**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.   
  
**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.   
  
**4)  When to use Runnable vs Thread in Java**

As we know we can implement thread either by extending Thread class or implementing Runnable interface, 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 Runnablethen extends Thread if you also want to extend another class   
  
**6)  What is the difference between start() and run() method of Thread class?**

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.   
  
**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.   
  
**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.     
  
**9)  What is Java Memory model?**

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.

**11) How does volatile variable works in Java?**

**A.** volatile is a keyword . Volatile guarantee not just limited to the variable but also all the variables [two threads](http://javarevisited.blogspot.sg/2011/02/how-to-implement-thread-in-java.html) see known as “happens before” relationship. Another important aspect of making a variable volatile is that compiler **will not reorder the variable** when switching from client to server configuration or while performing optimization.

 the volatile keyword in Java is used as an indicator to Java compiler and Thread that do not cache value of this variable and always read it from [main memory](http://javarevisited.blogspot.com/2011/05/java-heap-space-memory-size-jvm.html)  
The Java volatile keyword cannot be used with method or class and it can only be used with a variable.  
Java volatile keyword also guarantees visibility and ordering, after Java 5 write to any volatile variable happens before any read into the volatile variable

use of volatile keyword also prevents compiler or JVM from the reordering of code or moving away them from synchronization barrier.

Example :

|  |  |
| --- | --- |
| public class Singleton{  private static volatile Singleton \_instance; //volatilevariable  public static Singleton getInstance(){  if(\_instance == null){  synchronized(Singleton.class){  if(\_instance == null)  \_instance = new Singleton(); }  }  return \_instance; } | If you look at the code carefully you will be able to figure out: 1) We are only creating instance one time 2) We are creating instance lazily at the time of the first request comes |

If we do not make the \_instance variable volatile than the Thread which is creating instance of Singleton is not able to communicate other thread, that instance has been created until it comes out of the Singleton block, so if Thread A is creating Singleton instance and just after creation lost the CPU, all other thread will not be able to see value of \_instance as not null and they will believe its still [null](http://javarevisited.blogspot.sg/2012/06/common-cause-of-javalangnullpointerexce.html).

**When to use :**

1) You can use Volatile variable if you want to read and write long and [double](http://javarevisited.blogspot.sg/2011/10/convert-double-to-string-example.html) variable atomically. long and double both are [64 bit](http://javarevisited.blogspot.sg/2012/01/find-jvm-is-32-or-64-bit-java-program.html) data type and by default writing of long and double is not atomic and platform dependence. Many platform perform write in long and double variable 2 step, writing 32 bit in each step, due to this its possible for a Thread to see 32 bit from two different write. You can avoid this issue by making long and double variable volatile in Java  
  
2) A volatile variable can be used as an alternative way of achieving [synchronization in Java](http://javarevisited.blogspot.sg/2011/04/synchronization-in-java-synchronized.html) in some cases, like Visibility. with volatile variable, it's guaranteed that all reader thread will see updated value of the volatile variable once write operation completed, without volatile keyword different reader thread may see different values.

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

4) Another place where a volatile variable can be used is to fixing double checked locking in Singleton pattern  
  
**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.   
  
**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.   
  
**14) What happens when an Exception occurs in a thread?**   
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 ts 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 BlockingQueueIt implements Producer consumer pattern using wait and notify methods, which involves sharing objects between two threads.

**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.  
  
**17) Why wait, notify and notifyAll are not inside 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.

**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?**(answer)  
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 [static method](http://java67.blogspot.com/2012/11/what-is-static-class-variable-method.html) Thread.interrupted(), interrupt status is cleared. The non-static isInterrupted() method, which is used by one thread to query the interrupt status of another, does not change the interrupt status flag. By convention, any method that exits by throwing an InterruptedException clears interrupt status when it does so. However, it's always possible that interrupt status will immediately be set again, by another thread invoking interrupt  
  
**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.   
  
**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.

**25) What is thread pool? Why should you thread pool in Java?**(answer)  
Creating thread is expensive in terms of time and resource. If you create thread at time of request processing it will slow down your response time, also there is only a limited number of threads a process can create. To avoid both of these issues, a pool of thread is created when application starts-up and threads are reused for request processing. This pool of thread is known as "thread pool" and threads are known as worker thread. From JDK 1.5 release, Java API provides Executor framework, which allows you to create different types of thread pools e.g. single thread pool, which process one task at a time, fixed thread pool (a pool of fixed number of threads) or cached thread pool (an expandable thread pool suitable for applications with many short lived tasks). See this [article](http://javarevisited.blogspot.com/2013/07/how-to-create-thread-pools-in-java-executors-framework-example-tutorial.html) to learn more about thread pools in Java to prepare detailed answer of this question.  
  
**26) Write code to solve Producer Consumer problem in Java?**([answer](http://java67.blogspot.com/2015/12/producer-consumer-solution-using-blocking-queue-java.html))  
Most of the threading problem you solved in the real world 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, as shown in this [tutorial](http://javarevisited.blogspot.sg/2012/02/producer-consumer-design-pattern-with.html).  
  
  
**27) How do you avoid deadlock in Java? Write Code?**

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?**   
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?**   
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?**   
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?** 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,   
  
**34) What does yield method of Thread class do?**   
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.   
  
**35) What is the concurrency level of ConcurrentHashMap in Java**)  
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,   
  
**36) What is Semaphore in Java?** Semaphore in Java is a new kind of synchronizer. It's a counting semaphore. Conceptually, a semaphore maintains a set of permits. Each acquire() blocks if necessary until a permit is available, and then takes it. Each release() adds a permit, potentially releasing a blocking acquirer. However, no actual permit objects are used; the Semaphore just keeps a count of the number available and acts accordingly. Semaphore is used to protect an expensive resource which is available in fixed number e.g. database connection in the pool.

**37) What happens if you submit a task when the queue of the thread pool is already filled?**   
This is another tricky question on my list. Many programmers will think that it will block until a task is cleared but its true. ThreadPoolExecutor's submit() method throws RejectedExecutionException if the task cannot be scheduled for execution.  
  
**38) What is the difference between the submit() and execute() method thread pool in Java?**   
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.   
  
  
**39) What is blocking method in Java?**   
A blocking method is a method which blocks until the task is done, for example, accept() method of ServerSocket blocks until a client is connected. here blocking means control will not return to the caller until the task is finished. On the other hand, there is an asynchronous or non-blocking method which returns even before the task is finished.   
  
**40) Is Swing thread-safe? What do you mean by Swing thread-safe?** 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?** 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.   
  
**42) Which method of Swing API are thread-safe in Java?**   
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?**   
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.   
  
**44) What is ReadWriteLock in Java?**   
In general, read write lock is the result of lock stripping technique to improve the performance of concurrent applications. In Java, ReadWriteLock is an interface which was added in Java 5 release. A ReadWriteLock maintains a pair of associated locks, one for read-only operations and one for writing. The read lock may be held simultaneously by multiple reader threads, so long as there are no writers. The write lock is exclusive. If you want you can implement this interface with your own set of rules, otherwise you can use ReentrantReadWriteLock, which comes along with JDK and supports a maximum of 65535 recursive write locks and 65535 read locks.  
  
  
**45) What is busy spin in multi-threading?**   
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.  
  
**46) What is the difference between the volatile and atomic variable in Java?**   
This is an interesting question for Java programmer, at first, volatile and atomic variable look very similar, but they are different. Volatile variable provides you 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?** 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,   
  
**49) How to create thread-safe Singleton in Java**  
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  
  
**50) List down 3 multi-threading best practice you follow?**   
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 Exchangersimplifies 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.

**51) How do you force to start a Thread in Java?**   
This question is like how do you force garbage collection in Java, there is no way though you can make a request using System.gc() but it's not guaranteed. On Java multi-threading there is 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?**   
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/joinframework 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.

**53) What is the difference between calling wait() and sleep() method in Java multi-threading**

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

### What is the difference between Process and Thread?

A process is a self contained execution environment and it can be seen as a program or application whereas Thread is a single task of execution within the process. Java runtime environment runs as a single process which contains different classes and programs as processes. Thread can be called lightweight process. Thread requires less resources to create and exists in the process, thread shares the process resources.

### What are the benefits of multi-threaded programming?

In Multi-Threaded programming, multiple threads are executing concurrently that improves the performance because CPU is not idle incase some thread is waiting to get some resources. Multiple threads share the heap memory, so it’s good to create multiple threads to execute some task rather than creating multiple processes. For example, Servlets are better in performance than CGI because Servlet support multi-threading but CGI doesn’t.

### What is difference between user Thread and daemon Thread?

When we create a Thread in java program, it’s known as user thread. A daemon thread runs in background and doesn’t prevent JVM from terminating. When there are no user threads running, JVM shutdown the program and quits. A child thread created from daemon thread is also a daemon thread.

How can we create a Thread in Java?

There are two ways to create Thread in Java – first by implementing Runnable interface and then creating a Thread object from it and second is to extend the Thread Class. Read this post to learn more about [creating threads in java](https://www.journaldev.com/1016/java-thread-example).

### What are different states in lifecycle of Thread?

When we create a Thread in java program, its state is New. Then we start the thread that change it’s state to Runnable. Thread Scheduler is responsible to allocate CPU to threads in Runnable thread pool and change their state to Running. Other Thread states are Waiting, Blocked and Dead. Read this post to learn more about [life cycle of thread](https://www.journaldev.com/1044/thread-life-cycle-in-java-thread-states-in-java).

### Can we call run() method of a Thread class?

### Yes, we can call run() method of a Thread class but then it will behave like a normal method. To actually execute it in a Thread, we need to start it using Thread.start() method.

### How can we pause the execution of a Thread for specific time?

We can use Thread class sleep() method to pause the execution of Thread for certain time. Note that this will not stop the processing of thread for specific time, once the thread awake from sleep, it’s state gets changed to runnable and based on thread scheduling, it gets executed.

### What do you understand about Thread Priority?

Every thread has a priority, usually higher priority thread gets precedence in execution but it depends on Thread Scheduler implementation that is OS dependent. We can specify the priority of thread but it doesn’t guarantee that higher priority thread will get executed before lower priority thread. Thread priority is an int whose value varies from 1 to 10 where 1 is the lowest priority thread and 10 is the highest priority thread.

### What is Thread Scheduler and Time Slicing?

Thread Scheduler is the Operating System service that allocates the CPU time to the available runnable threads. Once we create and start a thread, it’s execution depends on the implementation of Thread Scheduler. Time Slicing is the process to divide the available CPU time to the available runnable threads. Allocation of CPU time to threads can be based on thread priority or the thread waiting for longer time will get more priority in getting CPU time. Thread scheduling can’t be controlled by java, so it’s always better to control it from application itself.

### What is context-switching in multi-threading?

Context Switching is the process of storing and restoring of CPU state so that Thread execution can be resumed from the same point at a later point of time. Context Switching is the essential feature for multitasking operating system and support for multi-threaded environment.

### How can we make sure main() is the last thread to finish in Java Program?

We can use Thread join() method to make sure all the threads created by the program is dead before finishing the main function.

### How does thread communicate with each other?

When threads share resources, communication between Threads is important to coordinate their efforts. Object class wait(), notify() and notifyAll() methods allows threads to communicate about the lock status of a resource.

### Why thread communication methods wait(), notify() and notifyAll() are in Object class?

In Java every Object has a monitor and wait, notify methods are used to wait for the Object monitor or to notify other threads that Object monitor is free now. There is no monitor on threads in java and synchronization can be used with any Object, that’s why it’s part of Object class so that every class in java has these essential methods for inter thread communication.

### Why wait(), notify() and notifyAll() methods have to be called from synchronized method or block?

When a Thread calls wait() on any Object, it must have the monitor on the Object that it will leave and goes in wait state until any other thread call notify() on this Object. Similarly when a thread calls notify() on any Object, it leaves the monitor on the Object and other waiting threads can get the monitor on the Object. Since all these methods require Thread to have the Object monitor, that can be achieved only by synchronization, they need to be called from synchronized method or block.

### Why Thread sleep() and yield() methods are static?

Thread sleep() and yield() methods work on the currently executing thread. So there is no point in invoking these methods on some other threads that are in wait state. That’s why these methods are made static so that when this method is called statically, it works on the current executing thread and avoid confusion to the programmers who might think that they can invoke these methods on some non-running threads.

### How can we achieve thread safety in Java?

There are several ways to achieve thread safety in java – synchronization, atomic concurrent classes, implementing concurrent Lock interface, using volatile keyword, using immutable classes and Thread safe classes.

### What is volatile keyword in Java

When we use volatile keyword with a variable, all the threads read it’s value directly from the memory and don’t cache it. This makes sure that the value read is the same as in the memory.

### Which is more preferred – Synchronized method or Synchronized block?

Synchronized block is more preferred way because it doesn’t lock the Object, synchronized methods lock the Object and if there are multiple synchronization blocks in the class, even though they are not related, it will stop them from execution and put them in wait state to get the lock on Object.

How to create daemon thread in Java?

Thread class setDaemon(true) can be used to create daemon thread in java. We need to call this method before calling start() method else it will throw IllegalThreadStateException.

### What is ThreadLocal?

Java ThreadLocal is used to create thread-local variables. We know that all threads of an Object share it’s variables, so if the variable is not thread safe, we can use synchronization but if we want to avoid synchronization, we can use ThreadLocal variables.  
Every thread has it’s own ThreadLocal variable and they can use it’s get() and set() methods to get the default value or change it’s value local to Thread. ThreadLocal instances are typically private static fields in classes that wish to associate state with a thread.

### What is Thread Group? Why it’s advised not to use it?

ThreadGroup is a class which was intended to provide information about a thread group. ThreadGroup API is weak and it doesn’t have any functionality that is not provided by Thread. Two of the major feature it had are to get the list of active threads in a thread group and to set the uncaught exception handler for the thread. But Java 1.5 has added setUncaughtExceptionHandler(UncaughtExceptionHandler eh) method using which we can add uncaught exception handler to the thread. So ThreadGroup is obsolete and hence not advised to use anymore.

t1.setUncaughtExceptionHandler(new UncaughtExceptionHandler(){

@Override

public void uncaughtException(Thread t, Throwable e) {

System.out.println("exception occured:"+e.getMessage());

}

});

### What is Java Thread Dump, How can we get Java Thread dump of a Program?

Thread dump is list of all the threads active in the JVM, thread dumps are very helpful in analyzing bottlenecks in the application and analyzing deadlock situations. There are many ways using which we can generate Thread dump – Using Profiler, Kill -3 command, jstack tool etc. I prefer jstack tool to generate thread dump of a program because it’s easy to use and comes with JDK installation. Since it’s a terminal based tool, we can create script to generate thread dump at regular intervals to analyze it later on.

### What is Deadlock? How to analyze and avoid deadlock situation?

Deadlock is a programming situation where two or more threads are blocked forever, this situation arises with at least two threads and two or more resources.

To analyze a deadlock, we need to look at the java thread dump of the application, we need to look out for the threads with state as BLOCKED and then the resources it’s waiting to lock, every resource has a unique ID using which we can find which thread is already holding the lock on the object.

Avoid Nested Locks, Lock Only What is Required and Avoid waiting indefinitely are common ways to avoid deadlock situation,

### What is Java Timer Class? How to schedule a task to run after specific interval?

java.util.Timer is a utility class that can be used to schedule a thread to be executed at certain time in future. Java Timer class can be used to schedule a task to be run one-time or to be run at regular intervals.

java.util.TimerTask is an [**abstract class**](https://www.journaldev.com/1582/abstract-class-in-java) that implements Runnable interface and we need to extend this class to create our own TimerTask that can be scheduled using java Timer class.

### 

### What is Thread Pool? How can we create Thread Pool in Java?

A thread pool manages the pool of worker threads, it contains a queue that keeps tasks waiting to get executed.

A thread pool manages the collection of Runnable threads and worker threads execute Runnable from the queue.

java.util.concurrent.Executors provide implementation of java.util.concurrent.Executor interface to create the thread pool in java. [Thread Pool Example](https://www.journaldev.com/1069/threadpoolexecutor-java-thread-pool-example-executorservice) program shows how to create and use Thread Pool in java. Or read [ScheduledThreadPoolExecutor Example](https://www.journaldev.com/2340/java-scheduler-scheduledexecutorservice-scheduledthreadpoolexecutor-example) to know how to schedule tasks after certain delay.

### What will happen if we don’t override Thread class run() method?

Thread class run() method code is as shown below.

public void run() {

if (target != null) {

target.run();

}

}

Above target set in the init() method of Thread class and if we create an instance of Thread class as new TestThread(), it’s set to null. So nothing will happen if we don’t override the run() method. Below is a simple example demonstrating this.

public class TestThread extends Thread {

//not overriding Thread.run() method

//main method, can be in other class too

public static void main(String args[]){

Thread t = new TestThread();

System.out.println("Before starting thread");

t.start();

System.out.println("After starting thread");

}

}

It will print only below output and terminate.

Before starting thread

After starting thread

### What is atomic operation? What are atomic classes in Java Concurrency API?

Atomic operations are performed in a single unit of task without interference from other operations. Atomic operations are necessity in multi-threaded environment to avoid data inconsistency.

int++ is not an atomic operation. So by the time one threads read it’s value and increment it by one, other thread has read the older value leading to wrong result.

To solve this issue, we will have to make sure that increment operation on count is atomic, we can do that using Synchronization but Java 5 java.util.concurrent.atomic provides wrapper classes for int and long that can be used to achieve this atomically without usage of Synchronization.

### What is Lock interface in Java Concurrency API? What are it’s benefits over synchronization?

Lock interface provide more extensive locking operations than can be obtained using synchronized methods and statements. They allow more flexible structuring, may have quite different properties, and may support multiple associated Condition objects.  
The advantages of a lock are

* + it’s possible to make them fair
  + it’s possible to make a thread responsive to interruption while waiting on a Lock object.
  + it’s possible to try to acquire the lock, but return immediately or after a timeout if the lock can’t be acquired
  + it’s possible to acquire and release locks in different scopes, and in different orders

### 

### What is Executors Framework?

In Java 5, Executor framework was introduced with the java.util.concurrent.Executor interface.

The Executor framework is a framework for standardizing invocation, scheduling, execution, and control of asynchronous tasks according to a set of execution policies.

Creating a lot many threads with no bounds to the maximum threshold can cause application to run out of heap memory. So, creating a ThreadPool is a better solution as a finite number of threads can be pooled and reused. Executors framework facilitate process of creating Thread pools in java.

### What is BlockingQueue? How can we implement Producer-Consumer problem using Blocking Queue?

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

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

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

BlockingQueue interface is part of [java collections framework](https://www.journaldev.com/1260/collections-in-java-tutorial) and it’s primarily used for implementing producer consumer problem.

### What is Callable and Future?

Java 5 introduced java.util.concurrent.Callable interface in concurrency package that is similar to Runnable interface but it can return any Object and able to throw Exception.

Callable interface use Generic to define the return type of Object. Executors class provide useful methods to execute Callable in a thread pool. Since callable tasks run in parallel, we have to wait for the returned Object. Callable tasks return java.util.concurrent.Future object. Using Future we can find out the status of the Callable task and get the returned Object. It provides get() method that can wait for the Callable to finish and then return the result.

### What is FutureTask Class?

FutureTask is the base implementation class of Future interface and we can use it with Executors for asynchronous processing. Most of the time we don’t need to use FutureTask class but it comes real handy if we want to override some of the methods of Future interface and want to keep most of the base implementation. We can just extend this class and override the methods according to our requirements.

### What are Concurrent Collection Classes?

Java Collection classes are fail-fast which means that if the Collection will be changed while some thread is traversing over it using iterator, the iterator.next() will throw ConcurrentModificationException.

Concurrent Collection classes support full concurrency of retrievals and adjustable expected concurrency for updates.  
Major classes are ConcurrentHashMap, CopyOnWriteArrayList and CopyOnWriteArraySet

### What is Executors Class?

Executors class provide utility methods for Executor, ExecutorService, ScheduledExecutorService, ThreadFactory, and Callable classes.

Executors class can be used to easily create Thread Pool in java, also this is the only class supporting execution of Callable implementations.

### What are some of the improvements in Concurrency API in Java 8?

Some important concurrent API enhancements are:

* + ConcurrentHashMap compute(), forEach(), forEachEntry(), forEachKey(), forEachValue(), merge(), reduce() and search() methods.
  + CompletableFuture that may be explicitly completed (setting its value and status).
  + Executors newWorkStealingPool() method to create a work-stealing thread pool using all available processors as its target parallelism level.

**Question 1. What is Thread in java?**.

* Threads consumes CPU in best possible manner, hence enables multi processing. Multi threading reduces idle time of CPU which improves performance of application.
* Thread are light weight process.
* A thread class belongs to java.lang package.
* We can create multiple threads in java, even if we don’t create any Thread, one Thread at least  do exist i.e. main thread.
* Multiple threads run parallely in java.
* Threads have their own stack.
* Advantage of Thread : Suppose one thread needs 10 minutes to get certain task, 10 threads used at a time could complete that task in 1 minute, because threads can run parallely.

**Question 2. What is difference between Process and Thread in java?**

Answer.  One process can have multiple Threads,

Thread are subdivision of Process. One or more Threads runs in the context of process. Threads can execute any part of process. And same part of process can be executed by multiple Threads.

Processes have their own copy of the data segment of the parent process while Threads have direct access to the data segment of its process.

Processes have their own address while Threads share the address space of the process that created it.

Process creation needs whole lot of stuff to be done, we might need to copy whole parent process, but Thread can be easily created.

Processes can easily communicate with child processes but interprocess communication is difficult. While, Threads can easily communicate with other threads of the same process using [wait() and notify() methods](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html).

In process all threads share system resource like heap Memory etc. while Thread has its own stack.

Any change made to process does not affect child processes, but any change made to thread can affect the behavior of the other threads of the process.

[Example to see where threads on are created on different processes and same process.](http://www.javamadesoeasy.com/2015/03/when-threads-are-not-lightweight.html)

**Question 3. How to implement Threads in java?**

Answer.  This is very basic threading question. Threads can be created in two ways i.e. by [implementing java.lang.Runnable interface or extending java.lang.Thread class](http://www.javamadesoeasy.com/2015/03/implementing-threads-in-java-by.html) and then extending run method.

Thread has its own variables and methods, it lives and dies on the heap. [But a thread of execution is an individual process that has its own call stack](http://www.javamadesoeasy.com/2015/03/threads-implement-their-own-stack.html). Thread are lightweight process in java.

1. Thread creation by  implementingjava.lang.Runnableinterface.

We will create object of class which implements Runnable interface :

MyRunnable runnable=new MyRunnable();

Thread thread=new Thread(runnable);

 2) And then create Thread object by calling constructor and passing reference of Runnable interface i.e.  runnable object :

Thread thread=new Thread(runnable);

**Question 4 . Does Thread implements their own Stack, if yes how? (Important)**

Answer.  Yes, [Threads have their own stack](http://www.javamadesoeasy.com/2015/03/threads-implement-their-own-stack.html). This is very interesting question, where interviewer tends to check your basic knowledge about how [threads internally maintains their own stacks](http://www.javamadesoeasy.com/2015/03/threads-implement-their-own-stack.html). I’ll be explaining you the concept by diagram.

**Question 5. We should implement Runnable interface or extend Thread class. What are differences between implementing Runnable and extending Thread?**

Answer. Well the answer is you must [extend Thread](http://www.javamadesoeasy.com/2015/03/implementing-threads-in-java-by.html) only when you are looking to modify run() and other methods as well. If you are simply looking to modify only the run() method [implementing Runnable](http://www.javamadesoeasy.com/2015/03/implementing-threads-in-java-by.html) is the best option (Runnable interface has only one abstract method i.e. run() ).

[Differences between implementing Runnable interface and extending Thread class](http://www.javamadesoeasy.com/2015/03/differences-between-implementing.html) -

1. Multiple inheritance in not allowed in java : When we [implement Runnable](http://www.javamadesoeasy.com/2015/03/implementing-threads-in-java-by.html) interface we can extend another class as well, but if we extend Thread class we cannot extend any other class because java does not allow multiple inheritance. So, same work is done by implementing Runnable and [extending Thread](http://www.javamadesoeasy.com/2015/03/implementing-threads-in-java-by.html) but in case of implementing Runnable we are still left with option of extending some other class. So, it’s better to implement Runnable.
2. [Thread safety](http://www.javamadesoeasy.com/2015/03/guidelines-to-thread-safe-code-most.html) : When we implement Runnable interface, same object is shared amongst multiple threads, but when we extend Thread class each and every thread gets associated with new object.
3. Inheritance (Implementing Runnable is lightweight operation) : When we extend Thread unnecessary all Thread class features are inherited, but when we implement Runnable interface no extra feature are inherited, as Runnable only consists only of one abstract method i.e. run() method. So, implementing Runnable is lightweight operation.
4. Coding to interface : Even java recommends coding to interface. So, we must implement Runnable rather than extending thread. Also, Thread class implements Runnable interface.
5. Don’t extend unless you wanna modify fundamental behaviour of class, Runnable interface has only one abstract method i.e. run()  : We must [extend Thread](http://www.javamadesoeasy.com/2015/03/implementing-threads-in-java-by.html) only when you are looking to modify run() and other methods as well. If you are simply looking to modify only the run() method [implementing Runnable](http://www.javamadesoeasy.com/2015/03/implementing-threads-in-java-by.html) is the best option (Runnable interface has only one abstract method i.e. run() ). We must not extend Thread class unless we're looking to modify fundamental behaviour of Thread class.
6. Flexibility in code when we implement Runnable : When we extend Thread first a fall all thread features are inherited and our class becomes direct subclass of Thread , so whatever action we are doing is in Thread class. But, when we implement Runnable we create a new thread and pass runnable object as parameter,we could pass runnable object to executorService & much more. So, we have more options when we implement Runnable and our code becomes more flexible.
7. ExecutorService : If we implement Runnable, we can start multiple thread created on runnable object  with ExecutorService (because we can start Runnable object with new threads), but not in the case when we extend Thread (because thread can be started only once).

**Question 6. How can you say Thread behaviour is unpredictable? (Important)**

Answer. The solution to question is quite simple, [Thread behaviour is unpredictable](http://www.javamadesoeasy.com/2015/03/thread-behaviour-is-unpredictable.html) because execution of Threads depends on Thread scheduler, thread scheduler may have different implementation on different platforms like windows, unix etc. Same threading program may produce different output in subsequent executions even on same platform.

To achieve we are going to create 2 threads on same Runnable Object, create for loop in run() method and start  both threads. There is no surety that which threads will complete first,  both threads will enter anonymously in for loop.

**Question 7 . When threads are not lightweight process in java?**

Answer. Threads are [lightweight process](http://www.javamadesoeasy.com/2015/03/when-threads-are-not-lightweight.html) only if threads of same process are executing concurrently. But if threads of different processes are executing concurrently then threads are [heavy weight process](http://www.javamadesoeasy.com/2015/03/when-threads-are-not-lightweight.html).

**Question 8. How can you ensure all threads that started from main must end in order in which they started and also main should end in last? (Important)**

Answer.  Interviewers tend to know interviewees knowledge about Thread methods. So this is time to prove your point by answering correctly. We can use [join() method](http://www.javamadesoeasy.com/2015/03/join-method-ensure-all-threads-that.html)to ensure all threads that started from main must end in order in which they started and also main should end in last.In other words waits for this thread to die. Calling join() method internally calls join(0);

[DETAILED DESCRIPTION : Join() method - ensure all threads that started from main must end in order in which they started and also main should end in last. Types of join() method with programs- 10 salient features of join.](http://www.javamadesoeasy.com/2015/03/join-method-ensure-all-threads-that.html)

**Question 9.What is difference between starting thread with run() and start() method? (Important)**

Answer. This is quite interesting question, it might confuse you a bit and at time may make you think is there really any [difference between starting thread with run() and start() method](http://www.javamadesoeasy.com/2015/03/difference-between-starting-thread-with.html).

When you call start() method, main thread internally calls run() method to start newly created Thread, so run() method is ultimately called by newly created thread.

When you call run() method main thread rather than starting run() method with newly thread it start run() method by itself.

**Question 10. What is significance of using**[**Volatile**](http://www.javamadesoeasy.com/2015/03/volatile-keyword-in-java-difference.html)**keyword? (Important)**

Answer. Java allows threads to access shared variables. As a rule, to ensure that shared variables are consistently updated, a thread should ensure that it has exclusive use of such variables by obtaining a lock that enforces mutual exclusion for those shared variables.

If a field is declared [volatile](http://www.javamadesoeasy.com/2015/03/volatile-keyword-in-java-difference.html), in that case the Java memory model ensures that all threads see a consistent value for the variable.

Few small questions>

Q. Can we have [volatile](http://www.javamadesoeasy.com/2015/03/volatile-keyword-in-java-difference.html) methods in java?

1. No, volatile is only a keyword, can be used only with variables.

Q. Can we have synchronized variable in java?

1. No, synchronized can be used only with methods, i.e. in method declaration.

**Question 11. Differences between synchronized and volatile keyword in Java? (Important)**

Answer.Its very important question from interview perspective.

1. [Volatile](http://www.javamadesoeasy.com/2015/03/volatile-keyword-in-java-difference.html)can be used as a keyword against the variable, we cannot use volatile against method declaration.

volatile void method1(){} //it’s illegal, compilation error.

While [synchronization](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html) can be used in method declaration or we can create synchronization blocks (In both cases thread acquires lock on object’s monitor). Variables cannot be synchronized.

Synchronized method:

synchronized void method2(){} //legal

Synchronized block:

void method2(){

synchronized (this) { //code inside synchronized block. }}

Synchronized variable (illegal):

synchronized int i;//it’s illegal, compilatiomn error.

1. [Volatile](http://www.javamadesoeasy.com/2015/03/volatile-keyword-in-java-difference.html) does not acquire any lock on variable or object, but [Synchronization](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html) acquires lock on method or block in which it is used.
2. [Volatile](http://www.javamadesoeasy.com/2015/03/volatile-keyword-in-java-difference.html) variables are not cached, but variables used inside [synchronized](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html) method or block are cached.
3. When volatile is used will never create deadlock in program, as volatile never obtains any kind of lock . But in case if synchronization is not done properly, we might end up creating dedlock in program.
4. Synchronization may cost us performance issues, as one thread might be waiting for another thread to release lock on object. But volatile is never expensive in terms of performance.

### DETAILED DESCRIPTION : [Differences between synchronized and volatile keyword in detail with programs.](http://www.javamadesoeasy.com/2015/03/differences-between-synchronized-and.html)

**Question 12. Can you again start Thread?**

Answer.No, [we cannot start Thread again](http://www.javamadesoeasy.com/2015/03/can-we-start-thread-again.html), doing so will throw runtimeException java.lang.IllegalThreadStateException. The reason is once run() method is executed by Thread, it goes into [dead state](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html).

Let’s take an example-

Thinking of starting thread again and calling start() method on it (which internally is going to call run() method) for us is some what like asking dead man to wake up and run. As, after completing his life person goes to dead state.

**Question 13. What is race condition in multithreading and how can we solve it? (Important)**

Answer. This is very important question, this forms the core of multi threading, you should be able to explain about [race condition in detail](http://www.javamadesoeasy.com/2015/03/race-condition-in-multithreading-and.html). When more than one thread try to access same resource without synchronization causes race condition.

So we can [solve race condition](http://www.javamadesoeasy.com/2015/03/race-condition-in-multithreading-and.html) by using either [synchronized block or synchronized method](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html). When no two threads can access same resource at a time phenomenon is also called as mutual exclusion.

Few sub questions>

What if two threads try to read same resource without [synchronization](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html)?

When two threads try to read on same resource without synchronization, it’s never going to create any problem.

What if two threads try to write to same resource without [synchronization](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html)?

When two threads try to write to same resource without synchronization, it’s going to create synchronization problems.

**Question 14.** How threads communicate between each other?

Answer. This is very must know question for all the interviewees, you will most probably face this question in almost every time you go for interview.

Threads can communicate with each other by using [wait(), notify() and notifyAll()](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) methods.

**Question 15. Why wait(), notify()  and notifyAll() are in Object class and not in Thread class? (Important)**

Answer.

1. Every Object has a monitor, acquiring that monitors allow thread to hold lock on object. But Thread class does not have any monitors.
2. wait(), notify() and notifyAll()are called on objects only >When wait() method is called on object by thread it waits for another thread on that object to release object monitor by calling [notify() or notifyAll()](http://www.javamadesoeasy.com/2015/03/difference-between-notify-and-notifyall.html) method on that object.

When notify() method is called on object by thread it notifies all the threads

which are waiting for that object monitor that object monitor is available now.

So, this shows that wait(), notify() and notifyAll() are called on objects only.

[Now, Straight forward question that comes to mind is how thread acquires object lock by](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html)

[acquiring object monitor? Let’s try to understand this basic concept in detail?](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html)

1. Wait(), notify() and notifyAll() method being in Object class allows all the threads created on that object to communicate with other.  .
2. As multiple threads exists on same object. Only one thread can hold object monitor at a time. As a result thread can notify other threads of same object that lock is available now. But, thread having these methods does not make any sense because multiple threads exists on object its not other way around (i.e. multiple objects exists on thread).
3. Now let’s discuss one hypothetical scenario, what will happen if Thread class contains wait(), notify() and notifyAll() methods?

Having wait(), notify() and notifyAll() methods means Thread class also must have their monitor.

Every thread having their monitor will create few problems -

>Thread communication problem.

>Synchronization on object won’t be possible- Because object has monitor, one object can have multiple threads and thread hold lock on object by holding object monitor. But if each thread will have monitor, we won’t have any way of achieving synchronization.

>Inconsistency in state of object (because synchronization won't be possible).

**Question 16. Is it important to acquire object lock before calling wait(), notify() and notifyAll()?**

Answer.Yes, it’s mandatory to acquire object lock before calling these methods on object. As discussed above wait(), notify()  and notifyAll() methods are always called from [Synchronized block](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html) only, and as soon as thread enters synchronized block it acquires object lock (by holding object monitor). If we call these methods without acquiring object lock i.e. from outside synchronize block then java.lang. IllegalMonitorStateException is thrown at runtime.

Wait() method needs to enclosed in try-catch block, because it throws compile time exception i.e. InterruptedException.

**Question 17. How can you solve consumer producer problem by using wait() and notify() method? (Important)**

Answer.  Here come the time to answer very very important question from interview perspective. Interviewers tends to check how sound you are in threads inter communication. Because for solving this problem we got to use synchronization blocks, wait() and notify() method very cautiously. If you misplace synchronization block or any of the method, that may cause your program to go horribly wrong. So, before going into this question first i’ll recommend you to understand how to use synchronized blocks, wait() and notify() methods.

Key points we need to ensure before programming :

>Producer will produce total of 10 products and cannot produce more than 2 products at a time until products are being consumed by consumer.

Example> when sharedQueue’s size is 2, wait for consumer to consume (consumer will consume by calling remove(0) method on sharedQueue and reduce sharedQueue’s size). As soon as size is less than 2, producer will start producing.

>Consumer can consume only when there are some products to consume.

Example> when sharedQueue’s size is 0, wait for producer to produce (producer will produce by calling add() method on sharedQueue and increase sharedQueue’s size). As soon as size is greater than 0, consumer will start consuming.

Explanation of Logic >

We will create sharedQueue that will be shared amongst Producer and Consumer. We will now start consumer and producer thread.

Note: it does not matter order in which threads are started (because rest of code has taken care of synchronization and key points mentioned above)

First we will start consumerThread >

consumerThread.start();

consumerThread will enter run method and call consume() method. There it will check for sharedQueue’s size.

-if size is equal to 0 that means producer hasn’t produced any product, wait for producer to produce by using below piece of code-

synchronized (sharedQueue) {

while (sharedQueue.size() == 0) {

sharedQueue.wait();

} }

-if size is greater than 0, consumer will start consuming by using below piece of code.

synchronized (sharedQueue) {

Thread.sleep((long)(Math.random() \* 2000));

System.out.println("consumed : "+ sharedQueue.remove(0));

sharedQueue.notify();

}

Than we will start producerThread >

|  |
| --- |
| producerThread.start(); |

producerThread will enter run method and call produce() method. There it will check for sharedQueue’s size.

-if size is equal to 2 (i.e. maximum number of products which sharedQueue can hold at a time), wait for consumer to consume by using below piece of code-

synchronized (sharedQueue) {

while (sharedQueue.size() == maxSize) { //maxsize is 2

sharedQueue.wait();

} }

-if size is less than 2, producer will start producing by using below piece of code.

synchronized (sharedQueue) {

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

sharedQueue.add(i);

Thread.sleep((long)(Math.random() \* 1000));

sharedQueue.notify();

}

**Question 18.**[**How to solve Consumer Producer problem without using wait() and notify() methods, where consumer can consume only when production is over.**](http://www.javamadesoeasy.com/2015/03/how-to-solve-consumer-producer-problem.html)**?**

Answer. In this problem, producer will allow consumer to consume only when 10 products have been produced (i.e. when production is over).

We will approach by keeping one boolean variable productionInProcess and initially setting it to true, and later when production will be over we will set it to false.

**Question 19. How can you solve consumer producer pattern by using BlockingQueue? (Important)**

Answer. Now it’s time to gear up to face question which is most probably going to be followed up by previous question i.e. after how to solve consumer producer problem using wait() and notify() method. Generally you might wonder why interviewer's are so much interested in asking about [solving consumer producer problem using BlockingQueue](http://www.javamadesoeasy.com/2015/03/solve-consumer-producer-problem-by.html), answer is they want to know how strong knowledge you have about java concurrent Api’s, this Api use consumer producer pattern in very optimized manner, BlockingQueue is designed is such a manner that it offer us the best performance.

[BlockingQueue is a interface and we will use its implementation class LinkedBlockingQueue.](http://www.javamadesoeasy.com/2015/03/solve-consumer-producer-problem-by.html)

Key methods for solving consumer producer pattern are >

put(i); //used by producer to put/produce in sharedQueue.

take();//used by consumer to take/consume from sharedQueue.

**Question 20. What is**[**deadlock**](http://www.javamadesoeasy.com/2015/03/deadlock-in-multithreading-program-to.html)**in multithreading? Write a program to form [DeadLock](http://www.javamadesoeasy.com/2015/03/deadlock-in-multithreading-program-to.html) in multi threading and also how to solve DeadLock situation. What measures you should take to avoid deadlock? (Important)**

Answer.  This is very important question from interview perspective. But, what makes this question important is it checks interviewees capability of [creating and detecting deadlock](http://www.javamadesoeasy.com/2015/03/deadlock-in-multithreading-program-to.html). If you can write a code to form deadlock, than I am sure you must be well capable in solving that deadlock as well. If not, later on this post we will learn how to solve deadlock as well.

First question comes to mind is, [what is deadlock in multi threading program](http://www.javamadesoeasy.com/2015/03/deadlock-in-multithreading-program-to.html)?

Deadlock is a situation where two threads are waiting for each other to release lock holded by them on resources.

But how [deadlock](http://www.javamadesoeasy.com/2015/03/deadlock-in-multithreading-program-to.html) could be formed :

Thread-1 acquires lock on String.class and then calls [sleep()](http://www.javamadesoeasy.com/2015/03/sleep-method-in-threads-10-key-features.html) method which gives Thread-2 the chance to execute immediately after Thread-1 has acquired lock on String.class and Thread-2 acquires lock on Object.class then calls sleep() method and now it waits for Thread-1 to release lock on String.class.

Conclusion:

Now, Thread-1 is waiting for Thread-2 to release lock on Object.class and Thread-2 is waiting for Thread-1 to release lock on String.class and deadlock is formed.

//Code called by Thread-1

public void run() {

synchronized (String.class) {

Thread.sleep(100);

synchronized (Object.class) { } }}

//Code called by Thread-2

publicvoid run() {

synchronized (Object.class) {

Thread.sleep(100); synchronized (String.class) { } }}

Here comes the important part, how above formed deadlock could be solved :

Thread-1 acquires lock on String.class and then calls [sleep()](http://www.javamadesoeasy.com/2015/03/sleep-method-in-threads-10-key-features.html) method which gives Thread-2 the chance to execute immediately after Thread-1 has acquired lock on String.class and Thread-2 tries to acquire lock on String.class but lock is holded by Thread-1. Meanwhile, Thread-1 completes successfully. As Thread-1 has completed successfully it releases lock on String.class, Thread-2 can now acquire lock on String.class and complete successfully without any deadlock formation.

Conclusion: No deadlock is formed.

//Code called by Thread-1

publicvoid run() {

synchronized (String.class) {

Thread.sleep(100);

synchronized (Object.class) {

}}}

//Code called by Thread-2

publicvoid run() {

synchronized (String.class) {

Thread.sleep(100);

synchronized (Object.class) {

} }}

Few important measures to avoid [Deadlock](http://www.javamadesoeasy.com/2015/03/deadlock-in-multithreading-program-to.html) >

1. Lock specific member variables of class rather than locking whole class: We must try to lock specific member variables of class rather than locking whole class.
2. Use join() method: If possible try touse join() method, although it may refrain us from taking full advantage of multithreading environment because threads will start and end sequentially, but it can be handy in avoiding deadlocks.
3. If possible try avoid using nested synchronization blocks.

**Question 21. Have you ever generated thread dumps or analyzed Thread Dumps? (Important)**

Answer. Answering this questions will show your in depth knowledge of Threads. Every experienced must know how to generate Thread Dumps.

[VisualVM](http://www.javamadesoeasy.com/2015/03/visualvm-thread-dumps-generating-and_74.html)  is most popular way to generate Thread Dump and is most widely used by developers. It’s important to understand usage of VisualVM for in depth knowledge of VisualVM. I’ll recommend every developer must understand this topic to become master in multi threading.

It helps us in analyzing threads performance, [thread states](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html), CPU consumed by threads, garbage collection and much more.  For detailed information see [Generating and analyzing Thread Dumps using VisualVM - step by step detail to setup VisualVM with screenshots](http://www.javamadesoeasy.com/2015/03/visualvm-thread-dumps-generating-and_74.html)

[jstack](http://www.javamadesoeasy.com/2015/03/jstack-thread-dumps-generating-and.html) is very easy way to generate Thread dump and is widely used by developers. I’ll recommend every developer must understand this topic to become master in multi threading. For creating Thread dumps we need not to download any jar or any extra software. For detailed information see [Generating and analyzing Thread Dumps using JSATCK - step by step detail to setup JSTACK with screenshots](http://www.javamadesoeasy.com/2015/03/jstack-thread-dumps-generating-and.html).

**Question 22. What is life cycle of Thread, explain thread states? (Important)**

Answer.  [Thread states/ Thread life cycle](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html) is very basic question, before going deep into concepts we must understand Thread life cycle.

Thread have following states >

* New
* Runnable
* Running
* Waiting/blocked/sleeping
* Terminated (Dead)

Thread states/ Thread life cycle in diagram >

Thread states in detail >

New : When instance of thread is created using new operator it is in new state, but the start() method has not been invoked on the thread yet, thread is not eligible to run yet.

Runnable : When start() method is called on thread it enters runnable state.

Running : Thread scheduler selects thread to go fromrunnable to running state. In running state Thread starts executing by entering run() method.

Waiting/blocked/sleeping : In this state a thread is not eligible to run.

>Thread is still alive, but currently it’s not eligible to run. In other words.

> How can Thread go from running to waiting state?

 By calling wait()[method](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) thread go from running to waiting state. In waiting state it will wait for other threads to release object monitor/lock.

> How can Thread go from running to sleeping state?

 By calling sleep() [method](http://www.javamadesoeasy.com/2015/03/sleep-method-in-threads-10-key-features.html)thread go from running to sleeping state. In sleeping state it will wait for sleep time to get over.

Terminated (Dead) : A thread is considered dead when its run() method completes.

**Question 23. Are you aware of preemptive scheduling and time slicing?**

Answer. In preemptive scheduling, the highest priority thread executes until it enters into the [waiting or dead state](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html).

In time slicing, a thread executes for a certain predefined time and then enters runnable pool. Than thread can enter running state when selected by thread scheduler.

**Question 24. What are**[**daemon threads**](http://www.javamadesoeasy.com/2015/03/daemon-threads-12-salient-features-of.html)**?**

Answer.[Daemon threads](http://www.javamadesoeasy.com/2015/03/daemon-threads-12-salient-features-of.html) are low priority threads which runs intermittently in background for doing garbage collection.

 12 Few salient features of [daemon() threads](http://www.javamadesoeasy.com/2015/03/daemon-threads-12-salient-features-of.html)>

* Thread scheduler schedules these threads only when CPU is idle.
* [Daemon threads](http://www.javamadesoeasy.com/2015/03/daemon-threads-12-salient-features-of.html) are service oriented threads, they serves all other threads.
* These threads are created before user threads are created and die after all other user threads dies.
* Priority of daemon threads is always 1 (i.e. MIN\_PRIORITY).
* User created threads are non daemon threads.
* JVM can exit when only daemon threads exist in system.
* we can use isDaemon() method to check whether thread is daemon thread or not.
* we can use setDaemon(boolean on) method to make any user method a daemon thread.
* If setDaemon(boolean on) is called on thread after calling start() method than IllegalThreadStateException is thrown.
* You may like to see how daemon threads work, for that you can use VisualVM or jStack. I have provided Thread dumps over there which shows daemon threads which were intermittently running in background.

Some of the daemon threads which intermittently run in background are >

|  |
| --- |
| "RMI TCP Connection(3)-10.175.2.71" daemon"RMI TCP Connection(idle)" daemon"RMI Scheduler(0)" daemon"C2 CompilerThread1" daemon"GC task thread#0 (ParallelGC)" |

**Question 25. Why**[**suspend() and resume() methods are deprecated**](http://www.javamadesoeasy.com/2015/03/reason-why-suspend-and-resume-methods.html)**?**

**A**nswer.[Suspend()](http://www.javamadesoeasy.com/2015/03/using-suspend-and-resume-method-in.html) method is [deadlock](http://www.javamadesoeasy.com/2015/03/deadlock-in-multithreading-program-to.html) prone. If the target thread holds a lock on object when it is suspended, no thread can lock this object until the target thread is [resumed](http://www.javamadesoeasy.com/2015/03/using-suspend-and-resume-method-in.html). [If the thread that would resume the target thread attempts to lock this monitor prior to calling resume, it results in deadlock formation](http://www.javamadesoeasy.com/2015/03/reason-why-suspend-and-resume-methods.html).

These [deadlocks](http://www.javamadesoeasy.com/2015/03/deadlock-in-multithreading-program-to.html)are generally called Frozen processes.

Suspend() method puts thread from [running to waiting state](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html). And thread can go from waiting to runnable state only when resume() method is called on thread. It is deprecated method.

Resume() method is only used with suspend() method that’s why it’s also deprecated method.

**Question 26. Why destroy() methods is deprecated?**

Answer. This question is again going to check your in depth knowledge of thread methods i.e. [destroy() method](http://www.javamadesoeasy.com/2015/03/destroy-method-in-java-usage-reason-why.html) is [deadlock](http://www.javamadesoeasy.com/2015/03/deadlock-in-multithreading-program-to.html) prone. If the target thread holds a lock on object when it is destroyed, no thread can lock this object (Deadlock formed are similar to deadlock formed when suspend() and resume() methods are used improperly). It results in deadlock formation. These [deadlocks](http://www.javamadesoeasy.com/2015/03/deadlock-in-multithreading-program-to.html)are generally called Frozen processes.

Additionally you must know calling destroy() method on Threads throw runtimeException i.e. NoSuchMethodError. [Destroy() method](http://www.javamadesoeasy.com/2015/03/destroy-method-in-java-usage-reason-why.html) puts thread from running to [dead state](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html).

**Question 27. As stop() method is deprecated,  How can we terminate or stop infinitely running thread in java? (Important)**

Answer. This is very interesting question where interviewees thread basics basic will be tested. Interviewers tend to know user’s knowledge about main thread’s and thread invoked by main thread.

We will try to address the problem by creating new thread which will run infinitely until certain condition is satisfied and will be called by main Thread.

1. Infinitely running thread can be stopped using boolean variable.
2. [Infinitely running thread can be stopped using interrupt() method](http://www.javamadesoeasy.com/2015/03/2-alternate-ways-to-stop-thread-as-stop.html).

Let’s understand Why stop() method is deprecated :

Stopping a thread with Thread.stop() causes it to release all of the monitors that it has locked. If any of the objects previously protected by these monitors were in an inconsistent state, the damaged objects become visible to other threads, which might lead to unpredictable behavior.

**Question 28. what is significance of yield() method, what state does it put thread in?**

[yield()](http://www.javamadesoeasy.com/2015/03/yield-method-in-threads-8-key-features.html) is a native method it’s implementation in java 6 has been changed as compared to its implementation java 5. As method is native it’s implementation is provided by JVM.

In java 5, yield() method internally used to call [sleep()](http://www.javamadesoeasy.com/2015/03/sleep-method-in-threads-10-key-features.html) method giving all the other threads of same or higher priority to execute before yielded thread by leaving allocated CPU for time gap of 15 millisec.

But java 6, calling yield() method gives a hint to the thread scheduler that the current thread is willing to yield its current use of a processor. The thread scheduler is free to ignore this hint. So, sometimes even after using yield() method, you may not notice any difference in output.

salient features of [yield()](http://www.javamadesoeasy.com/2015/03/yield-method-in-threads-8-key-features.html) method >

* Definition : [yield()](http://www.javamadesoeasy.com/2015/03/yield-method-in-threads-8-key-features.html) method when called on thread gives a hint to the thread scheduler that the current thread is willing to yield its current use of a processor.The thread scheduler is free to ignore this hint.
* [Thread state](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html) : when yield() method is called on thread it goes from running to runnable state, not in waiting state. Thread is eligible to run but not running and could be picked by scheduler at anytime.
* Waiting time : yield() method stops thread for unpredictable time.
* Static method : yield()is a static method, hence calling Thread.yield() causes currently executing thread to yield.
* Native method : implementation of yield() method is provided by JVM.

Let’s see definition of yield() method as given in java.lang.Thread -

|  |
| --- |
| public static native void yield(); |

* [synchronized block](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html) : thread need not to to acquire object lock before calling yield()method i.e. yield() method can be called from outside synchronized block.

**Question 29.What is significance of sleep() method in detail, what [state](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html)does it put thread in ?**

[sleep()](http://www.javamadesoeasy.com/2015/03/sleep-method-in-threads-10-key-features.html) is a native method, it’s implementation is provided by JVM.

10 salient features of [sleep()](http://www.javamadesoeasy.com/2015/03/sleep-method-in-threads-10-key-features.html) method >

* Definition : [sleep()](http://www.javamadesoeasy.com/2015/03/sleep-method-in-threads-10-key-features.html) methods causes current thread to sleep for specified number of milliseconds (i.e. time passed in sleep method as parameter). Ex- Thread.sleep(10) causes currently executing thread to sleep for 10 millisec.
* [Thread state](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html) : when sleep() is called on thread it goes from running to waiting state and can return to runnable state when sleep time is up.
* Exception : sleep() method must catch or throw compile time exception i.e. InterruptedException.
* Waiting time : sleep() method have got few options.
  1. sleep(long millis) - Causes the currently executing thread to sleep for the specified number of milliseconds

|  |
| --- |
| public static native void sleep(long millis) throws InterruptedException; |

1. sleep(long millis, int nanos) - Causes the currently executing thread to sleep for the specified number of milliseconds plus the specified number of nanoseconds.

|  |
| --- |
| public static native void sleep(long millis,int nanos) throws InterruptedException; |

* static method : sleep()is a static method, causes the currently executing thread to sleep for the specified number of milliseconds.
* Belongs to which class :[sleep()](http://www.javamadesoeasy.com/2015/03/sleep-method-in-threads-10-key-features.html) method belongs to java.lang.Thread class.
* synchronized block : thread need not to to acquire object lock before calling sleep()method i.e. sleep() method can be called from outside synchronized block.

**Question 30. Difference between**[**wait()**](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html)**and**[**sleep()**](http://www.javamadesoeasy.com/2015/03/sleep-method-in-threads-10-key-features.html)**? (Important)**

* Should be called from [synchronized block](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html) :wait() method is always called from synchronized block i.e. [wait()](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) method needs to lock object monitor before object on which it is called.  But sleep() method can be called from outside synchronized block i.e. sleep() method doesn’t need any object monitor.
* IllegalMonitorStateException : if wait() method is called without acquiring object lock than IllegalMonitorStateException is thrown at runtime, but sleep() methodnever throws such exception.
* Belongs to which class : wait() method belongs to java.lang.Object class but sleep() method belongs to java.lang.Thread class.
* Called on object or thread : wait() method is called on objects but sleep() method is called on Threads not objects.
* [Thread state](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html) : when wait() method is called on object, thread that holded object’s monitor goes from running to waiting state and can return to runnable state only when notify() or notifyAll()method is called on that object. And later thread scheduler schedules that thread to go from from runnable to running state.

when sleep() is called on thread it goes from running to waiting state and can return to runnable state when sleep time is up.

* When called from [synchronized block](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html) :when wait() method is called thread leaves the object lock.  But sleep()method when called from synchronized block or method thread doesn’t leaves object lock.

**Question 31. Differences and similarities between yield() and sleep()?**

Differences [yield()](http://www.javamadesoeasy.com/2015/03/yield-method-in-threads-8-key-features.html) and [sleep()](http://www.javamadesoeasy.com/2015/03/sleep-method-in-threads-10-key-features.html) :

* Definition : yield() method when called on thread gives a hint to the thread scheduler that the current thread is willing to yield its current use of a processor.The thread scheduler is free to ignore this hint. sleep() methods causes current thread to sleep for specified number of milliseconds (i.e. time passed in sleep method as parameter). Ex- Thread.sleep(10) causes currently executing thread to sleep for 10 millisec.
* [Thread state](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html) : when sleep() is called on thread it goes from running to waiting state and can return to runnable state when sleep time is up. when yield() method is called on thread it goes from running to runnable state, not in waiting state. Thread is eligible to run but not running and could be picked by scheduler at anytime.
* Exception : yield() method need not to catch or throw any exception. But sleep() method must catch or throw compile time exception i.e. InterruptedException.
* Waiting time : yield() method stops thread for unpredictable time, that depends on thread scheduler. But sleep() method have got few options.
  1. sleep(long millis) - Causes the currently executing thread to sleep for the specified number of milliseconds
  2. sleep(long millis, int nanos) - Causes the currently executing thread to sleep for the specified number of milliseconds plus the specified number of nanoseconds.

similarity between [yield()](http://www.javamadesoeasy.com/2015/03/yield-method-in-threads-8-key-features.html) and [sleep()](http://www.javamadesoeasy.com/2015/03/sleep-method-in-threads-10-key-features.html):

> yield() and sleep() method belongs to java.lang.Thread class.

> yield() and sleep() method can be called from outside synchronized block.

> yield() and sleep() method are called on Threads not objects.

**Question 32. Mention some g**[**uidelines to write thread safe code, most important point we must take care of in multithreading programs**](http://www.javamadesoeasy.com/2015/03/guidelines-to-thread-safe-code-most.html)**?**

Answer.  In multithreading environment it’s important very important to [write thread safe code](http://www.javamadesoeasy.com/2015/03/guidelines-to-thread-safe-code-most.html), thread unsafe code can cause a major threat to your application. I have posted many articles regarding thread safety. So overall this will be revision of what we have learned so far i.e. writing thread safe healthy code and avoiding any kind of [deadlocks](http://www.javamadesoeasy.com/2015/03/deadlock-in-multithreading-program-to.html).

1. If method is exposed in multithreading environment and it’s not synchronized (thread unsafe) than it might lead us to [race condition](http://www.javamadesoeasy.com/2015/03/race-condition-in-multithreading-and.html), we must try to use [synchronized block and synchronized methods](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html). [Multiple threads may exist on same object](http://www.javamadesoeasy.com/2015/03/suppose-you-have-2-threads-thread-1-and_5.html) but only one thread of that object can enter synchronized method at a time, though  [threads on different object](http://www.javamadesoeasy.com/2015/03/suppose-you-have-2-threads-thread-1-on.html) can enter same method at same time.
2. Even static variables are not thread safe, they are used in static methods and if static methods are not synchronized then thread on same or different object can enter method concurrently. Multiple threads may exist on [same](http://www.javamadesoeasy.com/2015/03/suppose-you-have-2-threads-thread-1-and_46.html) or [different objects](http://www.javamadesoeasy.com/2015/03/suppose-you-have-2-threads-thread-1-on_5.html) of class but only one thread can enter [static synchronized method](http://www.javamadesoeasy.com/2015/03/acquiring-lock-on-class-2-ways-to.html) at a time, we must consider making [static methods as synchronized](http://www.javamadesoeasy.com/2015/03/acquiring-lock-on-class-2-ways-to.html).
3. If possible, try to use [volatile variables](http://www.javamadesoeasy.com/2015/03/volatile-keyword-in-java-difference.html). If a field is declared volatile all threads see a consistent value for the variable. Volatile variables at times can be used as alternate to synchronized methods as well.
4. Final variables are thread safe because once assigned some reference of object they cannot point to reference of other object.

s is pointing to String object.

public class MyClass {

final String s=new String("a");

void method(){

s="b"; //compilation error, s cannot point to new reference. } }

If final is holding some primitive value it cannot point to other value.

public class MyClass {

final inti=0;

void method(){

i=0; //compilation error, i cannot point to new value.

} }

1. Usage of local variables : If possible try to use local variables, local variables are thread safe, because every [thread has its own stack](http://www.javamadesoeasy.com/2015/03/threads-implement-their-own-stack.html), i.e. every thread has its own local variables and its pushes all the local variables on stack.

public class MyClass {

void method(){

inti=0; //Local variable, is thread safe.

} }

1. Using thread safe collections : Rather than using ArrayList we must Vector and in place of using HashMap we must use ConcurrentHashMap or HashTable.
2. We must use [VisualVM](http://www.javamadesoeasy.com/2015/03/visualvm-thread-dumps-generating-and_74.html)  or [jstack](http://www.javamadesoeasy.com/2015/03/jstack-thread-dumps-generating-and.html)  to detect problems such as deadlocks and time taken by threads to complete in multi threading programs.
3. Using [ThreadLocal](http://www.javamadesoeasy.com/2015/03/threadlocal-in-multithreading-in-java.html):ThreadLocal is a class which provides thread-local variables. Every thread has its own ThreadLocal value that makes ThreadLocal value threadsafe as well.
4. Rather than StringBuffer try using immutable classes such as String. Any change to String produces new String.

**Question 33. How thread can enter waiting, sleeping and blocked state and how can they go to runnable state ?**

Answer.  This is very prominently asked question in interview which will test your knowledge about [thread states](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html). And it’s very important for developers to have in depth knowledge of this [thread state](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html)transition. I will try to explain this thread state transition by framing few sub questions. I hope reading sub questions will be quite interesting.

> How can Thread go from running to waiting state ?

 By calling wait()[method](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) thread go from running to waiting state. In waiting state it will wait for other threads to release object monitor/lock.

> How can Thread return from waiting to runnable state ?

 Once notify() or notifyAll()[method](http://www.javamadesoeasy.com/2015/03/difference-between-notify-and-notifyall.html) is called object monitor/lock becomes available and thread can again return to runnable state.

> How can Thread go from running to sleeping state ?

 By calling sleep() [method](http://www.javamadesoeasy.com/2015/03/sleep-method-in-threads-10-key-features.html)thread go from running to [sleeping](http://www.javamadesoeasy.com/2015/03/sleep-method-in-threads-10-key-features.html) state. In sleeping state it will wait for sleep time to get over.

> How can Thread return from sleeping to runnable state ?

 Once specified sleep time is up thread can again return to runnable state.

Suspend() [method](http://www.javamadesoeasy.com/2015/03/using-suspend-and-resume-method-in.html) can be used to put thread in waiting state and resume() method is the only way which could put thread in runnable state.

Thread also may go from running to waiting state if it is waiting for some I/O operation to take place. Once input is available thread may return to running state.

>When threads are in running state, yield()[method](http://www.javamadesoeasy.com/2015/03/yield-method-in-threads-8-key-features.html) can make thread to go in Runnable state.

**Question 34. Difference between notify() and notifyAll() methods, can you write a code to prove your point?**

Answer. Goodness. Theoretically you must have heard or you must be aware of differences between [notify() and notifyAll()](http://www.javamadesoeasy.com/2015/03/difference-between-notify-and-notifyall.html).But have you created program to achieve it? If not let’s do it.

First, I will like give you a brief description of what notify() and notifyAll() methods do.

notify()- Wakes up a single thread that is [waiting](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is random and occurs at the discretion of the implementation. A thread [waits](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) on an object's monitor by calling one of the wait methods.

[The awakened threads will not be able to proceed until the current thread relinquishes the lock on this object.](http://www.javamadesoeasy.com/2015/03/the-awakened-threads-will-not-be-able.html)

|  |
| --- |
| public final native void notify(); |

notifyAll()- Wakes up all threads that are waiting on this object's monitor. A thread waits on an object's monitor by calling one of the wait methods.

[The awakened threads will not be able to proceed until the current thread relinquishes the lock on this object.](http://www.javamadesoeasy.com/2015/03/the-awakened-threads-will-not-be-able.html)

|  |
| --- |
| public final native void notifyAll(); |

[Now it’s time to write down a program to prove the point.](http://www.javamadesoeasy.com/2015/03/difference-between-notify-and-notifyall.html)

**Question 35. Does thread leaves object lock when**[**sleep()**](http://www.javamadesoeasy.com/2015/03/sleep-method-in-threads-10-key-features.html)**method is called?**

Answer. When [sleep()](http://www.javamadesoeasy.com/2015/03/sleep-method-in-threads-10-key-features.html) method is called Thread does not leaves object lock and goes from running to waiting state. Thread [waits](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) for sleep time to over and once sleep time is up it goes from [waiting to runnable state](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html).

**Question 36. Does thread leaves object lock when wait() method is called?**

Answer. When [wait()](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) method is called Thread leaves the object lock and goes from [running to waiting state](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html). Thread waits for other threads on same object to call notify() or notifyAll() and once any of [notify() or notifyAll()](http://www.javamadesoeasy.com/2015/03/difference-between-notify-and-notifyall.html) is called it goes from waiting to runnable state and again acquires object lock.

**Question 37. What will happen if we don’t override run method?**

Answer.  This question will test your basic knowledge how start and run methods work internally in Thread Api.

When we call start() method on thread, it internally calls run() method with newly created thread. So, if we don’t override run() method newly created thread won’t be called and nothing will happen.

class MyThread extends Thread {

//don't override run() method

}

publicclass DontOverrideRun {

publicstaticvoid main(String[] args) {

System.out.println("main has started.");

MyThread thread1=new MyThread();

thread1.start();

System.out.println("main has ended");

}}

/\*OUTPUT

main has started.

main has ended.

\*/

As we saw in output, we didn’t override run() method that’s why on calling start() method nothing happened.

**Question 38. What will happen if we override start method?**

Answer. This question will again test your basic core java knowledge how overriding works at runtime, what what will be called at runtime and how start and run methods work internally in Thread Api.

When we call start() method on thread, it internally calls run() method with newly created thread. So, if we override start() method, run() method will not be called until we write code for calling run() method.

class MyThread extends Thread {

@Override

publicvoid run() {

System.out.println("in run() method");

}

@Override

publicvoid start(){

System.out.println("In start() method");

}}

publicclass OverrideStartMethod {

publicstaticvoid main(String[] args) {

System.out.println("main has started.");

MyThread thread1=new MyThread();

thread1.start();

System.out.println("main has ended."); }

}

/\*OUTPUT

main has started.

In start() method

main has ended.

\*/

**Question 39. Can we acquire lock on class? What are ways in which you can acquire lock on class?**

Answer.  Yes, we can acquire lock on [class’s class object in 2 ways to acquire lock on class](http://www.javamadesoeasy.com/2015/03/acquiring-lock-on-class-2-ways-to.html).

Thread can acquire lock on class’s class object by-

1. Entering synchronized block or

 Let’s say there is one class MyClass. Now we can create synchronization block, and parameter passed with synchronization tells which class has to be synchronized. In below code, we have synchronized MyClass

 synchronized (MyClass.class) {

   //thread has acquired lock on MyClass’s class object.

 }

1. by entering static synchronized methods.

 public staticsynchronizedvoid method1() {

   //thread has acquired lock on MyRunnable’s class object.

 }

As soon as thread entered Synchronization method, thread acquired lock on class’s class object.

Thread will leave lock when it exits static synchronized method.

**Question 40. Difference between object lock and class lock?**

Answer.  It is very important question from multithreading point of view. We must understand [difference between object lock and class lock](http://www.javamadesoeasy.com/2015/03/difference-between-object-lock-and.html) to answer interview, ocjp answers correctly.

|  |  |
| --- | --- |
| [Object lock](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html) | [Class lock](http://www.javamadesoeasy.com/2015/03/acquiring-lock-on-class-2-ways-to.html) |
| Thread can acquire [object lock](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html) by-   1. Entering synchronized block or 2. by entering synchronized methods. | Thread can acquire lock on [class’s class object](http://www.javamadesoeasy.com/2015/03/acquiring-lock-on-class-2-ways-to.html) by-   1. Entering synchronized block or 2. by entering static synchronized methods. |
| [Multiple threads may exist on same object but only one thread of that object can enter synchronized method at a time.](http://www.javamadesoeasy.com/2015/03/suppose-you-have-2-threads-thread-1-and_5.html)  [Threads on different object can enter same method at same time.](http://www.javamadesoeasy.com/2015/03/suppose-you-have-2-threads-thread-1-on.html) | Multiple threads may exist on [same](http://www.javamadesoeasy.com/2015/03/suppose-you-have-2-threads-thread-1-and_46.html) or [different objects](http://www.javamadesoeasy.com/2015/03/suppose-you-have-2-threads-thread-1-on_5.html) of class but only one thread can enter static synchronized method at a time. |
| Multiple objects of class may exist and every object has it’s own lock. | Multiple objects of class may exist but there is always one class’s class object lock available. |
| First let’s acquire [object lock](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html) by entering synchronized block.  Example- Let’s say there is one class MyClassand we have created it’s object and reference to that object is myClass. Now we can create synchronization block, and parameter passed with synchronization tells which object has to be synchronized. In below code, we have synchronized object reference by myClass.  MyClass myClass=newMyclass();   synchronized (myClass) {   }  As soon thread entered Synchronization block, thread acquired object lock on object referenced by myClass (by acquiring object’s monitor.)  Thread will leave lock when it exits synchronized block. | First let’s acquire lock on [class’s class object](http://www.javamadesoeasy.com/2015/03/acquiring-lock-on-class-2-ways-to.html) by entering synchronized block.  Example- Let’s say there is one class MyClass. Now we can create synchronization block, and parameter passed with synchronization tells which class has to be synchronized. In below code, we have synchronized MyClass   synchronized (MyClass.class) {   }  As soon as thread entered Synchronization block, thread acquired MyClass’s class object. Thread will leave lock when it exits synchronized block. |
| publicsynchronizedvoid method1() {  }  As soon as thread entered Synchronization method, thread acquired [object lock](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html).  Thread will leave lock when it exits synchronized method. | public staticsynchronizedvoid method1() {}  As soon as thread entered static Synchronization method, thread acquired lock on [class’s class object](http://www.javamadesoeasy.com/2015/03/acquiring-lock-on-class-2-ways-to.html).  Thread will leave lock when it exits synchronized method. |

**Question 41.**Suppose you have 2 threads (Thread-1 and Thread-2) on same object. Thread-1 is in synchronized method1(), can Thread-2 enter synchronized method2() at same time?

Answer.No, here when Thread-1 is in synchronized method1() it must be holding [lock on object’s monitor](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html)and will release lock on object’s monitor only when it exits synchronized method1(). So, Thread-2 will have to [wait](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html)for Thread-1 to release lock on object’s monitor so that it could enter synchronized method2().

Likewise, Thread-2 even cannot enter synchronized method1() which is being executed by Thread-1. Thread-2 will have to [wait](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) for Thread-1 to release lock on object’s monitor so that it could enter synchronized method1(). [Now, let’s see a program to prove our point.](http://www.javamadesoeasy.com/2015/03/suppose-you-have-2-threads-thread-1-and_5.html)

**Question 42.** Suppose you have 2 threads (Thread-1 and Thread-2) on same object. Thread-1 is in static synchronized method1(), can Thread-2 enter static synchronized method2() at same time?

Answer.No, here when Thread-1 is in static synchronized method1() it must be holding lock on [class class’s object](http://www.javamadesoeasy.com/2015/03/acquiring-lock-on-class-2-ways-to.html) and will release lock on class’s classobject only when it exits static synchronized method1(). So, Thread-2 will have to [wait](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html) for Thread-1 to release lock on class’s classobject so that it could enter static synchronized method2().

Likewise, Thread-2 even cannot enter static synchronized method1() which is being executed by Thread-1. Thread-2 will have to [wait](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) for Thread-1 to release lock on  class’s classobject so that it could enter static synchronized method1(). [Now, let’s see a program to prove our point.](http://www.javamadesoeasy.com/2015/03/suppose-you-have-2-threads-thread-1-and_46.html)

**Question 43.**Suppose you have 2 threads (Thread-1 and Thread-2) on same object. Thread-1 is in synchronized method1(), can Thread-2 enter static synchronized method2() at same time?

Answer.Yes, here when Thread-1 is in synchronized method1() it must be holding [lock on object’s monitor](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html) and Thread-2 can enter static synchronized method2() by acquiring lock on [class’s class object](http://www.javamadesoeasy.com/2015/03/acquiring-lock-on-class-2-ways-to.html). [Now, let’s see a program to prove our point.](http://www.javamadesoeasy.com/2015/03/suppose-you-have-2-threads-thread-1-and_65.html)

**Question 44.** Suppose you have thread and it is in synchronized method and now can thread enter other synchronized method from that method?

Answer.Yes, here when thread is in synchronized method it must be holding [lock on object’s monitor](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html) and using that lock thread can enter other synchronized method. [Now, let’s see a program to prove our point.](http://www.javamadesoeasy.com/2015/03/suppose-you-have-thread-and-it-is-in_5.html)

**Question 45.**Suppose you have thread and it is in static synchronized method and now can thread enter other static synchronized method from that method?

Answer.  Yes, here when thread is in static synchronized method it must be holding lock on [class’s class object](http://www.javamadesoeasy.com/2015/03/acquiring-lock-on-class-2-ways-to.html) and using that lock thread can enter other static synchronized method. [Now, let’s see a program to prove our point.](http://www.javamadesoeasy.com/2015/03/suppose-you-have-thread-and-it-is-in_16.html)

**Question 46.** Suppose you have thread and it is in static synchronized method and now can thread enter other non static synchronized method from that method?

Answer.Yes, here when thread is in static synchronized method it must be holding lock on [class’s class object](http://www.javamadesoeasy.com/2015/03/acquiring-lock-on-class-2-ways-to.html) and when it enters synchronized method it will hold [lock on object’s monitor](http://v/) as well.

So, now thread holds 2 locks (it’s also called nested synchronization)-

>first one on class’s class object.

>second one on object’s monitor (This lock will be released when thread exits non static method).[Now, let’s see a program to prove our point.](http://www.javamadesoeasy.com/2015/03/suppose-you-have-thread-and-it-is-in_41.html)

**Question 47.** Suppose you have thread and it is in synchronized method and now can thread enter other static synchronized method from that method?

Answer.Yes, here when thread is in synchronized method it must be holding [lock on object’s monitor](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html) and when it enters static synchronized method it will hold lock on [class’s class object](http://www.javamadesoeasy.com/2015/03/acquiring-lock-on-class-2-ways-to.html) as well.

So, now thread holds 2 locks (it’s also called nested synchronization)-

>first one on [object’s monitor](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html).

>second one on class’s class object.(This lock will be released when thread exits static method).[Now, let’s see a program to prove our point.](http://www.javamadesoeasy.com/2015/03/suppose-you-have-thread-and-it-is-in_17.html)

**Question 48.** Suppose you have 2 threads (Thread-1 on object1 and Thread-2 on object2). Thread-1 is in synchronized method1(), can Thread-2 enter synchronized method2() at same time?

Answer.Yes, here when Thread-1 is in synchronized method1() it must be holding [lock on object1’s monitor](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html). Thread-2 will acquire lock on object2’s monitor and enter synchronized method2().

Likewise, Thread-2 even enter synchronized method1() as well which is being executed by Thread-1 (because threads are created on different objects). [Now, let’s see a program to prove our point.](http://www.javamadesoeasy.com/2015/03/suppose-you-have-2-threads-thread-1-on.html)

**Question 49.** Suppose you have 2 threads (Thread-1 on object1 and Thread-2 on object2). Thread-1 is in static synchronized method1(), can Thread-2 enter static synchronized method2() at same time?

Answer.No, it might confuse you a bit that threads are created on different objects. But, not to forgot that multiple objects may exist but there is always one [class’s class object](http://www.javamadesoeasy.com/2015/03/acquiring-lock-on-class-2-ways-to.html) lock available.

Here, when Thread-1 is in static synchronized method1() it must be holding lock on class class’s object and will release lock on class’s classobject only when it exits static synchronized method1(). So, Thread-2 will have to [wait](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) for Thread-1 to release lock on class’s classobject so that it could enter static synchronized method2().

Likewise, Thread-2 even cannot enter static synchronized method1() which is being executed by Thread-1. Thread-2 will have to [wait](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) for Thread-1 to release lock on  [class’s classobject](http://www.javamadesoeasy.com/2015/03/acquiring-lock-on-class-2-ways-to.html) so that it could enter static synchronized method1(). [Now, let’s see a program to prove our point.](http://www.javamadesoeasy.com/2015/03/suppose-you-have-2-threads-thread-1-on_5.html)

**Question 50.**Difference between wait() and wait(long timeout), What are [thread states](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html) when these method are called?

Answer.

|  |  |
| --- | --- |
| [wait()](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) | wait(long timeout) |
| When [wait()](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) method is called on object, it causes causes the current thread to wait until another thread invokes the notify() or notifyAll() method for this object. | wait(long timeout) - Causes the current thread to wait until either another thread invokes the notify() or notifyAll() methods for this object, or a specified timeout time has elapsed. |
| When [wait()](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) is called on object - Thread enters from [running to waiting state](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html).  It [waits](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) for some other thread to call notify so that it could enter runnable state. | When wait(1000) is called on object - Thread enters from running to waiting state. Than even if notify() or notifyAll() is not called after  timeout time has elapsed thread will go from [waiting to runnable state](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html). |

**Question 51.**How can you implement your own Thread Pool in java?

What is [ThreadPool](http://www.javamadesoeasy.com/2015/03/implement-thread-pool-in-java.html)?

ThreadPool is a pool of threads which reuses a fixed number of threads  to execute tasks.

At any point, at most nThreads threads will be active processing tasks. If additional tasks are submitted when all threads are active, they will wait in the queue until a thread is available.

ThreadPool implementation internally uses [LinkedBlockingQueue](http://www.javamadesoeasy.com/2015/03/custom-implementation-of.html) for adding and removing tasks.

In this post i will be using LinkedBlockingQueue provide by java Api, you can refer this post for [implementing ThreadPool using custom LinkedBlockingQueue](http://www.javamadesoeasy.com/2015/03/implementing-threadpool-using-custom.html).

Need/Advantage of ThreadPool?

Instead of creating new thread every time for executing tasks, we can create ThreadPool which reuses a fixed number of threads for executing tasks.

As threads are reused, performance of our application improves drastically.

How ThreadPool works?

We will instantiate ThreadPool, in ThreadPool’s constructor nThreads number of threads are created and started.

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

Here 2 threads will be created and started in ThreadPool.

Then, threads will enter run() method of ThreadPoolsThread class and will call take() method on taskQueue.

* If tasks are available thread will execute task by entering run() method of task (As tasks executed always implements Runnable).

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

* Else waits for tasks to become available.

When tasks are added?

When execute() method of ThreadPool is called, it internally calls put() method on taskQueue to add tasks.

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

Once tasks are available all waiting threads are notified that task is available.

**Question 52.  What is significance of using [ThreadLocal](http://www.javamadesoeasy.com/2015/03/threadlocal-in-multithreading-in-java.html)?**

Answer.  This question will test your command in multi threading, can you really create some perfect multithreading application or not. [ThreadLocal](http://www.javamadesoeasy.com/2015/03/threadlocal-in-multithreading-in-java.html) is a class which provides thread-local variables.

What is [ThreadLocal](http://www.javamadesoeasy.com/2015/03/threadlocal-in-multithreading-in-java.html) ?

ThreadLocal is a class which provides thread-local variables. Every thread has its own ThreadLocal value that makes ThreadLocal value threadsafe as well.

For how long Thread holds ThreadLocal value?

Thread holds ThreadLocal value till it hasn’t entered [dead state](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html).

Can one thread see other thread’s ThreadLocal value?

No, thread can see only it’s ThreadLocal value.

Are ThreadLocal variables thread safe. Why?

Yes, ThreadLocal variables are thread safe. As every thread has its own ThreadLocal value and one thread can’t see other threads ThreadLocal value.

Application of [ThreadLocal](http://www.javamadesoeasy.com/2015/03/threadlocal-in-multithreading-in-java.html)?

1. ThreadLocal are used by many web frameworks for maintaining some context (may be session or request) related value.
   * In any single threaded application, same thread is assigned for every request made to same action, so ThreadLocal values will be available in next request as well.
   * In multi threaded application, different thread is assigned for every request made to same action, so ThreadLocal values will be different for every request.
2. When threads have started at different time they might like to store time at which they have started. So, thread’s start time can be stored in ThreadLocal.

Creating ThreadLocal >

|  |
| --- |
| private ThreadLocal<String> threadLocal =  new ThreadLocal<String>(); |

We will create instance of ThreadLocal. ThreadLocal is a generic class, i will be using String to demonstrate threadLocal.

All threads will see same instance of ThreadLocal, but a thread will be able to see value which was set by it only.

How thread set value of ThreadLocal >

|  |
| --- |
| threadLocal.set( new Date().toString()); |

Thread set value of ThreadLocal by calling set(“”) method on threadLocal.

How thread get value of ThreadLocal >

|  |
| --- |
| threadLocal.get() |

Thread get value of ThreadLocal by calling get() method on threadLocal.

See here for detailed explanation of [threadLocal](http://www.javamadesoeasy.com/2015/03/threadlocal-in-multithreading-in-java.html).

**Question 53. What is busy spin?**

Answer.

What is [busy spin](http://www.javamadesoeasy.com/2015/03/busy-spin-what-is-busy-spin-consumer.html)?

When one thread loops continuously waiting for another thread to signal.

Performance point of view - Busy spin is very bad from performance point of view, because one thread keeps on looping continuously ( and consumes CPU) waiting for another thread to signal.

Solution to busy spin -

We must use [sleep()](http://www.javamadesoeasy.com/2015/03/sleep-method-in-threads-10-key-features.html) or [wait() and notify()](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) method. Using wait() is better option.

Why using wait() and notify() is much better option to solve busy spin?

Because in case when we use sleep() method, thread will wake up again and again after specified sleep time until boolean variable is true. But, in case of wait() thread will wake up only when when notified by calling [notify() or notifyAll()](http://www.javamadesoeasy.com/2015/03/difference-between-notify-and-notifyall.html), hence end up consuming CPU in best possible manner.

Program - Consumer Producer problem with busy spin >

Consumer thread continuously execute (busy spin) in while loop tillproductionInProcess is true. Once producer thread has ended it will make boolean variable productionInProcess false and busy spin will be over.

while(productionInProcess){

System.out.println("BUSY SPIN - Consumer waiting for production to get over");

}

**Question 54. Can a constructor be synchronized?**

Answer.  No, constructor cannot be synchronized. Because constructor is used for instantiating object, when we are in constructor object is under creation. So, until object is not instantiated it does not need any synchronization.

Enclosing constructor in synchronized block will generate compilation error.

Using synchronized in constructor definition will also show compilation error.

COMPILATION ERROR = Illegal modifier for the constructor in type ConstructorSynchronizeTest; only public, protected & private are permitted

Though we can use synchronized block inside constructor.

Read More about : [Constructor in java cannot be synchronized](http://www.javamadesoeasy.com/2015/03/constructor-in-java-cannot-be.html)

**Question 55. Can you find whether thread holds lock on object or not?**

Answer.  holdsLock(object) method can be used to find out whether current thread holds the lock on monitor of specified object.

holdsLock(object) method returns true if the current thread holds the lock on monitor of specified object.

**Question 56. What do you mean by thread starvation?**

Answer.  When thread does not enough CPU for its execution Thread starvation happens.

Thread starvation may happen in following scenarios >

* Low priority threads gets less CPU (time for execution) as compared to high priority threads. Lower priority thread may starve away waiting to get enough CPU to perform calculations.
* In [deadlock](http://www.javamadesoeasy.com/2015/03/deadlock-in-multithreading-program-to.html) two threads waits for each other to release lock holded by them on resources. There both Threads starves away to get CPU.
* Thread might be waiting indefinitely for lock on object’s monitor (by calling [wait()](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) method), because no other thread is calling [notify()/notifAll()](http://www.javamadesoeasy.com/2015/03/difference-between-notify-and-notifyall.html) method on object. In that case, Thread starves away to get CPU.
* Thread might be waiting indefinitely for lock on object’s monitor (by calling wait() method), but notify() may be repeatedly awakening some other threads. In that case also Thread starves away to get CPU.

**Question 57. What is addShutdownHook method in java?**

Answer.  [addShutdownHook](http://www.javamadesoeasy.com/2015/03/threads-addshutdownhook-method-in-java.html) method in java >

* addShutdownHook method registers a new virtual-machine shutdown hook.
* A shutdown hook is a initialized but unstarted thread.
* When JVM starts its shutdown it will start all registered shutdown hooks in some unspecified order and let them run concurrently.

When JVM (Java virtual machine)  shuts down >

* When the last non-[daemon](http://www.javamadesoeasy.com/2015/03/daemon-threads-12-salient-features-of.html) thread finishes, or
* when the System.exit is called.

Once JVM’s shutdown has begunnew shutdown hook cannot be registered neither  previously-registered hook can be de-registered. Any attempt made to do any of these operations causes an IllegalStateException.

For more detail with program read : [Threads addShutdownHook method in java](http://www.javamadesoeasy.com/2015/03/threads-addshutdownhook-method-in-java.html)

**Question 58. How you can handle uncaught runtime exception generated in run method?**

Answer.  We can use [setDefaultUncaughtExceptionHandler](http://www.javamadesoeasy.com/2015/03/handling-uncaught-runtime-exception.html) method which can handle uncaught unchecked(runtime) exception generated in run() method.

What is setDefaultUncaughtExceptionHandler method?

setDefaultUncaughtExceptionHandler method sets the default handler which is called when a thread terminates due to an uncaught unchecked(runtime) exception.

setDefaultUncaughtExceptionHandler method features >

* setDefaultUncaughtExceptionHandler method sets the default handler which is called when a thread terminates due to an uncaught unchecked(runtime) exception.
* setDefaultUncaughtExceptionHandler is a static method method, so we can directly call  Thread.setDefaultUncaughtExceptionHandler to set the default handler to handle uncaught unchecked(runtime) exception.
* It avoids abrupt termination of thread caused by uncaught runtime exceptions.

Defining setDefaultUncaughtExceptionHandler method >

Thread.setDefaultUncaughtExceptionHandler(new Thread.UncaughtExceptionHandler(){

publicvoid uncaughtException(Thread thread, Throwable throwable) { System.out.println(thread.getName() + " has thrown " + throwable); } });

**Question 59. What is ThreadGroup in java, What is default priority of newly created threadGroup, mention some important ThreadGroup methods ?**

Answer.  When program starts JVM creates  a ThreadGroup named main. Unless specified, all  newly created threads become members of the main thread group.

ThreadGroup is initialized with default priority of 10.

ThreadGroup important methods >

* getName()
  + name of ThreadGroup.
* activeGroupCount()
  + count of active groups in ThreadGroup.
* activeCount()
  + count of active threads in ThreadGroup.
* list()
  + list() method has prints ThreadGroups information
* getMaxPriority()
  + Method returns the maximum priority of ThreadGroup.
* setMaxPriority(int pri)
  + Sets the maximum priority of ThreadGroup.

**Question 60. What are thread priorities?**

Answer.

[Thread Priority](http://www.javamadesoeasy.com/2015/03/thread-priorities-setpriority-and.html) range is from 1 to 10.

Where 1 is minimum priority and 10 is maximum priority.

Thread class provides variables of final static int type for setting thread priority.

/\* The minimum priority that a thread can have. \*/

publicfinalstaticintMIN\_PRIORITY= 1;

/\* The default priority that is assigned to a thread. \*/

publicfinalstaticintNORM\_PRIORITY= 5;

/\* The maximum priority that a thread can have. \*/

publicfinalstaticintMAX\_PRIORITY= 10;

|  |
| --- |
|  |

Thread with MAX\_PRIORITY is likely to get more CPU as compared to low priority threads. But occasionally low priority thread might get more CPU. Because thread scheduler schedules thread on discretion of implementation and [thread behaviour is totally unpredictable](http://www.javamadesoeasy.com/2015/03/thread-behaviour-is-unpredictable.html).

Thread with MIN\_PRIORITY is likely to get less CPU as compared to high priority threads. But occasionally high priority thread might less CPU. Because thread scheduler schedules thread on discretion of implementation and thread behaviour is totally unpredictable.

setPriority()method is used for Changing the priority of thread.

getPriority()method returns the thread’s priority.