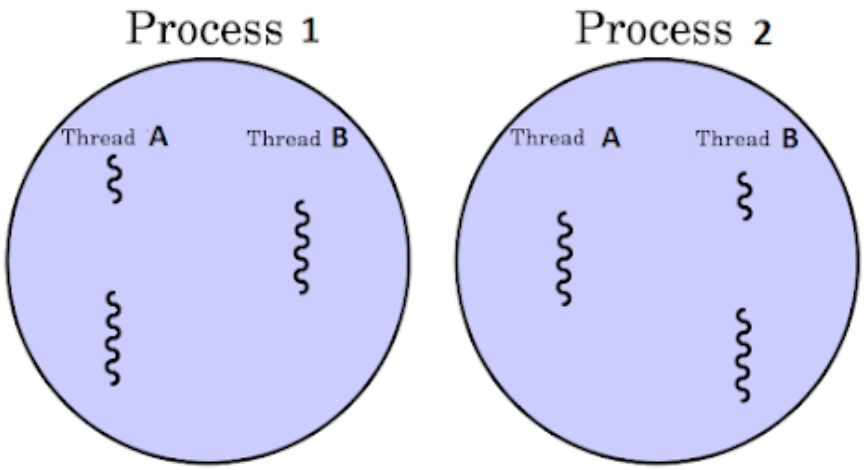
50 Interview questions from Java Multithreading and Concurrency with Answers

Here is our list of top questions from Java thread, concurrency, and multi-threading. You can use this list to prepare well for your Java interview.

1) What is Thread in Java? (answer)
The thread is an independent path of execution. It's a way to take advantage of multiple CPUs available on a machine. By employing multiple threads you can speed up CPU-bound tasks. For example, if one thread takes 100 milliseconds to do a job, you can use 10 threads to reduce that task into 10 milliseconds. Java provides excellent support for multithreading at the language level, and it's also one of the strong selling points.

2) What is the difference between Thread and Process in Java? (answer)
The thread is a subset of Process, in other words, one process can contain multiple threads. Two processes run on different memory spaces, but all threads share the same memory space. Don't confuse this with stack memory, which is different for the different threads and used to store local data to that thread. For more detail see the answer.



3) How do you implement Thread in Java? (answer)
At the language level, there are two ways to implement Thread in Java. An instance of `java.lang.Thread` represents a thread but it needs a task to execute, which is an instance of `interface java.lang.Runnable`.

Since the `Thread` class itself implement `Runnable`, you can override the `run()` method either by extending the Thread class or just implementing the Runnable interface. For a detailed answer and discussion see this article.

4) When to use Runnable vs Thread in Java? (answer)
This is a follow-up to a previous multi-threading interview question. As we know we can implement thread either by extending the `Thread` class or implementing `Runnable` interface, the question arises, which one is better and when to use one? This question will be easy to

answer if you know that the Java programming language doesn't support multiple inheritances of class, but it allows you to implement multiple interfaces.

This means it's better to implement `Runnable` than extend `Thread` if you also want to extend another class e.g. `Canvas` or `CommandListener`. For more points and discussion you can also refer to this post.

6) What is the difference between the start() and run() method of the Thread class? (answer)
One of trick Java questions from early days, but still good enough to differentiate between shallow understanding of Java threading model `start()` method is used to start a newly created thread, while `start()` internally calls `run()` method, there is difference calling `run()` method directly.

When you invoke `run()` as a normal method, it's called in the same thread, no new thread is started, which is the case when you call the `start()` method. Read this answer for a much more detailed discussion.

7) What is the difference between Runnable and Callable in Java? (answer)
Both Runnable and Callable represent task which is intended to be executed in a separate thread. Runnable is there from JDK 1.0 while Callable was added on JDK 1.5. The main difference between these two is that Callable's `call()` method can return value and throw Exception, which was not possible with Runnable's `run()` method. Callable return Future object, which can hold the result of the computation. See my blog post on the same topic for a more in-depth answer to this question.

8) What is the difference between CyclicBarrier and CountDownLatch in Java? (answer)
Though both CyclicBarrier and CountDownLatch wait for a number of threads on one or more events, the main difference between them is that you can not re-use `CountDownLatch` once the count reaches to zero, but you can reuse the same `CyclicBarrier` even after the barrier is broken. See this answer for a few more points and a sample code example.

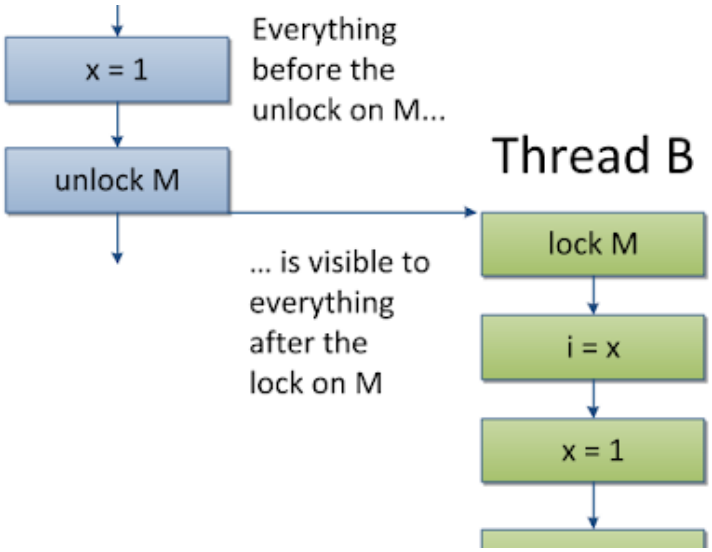
9) What is the Java Memory model? (answer)
Java Memory model is a set of rules and guidelines which allows Java programs to behave deterministically across multiple memory architecture, CPU, and operating systems. It's particularly important in the case of multi-threading.

Java Memory Model provides some guarantee on which changes made by one thread should be visible to others, one of them is the happens-before relationship. This relationship defines several rules which allow programmers to anticipate and reason the behavior of concurrent Java programs. For example, happens-before relationship guarantees :

- Each action in a thread happens-before every action in that thread that comes later in the program order, this is known as the program order rule.
- An unlock on a monitor lock happens before every subsequent lock on that same monitor lock, also known as the Monitor lock rule.

- A write to a volatile field happens before every subsequent read of that same field, known as the Volatile variable rule.
- A call to Thread.start on a thread happens-before any other thread detects that thread has terminated, either by successfully returning from `Thread.join()` or by `Thread.isAlive()` returning false, also known as Thread start rule.
- A thread calling `interrupt()` on another thread happens before the interrupted thread detects the interrupt(either by having InterruptedException thrown, or invoking interrupted or interrupted), popularly known as the Thread Interruption rule.
- The end of a constructor for an object happens before the start of the finalizer for that object is known as the Finalizer rule.
- If A happens before B, and B happens before C, then A happens-before C, which means happens-before guarantees Transitivity.

I strongly suggest reading Chapter 16 of Java Concurrency in Practice to understand the Java Memory model in more detail.



10) What is a volatile variable in Java? (answer)
volatile is a special modifier, which can only be used with instance variables. In concurrent Java programs, changes made by multiple threads on instance variables are not visible to others in absence of any synchronizers like synchronized keywords or locks.

Volatile variable guarantees that a write will happen before any subsequent read: as stated: *"volatile variable rule"* in the previous question. Read this answer to learn more about volatile variables and when to use them.

11) What is thread-safety? is Vector a thread-safe class? (Yes, see details)
Thread safety is a property of an object or code which guarantees that if executed or used by multiple threads in any manner e.g. read vs writing it will behave as expected. For example, a thread-safe counter object will not miss any count if the same instance of that counter is shared among multiple threads.

Apparently, you can also divide collection classes into two categories, thread-safe and non-thread-safe. Vector is indeed a thread-safe class and it achieves thread-safety by

synchronizing methods that modify the state of Vector, on the other hand, its counterpart `ArrayList` is not thread-safe.

12) What is a race condition in Java? Given one example? (answer)
Race conditions are caused by some subtle programming bugs when Java programs are exposed to a concurrent execution environment. As the name suggests, a race condition occurs due to race between multiple threads, if a thread that is supposed to execute first lost the race and is executed second, the behavior of code changes, which surface as non-deterministic bugs.

This is one of the hardest bugs to find and re-produce because of the random nature of racing between threads. One example of race conditions is out-of-order processing, see this answer for some more examples of race conditions in Java programs.

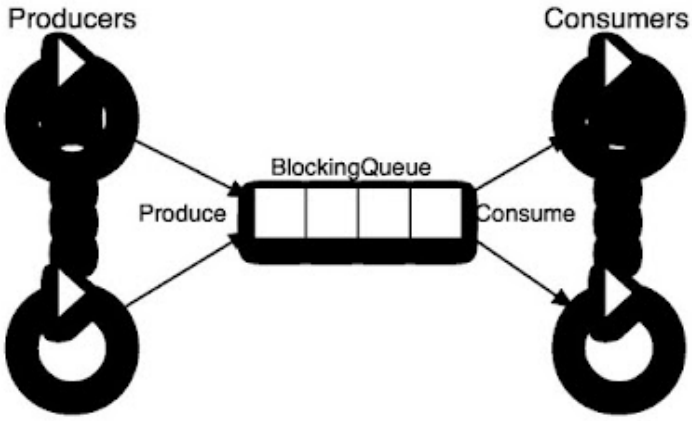
13) How to stop a thread in Java? (answer(answer)
I always said that Java provides rich APIs for everything but ironically Java doesn't provide a sure-shot way of stopping the thread. There were some control methods in JDK 1.0 e.g. `stop()`, `suspend()`, and `resume()` which were deprecated in later releases due to potential deadlock threats, then Java API designers have not made any effort to provide a consistent, thread-safe, and elegant way to stop threads.

Programmers mainly rely on the fact that thread stops automatically as soon as they finish execution of `run()` or `call()` method. To manually stop, programmers either take advantage of volatile boolean variables and check in every iteration if the run method has loops or interrupt threads to abruptly cancel tasks. See this tutorial for a sample code of stopping thread in Java.

14) What happens when an Exception occurs in a thread? (answer)
This is one of the good tricky Java questions I have seen in interviews. In simple words, If not caught thread will die, if an uncaught exception handler is registered then it will get a callback. `Thread.UncaughtExceptionHandler` is an interface, defined as a nested interface for handlers invoked when a Thread abruptly terminates due to an uncaught exception.

When a thread is about to terminate due to an uncaught exception the Java Virtual Machine will query the thread for its `UncaughtExceptionHandler` using `Thread.getUncaughtExceptionHandler()` and will invoke the handler's `uncaughtException()` method, passing the thread and the exception as arguments.

15) How do you share data between two threads in Java? (answer)
You can share data between threads by using shared objects, or concurrent data structures like BlockingQueue. See this tutorial to learn inter-thread communication in Java. It implements the Producer consumer pattern using wait and notify methods, which involves sharing objects between two threads.



16) What is the difference between notify and notifyAll in Java? (answer)
This is another tricky question from core Java interviews since multiple threads can wait on a single monitor lock, Java API designer provides a method to inform only one of them or all of them, once the waiting condition changes, but they provide the half implementation.

There `notify()` method doesn't provide any way to choose a particular thread, that's why it's only useful when you know that there is only one thread is waiting.

On the other hand, `notifyAll()` sends a notification to all threads and allows them to compete for locks, which ensures that at least one thread will proceed further. See my blog post on a similar topic for a more detailed answer and code example.

17) Why wait, notify, and notifyAll are not inside the thread class? (answer)
This is a design-related question, which checks what the candidate thinks about the existing system or does he ever thought of something which is so common but looks inappropriate at first. In order to answer this question, you have to give some reasons why it makes sense for these three methods to be in the Object class, and why not on Thread class.

One reason which is obvious is that Java provides lock at the object level, not at the thread level. Every object has a lock, which is acquired by a thread. Now if the thread needs to wait for a certain lock it makes sense to call wait() on that object rather than on that thread.

Had `wait()` method declared on Thread class, it was not clear for which lock thread was waiting. In short, since wait, notify and notifyAll operate at lock level, it makes sense to define it on object class because the lock belongs to object. You can also see this article for a more elaborate answer to this question.

18) What is the ThreadLocal variable in Java? (answer)
ThreadLocal variables are a special kind of variable available to Java programmers. Just like instance, the variable is per instance, `ThreadLocal` variable is per thread. It's a nice way to achieve thread-safety of expensive-to-create objects, for example, you can make SimpleDateFormat thread-safe using `ThreadLocal`. Since that class is expensive, it's not good to use it in local scope, which requires separate instances on each invocation.

By providing each thread their own copy, you shoot two birds with one arrow. First, you reduce the number of instances of expensive objects by reusing a fixed number of instances, and Second, you achieve thread safety without paying the cost of synchronization or immutability.

Another good example of a thread-local variable is `ThreadLocalRandom` class, which reduces the number of instances of `expensive-to-create` Random objects in a multi-threading environment. See this answer to learn more about thread-local variables in Java.

19) What is FutureTask in Java? (answer)
`FutureTask` represents a cancellable asynchronous computation in concurrent Java applications. This class provides a base implementation of Future, with methods to start and cancel a computation, query to see if the computation is complete and retrieve the result of the computation.

The result can only be retrieved when the computation has been completed; the get methods will block if the computation has not yet been completed. A `FutureTask` object can be used to wrap a Callable or Runnable object. Since `FutureTask` also implements `Runnable`, it can be submitted to an Executor for execution.

20) What is the difference between the interrupted() and isInterrupted() method in Java? (answer)
The main difference between `interrupted()` and `isInterrupted()` is that the former clears the interrupt status while the latter does not. The interrupt mechanism in Java multi-threading is implemented using an internal flag known as the interrupt status. Interrupting a thread by calling `Thread.interrupt()` sets this flag.

When interrupted thread checks for an interrupt by invoking the static method `Thread.interrupted()`, interrupt status is cleared. The non-static `isInterrupted()` method, which is used by one thread to query the interrupt status of another, does not change the interrupt status flag.

By convention, any method that exits by throwing an `InterruptedException` clears interrupt status when it does so. However, it's always possible that interrupt status will immediately be set again, by another thread invoking interrupt

21) Why wait and notify methods are called from the synchronized block? (answer)
The main reason for calling the wait and notify method from either synchronized block or method is that it is made mandatory by Java API. If you don't call them from a synchronized context, your code will throw `IllegalMonitorStateException`. A more subtle reason is to avoid the race condition between wait and notify calls. To learn more about this, check my similarly titled post here.

22) Why should you check the condition for waiting in a loop? (answer)
It's possible for a waiting thread to receive false alerts and spurious wake-up calls, if it doesn't check the waiting condition in a loop, it will simply exit even if the condition is not met. As such, when awaiting thread wakes up, it cannot assume that the state it was waiting for is still valid. It may have been valid in the past, but the state may have been changed after the `notify()` method was called and before the waiting thread woke up.

That's why it is always better to call the wait() method from a loop, you can even create a template for calling wait and notify in Eclipse. To learn more about this question, I would

recommend you to read Effective Java items on thread and synchronization.

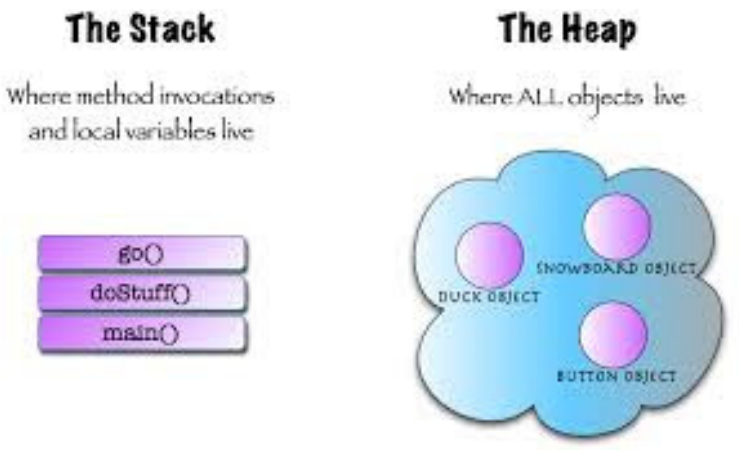
23) What is the difference between synchronized and concurrent collection in Java? (answer)
Though both synchronized and concurrent collection provides thread-safe collection suitable for multi-threaded and concurrent access, later is more scalable than former. Before Java 1.5, Java programmers only had synchronized collection which becomes a source of contention if multiple threads access them concurrently, which hampers the scalability of the system.

Java 5 introduced concurrent collections like `ConcurrentHashMap`, which not only provides thread safety but also improves scalability by using modern techniques like lock stripping and partitioning internal tables. See this answer for more differences between synchronized and concurrent collection in Java.

24) What is the difference between Stack and Heap in Java? (answer)
Why does someone this question as part of multi-threading and concurrency? because Stack is a memory area that is closely associated with threads. To answer this question, both stack and heap are specific memories in Java applications.

Each thread has its own stack, which is used to store local variables, method parameters, and call stack. Variable stored in one Thread's stack is not visible to other. On another hand, the heap is a common memory area that is shared by all threads.

Objects whether local or at any level is created inside the heap. To improve performance thread tends to cache values from the heap into their stack, which can create problems if that variable is modified by more than one thread, this is where volatile variables come into the picture. volatile suggest threads read the value of variable always from main memory. See this article for learning more about stack and heap in Java to answer this question in greater detail.



25) What is a thread pool? Why should you thread pool in Java? (answer)
Creating a thread is expensive in terms of time and resources. If you create a thread at the time of request processing it will slow down your response time, also there is only a limited number of threads a process can create. To avoid both of these issues, a spool of threads is created when the application starts up and threads are reused for request processing.

This pool of thread is known as "thread pool" and threads are known as a worker thread. From JDK 1.5 release, Java API provides Executor framework, which allows you to create different types of thread pools e.g. single thread pool, which processes one task at a time, fixed thread pool (a pool of fixed number of threads), or cached thread pool (an expandable thread pool suitable for applications with many short-lived tasks). See this article to learn more about thread pools in Java to prepare a detailed answer to this question.

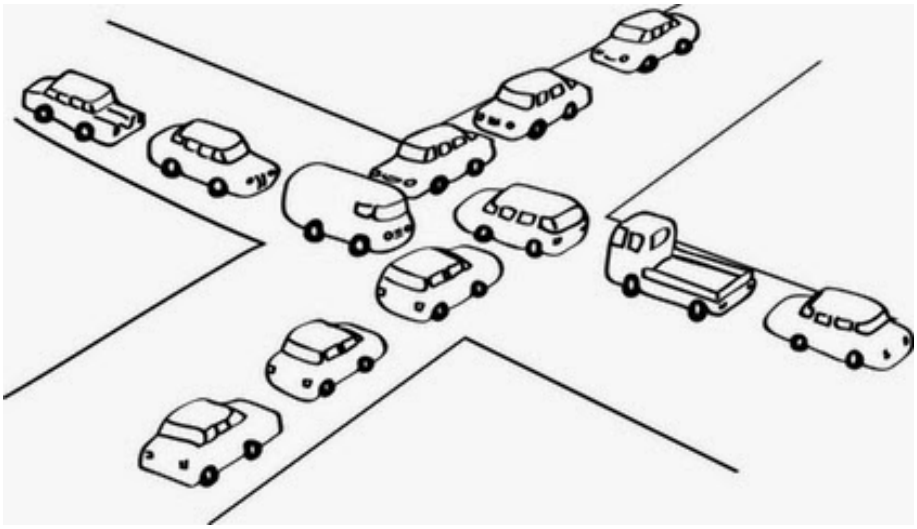
26) Write code to solve Producer Consumer problems in Java? (answer)
Most of the threading problems you solved in the real world are of the category of Producer consumer pattern, where one thread is producing a task and another thread is consuming that. You must know how to do inter-thread communication to solve this problem. At the lowest level, you can use wait and notify to solve this problem, and at a high level, you can leverage Semaphore or BlockingQueue to implement a Producer consumer pattern, as shown in this tutorial.

27) How do you avoid deadlock in Java? Write Code? (answer)
Deadlock is a condition in which two threads wait for each other to take action which allows them to move further. It's a serious issue because when it happens your program hangs and doesn't do the task it is intended for.

In order for the deadlock to happen, the following four conditions must be true:

- **Mutual Exclusion:** At least one resource must be held in a non-shareable mode. Only one process can use the resource at any given instant of time.
- **Hold and Wait:** A process is currently holding, at least, one resource and requesting additional resources which are being held by other processes.
- **No Pre-emption:** The operating system must not de-allocate resources once they have been allocated; they must be released by the holding process voluntarily.
- **Circular Wait:** A process must be waiting for a resource that is being held by another process, which in turn is waiting for the first process to release the resource.

The easiest way to avoid deadlock is to prevent *Circular wai*t, and this can be done by acquiring locks in a particular order and releasing them in reverse order so that a thread can only proceed to acquire a lock if it held the other one. Check this tutorial for the actual code example and detailed discussion on techniques for avoiding deadlock in Java.



28) What is the difference between livelock and deadlock in Java? (answer)
This question is an extension of the previous interview question. A livelock is similar to a deadlock, except that the states of the threads or processes involved in the livelock constantly change with regard to one another, without anyone progressing further.

Livelock is a special case of resource starvation. A real-world example of livelock occurs when two people meet in a narrow corridor, and each tries to be polite by moving aside to let the other pass, but they end up swaying from side to side without making any progress because they both repeatedly move the same way at the same time. In short, the main difference between livelock and deadlock is that in the former state of process change but no progress is made.

29) How do you check if a thread holds a lock or not? (answer)
I didn't even know that you can check if a Thread already holds a lock before this question hits me in a telephonic round of Java interviews. There is a method called `holdsLock()` on `java.lang.Thread`, it returns true if and only if the current thread holds the monitor lock on the specified object. You can also check this article for a more detailed answer.

30) How do you take thread dump in Java? (answer)
There are multiple ways to take a thread dump of the Java process depending upon the operating system. When you take thread dump, JVM dumps the state of all threads in log files or standard error console. In windows, you can use `Ctrl + Break` key combination to take thread dump, on Linux you can use the `kill -3` command for the same. You can also use a tool called `jstack` for taking thread dump, it operates on process id, which can be found using another tool called `jps`.

31) Which JVM parameter is used to control the stack size of a thread? (answer)
This is the simple one, -Xss parameter is used to control the stack size of Thread in Java. You can see this list of JVM options to learn more about this parameter.

32) What is the difference between synchronized and ReentrantLock in Java? (answer)
There were days when the only way to provide mutual exclusion in Java was via synchronized keyword, but it has several shortcomings e.g. you can not extend lock beyond a method or block boundary, you can not give up trying for a lock, etc.

Java 5 solves this problem by providing more sophisticated control via the Lock interface. ReentrantLock is a common implementation of the Lock interface and provides re-entrant mutual exclusion Lock with the same basic behavior and semantics as the implicit monitor lock accessed using synchronized methods and statements, but with extended capabilities.
See this article to learn about those capabilities and some more differences between synchronized vs ReentrantLock in Java.

33) There are three threads T1, T2, and T3? How do you ensure sequence T1, T2, T3 in Java? (answer)
Sequencing in multi-threading can be achieved by different means but you can simply use the join() method of thread class to start a thread when another one has finished its execution. To ensure three threads execute you need to start the last one first e.g. T3 and then call join methods in reverse order e.g. T3 calls T2. join and T2 calls T1.join, these ways T1 will finish first and T3 will finish last. To learn more about the join method, see this tutorial.

34) What does the yield method of the Thread class do? (answer)
The yield method is one way to request the current thread to relinquish CPU so that other threads can get a chance to execute. Yield is a static method and only guarantees that the current thread will relinquish the CPU but doesn't say anything about which other thread will get CPU. It's possible for the same thread to get the CPU back and start its execution again. See this article to learn more about the yield method and to answer this question better.

35) What is the concurrency level of ConcurrentHashMap in Java? (answer)
ConcurrentHashMap achieves its scalability and thread-safety by partitioning the actual map into a number of sections. This partitioning is achieved using a concurrency level.

Its optional parameter of ConcurrentHashMap constructor and its default value is 16. The table is internally partitioned to try to permit the indicated number of concurrent updates without contention. To learn more about concurrency level and internal resizing, see my post How ConcurrentHashMap works in Java.

36) What is Semaphore in Java? (answer)
Semaphore in Java is a new kind of synchronizer. It's a counting semaphore. Conceptually, a semaphore maintains a set of permits. Each acquire() blocks if necessary until a permit is available and then takes it. Each release() adds a permit, potentially releasing a blocking acquirer.

However, no actual permit objects are used; the Semaphore just keeps a count of the number available and acts accordingly. Semaphore is used to protect an expensive resource that is available in fixed numbers e.g. database connection in the pool. See this article to learn more about counting Semaphore in Java.

37) What happens if you submit a task when the queue of the thread pool is already filled? (answer)
This is another tricky question on my list. Many programmers will think that it will block until a task is cleared but it's true. `ThreadPoolExecutor's submit()` method throws `RejectedExecutionException` if the task cannot be scheduled for execution.

38) What is the difference between the submit() and execute() method thread pool in Java? (answer)
Both methods are ways to submit a task to thread pools but there is a slight difference between them. execute(Runnable command) is defined in Executor interface and executes given task in future, but more importantly, it does not return anything. Its return type is void.

On other hand `submit()` is an overloaded method, it can take either `Runnable` or `Callable` task and can return Future object which can hold the pending result of the computation.

This method is defined on `ExecutorService` interface, which extends `Executor` interface, and every other thread pool class e.g. `ThreadPoolExecutor` or `ScheduledThreadPoolExecutor` gets these methods. To learn more about thread pools you can check this article.

39) What is the blocking method in Java? (answer)
A blocking method is a method that blocks until the task is done, for example, accept() method of ServerSocket blocks until a client is connected. here blocking means control will not return to the caller until the task is finished. On the other hand, there is an asynchronous or non-blocking method that returns even before the task is finished. To learn more about the blocking method see this answer.

40) Is Swing thread-safe? What do you mean by Swing thread-safe? (answer)
You can simply this question as No, Swing is not thread-safe, but you have to explain what you mean by that even if the interviewer doesn't ask about it. When we say swing is not thread-safe we usually refer to its component, which can not be modified in multiple threads.

All updates to GUI components have to be done on the AWT thread, and Swing provides synchronous and asynchronous callback methods to schedule such updates. You can also read my article to learn more about swing and thread safety to better answer this question. Even next two questions are also related to this concept.

41) What is the difference between invokeAndWait and invokeLater in Java? (answer)
These are two methods Swing API provides Java developers for updating GUI components from threads other than the Event dispatcher thread. `InvokeAndWait()` synchronously update GUI component, for example, a progress bar, once progress is made, the bar should also be updated to reflect that change.

If progress is tracked in a different thread, it has to call `invokeAndWait()` to schedule an update of that component by the Event dispatcher thread. On another hand, `invokeLater()` is an asynchronous call to update components. You can also refer to this answer for more points.

42) Which method of Swing API is thread-safe in Java? (answer)
This question is again related to swing and thread-safety though components are not thread-safe there is a certain method that can be safely called from multiple threads. I know about `repaint()`, and `revalidate()` being thread-safe but there are other methods on different swing components e.g. `setText()` method of `JTextComponent`, `insert()` and `append()` method of `JTextArea` class.

43) How to create an Immutable object in Java? (answer)
This question might not look related to multi-threading and concurrency, but it is. Immutability helps to simplify already complex concurrent code in Java. Since immutable objects can be

shared without any synchronization it's very dear to Java developers. Core value object, which is meant to be shared among threads should be immutable for performance and simplicity.

Unfortunately, there is no `@Immutable` annotation in Java, which can make your object immutable, hard work must be done by Java developers. You need to keep basics like initializing state in the constructor, no setter methods, no leaking of reference, keeping a separate copy of the mutable object to create an Immutable object. For step by step guide see my post, how to make an object Immutable in Java. This will give you enough material to answer this question with confidence.

44) What is ReadWriteLock in Java? (answer)
In general, the read-write lock is the result of the lock stripping technique to improve the performance of concurrent applications. In Java, `ReadWriteLock` is an interface that was added in Java 5 release.

A `ReadWriteLock` maintains a pair of associated locks, one for read-only operations and one for writing. The read lock may be held simultaneously by multiple reader threads, so long as there are no writers.

The write lock is exclusive. If you want you can implement this interface with your own set of rules, otherwise you can use `ReentrantReadWriteLock`, which comes along with JDK and supports a maximum of 65535 recursive write locks and 65535 read locks.

45) What is a busy spin in multi-threading? (answer)
Busy spin is a technique that concurrent programmers employ to make a thread wait on certain conditions. Unlike traditional methods e.g. wait(), sleep(), or yield() which all involve relinquishing CPU control, this method does not relinquish CPU, instead, it just runs the empty loop. Why would someone do that? to preserve CPU caches.

In a multi-core system, it's possible for a paused thread to resume on a different core, which means rebuilding the cache again. To avoid the cost of rebuilding cache, programmers prefer to wait for a much smaller time doing busy spin. You can also see this answer to learn more about this question.

46) What is the difference between the volatile and atomic variables in Java? (answer)
This is an interesting question for Java programmers, at first, volatile and atomic variables look very similar, but they are different. A volatile variable provides you happens-before guarantee that a write will happen before any subsequent write, it doesn't guarantee atomicity.

For example, the count++ operation will not become atomic just by declaring the count variable as volatile. On the other hand, `AtomicInteger` class provides an atomic method to perform such compound operation atomically e.g. `getAndIncrement()` is the atomic replacement of increment operator. It can be used to atomically increment the current value by one.

Similarly, you have an atomic version for other data types and reference variables as well.

47) What happens if a thread throws an Exception inside a synchronized block? (answer)

This is one more tricky question for the average Java programmer if he can bring the fact about whether the lock is released or not is a key indicator of his understanding.

To answer this question, no matter how you exist synchronized block, either normally by finishing execution or abruptly by throwing an exception, the thread releases the lock it acquired while entering that synchronized block.

This is actually one of the reasons I like synchronized block over lock interface, which requires explicit attention to release lock, generally, this is achieved by releasing the lock in a finally block.

48) What is double-checked locking of Singleton? (answer)
This is one of the very popular questions on Java interviews, and despite its popularity, the chances of candidates answering this question satisfactory is only 50%. Half of the time, they failed to write code for double-checked locking, and half of the time they failed how it was broken and fixed on Java 1.5.

This is actually an old way of creating a thread-safe singleton, which tries to optimize performance by only locking when the Singleton instance is created the first time, but because of complexity and the fact it was broken for JDK 1.4, I personally don't like it.

Anyway, even if you do not prefer this approach it's good to know from an interview point of view. Since this question deserves a detailed answer, I have answered in a separate post, you can read my post how double-checked locking on Singleton works to learn more about it.

49) How to create thread-safe Singleton in Java? (answer)
This question is actually a follow-up to the previous question. If you say you don't like double-checked locking then the Interviewer is bound to ask about alternative ways of creating a thread-safe Singleton class.

There is actually man, you can take advantage of class loading and static variable initialization feature of JVM to create an instance of Singleton, or you can leverage powerful enumeration type in Java to create Singleton. I actually preferred that way, you can also read this article to learn more about it and see some sample code.

50) List down 3 multi-threading best practices you follow? (answer)
This is my favorite question because I believe that you must follow certain best practices while writing concurrent code which helps in performance, debugging, and maintenance. Following are three best practices, I think an average Java programmer should follow:

- **Always give a meaningful name to your thread**
  This goes a long way to find a bug or trace execution in concurrent code. `OrderProcessor`, `QuoteProcessor`, or `TradeProcessor` is much better than Thread-1. Thread-2 and Thread-3. The name should say about task done by that thread. All major frameworks and even JDK follow this best practice.
- **Avoid locking or Reduce scope of Synchronization**
  Locking is costly and context switching is even costlier. Try to avoid synchronization and locking as much as possible and at a bare minimum, you should reduce critical sections. That's why I prefer synchronized block over synchronized method because it gives you absolute control over the scope of locking.

- **Prefer Synchronizers over wait and notify**
  Synchronizers
  like `CountDownLatch`, `Semaphore`, `CyclicBarrier` or `Exchanger` simplify coding. It's very difficult to implement complex control flow right using wait and notify. Secondly, these classes are written and maintained by the best in business and there is a good chance that they are optimized or replaced by better performance code in subsequent JDK releases. By using higher-level synchronization utilities, you automatically get all these benefits.
- **Prefer Concurrent Collection over Synchronized Collection**
  This is another simple best practice that is easy to follow but reap good benefits. Concurrent collections are more scalable than their synchronized counterpart, that's why it's better to use them while writing concurrent code. So next time if you need a map, think about `ConcurrentHashMap` before thinking `Hashtable`. See my article Concurrent Collections in Java, to learn more about modern collection classes and how to make best use of them.

51) How do you force to start a thread in Java? (answer)
This question is like how do you force garbage collection in Java, there is no way though you can make a request using `System.gc()` but it's not guaranteed. On Java multi-threading there is absolutely no way to force start a thread, this is controlled by thread scheduler and Java exposes no API to control thread scheduling. This is still a random bit in Java.

52) What is the fork-join framework in Java? (answer)
The fork-join framework, introduced in JDK 7 is a powerful tool available to Java developers to take advantage of multiple processors of modern-day servers. It is designed for work that can be broken into smaller pieces recursively.

The goal is to use all the available processing power to enhance the performance of your application. One significant advantage of The `fork/join` framework is that it uses a work-stealing algorithm. Worker threads that run out of things to do can steal tasks from other threads that are still busy. See this article for a much more detailed answer to this question.

53) What is the difference between the calling wait() and sleep() method in Java multi-threading? (answer)
Though both wait and sleep introduce some form of pause in Java applications, they are tools for different needs. The wait method is used for inter-thread communication, it relinquishes lock if waiting for a condition is true and waits for notification when due to an action of another thread waiting condition becomes false.

On the other hand `sleep()` method is just to relinquish CPU or stop the execution of the current thread for a specified time duration. The calling sleep method doesn't release the lock held by the current thread. You can also take look at this article to answer this question with more details.

That's all on this list of **top 50 Java multi-threading and concurrency interview questions**. I have not shared answers to all the questions but provided enough hints and links to explore