**Java Concurrency Interview Questions**

(Mohan\_oon)

What Is the Difference Between a Process and a Thread?

How Can You Create a Thread Instance and Run It?

Describe the Different States of a Thread and When Do the State Transitions Occur.

What Is the Difference Between the Runnable and Callable Interfaces? How Are They Used?

What Is a Daemon Thread, What Are Its Use Cases? How Can You Create a Daemon Thread?

What Is the Thread’s Interrupt Flag? How Can You Set and Check It? How Does It Relate to the Interruptedexception?

What Are Executor and Executorservice? What Are the Differences Between These Interfaces?

What Are the Available Implementations of Executorservice in the Standard Library?

What Is Java Memory Model (Jmm)? Describe Its Purpose and Basic Ideas.

What Is a Volatile Field and What Guarantees Does the Jmm Hold for Such Field?

Which of the Following Operations Are Atomic?

What Special Guarantees Does the Jmm Hold for Final Fields of a Class?

What Is the Meaning of a Synchronized Keyword in the Definition of a Method? of a Static Method? Before a Block?

If Two Threads Call a Synchronized Method on Different Object Instances Simultaneously, Could One of These Threads Block? What If the Method Is Static?

Describe the Conditions of Deadlock, Livelock, and Starvation. Describe the Possible Causes of These Conditions.

Describe the Purpose and Use-Cases of the Fork/Join Framework.

­\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**What Is the Difference Between a Process and a Thread?**

Both processes and threads are units of concurrency, but they have a fundamental difference: processes do not share a common memory, while threads do.

From the operating system's point of view, a process is an independent piece of software that runs in its own virtual memory space. Any multitasking operating system (which means almost any modern operating system) has to separate processes in memory so that one failing process wouldn't drag all other processes down by scrambling common memory.

The processes are thus usually isolated, and they cooperate by the means of inter-process communication which is defined by the operating system as a kind of intermediate API.

On the contrary, a thread is a part of an application that shares a common memory with other threads of the same application. Using common memory allows to shave off lots of overhead, design the threads to cooperate and exchange data between them much faster.

**How Can You Create a Thread Instance and Run It?**

To create an instance of a thread, you have two options. First, pass a Runnable instance to its constructor and call start(). Runnable is a functional interface, so it can be passed as a lambda expression:

Thread thread1 = new Thread(() ->

System.out.println("Hello World from Runnable!"));

thread1.start();

Thread also implements Runnable, so another way of starting a thread is to create an anonymous subclass, override its run() method, and then call start():

Thread thread2 = new Thread() {

@Override

public void run() {

System.out.println("Hello World from subclass!");

}

};

thread2.start();

**Describe the Different States of a Thread and When Do the State Transitions Occur.**

The state of a Thread can be checked using the Thread.getState() method. Different states of a Thread are described in the Thread.State enum. They are:

NEW — a new Thread instance that was not yet started via Thread.start()

RUNNABLE — a running thread. It is called runnable because at any given time it could be either running or waiting for the next quantum of time from the thread scheduler. A NEW thread enters the RUNNABLE state when you call Thread.start() on it

BLOCKED — a running thread becomes blocked if it needs to enter a synchronized section but cannot do that due to another thread holding the monitor of this section

WAITING — a thread enters this state if it waits for another thread to perform a particular action. For instance, a thread enters this state upon calling the Object.wait() method on a monitor it holds, or the Thread.join() method on another thread

TIMED\_WAITING — same as the above, but a thread enters this state after calling timed versions of Thread.sleep(), Object.wait(), Thread.join() and some other methods

TERMINATED — a thread has completed the execution of its Runnable.run() method and terminated

What Is the Difference Between the Runnable and Callable Interfaces? How Are They Used?

The Runnable interface has a single run method. It represents a unit of computation that has to be run in a separate thread. The Runnable interface does not allow this method to return value or to throw unchecked exceptions.

The Callable interface has a single call method and represents a task that has a value. That's why the call method returns a value. It can also throw exceptions. Callable is generally used in ExecutorService instances to start an asynchronous task and then call the returned Future instance to get its value.

**What Is a Daemon Thread, What Are Its Use Cases? How Can You Create a Daemon Thread?**

A daemon thread is a thread that does not prevent JVM from exiting. When all non-daemon threads are terminated, the JVM simply abandons all remaining daemon threads. Daemon threads are usually used to carry out some supportive or service tasks for other threads, but you should take into account that they may be abandoned at any time.

To start a thread as a daemon, you should use the setDaemon() method before calling start():

Thread daemon = new Thread(()

-> System.out.println("Hello from daemon!"));

daemon.setDaemon(true);

daemon.start();

Curiously, if you run this as a part of the main() method, the message might not get printed. This could happen if the main() thread would terminate before the daemon would get to the point of printing the message. You generally should not do any I/O in daemon threads, as they won't even be able to execute their finally blocks and close the resources if abandoned.

**What Is the Thread’s Interrupt Flag? How Can You Set and Check It? How Does It Relate to the Interruptedexception?**

The interrupt flag, or interrupt status, is an internal Thread flag that is set when the thread is interrupted. To set it, simply call thread.interrupt() on the thread object.

If a thread is currently inside one of the methods that throw InterruptedException (wait, join, sleep etc.), then this method immediately throws InterruptedException. The thread is free to process this exception according to its own logic.

If a thread is not inside such method and thread.interrupt() is called, nothing special happens. It is thread's responsibility to periodically check the interrupt status using static Thread.interrupted() or instance isInterrupted() method. The difference between these methods is that the static Thread.interrupted() clears the interrupt flag, while isInterrupted() does not.

**What Are Executor and Executorservice? What Are the Differences Between These Interfaces?**

Executor and ExecutorService are two related interfaces of java.util.concurrent framework. Executor is a very simple interface with a single execute method accepting Runnable instances for execution. In most cases, this is the interface that your task-executing code should depend on.

ExecutorService extends the Executor interface with multiple methods for handling and checking the lifecycle of a concurrent task execution service (termination of tasks in case of shutdown) and methods for more complex asynchronous task handling including Futures.

**What Are the Available Implementations of Executorservice in the Standard Library?**

The ExecutorService interface has three standard implementations:

ThreadPoolExecutor — for executing tasks using a pool of threads. Once a thread is finished executing the task, it goes back into the pool. If all threads in the pool are busy, then the task has to wait for its turn.

ScheduledThreadPoolExecutor allows to schedule task execution instead of running it immediately when a thread is available. It can also schedule tasks with fixed rate or fixed delay.

ForkJoinPool is a special ExecutorService for dealing with recursive algorithms tasks. If you use a regular ThreadPoolExecutor for a recursive algorithm, you will quickly find all your threads are busy waiting for the lower levels of recursion to finish. The ForkJoinPool implements the so-called work-stealing algorithm that allows it to use available threads more efficiently.

**What Is Java Memory Model (Jmm)? Describe Its Purpose and Basic Ideas.**

Java Memory Model is a part of Java language specification described in Chapter 17.4. It specifies how multiple threads access common memory in a concurrent Java application, and how data changes by one thread are made visible to other threads. While being quite short and concise, JMM may be hard to grasp without strong mathematical background.

The need for memory model arises from the fact that the way your Java code is accessing data is not how it actually happens on the lower levels. Memory writes and reads may be reordered or optimized by the Java compiler, JIT compiler, and even CPU, as long as the observable result of these reads and writes is the same.

This can lead to counter-intuitive results when your application is scaled to multiple threads because most of these optimizations take into account a single thread of execution (the cross-thread optimizers are still extremely hard to implement). Another huge problem is that the memory in modern systems is multilayered: multiple cores of a processor may keep some non-flushed data in their caches or read/write buffers, which also affects the state of the memory observed from other cores.

To make things worse, the existence of different memory access architectures would break the Java's promise of “write once, run everywhere”. Happily for the programmers, the JMM specifies some guarantees that you may rely upon when designing multithreaded applications. Sticking to these guarantees helps a programmer to write multithreaded code that is stable and portable between various architectures.

The main notions of JMM are:

Actions, these are inter-thread actions that can be executed by one thread and detected by another thread, like reading or writing variables, locking/unlocking monitors and so on

Synchronization actions, a certain subset of actions, like reading/writing a volatile variable, or locking/unlocking a monitor

Program Order (PO), the observable total order of actions inside a single thread

Synchronization Order (SO), the total order between all synchronization actions — it has to be consistent with Program Order, that is, if two synchronization actions come one before another in PO, they occur in the same order in SO

synchronizes-with (SW) relation between certain synchronization actions, like unlocking of monitor and locking of the same monitor (in another or the same thread)

Happens-before Order — combines PO with SW (this is called transitive closure in set theory) to create a partial ordering of all actions between threads. If one action happens-before another, then the results of the first action are observable by the second action (for instance, write of a variable in one thread and read in another)

Happens-before consistency — a set of actions is HB-consistent if every read observes either the last write to that location in the happens-before order, or some other write via data race

Execution — a certain set of ordered actions and consistency rules between them

For a given program, we can observe multiple different executions with various outcomes. But if a program is correctly synchronized, then all of its executions appear to be sequentially consistent, meaning you can reason about the multithreaded program as a set of actions occurring in some sequential order. This saves you the trouble of thinking about under-the-hood reorderings, optimizations or data caching.

**What Is a Volatile Field and What Guarantees Does the Jmm Hold for Such Field?**

A volatile field has special properties according to the Java Memory Model (see Q9). The reads and writes of a volatile variable are synchronization actions, meaning that they have a total ordering (all threads will observe a consistent order of these actions). A read of a volatile variable is guaranteed to observe the last write to this variable, according to this order.

If you have a field that is accessed from multiple threads, with at least one thread writing to it, then you should consider making it volatile, or else there is a little guarantee to what a certain thread would read from this field.

Another guarantee for volatile is atomicity of writing and reading 64-bit values (long and double). Without a volatile modifier, a read of such field could observe a value partly written by another thread.

**Which of the Following Operations Are Atomic?**

writing to a non-volatile int;

writing to a volatile int;

writing to a non-volatile long;

writing to a volatile long;

incrementing a volatile long?

A write to an int (32-bit) variable is guaranteed to be atomic, whether it is volatile or not. A long (64-bit) variable could be written in two separate steps, for example, on 32-bit architectures, so by default, there is no atomicity guarantee. However, if you specify the volatile modifier, a long variable is guaranteed to be accessed atomically.

The increment operation is usually done in multiple steps (retrieving a value, changing it and writing back), so it is never guaranteed to be atomic, wether the variable is volatile or not. If you need to implement atomic increment of a value, you should use classes AtomicInteger, AtomicLong etc.

**What Special Guarantees Does the Jmm Hold for Final Fields of a Class?**

JVM basically guarantees that final fields of a class will be initialized before any thread gets hold of the object. Without this guarantee, a reference to an object may be published, i.e. become visible, to another thread before all the fields of this object are initialized, due to reorderings or other optimizations. This could cause racy access to these fields.

This is why, when creating an immutable object, you should always make all its fields final, even if they are not accessible via getter methods.

**What Is the Meaning of a Synchronized Keyword in the Definition of a Method? of a Static Method? Before a Block?**

The synchronized keyword before a block means that any thread entering this block has to acquire the monitor (the object in brackets). If the monitor is already acquired by another thread, the former thread will enter the BLOCKED state and wait until the monitor is released.

synchronized(object) {

// ...

}

A synchronized instance method has the same semantics, but the instance itself acts as a monitor.

synchronized void instanceMethod() {

// ...

}

For a static synchronized method, the monitor is the Class object representing the declaring class.

static synchronized void staticMethod() {

// ...

}

**If Two Threads Call a Synchronized Method on Different Object Instances Simultaneously, Could One of These Threads Block? What If the Method Is Static?**

If the method is an instance method, then the instance acts as a monitor for the method. Two threads calling the method on different instances acquire different monitors, so none of them gets blocked.

If the method is static, then the monitor is the Class object. For both threads, the monitor is the same, so one of them will probably block and wait for another to exit the synchronized method.

**What Is the Purpose of the Wait, Notify and Notifyall Methods of the Object Class?**

A thread that owns the object's monitor (for instance, a thread that has entered a synchronized section guarded by the object) may call object.wait() to temporarily release the monitor and give other threads a chance to acquire the monitor. This may be done, for instance, to wait for a certain condition.

When another thread that acquired the monitor fulfills the condition, it may call object.notify() or object.notifyAll() and release the monitor. The notify method awakes a single thread in the waiting state, and the notifyAll method awakes all threads that wait for this monitor, and they all compete for re-acquiring the lock.

The following BlockingQueue implementation shows how multiple threads work together via the wait-notify pattern. If we put an element into an empty queue, all threads that were waiting in the take method wake up and try to receive the value. If we put an element into a full queue, the put method waits for the call to the get method. The get method removes an element and notifies the threads waiting in the put method that the queue has an empty place for a new item.

public class BlockingQueue<T> {

private List<T> queue = new LinkedList<T>();

private int limit = 10;

public synchronized void put(T item) {

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

try {

wait();

} catch (InterruptedException e) {}

}

if (queue.isEmpty()) {

notifyAll();

}

queue.add(item);

}

public synchronized T take() throws InterruptedException {

while (queue.isEmpty()) {

try {

wait();

} catch (InterruptedException e) {}

}

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

notifyAll();

}

return queue.remove(0);

}

}

Describe the Conditions of Deadlock, Livelock, and Starvation. Describe the Possible Causes of These Conditions.

Deadlock is a condition within a group of threads that cannot make progress because every thread in the group has to acquire some resource that is already acquired by another thread in the group. The most simple case is when two threads need to lock both of two resources to progress, the first resource is already locked by one thread, and the second by another. These threads will never acquire a lock to both resources and thus will never progress.

Livelock is a case of multiple threads reacting to conditions, or events, generated by themselves. An event occurs in one thread and has to be processed by another thread. During this processing, a new event occurs which has to be processed in the first thread, and so on. Such threads are alive and not blocked, but still, do not make any progress because they overwhelm each other with useless work.

Starvation is a case of a thread unable to acquire resource because other thread (or threads) occupy it for too long or have higher priority. A thread cannot make progress and thus is unable to fulfill useful work.

**Describe the Purpose and Use-Cases of the Fork/Join Framework.**

The fork/join framework allows parallelizing recursive algorithms. The main problem with parallelizing recursion using something like ThreadPoolExecutor is that you may quickly run out of threads because each recursive step would require its own thread, while the threads up the stack would be idle and waiting.

The fork/join framework entry point is the ForkJoinPool class which is an implementation of ExecutorService. It implements the work-stealing algorithm, where idle threads try to “steal” work from busy threads. This allows to spread the calculations between different threads and make progress while using fewer threads than it would require with a usual thread pool.