1. **What do you mean by multithreaded program?**
2. Multithreading in java is a process of executing multiple threads simultaneously.

For simple application like HelloWorld application, the code runs in the main thread.

So, a java code needs at least one thread to run.

It is not physically possible to run more than one piece of code on a single CPU core at any one time, it is job of JVM to manage these threads and schedule which thread to run and when.

Java Multithreading is mostly used in games, animation, request-response based applications etc.

1. **Give some multithreaded program example?**
2. **What is better? Single threading or multithreading?**
3. **Multithreading Benefits**

* Improved responsiveness — Users usually report improved responsiveness compared to single thread applications.
* Faster applications — You can perform many operations together so it saves time.
* Prioritization — Threads can be assigned a priority which would allow higher priority tasks to take precedence over lower priority tasks.
* It doesn't block the user because threads are independent and you can perform multiple operations at same time.
* Threads are independent so it doesn't affect other threads if exception occur in a single thread.

**Single Threading Benefits**

* Programming and debugging —These activities are easier compared to multithreaded applications due to the reduced complexity
* Less Overhead — Threads add overhead to an application

**When developing multi-threaded applications, the following must be considered.**

* Deadlocks occur when two threads hold a monitor that the other one requires. In essence each task is blocking the other and both tasks are waiting for the other monitor to be released. This forces an application to hang or deadlock.
* Resource allocation is used to prevent deadlocks because the system determines if approving the resource request will render the system in an unsafe state. An unsafe state could result in a deadlock. The system only approves requests that will lead to safe states.
* Thread Synchronization is used when multiple threads use the same instance of an object. The threads accessing the object can then be locked and then synchronized so that each task can interact with the static object on at a time.

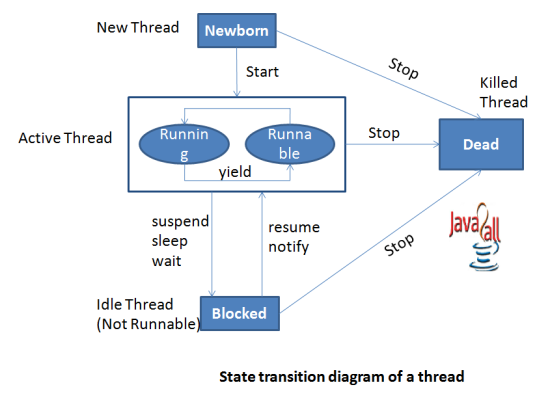
1. **How thread can be prioritized?**
2. **What are the best practices of muti-threaded programming?**

[**http://javarevisited.blogspot.com/2015/05/top-10-java-multithreading-and.html**](http://javarevisited.blogspot.com/2015/05/top-10-java-multithreading-and.html)

1. **How can a dead thread be restarted?**
   1. A dead thread cannot be restarted.

When the execution of run() method is over, as the job it is meant is done, it is brought to dead state. It is done implicitly by JVM. In dead state, the thread object is garbage collected. It is the end of the life cycle of thread. Once a thread is removed, it cannot be restarted again (as the thread object does not exist). This state can be compared with destroy() method of applets.

A thread can be killed and brought to dead state, anytime from any state, by calling explicitly stop() (deprecated) method.



- See more at: <http://way2java.com/multithreading/life-cycle-of-thread/#sthash.RRP5ZbH8.dpuf>

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

.

### 1) New

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

### 2) Runnable

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

### 3) Running

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

### 4) Non-Runnable (Blocked)

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

### Thread can be brought to blocked states by below mentioned way:

### sleep()

### wait()

### join()

### When a thread attempts to read data not yet available from a network

### When a thread waits to acquire a lock, it is in the blocked state.

### When a blocked thread unblocks, that thread moves to the runnable state.

### 5) Terminated

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

1. **What are the ways in which a thread can enter the waiting state?**
   1. A thread can enter the waiting state by :

* A running thread may enter a blocked/waiting state by a wait(), sleep(), or join() call.
* A running thread may enter a blocked/waiting state because it can't acquire the lock for a synchronized block of code.
* It can also enter the waiting state by invoking its (deprecated) suspend() method.
* When a thread attempts to read data not yet available from a network

When the sleep or wait is over, or an object's lock becomes available, the thread can only reenter the runnable state. It will go directly from waiting to running (well, for all practical purposes anyway).

1. **What are the two ways in which Thread can be created?**
2. Thread can be created by either overriding run() method of Thread class or implementing Runnable interface and then calling start() method on thread

* ***Provide a Runnable object.*** The [Runnable](https://docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html) interface defines a single method run, meant to contain the code executed in the thread. The Runnable object is passed to the Thread constructor, as in the [HelloRunnable](https://docs.oracle.com/javase/tutorial/essential/concurrency/examples/HelloRunnable.java) example:

public class HelloRunnable implements Runnable {

public void run() {

System.out.println("Hello from a thread!");

}

public static void main(String args[]) {

(new Thread(new HelloRunnable())).start();

}

}

* ***Subclass Thread.***The Thread class itself implements Runnable, though its run method does nothing. An application can subclass Thread, providing its own implementation of run, as in the [HelloThread](https://docs.oracle.com/javase/tutorial/essential/concurrency/examples/HelloThread.java" \t "_blank)example:

public class HelloThread extends Thread {

public void run() {

System.out.println("Hello from a thread!");

}

public static void main(String args[]) {

(new HelloThread()).start();

}

}

The Java Virtual Machine calls the run method of this thread.

The result is that two threads are running concurrently: the current thread (which returns from the call to the start method) and the other thread (which executes its run method).

It is never legal to start a thread more than once. In particular, a thread may not be restarted once it has completed execution.

1. **Which one is better to implement thread in Java extending Thread class or implementing Runnable?**

A.

1. [Java doesn't support multiple inheritance](http://javarevisited.blogspot.com/2011/07/why-multiple-inheritances-are-not.html), which means you can only extend one class in Java so once you extended Thread class you lost your chance and cannot extend or inherit another [class in Java](http://javarevisited.blogspot.com/2011/10/class-in-java-programming-general.html).
2. In Object oriented programming extending a class generally means adding new functionality, modifying or improving behaviors. If we are not making any modification on Thread than use Runnable interface instead.
3. Runnable interface represent a Task which can be executed by either plain Thread or Executors or any other means. So logical separation of Task as Runnable than Thread is good design decision.
4. Separating task as Runnable means we can reuse the task and also has liberty to execute it from different means. Since you cannot restart a Thread once it completes. again Runnable vs Thread for task, Runnable is winner.
5. Java designer recognizes this and that's why Executors accept Runnable as Task and they have worker thread which executes those task.
6. Inheriting all Thread methods are additional overhead just for representing a Task which can be done easily with Runnable.

Read more: <http://javarevisited.blogspot.com/2012/01/difference-thread-vs-runnable-interface.html#ixzz48AlxJzDP>

1. **What is difference between Callable interface vs Runnable interface in Java?**

**Callable interface**

***public interface Callable<V>,*** where V is the return type of the method call. This interface has a single method 'call', which needs to be defined by all the classes which implement this interface. This method takes no arguments and returns a result of type V. This method can throw checked exceptions as well.

**Runnable interface**

***public interface Runnable*** - this interface is implemented by those classes whose instances are supposed to be executed in a different thread. This interface has only one method 'run', which takes no arguments and obviously all the classes implementing this interface need to define this method.

This interface is implemented by the Thread class as well and it's a common protocol for all the objects who wish to execute in a different thread. It's one of the ways of creating threads in Java. The other way to create a thread is by sub classing the Thread class. A class implementing Runnable interface can simply pass itself to create a Thread instance and can run thereafter. This eliminates the need of sub classing the Thread class for the purpose of executing the code in a separate thread.

As long as we don't wish to override other methods of the Thread class, it may be a better idea to implement the Runnable interface to enable multithreading capabilities to a class than enabling the same by extending the Thread class.

**Callable vs Runnable**

Though both the interfaces are implemented by the classes who wish to execute in a different thread of execution, but there are few differences between the two interface which are:-

* A Callable<V> instance returns a result of type V, whereas a Runnable instance doesn't
* A Callable<V> instance may throw checked exceptions, whereas a Runnable instance can't

The designers of Java felt a need of extending the capabilities of the Runnable interface, but they didn't want to affect the uses of the Runnable interface and probably that was the reason why they went for having a separate interface named Callable in Java 1.5 than changing the already existing Runnable interface which has been a part of Java since Java 1.0.

1. **What are wait(), notify() and notifyAll() methods?**

The Object class in Java has three final methods that allow threads to communicate about the locked status of a resource. These are :

1. **wait() :** It tells the calling thread to give up the lock and go to sleep until some other thread enters the same monitor and calls notify(). The wait() method releases the lock prior to waiting and reacquires the lock prior to returning from the wait() method. The wait() method is actually tightly integrated with the synchronization lock, using a feature not available directly from the synchronization mechanism. In other words, it is not possible for us to implement the wait() method purely in Java: it is a native method.

General syntax for calling wait() method is like this:

|  |
| --- |
| synchronized( lockObject )  {      while( ! condition )      {          lockObject.wait();      }        //take the action here;  } |

1. notify() : It wakes up one single thread that called wait() on the same object. It should be noted that calling notify() does not actually give up a lock on a resource. It tells a waiting thread that that thread can wake up. However, the lock is not actually given up until the notifier’s synchronized block has completed. So, if a notifier calls notify() on a resource but the notifier still needs to perform 10 seconds of actions on the resource within its synchronized block, the thread that had been waiting will need to wait at least another additional 10 seconds for the notifier to release the lock on the object, even though notify() had been called.

General syntax for calling notify() method is like this:

|  |
| --- |
| synchronized(lockObject)  {      //establish\_the\_condition;        lockObject.notify();        //any additional code if needed  } |

1. notifyAll() : It wakes up all the threads that called wait() on the same object. The highest priority thread will run first in most of the situation, though not guaranteed. Other things are same as notify() method above.

General syntax for calling notify() method is like this:

|  |
| --- |
| synchronized(lockObject)  {      establish\_the\_condition;        lockObject.notifyAll();  } |

In general, a thread that uses the wait() method confirms that a condition does not exist (typically by checking a variable) and then calls the wait() method. When another thread establishes the condition (typically by setting the same variable), it calls the notify() method. The wait-and-notify mechanism does not specify what the specific condition/ variable value is. It is on developer’s hand to specify the condition to be checked before calling wait() or notify().

[**http://howtodoinjava.com/core-java/multi-threading/how-to-work-with-wait-notify-and-notifyall-in-java/**](http://howtodoinjava.com/core-java/multi-threading/how-to-work-with-wait-notify-and-notifyall-in-java/)

1. **What's the difference between the methods sleep() and wait()?**

|  |  |  |
| --- | --- | --- |
|  | Wait | Sleep |
| Definition | It is a method **in Object class**. It makes the current thread into the "Not Runnable" state. Wait is called on an object, not a thread. Before calling wait() method, the object should be synchronized, means the object should be inside synchronized block. The **call to wait() releases the acquired lock.** | It is a static method **in Thread class**. It makes the current thread into the "Not Runnable" state for specified amount of time. During this time, the **thread keeps the lock (monitors) it has acquired.** |
| Synchonized | wait **should be called from synchronized context** i.e. from block or method, If you do not call it using synchronized context, it will throw IllegalMonitorStateException | It **need not be called from synchronized block or methods** |
| Calls on | wait method **operates on Object** and defined in Object class | Sleep method **operates on current thread** and is in java.lang.Thread |
| Release of lock | wait release lock of object on which it is called and also other locks if it holds any | Sleep method does not release lock at all |
| Wake up condition | until call notify() or notifyAll() from Object class | Until time expires or calls interrupt() |
| static | wait is non static method | sleep is static method |

<http://www.logicaltrinkets.com/wordpress/?p=153>

<http://howtodoinjava.com/2013/03/08/difference-between-sleep-and-wait/>

<http://www.programcreek.com/2009/02/notify-and-wait-example/>

<http://www.tutorialspoint.com/java/lang/thread_sleep_millis.htm>

**public** **class** ThreadA {

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

ThreadB b = **new** ThreadB();

b.start();

**synchronized**(b){

**try**{

System.out.println("Waiting for b to complete...");

b.wait();

}**catch**(InterruptedException e){

e.printStackTrace();

}

System.out.println("Total is: " + b.total);

}

}

}

**class** ThreadB **extends** Thread{

**int** total;

@Override

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

**synchronized**(**this**){

**for**(**int** i=0; i<100 ; i++){

total += i;

}

notify();

}

}

}

1. **Where to use wait and sleep method in Java**

By reading properties and behavior of wait and sleep method it's clear that wait() method should be used in conjunction with notify() or notifyAll() method and intended for communication between two threads in Java while Thread.sleep() method is a utility method to introduce short pauses during program or thread execution. Given the requirement of synchronization for wait, it should not be used just to introduce pause or sleep in Java.

1. **Can we awake a sleeping or waiting thread in Java?**

**Call on:**

wait(): Call on an object; current thread must synchronize on the lock object.

sleep(): Call on a Thread; always currently executing thread.

**Synchronized:**

wait(): when synchronized multiple threads access same Object one by one.

sleep(): when synchronized multiple threads wait for sleep over of sleeping thread.

**Hold lock:**

wait(): release the lock for other objects to have chance to execute.

sleep(): keep lock for at least t times if timeout specified or somebody interrupt.

**Wake-up condition:**

wait(): until call notify(), notifyAll() from object

sleep(): until at least time expire or call interrupt().

**Usage:**

sleep(): for time-synchronization and;

wait(): for multi-thread-synchronization.

1. **What difference is between start and run method in Java Thread?**  
   Main difference is that when program calls start() method a new Thread is created and code inside run() method is executed in new Thread

While if you call run() method directly no new Thread is created and code inside run() will execute on current Thread.

If you want to perform time consuming task than always call start() method otherwise your [main thread](http://javarevisited.blogspot.com/2011/12/main-public-static-java-void-method-why.html) will stuck while performing time consuming task if you call run() method directly.

Another difference between start vs run in Java thread is that you cannot call start() method twice on thread object. once started, second call of start() will throw IllegalStateException in Java while you can call run() method twice.

1. **How Object level Locking is different from Class level Locking in Java?**

<http://howtodoinjava.com/core-java/multi-threading/thread-synchronization-object-level-locking-and-class-level-locking/>

1. **Explain main thread under Thread class execution?**
2. The main thread is created automatically and it begins to execute immediately when a program starts. It is a thread from which all other child threads originate.
3. **What is synchronization?**
4. Synchronization is the capability to control the access of multiple threads to shared resources. Synchronized keyword in java provides locking which ensures mutual exclusive access of shared resource and prevent data race.
5. **What are synchronized methods and synchronized statements?**

**A:** Synchronized methods are methods that are used to control access to an object. A synchronized statement can only be executed after a thread has acquired the lock for the object or class referenced in the synchronized statement.

To make a method synchronized, simply add the synchronized keyword to its declaration:

public class SynchronizedCounter {

private int c = 0;

public synchronized void increment() {

c++;

}

public synchronized void decrement() {

c--;

}

public synchronized int value() {

return c;

}

}

Another way to create synchronized code is with synchronized statements. Unlike synchronized methods, synchronized statements must specify the object that provides the intrinsic lock:

public void addName(String name) {

synchronized(this) {

lastName = name;

nameCount++;

}

nameList.add(name);

}

<https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html>

1. **What do you mean by synchronized Non Access Modifier?**
2. Java provides these modifiers for providing functionalities other than Access Modifiers, synchronized used to indicate that a method can be accessed by only one thread at a time.

* **The synchronized modifier applies only to methods and code blocks.**
* synchronized methods can have any access control and can also be marked final.
* Instance variables can't be abstract, synchronized, native, or strictfp.
* **~~The StringBuffer's API is the same as the new StringBuilder's API, except that StringBuilder's methods are not synchronized for thread safety.~~**

# ~~Vector: It's like a slower ArrayList, but it has synchronized methods.~~

# ~~Hashtable: Like a slower HashMap (as with Vector, due to its synchronized methods). No null values or null keys allowed.~~

* A running thread may enter a blocked/waiting state because it can't acquire the lock for a synchronized block of code.

# Synchronized methods prevent more than one thread from accessing an object's critical method code simultaneously.

# You can use the synchronized keyword as a method modifier, or to start a synchronized block of code.

# While only one thread can be accessing synchronized code of a particular instance, multiple threads can still access the same object's unsynchronized code.

# Static methods can be synchronized, using the lock from the java.lang.Class instance representing that class.

<http://www.javatpoint.com/static-synchronization-example>

1. **What invokes a thread's run() method?**
2. After a thread is started, via its start() method of the Thread class, the JVM invokes the thread's run() method when the thread is initially executed.

So what happens after you call start().The good stuff:

1. A new thread of execution starts (with a new call stack).
2. The thread moves from the new state to the runnable state.
3. When the thread gets a chance to execute, its target run() method will run.

Be sure you remember the following: You start a Thread, not a Runnable. You call start() on a Thread instance, not on a Runnable instance. The following example demonstrates what we've covered so far—defining, instantiating, and starting a thread:

class FooRunnable implements Runnable {

public void run() {

for(int x = 1; x < 6; x++) {

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

}

}

}

public class TestThreads {

public static void main (String [] args) {

FooRunnable r = new FooRunnable();

Thread t = new Thread(r);

t.start();

}

}

So if you see code that calls the run() method on a Runnable (or even on a Thread instance), that’s perfectly legal. But it doesn’t mean the run() method will run in a separate thread! Calling a run() method directly just means you’re invoking a method from whatever thread is currently executing, and the run() method goes onto the current call stack rather than at the beginning of a new call stack. The following code does not start a new thread of execution:

Thread t = new Thread();

t.run(); // Legal, but does not start a new thread

1. **What is the difference between yielding and sleeping?**
2. 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.

* Sleeping is used to delay execution for a period of time, and no locks are released when a thread goes to sleep.
* A sleeping thread is guaranteed to sleep for at least the time specified in the argument to the sleep() method (unless it's interrupted), but there is no guarantee as to when the newly awakened thread will actually return to running.
* The sleep() method is a static method that sleeps the currently executing thread's state. One thread cannot tell another thread to sleep.
* The yield() method may cause a running thread to back out if there are runnable threads of the same priority. There is no guarantee that this will happen, and there is no guarantee that when the thread backs out there will be a different thread selected to run. A thread might yield and then immediately reenter the running state.

1. **Explain suspend() method under Thread class>**
2. It is used to pause or temporarily stop the execution of the thread.
3. **Explain isAlive() method under Thread class?**
4. It is used to find out whether a thread is still running or not.
5. **What is currentThread()?**
6. It is a public static method used to obtain a reference to the current thread.

package com.tutorialspoint;

import java.lang.\*;

public class ThreadDemo implements Runnable {

public void run() {

Thread t = Thread.currentThread();

// tests if this thread is alive

System.out.println("status = " + t.isAlive());

}

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

Thread t = new Thread(new ThreadDemo());

// this will call run() function

t.start();

// waits for this thread to die

t.join();

// tests if this thread is alive

System.out.println("status = " + t.isAlive());

}

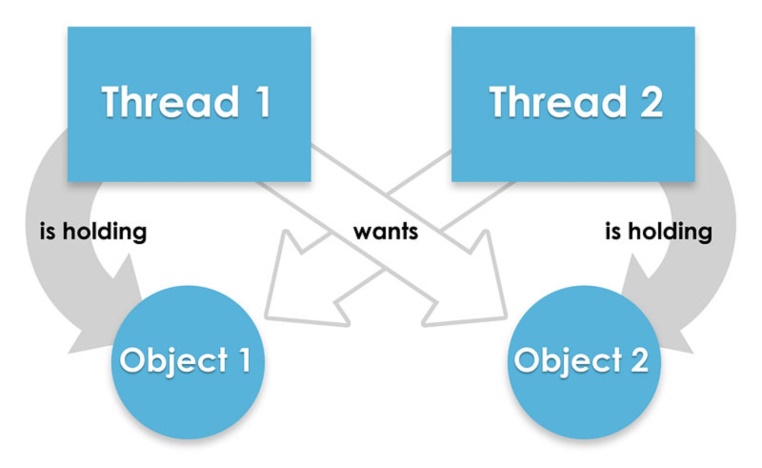
}

status = true

status = false

1. **What is deadlock?**

when two or more threads are 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.



Object A = new Object();

Object B = new Object();

Thread 1:

synchronized(A)

{ // <--- preemption

synchronized(B)

{ //...

}

}

Thread 2:

synchronized(B)

{ synchronized(A)

{ //...

}

}

Here's the deadlock scenario:

* Thread 1 acquires A, but is then preempted for some reason.
* Thread 2 wakes up, acquires B, but can't get A because Thread 1 has it, so is suspended.
* Thread 1 wakes up, tries to acquire B, but can't because Thread 2 has it, so is suspended.
* Both threads are now suspended forever. They're deadlocked.

<http://javarevisited.blogspot.com/2010/10/what-is-deadlock-in-java-how-to-fix-it.html#ixzz47pvXmnWM>

<http://www.journaldev.com/1058/java-deadlock-example-and-how-to-analyze-deadlock-situation>

1. **Write code to avoid deadlock in Java where n threads are accessing n shared resources?**

Real reason for deadlock is not multiple threads but ***the way they are requesting lock*** , if you provide an ordered access then problem will be resolved , here is my fixed version, which avoids deadlock by avoiding circular wait with no preemption.

public class DeadLockFixed {

/\*\*

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

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

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

\* in consistent order.

\*/

public void method1() {

synchronized (Integer.class) {

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

synchronized (String.class) {

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

}

}

}

public void method2() {

synchronized (Integer.class) {

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

synchronized (String.class) {

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

}

}

}

}

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

**How to Analyze Deadlock**

To analyze a deadlock, we need to look at the [java thread dump](http://www.journaldev.com/1053/how-to-generate-thread-dump-in-java) of the application, in last post I explained how we can [generate thread dump](http://www.journaldev.com/1053/how-to-generate-thread-dump-in-java) using VisualVM profiler or using jstack utility.

Here is the thread dump of above program.

2012-12-27 19:08:34

Full thread dump Java HotSpot(TM) 64-Bit Server VM (23.5-b02 mixed mode):

"Attach Listener" daemon prio=5 tid=0x00007fb0a2814000 nid=0x4007 waiting on condition [0x0000000000000000]

java.lang.Thread.State: RUNNABLE

"DestroyJavaVM" prio=5 tid=0x00007fb0a2801000 nid=0x1703 waiting on condition [0x0000000000000000]

java.lang.Thread.State: RUNNABLE

"t3" prio=5 tid=0x00007fb0a204b000 nid=0x4d07 waiting for monitor entry [0x000000015d971000]

java.lang.Thread.State: BLOCKED (on object monitor)

at com.journaldev.threads.SyncThread.run(ThreadDeadlock.java:41)

- waiting to lock <0x000000013df2f658> (a java.lang.Object)

- locked <0x000000013df2f678> (a java.lang.Object)

at java.lang.Thread.run(Thread.java:722)

"t2" prio=5 tid=0x00007fb0a1073000 nid=0x4207 waiting for monitor entry [0x000000015d209000]

java.lang.Thread.State: BLOCKED (on object monitor)

at com.journaldev.threads.SyncThread.run(ThreadDeadlock.java:41)

- waiting to lock <0x000000013df2f678> (a java.lang.Object)

- locked <0x000000013df2f668> (a java.lang.Object)

at java.lang.Thread.run(Thread.java:722)

"t1" prio=5 tid=0x00007fb0a1072000 nid=0x5503 waiting for monitor entry [0x000000015d86e000]

java.lang.Thread.State: BLOCKED (on object monitor)

at com.journaldev.threads.SyncThread.run(ThreadDeadlock.java:41)

- waiting to lock <0x000000013df2f668> (a java.lang.Object)

- locked <0x000000013df2f658> (a java.lang.Object)

at java.lang.Thread.run(Thread.java:722)

"Service Thread" daemon prio=5 tid=0x00007fb0a1038000 nid=0x5303 runnable [0x0000000000000000]

java.lang.Thread.State: RUNNABLE

"C2 CompilerThread1" daemon prio=5 tid=0x00007fb0a1037000 nid=0x5203 waiting on condition [0x0000000000000000]

java.lang.Thread.State: RUNNABLE

"C2 CompilerThread0" daemon prio=5 tid=0x00007fb0a1016000 nid=0x5103 waiting on condition [0x0000000000000000]

java.lang.Thread.State: RUNNABLE

"Signal Dispatcher" daemon prio=5 tid=0x00007fb0a4003000 nid=0x5003 runnable [0x0000000000000000]

java.lang.Thread.State: RUNNABLE

"Finalizer" daemon prio=5 tid=0x00007fb0a4800000 nid=0x3f03 in Object.wait() [0x000000015d0c0000]

java.lang.Thread.State: WAITING (on object monitor)

at java.lang.Object.wait(Native Method)

- waiting on <0x000000013de75798> (a java.lang.ref.ReferenceQueue$Lock)

at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:135)

- locked <0x000000013de75798> (a java.lang.ref.ReferenceQueue$Lock)

at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:151)

at java.lang.ref.Finalizer$FinalizerThread.run(Finalizer.java:177)

"Reference Handler" daemon prio=5 tid=0x00007fb0a4002000 nid=0x3e03 in Object.wait() [0x000000015cfbd000]

java.lang.Thread.State: WAITING (on object monitor)

at java.lang.Object.wait(Native Method)

- waiting on <0x000000013de75320> (a java.lang.ref.Reference$Lock)

at java.lang.Object.wait(Object.java:503)

at java.lang.ref.Reference$ReferenceHandler.run(Reference.java:133)

- locked <0x000000013de75320> (a java.lang.ref.Reference$Lock)

"VM Thread" prio=5 tid=0x00007fb0a2049800 nid=0x3d03 runnable

"GC task thread#0 (ParallelGC)" prio=5 tid=0x00007fb0a300d800 nid=0x3503 runnable

"GC task thread#1 (ParallelGC)" prio=5 tid=0x00007fb0a2001800 nid=0x3603 runnable

"GC task thread#2 (ParallelGC)" prio=5 tid=0x00007fb0a2003800 nid=0x3703 runnable

"GC task thread#3 (ParallelGC)" prio=5 tid=0x00007fb0a2004000 nid=0x3803 runnable

"GC task thread#4 (ParallelGC)" prio=5 tid=0x00007fb0a2005000 nid=0x3903 runnable

"GC task thread#5 (ParallelGC)" prio=5 tid=0x00007fb0a2005800 nid=0x3a03 runnable

"GC task thread#6 (ParallelGC)" prio=5 tid=0x00007fb0a2006000 nid=0x3b03 runnable

"GC task thread#7 (ParallelGC)" prio=5 tid=0x00007fb0a2006800 nid=0x3c03 runnable

"VM Periodic Task Thread" prio=5 tid=0x00007fb0a1015000 nid=0x5403 waiting on condition

JNI global references: 114

Found one Java-level deadlock:

=============================

"t3":

waiting to lock monitor 0x00007fb0a1074b08 (object 0x000000013df2f658, a java.lang.Object),

which is held by "t1"

"t1":

waiting to lock monitor 0x00007fb0a1010f08 (object 0x000000013df2f668, a java.lang.Object),

which is held by "t2"

"t2":

waiting to lock monitor 0x00007fb0a1012360 (object 0x000000013df2f678, a java.lang.Object),

which is held by "t3"

Java stack information for the threads listed above:

===================================================

"t3":

at com.journaldev.threads.SyncThread.run(ThreadDeadlock.java:41)

- waiting to lock <0x000000013df2f658> (a java.lang.Object)

- locked <0x000000013df2f678> (a java.lang.Object)

at java.lang.Thread.run(Thread.java:722)

"t1":

at com.journaldev.threads.SyncThread.run(ThreadDeadlock.java:41)

- waiting to lock <0x000000013df2f668> (a java.lang.Object)

- locked <0x000000013df2f658> (a java.lang.Object)

at java.lang.Thread.run(Thread.java:722)

"t2":

at com.journaldev.threads.SyncThread.run(ThreadDeadlock.java:41)

- waiting to lock <0x000000013df2f678> (a java.lang.Object)

- locked <0x000000013df2f668> (a java.lang.Object)

at java.lang.Thread.run(Thread.java:722)

Found 1 deadlock.

The thread dump output clearly shows the deadlock situation and threads and resources involved causing deadlock situation.

For analyzing deadlock, we need to look out for the threads with state as ***BLOCKED*** and then the resources it’s waiting to lock, every resource has a unique ID using which we can find which thread is already holding the lock on the object. For example Thread “t3” is waiting to lock 0x000000013df2f658 but it’s already locked by thread “t1”.

Once we analyze the deadlock situation and found out the threads which are causing deadlock, we need to make code changes to avoid deadlock situation.

**How to avoid deadlocks?**

* **Avoid Nested Locks**: This is the most common reason for deadlocks, avoid locking another resource if you already hold one. It’s almost impossible to get deadlock situation if you are working with only one object lock. For example, here is the another implementation of run() method without nested lock and program runs successfully without deadlock situation.

|  |
| --- |
| public void run() {      String name = Thread.currentThread().getName();      System.out.println(name + " acquiring lock on " + obj1);      synchronized (obj1) {          System.out.println(name + " acquired lock on " + obj1);          work();      }      System.out.println(name + " released lock on " + obj1);      System.out.println(name + " acquiring lock on " + obj2);      synchronized (obj2) {          System.out.println(name + " acquired lock on " + obj2);          work();      }      System.out.println(name + " released lock on " + obj2);      System.out.println(name + " finished execution.");  } |

* **Lock Only What is Required**: You should acquire lock only on the resources you have to work on, for example in above program I am locking the complete Object resource but if we are only interested in one of it’s fields, then we should lock only that specific field not complete object.
* **Avoid waiting indefinitely**: You can get deadlock if two threads are waiting for each other to finish indefinitely using [thread join](http://www.journaldev.com/1024/java-thread-join-example-with-explanation). If your thread has to wait for another thread to finish, it’s always best to use join with maximum time you want to wait for thread to finish.

Well this is another frequently asked questions on any Java thread interview. Essentially these are [two way to implement Thread in Java](http://javarevisited.blogspot.sg/2011/02/how-to-implement-thread-in-java.html) and by extending class you are using your chance to extend one any only one class as Java does not support multiple inheritance, by implementing Runnable interface you can still extend another class. So extending Runnable or even Callable is better choice. see [Runnable vs Thread class in Java](http://javarevisited.blogspot.sg/2012/01/difference-thread-vs-runnable-interface.html) for more answers on this questions. Given it's simplicity and fact based nature, this question mostly appear on either telephonic round or initial screening rounds. Key points to mention, while answering this question includes, multiple inheritance at class level and separation of defining a task and execution of task. Runnable only represent a task, while Thread represent both task and it's execution.  
  
Read more: <http://java67.blogspot.com/2012/08/5-thread-interview-questions-answers-in.html#ixzz48AhRtRsL>

<http://www.holub.com/publications/other/deadlock.html>

Read more: [http://javarevisited.blogspot.com/2010/10/what-is-deadlock-in-java-how-to-**fix**-it.html#ixzz487hbkVTw](http://javarevisited.blogspot.com/2010/10/what-is-deadlock-in-java-how-to-fix-it.html#ixzz487hbkVTw)

[**http://javarevisited.blogspot.in/2010/10/what-is-deadlock-in-java-how-to-fix-it.html**](http://javarevisited.blogspot.in/2010/10/what-is-deadlock-in-java-how-to-fix-it.html)

[**http://java67.blogspot.in/2012/08/5-thread-interview-questions-answers-in.html**](http://java67.blogspot.in/2012/08/5-thread-interview-questions-answers-in.html)

<http://java67.blogspot.com/2012/08/5-thread-interview-questions-answers-in.html#ixzz47kSGU67P>

**7) How do you solve producer consumer problem in Java?**  
**Producer consumer pattern** is everywhere 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 and used most commonly while writing multi-threaded or concurrent code. Here are 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 can be implemented or solved by different ways in Java,

1. [wait and notify method](http://javarevisited.blogspot.com/2011/05/wait-notify-and-notifyall-in-java.html)
2. BlockingQueue Data structure (JAVA 5 onwards)

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

[**https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html)

What happen if you have multiple producer or multiple consumer, what will happen if producer is faster than consumer thread or vice-versa.   
  
What are concurrent Collections?

<http://javarevisited.blogspot.in/2013/02/concurrent-collections-from-jdk-56-java-example-tutorial.html>

**8) Why ConcurrentHashMap is faster than Hashtable in Java?**  
ConcurrentHashMap is introduced as alternative of [Hashtable](http://javarevisited.blogspot.com/2012/01/java-hashtable-example-tutorial-code.html)in Java 5, it is faster because of it's design. ConcurrentHashMap divides whole map into different segments and only lock a particular segment during update operation, instead of Hashtable, which locks whole Map. ConcurrentHashMap also provides lock free read, which is not possible in Hashtable, because of this and lock striping, ConcurrentHashMap is faster than Hashtable, especially when number of reader is more than number of writers. In order to better answer this popular Java concurrency interview questions, I suggest reading my post about i[nternal working of ConcurrentHashMap in Java](http://javarevisited.blogspot.com/2013/02/concurrenthashmap-in-java-example-tutorial-working.html).  
 **9) What is difference between submit() and execute() method of Executor and ExecutorService in Java?**  
Main difference between submit and execute method from ExecutorService interface is that former return a result in form of Future object, while later doesn't return result. By the way both are used to submit task to thread pool in Java but one is defined in Executor interface,while other is added into ExecutorService interface. This multithreading interview question is also asked at first round of Java interviews.  
  
**10) How do you share data between two threads in Java?**  
One more Java multithreading question from telephonic round of interview. You can share data between thread by using shared object or shared data structures like Queue. Depending upon, what you are using, you need to provide thread-safety guarantee, and one way of providing thread-safety is using synchronized keyword. If you use concurrent collection classes from Java 5 e.g. [BlockingQueue](http://javarevisited.blogspot.com/2012/12/blocking-queue-in-java-example-ArrayBlockingQueue-LinkedBlockingQueue.html), you can easily share data without being bothered about thread safety and inter thread communication. I like this thread question, because of it's simplicity and effectiveness. This also leads further follow-up questions on issues which arises due to sharing data between threads e.g. race conditions.  
  
**11) What is ReentrantLock in Java? Have you used it before?**  
ReentrantLock is an alternative of synchronized keyword in Java, it is introduced to handle some of the limitations of synchronized keywords. Many concurrency utility classes and concurrent collection classes from Java 5, including ConcurrentHashMap uses ReentrantLock, to leverage optimization. ReentrantLock mostly uses atomic variable and faster CAS operation to provides better performance. Key points to mention is [difference between ReentrantLock and synchronized keyword in Java](http://javarevisited.blogspot.com/2013/03/reentrantlock-example-in-java-synchronized-difference-vs-lock.html), which includes ability to acquire lock interruptibly, timeout feature while waiting for lock etc. ReentrantLock also gives option to create fair lock in Java.Once again a very good Java concurrency interview question for experienced Java programmers.  
  
**12) What is ReadWriteLock in Java? What is benefit of using ReadWriteLock in Java?**  
This is usually a followup question of previous Java concurrency questions. ReadWriteLock is again based upon lock striping by providing separate lock for reading and writing operations. If you have noticed before, reading operation can be done without locking if there is no writer and that can hugely improve performance of any application. ReadWriteLock leverage this idea and provide policies to allow maximum concurrency level. Java Concurrency API also provides an implementation of this concept as ReentrantReadWriteLock. Depending upon Interviewer and experience of candidate, you can even expect to provide your own implementation of ReadWriteLock, so be prepare for that as well.  
  
These were some of my favorite*interview questions based on multithreading and concurrent in Java*. Threading and Concurrency is a big topic in Java and has lots of interesting, [tricky and tough question](http://java67.blogspot.com/2012/09/top-10-tricky-java-interview-questions-answers.html) but for starters and freshers these questions certainly help to clear any thread interview in Java. As I said, mentioning key points are very important while answering questions on multithreading and concurrency. I also suggest further reading on locking, synchronization, concurrent collections and concurrency utilities classes to do well in core Java and multithreading interviews.  
  
Read more: <http://java67.blogspot.com/2012/08/5-thread-interview-questions-answers-in.html#ixzz48ApbNPyU>

1. Write a program consists of two threads printing as mentioned below.

1-t1

2-t2

3-t1

4-t2

5-t1

1. Is servlet multithreaded?
2. How then objects are shared between threads?
3. **31. How we can stop multiple thread manipulating singleton state?**

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

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

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

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

### What do you understand about Thread Priority?

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

### What is Thread Scheduler and Time Slicing?

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

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

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

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

We can use Thread join() method to make sure all the threads created by the program is dead before finishing the main function. Here is an article about [Thread join method](http://www.journaldev.com/1024/java-thread-join-example-with-explanation).

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

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

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

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

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

There are several ways to achieve thread safety in java – synchronization, atomic concurrent classes, implementing concurrent Lock interface, using volatile keyword, using immutable classes and Thread safe classes. Learn more at [thread safety tutorial](http://www.journaldev.com/1061/java-synchronization-and-thread-safety-tutorial-with-examples).

### What is volatile keyword in Java

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

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

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

### What is ThreadLocal?

Java ThreadLocal is used to create thread-local variables. We know that all threads of an Object share it’s variables, so if the variable is not thread safe, we can use synchronization but if we want to avoid synchronization, we can use ThreadLocal variables.  
Every thread has it’s own ThreadLocal variable and they can use it’s get() and set() methods to get the default value or change it’s value local to Thread. ThreadLocal instances are typically private static fields in classes that wish to associate state with a thread. Check this post for small example program showing [ThreadLocal Example](http://www.journaldev.com/1076/java-threadlocal-example-to-create-thread-local-variables).

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

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | t1.setUncaughtExceptionHandler(new UncaughtExceptionHandler(){        @Override      public void uncaughtException(Thread t, Throwable e) {          System.out.println("exception occured:"+e.getMessage());      }    }); |

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

Thread dump is list of all the threads active in the JVM, thread dumps are very helpful in analyzing bottlenecks in the application and analyzing deadlock situations. There are many ways using which we can generate Thread dump – Using Profiler, Kill -3 command, jstack tool etc. I prefer jstack tool to generate thread dump of a program because it’s easy to use and comes with JDK installation. Since it’s a terminal based tool, we can create script to generate thread dump at regular intervals to analyze it later on. Read this post to know more about [generating thread dump in java](http://www.journaldev.com/1053/how-to-generate-thread-dump-in-java).

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

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

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

Avoid Nested Locks, Lock Only What is Required and Avoid waiting indefinitely are common ways to avoid deadlock situation, read this post to learn how to [analyze deadlock in java](http://www.journaldev.com/1058/java-deadlock-example-and-how-to-analyze-deadlock-situation) with sample program.

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

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

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

Check this post for [java Timer example](http://www.journaldev.com/1050/java-timer-and-timertask-example-tutorial).

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

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

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

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

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

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

|  |  |
| --- | --- |
| 1  2  3  4  5 | public void run() {      if (target != null) {          target.run();      }  } |

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

|  |  |
| --- | --- |
| TestThread.java | |
| 1  2  3  4  5  6  7  8  9  10  11  12 | public class TestThread extends Thread {        //not overriding Thread.run() method        //main method, can be in other class too      public static void main(String args[]){          Thread t = new TestThread();          System.out.println("Before starting thread");          t.start();          System.out.println("After starting thread");      }  } |

It will print only below output and terminate.

|  |  |
| --- | --- |
| 1  2 | Before starting thread  After starting thread |

[/sociallocker]

## Java Concurrency Interview Questions Answers

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

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

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

To solve this issue, we will have to make sure that increment operation on count is atomic, we can do that using Synchronization but Java 5 java.util.concurrent.atomic provides wrapper classes for int and long that can be used to achieve this atomically without usage of Synchronization. Go to this article to learn more about [atomic concurrent classes](http://www.journaldev.com/1095/java-atomic-operations-atomicinteger-example).

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

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

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

Read more at [**Java Lock Example**](http://www.journaldev.com/2377/java-lock-example-and-concurrency-lock-vs-synchronized).

### What is Executors Framework?

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

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

Creating a lot many threads with no bounds to the maximum threshold can cause application to run out of heap memory. So, creating a ThreadPool is a better solution as a finite number of threads can be pooled and reused. Executors framework facilitate process of creating Thread pools in java. Check out this post to learn with example code to [create thread pool using Executors framework](http://www.journaldev.com/1069/java-thread-pool-example-using-executors-and-threadpoolexecutor).

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

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

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

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

BlockingQueue interface is part of java collections framework and it’s primarily used for implementing producer consumer problem.  
Check this post for [producer-consumer problem implementation using BlockingQueue](http://www.journaldev.com/1034/java-blockingqueue-example-implementing-producer-consumer-problem).

### What is Callable and Future?

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

Callable interface use Generic to define the return type of Object. Executors class provide useful methods to execute Callable in a thread pool. Since callable tasks run in parallel, we have to wait for the returned Object. Callable tasks return java.util.concurrent.Future object. Using Future we can find out the status of the Callable task and get the returned Object. It provides get() method that can wait for the Callable to finish and then return the result.  
Check this post for [Callable Future Example](http://www.journaldev.com/1090/java-callable-future-example).

### What is FutureTask Class?

FutureTask is the base implementation class of Future interface and we can use it with Executors for asynchronous processing. Most of the time we don’t need to use FutureTask class but it comes real handy if we want to override some of the methods of Future interface and want to keep most of the base implementation. We can just extend this class and override the methods according to our requirements. Check out [**Java FutureTask Example**](http://www.journaldev.com/1650/java-futuretask-example-program) post to learn how to use it and what are different methods it has.

### What are Concurrent Collection Classes?

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

Concurrent Collection classes support full concurrency of retrievals and adjustable expected concurrency for updates.  
Major classes are ConcurrentHashMap, CopyOnWriteArrayList and CopyOnWriteArraySet, check this post to learn [how to avoid ConcurrentModificationException when using iterator](http://www.journaldev.com/378/how-to-avoid-concurrentmodificationexception-when-using-an-iterator).

### What is Executors Class?

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

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

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

Some important concurrent API enhancements are:

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

<http://www.journaldev.com/1162/java-multi-threading-concurrency-interview-questions-with-answers>