As a developer the single best thing we can do for producing clean, understandable code that won’t fall over is to avoid putting any multi threading into your code base.  The best programmers can break almost any problem down to be solved without complex threading. Most programmers will throw threading in to fix a problem they don’t understand or to prematurely optimise.

Java threading interview questions are exceptionally popular for interviews so it’s important to be able to demonstrate a sound understanding of how threads work in Java, when and why we use them, and how to actually program them in a safe manner.

In this article we will tackle threading from the ground up and cover what you need to know to be able to nail any questions that come up. This is the first in a multi-part series, because there is so much to cover.

**Introduction to Threading**

**Java Interview Questions:**

* **What is a Thread?**
* **How do we use Threads in java?**

Threads allow us to do more than one thing at once.  Without Threads code is executed in a linear fashion, one command after another. By using threads we can perform multiple actions at the same time.  This takes advantage of the fact that computers now have multiple processors as well as the ability to time slice tasks, so multiple threads can even be run on a single processors.

There are 2 main ways to create a new Thread in java.  Firstly, you can extend Java Thread and override the run method to execute your code.

1. public class SayHello extends Thread {
2. public static void main(String[] args) {
3. new SayHello().start();
4. }
5. @Override
6. public void run() {
7. for (int i = 0; i < 100; i++) {
8. System.out.println("Hi there!");
9. }
10. }
11. }

Generally we avoid this.  Java can’t cope with multiple inheritance so this option will limit your ability to extend anything else.  More importantly, it’s just not a very pretty way of doing it.  As good developers we aim to favour composition over inheritance.

Option two is much nicer, allowing us to implement the Runnable interface. The interface has one method on it, run.

1. public class SayHelloRunner implements Runnable {
2. public static void main(String[] args) {
3. new Thread(new SayHelloRunner()).start();
4. }
5. @Override
6. public void run() {
7. for (int i = 0; i < 100; i++) {
8. System.out.println("Hi there!");
9. }
10. }
11. }

We pass the new Runnable to a Thread to start it. We always use **start**to run the thread; using run() will run execute in the same thread as the caller.

**Java Interview Question: How do we stop a Thread in java? What is the volatile keyword?**

There is actually no API level way to stop a Thread reliably in java. The only guarantee is that when the code being executed in the thread completes, the thread will finish.  It is therefore down to the developer to implement a way of telling a Thread to stop.

1. public class SayHelloRunner implements Runnable {
2. private volatile boolean running = true;
3. public void stopIt(){
4. running = false;
5. }
6. @Override
7. public void run() {
8. while(running)
9. System.out.println("Hi there!");
10. }
11. }

As you can see, the flag to check if the thread is running or not has the keyword **volatile**.

In modern processors a lot goes on to try and optimise memory and throughput, and one of the major features is caching.  This is where regularly accessed variables are kept in a small amount of memory close to a specific processor to increase processing speed; it reduces the amount the processor has to go to disk to get information (which can be very slow).  However, in a multithreaded environment this can be very dangerous as it means that two threads could see a different version of a variable if it has been updated in cache and not written back to memory; processor A may thing int i = 2 wheras processor B thinks int i = 4.  This is where **volatile**comes in. It tells Java that this variable is accessed by multiple threads and therefore should not be cached, thus guaranteeing the value is correct when accessing it.  It also stops the compiler from trying to optimise the code by reordering it.

The downside is that there is a performance penalty on using volatile variables.  There is a greater distance to travel to get access to the information as it cannot be stored on the cache. However volatile is usually a much better option speed wise than using a synchronize block as it will not cause threads to block and as such is much faster than other options for synchronisation.

**Java Interview Question: : Explain the synchronized keyword. What does it do? Why do we need it?**

Synchronized is a keyword in java that can be used as a block around a piece of code or as part of the method signature.

1. public class Bank {
2. private int funds = 100;
3. public void deposit(int money){
4. synchronized (this){
5. funds += money;
6. }
7. }
8. public synchronized void withdraw(int money){
9. if(funds > money)
10. funds -= money;
11. }
12. }

Synchronized exists in Java to allow multiple threads who can both see an object to safely access it to ensure that the data is correct. The classic example is that of the bank account. Imagine if you remove the synchronisation from the above example. Thread one attempts to remove 100 from the account. At the exact same time, Thread two attempts to remove 100 from the account. For both threads when checking if there are sufficient funds the if statement returns true, resulting in the both withdrawing and a resultant balance of -100 which is no allowed.  By using synchronized only a single thread can access the section of code in the synchronized block, which can help us to ensure correct state.

When a synchronized keyword is placed on a method such as on withdraw(int money), it has the same effect as wrapping all of a methods code in synchronized(this), as in deposit(int money).  If any Thread tries to execute any synchronized method on the object it will be blocked.

**Java Interview Question:  What are the downsides of using synchronized? How can we avoid or mitigate them?**

Locks are slow. Really slow. Particularly when using the synchronized keyword, it can have a huge effect on performance. The reason for this is explained wonderfully in a [paper](http://lmax-exchange.github.io/disruptor/files/Disruptor-1.0.pdf) LMAX created on how they built the Disruptor, a super low latency concurrency component.  It is a brilliant read for anyone who wants to get into the fine details of threading and mechanical sympathy.

*” Locks are incredibly expensive because they require arbitration when contended. This arbitration is achieved by a context switch to the operating system kernel which will suspend threads waiting on a lock until it is released. During such a context switch, as well as releasing control to the operating system which may decide to do other house-keeping tasks while it has control, execution context can lose previously cached data and instructions. This can have a serious performance impact on modern processors.”*

Quite the mouthful and you probably don’t need to quote that in your interview.  It should be sufficient to say the along with the impact of threads being blocked whilst they wait for a resource, there is an impact at an OS level which causes huge performance damage.  In the same paper an experiment is carried out to see the impact on latency of different types of lock; without locking the process took 300ms, whereas with 2 threads contesting a lock it took 224,000ms.  Getting your threading right is hard but very important.

If you insist on using synchronized then minimising the size of the code block that is synchronized is a good start. The smaller this is will help to minimise impact.  Even better, you can lock on specific objects, or use a lock object if using a primitive, so that the entire object does not need to be contended.

1. public class Bank {
2. private int funds = 100;
3. private Object fundLock;
4. private UserDetails details = new UserDetails();
5. public void deposit(int money){
6. synchronized (fundLock){ //Lock Object needs to be acquired to update funds
7. funds += money;
8. }
9. }
10. public void createUsername(){
11. synchronized (details){ //Lock on the details object mean that the details object will block, but Bank will not.
12. details.setName("Bob");
13. }
14. System.out.println("His name is Bob");
15. }
16. }

Ideally though we want to avoid using synchronized where possible.

**Java Interview Question: What other options do we have for creating Thread safe code?**

As previously discussed, making a variable volatile is more performant and less complex.  However Java has a number of classes that aid us with threading so that we can do it in an efficient way.

**The Atomic Classes**

**Java Interview Question: What is CAS? Why is it good?**

Java has a number of classes such as AtomicInteger, AtomicDouble and AtomicBoolean (see more [here](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/package-summary.html)). These are completely thread safe and do not involve locking, but instead use Compare and Swap (CAS) operations.  A CAS operation is an atomic one (as in, it happens as one single operation) where the field is checked that the value is what was expected (it hasn’t been modified by a thread) and if so it sets the value to the new amount.  This is exceptionally quick; CAS is actually an operation available on most processors and no blocking is required. It is still slower than single threaded (the LMAX experiment concluded it was 100x slower) but it is the best available away of ensuring thread safe access to values.

It is important to understand the difference between volatile and CAS.  if you try and perform any operation that uses the original value it will render the volatile keyword useless. For example, “i++” breaks down into “i=i + 1″.  In the time between i being read (i + 1) and it being written (i = result) another thread can jump in between and update the value. Because the Atomic classes checks and writes values as an **atomic** operation, it is a much safer option.

**Immutability**

Immutable objects are brilliant. An immutable object is an object whose state cannot be changed after creation.  By this definition all immutable objects are thread-safe.  Programming to use immutable objects can be difficult and is a paradigm shift from standard Object Oriented Programming. How to create immutable objects in Java is an article in itself.  Functional languages like Scala make heavy use of immutable objects which allows them to scale well and be highly concurrent.  The more we can rely on immutable objects the less threading we will need.

**Thread methods**

**Java Interview Questions**

* **What does yield() do?**
* **What does interrupt() do?**
* **What does join() do?**

The Thread class has a whole bunch of methods on it which you need to know for your interview.  Interviewers love testing people on their ability to remember the details around these methods. In reality they are very rarely used in practice which is why it’s good to read up on them beforehand.

**Yield** does exactly what it says; it gives up. Imagine you have a single CPU only, if a thread refuses to allow other threads CPU time they will remain permanently blocked.  By calling yield() the thread is generously saying “who else wants a turn?”.  In reality it is unreliable and isn’t normally used. The implementation is different on any system or JVM, and can often be different between Java versions. There is no guarantee of what other thread will get the next access, nor when your thread will next get an opportunity to run.

You will have noticed over the years that a whole bunch of stuff (most commonly Thread.sleep()) forces you to catch an InterruptedException.  If we need to stop a Thread for whatever reason we can do this by calling **interrupt**() on it.  This sets the “interrupted” flag on the target thread. If the target thread chooses to check this flag (using the interrupted() or isInterrupted()) then it can optionally throw an Exception (usually Interrupted exception).  It is predicated on the target Thread actually checking this.  However there are a number of methods, such as Thread.sleep() which automatically poll for this flag.  If the target thread is running one of these then an InterruptedException will be thrown straight away.

**join()**will simply tell the currently running thread to wait until completion of whichever thread join() is called on completes.

1. public class SayHelloRunner implements Runnable {
2. public static void main(String[] args) throws InterruptedException {
3. Thread thread = new Thread(new SayHelloRunner());
4. thread.start();
5. thread.join();
6. System.exit(3);
7. }
8. @Override
9. public void run() {
10. int i = 0;
11. while(i < 10000){
12. System.out.println(i);
13. i++;
14. }
15. }
16. }

In this example if we did not have the thread.join() the application will exit without printing anything out. The join() call effectively makes the call synchronous; it want’s the Thread to finish before proceeding.

**Object methods**

**Java Interview Questions:**

* **What do wait(), notify() and notifyAll() do?**
* **Why must they be called from a synchronised block?**
* **What alternatives are there to these?**

**wait(), notify() and notifyAll()**are used as a means of inter-thread communication.  When acquiring a lock on an object it may not be in the required state; perhaps a resource isn’t set yet, a value isn’t correct.  We can use wait() to put the thread to sleep until something changes.  When that something does change, the awaiting clients can be notified by calling notify() or notifyAll() on the object that is being waited for.  If all of your waiting  threads could in theory take action with the new information then use notifyAll(). However if there is a new lock to be acquired (so only one waiting Thread can take action), then call just notify().

Example:

1. public class Example {
2. public static void main(String[] args) {
3. ResourceCarrier carrier = new ResourceCarrier();
4. ThingNeedingResource thingNeedingResource = new ThingNeedingResource(carrier);
5. ThingNeedingResource thingNeedingResource2 = new ThingNeedingResource(carrier);
6. ThingNeedingResource thingNeedingResource3 = new ThingNeedingResource(carrier);
7. ResourceCreator resourceCreator = new ResourceCreator(carrier);
8. new Thread(thingNeedingResource).start();
9. new Thread(thingNeedingResource2).start();
10. new Thread(thingNeedingResource3).start();
11. new Thread(resourceCreator).start();
12. }
13. }public class ResourceCarrier {
14. private boolean resourceReady;
15. public boolean isResourceReady() {
16. return resourceReady;
17. }
18. public void resourceIsReady() {
19. resourceReady = true;
20. }
21. }
22. public class ResourceCreator implements Runnable {
23. private ResourceCarrier carrier;
24. public ResourceCreator(ResourceCarrier carrier) {
25. this.carrier = carrier;
26. }
27. @Override
28. public void run() {
29. try {
30. Thread.sleep(2000);
31. } catch (InterruptedException e) {
32. e.printStackTrace();
33. }
34. synchronized (carrier) {
35. carrier.resourceIsReady();
36. carrier.notifyAll();
37. }
38. }
39. }
40. package com.core.interview.thread;
41. public class ThingNeedingResource implements Runnable {
42. private ResourceCarrier carrier;
43. public ThingNeedingResource(ResourceCarrier carrier){
44. this.carrier = carrier;
45. }
46. @Override
47. public void run() {
48. synchronized (carrier){
49. while(!carrier.isResourceReady()){
50. try {
51. System.out.println("Waiting for Resource");
52. carrier.wait();
53. } catch (InterruptedException e) {
54. e.printStackTrace();
55. }
56. }
57. System.out.println("haz resource");
58. }
59. }
60. }
61. Sample output:
62. Waiting for Resource
63. Waiting for Resource
64. Waiting for Resource
65. haz resource
66. haz resource
67. haz resource

In this example we have a wrapper around a resource called “Resource Carrier” and 3 threads that want access to it’s resource. When it acquires the lock it sees the resource isn’t available and goes into wait mode by calling wait() on the ResourceCarrier object. The ResourceCreator swoops in later to create the resource and calls notify on the ResourceCarrier, at which point the three threads spring back to life.

**Deadlock**

**Java Interview Questions:**

* What is a deadlock?
* How do we prevent deadlocks?

I have saved the most important question until last. It is not that this is an inherently difficult question (because it really isn’t), but many candidates completely fluff this.  It’s an easy enough concept but can be tricky to explain (particularly over the phone).

A deadlock occurs when two or more threads are awaiting the actions of each other which prevents any further processing from happening. For example, Thread A has the lock on Object 1 and attempts to require a lock on Object 2.  Simultaneously, Thread B has a lock on Object 2 and attempts to acquire a lock on object 1.  As a result both Threads are kept busy waiting for the other one, which will never release.  This is a deadlock.

The easiest way to prevent deadlock is to ensure that the acquisition of locks is consistent.   If both Threads try to acquire Object 1 then Object 2 in that order then deadlock will not occur.  It is important to code defensively when acquiring locks.  Even better is to make synchronized blocks as small as possible and where possible avoid locking multiple objects.