1. **How to stop thread in Java?**

I always said that Java provides rich APIs for everything but ironically Java doesn't provide a sure shot way of stopping thread. There was some control methods in JDK 1.0 e.g. stop(), suspend() and resume() which was deprecated in later releases due to potential deadlock threats, from then Java API designers has not made any effort to provide a consistent, thread-safe and elegant way to stop threads. Programmers mainly rely on the fact that thread stops automatically as soon as they finish execution of run() or call() method.

To manually stop, programmers either take advantage of volatile boolean variable and check in every iteration if run method has loops or interrupt threads to abruptly cancel tasks. See this tutorial for sample code of stopping thread in Java.

1. **Why wait, notify and notifyAll are not inside thread class?**

This is a design related question, which checks what candidate thinks about existing system or does he ever thought of something which is so common but looks in-appropriate at first. In order to answer this question, you have to give some reasons why it make sense for these three method to be in Object class, and why not on Thread class. One reason which is obvious is that Java provides lock at object level not at thread level. Every object has lock, which is acquired by thread. Now if thread needs to wait for certain lock it make sense to call wait() on that object rather than on that thread. Had wait() method declared on Thread class, it was not clear that for which lock thread was waiting. In short, since wait, notify and notifyAll operate at lock level, it make sense to defined it on object class because lock belongs to object. You can also see this article for more elaborate answer of this question.

1. **How do you check if a Thread holds a lock or not?**

I didn't even know that you can check if a Thread already holds lock before this question hits me in a telephonic round of Java interview. There is a method called holdsLock() on java.lang.Thread, it returns true if and only if the current thread holds the monitor lock on the specified object. You can also check this article for more detailed answer.

1. **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 join() method of thread class to start a thread when another one is 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, this ways T1 will finish first and T3 will finish last. To learn more about join method

1. **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 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 same thread to get CPU back and start its execution again

1. **What is 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 tool for different needs. Wait method is used for inter thread communication, it relinquish lock if waiting condition is true and wait for notification when due to 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 current thread. You can also take look at this article to answer this question with more details.

1. **What is the difference between yield and sleeping?**

yield() - yield method pauses the currently executing thread temporarily for giving a chance to the remaining waiting threads of the same priority to execute.   
  
sleep() - It causes the current thread to suspend execution for a specified period. When a thread goes into sleep state it doesn't release the lock

##### What is a daemon thread?

Daemon thread in Java are those thread which runs in background and mostly created by JVM for performing background task like Garbage collection and other house keeping tasks. Thread.setDaemon(true) makes a thread daemon but it can only be called before starting Thread in Java.

##### What is Starvation?

Starvation is a situation where a thread is unable to get continuous access to shared resources and unable to make progress. This happen when some another thread use shared resource for longer time.

### Is it possible to start a thread twice?

No, there is no possibility to start a thread twice. If we does, it throws an exception.

### Can we make the user thread as daemon thread if thread is started?

No, if you do so, it will throw IllegalThreadStateException

##### .[What is the difference between preemptive scheduling and time slicing?](http://www.careerride.com/Java-thread-preemptive-scheduling-vs-time-slicing.aspx)

Under Preemptive scheduling, the task with the highest priority is executed until it enters the waiting or dead states or some other high priority task cones into existence for execution. On the other hand, the task under time slicing is executed for a predefined amount of time and then enters the pool of ready tasks.

1. **What is the difference between a synchronized method and a synchronized block?**

A synchronized method ensures that the whole method is executed completely by one thread before another thread can invoke it. A synchronized block restricts the mutual exclusion to only that block of code. One should try to make the synchronized block as small as possible.

1. **What is a livelock?**

Livelock is when two threads or processes get deadlocked because they try to perform the same action to come out of a potential deadlock. A real-world example of livelock occurs when two people meet in a corridor and each tries to be polite by moving aside. Problem is they move in the same direction and are still in each other's way.

1. **Why do you need synchronization?**

Synchronization is needed when a shared resource is being accessed and modified by different threads. Without Synchronization, the resource may be found in an inconsistent state if one thread gets pre-empted before it can complete it's action, e.g. a node is being added to a tree and another thread tries to traverse the tree.

1. **How is your Java program executed inside JVM?**
   1. When JVM executes a Java application, a runtime instance of JVM is born.This runtime instance invoke main() method of Java application.
   2. The main() method of an application serves as the starting point for that application's initial thread. The initial thread can in turn fire off other threads.
   3. This thread has a program counter(PC) and Java stack.Whenever main() method is invoked, a stack frame is pushed onto the stack,this then becomes the active tack frame.The program counter in the new Java stack frame will point to the beginning of the method.If there are more method invocations within main() method then this process of pushing new stack frame onto the stack for each method call is repeated as and when they are invoked.When a method returns, the active frame is popped from the stack and the one below becomes the active stack frame.
   4. The PC is set to the instruction after the method call and the method continues.There is only one heap corresponding to an instance of JVM and all objects created are stored here.This heap is shared by all threads created in an application.Inside the Java virtual machine, threads come in two flavors: daemon and non- daemon. A daemon thread is ordinarily a thread used by the virtual machine itself, such as a thread that performs garbage collection. The application, however, can mark any threads it creates as daemon threads. The initial thread of an application--the one that begins at main()--is a non- daemon thread.A Java application continues to execute (the virtual machine instance continues to live) as long as any non-daemon threads are still running. When all non-daemon threads of a Java application terminate, the virtual machine instance will exit. If permitted by the security manager, the application can also cause its own demise by invoking the exit() method of class Runtime or System.When main() returns,it terminates the application's only non-daemon thread, which causes the virtual machine instance to exit.

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |