JAVA MULTITHREADING BEGINNER INTERVIEW Q&A

By QuickTechie.com

Q1. How Java Thread is different than Regular Java Objects?

Answer: Java Thread are same as any regular Java Objects. They are instance of *java.lang.Thread* class. However, they have extra capabilities, whatever you written in *run*() method of Thread class instance will be executed in a separate thread.

Q2. What are all the ways by which a new Thread can be created?

Answer: There are mainly two ways to create new Thread. However, both method requires Thread class.

Approach 1: Using Thread Class, Create an Instance of Thread Class

```
Thread threadInstance = new Thread(); // Create Instance of a thread class
threadInstance.start(); // Starting thread
```

Approach 2: Using **Runnable** interface: In this approach we need to Implement **Runnable** interface.

```
public class TestRunnable implements Runnable {
            public void run(){}
}

Thread threadInstance = new Thread(new TestRunnable() ); // Create Instance of a thread class
threadInstance.start(); // Starting thread, it will execute run method of TestRunnable object
```

Q3. When we should use, approach in which we are sub-classing the Thread (Important)?

Answer: There is no particular rule for defining which approach is better. However, it is recommended when you need long running threads e.g. **ThreadPool** (Having like 10 of threads, pre-created). We should use Thread sub-classing approach. Hence, we have 10 long running threads and their responsibility is to run the task submitted to them.

Let's take an example: You are having a 10,000 small test files. Now you want to calculate number of words in each file. In this case, what we are going to do, create 10,000 tasks (not 10000 threads). We will submit these 10,000 tasks to **ThreadPool** (which has 10 threads). Each thread will run 1000 tasks. Hence, using only 10 threads we have completed our entire word count of 10,000 files. **These types of BigData Problems can be easily solved using Hadoop Framework (visit: www.HadoopExam.com)**

Q4. When should we use approach for Runnable interface?

Answer: As mentioned in above question, we should use Thread sub-classing approach, when we need *ThreadPool* (**Long running Threads**). And the tasks mentioned in above example should be Runnable. Hence, we will create 10,000 Runnable instances and submitting them to *ThreadPool*, and each thread from *ThreadPool* will execute the Runnable instance.

Q5. What happen if we call *run()* method directly on thread class instead of calling start() method on thread class instance?

Answer: Below code shows, how to start a new thread

QuickTechie

```
Thread threadInstance = new Thread(); // Create Instance of a thread class
threadInstance.start(); // Starting thread
```

Now if we call *threadInstance.run()* directly, **than it will not create a new Thread** and execution will be part of same thread which called run() method. Hence, it is must to call start() method.

Q6. How to assign name to a Thread and what is the use of it?

Answer: Assigning name to thread will help us debugging multithreaded application. When we see logs, we can easily find which thread is processing our requests and troubleshoot accordingly.

You can assign name to thread as following

- While creating thread instance ,we can give name to Thread

```
Thread thread = new Thread("HadoopExam.com"); //HadoopExam.com is name of the thread
```

- In case of Runnable, we can assign name to Thread as below.

```
TestRunnable runnable = new TestRunnable();
//Name of the Thread is HadoopExam.com and
//runnable is a task, which will be eacuted by threadInstance Thread
Thread threadInstance=new Thread(runnable,"HadoopExam.com");
```

To retrieve thread name. we can use following method.

```
Thread.currentThred.getName();
```

Q7: How to get handle/refrence of the currently Running thread?

Answer: We can use `Thread.currentThread()` methods, which will return reference to currently running thread.

Q8. When you start your applications using main() method, in which thread this applications starts?

Answer: Java application will always have at least one thread by default that is called **main thread**. If you do not have any explicit thread than still application will have one thread that is called main thread.

Q9. In what order threads will be executed?

Answer: There is no default order by which thread can be executed. It depends on operating system. However, we can control thread ordering multiple ways e.g. Priority, Synchronizing etc.

Q10. What is a "Critical Section" of code?

Answer: In a multithreaded application a code section which is executed by multiple threads are called Critical Section, if Sequence of execution have side effects on final results. For example

```
public int foo(){
		return i=i+1; //This line is critical section, if used in multithread application.
}
```

Q11. What is a Race Condition?

Ans : When multiple thread execute Critical section of code segment, and thread execution order can lead to different results can be considered as a Race Condition. Lets assume we have below code segment

```
Class HadoopExam
{
        Int counter=0;

        Int foo(){
                return counter= counter+1; //Critical Section
        }
}
```

Now there are two threads **ThreadA** and **ThreadB**. Both are executing above code. ThreadA read value of counter (which is 0 as of now) and will be updating it by one. However, as soon as ThreadA reads counter value as 0, at the same time ThreadB also read counter value which is also 0 (because ThreadA has not written back). Now finally when ThreadA and ThreadB both writes the Value. It will become 1 instead of 2 . Which is a wrong results, because counter is increased twice from 0 and result is still 1. This can be avoided by executing thread in proper order. This problem is known as a **Race Condition**. Race condition occurs only when Thread update the resource values, only reading will not create Race condition.

Q12. How can you avoid Race Condition?

Answer: As we can see in above example "*counter=counter+1*" is not an Atomic operation. To avoid Race condition we need to make operation atomic. This can be done using Synchronized block or method. So only one thread at a time can execute Critical Section of the code. Hence, ThreadA is executing "counter=counter+1" then ThreadB has to wait.

Q13. What is the Impact of Synchronizing "Critical Section" of code?

Answer: In above example we have two threads ThreadA and ThreadB, and by Synchronizing we are allowing only one thread to execute and other ThreadB needs to wait, which will impact the overall performance of multithreaded application. Hence, it is always advisable to break critical sections in multiple part and synchronized accordingly. So it will not impact overall throughput of multithreaded application.

If Critical Section of code is larger than, we have to be very careful while breaking multiple Synchronized block, else it could lead another problem

Q14. What do you mean by Thread Safe code?

Ans : Code which does not have any side effect , if multiple thread calls same code in different order is called Thread safe code. And code must not have any Race Condition, Deadlock ans Starvation.

Q15. What is local variables and why they are Thread Safe?

Ans : Variables you defined in a method is called local variable. See below example

```
        Int foo(){
                Int localCounter=10;
                return localCounter = localCounter +1; //Local Variable increment
        }
```

In above code "*localCounter*" is a local variable.

Thread safe: Local primitive variable are always thread safe because, they are always stored as part of thread's stack (And thread stack will never be shared across threads). So it is always used by one thread. Hence, it is thread safe in above code "*localCounter*" is created in thread stack and once thread is out of method block, this variable will be Garbage collected.

Q16. Is Local Object thread safe (Important)?

Answer: As local Objects are not same as local primitive values. Local objects references are always stored in Thread's local stack but actual instance of Object will be stored in **Shared Heap(memory area shared among thread).**

If we use this local object in the same method block and never shared with another thread than it is a thread safe. Even, you pass same object reference in another method and no alteration is done by any other thread on this object than still it is a Thread safe object.

However, if you pass primitive local variable to another thread it is thread safe, because it always pass a copy of the variable. But if you pass Objects it will pass a copy of reference, which will point to same object and can alter the object (And code will not be thread safe)

Q17. What is Object Member variable?

Answer: Object member variables are fields which are created as a member of class and not a method. See below example.

```
Class HadoopExam
{
        Int counter=0;

        Int foo(){
                return counter= counter+1; //Critical Section
        }
}
```

Here, counter is member variable of a Class HadoopExam.

Q18. Is Member variable are thread safer?

Answer: Object member variable similar to any other objects will be stored in heap (shared memory area). Hence, if we have single instance of HadoopExam class and two different thread TreadA and ThreadB tries to modify member variable counter than this code is not thread safe. Because same copy of the variable will be updated by two different thread. Which can cause Race condition.

If you have two different instances of HadoopExam class and each Thread works on different than there would not be any issue. Because each thread is working on different copy of member variable. **(What happen if member variable is Static and String constants?)**

Q19. What is Immutable objects and are they thread safe?

Answer: Any objects which is once created and cannot be modified after creation will be called Immutable objects. See below code

```
public class HadoopExam{

  private int counter = 0;

  public HadoopExam(int value){
    this.value = value;
  }

  public int getCounter(){
    return this.counter;
  }
}
```

Above class is defined that, you can create object as below.

HadoopExam instance = new HadoopExam(20);

Now, once instance is created there is no way by which we can change **instance.counter** value. Hence, this class is called Immutable.

Now, we have instance which cannot be modified once created. So as many number of threads use it, no problem. Because nobody(no thread) can modify this object. Hence, immutable object are always thread safe.

Q20. What is Java Memory Model?

Answer: Java Memory model defines, how JVM works with the computer main memory also known as RAM. Whenever you launch a Java application, it reserves some memory for RAM for your application also based on JVM arguments it can ask more memory during your application run if needed.

JVM itself you can assume as an OS, which will interact with your computer OS and it has its own way to manage memory reserved by your Java application, which is known as Java Memory Model.

Java memory model specifies how different thread see the shared variable in main memory. So it is about visibility of values across threads of shared variable.

Q21. What is thread stack and Heap space as per Java Memory model?

Answer: Java memory model divides reserved memory in two separate area known as thread stack and Heap Space.

Thread Stack: It is specific to each thread. If you have 100 threads running in your application. You will be having 100 thread stacks. Thread stack keeps information in “call stack”. Thread stack also keeps all local variables for each method. Local variable in thread stack will not be visible to another thread hence, thread safe.

Call Stack: Call stack keep the information about all the method it has executed till this point. For instance there are 5 method named as method1(), method2().. method5().. and you are currently on method3(). It will keep information on call stack as below. However, local variable are GCed as soon as thread completes the method execution.

method3()
method2() : local variable of method2 as well
method1() : local variable of method2 as well

Heap Space: It contains all objects created in your java application. It can be created by any thread. All objects will shared same memory space. Objects could be local variable or member variable all will be stored in heap space.

Q22. Static class variable stored in Heap space or in Thread stack?

Answer: Static class variables are stored on Heap space, with the class definition.

Q23. How java memory model mapped to Hardware memory on multiple CPU?

Answer: Let’s first understand Hardware memory architecture: There are mainly three sections of memory on any computer (**Remember your college time computer architecture syllabus**).

- RAM: Main Memory (It is shared across CPUs). It is the biggest memory area for JVM.
- CPU Cache: Each CPU has its own CPU cache. Accessing CPU cache is faster than main memory but slower than CPU registers.
- CPU Registers: Each CPU has its own Registers as part of CPU memory. Most of the calculations happen on CPU registers and it will be much faster.

Each CPU is capable of running single thread at any moment. So if we have multithreaded application (each thread will be running on separate CPU, assume we have 10 threads and 2 CPUs, then only two threads at a time can execute concurrently out of 10 threads.)

Insert an Image here:

Q24. What is the Role of RAM, CPU cache and Registers in java process?

Answer: Whenever, any calculations/process needs to be executed by a thread. It will read value from main memory and load into CPU cache. And from CPU cache to registers.

Once calculation is completed resulting variable returned from registers to cache and flushed from cache to main memory.

From CPU Cache to main memory flushing happens only when CPU needs more memory to do some calculations or stored new variable

Q25. How Thread stack and Heap Space mapped to this Hardware memory model.

Answer: So at hardware level there is no difference between Thread Stack and Heap Space. It is JVM responsibility to keep them separate. Thread Stack will share have their variables stored in any of the Hardware memory area e.g. Main Memory, CPU Cache and Registers. Similarly Heap Space can also share any of the memory area.

Insert Image Here

Q26. What all problems needs to be taken care because of these memory model differences between Hardware and JVM?

Answer: There are mainly below problems needs to be taken care

- Race conditions (Visibility of variables across threads) : Lets assume we are having a one variable named counter as below

  ```
  int counter=0;
  counter = counter+1;
  ```

  Now two threads TreadA and ThreadB , both are executing same code segment(Critical Section). Now ThreadA read the value of counter as counter=0 and load into CPU cache. At the same time another thread ThreadB also read this variable as counter=0 and load and keeps it in its own CPU cache. Both the thread will now update value by 1 independently on their own CPU cache. And their changes are not visible to each other Thread. Which cause wrong final value. Because both will be writing this value as 1 (**But it should be 2, because it is incremented twice**). This problem can be avoided by declaring counter variable as a **volatile or this critical section needs to be surrounded by Synchronized block.**

Q27. How Synchronized block will help in such scenario?

Answer: If we surround Critical section of the code by Synchronized block, it will guarantee following

- Only one thread at a time can access critical section of the code
- All the variables of Critical section will always be read from main memory
- Thread as soon as leave synchronized block, it will flush all the variables to main memory.

Because of all these guarantees, Race condition will never occur.

Q28. What happens, when you write code inside synchronize block?

Ans : If you add synchronized block around critical section of the code, only one thread at a time can execute this code section. Because before entering synchronized block thread (ThreadA) has to first get a lock on the object (also known as monitor). And only one thread can get a lock, if other thread (ThreadB) wants the lock than it has to wait for ThreadA to release the lock.

Q29. Which all the places, we can apply synchronized keyword?

Ans : We can apply synchronized block following places in code

- Method level (Both static and non-static)

- Code block (Inside both static and non-static method)

Q30. Give example of both method level as well as block level synchronization?

Answer:

- **Method level (Non static)**: In this case thread will have to take a lock on this object (current instance) from of which this foo method needs to be executed.

```
public synchronized void foo(int counter){
     this.counter += counter;
  }
```

- **Method level** (Static) : In this case thread will have to take a lock on Class object (Definition of Class stored in Heap Space) from of which this foo method needs to be executed.

```
public static synchronized void foo(int counter){
     this.counter += counter;
  }
```

- **Block Level (non-static method):** In this case thread will have to take a lock on this object (current instance) from of which this foo method needs to be executed.

```
public  void add(int value){
synchronized(this){
     count += value;
             }
  }
```

- **Block level (Inside static method)**

```
public static void add(int value){
synchronized(this){
     count += value;
             }
  }
```

Volatile keywords onwards

Q31: What is a Volatile variable?

Answer: If you are using “volatile” keyword for a variable than Java guarantees every change on volatile variable will be visible to other thread. It will always read a variable value from “Main memory” and never from CPU Cache and Registers. Similarly every write will also be done in “Main memory”. **In one line: volatile variables read and write always happen from Main memory**.

If thread is doing any operation on a volatile variable, it will always read from main memory and copy it in CPU cache than process the variable. Once processing is done immediately flush back from CPU cache to main memory (It always update main memory, so other thread will read from main memory , latest value).

Q32. What all guarantees java volatile variable?

Answer: “volatile variable” gives below guarantee

1. Let’s assume there are two threads ThreadA and ThreadB. If ThreadsA start writing to volatile variable and has few other variables in memory and at the same time ThreadB reads the same volatile variable. Then all variables which were visible to ThreadA before writing the volatile variable, same variable will be visible to ThreadB after reading “volatile variable”
2. Instruction reordering: For performance reason JVM re-order instructions). So before and after of volatile variable instructions may re-order, but volatile read and write instructions can not be re-ordered.

```
int count1=0;
int count2=0;
int count3=0;
volatile count4=0;
int coount5=0;
int count6=0;
int count7=0;
```

JVM can re-order count1 to count3 only if writing happens before writing count4 variable. Similarly JVM can re-order count5 to count7 only if writing happens before count4.

Suppose thread writes count4 from CPU cache to main memory and it also has other variable like count1 to count3 and count5 to count7 in memory, then it will also flush these variables to main memory.

Q33. Is volatile guarantees 100% visibility?

Answer: Because there is no lock involved. Hence, small time gap while reading and writing back to main memory can lead to “Race condition”.

Q34. When is volatile good to use?

Answer: If only one thread is writing and all other threads just need to read the variable. Hence, all the reading thread will always read the latest value written to main memory, and they are not going to change the value, hence no “Race condition”

Q35. Does volatile impact the overall performance?

Answer: Yes, because reading from main memory and writing to main memory is always expensive than reading and writing from CPU cache or registers.

Q36. What is a ThreadLocal?

Answer: ThreadLocal as name suggests, they are local to each thread. It cannot be shared across thread. So whatever read and write happens to ThreadLocal object it will be visible to only same local thread.

Q36. How you will use ThreadLocal?

Answer: ThreadLocal instance can be created as below.

```
private ThreadLocal threadLocal = new ThreadLocal();
```

We can set the value in threadLocal as below.

```
threadLocal.set("Value");
```

You read the value stored in a ThreadLocal like this:

```
String threadValue = (String) threadLocal.get();
```

Q37. What is thread signaling?

Ans : Thread signaling is a way by which thread can communicate with each other. For thread signaling threads will use a common objects lets say “CommonObjects”. And thread will call “wait()” ,”notify()” or “notifyAll()” method to communicate with each other.

Q38. Signaling methods are defined in which class?

Answer: As mentioned above all methods “wait()” ,”notify()” or “notifyAll()” are defined on Object level. Hence, every objects will have these methods. And also mentioned all thread must be using common object to get signal from each other. So thread will call these methods on common objects.

Q39. What is the use of “wait()” ,”notify()” or “notifyAll()” ?

Ans : Suppose we have two three threads ThreadA, ThreadB and ThreadC. All needs to execute below code block (this is not complete code)

```
synchronized write(){
        counter=counter+1;
        this.notify();
}
synchronized read(){
        this.wait()
        return counter;
}
```

Now ThreadA is executing *write()* method and ThreadB are executing *read()* method. To entering in synchronized block each thread has first take a lock on common objects (that is this object, in this case). Let’s assume ThreadB get a lock and to read the counter value, and ThreadA is waiting for getting lock on this object to enter *write*() method. As soon as ThreadB reaches to *this.wait()* it will release the lock and wait for notifications from other thread to return counter value. Now, ThreadA can get a lock and start writing counter value and once done it will *notify*() waiting thread (in this case ThreadB) and release the lock. As soon as thread get notification it will process and return the updated counter value.

Q40. Is it necessary to have wait() and notify() calls to be in synchronized block?

Answer: Yes, *wait*(), notify() and *notifyAll*() can be called from synchronized block only. They must have a lock on common object. Than only they can call these methods.

Q41. What is the difference between sleep() and wait() method, with regards to lock?

Answer: When thread calls *sleep*(), method it is not necessary to have it in synchronized block. However, if you are in synchronized block and you call *sleep*() method on thread. It will not release the lock. And also note that sleep is a Thread class method and not a part of "Common object".

Q42. What is the difference between sleep() , join() and yield() method of Thread class?

Ans : **sleep()** : causes the thread to definitely stop executing for a given amount of time; if no other thread or process needs to be run, the CPU will be idle (and probably enter a power saving mode).

yield() method pauses the currently executing thread temporarily for giving a chance to the remaining waiting threads of the same priority to execute. If there is no waiting thread or all the waiting threads have a lower priority then the same thread will continue its execution. The yielded thread when it will get the chance for execution is decided by the thread scheduler whose behavior is vendor dependent.

join() If any executing thread t1 calls join() on t2 i.e; t2.join() immediately t1 will enter into waiting state until t2 completes its execution.

Q43. What do you mean by missed signals?

Answer: It is possible that a thread can miss a signal. Let's assume there are two threads ThreadA and ThreadB. ThreadA is going to enter in wait() state and ThreadB is going to notify(). However, ThreadB notify() ThreadA before ThreadA enter in wait() block. In this case notify() signal from ThreadB is missed by ThreadA. Hence, ThreadA remain in wait state forever, which could be a big problem. Missed signal problem can be avoided using proper code block around shared object.

Q44. What is a spurious wakeup?

Answer: It is possible because of any surprise reason like hardware issue, thread gets signal. Lets assume again we have two threads ThreadA and ThreadB , ThreadA is waiting on notification from ThreadB. However, ThreadA gets notification that ThreadB has done with its processing. This is a false signal and known as a spurious wakeup of ThreadA.

To avoid spurious wakeup issue, we should check wait state of shared object in while loop. So in case of false signal it has to check condition on shared object again.

Q45. When should notifyAll() used?

Ans : notifyAll() should be used when many threads are waiting for signal.

Q46. Why we should avoid wait() or getting lock on a Constant strings or global objects?

Ans : We should avoid Constant String, because JVM internally store different Constant String with same value as a Single object.

String str1=new String(“HadoopExam.com”);

String str2=new String(“HadoopExam.com”);

JVM internally stored both the str1 and str2 as two separate reference variable pointing to single objects in String constant pool. Hence, if two different threads are working on two different object references can get a wrong/false signal.

Same problem occurs with globally shared objects.

Q47. What is Thread deadlock?

Answer: Deadlock is situation, when two or more threads are blocked and waiting to get lock on objects. Lets assume there are two threads ThreadA and ThreadB and there two shared objects obj1 and obj2. ThreadA is holding a lock on obj1 and also needs lock on obj2 to proceed further. Similarly ThreadB is holding lock on obj2 and needs lock on obj1 to proceed further. Now both ThreadA and ThreadB is waiting for each other to release the lock and no one can proceed further. This situation is called deadlock.

Q48. Why deadlock happens?

Answer: Deadlock occurs because two or more threads are waiting for each other locks at the same time. However, order of getting and releasing lock is not proper it can arise deadlock.

Q49. How can deadlock be prevented?

Answer: to prevent deadlock following techniques can be used

- **Lock Ordering**: Locks needs to be taken and released in same order by each thread
- **Lock Timeout and retry**: Threads which are waiting to get lock on threads does not get locks in specified time limit. Then it should release all the locks it has taken needs to re-try after sometime.
- **Deadlock Detection**: Detect the deadlock and try to avoid. You can use thread priority as well to avoid deadlock.

Q50. What is thread starvation and what causes it?

Answer: Starvation means thread is waiting for infinite time or for longer periods because of it is not getting lock or CPU time. Following are the reasons for thread starvation

1. Lower priority thread will never get a chance, because higher priority threads always exists in application.
2. Thread is waiting for indefinitely, because it is not able to get lock on thread.

Q51. What is the solution of Thread starvation?

Answer: Fairness is the solution for Starvation. To implement this we should use Lock objects provided by Java and locks only critical section of the code rather than entire method. You have to write code properly so it can never arrives a starvation situation.

Q52. What is a slipped condition?

Answer: Lets assume we have two threads ThreadA and ThreadB. Now both thread progress depends on a signal (boolean flag named goAhead). ThreadA checks the value of goAhead flag ant it is true. So it go ahead but not started yet. Now, ThreadB change the value of goAhead flag to false. A thread can only return from the wait() method if it reacquires the lock on the object it's waiting on.

The current thread must own this object's monitor. The thread releases ownership of this monitor and waits until another thread notifies threads waiting on this object's monitor to wake up either through a call to the notify method or the notifyAll method. The thread then waits until it can re-obtain ownership of the monitor and resumes execution.

Slipped condition can be avoided using properly placing thread synchronization.

Q53. What is the meaning of Lock Reentrance?

Answer: By default we are using re-entrant lock when we use synchronization block in Java. It means same thread has taken the lock and calling another synchronized method from a synchronized method. It is allowed. See below code example.

```
public class HadoopExam{

  public synchronized first(){
    second();
  }

  public synchronized second(){
    System.out.println("Its allowed");
  }
}
```

Q54. When you are using Lock object, what is the best place in a method to call unlock() method?

Answer: It is possible your code can throw an exception. And thread does not release a lock, because it cannot reach the code segment which does unlocking. Hence, it is always suggested to put unlock call in finally block of the method.

Q55. What problem is solved by read/write lock?

Answer: When there is a shared resource and many threads are reading them at the same time then there is no problem. We can use normal lock class. But what happen, when we want multiple threads reading and few threads are also updating the shared resource. In this case shared resource must be updated by single thread at a time. And while writing is going on than further reading and writing should be allowed. Such type of scenario can be handled using read/write lock.

Q56. What is a Counting Semaphore?

Ans : Java provides built in semaphore (which you can imagine a locked counter). 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.

Semaphores are often used to restrict the number of threads than can access some (physical or logical) resource.

Q57. What is a BlcokingQueue?

Answer: Lets assume you have 1000s of tasks (Runnable), you want to process it. And there is a ThreadPool of 10 threads(Consumer threads), which will process each tasks. We have a Queue (Task holder) with defined size of 100. So this queue can hold upto 100 tasks maximum at a time. Now at one end of the Queue you are keep adding the tasks (Runnable) using a Producer thread. And at other end ThreadPool consumer will dequeening tasks and processing the same. Main point here is , when Queue is full (100 tasks in a queue) , producer is blocked and cannot add more tasks to blocking queue. If Queue is empty than consumer thread cannot de-queue any tasks and will block until new tasks is added. Queue which has this behavior is called Blocking Queue.

QuickTechie