**Java Multi-Threading Interview Question**

**Q. What is multithreading?**

**Ans.** Multithreading is a process of executing multiple threads simultaneously. Its main advantage is:

1. Threads share the same address space.
2. Thread is lightweight.
3. Cost of communication between processes is low.

**Q. What invokes a thread's run() method?**

**Ans.** After a thread is started, via its start() method or that of the Thread class, the JVM invokes the thread's run() method when the thread is initially executed.

**Q. What state is a thread in when it is executing?**

**Ans.** An executing thread is in the running state.

**Q. How can a dead thread be restarted?**

**Ans.** A dead thread cannot be restarted.

**Q. What are three ways in which a thread can enter the waiting state?**

**Ans.** A thread can enter the waiting state by invoking its sleep() method, by blocking on I/O, by unsuccessfully attempting to acquire an object's lock, or by invoking an object's wait() method. It can also enter the waiting state by invoking its (deprecated) suspend() method.

**Q. What method must be implemented by all threads?**

**Ans.** All tasks must implement the run() method, whether they are a subclass of Thread or implement the Runnable interface.

**Q. What are synchronized methods and synchronized statements?**

**Ans.** Synchronized methods are methods that are used to control access to an object. A thread only executes a synchronized method after it has acquired the lock for the method's object or class. Synchronized statements are similar to synchronized methods. A synchronized statement can only be executed after a thread has acquired the lock for the object or class referenced in the synchronized statement.

**Q. What are the two basic ways in which classes that can be run as threads may be defined?**

**Ans.** A thread class may be declared as a subclass of Thread, or it may implement the Runnable interface.

**Q. How to make application thread-safe ?**

**Ans.** You should use the word synchronized to mark the critical section of code. You may also use other methods of thread synchronization (see wait(), notify(), notifyAll() etc.

**Q. What does the keyword "synchronize" mean in java. When do you use it? What are the**

**disadvantages of synchronization?**

**Ans.** Synchronize is used when u want to make your methods thread safe. The disadvantage of synchronise is it will end up in slowing down the program. Also if not handled properly it will end up in deadlock.

1. Only use (and minimize its use)synchronization when writing multithreaded code as there is a speed (up to five to six time slower, depending on the execution time of the synchronized/non-synchronized method ) cost associated with its use.

2. In case of synchronized method modifier, the bytecode generated is the exact same as non-synchronized method. The only difference is that a flag called ACC\_SYNCRONIZED property flag in method's method\_info structure is set if the synchronized method modifier is present.

3. Also, synchronized keyword can make the code larger in size if used in the body of the method as bytecode for monitorenter/monitorexit is generated in addition to any exception handling.

**Q. What method is invoked to cause an object to begin executing as a separate thread?**

**Ans.** The start() method of the Thread class is invoked to cause an object to begin executing as a separate thread.

**Q. What is thread?**

**Ans:** A thread is a lightweight subprocess.It is a separate path of execution. It is called separate path of execution because each thread runs in a separate stack frame.

**Q. What is the difference between preemptive scheduling and time slicing?**

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

**Q. What does join() method?**

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

**Q. What is difference between wait() and sleep() method?**

**Ans:**

**wait() sleep()**

1) The wait() method is defined in Object class. The sleep() method is defined in Thread class.

2) wait() method releases the lock. The sleep() method doesn't releases the lock.

**Q. Is it possible to start a thread twice?**

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

**Q. Can we call the run() method instead of start()?**

**Ans:** Yes, but it will not work as a thread rather it will work as a normal object so there will not be context-switching between the threads. Each thread starts in a separate call stack. Invoking the run() method from main thread, the run() method goes onto the current call stack rather than at the beginning of a new call stack.

class TestCallRun1 extends Thread{

public void run(){

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

}

public static void main(String args[]){

TestCallRun1 t1=new TestCallRun1();

t1.run();//fine, but does not start a separate call stack

}

}

**Q. What about the daemon threads?**

**Ans.** The daemon threads are basically the low priority thread that provides the background support to the user threads. It provides services to the user threads.

Daemon thread in java is a service provider thread that provides services to the user thread. Its life depend on the mercy of user threads i.e. when all the user threads dies, JVM terminates this thread automatically.

There are many java daemon threads running automatically e.g. gc, finalizer etc.

You can see all the detail by typing the jconsole in the command prompt. The jconsole tool provides information about the loaded classes, memory usage, running threads etc.

**Points to remember for Daemon Thread in Java**

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

Its life depends on user threads.

It is a low priority thread.

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

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

**Methods for Java Daemon thread by Thread class**

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

**Method** **Description**

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

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

**Simple example of Daemon thread in java**

File: MyThread.java

public class TestDaemonThread1 extends Thread{

public void run(){

if(Thread.currentThread().isDaemon()){//checking for daemon thread

System.out.println("daemon thread work");

}

else{

System.out.println("user thread work");

}

}

public static void main(String[] args){

TestDaemonThread1 t1=new TestDaemonThread1();//creating thread

TestDaemonThread1 t2=new TestDaemonThread1();

TestDaemonThread1 t3=new TestDaemonThread1();

t1.setDaemon(true);//now t1 is daemon thread

t1.start();//starting threads

t2.start();

t3.start();

}

}

**Q. Can we make the user thread as daemon thread if thread is started?**

**Ans:** No, if you do so, it will throw IllegalThreadStateException

**Q. What is shutdown hook?**

**Ans.** The shutdown hook is basically a thread i.e. invoked implicitly before JVM shuts down. So we can use it perform clean up resource.

The shutdown hook can be used to perform cleanup resource or save the state when JVM shuts down normally or abruptly. Performing clean resource means closing log file, sending some alerts or something else. So if you want to execute some code before JVM shuts down, use shutdown hook.

**Q. When does the JVM shut down?**

**Ans.** The JVM shuts down when:

user presses ctrl+c on the command prompt

System.exit(int) method is invoked

user logoff

user shutdown etc.

The addShutdownHook(Runnable r) method

The addShutdownHook() method of Runtime class is used to register the thread with the Virtual Machine. **Syntax:**

public void addShutdownHook(Runnable r){}

The object of Runtime class can be obtained by calling the static factory method getRuntime(). For example:

Runtime r = Runtime.getRuntime();

Factory method

The method that returns the instance of a class is known as factory method.

Simple example of Shutdown Hook

class MyThread extends Thread{

public void run(){

System.out.println("shut down hook task completed..");

}

}

public class TestShutdown1{

public static void main(String[] args)throws Exception {

Runtime r=Runtime.getRuntime();

r.addShutdownHook(new MyThread());

System.out.println("Now main sleeping... press ctrl+c to exit");

try{Thread.sleep(3000);}catch (Exception e) {}

}

}

**Q. When should we interrupt a thread?**

**Ans:** We should interrupt a thread if we want to break out the sleep or wait state of a thread.

**Interrupting a Thread:**

If any thread is in sleeping or waiting state (i.e. sleep() or wait() is invoked), calling the interrupt() method on the thread, breaks out the sleeping or waiting state throwing InterruptedException. If the thread is not in the sleeping or waiting state, calling the interrupt() method performs normal behaviour and doesn't interrupt the thread but sets the interrupt flag to true. Let's first see the methods provided by the Thread class for thread interruption.

The 3 methods provided by the Thread class for interrupting a thread

**public void interrupt()**

**public static boolean interrupted()**

**public boolean isInterrupted()**

**Example of interrupting a thread that stops working**

In this example, after interrupting the thread, we are propagating it, so it will stop working. If we don't want to stop the thread, we can handle it where sleep() or wait() method is invoked. Let's first see the example where we are propagating the exception.

class TestInterruptingThread1 extends Thread{

public void run(){

try{

Thread.sleep(1000);

System.out.println("task");

}catch(InterruptedException e){

throw new RuntimeException("Thread interrupted..."+e);

}

}

public static void main(String args[]){

TestInterruptingThread1 t1=new TestInterruptingThread1();

t1.start();

try{

t1.interrupt();

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

}

}

Test it Now

Output:Exception in thread-0

java.lang.RuntimeException: Thread interrupted...

java.lang.InterruptedException: sleep interrupted

at A.run(A.java:7)

**Example of interrupting a thread that doesn't stop working**

In this example, after interrupting the thread, we handle the exception, so it will break out the sleeping but will not stop working.

class TestInterruptingThread2 extends Thread{

public void run(){

try{

Thread.sleep(1000);

System.out.println("task");

}catch(InterruptedException e){

System.out.println("Exception handled "+e);

}

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

}

public static void main(String args[]){

TestInterruptingThread2 t1=new TestInterruptingThread2();

t1.start();

t1.interrupt();

}

}

Output:Exception handled

java.lang.InterruptedException: sleep interrupted

thread is running...

**Example of interrupting thread that behaves normally**

If thread is not in sleeping or waiting state, calling the interrupt() method sets the interrupted flag to true that can be used to stop the thread by the java programmer later.

class TestInterruptingThread3 extends Thread{

public void run(){

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

System.out.println(i);

}

public static void main(String args[]){

TestInterruptingThread3 t1=new TestInterruptingThread3();

t1.start();

t1.interrupt();

}

}

Test it Now

Output:1

2

3

4

5

**What about isInterrupted and interrupted method?**

The isInterrupted() method returns the interrupted flag either true or false. The static interrupted() method returns the interrupted flag afterthat it sets the flag to false if it is true.

public class TestInterruptingThread4 extends Thread{

public void run(){

for(int i=1;i<=2;i++){

if(Thread.interrupted()){

System.out.println("code for interrupted thread");

}

else{

System.out.println("code for normal thread");

}

}//end of for loop

}

public static void main(String args[]){

TestInterruptingThread4 t1=new TestInterruptingThread4();

TestInterruptingThread4 t2=new TestInterruptingThread4();

t1.start();

t1.interrupt();

t2.start();

}

}

Output:Code for interrupted thread

code for normal thread

code for normal thread

code for normal thread

**Q. What is synchronization?**

**Ans:** Synchronization is the capability of control the access of multiple threads to any shared resource. It is used:

To prevent thread interference.

To prevent consistency problem.

**Q. What is the purpose of Synchronized block?**

**Ans:** Synchronized block is used to lock an object for any shared resource.Scope of synchronized block is smaller than the method. Synchronized block can be used to perform synchronization on any specific resource of the method. Suppose you have 50 lines of code in your method, but you want to synchronize only 5 lines, you can use synchronized block. If you put all the codes of the method in the synchronized block, it will work same as the synchronized method.

Points to remember for Synchronized block:

Synchronized block is used to lock an object for any shared resource.

Scope of synchronized block is smaller than the method.

**Syntax to use synchronized block**

synchronized (object reference expression) {

//code block

}

**Example of synchronized block**

Let's see the simple example of synchronized block.

Program of synchronized block

class Table{

void printTable(int n){

synchronized(this){//synchronized block

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

System.out.println(n\*i);

try{

Thread.sleep(400);

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

}

}

}//end of the method

}

class MyThread1 extends Thread{

Table t;

MyThread1(Table t){

this.t=t;

}

public void run(){

t.printTable(5);

}

}

class MyThread2 extends Thread{

Table t;

MyThread2(Table t){

this.t=t;

}

public void run(){

t.printTable(100);

}

}

public class TestSynchronizedBlock1{

public static void main(String args[]){

Table obj = new Table();//only one object

MyThread1 t1=new MyThread1(obj);

MyThread2 t2=new MyThread2(obj);

t1.start();

t2.start();

}

}

**Q. Can Java object be locked down for exclusive use by a given thread?**

**Ans:** Yes. You can lock an object by putting it in a "synchronized" block. The locked object is inaccessible to any thread other than the one that explicitly claimed it.

**Q. What is static synchronization?**

**Ans:** If you make any static method as synchronized, the lock will be on the class not on object.

Problem without static synchronization.

Suppose there are two objects of a shared class(e.g. Table) named object1 and object2.In case of synchronized method and synchronized block there cannot be interference between t1 and t2 or t3 and t4 because t1 and t2 both refers to a common object that have a single lock.But there can be interference between t1 and t3 or t2 and t4 because t1 acquires another lock and t3 acquires another lock.I want no interference between t1 and t3 or t2 and t4.Static synchronization solves this problem.

**Example of static synchronization**

In this example we are applying synchronized keyword on the static method to perform static synchronization.

class Table{

synchronized static void printTable(int n){

for(int i=1;i<=10;i++){

System.out.println(n\*i);

try{

Thread.sleep(400);

}catch(Exception e){}

}

}

}

class MyThread1 extends Thread{

public void run(){

Table.printTable(1);

}

}

class MyThread2 extends Thread{

public void run(){

Table.printTable(10);

}

}

class MyThread3 extends Thread{

public void run(){

Table.printTable(100);

}

}

class MyThread4 extends Thread{

public void run(){

Table.printTable(1000);

}

}

public class TestSynchronization4{

public static void main(String t[]){

MyThread1 t1=new MyThread1();

MyThread2 t2=new MyThread2();

MyThread3 t3=new MyThread3();

MyThread4 t4=new MyThread4();

t1.start();

t2.start();

t3.start();

t4.start();

}

}

**Q. What is the difference between notify() and notifyAll()?**

**Ans:** The notify() is used to unblock one waiting thread whereas notifyAll() method is used to unblock all the threads in waiting state.

**Q. What is deadlock?**

**Ans:** Deadlock is a situation when two threads are waiting on each other to release a resource. Each thread waiting for a resource which is held by the other waiting thread.

**Q. Difference between Thread and Process in Java?**

**Ans:** Thread is 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 different thread and used to store local data to that thread.

**Q.  When to use Runnable vs Thread in Java?**

**Ans:** As we know we can implement thread either by extending Thread class or implementing Runnable interface, if you know that Java programming language doesn't support multiple inheritance of class, but it allows you to implement multiple interface. Which means, its better to implement Runnable than extends Thread, if you also want to extend another class e.g. Canvas or CommandListener. For more points and discussion you can also refer this [post](http://javarevisited.blogspot.sg/2012/01/difference-thread-vs-runnable-interface.html).  
  
**Q.  Difference between start() and run() method of Thread class?**

**Ans:** One of trick Java question from early days, but still good enough to differentiate between shallow understanding of Java threading model start() method is used to start newly created thread, while start() internally calls run() method, there is difference calling run() method directly. When you invoke run() as normal method, its called in the same thread, no new thread is started, which is the case when you call start()method.   
  
**Q. Difference between Runnable and Callable in Java?**  
Ans: Both Runnable and Callable represent task which is intended to be executed in 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 result of computation.

**Q.  Difference between CyclicBarrier and CountDownLatch in Java?**  
Ans: Though both CyclicBarrier and CountDownLatch wait for number of threads on one or more events, 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

**Q.  What is Java Memory model?**  
Ans: 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.

**Q. What is volatile variable in Java?**

**Ans: V**olatile 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.  
  
**Q. What is race condition in Java? Given one example?**

**Ans:** Race condition are cause of some subtle programming bugs when Java programs are exposed to concurrent execution environment. As name suggests, 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.

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

**Ans:** I always said that Java provides rich APIs for everything but ironically Java doesn't provide a sure shot way of stopping thread. There was some control methods in JDK 1.0 e.g. stop(), suspend() and resume() which was deprecated in later releases due to potential deadlock threats, from then Java API designers has not made any effort to provide a consistent, thread-safe and elegant way to stop threads. Programmers mainly rely on the fact that thread stops automatically as soon as they finish execution of run() or call() method. To manually stop, programmers either take advantage of volatile boolean variable and check in every iteration if run method has loops or interrupt threads to abruptly cancel tasks. See this [tutorial](http://javarevisited.blogspot.com/2011/10/how-to-stop-thread-java-example.html) for sample code of stopping thread in Java.

**Q. What happens when an Exception occurs in a thread?**

**Ans:** In simple words, If not caught thread will die, if an uncaught exception handler is registered then it will get a call back. Thread.UncaughtExceptionHandler is an interface, defined as nested interface for handlers invoked when a Thread abruptly terminates due to an uncaught exception. When a thread is about to terminate due to an uncaught exception the Java Virtual Machine will query the thread for its UncaughtExceptionHandler using Thread.getUncaughtExceptionHandler() and will invoke the handler's uncaughtException() method, passing the thread and the exception as arguments.

**Q. How do you share data between two thread in Java?**

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

**Ans:** 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.  
  
**Q. Why wait, notify and notifyAll are not inside thread class?**

**Ans:** This is a design related question. 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.

**Q. What is ThreadLocal variable in Java?**

**Ans:** 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 in 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.  
  
**Q. What is FutureTask in Java?**

**Ans:** 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. AFutureTask 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.

**Q. Difference between interrupted and isInterrupted method in Java?**

**Ans:** 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  
  
**Q. Why wait and notify methods are called from synchronized block?**

**Ans:** 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 race condition between wait and notify calls. To learn more about this, check my similarly titled post [here](http://javarevisited.blogspot.com/2011/05/wait-notify-and-notifyall-in-java.html).  
  
**Q. Why you should check condition for waiting in a loop?**

**Ans:** It’s 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.  
  
**Q. Difference between synchronized and concurrent collection in Java?**

**Ans:** 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.

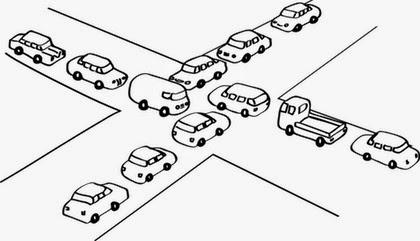
**Q. Difference between Stack and Heap in Java?**

**Ans:** 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 other hand, 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 comes in picture. volatile suggest threads to read value of variable always from main memory. See this [article](http://javarevisited.blogspot.com/2013/01/difference-between-stack-and-heap-java.html) to learn more about stack and heap in Java to answer this question in greater detail.

**Q. What is thread pool? Why should you thread pool in Java?**

Ans: 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 issue, 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 thread) 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.  
  
**Q. Write code to solve Producer Consumer problem in Java?**

**Ans:** Most of the threading problem you solved in real world are of category of Producer consumer pattern, where one thread is producing task and other thread is consuming that. You must know how to do inter thread communication to solve this problem. At lowest level, you can use wait and notify to solve this problem, and at 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).  
  
**Q. 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 happens your program hangs and doesn't do the task it is intended for. In order for deadlock to happen, following four condition 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.

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.

**Q. Difference between livelock and deadlock in Java?**  
**Ans:** 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, main difference between livelock and deadlock is that in former state of process change but no progress is made.  
  
**Q. How do you check if a Thread holds a lock or not?**  
**Ans:** 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.

**Q. How do you take thread dump in Java?**  
**Ans:** 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.  
  
**Q. Which JVM parameter is used to control stack size of thread?**  
**Ans**: 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.  
  
**Q. Difference between synchronized and ReentrantLock in Java?**  
**Ans:** There were days when 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 behaviour 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) to learn about those capabilities and some more differences between synchronized vs ReentrantLock in Java.  
  
**Q. There are three threads T1, T2 and T3? How do you ensure sequence T1, T2, T3 in Java?**

**Ans:**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.

**Q. What does yield method of Thread class do?**  
**Ans:** 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. See this [article](http://java67.blogspot.sg/2012/08/difference-between-yield-and-wait.html) to learn more about yield method and to answer this question better.  
  
**Q. What is concurrence level of ConcurrentHashMap in Java?**  
**Ans:** ConcurrentHashMap achieves it's scalability and thread-safety by partitioning actual map into number of sections. This partitioning is achieved using concurrency level. It's optional parameter of ConcurrentHashMap constructor and it's default value is 16. The table is internally partitioned to try to permit the indicated number of concurrent updates without contention. To learn more about concurrency level and internal resizing, see my post [How ConcurrentHashMap works in Java](http://javarevisited.blogspot.com/2013/02/concurrenthashmap-in-java-example-tutorial-working.html).  
  
**Q. What is Semaphore in Java?**  
**Ans:** 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 expensive resource which is available in fixed number e.g. database connection in pool. See this [article](http://javarevisited.blogspot.com/2012/05/counting-semaphore-example-in-java-5.html) to learn more about counting Semaphore in Java.  
  
**Q. What happens if you submit task, when queue of thread pool is already fill?**  
**Ans**: Many programmer 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.

**Q. Difference between submit() and execute() method thread pool in Java?**  
**Ans:** Both method are ways to submit task to thread pools but there is 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. It's return type is void. On other hand submit() is overloaded method, it can take either Runnable or Callable task and can return Future object which can hold pending result of computation. This method is defined on ExecutorService interface, which extends Executor interface, and every other thread pool class e.g. ThreadPoolExecutor or ScheduledThreadPoolExecutor gets these methods. To learn more about thread pools you can check this [article](http://javarevisited.blogspot.sg/2013/07/how-to-create-thread-pools-in-java-executors-framework-example-tutorial.html).  
  
**Q. What is blocking method in Java?**  
Ans: A blocking method is a method which blocks until task is done, for example accept() method of ServerSocket blocks until a client is connected. here blocking means control will not return to caller until task is finished. On the other hand there are asynchronous or non-blocking method which returns even before task is finished. To learn more about blocking method see this [answer](http://javarevisited.blogspot.sg/2012/02/what-is-blocking-methods-in-java-and.html).  
  
**Q. Is Swing thread-safe? What do you mean by Swing thread-safe?**  
Ans: You can simply this question as No, Swing is not thread-safe, but you have to explain what you mean by that even if interviewer doesn't ask about it. When we say swing is not thread-safe we usually refer its component, which cannot 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.  
  
**Q. Difference between invokeAndWait and invokeLater in Java?**  
Ans: These are two methods Swing API provides Java developers to update GUI components from threads other than Event dispatcher thread. InvokeAndWait() synchronously update GUI component, for example a progress bar, once progress is made, 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 other hand, invokeLater() is asynchronous call to update components.   
  
**Q. Which method of Swing API are thread-safe in Java?**  
Ans: This question is again related to swing and thread-safety, though components are not thread-safe there are certain method which can be safely call 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.  
  
**Q. How to create Immutable object in Java?**  
Ans: This question might not look related to multi-threading and concurrency, but it is. Immutability helps to simplify already complex concurrent code in Java. Since immutable object can be shared without any synchronization its very dear to Java developers. Core value object, which is meant to be shared among thread should be immutable for performance and simplicity. Unfortunately there is no @Immutable annotation in Java, which can make your object immutable, hard work must be done by Java developers. You need to keep basics like initializing state in constructor, no setter methods, no leaking of reference, keeping separate copy of mutable object to create Immutable object. For step by step guide see my post, [how to make an object Immutable in Java](http://javarevisited.blogspot.com/2013/03/how-to-create-immutable-class-object-java-example-tutorial.html). This will give you enough material to answer this question with confidence.  
  
**Q. What is ReadWriteLock in Java?**  
**Ans:** In general, read write lock is result of lock stripping technique to improve 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.  
  
**Q. What is busy spin in multi-threading?**  
Ans: 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 just runs empty loop. Why would someone do that? to preserve CPU caches. In multi core system, its possible for a paused thread to resume on different core, which means rebuilding cache again. To avoid cost of rebuilding cache, programmer prefer to wait for much smaller time doing busy spin. You can also see this [answer](http://java67.blogspot.com/2012/08/5-thread-interview-questions-answers-in.html) to learn more about this question.  
  
**Q. Difference between 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.  
  
**Q. What happens if a thread throws an Exception inside synchronized block?**  
Ans: This is one more tricky question for average Java programmer, if he can bring the fact about whether lock is released or not is 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 reason I like synchronized block over lock interface, which requires explicit attention to release lock, generally this is achieved by releasing lock in [finally block](http://javarevisited.blogspot.com/2012/11/difference-between-final-finally-and-finalize-java.html).  
  
**Q. What is double checked locking of Singleton?**  
**Ans:** This is one of the very popular question on Java interviews, and despite its popularity, chances of candidate answering this question satisfactory is only 50%. Half of the time, they failed to write code for double checked locking and half of the time they failed how it was broken and fixed on Java 1.5. This is actually an old way of creating thread-safe singleton, which tries to optimize performance by only locking when Singleton instance is created first time, but because of complexity and the fact it was broken for JDK 1.4,  I personally don't like it. Anyway, even if you not prefer this approach its good to know from interview point of view. Since this question deserve a detailed answer, I have answered in a separate post, you can read my post [how double checked locking on Singleton works](http://javarevisited.blogspot.sg/2014/05/double-checked-locking-on-singleton-in-java.html) to learn more about it.  
  
**Q. How to create thread-safe Singleton in Java?**  
Ans: This question is actually follow-up of previous question. If you say you don't like double checked locking then Interviewer is bound to ask about alternative ways of creating thread-safe Singleton class. There are actually man, you can take advantage of class loading and static variable initialization feature of JVM to create instance of Singleton, or you can leverage powerful enumeration type in Java to create Singleton. I actually preferred that way, you can also read this [article](http://javarevisited.blogspot.com/2012/12/how-to-create-thread-safe-singleton-in-java-example.html) to learn more about it and see some sample code.  
  
**61) List down 3 multi-threading best practice you follow?**  
Ans: This is my favourite 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. 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 more costlier. Try to avoid synchronization and locking as much as possible and at bare minimum, you should reduce critical section. That's why I prefer synchronized block over synchronized method, because it gives you absolute control on scope of locking.
* **Prefer Synchronizers over wait and notify**  
  Synchronizers like CountDownLatch, Semaphore, CyclicBarrier or Exchanger simplifies coding. It's very difficult to implement complex control flow right using wait and notify. Secondly, these classes are written and maintained by best in business and there is good chance that they are optimized or replaced by better performance code in subsequent JDK releases. By using higher level synchronization utilities, you automatically get all these benefits.
* **Prefer Concurrent Collection over Synchronized Collection**  
  This is another simple best practice which is easy to follow but reap good benefits. Concurrent collection are more scalable than their synchronized counterpart, that's why its better to use them while writing concurrent code. So next time if you need map, think about ConcurrentHashMap before thinking Hashtable. See my article [Concurrent Collections in Java](http://javarevisited.blogspot.com/2013/02/concurrent-collections-from-jdk-56-java-example-tutorial.html), to learn more about modern collection classes and how to make best use of them.

**62) How do you force start a Thread in Java?**  
Ans: This question is like how do you force garbage collection in Java, their is no way, though you can make request using System.gc() but its not guaranteed. On Java multi-threading their is absolute no way to force start a thread, this is controlled by thread scheduler and Java exposes no API to control thread schedule. This is still a random bit in Java.  
  
**63) What is fork join framework in Java?**  
Ans: The fork join framework, introduced in JDK 7 is a powerful tool available to Java developer to take advantage of multiple processors of modern day servers. It is designed for work that can be broken into smaller pieces recursively. The goal is to use all the available processing power to enhance the performance of your application. One significant advantage of The fork/join framework is that it uses a work-stealing algorithm. Worker threads that run out of things to do can steal tasks from other threads that are still busy.   
  
**64) What is difference between calling wait() and sleep() method in Java multi-threading?**  
Ans: 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](http://javarevisited.blogspot.com/2011/12/difference-between-wait-sleep-yield.html) to answer this question with more details.

**65) You have thread T1, T2 and T3, how will you ensure that thread T2 run after T1 and thread T3 run after T2?**

**Ans:** It can be achieved by using **join** method of Thread class.

package com;

import java.util.Vector;

public class ThreadTest {

private Vector<String> threadNames = new Vector<String>();

public static void main(String[] args) {

ThreadTest test = new ThreadTest();

test.threadTest(Integer.parseInt("3"));

System.out.println(test.threadNames);

}

private void threadTest(int numOfThreads) {

Thread[] threads = new Thread[numOfThreads];

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

threads[i] = new ThreadTest.MyThread();

//threads[i].start();

}

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

int i = 0;

while (i < threads.length) {

try {

if (i == 0) {

threads[i].start();

threads[i].join();

i++;

} else if (!threads[i - 1].isAlive()) {

threads[i].start();

threads[i].join();

i++;

}

} catch (InterruptedException ignore) {

}

}

}

class MyThread extends Thread {

public void run() {

for (int i = 0; i < 100000000; i++) {

i = i + 0;

}

threadNames.add(getName());

}

}

}

**66) What is the advantage of new Lock interface over synchronized block in Java? You need to implement a high performance cache which allows multiple reader but single writer to keep the integrity how will you implement it?**

**Ans:** The major advantage of lock interfaces on multi-threaded and concurrent programming is they provide two separate lock for reading and writing which enables you to write high performance data structure like [ConcurrentHashMap](http:///h) and [conditional blocking](http:///h). Its heavily used to build cache for electronic trading system on client and exchange connectivity space.

**67) What are differences between wait and sleep method in java?**

**Ans:**Only major difference is wait release the lock or monitor while sleep doesn't release any lock or monitor while waiting. Wait is used for inter-thread communication while sleep is used to introduce pause on execution. See my post [wait vs sleep in Java](http:///h) for more differences

1) wait is called from synchronized context only while sleep can be called without synchronized block. see [Why wait and notify needs to call from synchronized method](http:///h) for more detail.

2) wait is called on Object while sleep is called on Thread. see [Why wait and notify are defined in object class instead of Thread.](http:///h)

3) waiting thread can be awake by calling notify and notifyAll while sleeping thread can not be awaken by calling notify method.

4) wait is normally done on condition, Thread wait until a condition is true while sleep is just to put your thread on sleep.

5) wait release lock on object while waiting while sleep doesn’t release lock while waiting.

**68) Write code to implement blocking queue in Java?**

**Ans:**

A blocking queue is a queue that blocks when you try to dequeue from it and the queue is empty, or if you try to enqueue items to it and the queue is already full. A thread trying to dequeue from an empty queue is blocked until some other thread inserts an item into the queue. A thread trying to enqueue an item in a full queue is blocked until some other thread makes space in the queue, either by dequeuing one or more items or clearing the queue completely.

Here is a diagram showing two threads cooperating via a blocking queue:

|  |
| --- |
| A BlockingQueue with one thread putting into it, and another thread taking from it. |
| **A BlockingQueue with one thread putting into it, and another thread taking from it.** |

Java 5 comes with blocking queue implementations in the java.util.concurrent package. You can read about that class in my [java.util.concurrent.BlockingQueue](http:///h) tutorial. Even if Java 5 comes with a blocking queue implementation, it can be useful to know the theory behind their implementation.

import java.util.LinkedList;

import java.util.List;

public class BlockingQueue {

private List queue = new LinkedList();

private int limit = 10;

public BlockingQueue(int limit) {

this.limit = limit;

}

public synchronized void enqueue(Object item)

throws InterruptedException {

while (this.queue.size() == this.limit) {

wait();

}

if (this.queue.size() == 0) {

notifyAll();

}

this.queue.add(item);

}

public synchronized Object dequeue()

throws InterruptedException {

while (this.queue.size() == 0) {

wait();

}

if (this.queue.size() == this.limit) {

notifyAll();

}

return this.queue.remove(0);

}

}

Notice how notifyAll() is only called from enqueue() and dequeue() if the queue size is equal to the size bounds (0 or limit). If the queue size is not equal to either bound whenenqueue() or dequeue() is called, there can be no threads waiting to either enqueue or dequeue items.

**69) Write code to solve the Produce consumer problem in Java?**

**Ans:**

**Benefit of Producer Consumer Pattern**

**Producer Consumer Design pattern** is a classic concurrency or threading pattern which reduces coupling between Producer and Consumer by separating Identification of work with Execution of Work. In producer consumer design pattern a shared queue is used to control the flow and this separation allows you to code producer and consumer separately. It also addresses the issue of different timing require to produce item or consuming item. by using **producer consumer pattern** both Producer and Consumer Thread can work with different speed.

**Producer consumer pattern** is every where in real life and depict coordination and collaboration. Like one person is preparing food (Producer) while other one is serving food (Consumer), both will use shared table for putting food plates and taking food plates. Producer which is the person preparing food will wait if table is full and Consumer (Person who is serving food) will wait if table is empty. table is a shared object here. On Java library **Executor framework** itself implement Producer Consumer design pattern be separating responsibility of addition and execution of task.

Benefit of Producer Consumer Pattern

Its indeed a useful [design pattern](http:///h) and used most commonly while writing multi-threaded or concurrent code. here

is few of its benefit:

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

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

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

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

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

Producer Consumer Problem in Multi-threading

**Producer-Consumer Problem** is also a [popular java interview question](http:///h) where interviewer ask to implement producer consumer design pattern so that Producer should wait if Queue or bucket is full and Consumer should wait if queue or

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

**Using Blocking Queue to implement Producer Consumer Pattern**

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

import java.util.concurrent.BlockingQueue;

import java.util.concurrent.LinkedBlockingQueue;

import java.util.logging.Level;

import java.util.logging.Logger;

public class **ProducerConsumerPattern** {

public static void main(String args[]){

**//Creating shared object**

BlockingQueue sharedQueue = new LinkedBlockingQueue();

**//Creating Producer and Consumer Thread**

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

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

**//Starting producer and Consumer thread**

prodThread.start();

consThread.start();

}

}

**//Producer Class in java**

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

private final **BlockingQueue** sharedQueue;

public Producer(BlockingQueue sharedQueue) {

this.sharedQueue = sharedQueue;

}

@Override

public void run() {

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

try {

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

sharedQueue.put(i);

} catch (InterruptedException ex) {

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

}

}

}

}

**//Consumer Class in Java**

class Consumer implements Runnable{

private final BlockingQueue sharedQueue;

public Consumer (BlockingQueue sharedQueue) {

this.sharedQueue = sharedQueue;

}

@Override

public void run() {

while(true){

try {

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

} catch (InterruptedException ex) {

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

}

}

}

}

**Output:**

Produced: 0

Produced: 1

Consumed: 0

Produced: 2

Consumed: 1

Produced: 3

Consumed: 2

Produced: 4

Consumed: 3

Produced: 5

Consumed: 4

Produced: 6

Consumed: 5

Produced: 7

Consumed: 6

Produced: 8

Consumed: 7

Produced: 9

Consumed: 8

Consumed: 9

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

That’s all on **How to use Blocking Queue to solve Producer Consumer problem** or **example of Producer consumer design pattern**.

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

**Ans:** This is my favorite java thread interview question because even though deadlock is quite common while writing multi-threaded concurrent program many candidates not able to write deadlock free code and they simply struggle. Just ask them you have n resources and n thread and to complete an operation you require all resources. Here n can be replace with 2 for simplest case and higher number to make question more intimidating. see [How to avoid deadlock in java](http:///h) for more information on deadlock in Java.

**when two or more threads waiting for each other to release lock and get stuck for infinite time , situation is called deadlock . it will only happen in case of multitasking.**

**write code which will result in deadlock ?**

here is one of my version

public void method1(){

synchronized(String.class){

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

synchronized (Integer.class) {

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

}

}

}

public void method2(){

synchronized(Integer.class){

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

synchronized (String.class) {

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

}

}

}

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

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

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

public void method1(){

synchronized(Integer.class){

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

synchronized (String.class) {

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

}

}

}

public void method2(){

synchronized(Integer.class){

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

synchronized (String.class) {

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

}

}

}

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

**71) What is atomic operation? What are atomic operations in Java?**

**Ans:** Atomic means each action take place in one step without interruption or we can say that operation is performed as a single unit of work without the possibility of interference from other operations.

An Atomic operation cannot stop in the middle, either it happened completely or doesn’t happen at all.

re.

Java language specification guarantees that

* Reading or writing of a variable/reference is atomic unless the variable is of type long or double.
* Read and write are atomic for all variable declared volatile including long and double variables.

Atomic action can be used without fear of thread interference.

Using simple atomic variable access is more efficient than accessing the same variable through synchronized code. But it require more care and attention from programmer to avoid memory consistency .

Example:

Operation i++;

This operation is not atomic as it happens in 3 steps

1. Reading the current value of i
2. Increment the current value of i
3. Writing the new value of i

Since java 1.5, java language provides atomic variable e.g. AtomicInteger or AtomicLong which provides methods like getAndDecrement(),getAndIncrement() and getAndSet() in ***java.util.concurrent.atomic*** package which are all atomic.

Example:

class Counter {

private int c = 0;

public void increment() {

c++;

}

public void decrement() {

c--;

}

public int value() {

return c;

}

}

Counter class is designed so that invocation of increment () add 1 to the variable c and decrement () subtract 1 from c.

If Counter object is reference from multiple threads, interference b/w threads may prevent from happening as expected.

Suppose 2 thread are accessing the Counter object

1. Thread A: Retrieve c.
2. Thread B: Retrieve c.
3. Thread A: Increment retrieved value; result is 1.
4. Thread B: Decrement retrieved value; result is -1.
5. Thread A: Store result in c; c is now 1.
6. Thread B: Store result in c; c is now -1.

Thread A result is lost and overwritten by Thread B.

Under different circumstances, it might be Thread B result is lost and overwritten by Thread B, or there could be no error.

**72) What is volatile keyword in Java? How to use it? How is it different from synchronized method in Java?**

**Ans:** Volatile keyword in Java is used as an indicator to Java compiler and [Thread](http:///h) that do not cache value of this variable and always read it from [main memory](http:///h). So if you want to share any variable in which read and write operation is atomic by implementation e.g. read and write in int or boolean variable you can declare them as volatile variable. From Java 5 along with major changes like [Autoboxing](http:///h), [Enum](http:///h), Generics and [Variable arguments](http:///h) , Java introduces some change in Java Memory Model (JMM), Which guarantees visibility of changes made by one thread to another also as "happens-before" which solves the problem of memory writes that happen in one thread can "leak through" and be seen by another thread. [Javavolatile keyword](http:///h) cannot be used with method or class and it can only be used with variable. Java volatile keyword also guarantees visibility and ordering , after Java 5 write to any volatile variable happens before any read into volatile variable. By the way use of volatile keyword also prevents compiler or JVM from reordering of code or moving away them from [synchronization barrier](http:///h).

**Example of volatile keyword in Java:**

To Understand example of volatile keyword in java let’s go back to [Singleton pattern in Java](http:///h) and see [double checked locking in Singleton](http:///h) with Volatile and without volatile keyword in java.

/\*\*

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

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

\* every thread see updated value for \_instance.

\*

\* @author Javin Paul

\*/

**public** **class** Singleton{

**private** **static** **volatile** Singleton \_instance; *//volatile variable*

**public** **static** Singleton getInstance(){

if(\_instance == **null**){

**synchronized**(Singleton.**class**){

if(\_instance == **null**)

\_instance = **new** Singleton();

}

}

**return** \_instance;

}

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

1) We are only creating instance one time

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

If we do not make \_instance variable volatile then Thread which is creating [instance of Singleton](http:///h) 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:///h).

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

So in Summary apart from [synchronized keyword in java](http:///h), volatile keyword is also used to communicate content of memory between threads.

**Let’s see another example of volatile keyword in Java:**

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

private **boolean** bExit;

while(!bExit) {

checkUserPosition();

updateUserPosition();

}

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

1) You can use Volatile variable if you want to read and write long and [double](http:///h) variable atomically. long and double both are [64 bit](http:///h) data type and by default writing of long and double is not atomic and platform dependence. Many platform perform write in long and double variable 2 step, writing 32 bit in each step, due to this its possible for a Thread to see 32 bit from two different write. You can avoid this issue by making long and double variable volatile in Java.

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

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

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

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

while(isActive){

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

}

}

without volatile modifier its not guaranteed that one [Thread](http:///h) see the updated value of isActive from other thread. compiler is also free to cache value of isActive instead of reading it from main memory in every iteration. By making isActive a volatile variable you avoid these issue.

4) Another place where volatile variable can be used is to fixing double checked locking in Singleton pattern. As we discussed in [Why should you use Enum as Singleton](http:///h) that double checked locking was broken in Java 1.4 environment.

**Important points on Volatile keyword in Java**

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

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

3. In Java reads and writes are [atomic](http:///h) for all variables declared using **Java volatile keyword** (including long and double variables).

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

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

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

7. An access to a volatile variable in Java never has chance to block, since we are only doing a simple read or write, so unlike a synchronized block we will never hold on to any lock or wait for any [lock](http:///h).

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

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

10. If a variable is not shared between [multiple threads](http:///h) no need to use volatile keyword with that variable.

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

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

1. Volatile keyword in java is a field modifier, while [synchronized modifies code blocks and methods](http:///h).

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

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

4. [Synchronized method affects performance](http:///h) more than volatile keyword in Java.

5. Since volatile keyword in Java only synchronizes the value of one variable between Thread memory and "main" memory while synchronized synchronizes the value of all variable between thread memory and "main" memory and locks and releases a monitor to boot. Due to this reason [synchronized keyword in Java](http:///h) is likely to have more overhead than volatile.

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

7. From Java 5 Writing into a volatile field has the same memory effect as a monitor release, and reading from a volatile field has the same memory effect as a monitor acquire

In Summary ***volatile keyword in Java*** is not a replacement of synchronized block or method but in some situation is very handy and can save performance overhead which comes with [use of synchronization in Java](http:///h)

**73) What is race condition? How will you find and solve race condition?**

**Ans:** [Race condition in Java](http:///h)

**Race condition in Java** is a type of concurrency bug or issue which is introduced in your program because parallel execution of your program by multiple threads at same time, Since Java is a multi-threaded programming language hence risk of Race condition is higher in Java which demands clear understanding of what causes a race condition and how to avoid that. Anyway Race conditions are just one of hazards or risk presented by use of multi-threading in Java just like [deadlock in Java](http:///h). **Race conditions** occurs when two thread operate on same object without proper synchronization and there operation interleaves on each other. Classical **example of Race condition** is incrementing a counter since increment is not an atomic operation and can be further divided into three steps like read, update and write. if two [threads](http:///h) tries to increment count at same time and if they read same value because of interleaving of read operation of one thread to update operation of another thread, one count will be lost when one thread overwrite increment done by other thread. atomic operations are notsubject to race conditions because those operation cannot be interleaved.

Code Example of Race Condition in Java

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

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

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

public Singleton getInstance(){

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

\_instance = new Singleton();

}

}

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

**read-modify-update race conditions**

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

if(!hashtable.contains(key)){

hashtable.put(key,value);

}

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

**74) How will you take thread dump in Java? How will you analyze Thread dump?**

**Ans:** In UNIX you can use **kill -3** and then thread dump will print on log on windows you can use **"CTRL+Break".**it. Thread dump can be useful to analyze deadlock situations as well.

**75) Why we call start() method which in turns calls run() method, why not we directly call run() method ?**

**Ans:** When you call start() method it creates new Thread and execute code declared in run() while directly calling run() method doesn’t create any new thread and execute code on same calling thred

**76) How will you awake a blocked thread in java?**

**Ans:** This is tricky question on threading, blocking can result on many ways, if thread is blocked on IO then I don't think there is a way to interrupt the thread and if thread is blocked due to result of calling wait(), sleep() or join() method you can interrupt the thread and it will awake by throwing InterruptedException.

**77) What is difference between CyclicBarriar and CountdownLatch in Java ?**

**Ans:** One difference is that you can reuse CyclicBarrier once barrier is broken but you can not reuse ContdownLatch.

As per java docs, [CountDownLatch](http:///h) is a synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads completes.

simple example of CyclicBarrier in Java on which we initialize CyclicBarrier with 3 parties, means in order to cross barrier, 3 thread needs to call await() method. each thread calls await method in short duration but they don't proceed until all 3 threads reached barrier, once all thread reach barrier, barrier gets broker and each [thread](http:///h) started there execution from that point. Its much clear with the output of following example of CyclicBarrier in Java:

**import** java.util.concurrent.BrokenBarrierException;

**import** java.util.concurrent.CyclicBarrier;

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

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

/\*\*

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

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

\*

\* @author Javin Paul

\*/

**public** **class** CyclicBarrierExample {

*//Runnable task for each thread*

**private** **static** **class** Task **implements** [**Runnable**](http:///h) {

**private** **CyclicBarrier** barrier;

**public** Task(**CyclicBarrier** barrier) {

**this**.barrier = barrier;

}

@**Override**

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

**try** {

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

barrier.await();

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

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

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

} **catch** (**BrokenBarrierException** ex) {

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

}

}

}

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

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

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

@**Override**

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

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

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

}

});

*//starting each of thread*

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

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

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

t1.start();

t2.start();

t3.start();

}

}

**Output:**

**Thread** 1 is waiting on barrier

**Thread** 3 is waiting on barrier

**Thread** 2 is waiting on barrier

All parties are arrived at barrier, lets play

**Thread** 3 has crossed the barrier

**Thread** 1 has crossed the barrier

**Thread** 2 has crossed the barrier

**When to use CyclicBarrier in Java**

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

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

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

**Important point of CyclicBarrier in Java**

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

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

3. [Thread will block](http:///h) on await() until all parties reaches to barrier, another thread interrupt or await timed out.

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

java.util.concurrent.**BrokenBarrierException**

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

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

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

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

**78) What is immutable object? How does it help on writing concurrent application?**

**Ans:**

An object is considered immutable if its state cannot change after it is constructed. Maximum reliance on immutable objects is widely accepted as a sound strategy for creating simple, reliable code. Immutable objects are particularly useful in concurrent applications. Since they cannot change state, they cannot be corrupted by thread interference or observed in an inconsistent state. Examples of immutable objects from the JDK include String and Integer. Immutable objects greatly simplify your multi threaded program, since they are

Simple to construct, test, and use.

Automatically thread-safe and have no synchronization issues.

To create a object immutable You need to make the class final and all its member final so that once objects gets crated no one can modify its state. You can achieve same functionality by making member as non final but private and not modifying them except in constructor.

**79) What are some common problems you have faced in multi-threading environment? How did you resolve it?**

**Ans:** Memory-interference, race conditions, [deadlock](http:///h), live lock and starvation are example of some problems comes in multi-threading and concurrent programming. There is no end of problem if you get it wrong and they will be hard to detect and debug. This is mostly experienced based interview question on java thread instead of fact based.

**80) Deadlock Prevention**

**Ans:** In some situations it is possible to prevent deadlocks. I'll describe three techniques in this text:

1. [Lock Ordering](http://tutorials.jenkov.com/java-concurrency/deadlock-prevention.html#/h)
2. [Lock Timeout](http://tutorials.jenkov.com/java-concurrency/deadlock-prevention.html#/h)
3. [Deadlock Detection](http://tutorials.jenkov.com/java-concurrency/deadlock-prevention.html#/h)

**Lock Ordering**

* Deadlock occurs when multiple threads need the same locks but obtain them in different order.
* If you make sure that all locks are always taken in the same order by any thread, deadlocks cannot occur. Look at this example:

**Thread 1:**

lock A

lock B

**Thread 2:**

wait for A

lock C (when A locked)

**Thread 3:**

wait for A

wait for B

wait for C

If a thread, like Thread 3, needs several locks, it must take them in the decided order. It cannot take a lock later in the sequence until it has obtained the earlier locks.

For instance, neither Thread 2 or Thread 3 can lock C until they have locked A first. Since Thread 1 holds lock A, Thread 2 and 3 must first wait until lock A is unlocked. Then they must succeed in locking A, before they can attempt to lock B or C.

Lock ordering is a simple yet effective deadlock prevention mechanism. However, it can only be used if you know about all locks needed ahead of taking any of the locks. This is not always the case.

Q: How many types of synchronization?

Ans: There are two types of synchronization

1. Process Synchronization

2. Thread Synchronization Here, we will discuss only thread synchronization.

Q: What is thread synchronization?

Ans: There are two types of thread synchronization mutual exclusive and inter-thread communication.

● Mutual Exclusive

1. Synchronized method.
2. Synchronized block.
3. static synchronization.

● Cooperation (Inter-thread communication)

**Q4. Can a lock be acquired on a class?**

A. Yes, a lock can be acquired on a class. This lock is acquired on the class's Class object.

**Q5. What's new with the stop(), suspend() and resume() methods in JDK 1.2?**

A. The stop(), suspend() and resume() methods have been deprecated in JDK 1.2.

**Q. What state does a thread enter when it terminates its processing?**

A. When a thread terminates its processing, it enters the dead state.

**Q16. What is the difference between yielding and sleeping?**

A. When a task invokes its yield() method, it returns to the ready state. When a task invokes its sleep()

method, it returns to the waiting state.

**Q. What is the difference between preemptive scheduling and time slicing?**

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

next, based on priority and other factors.

**Q. When a thread blocks on I/O, what state does it enter?**

A. A thread enters the waiting state when it blocks on I/O.

**Q33. What is a task's priority and how is it used in scheduling?**

A. A task's priority is an integer value that identifies the relative order in which it should be executed with

respect to other tasks. The scheduler attempts to schedule higher priority tasks before lower priority tasks.

**Q34. When a thread is created and started, what is its initial state?**

A. A thread is in the ready state after it has been created and started.

**Q52. What is the purpose of the wait(), notify(), and notifyAll() methods?**

A. The wait(),notify(), and notifyAll() methods are used to provide an efficient way for threads to wait for a

shared resource. When a thread executes an object's wait() method, it enters the waiting state. It only enters

the ready state after another thread invokes the object's notify() or notifyAll() methods.

**Q55. What are the high-level thread states?**

A. The high-level thread states are ready, running, waiting, and dead.

**Q59. What is an object's lock and which object's have locks?**

A. An object's lock is a mechanism that is used by multiple threads to obtain synchronized access to the object.

A thread may execute a synchronized method of an object only after it has acquired the object's lock. All

objects and classes have locks. A class's lock is acquired on the class's Class object.

**Q66. What happens when a thread cannot acquire a lock on an object?**

A. If a thread attempts to execute a synchronized method or synchronized statement and is unable to acquire an object's lock, it enters the waiting state until the lock becomes available.

**Q83. How does multithreading take place on a computer with a single CPU?**

A. The operating system's task scheduler allocates execution time to multiple tasks. By quickly switching

between executing tasks, it creates the impression that tasks execute sequentially.

**Q89. What happens when you invoke a thread's interrupt method while it is sleeping or waiting?**

When a task's interrupt() method is executed, the task enters the ready state. The next time the task enters

the running state, an InterruptedException is thrown.