Interview Questions on Threading – 2021-2022

**Qn – 1)** **Immutable objects are automatically thread-safe –true/false?**

Ans) **True**. Since the state of the immutable objects can not be changed once they are created they are automatically synchronized/thread-safe.

**Qn – 2**) **What is the difference when the synchronized keyword is applied to a static method or to a non static method?**

Ans) **When a synch non static method is called a lock is obtained on the object**.

**When a synch static method is called a lock is obtained on the class and not on the object**.

The lock on the object and the lock on the class don’t interfere with each other.

It means, a thread accessing a synch non static method, then the other thread can access the

synch static method at the same time but can’t access the synch non static method.

**Qn – 3)** What is the difference between yield() and sleep()?

Ans) yield() allows the current the thread to release its lock from the object and scheduler gives the lock of the object to the other thread with same priority.

sleep() allows the thread to go to sleep state for x milliseconds. When a thread goes into sleep state it doesn’t release the lock.

**Qn – 4** ) What is the difference between wait() and sleep()?

Ans)

1) wait() is a method of Object class. sleep() is a method of Object class.

2) sleep() allows the thread to go to sleep state for x milliseconds.

When a thread goes into sleep state it doesn’t release the lock. wait() allows thread to release the lock and goes to suspended state.

The thread is only active when a notify() or notifAll() method is called for the same object.

Qn - 5 ) What is difference between notify() and notfiyAll()?

Ans) notify( ) wakes up the first thread that called wait( ) on the same object.

notifyAll( ) wakes up all the threads that called wait( ) on the same object. The

highest priority thread will run first.

Qn – 6 ) Can a class have both Synchronized and non-synchronized methods?

Ans) Yes a class can have both synchronized and non-synchronized methods.

Qn – 7 ) If a class has a synchronised method and non-synchronised method, can multiple threads execute the non-synchronised methods?

Ans) Yes. If a class has a synchronised and non-synchronised methods, multiple threads can access the non-synchronised methods.

Qn – 8 ) If a thread goes to sleep does it hold the lock?

Ans) Yes when a thread goes to sleep it does not release the lock.

Qn – 9 )Can a thread hold multiple locks at the same time?

Ans) Yes. A thread can hold multiple locks at the same time. Once a thread acquires a lock and enters into the synchronized method / block, it may call another synchronized method and acquire a lock on another object.

Qn – 10 ) Can a thread call multiple synchronized methods on the object of which it hold the lock?

Ans) Yes. Once a thread acquires a lock in some object, it may call any other synchronized method of that same object using the lock that it already holds.

Qn – 11 ) Can static methods be synchronized?

Ans) Yes. As static methods are class methods and have only one copy of static data for the class, only one lock for the entire class is required. Every class in java is represented by java.lang.Class instance. The lock on this instance is used to synchronize the static methods.

Qn – 12 ) Can two threads call two different static synchronized methods of the same class?

Ans) No. The static synchronized methods of the same class always block each other as only one lock per class exists. So no two static synchronized methods can execute at the same time.

Qn – 13 )Does a static synchronized method block a non-static synchronized method?

Ans)No As the thread executing the static synchronized method holds a lock on the class and the thread executing the non-satic synchronized method holds the lock on the object on which the method has been called, these two locks are different and these threads do not block each other.

Qn – 14 ) Once a thread has been started can it be started again?

Ans) No. Only a thread can be started only once in its lifetime. If you try starting a thread which has been already started once an IllegalThreadStateException is thrown, which is a runtime exception. A thread in runnable state or a dead thread can not be restarted.

Qn – 15 ) When does deadlock occur and how to avoid it?

Ans) When a locked object tries to access a locked object which is trying to access the first locked object. When the threads are waiting for each other to release the lock on a particular object, deadlock occurs .

Qn – 16 ) What is a better way of creating multithreaded application? Extending Thread class or implementing Runnable?

Ans) If a class is made to extend the thread class to have a multithreaded application then this subclass of Thread can not extend any other class and the required application will have to be added to this class as it can not be inherited from any other class. If a class is made to implement Runnable interface, then the class can extend other class or implement other interface.

Qn – 17 ) Can the start() method of the Thread class be overridden? If yes should it be overridden?

Ans) Yes the start() method can be overridden. But it should not be overridden as itâ€™s implementation in thread class has the code to create a new executable thread and is specialized.

Qn – 18 ) What are the methods of the thread class used to schedule the threads?

Ans) The methods are as follows:

public static void sleep(long millis) throws InterruptedException

public static void yield()

public final void join() throws InterruptedException

public final void setPriority(int priority)

public final void wait() throws InterruptedException

public final void notify()

public final void notifyAll()

Qn – 19 ) Which thread related methods are available in Object class?

Ans) The methods are:

public final void wait() throws Interrupted exception

public final void notify()

public final void notifyAll()

Qn – 20 ) Which thread related methods are available in Thread class?

Ans) Methods which are mainly used :

public static void sleep(long millis) throws Interrupted exception

public static void yield() public final void join() throws Interrupted exception

public final void setPriority(int priority)

public void start()

public void interrupt()

public final void join()

public void run()

public void resume()

Qn – 21 ) List the methods which when called the thread does not release the locks held?

Ans) Following are the methods.

notify()

join()

sleep()

yield()

Q16) List the methods which when called on the object the thread releases the locks held on that object?

Ans) wait()

Qn – 22 ) Does each thread has its own thread stack?

Ans) Yes each thread has its own call stack. For eg

Thread t1 = new Thread();

Thread t2 = new Thread();

Thread t3 = t1;

In the above example t1 and t3 will have the same stack and t2 will have its own independent stack.

Qn – 23 ) What is thread starvation?

Ans) In a multi-threaded environment thread starvation occurs if a low priority thread is not able to run or get a lock on the resoruce because of presence of many high priority threads. This is mainly possible by setting thread priorities inappropriately.

Qn – 24 ). If 2 different threads hit 2 different synchronized methods in an object at the same time will they both continue?

1. No. Only one method can acquire the lock.

**What is happen-before relationship ?**

<https://stackoverflow.com/questions/11970428/how-to-understand-happens-before-consistent>

Each thread can be on a different core with its own cache. This means that one thread can write to a value storing in a register, or its local cache, and this value is not visible to another thread for some time. (milli-seconds is not uncommon)

A more extreme example is that the reading thread's code is optimised with the assumption that since it never changes the value, it doesn't need to read it from memory. In this case the optimised code never sees the change performed by another thread.

In both cases, the use of volatile ensures that reads and write occur in a consistent order and both threads see the same value. This is sometimes described as always reading from main memory, though it doesn't have to be the case because the caches can talk to each other directly. (So the performance hit is much smaller than you might expect)

The Java Memory Model defines a *partial ordering* of all your actions of your program which is called *happens-before*.  
To guarantee that a thread Y is able to see the side-effects of action X (irrelevant if X occurred in different thread or not) a *happens-before* relationship is defined between X and Y.  
If such a relationship is not present the JVM may re-order the operations of the program.  
Now, if a variable is shared and accessed by many threads, and written by (at least) one thread if the reads and writes are not ordered by the *happens before* relationship, then you have a data race.  
In a correct program there are no data races.  
Example is 2 threads A and B synchronized on lock X.  
Thread A acquires lock (now Thread B is blocked) and does the write operations and then releases lock X. Now Thread B acquires lock X and since all the actions of Thread A were done *before releasing* the lock X, they are *ordered before* the actions of Thread B which acquired the lock X*after* thread A (and also visible to Thread B).  
Note that this occurs on actions synchronized *on the same lock*. There is *no* happens before relationship among threads synchronized *on different locks*

<https://docs.oracle.com/javase/specs/jls/se7/html/jls-17.html#jls-17.4.5>

Happens-before Order

Two actions can be ordered by a *happens-before* relationship. If one action *happens-before* another, then the first is visible to and ordered before the second.

If we have two actions *x* and *y*, we write *hb(x, y)* to indicate that *x happens-before y*.

* If *x* and *y* are actions of the same thread and *x* comes before *y* in program order, then *hb(x, y)*.
* There is a *happens-before* edge from the end of a constructor of an object to the start of a finalizer ([§12.6](https://docs.oracle.com/javase/specs/jls/se7/html/jls-12.html#jls-12.6)) for that object.
* If an action *x* *synchronizes-with* a following action *y*, then we also have *hb(x, y)*.
* If *hb(x, y)* and *hb(y, z)*, then *hb(x, z)*.

The wait methods of class Object ([§17.2.1](https://docs.oracle.com/javase/specs/jls/se7/html/jls-17.html#jls-17.2.1)) have lock and unlock actions associated with them; their *happens-before* relationships are defined by these associated actions.

It should be noted that the presence of a *happens-before* relationship between two actions does not necessarily imply that they have to take place in that order in an implementation. If the reordering produces results consistent with a legal execution, it is not illegal.

*For example, the write of a default value to every field of an object constructed by a thread need not happen before the beginning of that thread, as long as no read ever observes that fact.*

More specifically, if two actions share a *happens-before* relationship, they do not necessarily have to appear to have happened in that order to any code with which they do not share a *happens-before* relationship. Writes in one thread that are in a data race with reads in another thread may, for example, appear to occur out of order to those reads.

The *happens-before* relation defines when data races take place.

A set of synchronization edges, *S*, is *sufficient* if it is the minimal set such that the transitive closure of *S* with the program order determines all of the *happens-before* edges in the execution. This set is unique.

It follows from the above definitions that:

* An unlock on a monitor *happens-before* every subsequent lock on that monitor.
* A write to a volatile field ([§8.3.1.4](https://docs.oracle.com/javase/specs/jls/se7/html/jls-8.html#jls-8.3.1.4)) *happens-before* every subsequent read of that field.
* A call to start() on a thread *happens-before* any actions in the started thread.
* All actions in a thread *happen-before* any other thread successfully returns from a join() on that thread.
* The default initialization of any object *happens-before* any other actions (other than default-writes) of a program.

When a program contains two conflicting accesses ([§17.4.1](https://docs.oracle.com/javase/specs/jls/se7/html/jls-17.html#jls-17.4.1)) that are not ordered by a happens-before relationship, it is said to contain a *data race*.

The semantics of operations other than inter-thread actions, such as reads of array lengths ([§10.7](https://docs.oracle.com/javase/specs/jls/se7/html/jls-10.html#jls-10.7)), executions of checked casts ([§5.5](https://docs.oracle.com/javase/specs/jls/se7/html/jls-5.html#jls-5.5), [§15.16](https://docs.oracle.com/javase/specs/jls/se7/html/jls-15.html#jls-15.16)), and invocations of virtual methods ([§15.12](https://docs.oracle.com/javase/specs/jls/se7/html/jls-15.html#jls-15.12)), are not directly affected by data races.

*Therefore, a data race cannot cause incorrect behavior such as returning the wrong length for an array.*

A program is *correctly synchronized* if and only if all sequentially consistent executions are free of data races.

If a program is correctly synchronized, then all executions of the program will appear to be sequentially consistent ([§17.4.3](https://docs.oracle.com/javase/specs/jls/se7/html/jls-17.html#jls-17.4.3)).

*This is an extremely strong guarantee for programmers. Programmers do not need to reason about reorderings to determine that their code contains data races. Therefore they do not need to reason about reorderings when determining whether their code is correctly synchronized. Once the determination that the code is correctly synchronized is made, the programmer does not need to worry that reorderings will affect his or her code.*

*A program must be correctly synchronized to avoid the kinds of counterintuitive behaviors that can be observed when code is reordered. The use of correct synchronization does not ensure that the overall behavior of a program is correct. However, its use does allow a programmer to reason about the possible behaviors of a program in a simple way; the behavior of a correctly synchronized program is much less dependent on possible reorderings. Without correct synchronization, very strange, confusing and counterintuitive behaviors are possible.*

We say that a read *r* of a variable *v* is allowed to observe a write *w* to *v* if, in the *happens-before* partial order of the execution trace:

* *r* is not ordered before *w* (i.e., it is not the case that *hb(r, w)*), and
* there is no intervening write *w*' to *v* (i.e. no write *w*' to *v* such that *hb(w, w')* and *hb(w', r)*).

Informally, a read *r* is allowed to see the result of a write *w* if there is no *happens-before* ordering to prevent that read.

A set of actions *A* is *happens-before consistent* if for all reads *r* in *A*, where *W(r)* is the write action seen by *r*, it is not the case that either *hb(r, W(r))* or that there exists a write *w* in *A*such that *w.v* = *r.v* and *hb(W(r), w)* and *hb(w, r)*.

In a *happens-before consistent* set of actions, each read sees a write that it is allowed to see by the *happens-before* ordering.

<https://stackoverflow.com/questions/16248898/memory-consistency-happens-before-relationship-in-java>

*Thread visibility problems* may occur in a code that isn't properly synchronized according to the java memory model. Due to compiler & hardware optimizations, writes by one thread aren't always visible by reads of another thread. The Java Memory Model is a formal model that makes the rules of "properly synchronized" clear, so that programmers can avoid thread visibility problems.

*Happens-before* is a relation defined in that model, and it refers to specific executions. A write W that is proven to be *happens-before* a read R is guaranteed to be visible by that read, assuming that there's no other interfering write (i.e. one with no happens-before relation with the read, or one happening between them according to that relation).

The simplest kind of happens-before relation happens between actions in the same thread. A write W to V in thread P happens-before a read R of V in the same thread, assuming that W comes before R according to the program order.

The text you are referring to states that thread.start() and thread.join() also guarantee happens-before relationship. Any action that happens-before thread.start() also happens before any action within that thread. Similarly, actions within the thread happen before any actions that appear after thread.join().

What's the practical meaning of that? If for example you start a thread and wait for it to terminate in a non-safe manner (e.g. a sleep for a long time, or testing some non-synchronized flag), then when you'll try reading the data modifications done by the thread, you may see them partially, thus having the risk of data inconsistencies. The join() method acts as a barrier that guarantees that any piece of data published by the thread is visible completely and consistently by the other thread.

What is the use of fairness parameter in concurrency ? To give an opportunity to longest waiting thread.

When set true, under contention, locks favor granting access to the longest-waiting thread. Otherwise this lock does not guarantee any particular access order. Programs using fair locks accessed by many threads may display lower overall throughput (i.e., are slower; often much slower) than those using the default setting, but have smaller variances in times to obtain locks and guarantee lack of starvation. Note however, that fairness of locks does not guarantee fairness of thread scheduling. Thus, one of many threads using a fair lock may obtain it multiple times in succession while other active threads are not progressing and not currently holding the lock. Also note that the untimed tryLock method does not honor the fairness setting. It will succeed if the lock is available even if other threads are waiting.