**Multi-threading**

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

A thread in Java is a lightweight process that runs within another process or thread.

It is an independent path of execution in an application. JVM gives each thread its own method-call stack.

When we start JVM, Java starts one thread. This thread calls the main method of the class passed in argument to java call.

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 is the purpose of join() method in Thread class?**

In Java, Thread Scheduler controls thread scheduling. But we can use join() method on a thread to make current thread to wait for another thread to finish.

When we use join(), the current thread stops executing. It wait for the thread on which join() is called to finish.

This makes sure that current thread will continue only after the thread it joined finished running. Consider following example:

Public class ThreadJoin {

Thread importantThread = new Thread( new Runnable() {

public void run () {

//do something

}

}

);

Thread currentThread = new Thread(

new Runnable() {

public void run () {

//do something

}

}

);

importantThread.start(); // Line 1

importantThread.join(); // Line 2

currentThread.start(); // Line 3

}

In the above example, main thread is executing. On Line 1, a new thread called importantThread is ready to run. But at Line 2, main

thread joins the importantThread. Now it lets importantTread to finish and then it moves to Line 3. So currentThread at Line 3 will not start till the importantThread has finished.

1. **What is the fundamental difference between wait() and sleep() methods?**

The main difference between wait() and sleep() is that wait is an Object level method, whereas sleep() is a static method in Thread class. A waiting thread can be woken up by another thread by calling notify() on the monitor which is being waited on. But a sleeping thread cannot be woken up.

A wait() and notify() has to happen within the same block that is synchronized on the monitor object.

When we call wait() the current thread releases the monitor and goes to waiting state. Then another thread calls notify() to wake it up.

In case of sleep() current thread does not release the monitor or locks. It just sleeps for some pre-defined time period.

1. **Is it possible to call run() method instead of start() on a thread in Java?**

Yes. We can call run() method of a thread. But it does not work as a separate thread. It will just work as a normal object in main thread and there will not be context switching between the threads.

1. **How Multi-threading works in**

**Java?**

Java provides support for Multithreading. In a Multithreading environment, one process can execute multiple threads in parallel at the same time.

In Java, you can create process and then create multiple threads from that process. Each process can execute in parallel to perform independent tasks.

Java provides methods like- start(), notify(), wait(), sleep() etc. to maintain a multi-threading environment.

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.

1. **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.

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

In Java, a thread is a lightweight process that runs within another process or thread. It is an independent path of execution in an application. Each thread runs in a separate stack frame.

By default Java starts one thread when the main method of a class is called.

1. **What is a Thread’s priority and how it is used in scheduling?**

In Java, every Thread has a priority. This priority is specified as an integer value. The priority value is used in scheduling to pick up the thread with higher priority for execution. The threads with higher priority get more preference in execution than the threads with lower priority.

The task scheduler schedules the higher priority threads first, followed by the lower priority threads.

1. **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. **Is it possible to call run() method instead of start() on a thread in Java​?**

Yes. We can call run() method of a thread. But it does not work as a separate thread. It will just work as a normal object in main thread and there will not be context-switching between the threads.

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 synchronization in**

**Java?**

The concept of Synchronization in Java is used in Multi-threading programming.

It is a feature in Java that helps in controlling the access of multiple threads to a shared resource.

It is used to prevent Deadlock between multiple threads.

1. **What is the purpose of Synchronized block in Java?**

Synchronized block has many uses in Java multi-threading environment. Some of the uses are:

It can prevent thread interference

It is also used to avoid memory inconsistency issues

In general, scope of synchronized block is smaller than the scope of a method.

**271. What is static synchronization?**

We can make a static method as synchronized in Java. Adding synchronized keyword to a static method can do this.

In static synchronization, the lock is on class not on object.

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 the meaning of concurrency?**

Concurrency is the ability of a program to execute several programs simultaneously. This is achieved by distributing computations over multiple CPU cores of a machine or even over different machines within the same network.

It can increase the speed of execution of the overall program in multi-processor or multi-core system.

1. **What is the main difference between process and thread?**

As such both process and thread are independent sequences of execution.

The main difference is that a thread runs in a shared memory space, where as a process runs in its own memory space.

A process runs the execution in an environment provided by the operating system. A process has its own set of private resources (e.g. memory, open files, etc.).

A thread lives within a process and shares the resources like-memory, open files etc. with the other threads of the same process.

This ability to share resources between different threads makes thread more suitable for tasks where performance is a significant factor.

1. **What is a process and thread in the context of Java?**

In Java, a process refers to the running of Java Virtual Machine (JVM). But a thread lives within a JVM and it can be created or stopped by the Java application at runtime.

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 group the thread belongs to

1. **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 Runnablestate.
3. **Blocked**: A thread waiting for a monitor lock is inBlocked state.
4. **Waiting**: A thread waiting indefinitely for another threadto perform a particular action is in Waiting state.
5. **Timed\_waiting**: A thread waiting for another thread toperform an action for up to a specified waiting time is in Timed\_waiting state.
6. **Terminated**: A thread that has exited is in Terminatedstate.

1. **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. **Why we should not stop a thread by calling its stop() method?**

The stop() method in Thread class is a deprecated method. Its use is not recommended.

When we call stop() method, the thread unlocks all monitors that it has acquired. If any locked object was in an inconsistent state, this state gets visible to all other threads.

It can cause unexpected behavior when other threads work on this inconsistent object.

So calling stop() method to stop a thread is not advisable.

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

There are two main ways to create a thread in Java.

1. **Extend Thread class**: We can extend java.lang.Threadclass and implement run() method. On calling start() method it will start a new thread.
2. **Implement Runnable interface**: We can implementjava.lang.Runnable interface and pass the implemented object to the constructor of java.lang.Thread class. On calling start() it will start a new thread.

1. **How can we stop a thread in the middle of execution in Java?**

We can use a volatile variable as an indicator to stop the thread.

We can create a volatile reference pointing to the current thread. This reference can be set to null by other threads to flag that the current thread should stop execution.

In following example threadStopper is the volatile reference that can be set as null in stopThread() method by other threads.

Sample code is as follows:

public static class MyThread extends Thread {

private volatile Thread threadStopper;

public void start() {

threadStopper = new Thread(this);

threadStopper.start();

}

public void stopThread() {

threadStopper = null;

}

public void run() {

Thread currThread = Thread.currentThread(); while(currThread == threadStopper) {

try {

Thread.sleep(100);

} catch (InterruptedException e) {

}

}

}

}

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. **What is Busy waiting in Multi-threading?**

Busy waiting is also known as busy-looping or spinning. It is a multi-threading technique in which a process repeatedly checks if a condition is true.

For example, a process can keep checking if any keyboard input is available.

In general, busy waiting is considered as Anti-pattern that wastes processor time, so it should be avoided.

Sample code for busy waiting is as follows:

Thread thread = new Thread(new Runnable() { @Override

public void run() {

long timeToStop = System.currentTimeMillis() + 1000; long currentTime = System.currentTimeMillis();

// Busy waiting

while (timeToStop > currentTime) { currentTime = System.currentTimeMillis(); }

}

});

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. **Can we wake up a thread that has been put to sleep by using Thread.sleep() method?**

We can use interrupt() method of java.lang.Thread class to interrupt

a thread that is in sleep state. It will get InterruptedException to

wake up from the sleep.

Sample code is as follows:

public class ThreadInterrupt implements Runnable { public void run() {

try {

Thread.sleep(Long.MAX\_VALUE);

} catch (InterruptedException e) { SOP(“Interrupted by exception!");

}

}

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

{

Thread myThread = new Thread(new ThreadInterrupt(), “myThread");

myThread.start();

SOP(“Sleeping in main thread for 10 seconds”);

Thread.sleep(10000);

SOP(“Interrupting myThread");

myThread.interrupt();

}

}

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.

1. **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

1. **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.

1. **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.

1. **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. **What is a Livelock?**

Livelock is a scenario in which two or more block each other by responding to an action caused by another thread.

In a deadlock situation two or more threads wait in one specific state.

In a Livelock scenario, two more threads change their state in such a way that it prevents progress on their regular work.

E.g. Consider scenario in which two threads try to acquire two locks. They release a lock that they have acquired, when they cannot acquire the second lock.

In a Livelock situation, both threads concurrently try to acquire the locks. Only one thread would succeed, the second thread may succeed in acquiring the second lock.

Now both threads hold two different locks. And both threads want to have both locks. So they release their lock and try again from the beginning. This situation keeps repeating multiple times..

1. **What is Thread starvation?**

In a priority based scheduling, Threads with lower priority get lesser time for execution than higher priority threads.

If a lower priority thread performs a long running computation, it may happen that this thread does not get enough time to finish its computations just in time. In such a scenario, the tread with lower priority would starve. It will remain away from the threads with higher priority.

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 the use of yield() method in Thread class?**

The yield() method of Thread class is used to give a hint to scheduler that the current thread wants to free the processor.

The scheduler can either use this hint or just ignore this hint. Since the scheduler behavior is not guaranteed, it may happen that the current thread again gets the processor time.

It can be used for debugging or testing purposes. But there is rarely any concrete use of 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:

Do not provide "setter" methods that modify fields or objects referred to by fields.

Make all fields final and private.

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.

If the instance fields include references to mutable objects, do not allow those objects to be changed.

Do not provide methods that modify the mutable objects.

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.

1. **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. **What is scalability in a Software program?**

Scalability is the capability of a program to handle growing amount of work or its potential to be enlarged in order to accommodate growth.

A program is considered scalable, if it is suitable to handle a large amount of input data or a large number of users or a large number of nodes.

When we say a program does not scale, it means that program fails on increasing the size of task.

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.

1. **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 are the uses of Future interface in Java?**

We can use Future interface to represent the result of an asynchronous computation.

These are the operations whose result is not immediately available.

Therefore Future interface provides isDone() method to check if the asynchronous computation has finished or not.

We can also check if the task was cancelled by calling isCancelled() method.

Future also provides cancel() method to attempt the cancellation of a task.

1. **What is the difference in concurrency in HashMap and in Hashtable?**

In a Hashtable class all methods are synchronized.

In a HashMap implementation all the methods are not synchronized.

Therefore Hashtable is a thread-safe collection. HashMap is not a thread-safe collection.

In a multi-threading it is not advisable to use regular HashMap. We can use ConcurrentHashMap class in multi-threading applications.

1. **How will you create synchronized instance of List or Map Collection?**

In Java, Collections class provides methods to synchronize any collection.

It also provides synchronizedList(List) and synchronizedMap(Map) methods that can be used to convert a List or Map to a synchronized instance.

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.

Where as, 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.