting.
* The Producer thread calls the wait () method and goes into waiting state.

So due to [race condition](http://javarevisited.blogspot.com/2012/02/what-is-race-condition-in.html) 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.

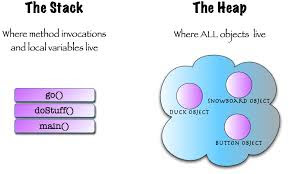
Now let's think how does this potential race condition get resolved? This race condition is resolved by using [synchronized keyword](http://javarevisited.blogspot.com/2011/04/synchronization-in-java-synchronized.html) and locking provided by Java. In order to call the **wait (), notify () or notifyAll () methods in Java**, we must have obtained the lock for the object on which we're calling the method.  
  
Since the wait() method in Java also releases the lock prior to waiting and reacquires the lock prior to returning from the [wait() method](http://javarevisited.blogspot.com/2011/12/difference-between-wait-sleep-yield.html), 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 to avoid:

* **IllegalMonitorStateException in Java** which will occur if we don't call wait (), notify () or notifyAll () method from synchronized context.
* **Any potential race condition between wait and notify method in Java**.

1. Why should you check condition for waiting in a loop? ([answer](http://javarevisited.blogspot.com/2015/07/how-to-use-wait-notify-and-notifyall-in.html))  
   Its possible for a waiting thread to receive false alerts and spurious wake up calls, if it doesn't check the waiting condition in loop, it will simply exit even if condition is not met. As such, when a waiting thread wakes up, it cannot assume that the state it was waiting for is still valid. It may have been valid in the past, but the state may have been changed after the notify() method was called and before the waiting thread woke up. That's why it always better to call wait() method from loop, you can even create template for calling wait and notify in Eclipse. To learn more about this question, I would recommend you to read Effective Java items on thread and synchronization.
2. What is the difference between synchronized and concurrent collection in Java?   
   Though both synchronized and concurrent collection provides thread-safe collection suitable for multi-threaded and concurrent access, later is more scalable than former. Before Java 1.5, Java programmers only had synchronized collection which becomes source of contention if multiple thread access them concurrently, which hampers scalability of system. Java 5 introduced concurrent collections like ConcurrentHashMap, which not only provides thread-safety but also improves scalability by using modern techniques like lock stripping and partitioning internal table. See this [answer](http://javarevisited.blogspot.com/2010/10/what-is-difference-between-synchronized.html) for more differences between synchronized and concurrent collection in Java.
3. What is the difference between Stack and Heap in Java? ([answer](http://java67.blogspot.com/2014/07/21-frequently-asked-java-interview-questions-answers.html))  
   Why does someone this question as part of multi-threading and concurrency? Because Stack is a memory area which is closely associated with threads. To answer this question, both stack and heap are specific memories in Java application. Each thread has their own stack, which is used to store local variables, method parameters and call stack. Variable stored in one Thread's stack is not visible to other. On another hand, the heap is a common memory area which is shared by all threads. Objects whether local or at any level is created inside heap. To improve performance thread tends to cache values from heap into their stack, which can create problems if that variable is modified by more than one thread, this is where volatile variables come into the picture. volatile suggest threads read the value of variable always from main memory. See this [article](http://javarevisited.blogspot.com/2013/01/difference-between-stack-and-heap-java.html) for learning more about stack and heap in Java to answer this question in greater detail.

[](https://3.bp.blogspot.com/-vJvHCwr7ozY/VuBB4nlNpkI/AAAAAAAAFCk/8mqWs5unUK4/s1600/Heap+vs+Stack+in+Java.jpg)

1. What is thread pool? Why should you thread pool in Java?    
   Creating thread is expensive in terms of time and resource. If you create thread at time of request processing it will slow down your response time, also there is only a limited number of threads a process can create. To avoid both of these issues, a pool of thread is created when application starts-up and threads are reused for request processing. This pool of thread is known as "thread pool" and threads are known as worker thread. From JDK 1.5 release, Java API provides Executor framework, which allows you to create different types of thread pools e.g. single thread pool, which process one task at a time, fixed thread pool (a pool of fixed number of threads) or cached thread pool (an expandable thread pool suitable for applications with many short lived tasks). See this [article](http://javarevisited.blogspot.com/2013/07/how-to-create-thread-pools-in-java-executors-framework-example-tutorial.html) to learn more about thread pools in Java to prepare detailed answer of this question.

**Benefits of Thread Pool in Java**

Thread Pool offers several benefit to Java application, biggest of them is separating submission of task to execution of task ,which result if more loose coupled and flexible design than tightly coupled create and execute pattern. Here are some more benefits of using Thread pool in Java:

* Use of Thread Pool reduces response time by avoiding thread creation during request or task processing.
* Use of Thread Pool allows you to change your execution policy as you need. You can go from [single thread](http://javarevisited.blogspot.sg/2011/02/how-to-implement-thread-in-java.html) to multiple thread by just replacing ExecutorService implementation.
* Thread Pool in Java application increases stability of system by creating a configured number of threads decided based on system load and available resource.
* Thread Pool frees application developer from thread management stuff and allows focusing on business logic.

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

public class ThreadPoolExample {

public static void main(String args[]) {

ExecutorService service = Executors.newFixedThreadPool(3);

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

service.submit(new Task(i));

}

service.shutdown();

}

}

final class Task implements Runnable{

private int taskId;

public Task(int id){

this.taskId = id;

}

@Override

public void run() {

System.out.println("Task ID : " + this.taskId +" performed by " + Thread.currentThread().getName());

}

}

1. Write code to solve Producer Consumer problem in Java? ([answer](http://java67.blogspot.com/2015/12/producer-consumer-solution-using-blocking-queue-java.html))  
   Most of the threading problem you solved in the real world is of the category of Producer consumer pattern, where one thread is producing task and another thread is consuming that. You must know how to do inter thread communication to solve this problem. At the lowest level, you can use wait and notify to solve this problem, and at a high level, you can leverage Semaphore or BlockingQueue to implement Producer consumer pattern, as shown in this [tutorial](http://javarevisited.blogspot.sg/2012/02/producer-consumer-design-pattern-with.html).

### How do you avoid deadlock in Java?

Deadlock is a condition in which two threads wait for each other to take action which allows them to move further. It's a serious issue because when it happens your program hangs and doesn't do the task it is intended for. In order for deadlock to happen, following four conditions must be true:

**Mutual Exclusion:** At least one resource must be held in a non-shareable mode. Only one process can use the resource at any given instant of time.

**Hold and Wait:** A process is currently holding, at least, one resource and requesting additional resources which are being held by other processes.

**No Pre-emption:** The operating system must not de-allocate resources once they have been allocated; they must be released by the holding process voluntarily.

**Circular Wait:**A process must be waiting for a resource which is being held by another process, which in turn is waiting for the first process to release the resource.

The easiest way to avoid deadlock is to prevent *Circular wai*t, and this can be done by acquiring locks in a particular order and releasing them in reverse order so that a thread can only proceed to acquire a lock if it held the other one. Check this [tutorial](http://javarevisited.blogspot.com/2010/10/what-is-deadlock-in-java-how-to-fix-it.html) for the actual code example and detailed discussion on techniques for avoiding deadlock in Java.  
  
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**, it will show you exactly which threads are getting locked and on which object.

Write a Java program which will result in deadlock?

Once you answer this, they may ask you to **write code which will result in deadlock?**  
Here is one of my versions

public class DeadLockDemo {

public void method1() {

synchronized (String.class) {

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

synchronized (Integer.class) {

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

}

}

}

/\*

\* This method also requests same two lock but in exactly

\* Opposite order i.e. first Integer and then String.

\* This creates potential deadlock, if one thread holds String lock

\* and other holds Integer lock and they wait for each other, forever.

\*/

public void method2() {

synchronized (Integer.class) {

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

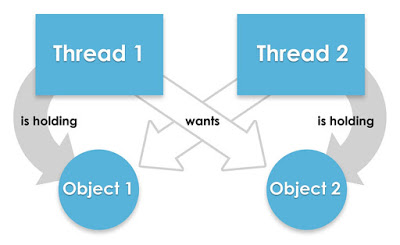
synchronized (String.class) {

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

}

}

If method1() and method2() both will be called by two or many threads , there is a good chance of deadlock because if thread 1 acquires lock on Sting object while executing method1() and thread 2 acquires lock on Integer object while executing method2() both will be waiting for each other to release lock on Integer and String to proceed further which will never happen.  
  
This diagram exactly demonstrate our program, where one thread holds lock on one object and waiting for other object lock which is held by other thread.

[](http://2.bp.blogspot.com/-63RZ-BTlAFs/VfGeHMnGdFI/AAAAAAAADuw/gwqtrVliMsM/s1600/Deadlock+of+Threads.jpg)

How to avoid deadlock in Java?

Now interviewer comes to final part, one of the most important in my view; *how do you fix deadlock?* Or **How to avoid deadlock in Java?**  
If you have looked above code carefully then you may have figured out that real reason for deadlock is not multiple threads but ***the way they are requesting lock***, if you provide an ordered access then problem will be resolved, here is my fixed version, which avoids deadlock by avoiding circular wait with no preemption.

public class DeadLockFixed {

/\*\*

\* Both method are now requesting lock in same order, first Integer and then String.

\* You could have also done reverse e.g. first String and then Integer,

\* both will solve the problem, as long as both method are requesting lock

\* in consistent order.

\*/

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 methods are accessing lock on Integer and String class literal 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.

1. What is the difference between livelock and deadlock in Java?   
   This question is extension of previous interview question. A livelock is similar to a deadlock, except that the states of the threads or processes involved in the livelock constantly change with regard to one another, without any one progressing further. Livelock is a special case of resource starvation. A real-world example of livelock occurs when two people meet in a narrow corridor, and each tries to be polite by moving aside to let the other pass, but they end up swaying from side to side without making any progress because they both repeatedly move the same way at the same time. In short, the main difference between livelock and deadlock is that in former state of process change but no progress is made.
2. How do you check if a Thread holds a lock or not?   
   I didn't even know that you can check if a Thread already holds lock before this question hits me in a telephonic round of Java interview. There is a method called holdsLock() on java.lang.Thread, it returns true if and only if the current thread holds the monitor lock on the specified object. You can also check this article for the more detailed [answer](http://javarevisited.blogspot.com/2010/10/how-to-check-if-thread-has-lock-on.html). 2 ways to find if thread holds lock on object in Java  
   I thought about IllegalMonitorStateException which [wait() and notify() methods](http://javarevisited.blogspot.com/2011/05/wait-notify-and-notifyall-in-java.html) throw when they get called from non-synchronized context so I said I would call newspaper.wait() and if this call [throws exception](http://javarevisited.blogspot.com/2012/02/difference-between-throw-and-throws-in.html) it means thread in java is not holding lock, otherwise thread holds lock.  
   Later I discovered that thread is a static method called holdsLock(Object obj) which returns true or false based on whether threads holds lock on object passed
3. How do you take thread dump in Java? ([answer](http://javarevisited.blogspot.com/2011/07/java-multi-threading-interview.html))  
   There are multiple ways to take thread dump of Java process depending upon operating system. When you take thread dump, JVM dumps state of all threads in log files or standard error console. In windows you can use Ctrl + Break key combination to take thread dump, on Linux you can use kill -3 command for same. You can also use a tool called jstack for taking thread dump, it operate on process id, which can be found using another tool called jps.
4. Which JVM parameter is used to control stack size of a thread? This is the simple one, -Xss parameter is used to control stack size of Thread in Java. You can see this [list of JVM options](http://javarevisited.blogspot.com/2011/11/hotspot-jvm-options-java-examples.html) to learn more about this parameter.
5. What is the difference between synchronized and ReentrantLock in Java?   
   There were days when the only way to provide mutual exclusion in Java was via synchronized keyword, but it has several shortcomings e.g. you cannot extend lock beyond a method or block boundary, you cannot give up trying for a lock etc. Java 5 solves this problem by providing more sophisticated control via Lock interface. ReentrantLock is a common implementation of Lock interface and provides re-entrant mutual exclusion Lock with the same basic behavior and semantics as the implicit monitor lock accessed using synchronized methods and statements, but with extended capabilities. See [this article](http://javarevisited.blogspot.com/2013/03/reentrantlock-example-in-java-synchronized-difference-vs-lock.html) learn about those capabilities and some more differences between synchronized vs ReentrantLock in Java.

**What is ReentrantLock in Java**

On class level, ReentrantLock is a concrete implementation of Lock [interface](http://javarevisited.blogspot.sg/2012/04/10-points-on-interface-in-java-with.html) provided in Java concurrency package from Java 1.5 onwards. As per Javadoc, ReentrantLock is mutual exclusive lock, similar to implicit locking provided by [synchronized keyword in Java](http://javarevisited.blogspot.sg/2011/04/synchronization-in-java-synchronized.html), with extended feature like fairness, which can be used to provide lock to longest waiting thread. Lock is acquired by lock() method and held by [Thread](http://javarevisited.blogspot.com/2011/02/how-to-implement-thread-in-java.html) until a call to unlock() method. Fairness parameter is provided while creating instance of ReentrantLock in constructor. ReentrantLock provides same visibility and ordering guarantee, provided by implicitly locking, which means, unlock() happens before another thread get lock().

**Difference between ReentrantLock and synchronized keyword in Java**

Though ReentrantLock provides same visibility and orderings guaranteed as implicit lock, acquired by synchronized keyword in Java, it provides more functionality and differ in certain aspect. As stated earlier, main difference between synchronized and ReentrantLock is ability to trying for lock interruptibly, and with timeout. [Thread](http://javarevisited.blogspot.com/2012/01/difference-thread-vs-runnable-interface.html) doesn’t need to block infinitely, which was the case with synchronized. Let’s see few more differences between synchronized and Lock in Java.

* Another significant difference between ReentrantLock and synchronized keyword is fairness. synchronized keyword doesn't support fairness. Any thread can acquire lock once released, no preference can be specified, on the other hand you can make ReentrantLock fair by specifying fairness property, while creating instance of ReentrantLock. Fairness property provides lock to longest waiting thread, in case of contention.
* Second difference between synchronized and Reentrant lock is **tryLock**() method. ReentrantLock provides convenient tryLock() method, which acquires lock only if its available or not held by any other thread. This reduces [blocking](http://javarevisited.blogspot.com/2012/02/what-is-blocking-methods-in-java-and.html) of thread waiting for lock in Java application.
* One more worth noting difference between ReentrantLock and synchronized keyword in Java is, ability to interrupt Thread while waiting for Lock. In case of [synchronized](http://javarevisited.blogspot.com/2012/03/mixing-static-and-non-static.html) keyword, a thread can be blocked waiting for lock, for an indefinite period of time and there was no way to control that. ReentrantLock provides a method called lockInterruptibly(), which can be used to interrupt thread when it is [waiting for lock](http://javarevisited.blogspot.com/2011/05/wait-notify-and-notifyall-in-java.html). Similarly tryLock() with timeout can be used to timeout if lock is not available in certain time period.
* ReentrantLock also provides convenient method to get List of all threads waiting for lock.

So, you can see, lot of significant differences between synchronized keyword and ReentrantLock in Java. In short, Lock interface adds lot of power and flexibility and allows some control over lock acquisition process, which can be leveraged to write highly scalable systems in Java.

Benefits of ReentrantLock in Java

Most of the benefits derives from the *differences covered between synchronized vs ReentrantLock* in last section. Here is summary of benefits offered by ReentrantLock over synchronized in Java:

* Ability to lock interruptibly.
* Ability to timeout while waiting for lock.
* Power to create fair lock.
* API to get list of waiting thread for lock.
* Flexibility to try for lock without blocking.

**Disadvantages of ReentrantLock in Java**

Major drawback of using ReentrantLock in Java is wrapping method body inside [try-finally block](http://javarevisited.blogspot.com/2012/11/difference-between-final-finally-and-finalize-java.html), which makes code unreadable and hides business logic. It’s really cluttered and I hate it most, though IDE like [Eclipse](http://javarevisited.blogspot.com/2013/02/must-override-superclass-method-java-eclipse.html) and Netbeans can add those try catch block for you. Another disadvantage is that, now programmer is responsible for acquiring and releasing lock, which is a power but also opens gate for new subtle bugs, when programmer forget to release the lock in finally block.

import java.util.concurrent.locks.ReentrantLock;

import java.util.logging.Level;

import java.util.logging.Logger;

public class ReentrantLockHowto {

private static final ReentrantLock lock = new ReentrantLock();

private static int count = 0;

//Locking using Lock and ReentrantLock

public static int getCount() {

lock.lock();

try {

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

return count++;

} finally {

lock.unlock();

}

}

//Implicit locking using synchronized keyword

public static synchronized int getCountTwo() {

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

return count++;

}

public static void main(String args[]) {

// final ThreadTest counter = new ThreadTest();

Thread t1 = new Thread() {

@Override

public void run() {

while (getCount() < 5) {

try {

Thread.sleep(1000);

} catch (InterruptedException ex) {

ex.printStackTrace(); }

}

}

};

Thread t2 = new Thread() {

@Override

public void run() {

while (getCount() < 5) {

try {

Thread.sleep(100);

} catch (InterruptedException ex) {

ex.printStackTrace();

}

}

}

};

t1.start();

t2.start();

}

}

1. How do you ensure sequence T1, T2, T3 threads in Java?   
   Sequencing in multi-threading can be achieved by different means but you can simply use the join() method of thread class to start a thread when another one has finished its execution. To ensure three threads execute you need to start the last one first e.g. T3 and then call join methods in reverse order e.g. T3 calls T2. join and T2 calls T1.join, these ways T1 will finish first and T3 will finish last. To learn more about join method, see this [tutorial](http://javarevisited.blogspot.sg/2013/02/how-to-join-multiple-threads-in-java-example-tutorial.html).

class TestJoinMethod1 implements Runnable{

public void run(){

try{

Thread.sleep(500);

System.out.println(Thread.currentThread().getName());

}catch(Exception e){System.out.println(e);}

}

public static void main(String args[]){

TestJoinMethod1 test=new TestJoinMethod1();

Thread[] threads = new Thread[7];

try{

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

threads[i] = new Thread(test,"Thread : "+i);

threads[i].start();

threads[i].join();

}

Thread t1 = new Thread(test,"FIRST");

Thread t2 = new Thread(test,"SECOND");

Thread t3 = new Thread(test,"THIRD");

t1.start();

t1.join();

t2.start();

t2.join();

t3.start();

}catch(Exception e){System.out.println(e);}

}  
}

1. Explain yield(), wait() and sleep() methods?   
   Yield method is one way to request current thread to relinquish CPU so that other thread can get a chance to execute. Yield is a static method and only guarantees that current thread will relinquish the CPU but doesn't say anything about which other thread will get CPU. Its possible for the same thread to get CPU back and start its execution again. See this [article](http://java67.blogspot.sg/2012/08/difference-between-yield-and-wait.html) to learn more about yield method and to answer this question better.

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.

**List of difference between wait and sleep in Java:**

* 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 for more detail.
* wait is called on Object while sleep is called on Thread. See Why wait and notify are defined in object class instead of Thread.
* waiting thread can be awake by calling notify and notifyAll while sleeping thread cannot be awaken by calling notify method.
* wait is normally done on condition, Thread wait until a condition is true while sleep is just to put your thread on sleep.
* 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. In this example we have put Main thread in Sleep for 1 second.

publicclass 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 now

* Thread.sleep() method is used to pause the execution, relinquish the CPU and return it to thread scheduler.
* Thread.sleep() method is a static method and always puts current thread on sleep.
* 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.
* Unlike wait() method in Java, sleep() method of Thread class doesn't relinquish the lock it has acquired.
* sleep() method throws Interrupted Exception if another thread interrupt a sleeping thread in java.
* With sleep() in Java it’s 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.
* 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.

1. What is the concurrency level of ConcurrentHashMap in Java?  
   ConcurrentHashMap achieves its scalability and thread-safety by partitioning actual map into a number of sections. This partitioning is achieved using concurrency level. Its optional parameter of ConcurrentHashMap constructor and it's default value is 16. The table is internally partitioned to try to permit the indicated number of concurrent updates without contention. To learn more about concurrency level and internal resizing, see my post [How ConcurrentHashMap works in Java](http://javarevisited.blogspot.com/2013/02/concurrenthashmap-in-java-example-tutorial-working.html).
2. What is Semaphore in Java?  
   Semaphore in Java is a new kind of synchronizer. It's a counting semaphore. Conceptually, a semaphore maintains a set of permits. Each acquire() blocks if necessary until a permit is available, and then takes it. Each release() adds a permit, potentially releasing a blocking acquirer. However, no actual permit objects are used; the Semaphore just keeps a count of the number available and acts accordingly. Semaphore is used to protect an expensive resource which is available in fixed number e.g. database connection in the pool. See this [article](http://javarevisited.blogspot.com/2012/05/counting-semaphore-example-in-java-5.html) to learn more about counting Semaphore in Java.

Counting Semaphore in Java is a synchronizer which allows imposing a bound on resource is added in Java 5 along with other popular concurrent utilities like CountDownLatch, CyclicBarrier and Exchanger etc. Counting Semaphore in Java maintains specified number of pass or permits, In order to access a shared resource, Current Thread must acquire a permit. If permit is already exhausted by other thread than it can wait until a permit is available due to release of permit from different thread. This concurrency utility can be very useful to implement producer consumer design pattern or implement bounded pool or resources like Thread Pool, DB Connection pool etc. java.util.Semaphore class represent a Counting semaphore which is initialized with number of permits. **Semaphore provides two main method acquire() and release() for getting permits and releasing permits**. acquire() method blocks until permit is available. Semaphore provides both blocking method as well as unblocking method to acquire permits. This Java concurrency tutorial focus on a very simple example of Binary Semaphore and demonstrate how mutual exclusion can be achieved using Semaphore in Java.

**Counting Semaphore Example in Java (Binary Semaphore)**

A Counting semaphore with one permit is known as binary semaphore because it has only two state permits available or permit unavailable. Binary semaphore can be used to implement mutual exclusion or critical section where only one thread is allowed to execute. Thread will wait on acquire() until Thread inside critical section release permit by calling release() on semaphore.

here is a simple example of counting semaphore in Java where we are using binary semaphore to provide mutual exclusive access on critical section of code in java:

import java.util.concurrent.Semaphore;

public class SemaphoreTest {

Semaphore binary = new Semaphore(1);

public static void main(String args[]) {

final SemaphoreTest test = new SemaphoreTest();

newThread(){

@Override

public void run(){

test.mutualExclusion();

}

}.start();

newThread(){

@Override

public void run(){

test.mutualExclusion();

}

}.start();

}

private void mutualExclusion() {

try {

binary.acquire();

//mutual exclusive region

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

Thread.sleep(1000);

} catch (InterruptedException i.e.) {

ie.printStackTrace();

} finally {

binary.release();

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

}

}

}

Output:

Thread-0 inside mutual exclusive region

Thread-0 outside of mutual exclusive region

Thread-1 inside mutual exclusive region

Thread-1 outside of mutual exclusive region

Some Scenario where Semaphore can be used:

* To implement better Database connection pool which will block if no more connection is available instead of failing and handover Connection as soon as it’s available.
* To put a bound on collection classes. By using semaphore you can implement bounded collection whose bound is specified by counting semaphore.

That's all on Counting semaphore example in Java. Semaphore is real nice concurrent utility which can greatly simply design and implementation of bounded resource pool. Java 5 has added several useful concurrent utility and deserves a better attention than casual look.

Important points of Counting Semaphore in Java

* Semaphore class supports various overloaded version of tryAquire() method which acquires permit from semaphore only if its available during time of call.
* Another worth noting method from Semaphore is acquireUninterruptibly() which is a blocking call and wait until a permit is available.

1. What happens if you submit a task when the queue of the thread pool is already filled?   
   This is another tricky question on my list. Many programmers will think that it will block until a task is cleared but its true. ThreadPoolExecutor's submit () method throws RejectedExecutionException if the task cannot be scheduled for execution.
2. What is the difference between the submit() and execute() method thread pool?   
   Both methods are ways to submit a task to thread pools but there is a slight difference between them. execute (Runnable command) is defined in Executor interface and executes given task in future, but more importantly, it does not return anything. Its return type is void. On other hand submit () is an overloaded method, it can take either Runnable or Callable task and can return Future object which can hold the pending result of computation. This method is defined on ExecutorService interface, which extends Executor interface, and every other thread pool class e.g. ThreadPoolExecutor or ScheduledThreadPoolExecutor gets these methods. To learn more about thread pools you can check this [article](http://javarevisited.blogspot.sg/2013/07/how-to-create-thread-pools-in-java-executors-framework-example-tutorial.html).
3. What is blocking method in Java?  
   A blocking method is a method which blocks until the task is done, for example, accept() method of ServerSocket blocks until a client is connected. here blocking means control will not return to the caller until the task is finished. On the other hand, there is an asynchronous or non-blocking method which returns even before the task is finished. To learn more about blocking method see this [answer](http://javarevisited.blogspot.sg/2012/02/what-is-blocking-methods-in-java-and.html).
4. What is the difference between invokeAndWait and invokeLater in Java?  
   These are two methods Swing API provides Java developers for updating GUI components from threads other than Event dispatcher thread. InvokeAndWait() synchronously update GUI component, for example, a progress bar, once progress is made, the bar should also be updated to reflect that change. If progress is tracked in a different thread, it has to call invokeAndWait() to schedule an update of that component by Event dispatcher thread. On another hand, invokeLater() is an asynchronous call to update components. You can also refer this [answer](http://javarevisited.blogspot.com/2011/09/invokeandwait-invokelater-swing-example.html) for more points.
5. What is ReadWriteLock in Java?  
   In general, read write lock is the result of lock stripping technique to improve the performance of concurrent applications. In Java, ReadWriteLock is an interface which was added in Java 5 release. A ReadWriteLock maintains a pair of associated locks, one for read-only operations and one for writing. The read lock may be held simultaneously by multiple reader threads, so long as there are no writers. The write lock is exclusive. If you want you can implement this interface with your own set of rules, otherwise you can use ReentrantReadWriteLock, which comes along with JDK and supports a maximum of 65535 recursive write locks and 65535 read locks.

In this example, we are going to demonstrate the use of ReadWriteLock in Java.

Mutually Exclusive Locks, as the ReentrantLock discussed in the previous article, offer far less level of concurrency than non-mutually exclusive locks. If multiple threads are accessing an object for reading data, it does not make sense to use a synchronized block or any other mutually exclusive locks.

The ReadWriteLock offers two main methods Lock readLock() and Lock writeLock(). As the name suggests, the readLock() method is to acquire read-Lock and writeLock is called for acquiring the write-Lock.

**Implementations of ReadWriteLock**

ReadWriteLock is implemented by ReentrantReadWriteLock Class in java.util.concurrent.locks package.Multiple Threads can acquire multiple read Locks, but only a single Thread can acquire mutually-exclusive write Lock .Other threads requesting readLocks have to wait till the write Lock is released. A thread is allowed to degrade from write lock to read lock but not vice-versa. Allowing a read thread to upgrade would lead to a deadlock as more than one thread can try to upgrade its lock. The ReentrantReadWriteLock also supports all the features of the Reentrant lock like providing fair mechanism ,reentrantLocks, Condition Support (on a write Lock only), allowing interruption on read as well as write Locks.

An example below demonstrating how the ReentrantReadWriteLock class can be used.

**ThreadSafeArrayList.java:**

import java.util.ArrayList;

import java.util.List;

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReadWriteLock;

import java.util.concurrent.locks.ReentrantReadWriteLock;

public class ThreadSafeArrayList<E>{

private final ReadWriteLock readWriteLock = new ReentrantReadWriteLock();

private final Lock readLock = readWriteLock.readLock();

private final Lock writeLock = readWriteLock.writeLock();

private final List<E> list = new ArrayList<>();

public void set(E o) {

writeLock.lock();

try {

list.add(o);

System.out.println("Adding element by thread"+Thread.currentThread().getName());

}

finally {

writeLock.unlock();

}

}

public E get(int i) {

readLock.lock();

try {

System.out.println("Printing elements by thread"+Thread.currentThread().getName());

return list.get(i);

}

finally {

readLock.unlock();

}

}

public static void main(String[] args) {

ThreadSafeArrayList<String> threadSafeArrayList = new ThreadSafeArrayList<>();

threadSafeArrayList.set("1");

threadSafeArrayList.set("2");

threadSafeArrayList.set("3");

System.out.println("Printing the First Element : "+threadSafeArrayList.get(1));

}

}

OUTPUT :

1 Adding element by thread main

2 Adding element by thread main

3 Adding element by thread main

4 Printing elements by thread main

5 Reading from List : 2

**Closing words**

ReadWriteLocks offer greater level of concurrency as compared to traditional mutually-exclusive locks. However, this lock performs best when there are more readLocks required as compared to writeLocks. Using ReadWriteLocks without properly understanding its benefits and trade-offs may hamper the application throughput.

1. What is busy spin in multi-threading? ([answer](http://java67.blogspot.com/2015/09/60-java-interview-questions-for-quick.html" \t "_blank))  
   Busy spin is a technique which concurrent programmers employ to make a thread wait on certain condition. Unlike traditional methods e.g. wait (), sleep () or yield () which all involves relinquishing CPU control, this method does not relinquish CPU, instead it the just runs empty loop. Why would someone do that? To preserve CPU caches. In a multi-core system, it's possible for a paused thread to resume on a different core, which means rebuilding cache again. To avoid cost of rebuilding cache, programmer prefers to wait for much smaller time doing busy spin. You can also see this [answer](http://java67.blogspot.com/2012/08/5-thread-interview-questions-answers-in.html) to learn more about this question.
2. What is the difference between the volatile and atomic variable in Java?  
   This is an interesting question for Java programmer, at first, volatile and atomic variable look very similar, but they are different. Volatile variable provides you happen-before guarantee that a write will happen before any subsequent write, it doesn't guarantee atomicity. For example count++ operation will not become atomic just by declaring count variable as volatile. On the other hand AtomicInteger class provides atomic method to perform such compound operation atomically e.g. getAndIncrement() is atomic replacement of increment operator. It can be used to atomically increment current value by one. Similarly you have atomic version for other data type and reference variable as well.

A small toolkit of classes that support lock-free thread-safe programming on single variables. In essence, the classes in this package extend the notion of volatile values, fields, and array elements to those that also provide an atomic conditional update operation of the form:

**boolean compareAndSet(expectedValue, updateValue);**

This method (which varies in argument types across different classes) atomically sets a variable to the updateValue if it currently holds the expectedValue, reporting true on success. The classes in this package also contain methods to get and unconditionally set values, as well as a weaker conditional atomic update operation weakCompareAndSet described below.

The specifications of these methods enable implementations to employ efficient machine-level atomic instructions that are available on contemporary processors. However on some platforms, support may entail some form of internal locking. Thus the methods are not strictly guaranteed to be non-blocking -- a thread may block transiently before performing the operation.

Instances of classes AtomicBoolean, AtomicInteger, AtomicLong, and AtomicReference each provide access and updates to a single variable of the corresponding type. Each class also provides appropriate utility methods for that type. For example, classes AtomicLong and AtomicInteger provide atomic increment methods. One application is to generate sequence numbers, as in:

class Sequencer {

private final AtomicLong sequenceNumber = new AtomicLong(0);

public long next() {

return sequenceNumber.getAndIncrement();

}

}

|  |  |
| --- | --- |
| Modifier and Type | Method and Description |
| int | [addAndGet](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/atomic/AtomicInteger.html#addAndGet(int))(int delta)  Atomically adds the given value to the current value. |
| boolean | [compareAndSet](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/atomic/AtomicInteger.html#compareAndSet(int,%20int))(int expect, int update)  Atomically sets the value to the given updated value if the current value == the expected value. |
| int | [decrementAndGet](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/atomic/AtomicInteger.html#decrementAndGet())()  Atomically decrements by one the current value. |
| double | [doubleValue](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/atomic/AtomicInteger.html#doubleValue())()  Returns the value of the specified number as a double. |
| float | [floatValue](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/atomic/AtomicInteger.html#floatValue())()  Returns the value of the specified number as a float. |
| int | [get](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/atomic/AtomicInteger.html#get())()  Gets the current value. |
| int | [getAndAdd](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/atomic/AtomicInteger.html#getAndAdd(int))(int delta)  Atomically adds the given value to the current value. |
| int | [getAndDecrement](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/atomic/AtomicInteger.html#getAndDecrement())()  Atomically decrements by one the current value. |
| int | [getAndIncrement](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/atomic/AtomicInteger.html#getAndIncrement())()  Atomically increments by one the current value. |
| int | [getAndSet](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/atomic/AtomicInteger.html#getAndSet(int))(int newValue)  Atomically sets to the given value and returns the old value. |
| int | [incrementAndGet](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/atomic/AtomicInteger.html#incrementAndGet())()  Atomically increments by one the current value. |
| int | [intValue](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/atomic/AtomicInteger.html#intValue())()  Returns the value of the specified number as an int. |
| void | [lazySet](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/atomic/AtomicInteger.html#lazySet(int))(int newValue)  Eventually sets to the given value. |
| long | [longValue](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/atomic/AtomicInteger.html#longValue())()  Returns the value of the specified number as a long. |
| void | [set](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/atomic/AtomicInteger.html#set(int))(int newValue)  Sets to the given value. |
| [String](http://docs.oracle.com/javase/7/docs/api/java/lang/String.html) | [toString](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/atomic/AtomicInteger.html#toString())()  Returns the String representation of the current value. |
| boolean | [weakCompareAndSet](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/atomic/AtomicInteger.html#weakCompareAndSet(int,%20int))(int expect, int update)  Atomically sets the value to the given updated value if the current value == the expected value. |

1. What happens if a thread throws an Exception inside synchronized block?   
   This is one trickier question for average Java programmer, if he can bring the fact about whether lock is released or not is a key indicator of his understanding. To answer this question, no matter how you exist synchronized block, either normally by finishing execution or abruptly by throwing exception, thread releases the lock it acquired while entering that synchronized block. This is actually one of the reasons I like synchronized block over lock interface, which requires explicit attention to release lock, generally this is achieved by releasing the lock in a [finally block](http://javarevisited.blogspot.com/2012/11/difference-between-final-finally-and-finalize-java.html).
2. List down 3 multi-threading best practice you follow? ([answer](http://javarevisited.blogspot.com/2015/05/top-10-java-multithreading-and.html" \t "_blank))  
   This is my favorite question because I believe that you must follow certain best practices while writing concurrent code which helps in performance, debugging and maintenance. Following are three best practices, I think an average Java programmer should follow:

**Always give meaningful name to your thread:** This goes a long way to find a bug or trace an execution in concurrent code. OrderProcessor, QuoteProcessor or TradeProcessor is much better than Thread-1. Thread-2 and Thread-3. The name should say about task done by that thread. All major framework and even JDK follow this best practice.

**Avoid locking or Reduce scope of Synchronization**  
Locking is costly and context switching is even costlier. Try to avoid synchronization and locking as much as possible and at a bare minimum, you should reduce critical section. That's why I prefer synchronized block over synchronized method because it gives you absolute control on the scope of locking.

**Prefer Synchronizers over wait and notify**  
Synchronizers like CountDownLatch, Semaphore, CyclicBarrier or Exchanger simplifies coding. It's very difficult to implement complex control flow right using wait and notify. Secondly, these classes are written and maintained by best in business and there is good chance that they are optimized or replaced by better performance code in subsequent JDK releases. By using higher level synchronization utilities, you automatically get all these benefits.

**Prefer Concurrent Collection over Synchronized Collection**  
this is another simple best practice which is easy to follow but reap good benefits. Concurrent collections are more scalable than their synchronized counterpart, that's why it’s better to use them while writing concurrent code. So next time if you need map, think about ConcurrentHashMap before thinking Hashtable. See my article [Concurrent Collections in Java](http://javarevisited.blogspot.com/2013/02/concurrent-collections-from-jdk-56-java-example-tutorial.html), to learn more about modern collection classes and how to make best use of them.

1. How do you force to start a Thread in Java?  
   This question is like how do you force garbage collection in Java, there is no way though you can make a request using System.gc () but it's not guaranteed. On Java multi-threading there is absolute no way to force start a thread, this is controlled by thread scheduler and Java exposes no API to control thread schedule. This is still a random bit in Java.
2. What is the fork-join framework in Java?  
   The fork join framework, introduced in JDK 7 is a powerful tool available to Java developer to take advantage of multiple processors of modern day servers. It is designed for work that can be broken into smaller pieces recursively. The goal is to use all the available processing power to enhance the performance of your application. One significant advantage of the fork/join framework is that it uses a work-stealing algorithm. Worker threads that run out of things to do can steal tasks from other threads that are still busy. See this [article](http://javarevisited.blogspot.com/2011/09/fork-join-task-java7-tutorial.html) for the much more detailed answer to this question.