* What is Busy Spinning? Why Should You Use It in Java?

One of the interesting multithreading question to senior Java programmers, busy spinning is a *waiting strategy*, in which a thread just wait in a loop, without releasing the CPU for going to sleep. This is a very advanced and specialized waiting strategy used in the high-frequency trading application when wait time between two messages is very minimal.  
  
By not releasing the CPU your thread retains all the cached data and instruction, which may be lost if the thread was suspended and resumed back in a different core of CPU.   
  
This question is quite popular in high-frequency low latency programming domain, where programmers are trying for extremely low latency in the range of micro to milliseconds

Busy spinning/waiting is normally a bad idea from a performance standpoint. In most cases, it is preferable to sleep and wait for a signal when you are ready to run, than to do spinning. Take the scenario where there are two threads, and thread 1 is waiting for thread 2 to set a variable (say, it waits until var == true. Then, it would busy spin by just doing

while (var == false);

In this case, you will take up a lot of time that thread 2 can potentially be running, because when you wake up you are just executing the loop mindlessly. So, in a scenario where you are waiting for something like this to happen, it is better to let thread 2 have all control by putting yourself to sleep and having it wake you up when it is done.

BUT, in rare cases where the time you need to wait is **very short**, it is actually **faster to spinlock**. This is because of the time it takes to perform the signaling functions; spinning is preferable if the time used spinning is less than the time it would take to perform the signaling. So, in that way it may be beneficial and could actually improve performance, but this is definitely not the most frequent case.

* What is Read-Write Lock? Does ConcurrentHashMap in Java Uses The ReadWritLock?

## ReadWrite Lock is an implementation of *lock stripping* technique, where two separate locks are used for read and write operation. Since read operation doesn't modify the state of the object, it's safe to allow multiple thread access to a shared object for reading without locking, and by splitting one lock into read write lock, you can easily do that.  Java provides an implementation of read-write lock in the form of ReentrantReadWritLock class in the java.util.concurrent.lock package. This is worth looking before you decide to write your own read-write locking implementation.  Also, the current implementation of java.util.ConcurrentHashMap doesn't use the ReadWriteLock, instead, it divides the Map into several segments and locks them separately using different locks. This means any given time, *only a portion of the ConcurrentHashMap is locked*, instead of the whole Map. What is immutable object in Java? Can you change values of an immutable object?

A Java object is considered immutable when its state cannot change after it is created. Use of immutable objects is widely accepted as a sound strategy for creating simple, reliable code. Immutable objects are particularly useful in concurrent applications. Since they cannot change state, they cannot be corrupted by thread interference or observed in an inconsistent state. Java.lang.String and java.lang.Integer classes are the Examples of immutable objects from the Java Development Kit. Immutable objects simplify your program due to following characteristics:

* Immutable objects are simple to use test and construct.
* Immutable objects are automatically thread-safe.
* Immutable objects do not require a copy constructor.
* Immutable objects do not require an implementation of clone.
* Immutable objects allow hashCode to use lazy initialization, and to cache its return value.
* Immutable objects do not need to be copied defensively when used as a field.
* Immutable objects are good Map keys and Set elements (Since state of these objects must not change while stored in a collection).
* Immutable objects have their class invariant established once upon construction, and it never needs to be checked again.
* Immutable objects always have "failure atomicity" (a term used by Joshua Bloch) : if an immutable object throws an exception, it's never left in an undesirable or indeterminate state.

How to create an immutable object in Java? Does all property of immutable object needs to be final?

To create an object immutable You need to make the class final and all its member final so that once objects gets created no one can modify its state. You can achieve same functionality by making member as non-final but private and not modifying them except in constructor. Also its NOT necessary to have all the properties final since you can achieve same functionality by making member as non-final but private and not modifying them except in constructor.

What is difference between String, StringBuffer and StringBuilder? When to use them?

The main difference between the three most commonly used String classes as follows.

* StringBuffer and StringBuilder objects are mutable whereas String class objects are immutable.
* StringBuffer class implementation is synchronized while StringBuilder class is not synchronized.
* Concatenation operator "+" is internally implemented by Java using either StringBuffer or StringBuilder.

Criteria to choose among String, StringBuffer and StringBuilder

* If the Object value will not change in a scenario use String Class because a String object is immutable.
* If the Object value can change and will only be modified from a single thread, use a StringBuilder because StringBuilder is unsynchronized (means faster).
* If the Object value may change, and can be modified by multiple threads, use a StringBuffer because StringBuffer is thread safe (synchronized).

Why String class is final or immutable?

It is very useful to have strings implemented as final or immutable objects. Below are some advantages of String Immutability in Java

* Immutable objects are thread-safe. Two threads can both work on an immutable object at the same time without any possibility of conflict.
* Security: the system can pass on sensitive bits of read-only information without worrying that it will be altered
* You can share duplicates by pointing them to a single instance.
* You can create substrings without copying. You just create a pointer into an existing base String guaranteed never to change. Immutability is the secret that makes Java substring implementation very fast.
* Immutable objects are good fit for becoming Hashtable keys. If you change the value of any object that is used as a hash table key without removing it and re-adding it you will lose the object mapping.
* Since String is immutable, inside each String is a char [] exactly the correct length. Unlike a StringBuilder there is no need for padding to allow for growth.
* If String were not final, you could create a subclass and have two strings that look alike when "seen as Strings", but that are actually different.

Is Java Pass by Reference or Pass by Value?

The Java Spec says that everything in Java is pass-by-value. There is no such thing as "pass-by-reference" in Java. The difficult thing can be to understand that Java passes "objects as references" passed by value.

What is OutOfMemoryError in java? How to deal with java.lang.OutOfMemeryError error?

This Error is thrown when the Java Virtual Machine cannot allocate an object because it is out of memory, and no more memory could be made available by the garbage collector. **Note:** Its an Error (extends java.lang.Error) not Exception. Two important types of OutOfMemoryError are often encountered

1. java.lang.OutOfMemoryError: Java heap space

The quick solution is to add these flags to JVM command line when Java runtime is started:

-Xms1024m -Xmx1024m

1. java.lang.OutOfMemoryError: PermGen space

The solution is to add these flags to JVM command line when Java runtime is started:

-XX:+CMSClassUnloadingEnabled-XX:+CMSPermGenSweepingEnabled

**Long Term Solution**: Increasing the Start/Max Heap size or changing Garbage Collection options may not always be a long term solution for your Out Of Memory Error problem. Best approach is to understand the memory needs of your program and ensure it uses memory wisely and does not have leaks. You can use a Java memory profiler to determine what methods in your program are allocating large number of objects and then determine if there is a way to make sure they are no longer referenced, or to not allocate them in the first place.

What is the use of the finally block? Is finally block in Java guaranteed to be called? When finally block is NOT called?

Finally is the block of code that executes always. The code in finally block will execute even if an exception is occurred. Finally block is NOT called in following conditions

* If the JVM exits while the try or catch code is being executed, then the finally block may not execute. This may happen due to System.exit() call.
* if the thread executing the try or catch code is interrupted or killed, the finally block may not execute even though the application as a whole continues.
* If a exception is thrown in finally block and not handled then remaining code in finally block may not be executed.

Why there are two Date classes; one in java.util package and another in java.sql?

From the JavaDoc of java.sql.Date:

A thin wrapper around a millisecond value that allows JDBC to identify this as an SQL DATE value. A milliseconds value represents the number of milliseconds that have passed since January 1, 1970 00:00:00.000 GMT. To conform with the definition of SQL DATE, the millisecond values wrapped inside a java.sql.Date instance must be 'normalized' by setting the hours, minutes, seconds, and milliseconds to zero.

**Explanation**: A java.util.Date represents date and time of day, a java.sql.Date only represents a date (the complement of java.sql.Date is java.sql.Time, which only represents a time of day, but also extends java.util.Date).

What is Marker interface? How is it used in Java?

The marker interface is a design pattern, used with languages that provide run-time type information about objects. It provides a way to associate metadata with a class where the language does not have explicit support for such metadata. To use this pattern, a class implements a marker interface, and code that interact with instances of that class test for the existence of the interface. Whereas a typical interface specifies methods that an implementing class must support, a marker interface does not do so. The mere presence of such an interface indicates specific behavior on the part of the implementing class. There can be some hybrid interfaces, which both act as markers and specify required methods, are possible but may prove confusing if improperly used. Java utilizes this pattern very well and the example interfaces are

* java.io.Serializable - Serializability of a class is enabled by the class implementing the java.io.Serializable interface. The Java Classes that do not implement Serializable interface will not be able to serialize or de-serialize their state. All subtypes of a serializable class are themselves serializable. The serialization interface has no methods or fields and serves only to identify the semantics of being serializable.
* java.rmi.Remote - The Remote interface serves to identify interfaces whose methods may be invoked from a non-local virtual machine. Any object that is a remote object must directly or indirectly implement this interface. Only those methods specified in a "remote interface", an interface that extends java.rmi.Remote are available remotely.
* java.lang.Cloneable - A class implements the Cloneable interface to indicate to the Object.clone () method that it is legal for that method to make a field-for-field copy of instances of that class. Invoking Object's clone method on an instance that does not implement the Cloneable interface results in the exception CloneNotSupportedException being thrown.
* javax.servlet.SingleThreadModel - Ensures that servlets handle only one request at a time. This interface has no methods.
* java.util.EvenListener - A tagging interface that all event listener interfaces must extend.

The "instance of" keyword in java can be used to test if an object is of a specified type. So this keyword in combination with Marker interface can be used to take different actions based on type of interface an object implements.

Why main() is declared public static void ?

Public - main method is called by JVM to run the method which is outside the scope of project therefore the access specifier has to be public to permit call from anywhere outside the application static - When the JVM makes are call to the main method there is not object existing for the class being called therefore it has to have static method to allow invocation from class. void - Java is platform independent language therefore if it will return some value then the value may mean different to different platforms so unlike C it can not assume a behavior of returning value to the operating system. If main method is declared as private then - Program will compile properly but at run-time it will give "Main method not public." error.

# [Top 50 Java Thread Interview Questions Answers for Experienced](http://javarevisited.blogspot.in/2014/07/top-50-java-multithreading-interview-questions-answers.html)

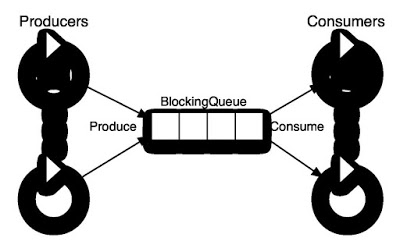
You go to any Java interview, senior or junior, experience or freshers,  you are bound to see a couple of questions from the thread, concurrency, and multi-threading. In fact, this built-in concurrency support is one of the strongest points of Java programming language and helped it to gain popularity among enterprise world and programmers equally. Most of lucrative Java developer position demands *excellent core Java multi-threading skills* and experience in developing, debugging and tuning high-performance low latency concurrent Java applications. This is the reason, it is one of the most sought after skill on Java interviews. The multithreading and concurrency are also hard to master concept and only good developers with solid experience can effectively deal with concurrency issues.  
  
In a typical Java interview, Interviewer slowly starts from basic concepts of Thread by asking questions like, why you need threads, how to create threads, which one is better way to create threads e.g. by extending thread class or implementing Runnable and then slowly goes into Concurrency issues, challenges faced during development of concurrent Java applications, Java memory model, higher-order concurrency utilities introduced in JDK 1.5, principles and design patterns of concurrent Java applications, classical multi-threading problems e.g. producer-consumer, dining philosopher, reader-writer or simply bounded buffer problems.  
  
Since its also not enough just to know basics of threading, you must know how to deal with concurrency problems e.g. deadlock, race conditions, memory inconsistency and various thread safety related issues. These skills are thoroughly get tested by presenting various multi-threading and concurrency problems.  
  
  
Many Java developers are used to only look and read interview questions before going for the interview, which is not bad but you should not be too far away. Also collecting questions and going through the same exercise is too much time consuming, that's why I have created this list of *top 50 Java multi-threading and concurrency related questions*, collected from various interviews. I am only going to add new and recent interview questions as and when I am going to discover them.  
  
Though you need good knowledge and solid experience to do well on Java interviews focused on advanced multithreading and concurrency skill, I strongly recommend Java programmers to read [Effective Java](http://www.amazon.com/dp/0321356683/?tag=javamysqlanta-20) and [Java Concurrency in Practice](http://www.amazon.com/dp/0321349601/?tag=javamysqlanta-20) twice before going to interview. They do not only help you to answer questions better but also help you to present your idea clearly.  
  
By the way, I have not provided answers to some questions here, Why? because I expect most of Java developer to know the answers to this question and if not, also answers are widely available by using Google. If you don't find the answer to any particular question, you can always ask me in the comments section. You can even find answers to few questions on the link provided or my earlier post [Top 12 Java Thread Questions with Answers](http://java67.blogspot.sg/2012/08/5-thread-interview-questions-answers-in.html).

## 50 Interview questions from Java Multi-threading and Concurrency

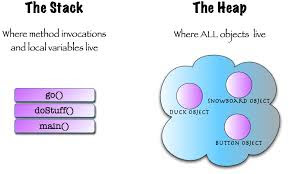
Here is our list of top questions from Java thread, concurrency, and multi-threading. You can use this list to prepare well for your Java interview.  
  
  
**1)  What is Thread in Java?**  
The thread is an independent path of execution. It's way to take advantage of multiple CPU available in a machine. 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 thread 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.  
  
  
**3)  How do you implement Thread in Java?**  
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. For detailed answer and discussion see this article.  
  
  
**4)  When to use Runnable vs Thread in Java?**  
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 CommandListener. For more points and discussion you can also refer this post.  
  
  
  
**6)  What is the difference between start() and run() method of Thread class?**  
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, its 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.  
  
  
**7)  What is the difference between Runnable and Callable in Java?**  
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.  
  
  
**8)  What is the difference between CyclicBarrier and CountDownLatch in Java?**  
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.  
  
  
**9)  What is Java Memory model?**(answer)  
Java Memory model is set of rules and guidelines which allows 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 allows 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.

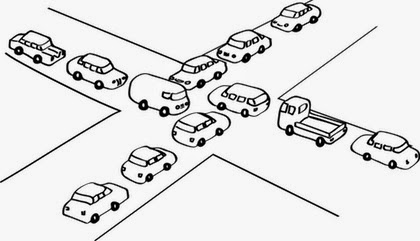
**10) What is volatile variable in Java?**  
Volatile is a special modifier, which can only be used with instance variables. In concurrent Java programs, changes made by multiple threads on instance variables is not visible to other in absence of any synchronizers e.g. synchronized keyword or locks. Volatile variable guarantees that a write will happen before any subsequent read: as stated: *"volatile variable rule"* in previous question. Read this [answer](http://javarevisited.blogspot.com/2011/06/volatile-keyword-java-example-tutorial.html) to learn more about volatile variable and when to use them.  
  
  
**11) What is thread-safety? is Vector a thread-safe class?**   
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 category, 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.  
  
  
**12) What is race condition in Java? Given one example?**   
Race condition are cause of some subtle programming bugs when Java programs are exposed to concurrent execution environment. As the name suggests, a race condition occurs due to race between multiple threads, if a thread which is supposed to execute first lost the race and executed second, behavior of code changes, which surface as non-deterministic bugs. This is one of the hardest bugs to find and re-produce because of random nature of racing between threads. One example of race condition is out-of-order processing, see this [answer](http://javarevisited.blogspot.com/2012/02/what-is-race-condition-in.html) for some more example of race conditions in Java programs.  
  
  
**13) How to stop a thread in Java?**  
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.  
  
  
  
**14) 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.  
  
  
**15) 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 . 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)

**16) What is the difference between notify and notifyAll in Java?**  
This is another tricky questions from core Java interviews, since multiple threads can wait on single monitor lock, Java API designer provides method to inform only one of them or all of them, once waiting condition changes, but they provide half implementation. There notify() method doesn't provide any way to choose a particular thread, that's why its only useful when you know that there is only one thread is waiting. On the other hand, notifyAll() sends notification to all threads and allows them to compete for locks, which ensures that at-least one thread will proceed further. See my [blog post](http://javarevisited.blogspot.com/2012/10/difference-between-notify-and-notifyall-java-example.html) on similar topic for a more detailed answer and code example.  
  
  
**17) Why wait, notify and notifyAll are not inside thread class?**   
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 make sense for these three method 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 make 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.  
  
  
**18) What is ThreadLocal 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.  
  
  
**19) 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.  
  
  
**20) 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 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  
  
  
  
**21) Why wait and notify method are called from synchronized block?**  
Main reason 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 .  
  
  
**22) Why should you check condition for waiting in a loop?**  
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.  
  
  
**23) 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.

**24) What is the difference between Stack and Heap in Java?**  
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.   
[](https://3.bp.blogspot.com/-vJvHCwr7ozY/VuBB4nlNpkI/AAAAAAAAFCk/8mqWs5unUK4/s1600/Heap+vs+Stack+in+Java.jpg)

**25) 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).   
  
**26) Write code to solve Producer Consumer problem in Java?**  
Most of the threading problem you solved in the real world are 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.  
  
  
**27) How do you avoid deadlock in Java? Write Code?**

[](http://4.bp.blogspot.com/-m2IldPcxiJI/U6-Zwvkdd1I/AAAAAAAABns/-zHIHjzM3nM/s1600/deadlock+in+Java.jpg)

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 happen 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.  
  
  
**28) 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.  
  
  
  
**29) 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.   
  
  
**30) How do you take thread dump in Java?**  
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.  
  
  
**31) Which JVM parameter is used to control stack size of a thread?**(answer)  
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.  
  
  
**32) What is the difference between synchronized and ReentrantLock in Java?**(answer)  
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 can not extend lock beyond a method or block boundary, you can not 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.  
  
  
**33) There are three threads T1, T2, and T3? How do you ensure sequence T1, T2, T3 in Java?**(answer)  
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).  
  
  
**34) What does yield method of Thread class do?**(answer)  
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.  
  
  
**35) What is the concurrency level of ConcurrentHashMap in Java?**(answer)  
ConcurrentHashMap achieves it's 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).  
  
  
  
**36) What is Semaphore in Java?**(answer)  
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.  
  
  
**37) What happens if you submit a task when the queue of the thread pool is already filled?**(answer)  
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.  
  
  
**38) What is the difference between the submit() and execute() method thread pool in Java?**(answer)  
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).  
  
  
**39) What is blocking method in Java?**(answer)  
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).  
  
  
**40) Is Swing thread-safe? What do you mean by Swing thread-safe?**(answer)  
You can simply this question as No, Swing is not thread-safe, but you have to explain what you mean by that even if the interviewer doesn't ask about it. When we say swing is not thread-safe we usually refer its component, which can not be modified in multiple threads. All update to GUI components has to be done on AWT thread, and Swing provides synchronous and asynchronous callback methods to schedule such updates. You can also read my article to learn more about [swing and thread-safety](http://javarevisited.blogspot.com/2013/08/why-swing-is-not-thread-safe-in-java-Swingworker-Event-thread.html) to better answer this question. Even next two questions are also related to this concept.  
  
  
**41) What is the difference between invokeAndWait and invokeLater in Java?**(answer)  
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.  
  
  
**42) Which method of Swing API are thread-safe in Java?**(answer)  
This question is again related to swing and thread-safety though components are not thread-safe there is a certain method which can be safely called from multiple threads. I know about repaint(), and revalidate() being thread-safe but there are other methods on different swing components e.g. setText()method of JTextComponent, insert() and append() method of JTextArea class.  
  
  
**43) How to create an Immutable object in Java?**(answer)  
This question might not look related to multi-threading and concurrency, but it is. Immutability helps to simplify already complex concurrent code in Java. Since immutable object can be shared without any synchronization its very dear to Java developers. Core value object, which is meant to be shared among thread should be immutable for performance and simplicity. Unfortunately there is no @Immutable annotation in Java, which can make your object immutable, hard work must be done by Java developers. You need to keep basics like initializing state in constructor, no setter methods, no leaking of reference, keeping separate copy of mutable object to create Immutable object. For step by step guide see my post, [how to make an object Immutable in Java](http://javarevisited.blogspot.com/2013/03/how-to-create-immutable-class-object-java-example-tutorial.html). This will give you enough material to answer this question with confidence.  
  
  
**44) What is ReadWriteLock in Java?**(answer)  
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.  
  
  
**45) What is busy spin in multi-threading?**([answer](http://java67.blogspot.com/2015/09/60-java-interview-questions-for-quick.html))  
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 prefer 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.  
  
  
  
**46) What is the difference between the volatile and atomic variable in Java?**(answer)  
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 happens-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.  
  
  
**47) What happens if a thread throws an Exception inside synchronized block?**(answer)  
This is one more tricky 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).  
  
  
**48) What is double checked locking of Singleton?**(answer)  
This is one of the very popular question on Java interviews, and despite its popularity, chances of candidate answering this question satisfactory is only 50%. Half of the time, they failed to write code for double checked locking and half of the time they failed how it was broken and fixed on Java 1.5. This is actually an old way of creating thread-safe singleton, which tries to optimize performance by only locking when Singleton instance is created first time, but because of complexity and the fact it was broken for JDK 1.4,  I personally don't like it. Anyway, even if you not prefer this approach its good to know from interview point of view. Since this question deserve a detailed answer, I have answered in a separate post, you can read my post [how double checked locking on Singleton works](http://javarevisited.blogspot.sg/2014/05/double-checked-locking-on-singleton-in-java.html) to learn more about it.  
  
  
**49) How to create thread-safe Singleton in Java?**(answer)  
This question is actually follow-up of the previous question. If you say you don't like double checked locking then Interviewer is bound to ask about alternative ways of creating thread-safe Singleton class. There are actually man, you can take advantage of class loading and static variable initialization feature of JVM to create instance of Singleton, or you can leverage powerful enumeration type in Java to create Singleton. I actually preferred that way, you can also read this [article](http://javarevisited.blogspot.com/2012/12/how-to-create-thread-safe-singleton-in-java-example.html) to learn more about it and see some sample code.  
  
  
**50) List down 3 multi-threading best practice you follow?**([answer](http://javarevisited.blogspot.com/2015/05/top-10-java-multithreading-and.html))  
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 collection are more scalable than their synchronized counterpart, that's why its 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.

**51) How do you force to start a Thread in Java?**(answer)  
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 absolutely 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.  
  
  
**52) What is the fork-join framework in Java?**(answer)  
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.  
  
  
**53) What is the difference between calling wait() and sleep() method in Java multi-threading?**(answer)  
Though both wait and sleep introduce some form of pause in Java application, they are the tool for different needs. Wait method is used for inter-thread communication, it relinquishes lock if waiting for a condition is true and wait for notification when due to an action of another thread waiting condition becomes false. On the other hand sleep() method is just to relinquish CPU or stop execution of current thread for specified time duration. Calling sleep method doesn't release the lock held by the current thread. You can also take look at this [article](http://javarevisited.blogspot.com/2011/12/difference-between-wait-sleep-yield.html) to answer this question with more details.  
  
  
That's all on this list of **top 50 Java multi-threading and concurrency interview questions**. I have not shared answers of all the questions but provided enough hints and links to explore further and find answers by yourselves. As I said, let me know if you don't find answer of any particular question and I will add answer here.  
  
You can use this list to not only to prepare for your core Java and programming interviews but also to check your knowledge about basics of threads, multi-threading, concurrency, design patterns and threading issues like race conditions, deadlock and thread safety problems.  
  
My intention is to make this list of question as the mother of all list of Java Multi-threading questions, but this can not be done without your help. You can also share any question with us, which has been asked to you or any question for which you yet to find an answer.  
  
This master list is equally useful to Java developers of all levels of experience. You can read through this list even if you have 2 to 3 years of working experience as a junior developer or 5 to 6 years as a senior developer. It's even useful for freshers and beginners to expand their knowledge. I will add new and latest multi-threading question as and when I come across, and I request you all to ask, share and answer questions via comments to keep this list relevant to all Java programmers.

***NEW***

Thread state for a thread which has not yet started.

***RUNNABLE***

Thread state for a runnable thread. A thread in the runnable

State is executing in the Java virtual machine but it may

Be waiting for other resources from the operating system

Such as processor.

***BLOCKED***

Thread state for a thread blocked waiting for a monitor lock.

A thread in the blocked state is waiting for a monitor lock

to enter a synchronized block/method or re-enter a synchronized

block/method after calling Object.wait.

***WAITING***,

Thread state for a waiting thread.

A thread is in the waiting state due to calling one of the

following methods:

Object.wait with no timeout

Thread.join with no timeout

LockSupport.park

A thread in the waiting state is waiting for another thread to perform a particular action.

For example, a thread that has called Object.wait() on an object is waiting for another thread to call Object.notify() or Object.notifyAll() on that object. A thread that has called Thread.join() is waiting for a specified thread to terminate.

***TIMED\_WAITING***

Thread state for a waiting thread with a specified waiting time.

A thread is in the timed waiting state due to calling one of

the following methods with a specified positive waiting time:

Thread.sleep

wait(long) Object.wait} with timeout

join(long) Thread.join} with timeout

LockSupport.parkNanos

LockSupport.parkUntil

***TERMINATED***

Thread state for a terminated thread. The thread has completed execution

## What are the usage of LockSupport ?

* public class **LockSupport** extends [Object](https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html)
* Basic thread blocking primitives for creating locks and other synchronization classes.
* This class associates, with each thread that uses it, a permit (in the sense of the [Semaphore](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Semaphore.html) class). A call to park will return immediately if the permit is available, consuming it in the process; otherwise it may block. A call to unpark makes the permit available, if it was not already available. (Unlike with Semaphores though, permits do not accumulate. There is at most one.)
* Methods park and unpark provide efficient means of blocking and unblocking threads that do not encounter the problems that cause the deprecated methods Thread.suspend and Thread.resume to be unusable for such purposes: Races between one thread invoking park and another thread trying to unpark it will preserve liveness, due to the permit. Additionally, park will return if the caller's thread was interrupted, and timeout versions are supported. The park method may also return at any other time, for "no reason", so in general must be invoked within a loop that rechecks conditions upon return. In this sense park serves as an optimization of a "busy wait" that does not waste as much time spinning, but must be paired with an unpark to be effective.
* The three forms of park each also support a blocker object parameter. This object is recorded while the thread is blocked to permit monitoring and diagnostic tools to identify the reasons that threads are blocked. (Such tools may access blockers using method [getBlocker(java.lang.Thread)](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/LockSupport.html#getBlocker(java.lang.Thread)).) The use of these forms rather than the original forms without this parameter is strongly encouraged. The normal argument to supply as a blocker within a lock implementation is this.
* These methods are designed to be used as tools for creating higher-level synchronization utilities, and are not in themselves useful for most concurrency control applications. The park method is designed for use only in constructions of the form:
* while (!canProceed()) { ... LockSupport.park(this); }
* where neither canProceed nor any other actions prior to the call to park entail locking or blocking. Because only one permit is associated with each thread, any intermediary uses of park could interfere with its intended effects.
* **Sample Usage.** Here is a sketch of a first-in-first-out non-reentrant lock class:
* class FIFOMutex {
* private final AtomicBoolean locked = new AtomicBoolean(false);
* private final Queue<Thread> waiters
* = new ConcurrentLinkedQueue<Thread>();
* public void lock() {
* boolean wasInterrupted = false;
* Thread current = Thread.currentThread();
* waiters.add(current);
* // Block while not first in queue or cannot acquire lock
* while (waiters.peek() != current ||
* !locked.compareAndSet(false, true)) {
* LockSupport.park(this);
* if (Thread.interrupted()) // ignore interrupts while waiting
* wasInterrupted = true;
* }
* waiters.remove();
* if (wasInterrupted) // reassert interrupt status on exit
* current.interrupt();
* }
* public void unlock() {
* locked.set(false);
* LockSupport.unpark(waiters.peek());
* }
* }

# **Multithreading in Java**

**Multithreading in java** is a process of executing multiple threads simultaneously. Thread is basically a lightweight sub-process, a smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking. But we use multithreading than multiprocessing because threads share a common memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process. Java Multithreading is mostly used in games, animation etc.

### Advantages of Java Multithreading

1) It **doesn't block the user** because threads are independent and you can perform multiple operations at same time.

2) You **can perform many operations together so it saves time**.

3) Threads are **independent** so it doesn't affect other threads if exception occur in a single thread.

Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved by two ways:

* Process-based Multitasking(Multiprocessing)
* Thread-based Multitasking(Multithreading)

1) Process-based Multitasking (Multiprocessing)

* Each process have its own address in memory i.e. each process allocates separate memory area.
* Process is heavyweight.
* Cost of communication between the process is high.
* Switching from one process to another require some time for saving and loading registers, memory maps, updating lists etc.

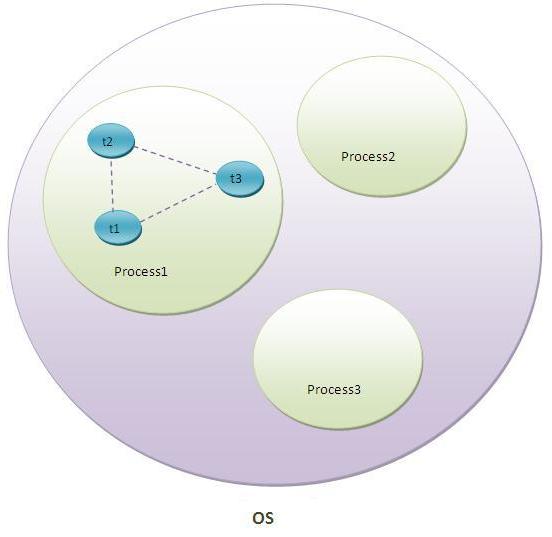
2) Thread-based Multitasking (Multithreading)

* Threads share the same address space.
* Thread is lightweight.
* Cost of communication between the thread is low.

## What is Thread in java

A thread is a lightweight sub process, a smallest unit of processing. It is a separate path of execution.

Threads are independent, if there occurs exception in one thread, it doesn't affect other threads. It shares a common memory area.



As shown in the above figure, thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the OS and one process can have multiple threads.

At a time only one thread is executed per process.

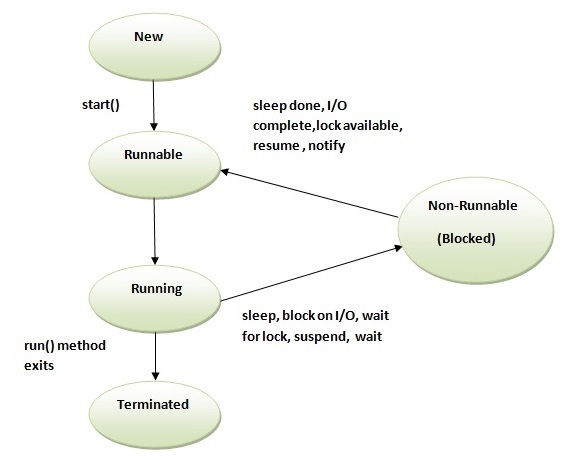
# **Life cycle of a Thread (Thread States)**

A thread can be in one of the five states. According to sun, there is only 4 states in **thread life cycle in java** new, runnable, non-runnable and terminated. There is no running state.

But for better understanding the threads, we are explaining it in the 5 states.

The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:

1. New
2. Runnable
3. Running
4. Non-Runnable (Blocked)
5. Terminated



|  |
| --- |
| 1) New The thread is in new state if you create an instance of Thread class but before the invocation of start() method. |

### 2) Runnable

The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.

### 3) Running

The thread is in running state if the thread scheduler has selected it.

### 4) Non-Runnable (Blocked)

This is the state when the thread is still alive, but is currently not eligible to run.

### 5) Terminated

A thread is in terminated or dead state when its run() method exits.

There are two ways to create a thread:

1. By extending Thread class
2. By implementing Runnable interface.

Thread class:

|  |
| --- |
| Thread class provide constructors and methods to create and perform operations on a thread.Thread class extends Object class and implements Runnable interface. |

Commonly used Constructors of Thread class:

|  |
| --- |
| * Thread() * Thread(String name) * Thread(Runnable r) * Thread(Runnable r,String name) |

Commonly used methods of Thread class:

|  |
| --- |
| 1. **public void run():**is used to perform action for a thread. 2. **public void start():**starts the execution of the thread.JVM calls the run() method on the thread. 3. **public void sleep(long miliseconds):**Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds. 4. **public void join():**waits for a thread to die. 5. **public void join(long miliseconds):**waits for a thread to die for the specified miliseconds. 6. **public int getPriority():**returns the priority of the thread. 7. **public int setPriority(int priority):**changes the priority of the thread. 8. **public String getName():**returns the name of the thread. 9. **public void setName(String name):**changes the name of the thread. 10. **public Thread currentThread():**returns the reference of currently executing thread. 11. **public int getId():**returns the id of the thread. 12. **public Thread.State getState():**returns the state of the thread. 13. **public boolean isAlive():**tests if the thread is alive. 14. **public void yield():**causes the currently executing thread object to temporarily pause and allow other threads to execute. 15. **public void suspend():**is used to suspend the thread(depricated). 16. **public void resume():**is used to resume the suspended thread(depricated). 17. **public void stop():**is used to stop the thread(depricated). 18. **public boolean isDaemon():**tests if the thread is a daemon thread. 19. **public void setDaemon(boolean b):**marks the thread as daemon or user thread. 20. **public void interrupt():**interrupts the thread. 21. **public boolean isInterrupted():**tests if the thread has been interrupted. 22. **public static boolean interrupted():**tests if the current thread has been interrupted. |

Runnable interface:

|  |
| --- |
| The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run(). |

|  |
| --- |
| 1. **public void run():**is used to perform action for a thread. |

Starting a thread:

|  |
| --- |
| **start() method** of Thread class is used to start a newly created thread. It performs following tasks:   * A new thread starts(with new callstack). * The thread moves from New state to the Runnable state. * When the thread gets a chance to execute, its target run() method will run. |

1) Java Thread Example by extending Thread class

1. **class** Multi **extends** Thread{
2. **public** **void** run(){
3. System.out.println("thread is running...");
4. }
5. **public** **static** **void** main(String args[]){
6. Multi t1=**new** Multi();
7. t1.start();
8. }
9. }

Output:thread is running...

2) Java Thread Example by implementing Runnable interface

1. **class** Multi3 **implements** Runnable{
2. **public** **void** run(){
3. System.out.println("thread is running...");
4. }
6. **public** **static** **void** main(String args[]){
7. Multi3 m1=**new** Multi3();
8. Thread t1 =**new** Thread(m1);
9. t1.start();
10. }
11. }

Output:thread is running...

|  |
| --- |
| If you are not extending the Thread class, your class object would not be treated as a thread object. So you need to explicitly create Thread class object. We are passing the object of your class that implements Runnable so that your class run() method may execute. |

# **Thread Scheduler in Java**

**Thread scheduler** in java is the part of the JVM that decides which thread should run.

There is no guarantee that which runnable thread will be chosen to run by the thread scheduler.

Only one thread at a time can run in a single process.

The thread scheduler mainly uses preemptive or time slicing scheduling to schedule the threads.

### Difference between preemptive scheduling and time slicing

Under preemptive scheduling, the highest priority task executes until it enters the waiting or dead states or a higher priority task comes into existence. Under time slicing, a task executes for a predefined slice of time and then reenters the pool of ready tasks. The scheduler then determines which task should execute next, based on priority and other factors.

# **Sleep method in java**

The sleep() method of Thread class is used to sleep a thread for the specified amount of time.

## Syntax of sleep() method in java

The Thread class provides two methods for sleeping a thread:

* public static void sleep(long miliseconds)throws InterruptedException
* public static void sleep(long miliseconds, int nanos)throws InterruptedException

## Example of sleep method in java

1. **class** TestSleepMethod1 **extends** Thread{
2. **public** **void** run(){
3. **for**(**int** i=1;i<5;i++){
4. **try**{Thread.sleep(500);}**catch**(InterruptedException e){System.out.println(e);}
5. System.out.println(i);
6. }
7. }
8. **public** **static** **void** main(String args[]){
9. TestSleepMethod1 t1=**new** TestSleepMethod1();
10. TestSleepMethod1 t2=**new** TestSleepMethod1();
12. t1.start();
13. t2.start();
14. }
15. }

Output:

1

1

2

2

3

3

4

4

As you know well that at a time only one thread is executed. If you sleep a thread for the specified time,the thread shedular picks up another thread and so on.

No. After starting a thread, it can never be started again. If you does so, an IllegalThreadStateException is thrown. In such case, thread will run once but for second time, it will throw exception.

Let's understand it by the example given below:

1. **public** **class** TestThreadTwice1 **extends** Thread{
2. **public** **void** run(){
3. System.out.println("running...");
4. }
5. **public** **static** **void** main(String args[]){
6. TestThreadTwice1 t1=**new** TestThreadTwice1();
7. t1.start();
8. t1.start();
9. }
10. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestThreadTwice1)

running

Exception in thread "main" java.lang.IllegalThreadStateException

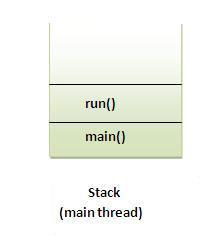
# **What if we call run() method directly instead start() method?**

|  |
| --- |
| * Each thread starts in a separate call stack. * Invoking the run() method from main thread, the run() method goes onto the current call stack rather than at the beginning of a new call stack. |

1. **class** TestCallRun1 **extends** Thread{
2. **public** **void** run(){
3. System.out.println("running...");
4. }
5. **public** **static** **void** main(String args[]){
6. TestCallRun1 t1=**new** TestCallRun1();
7. t1.run();//fine, but does not start a separate call stack
8. }
9. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestCallRun1)

Output:running...

 ***Problem if you direct call run() method***

1. **class** TestCallRun2 **extends** Thread{
2. **public** **void** run(){
3. **for**(**int** i=1;i<5;i++){
4. **try**{Thread.sleep(500);}**catch**(InterruptedException e){System.out.println(e);}
5. System.out.println(i);
6. }
7. }
8. **public** **static** **void** main(String args[]){
9. TestCallRun2 t1=**new** TestCallRun2();
10. TestCallRun2 t2=**new** TestCallRun2();
12. t1.run();
13. t2.run();
14. }
15. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestCallRun2)

Output:1

2

3

4

5

1

2

3

4

5

|  |
| --- |
| As you can see in the above program that there is no context-switching because here t1 and t2 will be treated as normal object not thread object. |

# **The join() method**

The join() method waits for a thread to die. In other words, it causes the currently running threads to stop executing until the thread it joins with completes its task.

t1.join(1000); // other threads will wait until t1 is dead or 1000 millis passed

t1.join(0); // it means wait until the thread is dead

t1.join(); // it means wait until the thread is dead

class TestJoinMethod2 extends Thread {

public void run() {

System.out.println("id:"+this.getId());

for (int i = 1; i <= 5; i++) {

try {

Thread.sleep(500);

} catch (Exception e) {

System.out.println(e);

}

System.out.println(i);

}

}

public static void main(String args[]) {

TestJoinMethod2 t1 = new TestJoinMethod2();

TestJoinMethod2 t2 = new TestJoinMethod2();

TestJoinMethod2 t3 = new TestJoinMethod2();

t1.start();

try {

t1.join(1000);

// t1.join(0);

// t1.join();

} catch (Exception e) {

System.out.println(e);

}

t2.start();

t3.start();

}

}

Output:1

2

3

4

5

1

1

2

2

3

3

4

4

5

5

**Priority of a Thread (Thread Priority):**

Each thread have a priority. Priorities are represented by a number between 1 and 10. In most cases, thread schedular schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.

3 constants defiend in Thread class:

public static int MIN\_PRIORITY

public static int NORM\_PRIORITY

public static int MAX\_PRIORITY

Default priority of a thread is 5 (NORM\_PRIORITY). The value of MIN\_PRIORITY is 1 and the value of MAX\_PRIORITY is 10.

class TestMultiPriority1 extends Thread {

public void run() {

System.out.println("running thread name is:" + Thread.currentThread().getName());

System.out.println("running thread priority is:" + Thread.currentThread().getPriority());

}

public static void main(String args[]) {

TestMultiPriority1 m1 = new TestMultiPriority1();

TestMultiPriority1 m2 = new TestMultiPriority1();

m1.setPriority(Thread.MIN\_PRIORITY);

m2.setPriority(Thread.MAX\_PRIORITY);

m1.start();

try {

Thread.currentThread().sleep(10);

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

m2.start();

}

}

running thread name is:Thread-0

running thread name is:Thread-1

running thread priority is:1

running thread priority is:10

**Daemon Thread in Java**

A daemon thread is a thread that does not prevent the JVM from exiting when the program finishes but the thread is still running. An example for a daemon thread is the garbage collection.

You can use the setDaemon(boolean) method to change the Thread daemon properties before the thread starts.

.when the program finishes." What that means is, if the program does not explicitly kill the JVM, then the JVM will automatically kill itself when the last *non*-daemon thread ends. Normal threads define "when the program exits." Daemon threads don't.

So this line thread that does not prevent the JVM from exiting when the program finishes but the thread is still running basically means the JVM process that started the thread doesn't care if the daemon thread finished executing or not, it will just end itself if all the normal threads have finished execution.

When a new thread is created it inherits the daemon status of its parent.

Normal thread and daemon threads differ in what happens when they exit. When the JVM halts any remaining daemon threads are abandoned:

finally blocks are not executed,

stacks are not unwound - the JVM just exits.

Due to this reason daemon threads should be used sparingly and it is dangerous to use them for tasks that might perform any sort of I/O.

Daemon threads are like a service providers for other threads or objects running in the same process as the daemon thread. Daemon threads are used for background supporting tasks and are only needed while normal threads are executing. If normal threads are not running and remaining threads are daemon threads then the interpreter exits.

For example, the HotJava browser uses up to four daemon threads named "Image Fetcher" to fetch images from the file system or network for any thread that needs one.

Daemon threads are typically used to perform services for your application/applet (such as loading the "fiddley bits"). The core difference between user threads and daemon threads is that the JVM will only shut down a program when all user threads have terminated. Daemon threads are terminated by the JVM when there are no longer any user threads running, including the main thread of execution.

**Daemon thread in java** is a service provider thread that provides services to the user thread. Its life depend on the mercy of user threads i.e. when all the user threads dies, JVM terminates this thread automatically.There are many java daemon threads running automatically e.g. gc, finalizer etc. Points to remember for Daemon Thread in Java

It provides services to user threads for background supporting tasks. It has no role in life than to serve user threads.

Its life depends on user threads.

It is a low priority thread.

Why JVM terminates the daemon thread if there is no user thread?

The sole purpose of the daemon thread is that it provides services to user thread for background supporting task. If there is no user thread, why should JVM keep running this thread. That is why JVM terminates the daemon thread if there is no user thread.

Methods for Java Daemon thread by Thread class

The java.lang.Thread class provides two methods for java daemon thread.

No. Method Description

1) public void setDaemon(boolean status) is used to mark the current thread as daemon thread or user thread.

2) public boolean isDaemon() is used to check that current is daemon.

setDaemon property should be set before starting the thread otherwise it will throw IllegalThreadStateException.

# **Java Thread Pool**

**Java Thread pool** represents a group of worker threads that are waiting for the job and reuse many times.

In case of thread pool, a group of fixed size threads are created. A thread from the thread pool is pulled out and assigned a job by the service provider. After completion of the job, thread is contained in the thread pool again.

Better performance It saves time because there is no need to create new thread.

It is used in Servlet and JSP where container creates a thread pool to process the request.

Most of the executor implementations in java.util.concurrent use thread pools, which consist of worker threads. This kind of thread exists separately from the Runnable and Callable tasks it executes and is often used to execute multiple tasks.

Using worker threads minimizes the overhead due to thread creation. Thread objects use a significant amount of memory, and in a large-scale application, allocating and deallocating many thread objects creates a significant memory management overhead.

One common type of thread pool is the fixed thread pool. This type of pool always has a specified number of threads running; if a thread is somehow terminated while it is still in use, it is automatically replaced with a new thread. Tasks are submitted to the pool via an internal queue, which holds extra tasks whenever there are more active tasks than threads.

An important advantage of the fixed thread pool is that applications using it degrade gracefully. To understand this, consider a web server application where each HTTP request is handled by a separate thread. If the application simply creates a new thread for every new HTTP request, and the system receives more requests than it can handle immediately, the application will suddenly stop responding to all requests when the overhead of all those threads exceed the capacity of the system. With a limit on the number of the threads that can be created, the application will not be servicing HTTP requests as quickly as they come in, but it will be servicing them as quickly as the system can sustain.

A simple way to create an executor that uses a fixed thread pool is to invoke the newFixedThreadPool factory method in java.util.concurrent.Executors. This class also provides the following factory methods:

The newCachedThreadPool method creates an executor with an expandable thread pool. This executor is suitable for applications that launch many short-lived tasks.

The newSingleThreadExecutor method creates an executor that executes a single task at a time.

Several factory methods are ScheduledExecutorService versions of the above executors.

If none of the executors provided by the above factory methods meet your needs, constructing instances of java.util.concurrent.ThreadPoolExecutor or java.util.concurrent.ScheduledThreadPoolExecutor will give you additional options.

Both **ExecutorService**and **ScheduledExecutorService**are interfaces, with ScheduledExecutorService extending ExecutorService. They provide functionality for concurrent execution of Runnable and Callable tasks. The difference is that ScheduledExecutorService provides additional methods to execute a task with delay or/and every fixed time period.

**ExecutorService**provides an interface for submitting a Runnable or Callable task or a collection of these tasks for execution once, also providing methods for the proper shutdown of the executor. Some of the notable classes that implement this interface are ForkJoinPool and ThreadPoolExecutor.

Example of usage from Java documentation:

1. **class** NetworkService implements Runnable {
2. **private** final ServerSocket serverSocket;
3. **private** final ExecutorService pool;
5. **public** NetworkService(**int** port, **int** poolSize)
6. throws IOException {
7. serverSocket = **new** ServerSocket(port);
8. pool = Executors.newFixedThreadPool(poolSize);
9. }
11. **public** **void** run() { // run the service
12. **try** {
13. **for** (;;) {
14. pool.execute(**new** Handler(serverSocket.accept()));
15. }
16. } **catch** (IOException ex) {
17. pool.shutdown();
18. }
19. }
20. }
22. **class** Handler implements Runnable {
23. **private** final Socket socket;
24. Handler(Socket socket) { **this**.socket = socket; }
25. **public** **void** run() {
26. // read and service request on socket
27. }
28. }

**ScheduledExecutorService**is an extension of an ExecutorService, so it provides the same functionality with the addition of several methods that deal with scheduling execution. These are schedule with input parameters of a Runnable/Callable, delay, and TimeUnit. This will execute the provided task after a delay of set time units. There is also scheduleAtFixedRate that adds a period of time to schedule. Each period of time the task will be scheduled for execution. The most notable class that implements this interface is ScheduledThreadPoolExecutor.

Example of usage from Java documentation:

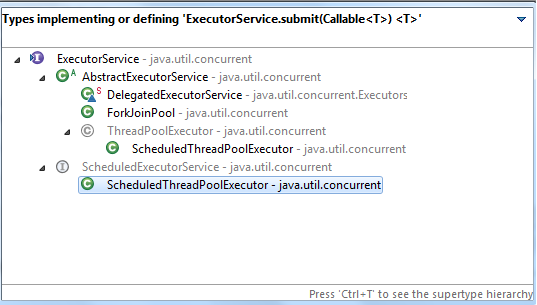
1. **import** **static** java.util.concurrent.TimeUnit.\*;
2. **class** BeeperControl {
3. **private** final ScheduledExecutorService scheduler =
4. Executors.newScheduledThreadPool(1);
6. **public** **void** beepForAnHour() {
7. final Runnable beeper = **new** Runnable() {
8. **public** **void** run() { System.**out**.println("beep"); }
9. };
10. final ScheduledFuture<?> beeperHandle =
11. scheduler.scheduleAtFixedRate(beeper, 10, 10, SECONDS);
12. scheduler.schedule(**new** Runnable() {
13. **public** **void** run() { beeperHandle.cancel(**true**); }
14. }, 60 \* 60, SECONDS);
15. }
16. }

public abstract class AbstractExecutorService implements ExecutorService

public class ThreadPoolExecutor extends AbstractExecutorService

public class ScheduledThreadPoolExecutor extends ThreadPoolExecutor implements ScheduledExecutorService

public interface ScheduledExecutorService extends ExecutorService



What is CyclicBarrier ?

CyclicBarrier in Java is a synchronizer introduced in JDK 5 on java.util.Concurrent package along with other concurrent utility like Counting Semaphore, BlockingQueue, ConcurrentHashMap etc. CyclicBarrier is similar to CountDownLatch which we have seen in the last article What is CountDownLatch in Java and allows multiple threads to wait for each other (barrier) before proceeding. The difference between CountDownLatch and CyclicBarrier is an also very popular multi-threading interview question in Java. CyclicBarrier is a natural requirement for a concurrent program because it can be used to perform final part of the task once individual tasks are completed. All threads which wait for each other to reach barrier are called parties, CyclicBarrier is initialized with a number of parties to wait and threads wait for each other by calling CyclicBarrier.await() method which is a blocking method in Java and blocks until all Thread or parties call await(). In general calling await() is shout out that Thread is waiting on the barrier. await() is a blocking call but can be timed out or Interrupted by other thread. In this Java concurrency tutorial, we will see What is CyclicBarrier in Java and an example of CyclicBarrier on which three Threads will wait for each other before proceeding further.

Difference between CountDownLatch and CyclicBarrier in Java

In our last article, we have to see how CountDownLatch can be used to implement multiple threads waiting for each other. If you look at CyclicBarrier it also the does the same thing but there is different you can not reuse CountDownLatch once the count reaches zero while you can reuse CyclicBarrier by calling reset() method which resets Barrier to its initial State. What it implies that CountDownLatch is a good for one-time events like application start-up time and CyclicBarrier can be used to in case of the recurrent event e.g. concurrently calculating a solution of the big problem etc. If you like to learn more about threading and concurrency in Java you can also check my post on When to use Volatile variable in Java and How Synchronization works in Java.

CyclicBarrier in Java – Example

Java CyclicBarrier Example and Tutorial programNow we know what is CyclicBarrier in Java and it's time to see an example of CyclicBarrier in Java. Here is a simple example of CyclicBarrier in Java on which we initialize CyclicBarrier with 3 parties, means in order to cross barrier, 3 thread needs to call await() method. each thread calls await method in short duration but they don't proceed until all 3 threads reached the barrier, once all thread reach the barrier, barrier gets broker and each thread started their execution from that point. Its much clear with the output of following example of CyclicBarrier in Java:

import java.util.concurrent.BrokenBarrierException;

import java.util.concurrent.CyclicBarrier;

import java.util.logging.Level;

import java.util.logging.Logger;

/\*\*

\* Java program to demonstrate how to use CyclicBarrier in Java. CyclicBarrier is a

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

\*

\* @author Javin Paul

\*/

public class CyclicBarrierExample {

//Runnable task for each thread

private static class Task implements Runnable {

private CyclicBarrier barrier;

public Task(CyclicBarrier barrier) {

this.barrier = barrier;

}

@Override

public void run() {

try {

System.out.println(Thread.currentThread().getName() + " is waiting on barrier");

barrier.await();

System.out.println(Thread.currentThread().getName() + " has crossed the barrier");

} catch (InterruptedException ex) {

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

} catch (BrokenBarrierException ex) {

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

}

}

}

public static void main(String args[]) {

//creating CyclicBarrier with 3 parties i.e. 3 Threads needs to call await()

final CyclicBarrier cb = new CyclicBarrier(3, new Runnable(){

@Override

public void run(){

//This task will be executed once all thread reaches barrier

System.out.println("All parties are arrived at barrier, lets play");

}

});

//starting each of thread

Thread t1 = new Thread(new Task(cb), "Thread 1");

Thread t2 = new Thread(new Task(cb), "Thread 2");

Thread t3 = new Thread(new Task(cb), "Thread 3");

t1.start();

t2.start();

t3.start();

}

}

Output:

Thread 1 is waiting on barrier

Thread 3 is waiting on barrier

Thread 2 is waiting on barrier

All parties have arrived at barrier, lets play

Thread 3 has crossed the barrier

Thread 1 has crossed the barrier

Thread 2 has crossed the barrier

When to use CyclicBarrier in Java

Given the nature of CyclicBarrier it can be very handy to implement map reduce kind of task similar to fork-join framework of Java 7, where a big task is broker down into smaller pieces and to complete the task you need output from individual small task e.g. to count population of India you can have 4 threads which count population from North, South, East, and West and once complete they can wait for each other, When last thread completed their task, Main thread or any other thread can add result from each zone and print total population. You can use CyclicBarrier in Java :

1) To implement multi player game which can not begin until all player has joined.

2) Perform lengthy calculation by breaking it into smaller individual tasks, In general, to implement Map reduce technique.

**Important point of CyclicBarrier in Java**

1. CyclicBarrier can perform a completion task once all thread reaches to the barrier, this can be provided while creating CyclicBarrier.

2. If CyclicBarrier is initialized with 3 parties means 3 thread needs to call await method to break the barrier.

3. The thread will block on await() until all parties reach to the barrier, another thread interrupt or await timed out.

4. If another thread interrupts the thread which is waiting on barrier it will throw BrokernBarrierException as shown below:

java.util.concurrent.BrokenBarrierException

at java.util.concurrent.CyclicBarrier.dowait(CyclicBarrier.java:172)

at java.util.concurrent.CyclicBarrier.await(CyclicBarrier.java:327)

5.CyclicBarrier.reset() put Barrier on its initial state, other thread which is waiting or not yet reached barrier will terminate with java.util.concurrent.BrokenBarrierException.

That's all on What is CyclicBarrier in Java When to use CyclicBarrier in Java and a Simple Example of How to use CyclicBarrier in Java . We have also seen the difference between CountDownLatch and CyclicBarrier in Java and got some idea where we can use CyclicBarrier in Java Concurrent code.

Read more: <http://javarevisited.blogspot.com/2012/07/cyclicbarrier-example-java-5-concurrency-tutorial.html#ixzz4x9gMmBaU>

# CyclicBarrier: concordinating the stages of a multithreaded operation

The [CountDownLatch](https://www.javamex.com/tutorials/threads/CountDownLatch.shtml) class is useful for various types of "one-off" thread coordination, in particular *setting threads off* together. However, it has at least two features that can be inconvenient in certain situations:

* a given CountDownLatch can only be used once, making it inconvenient for operations that occur in *stages*, with intermediate results from the different threads needing to be amalgamated between stages;
* the CountDownLatch doesn't explicitly allow one thread to tell the others to "stop waiting"1, which is sometimes useful, for example, if an error occurs in one of the threads.

The CyclicBarrier is generally more useful than CountDownLatch in cases where:

* a **multithreaded operation occurs in *stages* or *iterations***, and;
* a **single-threaded operation is required between stages/iterations**, for example, to **combine the results** of the previous multithreaded portion.

## Overview of CyclicBarrier

Firstly, the barrier is constructed with the following:

* the **number of threads** that will be participating in the parallel operation;
* optionally, an **amalgamation routine** to run at the end of each stage/iteration.

Then, at each stage (or on each iteration) of the operation:

* each thread carries out its portion of the work;
* after doing its portion of the work, each thread **calls the barrier's await()** method;
* the await() method **returns *only* when**:
  + *all* threads have called await();
  + the **amalgamation** method has run (the barrier calls this on the last thread to call await() before releasing the awaiting threads).
* if *any* of the threads is **interrupted or times out** while waiting for the barrier, then **the barrier is "broken"** and *all* other waiting threads receive a BrokenBarrierException.

in other words, this last point means there is a mechanism for an error/timeout in one of the worker threads to "ripple out" to all threads and for the operation to halt, or for the operation to be interrupted externally by interrupting just *one* of the threads.

**CyclicBarrier example: error handling**

In our introduction to CyclicBarrier, we mentioned that one potentially useful feature is that if, while waiting at the barrier, one of the waiting threads is interrupted or times out, then all of the other waiting threads wake up with a BrokenBarrierException.

But what happens if an exception occurs during a thread's execution of the actual task code— in other words, while it isn't in the await() call? Well in this case, one solution is to deliberately interrupt the thread and then call await()! Our thread will immediately return from the await() call with an InterruptedException (which we obviously don't care about), and all other threads will wake up from await() with a BrokenBarrierException.

The pattern looks as follows:

private volatile Throwable error;

...

public void run() {

try {

... perform task

barrier.await();

... perform task

barrier.await();

} catch (InterruptedException e) {

// don't care-- controller thread will get

// BrokenBarrierException

} catch (BrokenBarrierException e) {

// ditto

} catch (Throwable t) {

// some other exception occurred during our task

error = t;

Thread.currentThread().interrupt();

try {

barrier.await();

} catch (Exception e) {}

}

}

This is one of the few cases where we probably just want to "swallow" various exceptions, since the BrokenBarrierException will be thrown in all the other threads, including our "controller" thread. In the case of catching some other exception because it occurred during our task code, we may want to save it to a variable (as the error variable here). Then the controller thread, on being awoken with a BrokenBarrierException, can find the thread with the error variable set and throw the corresponding exception to the caller.

Error in the result amalgamation Runnable

There's actually a race condition that we haven't dealt with in the above code. When the last thread to get to the barrier calls the await() method, the barrier's Runnable (if any) is run before the await() method returns. If an unchecked exception is thrown by that Runnable, then the barrier is broken— in other words, other waiting threads potentially signalled— before that exception is thrown up to the caller of await(). In other words, the controller thread could be awoken before the error variable has been set.

To get round this problem, albeit in a slightly inelegant way, we can set another variable specially for errors that occur inside the Runnable. So that method looks as follows:

private volatile RuntimeException errorInAmalgRunnable;

CyclicBarrier barrier = new CyclicBarrier(noThreads, new Runnable() {

public void run() {

try {

... amalgamate results ...

} catch (RuntimeException e) {

errorInAmalgRunnable = e;

throw e;

}

}

});

# [What is CountDownLatch in Java - Concurrency Example Tutorial](http://javarevisited.blogspot.in/2012/07/countdownlatch-example-in-java.html)

**What is CountDownLatch in Java**

CountDownLatch in Java is a kind of synchronizer which allows one Thread  to wait for one or more Threads before starts processing. This is very crucial requirement and often needed in server side core Java application and having this functionality built-in as CountDownLatch greatly simplifies the development. CountDownLatch in Java is introduced on Java 5 along with other concurrent utilities like [CyclicBarrier](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), [ConcurrentHashMap](http://javarevisited.blogspot.sg/2011/04/difference-between-concurrenthashmap.html) and [BlockingQueue](http://javarevisited.blogspot.sg/2012/02/producer-consumer-design-pattern-with.html) in java.util.concurrent package. In this Java concurrency tutorial we will  what is CountDownLatch in Java, How CountDownLatch works in Java, an example of **CountDownLatch in Java** and finally some worth noting points about this concurrent utility. You can also implement same functionality using  [wait and notify mechanism](http://javarevisited.blogspot.sg/2011/05/wait-notify-and-notifyall-in-java.html) in Java but it requires lot of code and getting it write in first attempt is tricky,  With CountDownLatch it can  be done in just few lines. CountDownLatch also allows flexibility on number of thread for which [main thread](http://javarevisited.blogspot.sg/2011/12/main-public-static-java-void-method-why.html)should wait, It can wait for one thread or n number of thread, there is not much change on code.  Key point is that you need to figure out where to use CountDownLatch in Java application which is not difficult if you understand *What is CountDownLatch in Java*, What does CountDownLatch do and How CountDownLatch works in Java.

**How CountDownLatch works in Java**

Now we know What is CountDownLatch in Java, its time to find out How CountDownLatch works in Java. CountDownLatch works in latch principle,  main thread will wait until Gate is open. One [thread waits](http://javarevisited.blogspot.sg/2012/02/why-wait-notify-and-notifyall-is.html) for n number of threads specified while creating CountDownLatch in Java. Any thread, usually main thread of application,  which calls CountDownLatch.await() will wait until count reaches zero or its interruptedby another Thread. All other thread are required to do count down by calling CountDownLatch.countDown() once they are completed or ready to the job. as soon as count reaches zero, [Thread](http://javarevisited.blogspot.sg/2011/02/how-to-implement-thread-in-java.html) awaiting starts running. One of the disadvantage of CountDownLatch is that **its not reusable once count reaches to zero** you can not use CountDownLatch any more, but don't worry Java concurrency API has another concurrent utility called [CyclicBarrier](http://javarevisited.blogspot.sg/2012/07/cyclicbarrier-example-java-5-concurrency-tutorial.html) for such requirements.

## CountDownLatch Exmaple in Java

In this section we will see a full featured real world example of using *CountDownLatch in Java*. In following **CountDownLatch example**, Java program requires 3 services namely CacheService, AlertService  and ValidationService  to be started and ready before application can handle any [request](http://javarevisited.blogspot.sg/2011/09/servlet-interview-questions-answers.html) and this is achieved by using CountDownLatch in Java.

**import** java.util.Date;  
**import** java.util.concurrent.CountDownLatch;  
**import** java.util.logging.Level;  
**import** java.util.logging.Logger;  
  
/\*\*  
 \* Java program to demonstrate How to use CountDownLatch in Java. CountDownLatch is

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

 \* as illustrated in this CountDownLatch Example  
 \*   
 \* @author Javin Paul  
 \*/  
**public** **class** CountDownLatchDemo {  
  
    **public** **static** **void** main(**String** args[]) {  
       **final** **CountDownLatch** latch = **new** **CountDownLatch**(3);  
       **Thread** cacheService = **new** **Thread**(**new** Service("CacheService", 1000, latch));  
       **Thread** alertService = **new** **Thread**(**new** Service("AlertService", 1000, latch));  
       **Thread** validationService = **new** **Thread**(**new** Service("ValidationService", 1000, latch));  
        
       cacheService.start(); //separate thread will initialize CacheService  
       alertService.start(); //another thread for AlertService initialization  
       validationService.start();  
        
       *// application should not start processing any thread until all service is up*

*// and ready to do there job.*  
       *// Countdown latch is idle choice here, main thread will start with count 3*

*// and wait until count reaches zero. each thread once up and read will do*

*// a count down. this will ensure that main thread is not started processing*

*// until all services is up.*  
        
       *//count is 3 since we have 3 Threads (Services)*  
        
       **try**{  
            latch.await();  //main thread is waiting on CountDownLatch to finish  
            **System**.out.println("All services are up, Application is starting now");  
       }**catch**(**InterruptedException** ie){  
           ie.printStackTrace();  
       }  
        
    }  
    
}  
  
/\*\*  
 \* Service class which will be executed by Thread using CountDownLatch synchronizer.  
 \*/  
**class** Service **implements** **Runnable**{  
    **private** **final** **String** name;  
    **private** **final** **int** timeToStart;  
    **private** **final** **CountDownLatch** latch;  
    
    **public** Service(**String** name, **int** timeToStart, **CountDownLatch** latch){  
        **this**.name = name;  
        **this**.timeToStart = timeToStart;  
        **this**.latch = latch;  
    }  
    
    @**Override**  
    **public** **void** run() {  
        **try** {  
            **Thread**.sleep(timeToStart);  
        } **catch** (**InterruptedException** ex) {  
            **Logger**.getLogger(Service.**class**.getName()).log(**Level**.SEVERE, **null**, ex);  
        }  
        **System**.out.println( name + " is Up");  
        latch.countDown(); //reduce count of CountDownLatch by 1  
    }  
    
}  
  
**Output:**  
ValidationService is Up  
AlertService is Up  
CacheService is Up  
All services are up, Application is starting now

By looking at output of this CountDownLatch example in Java, you can see that Application is not started until all services started by individual Threads are completed.

### When should we use CountDownLatch in Java :

Use CountDownLatch when one of Thread like [main thread](http://javarevisited.blogspot.sg/2011/12/main-public-static-java-void-method-why.html), require to wait for one or more thread to complete, before its start doing processing. Classical example of using CountDownLatch in Java  is any server side core Java application which uses services architecture,  where multiple services is provided by multiple threads and application can not start processing  until all services have started successfully as shown in ourCountDownLatch example.

**CountDownLatch in Java – Things to remember**

Few points about Java CountDownLatch which is worth remembering:

1) You can not reuse CountDownLatch once count is reaches to zero, this is the main [difference between CountDownLatch and CyclicBarrier](http://javarevisited.blogspot.sg/2012/07/cyclicbarrier-example-java-5-concurrency-tutorial.html), which is frequently asked in [core Java interviews](http://javarevisited.blogspot.sg/2011/04/top-20-core-java-interview-questions.html) and [multi-threading  interviews](http://javarevisited.blogspot.sg/2011/07/java-multi-threading-interview.html).

2) Main Thread wait on Latch by calling CountDownLatch.await() method while other thread calls CountDownLatch.countDown() to inform that they have completed.

That’s all on **What is CountDownLatch in Java**, What does CountDownLatch do in Java, How CountDownLatch works in Java along with a real life CountDownLatch example in Java. This is a very useful concurrency utility and if you master *when to use CountDownLatch* and how to use CountDownLatch you will be able to reduce good amount of complex concurrency control code written using wait and notify in Java.

# [What is blocking methods in Java and how do deal with it?](http://javarevisited.blogspot.in/2012/02/what-is-blocking-methods-in-java-and.html)

**Blocking methods in Java** are those methods which **block the executing thread** until their operation finished. A famous

example of blocking method is InputStream read() method which blocks until all data from InputStream has been read

completely. A correct understanding of blocking methods are required if you are serious towards Java programming especially in early days because if not used carefully blocking method can freeze GUIs, hung your program and become non-responsive for a longer duration of time. In this post, we will see **What is Blocking methods in Java**, Examples of Blocking methods and Some **best practices around blocking methods** and how to use blocking methods in Java.

## What is Blocking methods in Java

[Example of blocking method in Java](http://javarevisited.blogspot.com/2011/11/decorator-design-pattern-java-example.html)As I said **Blocking methods** are those which blocks the current executing thread from further operation until function returns. So if you have just one thread in your program e.g. [main thread](http://javarevisited.blogspot.com/2011/12/main-public-static-java-void-method-why.html) and you call any blocking method e.g. reading from InputStream, your program will be blocked until reading of file finished. Javadoc clearly mention whether an API call is blocking or not but **most of  java IO methods are blocking**.   
  
If you have been doing GUI programming in Java using Swing than knowledge of blocking methods becomes even more important for you, because no body likes freezing or non responsive GUI. methods like [invokeAndWait](http://javarevisited.blogspot.com/2011/09/invokeandwait-invokelater-swing-example.html) are blocking in nature and should be used only when you are performing some operation on which user should wait for result. In most simple terms *blocking means your code in next line will not be executed* because Thread which is executing blocking function is waiting for method to return. here is a code example which help you to understand blocking calls:

public class BlcokingCallTest {

    public static void main(String args[]) throws FileNotFoundException, IOException  {

      System.out.println("Calling blocking method in Java");

      int input = System.in.read();

      System.out.println("Blocking method is finished");

    }

}

In this code example after executing first print statement your program will be blocked and will not execute second print statement until you enter some characters in console and press enter because **read() method blocks** until some input is available for reading.

## Examples of blocking methods in Java:

There are lots of blocking methods in Java API and good thing is that javadoc clearly informs about it and always mention whether a method call is blocking or not. In General methods related to [reading or writing file](http://javarevisited.blogspot.com/2011/12/read-and-write-text-file-java.html), opening network connection, reading from Socket, updating GUI synchronously uses blocking call. here are some of most common methods in Java which are blocking in nature:

1) **InputStream.read()** which blocks until input data is available, an exception is thrown or end of Stream is detected.

2) **ServerSocket.accept()** which listens for incoming socket connection in Java and blocks until a connection is made.

3) **InvokeAndWait()** wait until code is executed from [Event Dispatcher thread](http://javarevisited.blogspot.com/2011/09/swing-interview-questions-answers-in.html).

## Disadvantages of blocking method:

**Blocking methods** poses significant threat to **scalability** of System. Imagine you are writing a client server application and you can only serve one client at a time because your code blocks. there is no way you can make use of that System and its not even using resources properly because you might have high speed CPU sitting idle waiting for something. Yes there are *ways to mitigate blocking* and using [multiple threads](http://javarevisited.blogspot.com/2011/02/how-to-implement-thread-in-java.html) for serving multiple clients is a classical solution of blocking call. Though most important aspect is design because a poorly designed system even if its multi-threaded can not scale beyond a point, if you are relying solely of number of Threads for scalability means it can not be more than few hundred or thousands since there is limit on number of thread [JVM](http://javarevisited.blogspot.com/2011/12/jre-jvm-jdk-jit-in-java-programming.html) can support. Java5 addresses this issue by adding **non blocking and asynchronous alternative of blocking IO calls** and those utility can be used to write high performance

servers application in core Java.

### Best practices while calling blocking method in Java:

*Blocking methods* are for a purpose or may be due to limitation of API but there are guidelines available in terms of common and best practices to deal with them. here I am listing some standard ways which I employ while using *blocking method in Java*

1) If you are writing GUI application may be in Swing **never call blocking method in Event dispatcher thread** or

in the event handler. for example if you are reading a file or opening a network connection when a button is clicked

don't do that on actionPerformed() method, instead just create another worker thread to do that job and return from

actionPerformed(). this will keep your GUI responsive, but again it depends upon design if the operation is something which requires user to wait than consider using invokeAndWait() for synchronous update.

2) Always let separate worker thread handles time consuming operations e.g. reading and writing to file, [database](http://javarevisited.blogspot.com/2012/01/improve-performance-java-database.html) or

socket.

3) Use **timeout** while calling blocking method. so if your blocking call doesn't return in specified time period, consider

aborting it and returning back but again this depends upon scenario. if you are using Executor Framework for managing

your worker threads, which is by the way recommended way than you can use Future object whose get() methods support timeout, but ensure that you properly terminate a blocking call.

4) Extension of first practices, don't call blocking methods on keyPressed() or paint() method which are supposed to

return as quickly as possible.

5) Use call-back functions to process result of a blocking call.

### A word of caution:

Though multi-threading is a workaround of blocking method it poses its own risk like [thread-safety](http://javarevisited.blogspot.com/2012/01/how-to-write-thread-safe-code-in-java.html) and race condition.

Java 5 also provides better alternatives of blocking IO methods wrapped in java.nio package.

That's all on **Blocking methods in Java** and some of best practices to use while calling blocking functions. let's know

what is your experience while using blocking IO methods and what standard code practices you follow while using these

methods.

## Important points:

1. If a Thread is blocked in a blocking method it remain in any of blocking state e.g. WAITING, BLOCKED or TIMED\_WAITING.

2. Some of the blocking method throws checked InterrupptedException which indicates that they may allow cancel the task and return before completion like Thread.sleep() or BlockingQueue.put() or take() throws InterruptedException.

3. interupt() method of Thread class can be used to interuupt a thread blocked inside blocking operation, but this is mere

a request not guarantee and works most of the time.

# [Producer Consumer Design Pattern with Blocking Queue Example in Java](http://javarevisited.blogspot.in/2012/02/producer-consumer-design-pattern-with.html)

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

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

## Real World Example of Producer Consumer Design Pattern

[Producer Consumer design pattern BlockingQueue example Java](http://javarevisited.blogspot.com/2011/11/ldap-authentication-active-directory.html)**Producer consumer pattern** is every where in real life and depict coordination and collaboration. Like one person is preparing food (Producer) while other one is serving food (Consumer), both will use shared table for putting food plates and taking food plates. Producer which is the person preparing food will wait if table is full and Consumer (Person who is serving food) will wait if table is empty. table is a shared object here. On Java library **Executor framework** itself implement Producer Consumer design pattern be separating responsibility of addition and execution of task.

## Benefit of Producer Consumer Pattern

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

is few of its benefit:

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

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

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

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

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

## Producer Consumer Problem in Multi-threading

**Producer-Consumer Problem** is also a [popular java interview question](http://javarevisited.blogspot.com/2011/04/top-20-core-java-interview-questions.html) where interviewer ask to implement producer consumer design pattern so that Producer should wait if Queue or bucket is full and Consumer should wait if queue or

bucket is empty. This problem can be implemented or solved by different ways in Java, classical way is using [wait and notify method](http://javarevisited.blogspot.com/2011/05/wait-notify-and-notifyall-in-java.html) to communicate between **Producer and Consumer thread** and blocking each of them on individual condition like full queue and empty queue. With introduction of **BlockingQueue** Data Structure in Java 5 Its now much simpler because BlockingQueue provides this control implicitly by introducing[blocking methods](http://javarevisited.blogspot.com/2012/02/what-is-blocking-methods-in-java-and.html) put() and take(). Now you don't require to use wait and notify to communicate between Producer and Consumer. BlockingQueue put() method will block if Queue is full in case of Bounded Queue and take() will block if Queue is empty. In next section we will see a *code example of Producer Consumer design pattern*.

## Using Blocking Queue to implement Producer Consumer Pattern

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

import java.util.concurrent.BlockingQueue;

import java.util.concurrent.LinkedBlockingQueue;

import java.util.logging.Level;

import java.util.logging.Logger;

public class **ProducerConsumerPattern** {

    public static void main(String args[]){

**//Creating shared object**

     BlockingQueue sharedQueue = new LinkedBlockingQueue();

**//Creating Producer and Consumer Thread**

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

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

**//Starting producer and Consumer thread**

     prodThread.start();

     consThread.start();

    }

}

**//Producer Class in java**

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

    private final **BlockingQueue** sharedQueue;

    public Producer(BlockingQueue sharedQueue) {

        this.sharedQueue = sharedQueue;

    }

    @Override

    public void run() {

        for(int i=0; i<10; i++){

            try {

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

                sharedQueue.put(i);

            } catch (InterruptedException ex) {

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

            }

        }

    }

}

**//Consumer Class in Java**

class Consumer implements Runnable{

    private final BlockingQueue sharedQueue;

    public Consumer (BlockingQueue sharedQueue) {

        this.sharedQueue = sharedQueue;

    }

    @Override

    public void run() {

        while(true){

            try {

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

            } catch (InterruptedException ex) {

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

            }

        }

    }

}

**Output:**

Produced: 0

Produced: 1

Consumed: 0

Produced: 2

Consumed: 1

Produced: 3

Consumed: 2

Produced: 4

Consumed: 3

Produced: 5

Consumed: 4

Produced: 6

Consumed: 5

Produced: 7

Consumed: 6

Produced: 8

Consumed: 7

Produced: 9

Consumed: 8

Consumed: 9

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

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

The javadoc for LinkedBlockingQueue says:

An optionally-bounded blocking queue based on linked nodes.[...]

The optional capacity bound constructor argument serves as a way to prevent excessive queue expansion. The capacity, if unspecified, is equal to Integer.MAX\_VALUE.

The javadoc of ArrayBlockingQueue says:

A bounded blocking queue backed by an array.[...]

This is a classic "bounded buffer", in which a fixed-sized array holds elements inserted by producers and extracted by consumers. Once created, the capacity cannot be increased

So, a linked blocking queue can be bounded or unbounded, whereas an ArrayBlockingQueue is always bounded.