♦ Multithreading interview Q&A on differences between X and Y

Posted on February 28, 2015 by Arulkumaran Kumaraswamipillai — No Comments ↓

Q1. What is difference between 'Executor.submit()' and 'Executor.execute()' method ?
A1. The difference is that execute() doesn't return a "Future" object, so you can't wait for the completion of the Runnable, and get any exception it throws. The submit(…) method accepts Callable<V> and returns an instance of Future<V>, which you can use later in caller (or client) to retrieve the results of asynchronous computation.

Q2. What are the differences among CyclicBarrier, CountDownLatch and join in Java?

A2. join: “t1.join()” makes the current thread to wait for thread “t1” to complete.

join() waits for one thread to complete whereas CountDownLatch.await() allows N threads to wait until the countdown reaches 0. It can be used to make sure N threads start doing something at the same time.

The main difference between **CyclicBarrier** and **CountDownLatch** is that CyclicBarrier can be reused by calling reset() method which resets the barrier to its initial state whereas a CountDownLatch cannot be reused.

Q3. What is the difference between wait() and sleep() in Java?
A3. A waiting thread can be “woken up” by another thread calling notify on the monitor (i.e. lock) which is being waited on whereas a sleep cannot.

A wait (and notify/notifyAll) must happen within a block synchronized on the monitor (i.e. lock) object whereas sleep does not.

wait() is called on an object, whereas a sleep is called on a thread.

Q4. What is the difference between Runnable and Callable<V> interfaces in Java?
A4. The Callable<V> was introuduced in Java 5, and it is very similar to Runnable, except for:

- A Callable instance returns a result of type V, whereas a Runnable instance doesn’t.
- A Callable instance may throw checked exceptions, whereas a Runnable instance can’t throw checked exceptions.
- A Callable implementation needs to implement a call() method and a Runnable interface needs implement a run() method.

16 Technical Key Areas

```
1 <T> Future<T> submit(Callable<T> task);
2 Future<?> submit(Runnable task);
3 <T> Future<T> submit(Runnable task, T result);
```

Q5. What is the difference between start() and run() methods?
A5. The start() method starts the execution of the new thread and calls the run() method. The start() method returns immediately and the new thread normally continues until the run() method returns.

Calling run() directly just executes the code synchronously in the same thread without spawning a new worker thread, just as a normal method call.

Q6. What is the difference between daemon and non-daemon threads?
A6. Daemon threads in Java are those threads which run in background. Background threads performing house keeping tasks. These threads continue to execute even after the thread that spawned them exits or dies. For example, the JVM Garbage collector thread is a daemon thread. GC thread is a low priority thread which runs intermittently in the back ground doing the garbage collection operation.

When an application starts running, it creates a non-daemon thread, whose job is to execute the main() method. The JVM or process exits when the last non-daemon thread exits.

Q7. What is the difference between intrinsic lock and extrinsic locks in Java?
A7. Each Java class and object (i.e. instance of a class) has an **intrinsic lock** (aka monitor). Don't confuse this with **explicit lock** utility classes that were added in Java 1.5.

Explicit locks are laid out with 2 interfaces **Lock** and **ReadWriteLock**. It is more complicated to use it properly, and incorrect usage can lead to unexpected issues leading to deadlocks, thread starvation, etc. So, you need to remember the following best practices when using explicit locks.

- Release the explicit locks in a finally block.

- Favor intrinsic locks where possible to avoid bugs and to keep your code cleaner and easier to maintain.
- Use tryLock( ) if you don't want a thread waiting indefinitely to acquire a lock. This is similar to how databases prevent dead locks with wait lock timeouts.
- When using ReentrantLocks for frequent concurrent reads and occasional writes, be mindful of the possibility that a writer could wait a very long time sometimes forever) if there are constantly read locks held by other threads.

Both intrinsic and explicit locks in Java are **reentrant**. Reentrancy means that locks are acquired on a per-thread rather than per-invocation basis. In Java, both intrinsic and explicit locks are re-entrant.

Q8. What are the differences among Atomicity, Visibility, and Ordering in Java multithreading?
A8. Atomicity means an operation will either be completed or not done at all. Other threads will not be able to see the operation "in progress" — it will never be viewed in a partially complete state.

Visibility determines when the effects of one thread can be seen by another. In the absence of proper synchronization, the JVM may decide that you are reading a variable that you don't have to read again, and it can eliminate the repeated read as an optimization strategy.

Ordering determines when actions in one thread can be seen to occur out of order with respect to another. JVM will have a lot of freedom to reorder code in the absence of synchronization.

Q9. What is the difference between volatile and synchronized keywords in Java?
A9. The **volatile** keyword guarantees ordering, and prevents compiler or JVM from reordering of the code.

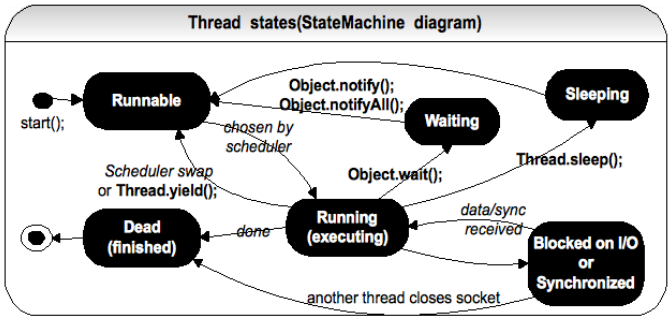
The volatile keyword is applied to variables of both primitives and objects, whereas the synchronized keyword is applied to

open all | close all
Career Making Know-
Job Hunting & Resum

only objects.

The volatile keyword only guarantees **visibility and ordering**, but **not atomicity**, whereas the synchronized keyword can guarantee both visibility and atomicity if done properly. So, the volatile variable has a limited use, and cannot be used in compound operations like incrementing a variable.

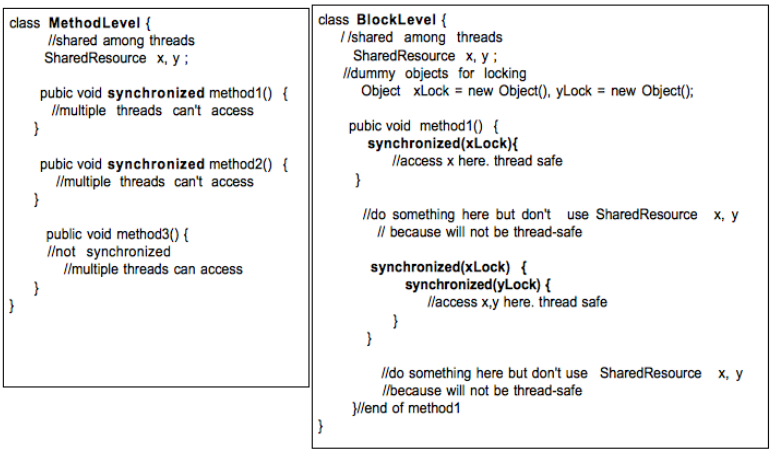
Q10. What are the different thread states?
A10. The **state chart diagram** below describes the thread states.



(**Diagram sourced from:** http://www.wilsonmar.com/1threads.htm)

- **Runnable** — waiting for its turn to be picked for execution by the thread scheduler based on thread priorities.
- **Running**: The processor is actively executing the thread code. It runs until it becomes blocked, or voluntarily gives up its turn with this static method *Thread.yield()*. Because of context switching overhead, *yield()* should not be used very frequently.
- **Waiting**: A thread is in a **blocked state** while it waits for some external processing such as file I/O to finish.
- **Sleeping**: Java threads are forcibly put to sleep (suspended) with this overloaded method: Thread.sleep(milliseconds), Thread.sleep(milliseconds, nanoseconds);
- **Blocked on I/O**: Will move to runnable after I/O condition like reading bytes of data etc changes.
- **Blocked on synchronization**: Will move to Runnable when a **lock is acquired.**
- **Dead**: The thread is finished working.

Q11. What is the difference between synchronized method and synchronized block?
A11. In Java programming, each object has a lock. A thread can acquire the lock for an object by using the **synchronized** keyword. The synchronized keyword can be applied in **method level** (coarse grained lock – can affect performance adversely) or **block level** of code (fine grained lock). Often using a lock on a method level is too coarse. Why lock up a piece of code that does not access any shared resources by locking up an entire method. Since each object has a lock, dummy objects can be created to implement block level synchronization. The block level is more efficient because it does not lock the whole method.



```
class MethodLevel {
      //shared among threads
      SharedResource  x, y ;

     pubic void synchronized method1()  {
       //multiple  threads  can't  access
    }

     pubic void synchronized method2()  {
       //multiple  threads  can't  access
    }

      public void method3() {
      //not  synchronized
         //multiple threads can access
    }
}
```

```
class BlockLevel {
    / /shared  among  threads
      SharedResource  x, y ;
    //dummy  objects  for  locking
       Object  xLock = new Object(), yLock = new Object();

     pubic void  method1()  {
        synchronized(xLock){
            //access x here. thread safe
      }

       //do  something  here  but  don't   use  SharedResource   x, y
          // because will not be thread-safe

        synchronized(xLock)  {
              synchronized(yLock) {
                  //access x,y here. thread safe
              }
        }

          //do something here but don't use  SharedResource   x, y
          //because will not be thread-safe
      }//end of method1
}
```

method level Vs block level locking

The JVM uses locks in conjunction with **monitors**. A monitor is basically a guardian who watches over a sequence of synchronized code and making sure only one thread at a time executes a synchronized piece of code. Each monitor is associated with an object reference. When a thread arrives at the first instruction in a block of code it must obtain a lock on the referenced object. The thread is not allowed to execute the code until it obtains the lock. Once it has obtained the lock, the thread enters the block of protected code. When the thread leaves the block, no matter how it leaves the block, it releases the lock on the associated object.

Q. Why synchronization is important?
A. Without synchronization, it is possible for one thread to modify a shared object while another thread is in the process of using or updating that object's value. This often causes dirty data and leads to significant errors. The disadvantage of synchronization is that it can cause deadlocks when two threads are waiting on each other to do something. Also synchronized code has the overhead of acquiring lock, which can adversely affect the performance.

Popular Posts

♦ 11 Spring boot interview questions & answers

823 views

♦ Q11-Q23: Top 50+ Core on Java OOP Interview Questions & Answers

765 views

18 Java scenarios based interview Questions and Answers

399 views

001A: ♦ 7+ Java integration styles & patterns interview questions & answers

388 views

01b: ♦ 13 Spring basics Q8 – Q13 interview questions & answers

295 views

♦ 7 Java debugging interview questions & answers

293 views

01: ♦ 15 Ice breaker questions asked 90% of the time in Java job interviews with hints

285 views

♦ 10 ERD (Entity-Relationship Diagrams) Interview Questions and Answers

279 views

♦ Q24-Q36: Top 50+ Core on Java classes, interfaces and generics interview questions & answers

239 views

001B: ♦ Java architecture & design concepts interview questions & answers

201 views

Bio | Latest Posts

Arulkumaran Kumaraswamipillai

Mechanical Eng to freelance Java developer in 3 yrs. Contracting since 2003, and attended 150+ Java job interviews, and often got 4 - 7 job offers to choose from. It pays to prepare. So, published Java interview Q&A books via Amazon.com in 2005, and sold 35,000+ copies. Books are outdated and replaced with this subscription based site.**945+** paid members. join my LinkedIn Group. **Reviews**

About Arulkumaran Kumaraswamipillai

Mechanical Eng to freelance Java developer in 3 yrs. Contracting since 2003, and attended 150+ Java job interviews, and often got 4 - 7 job offers to choose from. It pays to prepare. So, published Java interview Q&A books via Amazon.com in 2005, and sold 35,000+ copies. Books are outdated and replaced with this subscription based site.**945+** paid members. join my LinkedIn Group. **Reviews**

‹ Understanding atomicity with a Java example

How to become a self-taught Java (or any) developer everyone wants to hire? ›

Posted in Differences Between X & Y, member-paid

Leave a Reply

Logged in as geethika. Log out?

Comment

Post Comment

Empowers you to open more doors, and fast-track

Technical Know Hows

☀ Java generics in no time ☀ Top 6 tips to transforming your thinking from OOP to FP ☀ How does a HashMap internally work? What is a hashing function?
☀ 10+ Java String class interview Q&As ☀ Java auto un/boxing benefits & caveats ☀ Top 11 slacknesses that can come back and bite you as an experienced Java developer or architect

Non-Technical Know Hows

☀ 6 Aspects that can motivate you to fast-track your career & go places ☀ Are you reinventing yourself as a Java developer? ☀ 8 tips to safeguard your Java career against offshoring ☀ My top 5 career mistakes

Prepare to succeed

☀ Turn readers of your Java CV go from “Blah blah” to “Wow”? ☀ How to prepare for Java job interviews? ☀ 16 Technical Key Areas ☀ How to choose from multiple Java job offers?

Select Category

© Disclaimer