**Multi-threading**

1. **What is the priority of a Thread and how it is used in scheduling?**

In Java, every Thread has a priority. This priority is specified as a number between 1 to 10.

Scheduler in Java schedules different threads based on the priority of a thread. It is also known as pre-emptive scheduling.

The thread with higher priority gets preference in execution over a thread with lower priority.

1. **What is the default priority of a thread in Java?**

In Java, a new thread gets the same priority as the priority of the parent thread that creates it.

Default priority of a thread is 5 (NORM\_PRIORITY).

1. **What are the three different priorities that can be set on a Thread in Java?**

We can set following three priorities on a Thread object in Java:

1. MIN\_PRIORITY: This is the minimum priority that a thread can have.
2. NORM\_PRIORITY: This is the default priority that is assigned to a thread.
3. MAX\_PRIORITY: This is the maximum priority that a thread can have.

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

1. **What are the advantages of Multithreading?**

Main advantages of Multithreading are:

1. Improved performance: We can improve performance of a job by Multi-threading.
2. Simultaneous access to Multiple Applications: We can access multiple applications from a process by doing multithreading
3. Reduced number of Servers required: With Multi-threading we need lesser number of servers, since one process can spawn multiple threads.
4. Simplified Coding: In certain scenarios, it is easier to code multiple threads than managing it from same thread.
5. **What are the disadvantages of Multithreading?**

There are certain downsides to Multithreading. These are:

1. Difficult to Debug: Multithreading code is difficult to debug in case of an issue.
2. Difficult to manage concurrency: Due to multiple threads, we may experience different kinds of issues.
3. Difficulty of porting code: It is difficult to convert existing single threaded code into multi-threading code.
4. Deadlocks: In case of multi-threading we can experience deadlocks in threads that are waiting for same resource.
5. **What are the differences between Pre-emptive Scheduling Scheduler and Time Slicing Scheduler?**

In Pre-emptive scheduling, the highest priority task will keep getting time to execute until it goes to waiting state or dead state or a task with higher priority comes into queue for scheduling.

In Time slicing scheduling, every task gets a predefined slice of time for execution, and then it goes to the pool of tasks ready for execution. The scheduler picks up the next task for execution, based on priority and various other factors.

1. **How will you make a user thread into daemon thread if it has already started?**

No. We cannot make a user thread to daemon thread once it has already started.

If we do it by calling setDaemon(), it will throw IllegalThreadStateException

1. **Can we start a thread two times in Java?**

No. We can call start() method only once on a thread in Java. If we call it twice, it will give us exception.

1. **In what scenarios can we interrupt a thread?**

We can interrupt a thread if we want to wake it up from the sleep or wait state.

1. **In Java, is it possible to lock an object for exclusive use by a thread?**

Yes. We can use synchronized block to lock an object. The locked object is inaccessible to any other thread. Only the thread that has locked it can access it.

1. **How notify() method is different from notifyAll() method?**

In Java, notify() method is used to unblock a specific thread that is in waiting stated. Whereas, notifyAll() method is used to unblock all the threads that are in waiting state.

1. **What is a daemon thread in Java?**

A daemon thread in Java is a low priority thread that does not prevent the JVM from exiting when the program finishes. The thread keeps running. Garbage Collection is an example of daemon thread.

1. **How can we make a regular thread Daemon thread in Java?**

We can call setDaemon(boolean) method to change a thread to daemon thread before the thread starts.

1. **How will you make a user thread into daemon thread if it has already started?**

No. We cannot make a user thread to daemon thread once it has already started. If we do it by calling setDaemon(), it will throw IllegalThreadStateException

1. **Can we start a thread two times in Java?**

No. We can call start() method only once on a thread in Java. If we call it twice, it will give us exception.

1. **What is a Shutdown hook in Java?**

The shutdown hook is a thread that is invoked implicitly by JVM just before the shut down. It can be used to clean up unused resources etc.

We can use java.lang.Runtime.addShutdownHook(Thread hook) method to register a new virtual-machine shutdown hook.

1. **What is a Deadlock situation?**

A Deadlock is a situation in which two or more threads are waiting on each other to release a resource. Each thread is waiting for a resource that is held by the other waiting thread.

At times there is a circular wait when more than two threads are waiting on each other’s resources.

1. **What is a Scheduler?**

A scheduler is a program that is the implementation of a scheduling algorithm to manage access of processes and threads to limited resource like CPU or an I/O channel.

The goal of most scheduling algorithms is to provide load balancing for the available processes/threads and to guarantee that each process/thread will get a reasonable time frame to access the requested resource exclusively.

1. **What is the minimum number of Threads in a Java program?**

In a JVM, each Java program is executed within the main process that starts with java.exe. Therefore, each Java application has at least one thread.

1. **What are the properties of a Java thread?**

Each Java thread has following properties:

1. **Identifier**: An identifier of type long that is unique withinthe JVM
2. **Name: A name of type String**
3. **Priority: Priority of type int**
4. **State: A state of type java.lang.Thread.State**
5. **Group: A thread grou**p the thread belongs to
6. **What are the different states of a Thread in Java?**

Following are the different states of a Thread in Java:

1. **New: In the New state the thread has not yet.**
2. **Runnable: A thread executing in the JVM is in Runnable state.**
3. **Blocked: A thread waiting for a monitor lock is in Blocked state.**
4. **Waiting: A thread waiting indefinitely for another thread to perform a particular action is in Waiting state.**
5. **Timed\_waiting: A thread waiting for another thread to perform an action for up to a specified waiting time is in Timed\_waiting state.**
6. **Terminated**: A thread that has exited is in Terminatedstate.
7. **How will you set the priority of a thread in Java?**

The priority of a thread in Java can be set by using setPriority(int priority) method.

We can use constant Thread.MAX\_PRIORITY to set the maximum priority of a thread.

We can use constant Thread.MIN\_PRIORITY to set the minimum priority of a thread.

Or we can use constant Thread.NORM\_PRIORITY to set the default priority of a thread.

1. **What is the purpose of Thread Groups in Java?**

In Java, every thread belongs to a group of threads.

The JDK class java.lang.ThreadGroup provides methods to handle a whole group of Threads.

With the help of these methods we can interrupt all threads of a group or set the maximum priority of all threads of a group.

So, a thread group is used for taking collective actions on a group of threads.

1. **How do you access the current thread in a Java program?**

We can access the current thread in Java by calling the static method currentThread() of java.lang.Thread class.

Sample code is as follows:

public class MyThread {

public static void main(String[] args) {

// Get ID of Current Thread

long id = Thread.currentThread().getId();

// Get Name of Current Thread

String name = Thread.currentThread().getName();

}

}

1. **How can we prevent busy waiting in Java?**

There is a simple way to prevent busy-waiting in Java. We can just put the current thread to sleep for given amount of time.

It can be done by calling sleep() method of java.lang.Thread class. We can pass the number of milliseconds to sleep() method as an argument.

1. **Can we use Thread.sleep() method for real-time processing in Java?**

Java does not guarantee that Thread.sleep() will cause the thread to sleep for exactly N number of milliseconds. Sometime the thread can sleep for than N number of milliseconds.

In real-time processing we need precise time period for which a thread should run or sleep.

Therefore the invocation of Thread.sleep() method is not recommended for use in real-time processing.

1. **What are the two ways to check if a Thread has been interrupted?**

These are the two ways to check for thread interruption:

1. In Java, a Thread can call Thread.interrupted() method to check if it has been interrupted or not.
2. The other option is to call isInterrupted() method of Thread class to check if it has been interrupted or not.
3. **How can we make sure that Parent thread waits for termination of Child thread?**

We can use join() method for this purpose. On calling join() method, current thread waits for the child thread to which it joins to finish.

Sample code is as follows:

Thread myThread = new Thread(new Runnable() { public void run() {

}

});

myThread.start();

* Join on myThread myThread.join();

1. **How will you handle InterruptedException in Java?**

In Java we can get InterruptedException from sleep() or join() methods. Throwing InterruptedException is way to inform that another thread has interrupted this thread.

In general, the purpose of Interrupt is to ask current thread to stop its current execution and finish unexpectedly.

Therefore ignoring this exception by catching it and only logging it to the console or some log file is not the recommended approach.

The run() method of the Runnable interface does not allow that throwing any exceptions. So we cannot re-throw InterruptedException.

Therefore the correct way to handle this exception is that run() method should check and handle this exception by itself and take appropriate action.

1. **Which intrinsic lock is acquired by a synchronized method in Java?**

When we mark a method as synchronized and then call this method, then this method will first acquire the intrinsic lock of the object in which that method is mentioned.

Once the synchronized method returns, it releases the lock.

In case the synchronized method throws an exception, the intrinsic lock will be released.

Sample code equivalent to a synchronized method is:

public void myMethod() {

synchronized(this) {

}

}

1. **Can we mark a constructor as synchronized in Java?**

No. We cannot mark a constructor as synchronized.

This will lead to compiler error.

The reasoning behind this is that, in this case, only the constructing thread would have access to the object being constructed.

1. **Can we use primitive values for intrinsic locks?**

No. Java does not allow primitive values to be used for intrinsic locks.

1. **Do we have re-entrant property in intrinsic locks?**

Yes. An intrinsic lock can be accessed by the same thread multiple times. So an Intrinsic lock is re-entrant.

If it is not allowed then the code that acquires a lock would have to avoid acquiring the lock that it has already acquired.

1. **What is an atomic operation?**

An atomic operation is an operation that completes in a single step relative to other threads.

An Atomic operation is either executed completely or not at all.

There is no halfway mark in Atomic operation.

1. **Can we consider the statement i++ as an atomic operation in Java?**

No. The statement i++ is not an Atomic operation. It has more than one operation.

First JVM loads the current value of i in memory. Then it increments it. Finally it stores the new value back into variable i.

The current thread that executes this operation may be interrupted between any of the above-mentioned three steps. Therefore it is not an atomic operation.

1. **What are the Atomic operations in Java?**

Java language provides some basic Atomic operations. These operations can be used to make sure that concurrent threads always see the same value.

Some of these Atomic operations are:

1. Read operations on reference variables and primitive variables (except long and double)
2. Write operations on reference variables and primitive variables (except long and double)
3. Read operations on all variables declared as volatile
4. Write operations on all variables declared as volatile
5. **Can you check if following code is thread-safe?**

public class SingletonDoubleCheck {

private SingletonDoubleCheck instance = null;

public SingletonDoubleCheck getInstance() { if (instance == null) {

synchronized (SingletonDoubleCheck.class) { if (instance == null) {

instance = new SingletonDoubleCheck();

}

}

}

return instance;

}

}

The above-mentioned code is for creating a Singleton class. But this code is not thread-safe.

In this we check the value of instance second time in the synchronized block. But the JIT compiler can rearrange the Bytecode in such a way that the reference to SingletonDoubleCheck instance will be set before the execution of constructor.

Due to this the method getInstance() will return an object that may not have been initialized properly.

We can use the keyword volatile for instance to make this thread-safe code.

Any variables that is marked as volatile will be visible to other threads only after the completion of the constructor of the object.

1. **What are the minimum requirements for a Deadlock situation in a program?**

For a deadlock to occur following are the minimum requirements:

1. **Mutual exclusion**: There has to be a resource that can beaccessed by only one thread at any point of time.
2. **Resource holding**: One thread locks one resource andholds it, and at the same time it tries to acquire lock on another mutually exclusive resource.
3. **No preemption**: There is no pre-emption mechanism bywhich resource held by a thread can be freed after a specific period of time.
4. **Circular wait**: There can be a scenario in which two ormore threads lock one resource each and they wait for each other’s resource to get free. This causes circular wait among threads for same set of resources.
5. **How can we prevent a Deadlock?**

To prevent a Deadlock from occurring at least one requirement for a deadlock has to be removed:

1. **Mutual exclusion**: We can use optimistic locking toprevent mutual exclusion among resources.
2. **Resource holding**: A thread has to release all its exclusivelocks if it does not succeed in acquiring all exclusive locks for resources required.
3. **No preemption**: We can use timeout period for anexclusive lock to get free after a given amount of time.
4. **Circular wait**: We can check and ensure that circular waitdoes not occur, when all exclusive locks have been acquired by all the threads in the same sequence.
5. **How can we detect a Deadlock situation?**

We can use ThreadMXBean.findDeadlockedThreads() method to detect deadlocks in Java program. This bean comes with JDK:

Sample code is as follows:

ThreadMXBean bean = ManagementFactory.getThreadMXBean();

long[] threadIds = bean.findDeadlockedThreads(); // It will return

null for no deadlock

if (threadIds != null) {

ThreadInfo[] infos = bean.getThreadInfo(threadIds);

for (ThreadInfo info : infos) {

StackTraceElement[] stack = info.getStackTrace();

// Log or store stack trace information.

}

}

1. **How can a synchronized block cause Thread starvation in Java?**

It is not defined for synchronization that which thread will enter a synchronized block. It may happen that if many threads are waiting for the entry to a synchronized block, some threads may have to wait longer than other threads.

Hence these threads with lower priority will not get enough time to finish their work in time.

1. **What is a Race condition?**

A race condition is an unwanted situation in which a program attempts to perform two or more operations at the same time, but because of the logic of the program, the operations have to be performed in proper sequence to run the program correctly.

Since it is an undesirable behavior, it is considered as a bug in code.

Most of the time race condition occurs in “check then act” scenario. Both threads check and act on same value. But one of the threads acts in between check and act. See this example to understand race condition.

if (x == 3) // Check

{

y = x \* 5; // Act

* If another thread changes x
* between "if (x == 3)” and "y = x \* 5”,
* then y will not be equal to 15.

}

1. **What is a Fair lock in multi-threading?**

In Java there is a class ReentrantLock that is used for implementing Fair lock. This class accepts an optional parameter fairness. When fairness is set to true, the RenentrantLock will give access to the longest waiting thread.

The most popular use of Fair lock is in avoiding thread starvation. Since longest waiting threads are always given priority in case of contention, no thread can starve.

Downside of Fair lock is the low throughput of the program. Since low priority or slow threads are getting locks multiple time, it leads to slower execution of a program.

The only exception to a Fair lock is tryLock() method of ReentrantLock. This method does not honor the value of fairness parameter.

1. **Which two methods of Object class can be used to implement a Producer Consumer scenario?**

In a Producer Consumer scenario, one thread is a Producer and another thread is a Consumer.

For this scenario to start working, a Consumer has to know when the Producer has produced. In Object class, there is a wait() method. A Consumer calls wait method to wait on Producer. The Producer used notify() method of Object class to inform Consumer that it has produced.

In this way the processor time between produce and consume operations is freed due to the use of wait() and notify() methods.

1. **How JVM determines which thread should wake up on notify()?**

If multiple threads are waiting on an object’s monitor, JVM awakens one of them. As per Java specification the choice of this thread is arbitrary and it is at the discretion of the implementation. So there is no guarantee of rule that a specific thread will be awakened by JVM on notify() method call.

1. **Check if following code is thread-safe for retrieving an integer value from a Queue?**

public class QueueCheck {

Queue queue;

public Integer getNextInt() {

Integer retVal = null;

synchronized (queue) {

try {

while (queue.isEmpty()) {

queue.wait();

}

} catch (InterruptedException e) {

e.printStackTrace();

}

}

synchronized (queue) {

retVal = queue.poll();

if (retVal == null) {

System.err.println("retVal is null");

throw new IllegalStateException();

}

}

return retVal;

}

}

In the above code Queue is used as object monitor to handle concurrency issues. But it may not behave correctly in a multi-threading scenario.

There are two separate synchronized blocks in above code. In case two threads are woken up simultaneously by another thread, both

threads will enter one after in the second synchronized block.

Only one of the two threads will get new value from the queue and make it empty. The second thread will poll on an empty queue and it will not get any non-null return value.

1. **How can we check if a thread has a monitor lock on a given object?**

In Java, Thread class has a static method holdsLock(Object objToCheck) to check whether thread has a lock on objToLock object.

This method will return true if current thread holds the lock on the objToLock object that was passed as an argument to this method.

1. **What is an important point to consider while passing an object from one thread to another thread?**

This is a multi-threading scenario. In a multi-threading scenario, the most important point is to check whether two threads can update same object at the same time.

If it is possible for two threads to update the same object at the same time, it can cause issues like race condition.

So it is recommended to make the object Immutable. This will help in avoiding any concurrency issues on this object.

1. **What are the rules for creating Immutable Objects?**

As per Java specification, following are the rules for creating an

Immutable object:

1. Do not provide "setter" methods that modify fields or objects referred to by fields.
2. Make all fields final and private.
3. Do not allow subclasses to override methods. The simplest way to do this is to declare the class as final. A more sophisticated approach is to make the constructor private and construct instances in factory methods.
4. If the instance fields include references to mutable objects, do not allow those objects to be changed.
5. Do not provide methods that modify the mutable objects.
6. Do not share references to the mutable objects. Never store references to external, mutable objects passed to the constructor; if necessary, create copies, and store references to the copies. Similarly, create copies of your internal mutable objects when necessary to avoid returning the originals in your methods.
7. **What is the use of ThreadLocal class?**

ThreadLocal class provides thread-local variables. Each thread accesses only its own local variables. It has its own copy of the variable.

By using ThreadLocal, if thread X stores a variable with value x and another thread Y stores same variable with the value y, then X gets x from its ThreadLocal instance and Y gets y from its ThreadLocal instance.

Typically, ThreadLocal instances are private static fields that are associated with the state of a thread.

1. **What are the scenarios suitable for using ThreadLocal class?**

We can use instance of ThreadLocal class to transport information within an application.

One use case is to transport security or login information within an instance of ThreadLocal so that every method can access it.

Another use case is to transport transaction information across an application, without using the method-to-method communication.

1. **How will you improve the performance of an application by multi-threading?**

In an environment with more than one CPU, we can parallelize the computation tasks on multiple CPUs. This leads to parallel processing of a bigger task that takes lesser time due to multiple threads dividing the work among themselves.

One example is that if we have to process 1000 data files and calculate the sum of numbers in each file. If each file takes 5 minutes, then 1000 files will take 5000 minutes for processing.

But by using multi-threading we can process these files in 10 parallel threads. So each thread will take 100 files each. Since now work is happening in 10 parallel threads, the time taken will be around 500 minutes.

1. **How will you calculate the maximum speed up of an application by using multiple processors?**

Amdahl’s law gives the theoretical speedup in latency of the execution of a task at fixed workload.

It gives the formula to compute the theoretical maximum speed up that can be achieved by providing multiple processors to an application.

If S is the theoretical speedup then the formula is:

S(n) = 1 / (B + (1-B)/n)

where n is the number of processors

B is the fraction of the program that cannot be executed in parallel.

When n converges against infinity, the term (1-B)/n converges against zero. Therefore, the formula can be reduced in this special case to 1/B.

In general, the theoretical maximum speedup behaves in inverse proportion to the fraction that has to be executed serially. This means the lower this fraction is, the more theoretical speedup can be achieved.

1. **What is Lock contention in multi-threading?**

Lock contention is the situation when one thread is waiting for a lock/object that being held by another thread. The waiting thread cannot use this object until the other thread releases the lock on that object.

It is also known as Thread contention.

Ideally locks reduce the thread contention. Without locks, multiple threads can operate on same object and cause undesirable behavior. If locking is implemented correctly it reduces the occurrence of contention between multiple threads.

1. **What are the techniques to reduce Lock contention?**

There are following main techniques to reduce Lock contention:

1. Reduce the scope of lock.
2. Reduce object pooling.
3. Reduce the number of times a certain lock can be acquired.
4. Avoid synchronization at unnecessary places.
5. Implement hardware supported Optimistic locking in place of synchronization.
6. **What technique can be used in following code to reduce Lock contention?**

synchronized (map) {

Random r = new Random();

Integer value = Integer.valueOf(42);

String key = r.nextString(5);

map.put(key, value);

}

The code uses Random() to get a random string and it also used Integer to convert 42 in an object. Since these lines of code are specific to this thread, these can be moved out of Synchronization block.

Random r = new Random();

Integer value = Integer.valueOf(42);

String key = r.nextString(5);

synchronized (map) {

map.put(key, value);

}

1. **What is Lock splitting technique?**

Lock splitting is a technique to reduce Lock contention in multi-threading. It is applicable in scenario when one lock is used to synchronize access to different aspects of the same application.

Sometimes we put one lock to protect the whole array. There can be multiple threads trying to get the lock for same array. This single lock on array can cause Lock contention among threads. To resolve this we can give one lock to each element of the array. Or we can use modulus function to assign different locks to a small group of array elements. In this way we can reduced the chance of Lock contention. This is Lock splitting technique.

1. **Which technique is used in ReadWriteLock class for reducing Lock contention?**

ReadWriteLock uses two locks. One lock for read-only operations, another lock for write operations.

Its implementation is based on the premise that concurrent threads do not need a lock when they want to read a value while no other thread is trying to write.

In this implementation, read-only lock can be obtained by multiple threads. And the implementation guarantees that all read operation will see only the latest updated value as soon as the write lock is released.

1. **What is Lock striping?**

In Lock splitting we use different locks for different parts of the application. In Lock striping we use multiple locks to protect different parts of the same data structure.

ConcurrentHashMap class of Java internally uses different buckets to store its values. Each bucket is chosen based on the value of key. ConcurrentHashMap uses different locks to guard different buckets. When one thread that tries to access a hash bucket, it can acquire the lock for that bucket. While another thread can simultaneously acquire lock for another bucket and access it. In a synchronized version of HashMap, the whole map is has one lock.

Lock striping technique gives better performance than Synchronizing the whole data structure.

1. **What is a CAS operation?**

CAS is also known a Compare-And-Swap operation.

In a CAS operation, the processor provides a separate instruction that can update the value of a register only if the provided value is equal to the current value.

CAS operation can be used as an alternate to synchronization.

Let say thread T1 can update a value by passing its current value and the new value to be updated to the CAS operation. In case another thread T2 has updated the current value of previous thread, the previous thread T1’s current value is not equal to the current value of T2. Hence the update operation fails.

In this case, thread T1 will read the current value again and try to update it.

This is an example of optimistic locking.

1. **Which Java classes use CAS operation?**

Java classes like AtomicInteger or AtomicBoolean internally use CAS operations to support multi-threading.

These classes are in package java.util.concurrent.atomic.

1. **Is it always possible to improve performance by object pooling in a multi-threading application?**

By using Object pools in an application we limit the number of new objects to be created for a class. In a single thread operation, it can improve the performance by reusing an already created object from a pool.

In a multi-threading application, an object pool has to provide synchronized access to multiple threads. Due to this only one thread can access the pool at a time. Also there is additional cost due to Lock contention on pool. These additional costs can outweigh the cost saved by reuse of an object from the pool.

Therefore, using an Object pool may not always improve the performance in a multi-threading application.

1. **How can techniques used for performance improvement in a single thread application may degrade the performance in a multi-threading application?**

In a single thread applications we can use Object pool for performance optimization. Where as in multi-threading environment, it may not be a good idea to use an Object pool. Increased overhead of synchronization and lock contention can degrade the performance gained by using Object pool in a multi-threading application.

Another example is the implementation in which a List keeps a separate variable to hold the number of elements. This technique is useful in single thread application where size() method can return the value from this variable, without the need to count all the elements of list.

But in a multi-threading application, this separate variable can rather degrade the performance. This variable has to be access controlled by a lock since multiple concurrent threads can insert an element in a list. The additional cost of lock on this variable can outweigh the benefit gained by it in a multi-threading application.

1. **What is the relation between Executor and ExecutorService interface?**

Executor interface has only execute(Runnable) method. The implementing class of this interface has to execute the given Runnable instance passed to execute() method at some time in the future.

ExecutorService interface extends Executor interface. It provides additional methods like- invokeAny(), invokeAll(), shutdown(), awaitTermination(). These method provide the ability to shutdown the thread so that further requests can be rejected. Also it provides ability to invoke a collection of Callable tasks.

1. **What will happen on calling submit() method of an ExecutorService instance whose queue is already full?**

The implementation of ExecutorService will throw RejectedExecutionException, when its queue is already full and a new task is submitted by calling submit() method.

1. **What is a ScheduledExecutorService?**

ScheduledExecutorService interface extends the interface ExecutorService. It provides various schedule() methods that can be used to submit new tasks to be executed at a given point of time.

One of the schedule() method provides the ability to schedule a one-shot task that can be executed after given delay.

Another version of schedule() method provides the ability to execute ScheduleFuture after a given amount of delay.

In addition there are scheduleAtFixedRate() and scheduleWithFixedDelay() methods that can execute an action at a periodic interval of time.

1. **How will you create a Thread pool in Java?**

In Java, Executors framework provides a method newFixedThreadPool(int nThreads) that can be used to create a Thread pool with a fixed number of threads.

Sample code is as follows:

public static void main(String[] args) throws InterruptedException,

ExecutionException

{

ExecutorService myService = Executors.newFixedThreadPool(5); Future<Integer>[] futureList = new Future[5]; for (int i = 0; i < futureList.length; i++) {

futureList[i] = myService.submit(new MyCallable());

}

for (int i = 0; i < futureList.length; i++) { Integer retVal = futureList[i].get(); println(retVal);

}

myService.shutdown();

}

1. **What is the main difference between Runnable and Callable interface?**

Runnable interface defines run() method that does not return any value.

Callable interface allows call() method to return a value to its caller. A Callable interface can also throw an exception in case of an error. Also Callable is a newer addition to Java since version 1.5.

1. **What is a Semaphore in Java?**

Semaphore class in Java is used to implement a counting semaphore. It is used to restrict the number of threads that can access a physical or logical resource.

A Semaphore maintains a set of permits that should be acquired by competing threads.

We can also use it to control how many threads can access the critical section of a program or a resource concurrently.

The first argument in Semaphore constructor is the total number of permits available. Each invocation of acquire() method tries to obtain one of the available permits.

The acquire() method is used to acquire a permit from the semaphore. If we pass number of permits required to acquire() method, then it blocks the thread until that number of permits are available.

Once a thread has finished its work, we can use release() method to release the permits.

1. **What is a CountDownLatch in Java?**

CountDownLatch class helps in implementing synchronization in Java. It is used to implement the scenarios in which one or more threads have to wait until other threads have reached the same state such that all thread can start.

There is a synchronized counter that is decremented until it reaches the value zero. Once it reaches zero, it means that all waiting threads can proceed now.

It is a versatile tool that can be used for other Synchronization scenarios as well. It can also work as on/off latch or gate. All threads invoking await() method wait at the gate until it is opened by a thread invoking countdown() method.

1. **What is the difference between CountDownLatch and CyclicBarrier?**

CyclicBarrier takes an optional Runnable task that is run once the common barrier condition is achieved.

CountDownLatch is used in simple use cases where a simple start stop is required. A CyclicBarrier is useful in complex scenarios where more coordination is required. E.g. MapReduce algorithm implementation.

CyclicBarrier resets the internal value to the initial value once the value reaches zero. CyclicBarrier can be used to implement the scenarios in which threads have to wait for each other multiple times.

1. **What are the scenarios suitable for using Fork/Join framework?**

ForkJoinPool class is in the center of Fork/Join framework. It is a thread pool that can execute instances of ForkJoinTask.

ForkJoinTask class provides the fork() and join() methods. The fork() method is used to start the asynchronous execution of a task. The join() method is used to await the result of the computation.

Therefore, divide-and-conquer algorithms can be easily implemented with Fork/Join framework.

1. **What is the difference between RecursiveTask and RecursiveAction class?**

RecursiveAction class has compute() method that does not have to return a value.

RecursiveAction can be used when the action has to directly operate on a Data structure. It does not need to return any computed value.

In RecursiveTask class has compute() method that always returns a value.

Both RecursiveTask and RecursiveAction classes are used in ForkJoinTask implementations.

1. **In Java 8, can we process stream operations with a Thread pool?**

In Java 8, Collections provide parallelStream() method to create a stream that can be processed by a Thread pool.

We can also call the intermediate method parallel() on a given stream to convert it into a sequential stream of parallel tasks.

1. **What are the scenarios to use parallel stream in Java 8?**

A parallel stream in Java 8 has a much higher overhead compared to a sequential one.

It takes a significant amount of time to coordinate the threads.

We can use parallel stream in following scenarios:

When there are a large number of items to process and the processing of each item takes time and is parallelizable.

When there is a performance problem in the sequential processing. When current implementation is not already running in a multi-thread environment. If there is already a multi-threading environment, adding parallel stream can degrade the performance.

1. **How Stack and Heap work in Java multi-threading environment?**

In Java, Stack and heap are memory areas available to an application. Every thread has its own stack. It is used to store local variables, method parameters and call stack.

Local variables stored in Stack of one Thread are not visible to another Thread.

Whereas, Heap is a common memory area in JVM. Heap is shared by all threads. All objects are created inside heap.

To improve performance thread can cache the values from heap into their stack. This can create problem if the same variable is modified by more than one thread.

In such a scenario we should used volatile keyword to mark a variable volatile. For a volatile variable the thread always reads the value from main memory.

1. **How can we take Thread dump in Java?**

The steps to take Thread dump of Java process depends on the operating system.

On taking Thread dump, Java writes the state of all threads in log files or standard error console.

We can press Ctrl + Break key together to take thread dump in Windows.

We can execute kill -3 command for taking Thread dump on Linux. Another option to take Thread dump is jstack tool. We can pass process id of java process to this tool for taking Thread dump.

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 to learn more about this parameter.

1. **Which parameter can be used to control stack size of a thread in Java?**

We use –Xss parameter to control the stack size of a thread in Java.

If we set it as 1 MB, then every thread will get 1MB of stack size.

1. **There are two threads T1 and T2? How will you ensure that these threads run in sequence T1, T2 in Java?**

In Java there are multiple ways to execute threads in a sequence.

One of the simplest way for sequencing is join() method of Thread class.

We can call join() method to start a thread when another thread has finished.

We start with the last thread to execute first. And make this thread join on the next thread.

In this case we start thread T2 first. And then call T1.join() so that thread T2 waits for thread T1 to finish execution.

Once T1 completes execution, T2 thread starts executing.

**Question 1. in java?**

**Answer.**

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

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

**Answer.**  One process can have multiple Threads,

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

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

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

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

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

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

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

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

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

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

1. Thread creation by  implementing **java.lang.Runnable** interface.

We will create object of class which implements Runnable interface :

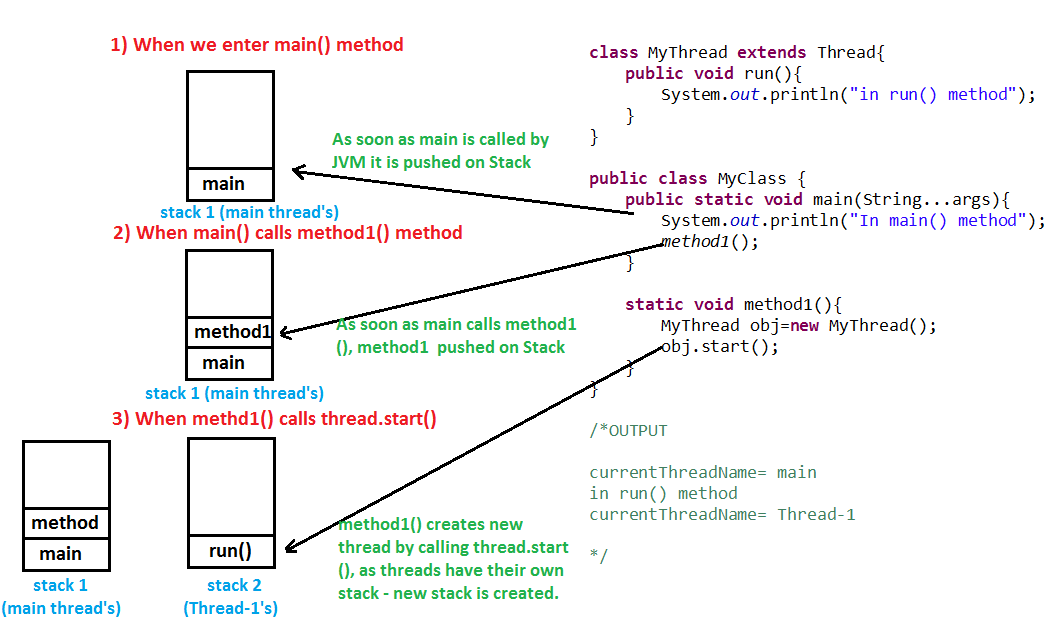
|  |
| --- |
| **MyRunnable runnable=new MyRunnable();**  **Thread thread=new Thread(runnable);** |

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

|  |
| --- |
| **Thread thread=new Thread(runnable);** |

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

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

[](http://www.javamadesoeasy.com/2015/03/threads-implement-their-own-stack.html)

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

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

*Differences between implementing Runnable interface and extending Thread class -*

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

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

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

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

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

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

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

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

10 salient features of **join()** method >

* **Definition** : join()We can use **join() method** to ensure all threads that started from main must end in order in which they started and also main should end in last.In other words **waits for thread to die on which thread has been called**.

* **Exception :** join**()** method [**throws**](http://www.javamadesoeasy.com/2015/05/throws-exception-in-java.html) **InterruptedException**, in our case we have thrown exception.

* **instance method :** join()is a **instance method**, hence we need to have thread  instance for calling this method.

* [**Thread state**](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html) **: when** join() **method is called on thread it goes from running to waiting state. And wait for thread to die.**

* **Not a native method :** implementation of join() method is provided in java.lang.Thread class.

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

|  |
| --- |
| **public** **final** **void** join() **throws** InterruptedException; |

* **synchronized block :** thread **need not to to acquire object lock** before calling join()method i.e. join() method **can be called from outside synchronized block**.

* **Waiting time :** join() **method have got few options.**
  1. **join() :** Waits for this thread to die.

|  |
| --- |
| **public** **final** **void** join() **throws** InterruptedException; |

This method internally calls **join(0).** And timeout of 0 means to wait forever;

* 1. **join(long millis) -** Waits at most millis milliseconds for this thread to die. A timeout of 0 means to wait forever.

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

* 1. **join(long millis, int nanos) -** Waits at most millis milliseconds plus nanos nanoseconds for this thread to die.

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

* **Belongs to which class :** join**() method belongs to java.lang.Thread** class.

To achieve we are going to create 2 threads on Runnable Object, create for loop in run() method and start  both threads. After starting each Thread call join() method on them to ensure they end in order in which they has started.

**Full Program to show usage of join() method>**

|  |
| --- |
| **class** MyRunnable **implements** Runnable{  **public** **void** run(){        System.*out*.println("in run() method");  **for**(**int** i=0;i<5;i++){                   System.*out*.println("i="+i+" ,ThreadName="+Thread.*currentThread*().getName());            }     }  }  /\*\* \*/  **public** **class** MyClass {  **public** **static** **void** main(String...args) **throws** InterruptedException{            System.*out*.println("In main() method");            MyRunnable runnable=**new** MyRunnable();            Thread thread1=**new** Thread(runnable);            Thread thread2=**new** Thread(runnable);            thread1.start();  **thread1.join();**            thread2.start();  **thread2.join();**            System.*out*.println("end main() method");     }  }  /\*OUTPUT  In main() method  in run() method  i=0 ,ThreadName=Thread-0  i=1 ,ThreadName=Thread-0  i=2 ,ThreadName=Thread-0  i=3 ,ThreadName=Thread-0  i=4 ,ThreadName=Thread-0  in run() method  i=0 ,ThreadName=Thread-1  i=1 ,ThreadName=Thread-1  i=2 ,ThreadName=Thread-1  i=3 ,ThreadName=Thread-1  i=4 ,ThreadName=Thread-1  end main() method  \*/ |

If we note output, all threads ended in order in which they were called and main thread has ended last.

First, main thread was called, it started Thread1 and then we called join() method on Thread1, once Thread1 ended main thread started Thread2 and we called join() method on Thread2, once Thread2 ended main thread also ended.

**In short - calling thread1.join()  made main thread to wait until Thread-1 dies.**

**Let’s discuss waiting time in detail :** join() **method have got few options.**

* 1. **join() :** Waits for this thread to die.

|  |
| --- |
| **public** **final** **void** join() **throws** InterruptedException; |

This method internally calls **join(0).** And timeout of 0 means to wait forever;

* 1. **join(long millis) -** Waits at most millis milliseconds for this thread to die. A timeout of 0 means to wait forever.

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

* 1. **join(long millis, int nanos) -** Waits at most millis milliseconds plus nanos nanoseconds for this thread to die.

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

Let’s create a program to use **join(long millis)** >

First, join(1000) will be called on Thread-1, **but once 1000 millisec are up, main thread can resume and start thread2 (main thread won’t wait for Thread-1 to die).**

|  |
| --- |
| **class MyRunnable implements Runnable{**  **public void run(){**  **System.*out*.println("in run() method");**  **for(int i=0;i<5;i++){**  **try {**  **Thread.*sleep*(500);**  **} catch (InterruptedException e) {**  **e.printStackTrace();**  **}**  **System.*out*.println("i="+i+" ,ThreadName="+Thread.*currentThread*().getName());**  **}**  **}**  **}**  /\*\* \*/  **public class MyClass {**  **public static void main(String...args) throws InterruptedException{**  **System.*out*.println("In main() method");**  **MyRunnable runnable=new MyRunnable();**  **Thread thread1=new Thread(runnable);**  **Thread thread2=new Thread(runnable);**  **thread1.start();**  **thread1.join(1000);  //once 1000 millisec are up, main thread can resume and start thread2.**  **thread2.start();**  **thread2.join();**  **System.*out*.println("end main() method");**  **}**  **}**  **/\*OUTPUT**  **In main() method**  **in run() method**  **i=0 ,ThreadName=Thread-0**  **i=1 ,ThreadName=Thread-0**  **in run() method**  **i=2 ,ThreadName=Thread-0**  **i=0 ,ThreadName=Thread-1**  **i=1 ,ThreadName=Thread-1**  **i=3 ,ThreadName=Thread-0**  **i=2 ,ThreadName=Thread-1**  **i=4 ,ThreadName=Thread-0**  **i=3 ,ThreadName=Thread-1**  **i=4 ,ThreadName=Thread-1**  **end main() method**  **\*/** |

***Important Thread methods (salient features, usage with programs)>***

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

### [Sleep() method in threads - 10 key features with programs](http://www.javamadesoeasy.com/2015/03/sleep-method-in-threads-10-key-features.html)

### [Yield() method in threads - 8 key features with programs](http://www.javamadesoeasy.com/2015/03/yield-method-in-threads-8-key-features.html)

### [Wait() and notify() methods- Definition, 8 key features, solving consumer producer problem with & without these methods and consequences of not using wait() and notify() methods.](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html)

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

### [2 alternate ways to stop thread, as stop() method is deprecated](http://www.javamadesoeasy.com/2015/03/2-alternate-ways-to-stop-thread-as-stop.html)

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

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

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

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

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

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

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

Few small questions>

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

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

Q. Can we have synchronized variable in java?

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

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

**Answer.** Its very important question from interview perspective.

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

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

**volatile** int i; **//legal**

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

Synchronized method:

**synchronized** **void** method2(){} //legal

Synchronized block:

**void** method2(){

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

          //code inside synchronized block.

          }

}

Synchronized variable (illegal):

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

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

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

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

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

Let’s take an example-

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

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

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

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

**Few sub questions>**

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

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

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

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

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

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

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

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

**Answer.**

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

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

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

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

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

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

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

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

Every thread having their monitor will create few problems -

>**Thread communication problem.**

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

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

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

**Answer.** **Yes**, it’s mandatory to acquire object lock before calling these methods on object. As discussed above [**wait(), notify()  and notifyAll()**](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) **methods are always called from** [**Synchronized block**](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html) **only**, and **as soon as thread enters synchronized block it acquires object lock** (by holding object monitor). If we call these methods without acquiring object lock i.e. from outside synchronize block then java.lang. IllegalMonitorStateException is thrown at runtime.

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

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

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

**Key points** we need to ensure before programming :

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

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

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

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

Explanation of **Logic** >

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

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

First we will start consumerThread >

|  |
| --- |
| consumerThread.start(); |

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

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

|  |
| --- |
| **synchronized** (sharedQueue) {  **while** (sharedQueue.size() == 0) {                 sharedQueue.wait();          }        } |

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

|  |
| --- |
| **synchronized** (sharedQueue) {            Thread.*sleep*((**long**)(Math.*random*() \* 2000));          System.*out*.println("consumed : "+ sharedQueue.remove(0));          sharedQueue.notify();         } |

Than we will start producerThread >

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

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

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

|  |
| --- |
| **synchronized** (sharedQueue) {  **while** (sharedQueue.size() == maxSize) { //maxsize is 2        sharedQueue.wait();       }   } |

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

|  |
| --- |
| **synchronized** (sharedQueue) {        System.*out*.println("Produced : " + i);        sharedQueue.add(i);        Thread.*sleep*((**long**)(Math.*random*() \* 1000));        sharedQueue.notify();   } |

**Key points** we need to ensure before programming :

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

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

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

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

Explanation of **Logic** >

It’s important to know that **sharedQueue  is a** [**queue implemented using Linked List**](http://javamadesoeasy.blogspot.in/2015/02/implement-queue-using-linked-list.html)**.**

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

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

First we will start consumerThread >

|  |
| --- |
| consumerThread.start(); |

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

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

|  |
| --- |
| **synchronized** (sharedQueue) {  **while** (sharedQueue.size() == 0) {                 sharedQueue.wait();          }        } |

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

|  |
| --- |
| **synchronized** (sharedQueue) {            Thread.*sleep*((**long**)(Math.*random*() \* 2000));          System.*out*.println("consumed : "+ sharedQueue.remove(0));          sharedQueue.notify();         } |

Than we will start producerThread >

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

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

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

|  |
| --- |
| **synchronized** (sharedQueue) {  **while** (sharedQueue.size() == maxSize) { //maxsize is 2        sharedQueue.wait();       }   } |

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

|  |
| --- |
| **synchronized** (sharedQueue) {        System.*out*.println("Produced : " + i);        sharedQueue.add(i);        Thread.*sleep*((**long**)(Math.*random*() \* 1000));        sharedQueue.notify();   } |

Full Program/sourceCode to solve consumer producer problem using wait() and notify() method>

|  |
| --- |
| **import java.util.LinkedList;**  **import java.util.List;**  **/\*\***  **\* Producer Class.**  **\*/**  **class Producer implements Runnable {**  **private List<Integer> sharedQueue;**  **private int maxSize=2; //maximum number of products which sharedQueue can hold at a time.**  **public Producer(List<Integer> sharedQueue) {**  **this.sharedQueue = sharedQueue;**  **}**  **@Override**  **public void run() {**  **for (int i = 1; i <= 10; i++) {  //produce 10 products.**  **try {**  **produce(i);**  **} catch (InterruptedException e) {  e.printStackTrace();   }**  **}**  **}**  **private void produce(int i) throws InterruptedException {**    **synchronized (sharedQueue) {**  **//if sharedQuey is full wait until consumer consumes.**  **while (sharedQueue.size() == maxSize) {**  **System.*out*.println("Queue is full, producerThread is waiting for "**  **+ "consumerThread to consume, sharedQueue's size= "+maxSize);**  **sharedQueue.wait();**  **}**  **}**    **/\* 2 Synchronized blocks have been used means before**  **\* producer produces by entering below synchronized**  **\* block consumer can consume.**  **\*/**    **//as soon as producer produces (by adding in sharedQueue) it notifies consumerThread.**  **synchronized (sharedQueue) {**  **System.*out*.println("Produced : " + i);**  **sharedQueue.add(i);**  **Thread.*sleep*((long)(Math.*random*() \* 1000));**  **sharedQueue.notify();**  **}**  **}**  **}**  **/\*\***  **\* Consumer Class.**  **\*/**  **class Consumer implements Runnable {**  **private List<Integer> sharedQueue;**  **public Consumer(List<Integer> sharedQueue) {**  **this.sharedQueue = sharedQueue;**  **}**    **@Override**  **public void run() {**  **while (true) {**  **try {**  **consume();**  **Thread.*sleep*(100);**  **} catch (InterruptedException e) {  e.printStackTrace();   }**  **}**  **}**  **private void consume() throws InterruptedException {**    **synchronized (sharedQueue) {**  **//if sharedQuey is empty wait until producer produces.**  **while (sharedQueue.size() == 0) {**  **System.*out*.println("Queue is empty, consumerThread is waiting for "**  **+ "producerThread to produce, sharedQueue's size= 0");**  **sharedQueue.wait();**  **}**  **}**    **/\* 2 Synchronized blocks have been used means before**  **\* consumer start consuming by entering below synchronized**  **\* block producer can produce.**  **\*/**    **/\*If sharedQueue not empty consumer will consume**  **\* (by removing from sharedQueue) and notify the producerThread.**  **\*/**  **synchronized (sharedQueue) {**  **Thread.*sleep*((long)(Math.*random*() \* 2000));**  **System.*out*.println("CONSUMED : "+ sharedQueue.remove(0));**  **sharedQueue.notify();**  **}**  **}**    **}**  /\*\* \*/  **public class ProducerConsumerWaitNotify {**  **public static void main(String args[]) {**  **List<Integer> sharedQueue = new LinkedList<Integer>(); //Creating shared object**    **Producer producer=new Producer(sharedQueue);**  **Consumer consumer=new Consumer(sharedQueue);**    **Thread producerThread = new Thread(producer, "ProducerThread");**  **Thread consumerThread = new Thread(consumer, "ConsumerThread");**  **producerThread.start();**  **consumerThread.start();**  **}**  **}**  **/\*OUTPUT**  **Queue is empty, consumerThread is waiting for producerThread to produce, sharedQueue's size= 0**  **Produced : 1**  **CONSUMED : 1**  **Produced : 2**  **CONSUMED : 2**  **Produced : 3**  **Produced : 4**  **CONSUMED : 3**  **Produced : 5**  **Queue is full, producerThread is waiting for consumerThread to consume, sharedQueue's size= 2**  **CONSUMED : 4**  **Produced : 6**  **Queue is full, producerThread is waiting for consumerThread to consume, sharedQueue's size= 2**  **CONSUMED : 5**  **Produced : 7**  **CONSUMED : 6**  **Produced : 8**  **Queue is full, producerThread is waiting for consumerThread to consume, sharedQueue's size= 2**  **CONSUMED : 7**  **Produced : 9**  **CONSUMED : 8**  **Produced : 10**  **CONSUMED : 9**  **CONSUMED : 10**  **Queue is empty, consumerThread is waiting for producerThread to produce, sharedQueue's size= 0**  **\*/** |

Producer will allow consumer to consume only when 10 products have been produced (i.e. when production is over).

It’s important to know that **sharedQueue  is a** [**queue implemented using Linked List**](http://javamadesoeasy.blogspot.in/2015/02/implement-queue-using-linked-list.html)**.**

**In program consumer thread will start() and wait by calling wait() method till producer is producing. Once production is over, producer thread will call notify() method, which will notify consumer thread and consumer will start consuming.**

|  |
| --- |
| **import java.util.LinkedList;**  **import java.util.List;**  **/\*\***  **\* Producer Class in java, Producer will allow consumer to**  **\* consume only when 10 products have been produced**  **\* (i.e. when production is over).**  **\*/**  **class Producer implements Runnable{**  **List<Integer> sharedQueue;**    **Producer(){**  **sharedQueue=new LinkedList<Integer>();**  **}**    **@Override**  **public void run(){**    **synchronized (this) {**  **for(int i=1;i<=10;i++){ //Producer will produce 10 products**  **sharedQueue.add(i);**  **System.*out*.println("Producer is still Producing, Produced : "+i);**    **try{**  **Thread.*sleep*(1000);**  **}catch(InterruptedException e){e.printStackTrace();}**    **}**  **System.*out*.println("Production is over, consumer can consume.");**  **//Production is over, notify consumer thread so that consumer can consume.**  **this.notify();**  **}**  **}**    **}**  **/\*\***  **\* Consumer Class.**  **\*/**  **class Consumer extends Thread{**  **Producer prod;**    **Consumer(Producer obj){**  **prod=obj;**  **}**    **public void run(){**  **/\***  **\* consumer will wait till producer is producing.**  **\*/**  **synchronized (this.prod) {**    **System.*out*.println("Consumer waiting for production to get over.");**  **try{**  **this.prod.wait();**  **}catch(InterruptedException e){e.printStackTrace();}**    **}**      **/\*production is over, consumer will start consuming.\*/**  **int productSize=this.prod.sharedQueue.size();**  **for(int i=0;i<productSize;i++)**  **System.*out*.println("CONSUMED : "+ this.prod.sharedQueue.remove(0) +" ");**    **}**    **}**  /\*\* \*/  **public class ProducerConsumerWithWaitNotify {**  **public static void main(String args[]) throws InterruptedException{**    **Producer prod=new Producer();**  **Consumer cons=new Consumer(prod);**    **Thread prodThread=new Thread(prod,"prodThread");**  **Thread consThread=new Thread(cons,"consThread");**    **consThread.start();     //start consumer thread.**  **Thread.*sleep*(100);      //This minor delay will ensure that consumer thread starts before producer thread.**  **prodThread.start();     //start producer thread.**      **}**  **}**  **/\*OUTPUT**  **Consumer waiting for production to get over.**  **Producer is still Producing, Produced : 1**  **Producer is still Producing, Produced : 2**  **Producer is still Producing, Produced : 3**  **Producer is still Producing, Produced : 4**  **Producer is still Producing, Produced : 5**  **Producer is still Producing, Produced : 6**  **Producer is still Producing, Produced : 7**  **Producer is still Producing, Produced : 8**  **Producer is still Producing, Produced : 9**  **Producer is still Producing, Produced : 10**  **Production is over, consumer can consume.**  **CONSUMED : 1**  **CONSUMED : 2**  **CONSUMED : 3**  **CONSUMED : 4**  **CONSUMED : 5**  **CONSUMED : 6**  **CONSUMED : 7**  **CONSUMED : 8**  **CONSUMED : 9**  **CONSUMED : 10**  **\*/** |

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

**Answer.**

Producer will allow consumer to consume only when 10 products have been produced (i.e. when production is over).

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

**In program producer thread will start() and it will start producing  and called sleep(1000) in between, which will give consumer thread chance to execute. consumer checks whether productionInProcess is true or not, if it's true,**

**consumer will sleep(4000) and wake up after specified time and again check whether productionInProcess is true or false. process will repeat till productionInProcess is true. Meanwhile, producer thread will complete production and ultimately make productionInProcess to false. Once productionInProcess is false, consumer will consume.**

|  |
| --- |
| **import** java.util.LinkedList;  **import** java.util.List;  /\*\*  \* Producer Class in java, Producer will allow consumer to consume only  \* when 10 products have been produced (i.e. when production is over).  \*/  **class** Producer **implements** Runnable{    **boolean** productionInProcess;     List<Integer> list;       Producer(){            //initially Producer will be producing, so make this productionInProcess true.            productionInProcess=**true**;            list=**new** LinkedList<Integer>();     }       @Override  **public** **void** run(){    **for**(**int** i=1;i<=10;i++){ //Producer will produce 10 products                   list.add(i);                   System.*out*.println("Producer is still Producing, Produced : "+i);    **try**{                         Thread.*sleep*(1000);                   }**catch**(InterruptedException e){e.printStackTrace();}              }              /\* Once production is over, make this productionInProcess false.            \* Production is over, consumer can consume.            \*/            productionInProcess=**false**;       }    }  /\*\*  \* Consumer Class.  \*/  **class** Consumer **extends** Thread{     Producer prod;       Consumer(Producer obj){      prod=obj;     }    **public** **void** run(){            /\*            \* consumer checks whether productionInProcess is true or not,            \* if it's true, consumer will sleep and wake up after certain time            \* and again check whether productionInProcess is true or false.            \* process will repeat till productionInProcess is true.            \* Once productionInProcess is false we'll exit below while loop.            \*/  **while**(**this**.prod.productionInProcess){              System.*out*.println("Consumer waiting for production to get over.");  **try**{                   Thread.*sleep*(4000);              }**catch**(InterruptedException e){e.printStackTrace();}              }                /\*productionInProcess is false means production is over,            \* consumer will start consuming. \*/            System.*out*.println("Production is over, consumer can consume.");  **int** productSize=**this**.prod.list.size();  **for**(**int** i=0;i<productSize;i++)                   System.*out*.println("CONSUMED : "+ **this**.prod.list.remove(0) +" ");       }    }  /\*\* \*/  **public** **class** ProducerConsumerWithoutWaitNotify {  **public** **static** **void** main(String args[]){              Producer prod=**new** Producer();            Consumer cons=**new** Consumer(prod);              Thread prodThread=**new** Thread(prod,"prodThread");            Thread consThread=**new** Thread(cons,"consThread");              prodThread.start();     //start producer thread.            consThread.start();     //start consumer thread.         }  }  /\*OUTPUT  Consumer waiting for production to get over.  Producer is still Producing, Produced : 1  Producer is still Producing, Produced : 2  Producer is still Producing, Produced : 3  Producer is still Producing, Produced : 4  Consumer waiting for production to get over.  Producer is still Producing, Produced : 5  Producer is still Producing, Produced : 6  Producer is still Producing, Produced : 7  Producer is still Producing, Produced : 8  Consumer waiting for production to get over.  Producer is still Producing, Produced : 9  Producer is still Producing, Produced : 10  Production is over, consumer can consume.  CONSUMED : 1  CONSUMED : 2  CONSUMED : 3  CONSUMED : 4  CONSUMED : 5  CONSUMED : 6  CONSUMED : 7  CONSUMED : 8  CONSUMED : 9  CONSUMED : 10  \*/ |

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

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

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

Key methods for solving consumer producer pattern are >

|  |
| --- |
| **put(i);**   //used by producer to put/produce in sharedQueue.  **take();** //used by consumer to take/consume from sharedQueue. |

Example/ Full Program/sourceCode to solve consumer producer problem using custom BlockingQueue in java >

|  |
| --- |
| **import java.util.concurrent.BlockingQueue;**  **import java.util.concurrent.LinkedBlockingQueue;**  **/\*\***  **\* Producer Class in java.**  **\*/**  **class Producer implements Runnable {**  **private final BlockingQueue<Integer> sharedQueue;**  **public Producer(BlockingQueue<Integer> sharedQueue) {**  **this.sharedQueue = sharedQueue;**  **}**  **@Override**  **public void run() {**  **for(int i=1; i<=10; i++){**  **try {**  **System.*out*.println("Produced : " + i);**  **//put/produce into sharedQueue.**  **sharedQueue.put(i);**  **} catch (InterruptedException ex) {**    **}**  **}**  **}**  **}**  **/\*\***  **\* Consumer Class in java.**  **\*/**  **class Consumer implements Runnable{**  **private BlockingQueue<Integer> sharedQueue;**  **public Consumer (BlockingQueue<Integer> sharedQueue) {**  **this.sharedQueue = sharedQueue;**  **}**  **@Override**  **public void run() {**  **while(true){**  **try {**  **//take/consume from sharedQueue.**  **System.*out*.println("CONSUMED : "+ sharedQueue.take());**  **} catch (InterruptedException ex) {**    **}**  **}**  **}**  **}**  /\*\* \*/  **public class ProducerConsumerBlockingQueue {**  **public static void main(String args[]){**    **//Creating shared object**  **BlockingQueue<Integer> sharedQueue = new LinkedBlockingQueue<Integer>();**    **Producer producer=new Producer(sharedQueue);**  **Consumer consumer=new Consumer(sharedQueue);**    **Thread producerThread = new Thread(producer, "ProducerThread");**  **Thread consumerThread = new Thread(consumer, "ConsumerThread");**  **producerThread.start();**  **consumerThread.start();**  **}**  **}**  **/\*OUTPUT**  **Produced : 1**  **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**  **Produced : 10**  **CONSUMED : 9**  **CONSUMED : 10**  **\*/** |

SUMMARY >

So in this thread concurrency tutorial we learned how to **Solve Consumer Producer problem by using BlockingQueue and LinkedBlockingQueue** in multithreading in java.

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

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

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

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

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

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

**Conclusion:**

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

|  |
| --- |
| **Code called by Thread-1**    **public** **void** run() {  **synchronized** (String.**class**) {                     Thread.sleep(100);  **synchronized** (Object.**class**) {                   }            }  }  **Code called by Thread-2**    **public** **void** run() {  **synchronized** (Object.**class**) {                     Thread.sleep(100);  **synchronized** (String.**class**) {                   }            }  } |

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

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

**Conclusion:** No deadlock is formed.

|  |
| --- |
| **Code called by Thread-1**    **public** **void** run() {  **synchronized** (String.**class**) {                     Thread.sleep(100);  **synchronized** (Object.**class**) {                   }            }  }  **Code called by Thread-2**    **public** **void** run() {  **synchronized** (String.**class**) {                     Thread.sleep(100);  **synchronized** (Object.**class**) {                   }            }  } |

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

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

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

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

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

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

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

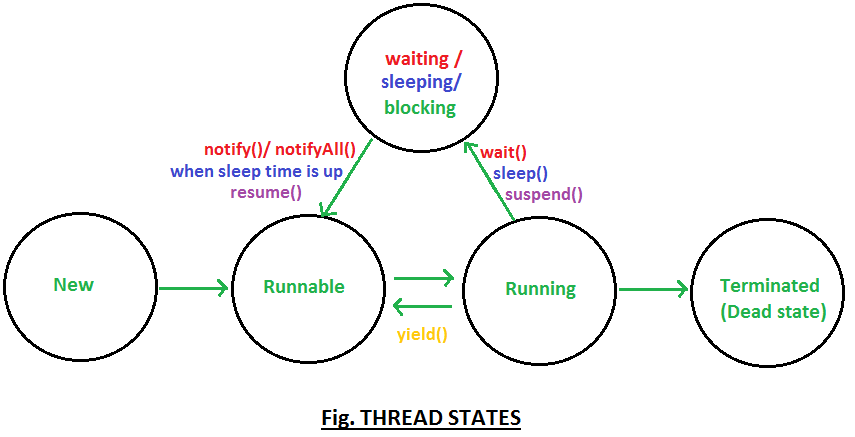
**Question 22. What is life cycle of Thread, explain thread states? (Important)**

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

**Thread have following states >**

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

**Thread states/ Thread life cycle in diagram >**

****

**Thread states in detail >**

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

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

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

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

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

**> How can Thread go from running to waiting state?**

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

**> How can Thread go from running to sleeping state?**

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

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

**You may like to have in depth knowledge of** [**Thread states/ Thread life cycle in java & explanation of thread methods which method puts thread from which state to which state.**](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

* **static method :** sleep()is a static method, causes the currently executing thread to sleep for the specified number of milliseconds.
* **Native method :** implementation of sleep() method is provided by **JVM**.

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

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

* **Belongs to which class :** [sleep**()**](http://www.javamadesoeasy.com/2015/03/sleep-method-in-threads-10-key-features.html) **method belongs to java.lang.Thread** class.
* **synchronized block :** thread **need not to to acquire object lock** before calling sleep()method i.e. sleep() method **can be called from outside synchronized block**.

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

**Answer.**

* **Should be called from** [**synchronized block**](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html) **:** wait() **method is always called from synchronized block** i.e. [wait()](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) method needs to lock object monitor before object on which it is called.  But sleep() **method can be called from outside synchronized block** i.e. sleep() method doesn’t need any object monitor.
* **IllegalMonitorStateException : if** wait() **method is called without acquiring object lock** than IllegalMonitorStateException is thrown at runtime, but sleep() method **never throws such exception**.
* **Belongs to which class : wait() method belongs to java.lang.Object** class but **sleep() method belongs to java.lang.Thread** class.
* **Called on object or thread :** wait() **method is called on objects** but sleep() **method is called on Threads** not objects**.**
* [**Thread state**](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html) **: when** wait() **method is called on object, thread that holded object’s monitor goes from running to waiting state** and can **return to runnable state only when notify() or notifyAll()** **method is called on that object**. And later thread scheduler schedules that thread to go from from runnable to running state.

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

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

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

**Answer.**

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

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

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

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

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

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

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

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

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

s is pointing to String object.

|  |
| --- |
| **public** **class** MyClass {  **final** String s=**new** String("a");  **void** method(){            s="b"; //compilation error, s cannot point to new reference.     }   } |

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

|  |
| --- |
| **public** **class** MyClass {  **final** **int** i=0;  **void** method(){            i=0;  //compilation error, i cannot point to new value.     }   } |

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

|  |
| --- |
| **public** **class** MyClass {  **void** method(){  **int** i=0; //Local variable, is thread safe.     }   } |

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

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

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

**> How can Thread go from running to waiting state ?**

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

**> How can Thread return from waiting to runnable state ?**

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

**> How can Thread go from running to sleeping state ?**

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

**> How can Thread return from sleeping to runnable state ?**

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

|  |
| --- |
| **class** MyThread **extends** Thread {     //don't override run() method  }  **public** **class** DontOverrideRun {  **public** **static** **void** main(String[] args) {            System.*out*.println("main has started.");            MyThread thread1=**new** MyThread();            thread1.start();            System.*out*.println("main has ended.");     }  }  /\*OUTPUT  main has started.  main has ended.  \*/ |

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

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

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

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

|  |
| --- |
| **class** MyThread **extends** Thread {     @Override  **public** **void** run() {            System.*out*.println("in run() method");     }       @Override  **public** **void** start(){            System.*out*.println("In start() method");     }    }  **public** **class** OverrideStartMethod {  **public** **static** **void** main(String[] args) {            System.*out*.println("main has started.");              MyThread thread1=**new** MyThread();            thread1.start();              System.*out*.println("main has ended.");     }  }  /\*OUTPUT  main has started.  In start() method  main has ended.  \*/ |

If we note output. we have overridden start method and didn’t called run() method from it, so, run() method wasn’t call.

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

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

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

1. Entering **synchronized block or**

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

**synchronized** (MyClass.class) {

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

      }

1. by entering **static synchronized methods.**

**public static** **synchronized void** method1() {

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

      }

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

Thread will leave lock when it exits static synchronized method.

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

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

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

**Let’s me give you some tricky situation based question,**

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

**>**first one on **class’s class object.**

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

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

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

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

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

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

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

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

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

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

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

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

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

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

**Answer.**

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

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

**Answer.**

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

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

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

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

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

*Need/Advantage of ThreadPool?*

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

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

*How ThreadPool works?*

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

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

Here 2 threads will be created and started in ThreadPool.

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

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

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

* Else waits for tasks to become available.

**When tasks are added?**

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

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

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

More detail on how to [Implement Thread pool in java](http://www.javamadesoeasy.com/2015/03/implement-thread-pool-in-java.html).

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

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

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

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

*For how long Thread holds ThreadLocal value?*

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

*Can one thread see other thread’s ThreadLocal value?*

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

*Are ThreadLocal variables thread safe. Why?*

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

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

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

*Creating ThreadLocal >*

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

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

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

*How thread set value of ThreadLocal >*

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

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

*How thread get value of ThreadLocal >*

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

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

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

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

**Answer.**

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

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

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

*Solution to busy spin -*

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

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

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

*Program - Consumer Producer problem with busy spin >*

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

|  |
| --- |
| **while(productionInProcess){**  **System.*out*.println("BUSY SPIN - Consumer waiting for production to get over");**  **}** |

[See here for Busy spin in detail](http://www.javamadesoeasy.com/2015/03/busy-spin-what-is-busy-spin-consumer.html).

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

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

**Enclosing** constructor in synchronized block will generate compilation error.

Using synchronized in **constructor definition** will also show compilation error.

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

**Though we can use synchronized block inside constructor.**

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

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

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

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

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

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

**Thread starvation** may happen in following scenarios >

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

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

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

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

When JVM (Java virtual machine)  shuts down >

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

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

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

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

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

What is setDefaultUncaughtExceptionHandler method?

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

*setDefaultUncaughtExceptionHandler method features >*

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

Defining setDefaultUncaughtExceptionHandler method >

|  |
| --- |
| Thread.***setDefaultUncaughtExceptionHandler***(**new** Thread.UncaughtExceptionHandler(){  **public** **void** uncaughtException(Thread thread, Throwable throwable) {     System.*out*.println(thread.getName() + " has thrown " + throwable);     }    }); |

*For more detail read :* [*Program to demonstrate* ***setDefaultUncaughtExceptionHandler*** *method.*](http://www.javamadesoeasy.com/2015/03/handling-uncaught-runtime-exception.html)

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

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

**ThreadGroup is initialized with default priority of 10.**

*ThreadGroup* ***important methods*** *>*

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

*Read more about* [***ThreadGroup in java***](http://www.javamadesoeasy.com/2015/03/threadgroup-in-java.html)*.*

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

**Answer.**

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

Where **1 is minimum priority** and **10 is maximum priority.**

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

|  |
| --- |
| /\* The minimum priority that a thread can have. \*/  **public** **final** **static** **int** ***MIN\_PRIORITY*** = 1;      /\* The default priority that is assigned to a thread. \*/  **public** **final** **static** **int** ***NORM\_PRIORITY*** = 5;     /\* The maximum priority that a thread can have. \*/  **public** **final** **static** **int** ***MAX\_PRIORITY*** = 10; |

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

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

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

***getPriority()* method returns the thread’s priority.**

**Question 1. What is executor framework in java?**

**Answer.** This is very important question to start your interview with. [Executor and ExecutorService](http://www.javamadesoeasy.com/2015/03/executor-and-executorservice-framework.html) are used for  following purposes >

* creating thread in java,
* starting threads in java,
* managing whole [life cycle of Threads](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html) in java.

Executor creates [pool of threads](http://www.javamadesoeasy.com/2015/03/implement-thread-pool-in-java.html) and manages life cycle of all threads in it.

In Executor framework, **Executor** interface and  **ExecutorService**  class are most prominently used in java.

*Executor* interface defines very important execute() method which executes command in java.

*ExecutorService* interface extends **Executor** interface.

An Executor interface provides following type of methods >

* methods for managing termination and
* methods that can produce a Future for tracking progress of tasks in java.

An Executor that provides methods to manage termination and methods that can produce a Future for tracking progress of one or more asynchronous tasks.

For more information read [**Executor and ExecutorService framework in java**](http://www.javamadesoeasy.com/2015/03/executor-and-executorservice-framework.html).

**Question 2. What are differences between execute() and submit() method of executor framework in java?**

**Answer.** This is basic question, beforehand you must know about [Executor Service Framework](http://www.javamadesoeasy.com/2015/03/executor-and-executorservice-framework.html).

|  |  |
| --- | --- |
| **execute()** method | **submit()** method |
| **execute()** method is defined in *Executor* interface in java. | **submit()** method is defined in *ExecutorService* interface in java. |
| It can be used for executing **runnable task in java in java**. | It can be used for executing **runnable  task** or **callable task**, submitted callable returns future and Future's get method will return the task's result in java. |
| **Signature of execute method is  >*****void*** *execute****(Runnable*** *task****)*** | submit method has 3 forms >  *<T> Future<T>* ***submit****(Callable<T>* ***task****)*  Submits a callable ***task*** for execution.  Method **returns** a Future which represents pending results of the task.  Once task is completed Future's get method will return the task's result.  *<T> Future<T>* ***submit****(Runnable* ***task****, T* ***result****)*  Submits a Runnable ***task*** for execution.  Method **returns** a Future which represents that task. Once task is completed Future's get method will return ***result***.  *Future<?>* ***submit****(Runnable* ***task****)*  Submits a Runnable ***task*** for execution.  Method **returns** a Future which represents that task. Once task is completed Future's get method will return null. |

**Question 3.What is Semaphore in java 7?**

**Answer.**  This is very important question for freshers and experienced. A [**semaphore**](http://www.javamadesoeasy.com/2015/03/semaphore-in-java.html) controls access to a shared resource by using permits in java.

* **If permits are greater than zero**, then semaphore **allow access to shared resource**.
* **If permits are zero or less than zero**, then semaphore **does not allow access to shared resource in java**.

These permits are sort of counters, which allow access to the shared resource. Thus, to access the resource, a thread must be granted a permit from the semaphore in java.

*Semaphore has 2 constructors >*

* **Semaphore**(int ***permits***)

***permits*** is the **initial number of permits available**.

This value can be negative, in which case releases must occur before any acquires will be granted, ***permits*** is number of threads that can access shared resource at a time.

If ***permits*** is 1, then only one threads that can access shared resource at a time in java.

* **Semaphore**(int **permits**, boolean **fair**)

**permits** is the initial number of permits available.

This value can be negative, in which case releases must occur before any acquires will be granted.

By setting **fair** to **true**, we ensure that **waiting threads are granted a permit in the order in which they requested access**.

*Semaphore’s acquire( ) method has 2 forms :*

* void **acquire**( ) throws InterruptedException

Acquires a permit if one is available and **reduces the number of available permits by 1**.

If no permit is available then the current thread becomes dormant until

>some other thread calls release() method on this semaphore or,

>some other thread interrupts the current thread.

* void **acquire**(int **permits**) throws InterruptedException

Acquires **permits** number of permits if available and **reduces the number of available permits by permits.**

If **permits** *number of* permits are not available then the current thread becomes dormant until  one of the following things happens -

>some other thread calls release() method on this semaphore and available permits become equal to **permits** or,

>some other thread interrupts the current thread.

*Semaphore’s release( ) method has 2 forms in java :*

* void **release**( )

Releases a permit and **increases the number of available permits by 1**.

For releasing lock by calling release() method it’s not mandatory that thread must have acquired permit by calling acquire() method in java.

* void **release**(int **permits**)

Releases **permits** number of permits and **increases the number of available permits by permits.**

For releasing lock by calling release(int *permits*) method it’s not mandatory that thread must have acquired permit by calling acquire()/acquire(int permit) method in java.

Read more about [**Semaphore in java**](http://www.javamadesoeasy.com/2015/03/semaphore-in-java.html).

**Question 4. How can you implement Producer Consumer pattern using Semaphore in java?**

**Answer.** This is tricky question for even experienced guys. [**Semaphore**](http://www.javamadesoeasy.com/2015/03/semaphore-in-java.html) **on producer is created with permit =1**. So, that **producer can get the permit to produce**.

**Semaphore on consumer is created with permit =0**. So, that **consumer could wait for permit to consume**. [because initially producer hasn’t produced any product]

**Producer gets permit by** calling **semaphoreProducer.acquire()** and **starts producing**, **after producing** it calls **semaphoreConsumer.release()**. So, that **consumer could get the  permit to consume**.

|  |
| --- |
| **semaphoreProducer.acquire();**  **System.*out*.println("Produced : "+i);**  **semaphoreConsumer.release();** |

**Consumer gets permit by** calling **semaphoreConsumer.acquire()** and **starts consuming**, **after consuming** it calls **semaphoreProducer.release()**. So, that **producer could get the  permit to produce**.

|  |
| --- |
| **semaphoreConsumer.acquire();**  **System.*out*.println("Consumed : "+i);**  **semaphoreProducer.release();** |

Read more on [**Semaphore used for implementing Producer Consumer pattern**](http://www.javamadesoeasy.com/2015/03/semaphore-used-for-implementing.html).

**Question 5. How can you implement your own Semaphore?**

**Answer. Experienced developers must be able to answer this question.** [**Implementation of custom/own Semaphore in java**](http://www.javamadesoeasy.com/2015/03/implementation-of-customown-semaphore.html)**.**

**Question 6. What is significance of atomic classes in java 7?**

**Answer.** Another important and basic question for freshers. Java provides some classes in [**java.util.concurrent.atomic**](http://www.javamadesoeasy.com/2015/03/atomic-operations-in-java.html) which offers an alternative to the other [synchronization](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html) in java.

*Classes found in* ***java.util.concurrent.atomic*** *are >*

* **AtomicInteger**,
* **AtomicLong,** and
* **AtomicBoolean.**

*Methods provided by these classes >*

* **get( ),**
* **set( ),**
* **getAndSet( )**,
* **compareAndSet( ), and**
* **decrementAndGet( ).**

In [multithreading](http://www.javamadesoeasy.com/2015/03/what-is-thread-in-java.html) environment we can use these classes without any explicit synchronization, as all these classes are [thread safe](http://www.javamadesoeasy.com/2015/03/guidelines-to-thread-safe-code-most.html) in java.

For more information on atomic read [**Atomic operations in java**](http://www.javamadesoeasy.com/2015/03/atomic-operations-in-java.html).

**Question 7. What are Future and Callable? How are they related in java?**

**Answer.**  This is **very very important** question. They are widely used in thread concurrency.

[*Future<V>*](http://www.javamadesoeasy.com/2015/03/executor-and-executorservice-framework.html) interface provides methods >

* for **returning result** of computation, wait until computation is not completed and
* for **cancelling** the computation in between.

*Future Methods >*

*V* ***get****()* method returns the result of computation, method waits for computation to complete.

*cancel method* cancels the task.

[*Callable<V>*](http://www.javamadesoeasy.com/2015/03/executor-and-executorservice-framework.html) interface provides method for computing a result and returning that computed result or throws an exception if unable to do so

Any class implementing Callable interface must override ***call****()* method for computing a result.

Method returns computed result or throws an exception if unable to do so in java.

what type of results Callable’s call() method can return in java?

The Callable<V> is a generic interface, so its call method can return generic result specified by *V*.

How Callable and Future are related?

If you submit a Callable object to an Executor returned object is of Future type.

|  |
| --- |
| Future<Double> futureDouble=executor.submit(**new** SquareDoubleCallable(2.2)); |

where, SquareDoubleCallable is a class which implements Callable.

This Future object can check the status of a Callable call’s method and wait until Callable’s call() method is not completed.

For more information read [**Executor and ExecutorService framework in java**](http://www.javamadesoeasy.com/2015/03/executor-and-executorservice-framework.html).

**Question 8. Similarity and differences between java.util.concurrent.Callable and  java.lang.Runnable in java?**

**Answer.** This is basic question.

Similarity between java.util.concurrent.Callable and  java.lang.Runnable in java?

Instances of class which implements callable are executed by another thread.

Difference between java.util.concurrent.Callable and  java.lang.Runnable in java?

Class implementing Callable interface must override call() method. call() method returns computed result or throws an exception if unable to do so.

Class implementing Runnable interface must override run() method.

A Runnable does not return a result and can neither throw a checked exception in java.

**Question 9. What is CountDownLatch in java?**

**Answer.**  This is very important question. Fresher and experienced bost be well versed with this. There might be situation where we might like our thread to wait until one or more threads completes certain operation in java.

A [CountDownLatch](http://www.javamadesoeasy.com/2015/03/countdownlatch-in-java.html) is initialized with a given ***count*** .

***count*** specifies the number of events that must occur before latch is released.

Every time a event happens ***count*** is reduced by 1. Once count reaches 0 latch is released.

*CountDownLatch’s  constructor >*

* **CountDownLatch**(int ***count***)

CountDownLatch is initialized with given ***count***.

***count*** specifies the number of events that must occur befor latch is released.

*CountDownLatch’s await() method has 2 forms :*

* void **await**( ) throws InterruptedException

Causes the current thread to wait until  one of the following things happens-

* latch ***count*** has down to reached 0, or
* unless the thread is interrupted.
* boolean **await**(long **timeout**, TimeUnit **unit**)

Causes the current thread to wait until  one of the following things happens-

* latch ***count*** has down to reached 0,
* unless the thread is interrupted, or
* specified **timeout** elapses.

*CountDownLatch’s countDown() method in java :*

* void **countDown**( )

Reduces latch ***count*** by 1.

If ***count*** reaches 0, all waiting threads are released.

Read more about [**CountDownLatch in java**](http://www.javamadesoeasy.com/2015/03/countdownlatch-in-java.html).

**Question 10. Where can you use CountDownLatch in real world?**

**Answer.**  Very interesting question. It will test your real time thread concurrency implementation skills. When you go in amusement park, you must have seen on certain rides there is mandate that at least 3 people (**3 is count**) should be there to take a ride. So, ride keeper (**ride keeper is main thread**) waits for 3 persons (**ride keeper has called await()**).

Every time a person comes count is reduced by 1 (**let’s say every person is calling countDown() method**). Ultimately when 3 persons reach count becomes 0 & wait for ride keeper comes to end.

Read more about [**CountDownLatch in java**](http://www.javamadesoeasy.com/2015/03/countdownlatch-in-java.html).

**Question 11. How can you implement your own CountDownLatch in java?**

**Answer. This is important and complex interview question for developers.** Please see [**Implementation of custom/own CountDownLatch in java**](http://www.javamadesoeasy.com/2015/03/implementation-of-customown_31.html).

**Question 12.What is CyclicBarrier in java?**

**Answer.**   This is very important question. Fresher and experienced bost be well versed with this. There might be situation where we might have to trigger event only when one or more threads completes certain operation in java.

**2 or more threads wait for each other to reach a common barrier point**. When all **threads** have **reached** common **barrier point** (i.e. when all threads have called await() method) >

* **All waiting threads are released**, and
* **Event can be triggered** as well.

[*CyclicBarrier’s*](http://www.javamadesoeasy.com/2015/03/cyclicbarrier-in-java.html) *constructor in java >*

* **CyclicBarrier(**int **parties)**

New CyclicBarrier is created where **parties** number of thread wait for each other to reach common barrier point, when all threads have reached common barrier point, **parties** number of waiting threads are released.

* **CyclicBarrier(**int **parties,** Runnable **barrierAction)**

New CyclicBarrier is created where **parties** number of thread wait for each other to reach common barrier point, when all threads have reached common barrier point, **parties** number of waiting threads are released and **barrierAction** (event)is triggered.

*CyclicBarrier’s await() method has 2 forms :*

* int **await**() throws InterruptedException, BrokenBarrierException

If the current thread is not the last to arrive(i.e. call await() method) then it waits until one of the following things happens -

* The last thread to call arrive(i,.e. call await() method), or
* Some other thread interrupts the current thread, or
* Some other thread interrupts one of the other waiting threads, or
* Some other thread times out while waiting for barrier, or
* Some other thread invokes reset() method on this cyclicBarrier.
* int **await**(long **timeout**, TimeUnit **unit**) throws InterruptedException, BrokenBarrierException, TimeoutException

If the current thread is not the last to arrive(i.e. call await() method) then it waits until one of the following things happens -

* The last thread to call arrive(i,.e. call await() method), or
* The specified **timeout** elapses, or
* Some other thread interrupts the current thread, or
* Some other thread interrupts one of the other waiting threads, or
* Some other thread times out while waiting for barrier, or
* Some other thread invokes reset() method on this cyclicBarrier.

Read more about [**CyclicBarrier in java**](http://www.javamadesoeasy.com/2015/03/cyclicbarrier-in-java.html).

**Question 13. Why is CyclicBarrier cyclic in java?**

**Answer.**  This is very interesting question for developers. The barrier is called *cyclic* because [CyclicBarrier](http://www.javamadesoeasy.com/2015/03/cyclicbarrier-in-java.html) can be reused after -

* All the waiting threads are released in java and
* event has been triggered in java.

**Question 14. Where could we use** [**CyclicBarrier**](http://www.javamadesoeasy.com/2015/03/cyclicbarrier-in-java.html) **in real world?**

**Answer.** Another very interesting question. It will test your real time thread concurrency implementation skills. Let’s say 10 friends (**friends are threads**) have planned for picnic on place A (Here **place A is common barrier** point). And they all decided to play certain game (**game is event**) only on everyones arrival at place A. So, all 10 friends must wait for each other to reach place A before launching event.

Now, when all **threads** have **reached** common **barrier point** (i.e. all friends have reached place A) >

* **All waiting threads are released**(All friends can play game), and
* **Event can be triggered** (they will start playing game).

**Question 15. How can you implement your own** [**CyclicBarrier**](http://www.javamadesoeasy.com/2015/03/cyclicbarrier-in-java.html) **in java?**

**Answer.  This is another important and complex interview for developers.** Please see [**Implementation of custom/own CyclicBarrier in java**](http://www.javamadesoeasy.com/2015/03/implementation-of-customown_39.html).

**Question 16. Similarity and Difference between CyclicBarrier and CountDownLatch in Java?**

**Answer.**   This is **very very important**  question. Fresher and experienced both must be well versed with this.

1. [**CyclicBarrier**](http://www.javamadesoeasy.com/2015/03/cyclicbarrier-in-java.html) and [**CountDownLatch**](http://www.javamadesoeasy.com/2015/03/countdownlatch-in-java.html) **are similar because** they wait for specified number of thread to reach certain point and make count/parties equal to 0. But,

for completing wait in CountDownLatch specified number of threads must call **countDown()** method in java.

for completing wait in CyclicBarrier specified number of threads must call **await()** method.

1. Let’ see there constructor’s >

|  |  |
| --- | --- |
| **CountDownLatch**(int ***count***)  CountDownLatch is initialized with given ***count***.  ***count*** specifies the number of events that must occur before latch is released. | **CyclicBarrier(**int **parties)**  New CyclicBarrier is created where **parties** number of thread wait for each other to reach common barrier point, when all threads have reached common barrier point, **parties** number of waiting threads are released. |

1. **CyclicBarrier** can be **awaited repeatedly**, but **CountDownLatch** can’t be awaited repeatedly. i.e. once count has become 0 cyclicBarrier can be used again but CountDownLatch cannot be used again in java.
2. **CyclicBarrier** can be used to trigger event, but **CountDownLatch** can’t be used to launch event. i.e. once count has become 0 cyclicBarrier can trigger event but CountDownLatch can’t in java.

**How can cyclicBarrier launch event?**

CyclicBarrier provides constructor for triggering event.

**CyclicBarrier(**int **parties,** Runnable **barrierAction)**

New CyclicBarrier is created where **parties** number of thread wait for each other to reach common barrier point, when all threads have reached common barrier point, **parties** number of waiting threads are released and **barrierAction (event) is triggered**.

**Question 17. What is Phaser in java? Is Phaser similar to CyclicBarrier?**

**Answer.** This is **another very important**  question. [Phaser](http://www.javamadesoeasy.com/2015/03/phaser-in-java_21.html) is somewhat **similar** in functionality of [CyclicBarrier](http://www.javamadesoeasy.com/2015/03/cyclicbarrier-in-java.html) and [CountDownLatch](http://www.javamadesoeasy.com/2015/03/countdownlatch-in-java.html) but it provides more flexibility than both of them.

Phaser provides us flexibility of registering and deRegistering parties at any time.

**For registering parties**, we may use any of the following -

* constructors, or
* int register(), or
* bulkRegister().

**For deRegistering parties**, we may use any of the following -

* arriveAndDeregister()

we have methods like **getPhase()** which returns the current phase number. And

**isTerminated()** method returns **true** if phaser has been **terminated**.

Read more about [**Phaser in java**](http://www.javamadesoeasy.com/2015/03/phaser-in-java_21.html)

**Question 18. Differences and similarity between Phaser and CyclicBarrier in java?**

**Answer.**  Another interesting question. Like a [**CyclicBarrier**](http://www.javamadesoeasy.com/2015/03/cyclicbarrier-in-java.html), a [Phaser](http://www.javamadesoeasy.com/2015/03/phaser-in-java_21.html) can be **awaited repeatedly**.

But, in CyclicBarrier we used to register parties in constructor but Phaser provides us flexibility of registering and deRegistering parties at any time in java.

**Question 19.Difference between arrive() and arriveAndAwaitAdvance() method of Phaser in java?**

**Answer.** This is question for experienced developers. **arrive**() method of [Phaser](http://www.javamadesoeasy.com/2015/03/phaser-in-java_21.html) **does not cause current thread to wait for other registered threads to complete current phase**. That means current thread can immediately start next phase without waiting for any other registered thread to complete current phase.

But, **arriveAndAwaitAdvance**() **method causes current thread to wait for other registered threads to complete current phase.** That means current thread can proceed to next phase only when all other threads have completed current phase (i.e. by calling **arriveAndAwaitAdvance() method**).

**Question 20. When is phaser terminated in java?**

**Answer.**  This is another question for experienced developers. When calling arriveAndDeregister() method of [Phaser](http://www.javamadesoeasy.com/2015/03/phaser-in-java_21.html) has caused the number of registered parties to become 0. Termination can also be triggered when an **onAdvance()** method returns **true**.

**Question 21. How can you control number of phase you want to execute in Phaser in java?**

**Answer.**  Yet another question for experienced developers. We can override the **onAdvance( )** method of [Phaser](http://www.javamadesoeasy.com/2015/03/phaser-in-java_21.html) to control number of phases which we want to execute.

Signature of onAdavance method is *boolean onAdvance(int* ***phase****, int* ***registeredParties****).*

Where, **phase** is the current phase number when we enter onAdvance() method i.e. before advancing to next phase.

**registeredParties** is the current number of registered parties

**Every Time before advancing to next phase overridden onAdvance() method is called** and returns either true or false.

If method returns **true** than **phaser is** **terminated** ,or

If method returns **false** then **phaser continues** and can **advance to next phase**.

[*Program to demonstrate usage of how we can* ***override*** *Phaser’s* ***onAdvance*** *method to control number of phase we want to execute*](http://www.javamadesoeasy.com/2015/03/phaser-in-java_21.html)

**Question 22. Where could we use** [**Phaser**](http://www.javamadesoeasy.com/2015/03/phaser-in-java_21.html) **in real world?**

**Answer.** Another interesting question. Software process management is done in phases.

* First phase could be **requirement gathering**,
* second could be **software development** and
* third could be **testing**.

Second phase will not start until first is not completed, like wise third phase will not start until second is not completed.

**Question 23. What is maximum number of parties that could be registered with phaser at a time in java ?**

**Answer.** Complex and challenging interview question even for experienced. Maximum number of parties that could be registered with phaser at a time is **65535**, if we try to register more parties **IllegalStateException** will be thrown in java.

**Question 24. What is exchanger in Java?**

**Answer.**  This is very important question for freshers and experienced. [Exchanger](http://www.javamadesoeasy.com/2015/03/exchanger-in-java.html) enables two threads to exchange their data between each other. Exchanger can be handy in solving Producer Consumer pattern where Producer and consumer threads can exchange their data.

* **exchange**(V **x**)

exchange() method enables two threads to exchange their data between each other.

**If current thread is first one to call exchange()** method then it will until one of following things happen >

* Some other thread calls exchange() method, or
* Some other thread interrupts the current thread, or

**If some other thread has already called exchanger()** method then it resumes its execution and following things happen -

* waiting thread is resumed and receives data from current thread.
* current thread receives data from that waiting thread and it returns immediately.
* V **exchange**(V **x**, long **timeout**, TimeUnit **unit**)

exchanger() method enables two threads to exchange their data between each other.

**If current thread is first one to call exchange()** method then it will until one of following things happen >

* Some other thread calls exchange() method, or
* Some other thread interrupts the current thread, or
* The specified **timeout** elapses.

**If some other thread has already called exchanger()** method then it resumes its execution and following things happen -

* waiting thread is resumed and receives data from current thread.
* current thread receives data from that waiting thread and it returns immediately.

Read more about [**Exchanger in java**](http://www.javamadesoeasy.com/2015/03/exchanger-in-java.html).

**Question 25. How can you implement Producer Consumer pattern using Exchanger in java?**

**Answer.**  Very interesting question for experienced developers. Exchanger is created,  which will enable Producer and consumer threads to exchange their data.

Producer thread produces and called exchanger() method, now it will wait for consumer thread to call exchange() method.

Consumer thread calls exchanger() method and following things will happens >

* current thread(consumerThread) will receive data from that waiting thread(producerThread) and it returns immediately.
* waiting thread (producerThread) will resume and receive data from current thread (consumerThread).

[Read program to implement Producer Consumer pattern using Exchanger](http://www.javamadesoeasy.com/2015/03/exchanger-in-java.html)

**Question 26. How can you solve consumer producer pattern by using BlockingQueue in java?**

**Answer.** It is **very very important** question for all developers.. Now it’s time to gear up to face question which is most probably going to be followed up by previous question i.e. after how to solve consumer producer problem using [wait() and notify() method](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html). Generally you might wonder why interviewer's are so much interested in asking about [solving consumer producer problem using BlockingQueue](http://www.javamadesoeasy.com/2015/03/solve-consumer-producer-problem-by.html), answer is they want to know how strong knowledge you have about java concurrent Api’s, this Api use consumer producer pattern in very optimized manner, BlockingQueue is designed is such a manner that it offer us the best performance.

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

Key methods for solving consumer producer pattern are >

|  |
| --- |
| **put(i);**   //used by producer to put/produce in sharedQueue.  **take();** //used by consumer to take/consume from sharedQueue. |

**Question 27. How can you implement your own LinkedBlockingQueue to solve consumer producer pattern in java?**

**Answer.**  Another challenging, logical and complex question, Please read [Producer Consumer pattern using Custom implementation of BlockingQueue interface](http://www.javamadesoeasy.com/2015/03/producer-consumer-pattern-using-custom.html)

**Question 28. What is Lock in java?**

**Answer.** Important question. The java.util.concurrent.locks.**Locks** is a  interface and its implementations provide more extensive locking operations than can be obtained using synchronized methods and statements.

**A lock helps in controlling access to a shared resource by multiple threads. Only one thread at a time can acquire the lock and access the shared resource in java.**

If a second thread attempts to acquire the lock on shared resource when it is acquired by another thread, the second thread will wait until the lock is released. In this way we can achieve [synchronization](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html) and [race conditions](http://www.javamadesoeasy.com/2015/03/race-condition-in-multithreading-and.html) can be avoided in java.

Read lock of a ReadWriteLock may allow concurrent access to a shared resource in java.

Read more about [**locks** and **ReEntrantLocks** in java](http://www.javamadesoeasy.com/2015/03/reentrantlocks-in-java.html)

**Question 29. Explain key methods of Lock interface in java?**

**Answer.**  This is very important question for freshers and experienced.

[*Lock interface*](http://www.javamadesoeasy.com/2015/03/reentrantlocks-in-java.html) *key methods in java >*

*void lock()*

Acquires the lock if it is not held by another thread. And sets **lock hold count** to 1.

If current thread already holds lock then **lock hold count** is increased by 1.

If the lock is held by another thread then the current thread waits for another thread to release lock.

*void unLock()*

If the current thread is the holding the lock then the **lock hold count** is decremented by 1. If the **lock hold count** has reached 0, then the lock is released.

If **lock hold count** is still greater than 0 then lock is not released.

If the current thread is not holding the lock then IllegalMonitorStateException is thrown.

*boolean tryLock()*

Acquires the lock if it is not held by another thread and returns true. And sets **lock hold count** to 1.

If current thread already holds lock then method returns true. And increments **lock hold count** by 1.

If lock is held by another thread then method return false.

*boolean tryLock(long timeout, TimeUnit unit)*

*throws InterruptedException*

Acquires the lock if it is not held by another thread and returns true.  And sets **lock hold count** to 1.

If current thread already holds lock then method returns true. And increments **lock hold count** by 1.

If lock is held by another thread then current thread will wait until one of the following things happen -

* **Another thread releases lock and the lock is acquired by the current thread, or**
* **Some other thread interrupts the current thread, or**
* **The specified timeout elapses .**

**If** the **lock is acquired** then method **returns true.** And sets **lock hold count** to 1.

**If specified timeout elapses then** method return false.

*Condition newCondition()*

Method returns a Condition instance to be used with this Lock instance.

Condition instance are similar to using [**Wait(), notify() and notifyAll()**](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) methods.

* IllegalMonitorStateException is thrown **if this lock is not held when** any of the **Condition waiting** or **signalling methods** are called.
* **Lock is released** when the **condition waiting methods are called** and before they return, the lock is reacquired and the **lock hold count** restored to what it was when the method was called.
* If a **thread is interrupted while waiting** then InterruptedException will be thrown and following things will happen -
  + the **wait will be over**, and
  + **thread's interrupted status will be cleared**.
* Waiting threads are signalled in FIFO (first in first out order) order.
* When lock is  ***fair***, first lock is obtained by longest-waiting thread.

If lock is not  ***fair***, any waiting thread could get lock, at discretion of implementation.

Read more about [**locks** in java](http://www.javamadesoeasy.com/2015/03/reentrantlocks-in-java.html)

**Question 30. Explain usage of newCondition() method of Lock interface in detail in java? And can it be used to implement producer consumer pattern in java?**

### Answer.  It is very complex question.Even most of the experienced developers are not aware of this question. Please read [ReentrantLock class provides implementation of Lock’s newCondition() method in java - description and solving producer consumer program using this method.](http://www.javamadesoeasy.com/2015/03/reentrantlock-class-provides.html)

**Question 31. Explain key methods of ReentrantLock class in java?**

**Answer.** This is very important question for freshers and experienced.

[ReentrantLock class](http://www.javamadesoeasy.com/2015/03/reentrantlocks-in-java.html) provides implementation of all Lock interface methods

*void lock()*

*void unLock()*

*boolean tryLock()*

*boolean tryLock(long timeout, TimeUnit unit)*

**Additional methods provided by** ReentrantLock class are >

*void lockInterruptibly() throws InterruptedException*

If current thread already holds lock then method returns true. And increments **lock hold count** by 1.

If the lock is held by another thread then the current thread waits until one of the following thing happens -

* The lock is acquired by the current thread, or
* Some other thread **interrupts the current thread**.

As soon as current thread acquires the lock it sets **lock hold count** to 1.

*int getWaitQueueLength(Condition condition)*

Method returns number of threads that may be waiting to acquire this lock.

Method is used just for monitoring purposes and not for any kind of synchronization purposes.

*boolean isHeldByCurrentThread()*

Method returns true if lock is held by current thread. Its similar to **Thread.holdsLock()** method.

Read more about [**ReEntrantLocks** in java](http://www.javamadesoeasy.com/2015/03/reentrantlocks-in-java.html)

**Question 32. Write Program to demonstrate usage of ReentrantLock in java?**

**Answer.**   Interesting question for developers. [Read program  to demonstrate usage of ReentrantLock](http://www.javamadesoeasy.com/2015/03/reentrantlocks-in-java.html).

**Question 33. How can you implement your own ReentrantLock in java?**

**Answer. This is important and complex interview question for developers.** [**Implementation of custom/own Lock and ReEntrantLock in java**](http://www.javamadesoeasy.com/2015/03/implementation-of-customown-lock-and.html)

**Question 34. What is Fork/Join Framework in java ?**

**Answer.**  This is **very very important**  question. Fresher and experienced both must be well versed with this..

*Fork/Join Framework has been added in* JDK 7 and is defined in the **java.util.concurrent** package in java.

*Fork/Join framework enables* ***parallel programming***. Parallel programming means taking **advantage two or more processors (multicore) in computers**.  Parallel programming improves program performance in java.

*The Fork/Join Framework also* ***improves program performance*** *in following ways >*

* Fork/Join framework makes use of multiple processors available in computer. Hence enabling parallel processing, and
* It managing whole [life cycle of Threads](http://www.javamadesoeasy.com/2015/03/thread-states-thread-life-cycle-in-java.html).

**Question 35. What is Divide-and-conquer in Fork/Join framework in java ?**

**Answer.** Basic question, which tests the developers in depth knowledge about the fork join framework in java. The **divide-and-conquer** strategy recursively divides a task into smaller subtasks until  subtask isn’t small enough to be solved independently.

**Question 36. What approach does ForkJoinPool uses for managing tasks in java?**

**Answer.** Another basic question, which tests the developers in depth knowledge about the fork join framework in java. [**ForkJoinPool**](http://www.javamadesoeasy.com/2015/03/forkjoin-framework-parallel-programming.html)uses ***work-stealing approach*** *for managing threads*. Each thread in ForkJoinPool maintains a queue of tasks. If one thread’s queue is empty, it can take task from another thread. This overall improves the program/applications performance in java.

**Question 37. What are ForkJoinPool and ForkJoinTask in java?**

##### Answer.

##### [***ForkJoinPool in java***](http://www.javamadesoeasy.com/2015/03/forkjoin-framework-parallel-programming.html)

**ForkJoinPool** implements ExecutorService framework. The execution of **ForkJoinTasks** takes place within a **ForkJoinPool**, which also manages the execution of the tasks.

*ForkJoinPool constructors >*

* ***ForkJoinPool( )***
* Creates a pool in java.
* **level of parallelism = number of processors available in the system**
* ***ForkJoinPool****(int* ***parallelism****)*
* The ***parallelism*** is the **level of parallelism**. Its value must be greater than 0 and must not be more than number of processors in system.
* **level of parallelism** determines the number of threads that can execute simultaneously. As a number of threads are determined it also determines number of tasks that could be executed **parallely** in java.

*ForkJoinPool important methods in java >*

After you have created an instance of **ForkJoinPool**, you can start a task in a number of different ways. **The first task started is the main task. Main task begins subtasks that are also managed by the pool**. Different methods for starting tasks have been discussed below >

* *<T> T* ***invoke****(ForkJoinTask<T>* ***task****)*

This method starts the ***task*** and returns the result of the ***task***.  Calling code waits until method returns.

* *void* ***execute****(ForkJoinTask<?>* ***task****)*

The execute() method can be used to start a ***task*** without waiting for its completion.

This method starts the ***task***. Calling code continues its execution asynchronously and does not waits for method completion like in invoke method.

* ***submit****() method comes in 4 different forms.*

submit() method can also be used for submitting task.

* *int* ***getParallelism****()*

The method returns **level of parallelism** i.e. number of processors available in the system.

* *void* ***shutdown****()*

Initiates shutdown, previously submitted tasks are executed, but no new tasks will be accepted.

* *List<Runnable>* ***shutdownNow****()*
* attempts to stop all actively executing tasks,
* submitted tasks may or may not execute.
* awaiting tasks will never execute, and
* method cancels both existing and unexecuted tasks, so it returns empty list.

##### [**ForkJoinTask<V> in java**](http://www.javamadesoeasy.com/2015/03/forkjoin-framework-parallel-programming.html)

**ForkJoinTask is abstract class for tasks that run within a ForkJoinPool.**

**ForkJoinTask<V>** is an abstract class that defines a task that can be managed by a **ForkJoinPool**.

The *V* specifies the result type of the task in java.

**Threads managed by ForkJoinPool executes ForkJoinTasks**. Small number of threads are used to serve large number of tasks.

*ForkJoinTask* ***important methods*** *>*

**ForkJoinTask** core methods are **fork( )** and **join()**

* *ForkJoinTask<V>* ***fork( )*** The fork( ) method **submits the task** for asynchronous execution, means that the thread that calls fork( ) method to submit task continues to run. Task are executed in the compute() method, which is running within a ForkJoinPool.
* *V* ***join( )***

The **join( )** method waits for task completion on which it is called. The method returns result of the task.

* *In short, about* ***fork( )*** *and* ***join( )*** are used for starting one or more new tasks and then wait for them to complete.
* *V* ***invoke****( )*

**The invoke() method combines the functionally of fork() and join()** methods. invoke() submits the task and waits for completion of submitted task.

The method returns result of task.

* *static void* ***invokeAll****(ForkJoinTask<?>* ***t1****, ForkJoinTask<?>* ***t2****)*

invokeAll() method  submits ***t1*** and ***t2*** and waits for completion of ***t1*** and ***t1***.

* *static void* ***invokeAll****(ForkJoinTask<?> …* ***tasks****)*

invokeAll() method submits list of tasks i.e. ***tasks*** and waits for completion of all tasks in list.

The ***invokeAll*( )** method can only be called from within the overridden compute() method of another **ForkJoinTask**, which is running within a **ForkJoinPool**.

*Some* ***other important methods*** *for checking status of submitted task -*

* *boolean* ***isDone****()* method returns true if a task completes.
* *boolean* ***isCompletedNormally****()* method returns true if a task completed normally without cancellation or without throwing any exception.
* *boolean* ***isCompletedAbnormally****()* returns true if a task completed abnormally either by cancellation or by throwing any exception.
* *boolean* ***isCancelled****()* returns true if the task was cancelled.

**Question 38. Similarity and Difference between RecursiveAction and RecursiveTask in java?**

**Answer.** Another important question.

##### ***Difference between RecursiveAction and RecursiveTask in java***

|  |  |
| --- | --- |
| *RecursiveAction* | *RecursiveTask<V>* |
| This **submits a task** and **does not return a result in java**. | This **submits a task** and **returns a result in java**. |
| Definition of compute method  *protected abstract void* ***compute()*** | *protected abstract V* ***compute()*** The *V* specifies the result type of the task. |

##### ***Similarity between RecursiveAction and RecursiveTask***

> Both **extends ForkJoinPool**.

> All **computations by tasks are performed inside compute() method**.

**Question 39. How can we use Fork/Join framework in real world?**

**Answer.** This question will test your real time thread concurrency implementation skills. We can use Fork/Join framework for calculating sum of array of 100000 or even may be more numbers. *Fork/Join framework uses* **divide-and-conquer** strategy for *enabling* ***parallel programming***. Divide-and-conquer strategy recursively divides a array into smaller subarrays until  subarray isn’t small enough to be solved independently.

Also, **ForkJoinPool** uses ***work-stealing approach*** *for managing threads*. Each thread in **ForkJoinPool** maintains a queue of tasks. If one thread’s queue is empty, it can take task from another thread. This overall improve the programs performance. Please see [program](http://www.javamadesoeasy.com/2015/03/forkjoin-framework-parallel-programming.html) to calculate sum of array of 100000 numbers.

/\*\* \*/

**Question 40. Difference between synchronized and ReentrantLock in java?**

**Answer.**   This is another **very very important**  question. Fresher and experienced both must be well versed with this.

|  |  |
| --- | --- |
| ***synchronized in java*** | ***ReentrantLock in java*** |
| **Does not provide any** **fair locks in java**. | **provides fair locks**, when lock is fair - first lock is obtained by longest-waiting thread in java.  Constructor to provide fairness -  ***ReentrantLock****(boolean* ***fair****)*  Creates an instance of ReentrantLock.  When ***fair*** is set true, first lock is obtained by longest-waiting thread in java.  If  ***fair*** is set false, any waiting thread could get lock, at discretion of implementation in java. |
| **Does not provide tryLock() method or its functionality**. Thread always waits for lock in java. | **Provide tryLock() method. If lock is held by another thread then method return false in java.**  *boolean tryLock()*  Acquires the lock if it is not held by another thread and returns true. And sets **lock hold count** to 1.  If current thread already holds lock then method returns true. And increments **lock hold count** by 1.  If lock is held by another thread then method return false in java. |
| There is **no method for lock interruptibility**, though current thread waits until one of the following thing happens -   * The **lock is acquired** **by** the **current thread in java**, or * Some other thread **interrupts the current thread in java**. | *void lockInterruptibly()*  If current thread already holds lock then method returns true. And increments **lock hold count** by 1.  If the lock is held by another thread then the current thread waits until one of the following thing happens -   * The **lock is acquired** **by** the **current thread in java**, or * Some other thread **interrupts the current thread**.   As soon as current thread acquires the lock it sets **lock hold count** to 1. |
| **Does not provide any method to return number of threads that may be waiting to acquire this lock in java**. | provide *int getQueueLength()* method to return number of threads that may be waiting to acquire this lock in java. |
| **holdsLock()** method is used to **find out whether lock is held by current thread or not**. If current thread holds lock method returns true in java. | *isHeldByCurrentThread()*method is **used to find out whether lock is held by current thread or not**. If current thread holds lock method returns true in java. |
| Thread can hold lock on object monitor only once in java. | if current thread **already holds lock** then **lock hold count** is increased by 1 when lock() method is called.  method to maintain **lock hold count** -  *void lock()*  Acquires the lock if it is not held by another thread. And sets **lock hold count** to 1.  If current thread already holds lock then **lock hold count** is increased by 1. |
| Does not provide any new condition() method in java. | provides *newCondition()* method.  Method returns a Condition instance to be used with this Lock instance.  Condition instance are similar to using [**Wait(), notify() and notifyAll()**](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) methods on object.   * IllegalMonitorStateException is thrown **if this lock is not held when** any of the **Condition waiting** or **signalling methods** are called. * **Lock is released** when the **condition waiting methods are called** and before they return, the lock is reacquired and the **lock hold count** restored to what it was when the method was called. * If a **thread is interrupted while waiting** then InterruptedException will be thrown and following things will happen -   + the **wait will be over**, and   + **thread's interrupted status will be cleared**. * Waiting threads are signalled in FIFO (first in first out order) order in java. * When lock is  ***fair***, first lock is obtained by longest-waiting thread in java.   If lock is not  ***fair***, any waiting thread could get lock, at discretion of implementation in java. |

**Question 41. Difference between traditional** [**multithreading**](http://www.javamadesoeasy.com/2015/03/what-is-thread-in-java.html) **and parallel programming in java?**

**Answer.** Basic interview question. Not a important one. MultiThreading primarily was designed to work with single CPU and utilize idle time of CPU. If two or more processors are there multithreading won’t be able to utilize multi processors but parallel programing using Fork/Join framework can utilize multiple processors available in computer in java.

**Question 42. Explain atomic operations in java?**

### Answer. Developers must have knowledge of atomic operations in thread concurrency java. Java provides some classes in java.util.concurrent.atomic which offers an alternative to the other [synchronization](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html) in java.

**Question 43. What is AtomicInteger in java? Explain key methods of AtomicInteger?**

**Answer.** Another very important question for freshers and experienced. *AtomicInteger  provides you with* ***int value*** *that is updated atomically. i.e. we can use these classes without any explicit synchronization in* [*multithreading*](http://www.javamadesoeasy.com/2015/03/what-is-thread-in-java.html) *environment, because any operation done on these classes is* [*thread safe*](http://www.javamadesoeasy.com/2015/03/guidelines-to-thread-safe-code-most.html) *in java.*

[*AtomicInteger*](http://www.javamadesoeasy.com/2015/03/atomicinteger-in-java.html) *important Methods >*

* *int* ***get****()*

method returns the current value

* *void* ***set****(int newValue)*

Sets to **newValue.**

* *int* ***getAndSet****(int newValue)*

Sets to **newValue** and returns the old value.

* *boolean* ***compareAndSet****(int expect, int update)*

Compare with *expect*, if equal, set to *update* and return true.

*Addition methods >*

* *int* ***addAndGet****(int value)*

adds *value* to the current value. And **return updated value.**

* *int* ***incrementAndGet****()*

increments current value by 1. And **return updated value.**

* *int* ***getAndAdd****(int value)*

Method **return current value**. And adds *value* to the current value.

* *int* ***getAndIncrement****()*

Method **return current value**. And increments current value by 1.

*Subtraction methods >*

* *int* ***decrementAndGet****()*

decrements current value by 1. And **return updated value.**

* *int* ***getAndDecrement****()*

Method **return current value**. And decrements current value by 1.

**Question 44. How can you implement your own AtomicInteger in java?**

**Answer.** Another important and complex interview question for developers. Please see

**/\*\***

**\***

**\* AtomicIntegerCustom provides you with int value that is updated atomically.**

**\*/**

**class AtomicIntegerCustom{**

**int currentValue;**

**int previousValue;**

**//AtomicintCustom constructors >**

**/\*\***

**\* Creates a new AtomicInteger and is initialized to 0.**

**\*/**

**public AtomicIntegerCustom(){**

**currentValue=0;**

**}**

**/\*\***

**\* Creates a new AtomicIntegerCustom and is initialized to specified initialValue.**

**\* @param initialValue**

**\*/**

**public AtomicIntegerCustom(int initialValue){**

**currentValue=initialValue;**

**}**

**//AtomicIntegerCustom  important Methods >**

**/\*\***

**\* method returns the current value**

**\***

**\*/**

**public synchronized int get(){**

**return currentValue;**

**}**

**/\*\***

**\* Sets to newValue.**

**\*/**

**public synchronized void set(int newValue){**

**currentValue=newValue;**

**}**

**/\*\***

**\* Sets to newValue and returns the old value.**

**\*/**

**public synchronized int getAndSet(int newValue){**

**previousValue=currentValue;**

**currentValue=newValue;**

**return previousValue;**

**}**

**/\*\***

**\* Compare with expect, if equal, set to update and return true.**

**\*/**

**public synchronized boolean compareAndSet(int expect, int update){**

**if(currentValue == expect){**

**currentValue=update;**

**return true;**

**}**

**else**

**return false;**

**}**

**//Addition methods >**

**/\*\***

**\* adds value to the current value. And return updated value.**

**\*/**

**public synchronized int addAndGet(int value){**

**return currentValue+=value;**

**}**

**/\*\***

**\* increments current value by 1. And return updated value.**

**\*/**

**public synchronized int incrementAndGet(){**

**return ++currentValue;**

**}**

**/\*\***

**\* Method return current value. And adds value to the current value.**

**\*/**

**public synchronized int getAndAdd(int value){**

**previousValue=currentValue;**

**currentValue+=value;**

**return previousValue;**

**}**

**/\*\***

**\* Method return current value. And increments current value by 1.**

**\***

**\*/**

**public synchronized int getAndIncrement(){**

**return currentValue++;**

**}**

**//Subtraction methods >**

**/\*\***

**\* decrements current value by 1. And return updated value.**

**\*/**

**public synchronized int decrementAndGet(){**

**return --currentValue;**

**}**

**/\*\***

**\* Method return current value. And decrements current value by 1.**

**\*/**

**public synchronized int getAndDecrement(){**

**return currentValue--;**

**}**

**@Override**

**public String toString() {**

**return "AtomicIntegerCustom = " + currentValue ;**

**}**

**}**

**Question 45. What is AtomicLong? Explain key methods of AtomicLong in java?**

**Answer.** This is another important question for freshers and experienced.AtomicLong  provides you with **long value** that is updated atomically. i.e. we can use these classes without any explicit synchronization in [multithreading](http://www.javamadesoeasy.com/2015/03/what-is-thread-in-java.html) environment, because any operation done on these classes is [thread safe](http://www.javamadesoeasy.com/2015/03/guidelines-to-thread-safe-code-most.html).

[*AtomicLong*](http://www.javamadesoeasy.com/2015/03/atomiclong-in-java.html) *important Methods >*

* *long* ***get****()*

method returns the current value

* *void* ***set****(long newValue)*

Sets to **newValue.**

* *long* ***getAndSet****(long newValue)*

Sets to **newValue** and returns the old value.

* *boolean* ***compareAndSet****(long expect, long update)*

*Addition methods >*

* *long* ***addAndGet****(long value)*

adds *value* to the current value. And **return updated value.**

* *long* ***incrementAndGet****()*

increments current value by 1. And **return updated value.**

* *long* ***getAndAdd****(long value)*

Method **return current value**. And adds *value* to the current value.

* *long* ***getAndIncrement****()*

Method **return current value**. And increments current value by 1.

*Subtraction methods >*

* *long* ***decrementAndGet****()*

decrements current value by 1. And **return updated value.**

* *long* ***getAndDecrement****()*

Method **return current value**. And decrements current value by 1.

**Question 46. How can you implement your own AtomicLong in java?**

**Answer.** Another important and complex interview question for developers. Please see [**Implementation of custom/own AtomicLong in java**](http://www.javamadesoeasy.com/2015/03/implementation-of-customown-atomiclong.html).

/\*\* \*/

**Question 47. What will be the output of below question in java?**



**Answer.**

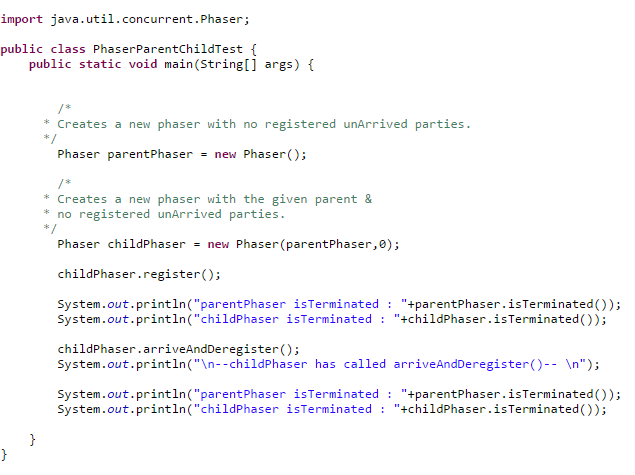
Thread-1 is Waiting to acquire lock

Thread-2 is Waiting to acquire lock

Thread-1 has acquired lock.

Thread-2 didn't got lock.

**Question 48. What will be the output of below question in java?**



**Answer.**

parentPhaser isTerminated : false

childPhaser isTerminated : false

--childPhaser has called arriveAndDeregister()--

parentPhaser isTerminated : true

childPhaser isTerminated : true

**Question 49. Which  atomic classes are available and which are not available in java 7? And why jdk developers didn’t created those classes?**

**Answer.** This is confusing question for freshers.

Java provides following classes in [**java.util.concurrent.atomic**](http://www.javamadesoeasy.com/2015/03/atomic-operations-in-java.html) >

*Classes found in* ***java.util.concurrent.atomic*** *are >*

* **AtomicInteger**,
* **AtomicLong,** and
* **AtomicBoolean.**

*Classes* ***NOT*** *found in* ***java.util.concurrent.atomic*** *are >*

* **AtomicByte,**
* **AtomicShort,**
* **AtomicFloat,**
* **AtomicDouble and**
* **AtomicCharacter**