Important points about Thread in Java

java.lang.Thread class but JVM plays an important role of all Thread activities. Thread is used to execute task in parallel and this task is coded inside run() method of Runnable interface. You can create Thread in Java programming language by either extending Thread class , implementing Runnable or by using Executor framework in Java. Remember Runnable doesn't represent Thread actually its a task which is executed by Thread. Read more about extends Thread vs implements Runnable here.

1. Thread in Java represent an independent path of execution. (classic definition but I still like it). Thread is represented by

2. During its life time thread remains on various Thread states like NEW, RUNNABLE, BLOCKED, WAITING, TIMED_WAITING which describe what thread is doing. NEW means thread is just created but not yet stated, RUNNABLE means thread is started but waiting for CPU to be assigned by thread scheduler. BLOCKED, WAITING and TIMED_WAITING means thread is not doing anything instead its been blocked and waiting for IO to finished, class or object lock, or any other thread etc.

3. There are two kinds of Thread in Java daemon or non daemon (also called user threads). Java programs runs until there is at least one non-daemon thread exists. First non-daemon thread started by JVM is main thread which is created by JVM and responsible for executing code insidemain method in Java. This is called "VM thread" in HotSpot Virtual Machine. Any thread created using java.lang.Thread start() methods from main thread is by default non-daemon but you can make it daemon by calling setDaemon(true) method. Newly created thread in Java inherits daemon property from the thread which creates it. Since main thread is non-daemon, any thread created by it by default remains non-daemon.

4. Every Java threads has priority and name. You can set priority and assign meaningful name while creating object of java.lang.Thread class in Java. its recommend practice to give every thread a meaningful name while creating it , it helps later when you debug your Java program or take thread dump for analysis. Otherwise Java will give your Thread default name like "Thread-number" if Thread is created using java.lang.Thread or "pool-number-thread-number"if Thread is created using ThreadFactory. In Java higher priority thread get preference in Execution over lower priority thread. you can check priority by using method like getProiroty() from thread class.

5.Creation of thread is a time-consuming job so having a Thread pool for performing task concurrently is modern day requirement of performance and scalability. Java 5 provides Executor framework which encapsulate task of creating and managing thread from application code. Consider using Thread pool if your application requires to handle load. In web and application server manages this thread pool because each request is processed in its own thread.

6. Thread.sleep() method is used to pause thread for specified duration. It is an overloaded method defined in java.lang.Thread class. On the other hand Thread.join() is used to wait for another thread to complete its task before running and yield() method is used to relinquish CPU so that other thread can acquire it. See difference between sleep, wait and yield for details.

7. wait() and notify() methods are used to communicate between two threads i.e. for interthread communication in Java. Always check condition of wait() method in loop and call them from synchronized context. wait() is a method which is defined in object class, and puts the current thread on hold and also releases the monitor (lock) held by this thread, while notify() and notifyAll() methods notifies all thread waiting on that monitor. There is no guarantee which thread will picked up by thread scheduler and given CPU to execute because of of notification. To learn more about how to use wait, notify and notifyAll method to achieve inter-thread communication, see my post about solving producer consumer problem in Java using wait and notify method.

8. Thread scheduling is done by Thread Scheduler which is platform dependent and stays inside JVM. There is no known way to control thread scheduler from Java and many of thread related decision is done by scheduler like if there are many threads is waiting then which thread will be awarded CPU.

9. `Thread.isActive()` method is used to check whether a thread is active or not. A thread is said to be active until it has not finished either by returning from `run()` method normally or due to any exception. `Thread.holdsLock()` method is used to check if a thread holds a lock or not. See how to check if thread holds lock or not for more details.

10. Every thread in Java has its own stack, which is used to store local variables and method calls. Size of this stack can be controlled using `-XX:ThreadStackSize` JVM option e.g. `-XX:ThreadStackSize=512`.

11. Java 5 introduced another way to define task for threads in Java by using `Callable`interface. It's similar to `Runnable` interface but provides some more capability e.g. it can return result of task execution, which was not possible in Runnable, because return type of `run()` method was void. Like its predecessor it define a `call()` method which can return`Future<T>` object, you can call `get()` method on this object to get the result of task execution. To learn more about Callable, please see difference between Runnable and Callable interface in Java.

12. Java provides `interrupt()` method to interrupt a thread in Java. You can interrupt a running thread, waiting thread or sleep thread. This is the control Java provides to prevent a blocked or hanged thread. Once you interrupt a thread, it will also throw `InterruptedException`, which is a checked exception to ensure that your code should take handle interrupts

13. Java provides two ways to achieve mutual exclusion in your code, either by using synchronized keyword or by using`java.util.concurrent.lock` implementations. Former is more popular and oldest way to achieve mutual exclusive code but `Lock`interface is more powerful and provides fine grained control and only available from Java 5, Tiger. You can use synchronized keyword to either make an entire method mutual exclusive or only critical section by declaring a synchronized block. Any thread needs to hold monitor or lock, required by that critical section in order to enter into synchronized block or method, they release that lock, once they exit, either normally or abruptly due to any error. Acquisition and release of monitor is done by Java itself, so its safe and easy for Java programmer, on the other hand if you decide to use Lock interface, you need to explicitly acquire lock and release it, this requires more caution. Popular idiom is to release the lock in finally block. See my post how to use ReentrantLock in Java for code example and few more details.

14. One more thing to know about Thread in Java is that its not started when you create object ofThread class e.g.

```
Thread t = new Thread();
```

In fact thread is started when you call `start()` method of `java.lang.Thread` class e.g. t.start() will start the thread. It puts your thread in `RUNNABLE` state and when thread scheduler assign CPU to this thread, it executes `run()` method. By default `run()` method Thread class is empty, so you need to override it to do some meaningful task.

Diffrence Between Thread and Runnable

1) Implementing Runnable is the preferred way to do it. Here, you're not really specializing or modifying the thread's behavior. You're just giving the thread something to run. That means composition is the better way to go.

2) Java only supports single inheritance, so you can only extend one class.

3) Instantiating an interface gives a cleaner separation between your code and the implementation of threads.

4) Implementing Runnable makes your class more flexible. If you extend thread then the action you're doing is always going to be in a thread. However, if you extend Runnable it doesn't have to be. You can run it in a thread, or pass it to some kind of executor service, or just pass it around as a task within a single threaded application.

5) By extending Thread, each of your threads has a unique object associated with it, whereas implementing Runnable, many threads can share the same runnable instance.

What are difference between wait and sleep in Java

Wait vs sleep in Java

Differences between wait and sleep method in Java Thread is one of the very old question asked in Java interviews. Though both wait and sleep puts thread on waiting state, they are completely different in terms of behaviour and use cases. Sleep is meant for introducing pause, releasing CPU and giving another thread opportunity to execute while wait is used for inter thread communication, by using `wait()` and `notify()` method two threads can communicate with each other which is key to solve many Concurrent problems like Produce consumer issue, Dining philosopher issue and to implement several Concurrency designs. In this tutorial we will see

- What is wait method in Java
- What is Sleep method in Java
- Difference between wait and sleep in Java
- Where to use wait and sleep in Java

What is wait and sleep method in Java

Wait method is defined in Object class and it available to all object,`wait()` method is always discussed along with its counterpart`notify()` and `notifyAll()` method and used in inter thread communication in Java. wait method puts a thread on wait by checking some condition like in Producer Consumer problem, producer thread should wait if Queue is full or Consumer thread should wait if Queue is empty. `notify()` method is used to wake up waiting thread by communicating that waiting condition is over now for example once producer thread puts an item on empty queue it can notify Consumer thread that Queue is not empty any more. On the other hand `Sleep()` method is used to introduce pause on Java application. You can put a Thread on sleep, where it does not do anything and relinquish the CPU for specified duration. When a Thread goes to Sleep it can be either wake up normally after sleep duration elapsed or it can be woken up abnormally by interrupting it.

Difference between Wait and Sleep method in Java Thread

In last section we saw *what is wait and sleep method* and in this section we will see what are differences between wait and sleep method in Java. As I told before apart from waiting they are completely different to each other:

1) First and most important difference between Wait and sleep method is that wait method must be called from synchronized context i.e. from synchronized method or block in Java. If you call wait method without synchronization, it will throw`IllegalMonitorStateException` in Java. On the other hand there is no requirement of synchronization for calling sleep method , you can call it normally.

2) Second worth noting difference between wait and sleep method is that, wait operates on `Object` and defined in `Object` class while sleep operates on current Thread and defined in java.lang.Thread class.

3) Third and another significant difference between wait and sleep in Java is that, `wait()` method releases the lock of object on which it has called, it does release other locks if it holds any while sleep method of Thread class does not release any lock at all.

4) wait method needs to be called from a loop in order to deal with false alarm i.e. waking even though waiting condition still holds true, while there is no such thing for sleep method in Java. its better not to call Sleep method from loop.

here is code snippet for calling wait and sleep method in Java

```
synchronized(monitor)
while(condition == true){ monitor.wait())  //releases monitor lock

Thread.sleep(100); //puts current thread on Sleep
```

5) One more difference between wait and sleep method which is not as significant as previous ones is that `wait()` is a non static method while `sleep()` is static method in Java.

Where to use wait and sleep method in Java

By reading properties and behavior of wait and sleep method it's clear that `wait()` method should be used in conjunction with`notify()` or `notifyAll()` method and intended for communication between two threads in Java while `Thread.sleep()`method is a utility method to introduce short pauses during program or thread execution. Given the requirement of synchronization for wait, it should not be used just to introduce pause or sleep in Java

Process vs Thread in Java

In this section we will see some more differences between Process vs Thread in Java to get a clear idea about What is process and What is Thread in Java :

1) Both process and Thread are independent path of execution but one process can have multiple Threads.

2) Every process has its own memory space, executable code and a `unique process identifier (PID)` while every thread has its own stack in Java but it uses process main memory and share it with other threads.

3) Threads are also refereed as task or `light weight process (LWP)` in operating system

4) Threads from same process can communicate with each other by using Programming language construct like wait and notify in Java and much simpler than inter process communication.

5) Another difference between Process and Thread in Java is that it's How Thread and process are created. It's easy to create Thread as compared to Process which requires duplication of parent process.

6) All Threads which is part of same process share system resource like `file descriptors` , Heap Memory and other resource but each Thread has its own Exception handler and own stack in Java.

There were some of the fundamental difference between Process and Thread in Java. Whenever you talk about Process vs Thread, just keep in mind that one process can spawn multiple Thread and share same memory in Java. Each thread has its own stack

Difference between synchronized block and method in Java
Synchronized block and synchronized methods are two ways to use synchronized keyword in Java and implement mutual exclusion on critical section of code. Since Java is mainly used to write multi-threading programs, which present various kinds of thread related issues like thread-safety, deadlock and race conditions, which plagues into code mainly because of poorunderstanding of synchronization mechanism provided by Java programming language. Java provides inbuilt `synchronized`and volatile keyword to achieve synchronization in Java. Main*difference between synchronized method and synchronizedblock* is selection of lock on which critical section is locked. Synchronized method depending upon whether its a static method or non static locks on either `class level lock` or `object lock`. Class level lock is one for each class and represented by class literal e.g. `Stirng.class`. Object level lock is provided by current object e.g. `this` instance, You should never mix static and non static synchronized method in Java.. On the other hand synchronized block locks on monitor evaluated by expression provided as parameter to synchronized block. In next section we will see an example of both synchronized method and synchronized block to understand this difference better.

Difference between synchronized method vs block in Java

Here are Some more differences between synchronized method and block in Java based upon experience and syntactical rules of synchronized keyword in Java. Though both block and method can be used to provide highest degree of synchronization in Java, use of synchronized block over method is considered as better Java coding practices.

1) One significant difference between synchronized method and block is that,
Synchronized block generally reduce scope of lock. As scope of lock is inversely proportional to performance, its always better to lock only critical section of code. One of the best example of using synchronized block is double checked locking in Singleton pattern where instead of locking whole`getInstance()` method we only lock critical section of code which is used to create Singleton instance. This improves performance drastically because locking is only required one or two times.

2) Synchronized block provide granular control over lock, as you can use arbitrary any lock to provide mutual exclusion to critical section code. On the other hand synchronized method always lock either on current object represented by this keyword or class level lock, if its static synchronized method.

3) Synchronized block can throw throw java.lang.NullPointerException if expression provided to block as parameter evaluates to null, which is not the case with synchronized methods.

4) In case of synchronized method, lock is acquired by thread when it enter method and released when it leaves method, either normally or by throwing Exception. On the other hand in case of synchronized block,
thread acquires lock when they enter synchronized block and release when they leave synchronized block.

Synchronized method vs synchronized block Example in Java

Here is an example of sample class which shows on which object synchronized method and block are locked and how to use them :

```
/**
  * Java class to demonstrate use of synchronization method and block in Java
  */
public class SycnronizationExample{


    public synchronized void lockedByThis(){
        System.out.println(" This synchronized method is locked by current"
instance of object i.e. this");
    }

    public static synchronized void lockedByClassLock(){
        System.out.println("This static synchronized method is locked by class
level lock of thisclass i.e. SychronizationExample.class");

    }

    public void lockedBySynchronizedBlock(){
        System.err.println("This line is executed without locking");

        Object obj = String.class; //class level lock of Stirng class

        synchronized(obj){
            System.out.println("synchronized block, locked by lock represented
using objvariable");
        }
    }
```

```
}
```

That's all on difference between synchronized method and block in Java. Favoring synchronized block over method is one of the Java best practices to follow as it reduces scope of lock and improves performance. On the other hand using synchronized method are rather easy but it also creates bugs when you mix non static and static synchronized methods, as both of them are locked on different monitors and if you use them to synchronize access of shared resource, it will most likely break

Why String is Immutable or Final in Java

String is Immutable in Java because String objects are cached in String pool. Since cached String literal is shared between multiple client there is always a risk, where one client's action would affect all other client. For example, if one client changes value of String "Test" to "TEST", all other client will also see that value as explained in first example. Since caching of String objects was important from performance reason this risk was avoided by making String class Immutable. At the same time, *String was made final* so that no one can compromise invariant of String class e.g. Immutability, Caching, hascode calculation etc by extending and overriding behaviors. Another reason of *why String class is immutable* could de due to HashMap.
Since Strings are very popular as HashMap key, it's important for them to be immutable so that they can retrieve the value object which was stored in HashMap. Since HashMap works in principle of hashing, which requires same has value to function properly. Mutable String would produce two different hashcode at the time of insertion and retrieval if contents of String was modified after insertion, potentially losing the value object in map. If you are an Indian cricket fan, you may be able to correlate with my next sentence. String is VVS Laxman of Java, i.e. very very special class. I have not seen a single Java program which is written without using String. That's why solid understanding of String is very important for a Java developer. Important and popularity of String as data type, transfer object and mediator has also make it popular on Java interviews. Why String is immutable in Java is one of the most frequently asked String Interview questions in Java, which starts with discussion of, What is String, How String in Java is different than String in C and C++, and then shifted towards what is immutable object in Java , what are the benefits of immutable object, why do you use them and which scenarios should you use them. This question some time also asked as *"Why String is final in Java"*. e

Why String is Final in Java

As I said, there could be many possible answer of this question, and only designer of String class can answer it with confidence, I think following two reasons make a lot of sense on why String class is made Immutable or final in Java : 1) Imagine String pool facility without making string immutable , its not possible at all because in case of string pool one string object/literal e.g. "Test" has referenced by many reference variables , so if any one of them change the value others will be automatically gets affected i.e. lets say

String A = "Test"
String B = "Test"

Now String B called `"Test".toUpperCase()` which change the same object into "TEST" , so A will also be "TEST" which is not desirable.

2)String has been widely used as parameter for many Java classes e.g. for opening network connection, you can pass hostname and port number as string , you can pass database URL as string for opening database connection, you can open any file in Java by passing name of file as argument to File I/O classes.

In case, if String is not immutable, this would lead serious security threat , I mean some one can access to any file for which he has authorization, and then can change the file name either deliberately or accidentally and gain access of those file. Because of immutability, you don't need to worry about those kind of threats. This reason also gel with,Why String is final in Java, by making `java.lang.String` final, Java designer ensured

that no one overrides any behavior of String class.

3)Since String is immutable it can safely shared between many threads ,which is very important for multithreaded programming and to avoid any synchronization issues in Java, Immutability also makes String instance thread-safe in Java, means you don't need to synchronize String operation externally. Another important point to note about String ismemory leak caused by SubString, which is not a thread related issues but something to be aware of.

4) Another reason of Why String is immutable in Java is to allow String to cache its hashcode , being immutable String in Java caches its hashcode, and do not calculate every time we call hashcode method of String, which makes it very fast as hashmap key to be used in hashmap in Java. This one is also suggested by Jaroslav Sedlacek in comments below. In short because String is immutable, no one can change its contents once created which guarantees hashCode of String to be same on multiple invocation.

5) Another good reason of Why String is immutable in Java suggested by Dan Bergh Johnsson on comments is: The absolutely most important reason that String is immutable is that it is used by the class loading mechanism, and thus have profound and fundamental security aspects. Had String been mutable, a request to load "`java.io.Writer`" could have been changed to load "`mil.vogoon.DiskErasingWriter`"

Security and String pool being primary reason of making String immutable, I believe there could be some more very convincing reasons as well, Please post those reasons as comments and I will include those on this post. By the way, above reason holds good to answer, another Java interview questions "Why String is final in Java". Also to be immutable you have to be final, so that your subclass doesn't break immutability. what do you guys think ?

What is Synchronization in Java

Synchronization in Java is an important concept since Java is a multi-threaded language where multiple threads run in parallel to complete program execution. In multi-threaded environment *synchronization of Java object or synchronization of Java class becomes extremely important*. Synchronization in Java is possible by using Java keywords*"synchronized"* and *"volatile"*. Concurrent access of shared objects in Java introduces to kind of errors: thread interference and memory consistency errors and to avoid these errors you need to properly synchronize your Java object to allow mutual exclusive access of critical section to two threads. By the way This Java Synchronization tutorial is in continuation of my article How HashMap works in Java and difference between HashMap and Hashtable in Java if you haven't read already you may find some useful information based on my experience in Java Collections.

Why do we need Synchronization in Java?

If your code is executing in multi-threaded environment, you need synchronization for objects, which are shared among multiple threads, to avoid any corruption of state or any kind of unexpected behavior. Synchronization in Java will only be needed if shared object is mutable. if your shared object is either read only or immutable object, than you don't need synchronization, despite running multiple threads. Same is true with what threads are doing with object if all the threads are only reading value then you don't require synchronization in Java. JVM guarantees that *Java synchronized code will only be executed by one thread at a time*. In Summary `Java synchronized Keyword` providesfollowing functionality essential for concurrent programming :

1) `synchronized` keyword in Java provides locking, which ensures mutual exclusive access of shared resource and prevent data race.

2) `synchronized` keyword also prevent reordering of code statement by compiler which can cause subtle concurrent issue if we don't use `synchronized` or `volatile` keyword.

3) `synchronized` keyword involve locking and unlocking. before entering into synchronized method or block thread needs to acquire the lock, at this point it reads data from main memory than cache and when it release the lock, it flushes write operation into main memory which eliminates memory inconsistency errors.

Synchronized keyword in Java

Prior to Java 1.5 `synchronized` keyword was only way to provide synchronization of shared object in Java. Any code written by using synchronized block or enclosed inside synchronized method will be mutually exclusive, and can only be executed by one thread at a time. You can have both static synchronized method and non static synchronized method and synchronized blocks in Java but we can not have synchronized variable in java. Using synchronized keyword with variable is illegal and will result in compilation error. Instead of `synchronized` variable in Java, you can have java `volatile`variable, which will instruct JVM threads to read value of volatile variable from main memory and don't cache it locally. *Block synchronization in Java is preferred over method synchronization in Java* because by using block synchronization, you only need to lock the critical section of code instead of whole method. Since synchronization in Java comes with cost of performance, we need to synchronize only part of code which absolutely needs to be synchronized.

Example of Synchronized Method in Java

Using synchronized keyword along with method is easy just apply synchronized keyword in front of method. What we need to take care is that static synchronized method locked on class object lock and non static synchronized method locks on current `object (this)`. So it's possible that both static and non static java synchronized method running in parallel. This is the common mistake a naive developer do while writing Java synchronized code.

```
public class Counter{

  private static int count = 0;

  public static synchronized int getCount(){
    return count;
  }

  public synchoronized setCount(int count){
     this.count = count;
  }

}
```

In this example of *Java synchronization code is not properly synchronized* because both `getCount()` and `setCount()` are not getting locked on same object and can run in parallel which may results in incorrect count. Here `getCount()` will lock in `Counter.class` object while `setCount()` will lock on current object (this). To make this code properly synchronized in Java *you need to either make both method static or non static or use java synchronized block instead of java synchronized method*.By the way this is one of the common mistake Java developers make while synchronizing their code.

Example of Synchronized Block in Java

Using synchronized block in java is also similar to using synchronized keyword in methods. Only important thing to note here is that if object used to lock synchronized block of code, `Singleton.class` in below example is `null` then Java synchronized block will throw a NullPointerException.

```
public class Singleton{

private static volatile Singleton _instance;

public static Singleton getInstance(){
   if(_instance == null){
```

```
            synchronized(Singleton.class){
              if(_instance == null)
              _instance = new Singleton();
            }
    }
    return _instance;
}
```

This is a classic example of double checked locking in Singleton. In this example of Java synchronized code, we have made only critical section (part of code which is creating instance of singleton) synchronized and saved some performance. If you make whole method synchronized than every call of this method will be blocked, while you only need blocking to create singleton instance on first call. By the way, this is not the only way to write threadsafe singleton in Java. You can use Enum, or lazy loading to avoid thread-safety issue during instantiation. Even above code will not behave as expected because prior to Java 1.5, double checked locking was broker and even with volatile variable you can view half initialized object. Introduction of Java memory model and happens before guarantee in Java 5 solves this issue. To read more about Singleton in Java see that.

Important points of synchronized keyword in Java

1. Synchronized keyword in Java is used to provide mutual exclusive access of a shared resource with multiple threads in Java. Synchronization in Java guarantees that, no two threads can execute a synchronized method which requires same lock simultaneously or concurrently.

2. You can use java `synchronized` keyword only on synchronized method or synchronized block.

3. When ever a thread enters into java synchronized method or block it acquires a lock and whenever it leaves java synchronized method or block it releases the lock. Lock is released even if thread leaves `synchronized` method after completion or due to any Error or Exception.

4. Java Thread acquires an object level lock when it enters into an instance synchronized java method and acquires a class level lock when it enters into static synchronized java method.

5. Java synchronized keyword is re-entrant in nature it means if a java synchronized method calls another synchronized method which requires same lock then current thread which is holding lock can enter into that method without acquiring lock.

6. Java Synchronization will throw NullPointerException if object used in java synchronized block is null e.g. `synchronized (myInstance)` will throws `java.lang.NullPointerException` if myInstance is `null`.

7. One Major disadvantage of Java synchronized keyword is that it doesn't allow concurrent read, which can potentially limit scalability. By using concept of lock stripping and using different locks for reading and writing, you can overcome this limitation of synchronized in Java. You will be glad to know that`java.util.concurrent.locks.ReentrantReadWriteLock` provides ready made implementation of `ReadWriteLock` in Java.

8. One more limitation of java synchronized keyword is that it can only be used to control access of shared object within the same JVM. If you have more than one JVM and need to synchronized access to a shared file system or database, the Java `synchronized` keyword is not at all sufficient. You need to implement a kind of global lock for that.

9. Java synchronized keyword incurs performance cost. Synchronized method in Java is very slow and can degrade performance. So use synchronization in java when it absolutely requires and consider using java synchronized block for synchronizing critical section only.

10. Java synchronized block is better than java synchronized method in Java because by using synchronized block you can only lock critical section of code and avoid locking whole method which can possibly degrade performance. A good example of java synchronization around this concept is getInstance() method Singleton class. See here.

11. Its possible that both static synchronized and non static synchronized method can run simultaneously or concurrently because they lock on different object.

12. From java 5 after change in Java memory model reads and writes are atomic for all variables declared using volatile keyword (including long and double variables) and simple atomic variable access is more efficient instead of accessing these variables via synchronized java code. But it requires more care and attention from the programmer to avoid memory consistency errors.

13. Java synchronized code could result in deadlock or starvation while accessing by multiple thread if synchronization is not implemented correctly. To know how to avoid deadlock in java see here.

14. According to the Java language specification you can not use Java synchronized keyword with constructor it's illegal and result in compilation error. So you can not synchronized constructor in Java which seems logical because other threads cannot see the object being created until the thread creating it has finished it.

15. You cannot apply java synchronized keyword with variables and can not use java volatile keyword with method.

16. `Java.util.concurrent.locks` extends capability provided by java synchronized keyword for writing more sophisticated programs since they offer more capabilities e.g. Reentrancy and interruptible locks.

17. Java synchronized keyword also synchronizes memory. In fact java synchronized synchronizes the whole of thread memory with main memory.

18. Important method related to synchronization in Java are `wait()`, `notify()` and `notifyAll()` which is defined in Object class. Do you know, why they are defined in `java.lang.object` class instead of `java.lang.Thread`? You can find some reasons, which make sense.

19. Do not synchronize on non final field on synchronized block in Java. because reference of non final field may change any time and then different thread might synchronizing on different objects i.e. no synchronization at all. example of synchronizing on non final field :

```
private String lock = new String("lock");

synchronized(lock){
    System.out.println("locking on :"  + lock);
}
```

any if you write synchronized code like above in java you may get warning "Synchronization on non-final field" in IDE like Netbeans and InteliJ

20. Its not recommended to use String object as lock in java synchronized block because string is immutable object and literal string and interned string gets stored in String pool. so by any chance if any other part of code or any third party library used same String as there lock then they both will be locked on same object despite being completely unrelated which could result in unexpected behavior and bad performance. instead of String object its advised to use new Object() for Synchronization in Java on synchronized block.

```
private static final String LOCK = "lock";   //not recommended
private static final Object OBJ_LOCK = new Object(); //better

public void process() {
   synchronized(LOCK) {
      ........
   }
}
```

21. From Java library Calendar and SimpleDateFormat classes are not thread-safe and requires external synchronization in Java to be used in multi-threaded environment.

Probably *most important point about Synchronization in Java* is that, in the absence of synchronized keyword or other construct e.g. volatile variable or atomic variable, compiler, JVM and hardware are free to make optimization, assumption, reordering or caching of code and data, which can cause subtle concurrency bugs in code. By introducing synchronization by using volatile, atomic variable or synchronized keyword, we instruct compiler and JVM to not to do that.

Update 1: Recently I have been reading several Java Synchronization and Concurrency articles in internet and I come across jeremymanson's blog which works in google and has worked on JSR 133 Java Memory Model, I would recommend some of this blog post for every java developer, he has covered certain details about concurrent programming , synchronization and volatility in simple and easy to understand language, here is the link atomicity, visibility and ordering.

Update 2: I am grateful to my readers, who has left some insightful comments on this post. They have shared lots of good information and experience and to provide them more exposure, I am including some of there comments on main article, to benefit new readers.

@Vikas wrote
Good comprehensive article about `synchronized` keyword in Java. to be honest I have never read all these details about synchronized block or method at one place. you may want to highlight some limitation of synchronized keyword in Java which is addressed by explicit locking using new concurrent package and Lock interface :

1. synchronized keyword doesn't allow separate locks for reading and writing. as we know that multiple thread can read without affecting thread-safety of class, synchronized keyword suffer performance due to contention in case of multiple reader and one or few writer.

2. if one thread is waiting for lock then there is no way to time out, thread can wait indefinitely for lock.
3. on similar note if thread is waiting for lock to acquired there is no way to interrupt the thread.

All these limitation of `synchronized` keyword is addressed and resolved by using `ReadWriteLock` and `ReentrantLock` in Java 5.

@George wrote
Just my 2 cents on your great list of Java Synchronization facts and best practices :

1) synchronized keyword in internally implemented using two byte code instructions `MonitorEnter` and `MonitorExit`, this is generated by compiler. Compiler also ensures that there must be a `MonitorExit` for every MonitorEnter in different code path e.g. normal execution and abrupt execution, because of Exception.

2) `java.util.concurrent` package different locking mechanism than provided by synchronized keyword, they mostly used `ReentrantLock`, which internally use CAS operations, volatile variables and atomic variables to get better performance.

3) With `synchronized` keyword, you have to leave the lock, once you exist a synchronized method or block, there is no way you can take the lock to other method. `java.util.concurrent.locks.ReentrantLock` solves this problem by providing control for acquiring and releasing lock, which means you can acquire lock in method A and can release in method B, if they both needs to be locked in same object lock. Though this could be risky as compiler will neither check nor warn you about any accidental leak of locks. Which means, this can potentially block other threads, which are waiting for same lock.

4) Prefer `ReentrantLock` over `synchronized` keyword, it provides more control on lock acquisition, lock release and better performance compared to synchronized keyword.

5) Any thread trying to acquire lock using synchronized method will block indefinitely, until lock is available. Instead this, `tryLock()` method of `java.util.concurrent.locks.ReentrantLock` will not block if lock is not

available.
Having said that, I must say, lots of good information.

Difference between Thread and Runnable interface in Java

Thread vs Runnable in Java
Here are some of my thoughts on whether I should use Thread or Runnable for implementing task in Java, though you have another choice as "`Callable`" for implementing thread which we will discuss later.

1) Java doesn't support multiple inheritance, which means you can only extend one class in Java so once you extended Thread class you lost your chance and can not extend or inherit another class in Java.

2) In Object oriented programming extending a class generally means adding new functionality, modifying or improving behaviors. If we are not making any modification on Thread than use Runnable interface instead.

3) Runnable interface represent a Task which can be executed by either plain `Thread` or `Executors` or any other means. so logical separation of Task as `Runnable` than Thread is good design decision.

4) Separating task as Runnable means we can reuse the task and also has liberty to execute it from different means. since you can not restart a Thread once it completes. again Runnable vs Thread for task, Runnable is winner.

5) Java designer recognizes this and that's why Executors accept Runnable as Task and they have worker thread which executes those task.

6) Inheriting all Thread methods are additional overhead just for representing a Task which can can be done easily with Runnable.

Atomicity, Visibility and Ordering

(Note: I've cribbed this from my doctoral dissertation. I tried to edit it heavily to ease up on the mangled academic syntax required by thesis committees, but I may have missed some / badly edited in places. Let me know if there is something confusingly written or just plain confusing, and I'll try to untangle it.)

There are these three concepts, you see. And they are fundamental to correct concurrent programming. When a concurrent program is not correctly written, the errors tend to fall into one of the three categories: *atomicity*, *visibility*, or *ordering*.

Atomicity deals with which actions and sets of actions have indivisible effects. This is the aspect of concurrency

most familiar to programmers: it is usually thought of in terms of mutual exclusion. *Visibility* determines when the effects of one thread can be seen by another. *Ordering* determines when actions in one thread can be seen to occur out of order with respect to another. Let's talk about them.

Everyone doing any serious concurrent programming knows what atomicity is (or will when I describe it) — I'm just putting it in for completeness's sake. If an action is (or a set of actions are) *atomic*, its result must be seen to happen ``all at once'', or indivisibly. Atomicity is the traditional bugbear of concurrent programming. Enforcing it usually means using locking to enforce mutual exclusion. To see atomicity in action (or in inaction, perhaps), consider this code:

```
class BrokenBankAccount {
 private int balance;

 synchronized int getBalance() {
  return balance;
 }

 synchronized void setBalance(int x)
  throws IllegalStateException {
  balance = x;
  if (balance < 0) {
   throw new IllegalStateException("Negative Balance");
  }
 }
void deposit(int x) {
  int b = getBalance();
  setBalance(b + x);
 }
void withdraw(int x) {
  int b = getBalance();
  setBalance(b - x);
 }
}
```

Since all accesses to the shared variable `balance` are guarded by locks, this code is free of what are called *data races*, which are basically what happens when you access a variable concurrently without some use of synchronization or volatile or the like (one of those accesses has to be a write to be a data race in the true sense).

When code is free from data races, we say it is *correctly synchronized*. So the code is correct, right?

No, of course not. This code is not at all correct, in the sense that it doesn't necessarily do what we want it to do. Think about what happens if one thread calls `deposit(5)` and another calls `withdraw(5)`; there is an initial balance of 10. Ideally, at the end of these two calls, there would still be a balance of 10. However, consider what would happen if:

1. The `deposit()` method sees a value of 10 for the balance, then
2. The `withdraw()` method sees a value of 10 for the balance and withdraws 5, leaving a balance of 5, and finally
3. The `deposit()` method uses the balance it originally saw (10) to calculate a new balance of 15.

As a result of this lack of "atomicity", the balance is 15 instead of 10. This effect is often referred to as a *lost update*, because the withdrawal is lost. A programmer writing multi-threaded code must use synchronization carefully to avoid this sort of error. In Java, if the `deposit()` and `withdraw()` methods are declared `synchronized`, it will ensure that locks are held for their duration: the actions of those methods will be seen to take place atomically.

Atomicity, of course, is only guaranteed when all the threads use synchronization correctly. If someone comes along and decides to read the balance without acquiring a lock, it can end up with all sorts of confusing results.

Atomicity is the most common problem you get when using synchronization. It is a common mistake to think that it is the only problem; it is not. Here's another one:

What's visibility, you ask? Well, if an action in one thread is *visible* to another thread, then the result of that action can be observed by the second thread. In order to guarantee that the results of one action are observable to a second action, then you have to use some form of synchronization to make sure that the second thread sees what the first thread did.

(Note: when I say synchronization in this post, I don't actually mean locking. I mean anything that guarantees visibility or ordering in Java. This can include final and volatile fields, as well as class initialization and thread starts and joins and all sorts of other good stuff.)

Here's an example of a code with visibility problems:

```
class LoopMayNeverEnd {
  boolean done = false;
void work() {
    while (!done) {
      // do work
    }
```

```
  }
 void stopWork() {
    done = true;
  }
}
```

In this code, imagine that two threads are created; one thread calls `work`, and at some point, the other thread calls `stopWork` on the same object. Because there is no synchronization between the two, the thread in the loop may never see the update to `done` performed by the other thread. In practice, this may happen if the compiler detects that no writes are performed to `done` in the first thread; the compiler may decide that the program only has to read `done` once, transforming it into an infinite loop.

(By the way, "compiler" in this context doesn't mean `javac` — it actually means the JVM itself, which plays lots of games with your code to get it to run more quickly. In this case, if the compiler decides that you are reading a variable that you don't have to read, it can eliminate the read quite nicely. As described above.)

To ensure that this does not happen, you have to use a mechanism that provides synchronization between the two threads. In `LoopMayNeverEnd`, if you want to do this, you can declare`done` to be `volatile`. Conceptually, all actions on volatiles happen in a single order, where each read sees the last write in that order. In other words, the compiler can't prevent a read of a volatile from seeing a write performed by another thread.

There is a side issue here; some architectures and virtual machines may execute this program without providing a guarantee that the thread that executes `work` will ever give up the CPU and allow other threads to execute. This would prevent the loop from ever terminating because of scheduling guarantees, not because of a lack of visibility guarantees. This is typically called *cooperative multithreading*. The only implementation I know that does this is the Oracle VM — check the box for details.

There's one more problem that crops up:

Ordering constraints describe what order things are seen to occur. You only get intuitive ordering constraints by synchronizing correctly. Here's an example of when ordering problems can bite you:

```
class BadlyOrdered {
  boolean a = false;
  boolean b = false;

  void threadOne() {
    a = true;
    b = true;
```

```
  }
 boolean threadTwo() {
    boolean r1 = b; // sees true
    boolean r2 = a; // sees false
    boolean r3 = a; // sees true
    return (r1 && !r2) && r3; // returns true
  }

}
```

Consider what happens if `threadOne()` gets invoked in one thread and `threadTwo()` gets invoked on the same object in another. Would it be possible for `threadTwo()` to return the value`true`? If `threadTwo()` returns true, it means that the thread saw both updates by `threadOne`, but that it saw the change to `b` before the change to `a`.

Well, this code fragment does not use synchronization correctly, so surprising things can happen! It turns out that Java allows this result, contrary to what a programmer might have expected.

The assignments to `a` and `b` in `threadOne()` can be seen to be performed out of order. Compilers have a lot of freedom to reorder code in the absence of synchronization; they could either reorder the writes in `threadOne` or the reads in `threadTwo` freely.

Reason Why Wait , Notify and NotifyAll are in Object Class:

Here are some thoughts on why they should not be in Thread class which make sense to me :

1) Wait and notify is not just normal methods or synchronization utility, more than that they are communication mechanism between two threads in Java. And Object class is correct place to make them available for every object if this mechanism is not available via any java keyword like synchronized. Remember synchronized and wait notify are two different area and don't confuse that they are same or related. Synchronized is to provide mutual exclusion and ensuringthread safety of Java class like race condition while wait and notify are communication mechanism between two thread.

2)Locks are made available on per Object basis, which is another reason wait and notify is declared in Object class rather then Thread class.

3) In Java in order to enter critical section of code, Threads needs lock and they wait for lock, they don't know which threads holds lock instead they just know the lock is hold by some thread and they should wait for lock instead of knowing which thread is inside the synchronized block and asking them to release lock.
this analogy fits with wait and notify being on object class rather than thread in Java.

Wait vs Sleep vs Yield in Java

Difference between Wait and Sleep in Java
Main *difference between wait and sleep is that wait()* method release the acquired monitor when thread is waiting while Thread.sleep() method keeps the lock or monitor even if thread is waiting. Also wait method in java should be called from synchronized method or block while there is no such requirement for

sleep() method. Another difference is Thread.sleep() method is a static method and applies on current thread, while wait() is an instance specific method and only got wake up if some other thread calls notify method on same object. also in case of sleep, sleeping thread immediately goes to Runnable state after waking up while in case of wait, waiting thread first acquires the lock and then goes into Runnable state. So based upon your need if you require a specified second of pause use sleep() method or if you want to implement inter-thread communication use wait method.

here is list of difference between wait and sleep in Java :

1) wait is called from synchronized context only while sleep can be called without synchronized block. see Why wait and notify needs to call from synchronized method for more detail.

2) wait is called on Object while sleep is called on Thread. see Why wait and notify are defined in object class instead of Thread.

3) waiting thread can be awake by calling notify and notifyAll while sleeping thread can not be awaken by calling notify method.

4) wait is normally done on condition, Thread wait until a condition is true while sleep is just to put your thread on sleep.

5) wait release lock on object while waiting while sleep doesn't release lock while waiting.

Difference between yield and sleep in java
Major difference between yield and sleep in Java is that 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. Yield methoddoesn't guarantee that current thread will pause or stop but it guarantee that CPU will be relinquish by current Thread as a result of call to Thread.yield() method in java.

Sleep method in Java has two variants one which takes millisecond as sleeping time while other which takes both mill and nano second for sleeping duration.

sleep(long millis)

or

sleep(long millis,int nanos)

Cause the currently executing thread to sleep for the specified number of milliseconds plus the specified number of nanoseconds.

Example of Thread Sleep method in Java
Here is sample code example of Sleep Thread in Java. In this example we have put Main thread in Sleep for 1 second.

```
/*
 * Example of Thread Sleep method in Java
 */
public class SleepTest {
```

```
        public static void main(String... args){

                System.out.println(Thread.currentThread().getName() + " is going to
sleep for 1 Second");

                try {

                        Thread.currentThread().sleep(1000);

                } catch (InterruptedException e) {

                        // TODO Auto-generated catch block

                        e.printStackTrace();

                }

                System.out.println("Main Thread is woken now");

        }


}
```

Output:

```
main is going to sleep for 1 Second

Main Thread is woken now
```


10 points about Thread sleep() method in Java
I have listed down some important and worth to remember points about Sleep() method of Thread Class in Java:

1) Thread.sleep() method is used to pause the execution, relinquish the CPU and return it to thread scheduler.

2) Thread.sleep() method is a static method and always *puts current thread on sleep*.

3) Java has two variants of sleep method in Thread class one with one argument which takes milliseconds as duration for sleep and other method with two arguments one is millisecond and other is nanosecond.

4) Unlike wait() method in Java, sleep() method of Thread class doesn't relinquish the lock it has acquired.

5) sleep() method throws Interrupted Exception if another thread interrupt a sleeping thread in java.

6) With sleep() in Java its not guaranteed that when sleeping thread woke up it will definitely get CPU, instead it will go to Runnable state and fight for CPU with other thread.

7) There is a misconception about sleep method in Java that calling t.sleep() will put Thread "t" into sleeping state, that's not true because Thread.sleep method is a static method it always put current thread into Sleeping state and not thread "t".

Why wait (), notify () and notifyAll () must be called from synchronized block or method in Java

Most of Java developer knows that wait() ,notify() and notifyAll() method of object class must have to be called inside synchronized method or synchronized block in Java but how many times we thought why ? Recently this questions was asked to in Java interview to one of my friend, he pondered for a moment and replied that if we don't call wait () or notify () method from synchronized context we will receive `IllegalMonitorStateException` in java. He was right in terms of behavior of language but as per him interviewer was not completely satisfied with the answer and wanted to explain more about it. After the interview he discussed the same questions with me and I thought he might have told aboutrace condition between wait () and notify () in Java that could exists if we don't call them inside synchronized method or block. Let's see how it could happen:

We use wait () and notify () or notifyAll () method mostly for inter-thread communication. One thread is waiting after checking a condition e.g. In Producer Consumer example Producer Thread is waiting if buffer is full and Consumer thread notify Producer thread after he creates a space in buffer by consuming an element. calling notify() or notifyAll() issues a notification to a single or multiple thread that a condition has changed and once notification thread leaves synchronized block , all the threads which are waiting fight for object lock on which they are waiting and lucky thread returns from wait() method after reacquiring the lock and proceed further. Let's divide this whole operation in steps to see a possibility of *race condition between wait () and notify () method in Java*, we will use Produce Consumer thread example to understand the scenario better:

1. The Producer thread tests the condition (buffer is full or not) and confirms that it must wait (after finding buffer is full).

2. The Consumer thread sets the condition after consuming an element from buffer.

3. The Consumer thread calls the notify () method; this goes unheard since the Producer thread is not yet waiting.

4. The Producer thread calls the wait () method and goes into waiting state.

So due to race condition here we potential lost a notification and if we use buffer or just one element Produce thread will be waiting forever and your program will hang.

Now let's think how does this potential race condition get resolved? This race condition is resolved by using synchronized keyword and locking provided by java. In order to call the wait (), notify () or notifyAll () methods in Java, we must have obtained the lock for the object on which we're calling the method. Since the wait () method in Java also releases the lock prior to waiting and reacquires the lock prior to returning from the wait () method, we must use this lock to ensure that checking the condition (buffer is full or not) and setting the condition (taking element from buffer) is atomic which can be achieved by using synchronized method or block in Java.

I am not sure if this is what interviewer was actually expecting but this what I thought would at least make sense, please correct me If I wrong and let us know if there is any other convincing reason of calling wait(), notify() or notifyAll method in Java.

Just to summarize we call wait (), notify () or notifyAll method in Java from synchronized method or synchronized block in Java to avoid:

1) IllegalMonitorStateException in Java which will occur if we don't call wait (), notify () or notifyAll () method from synchronized context.

2) Any potential race condition between wait and notify method in Java.

How to check if a thread holds lock on a particular object in Java

Think about a scenario where you would have to find at run time that whether a java thread has lock on a particular object e.g. find out whether thread `NewsReader` has lock on `NewsPaper` object or not ?
If this questions came in any core java interview then I would automatically assume that there could be at least two answer one is hard earned raw answer which programmer would like to figure out based on fundamentals and other could be some rarely used API calls which is available in java , by the way this is actually asked to me in an interview of one of the biggest global bank.

2 ways to find if thread holds lock on object in Java

Here I am giving my answer and what I had discovered after interview

1)I thought about IllegalMonitorStateException which wait() and notify() methods throw when they get called from non-synchronized context so I said I would call newspaper.wait() and if this call throws exception it means thread in java is not holding lock, otherwise thread holds lock.

2)Later I discovered that thread is a static method called `holdsLock(Object obj)` which returns true or false based on whether threads holds lock on object passed.

ReentrantLock Example in Java, Difference between synchronized vs ReentrantLock

ReentrantLock in Java is added on java.util.concurrent package in Java 1.5 along with other concurrent utilities like CountDownLatch, Executors and CyclicBarrier. `ReentrantLock` is one of the most useful addition in Java concurrency package and several of concurrent collection classes from `java.util.concurrent` package is written using `ReentrantLock`, including `ConcurrentHashMap`, see How ConcurrentHashMap works in Java for more details. Two key feature of `ReentrantLock`, which provides more control on lock acquisition is trying to get a lock with ability to interrupt, and a timeout on waiting for lock, these are key for writing responsive and scalable systems in Java. In short, `ReentrantLock` extends functionality of synchronized keyword in Java and open path for more controlled locking in Java.

In this Java concurrency tutorial we will learn :

- What is ReentrantLock in Java ?
- Difference between ReentrantLock and synchronized keyword in Java?
- Benefits of using Reentrant lock in Java?
- Drawbacks of using Reentrant lock in concurrent program?
- Code Example of ReentrantLock in Java?

What is ReentrantLock in Java

On class level, ReentrantLock is a concrete implementation of Lock interface provided in Java concurrency package from Java 1.5 onwards. As per Javadoc, `ReentrantLock` is mutual exclusive lock, similar to implicit locking provided by synchronized keyword in Java, with extended feature like fairness, which can be used to provide lock to longest waiting thread. Lock is acquired by `lock()` method and held by Thread until a call to `unlock()` method. Fairness parameter is provided while creating instance of `ReentrantLock` in constructor. ReentrantLock provides same visibility and ordering guarantee, provided by implicitly locking, which means, `unlock()` happens before another thread get `lock()`.

Difference between ReentrantLock and synchronized keyword in Java

Though `ReentrantLock` provides same visibility and orderings guaranteed as implicit lock, acquired by synchronized keyword in Java, it provides more functionality and differ in certain aspect. As stated earlier, main difference between synchronized and ReentrantLock is ability to trying for lock interruptibly, and with timeout. Thread doesn't need to block infinitely, which was the case with synchronized. Let's see few more differences between synchronized and Lock in Java.

1) Another significant difference between `ReentrantLock` and synchronized keyword is fairness. `synchronized` keyword doesn't support fairness. Any thread can acquire lock once released, no preference can be specified, on the other hand you can make `ReentrantLock` fair by specifying `fairness` property, while creating instance of `ReentrantLock`. Fairness property provides lock to longest waiting thread, in case of contention.

2) Second difference between synchronized and Reentrant lock is `tryLock()` method. ReentrantLock provides convenient `tryLock()` method, which acquires lock only if its available or not held by any other thread. This reduce blocking of thread waiting for lock in Java application.

3) One more worth noting difference between `ReentrantLock` and synchronized keyword in Java is, ability to interrupt Thread while waiting for `Lock`. In case of `synchronized` keyword, a thread can be blocked waiting for lock, for an indefinite period of time and there was no way to control that. `ReentrantLock` provides a method called `lockInterruptibly()`, which can be used to interrupt thread when it is waiting for lock. Similarly `tryLock()` with timeout can be used to timeout if lock is not available in certain time period.

4) ReentrantLock also provides convenient method to get List of all threads waiting for lock.

So, you can see, lot of significant differences between synchronized keyword and ReentrantLock in Java. In short, Lock interface adds lot of power and flexibility and allows some control over lock acquisition process, which can be leveraged to write highly scalable systems in Java.

Benefits of ReentrantLock in Java

Most of the benefits derives from the *differences covered between synchronized vs ReentrantLock* in last section. Here is summary of benefits offered by ReentrantLock over synchronized in Java:

1) Ability to lock interruptibly.

2) Ability to timeout while waiting for lock.

3) Power to create fair lock.

4) API to get list of waiting thread for lock.

5) Flexibility to try for lock without blocking.

Disadvantages of ReentrantLock in Java

Major drawback of using ReentrantLock in Java is wrapping method body inside try-finally block, which makes code unreadable and hides business logic. It's really cluttered and I hate it most, though IDE like Eclipse and Netbeans can add those try catch block for you. Another disadvantage is that, now programmer is responsible for acquiring and releasing lock, which is a power but also opens gate for new subtle bugs, when programmer forget to release the lock in finally block.

Lock and ReentrantLock Example in Java

Here is a complete code example of How to use `Lock` interface and `ReentrantLock` in Java. This program locks a method called `getCount()`, which provides unique count to each caller. Here we will see both synchronized and ReentrantLock version of same program. You can see code with synchronized is more readable but it's not as flexible as locking mechanism provided by `Lock` interface.

```
import java.util.concurrent.locks.ReentrantLock;
import java.util.logging.Level;
import java.util.logging.Logger;

/**
 * Java program to show, how to use ReentrantLock in Java.
 * Reentrant lock is an alternative way of locking
 * apart from implicit locking provided by synchronized keyword in Java.
 *
 * @author  Javin Paul
 */
public class ReentrantLockHowto {

    private final ReentrantLock lock = new ReentrantLock();
    private int count = 0;

     //Locking using Lock and ReentrantLock
     public int getCount() {
        lock.lock();
        try {
            System.out.println(Thread.currentThread().getName() + " gets Count:
" + count);
            return count++;
        } finally {
            lock.unlock();
        }
     }

     //Implicit locking using synchronized keyword
     public synchronized int getCountTwo() {
            return count++;
     }
```

```
public static void main(String args[]) {
    final ThreadTest counter = new ThreadTest();
    Thread t1 = new Thread() {

        @Override
        public void run() {
            while (counter.getCount() &lt; 6) {
                try {
                    Thread.sleep(100);
                } catch (InterruptedException ex) {
                    ex.printStackTrace();                    }
            }
        }
    };

    Thread t2 = new Thread() {

        @Override
        public void run() {
            while (counter.getCount() &lt; 6) {
                try {
                    Thread.sleep(100);
                } catch (InterruptedException ex) {
                    ex.printStackTrace();
                }
            }
        }
    };
```

```
        t1.start();

        t2.start();


    }
}


Output:
Thread-0 gets Count: 0
Thread-1 gets Count: 1
Thread-1 gets Count: 2
Thread-0 gets Count: 3
Thread-1 gets Count: 4
Thread-0 gets Count: 5
Thread-0 gets Count: 6
Thread-1 gets Count: 7
```

That's all on What is ReentrantLock in Java, How to use with simple example, and difference between `ReentrantLock` and synchronized keyword in Java. We have also seen significant enhancement provided by `Lock` interface over synchronized e.g. trying for lock, timeout while waiting for lock and ability to interrupt thread while waiting for lock. Just be careful to release lock in finally block.

What are difference between wait and sleep in Java

Wait vs sleep in Java

Differences between wait and sleep method in Java Thread is one of the very old question asked in Java interviews. Though both wait and sleep puts thread on waiting state, they are completely different in terms of behaviour and use cases. Sleep is meant for introducing pause, releasing CPU and giving another thread opportunity to execute while wait is used for inter thread communication, by using `wait()` and `notify()` method two threads can communicate with each other which is key to solve many Concurrent problems like Produce consumer issue, Dining philosopher issue and to implement several Concurrency designs. In this tutorial we will see

- What is wait method in Java
- What is Sleep method in Java
- Difference between wait and sleep in Java
- Where to use wait and sleep in Java

What is wait and sleep method in Java

Wait method is defined in Object class and it available to all object,`wait()` method is always discussed along with its counterpart`notify()` and `notifyAll()` method and used in inter thread communication in Java. wait method puts a thread on wait by checking some condition like in Producer Consumer problem, producer thread should wait if Queue is full or Consumer thread should wait if Queue is empty. `notify()` method is used to wake up waiting thread by communicating that waiting condition is over now for example once producer thread puts an item on empty queue it can notify Consumer thread that Queue is not empty any more. On the other hand `Sleep()` method is used to introduce pause on Java application. You can put a Thread on sleep, where it does not do anything and relinquish the CPU for specified duration. When a Thread goes to Sleep it can be either wake up normally after sleep duration elapsed or it can be woken up abnormally by interrupting it.

Difference between Wait and Sleep method in Java Thread

In last section we saw *what is wait and sleep method* and in this section we will see what are differences between wait and sleep method in Java. As I told before apart from waiting they are completely different to each other:

1) First and most important difference between Wait and sleep method is that wait method must be called from synchronized context i.e. from synchronized method or block in Java. If you call wait method without synchronization, it will throw`IllegalMonitorStateException` in Java. On the other hand there is no requirement of synchronization for calling sleep method , you can call it normally.

2) Second worth noting difference between wait and sleep method is that, wait operates on `Object` and defined in `Object` class while sleep operates on current Thread and defined in java.lang.Thread class.

3) Third and another significant difference between wait and sleep in Java is that, `wait()` method releases the lock of object on which it has called, it does release other locks if it holds any while sleep method of Thread class does not release any lock at all.

4) wait method needs to be called from a loop in order to deal with false alarm i.e. waking even though waiting condition still holds true, while there is no such thing for sleep method in Java. its better not to call Sleep method from loop.

here is code snippet for calling wait and sleep method in Java

```
synchronized(monitor)
while(condition == true){ monitor.wait())  //releases monitor lock


Thread.sleep(100); //puts current thread on Sleep
```

5) One more difference between wait and sleep method which is not as significant as previous ones is that `wait()` is a non static method while `sleep()` is static method in Java.

Where to use wait and sleep method in Java

By reading properties and behavior of wait and sleep method it's clear that `wait()` method should be used in conjunction with`notify()` or `notifyAll()` method and intended for communication between two threads in Java while `Thread.sleep()` method is a utility method to introduce short pauses during program or thread execution. Given the requirement of synchronization for wait, it should not be used just to introduce pause or sleep in Java.

Java program to solve Producer Consumer Problem in Java

Here is complete Java program to solve producer consumer problem in Java programming language. In this program we have used wait and notify method from java.lang.Object class instead of using BlockingQueue for flow control.

```
import java.util.Vector;
import java.util.logging.Level;
import java.util.logging.Logger;

/**
 * Java program to solve Producer Consumer problem using wait and notify
 * method in Java. Producer Consumer is also a popular concurrency design pattern.
 *
 * @author Javin Paul
 */
public class ProducerConsumerSolution {

    public static void main(String args[]) {
        Vector sharedQueue = new Vector();
        int size = 4;
        Thread prodThread = new Thread(new Producer(sharedQueue,
size), "Producer");
        Thread consThread = new Thread(new Consumer(sharedQueue,
size), "Consumer");
        prodThread.start();
        consThread.start();
    }
```

```
}

class Producer implements Runnable {

    private final Vector sharedQueue;
    private final int SIZE;

    public Producer(Vector sharedQueue, int size) {
        this.sharedQueue = sharedQueue;
        this.SIZE = size;
    }

    @Override
    public void run() {
        for (int i = 0; i < 7; i++) {
            System.out.println("Produced: " + i);
            try {
                produce(i);
            } catch (InterruptedException ex) {
                Logger.getLogger(Producer.class.getName()).log(Level.SEVERE, null,
ex);
            }

        }
    }

    private void produce(int i) throws InterruptedException {

        //wait if queue is full
        while (sharedQueue.size() == SIZE) {
            synchronized (sharedQueue) {
                System.out.println("Queue is full
" + Thread.currentThread().getName()
                                    + " is waiting , size: " + sharedQueue.size());

                sharedQueue.wait();
            }
        }
```

```
        //producing element and notify consumers
        synchronized (sharedQueue) {
            sharedQueue.add(i);
            sharedQueue.notifyAll();
        }
    }
}

class Consumer implements Runnable {

    private final Vector sharedQueue;
    private final int SIZE;

    public Consumer(Vector sharedQueue, int size) {
        this.sharedQueue = sharedQueue;
        this.SIZE = size;
    }

    @Override
    public void run() {
        while (true) {
            try {
                System.out.println("Consumed: " + consume());
                Thread.sleep(50);
            } catch (InterruptedException ex) {
                Logger.getLogger(Consumer.class.getName()).log(Level.SEVERE, null,
ex);
            }

        }
    }

    private int consume() throws InterruptedException {
        //wait if queue is empty
        while (sharedQueue.isEmpty()) {
            synchronized (sharedQueue) {
```

```
                System.out.println("Queue is empty
" + Thread.currentThread().getName()
                                    + " is waiting , size: " + sharedQueue.size());

                sharedQueue.wait();
            }
        }

        //Otherwise consume element and notify waiting producer
        synchronized (sharedQueue) {
            sharedQueue.notifyAll();
            return (Integer) sharedQueue.remove(0);
        }
    }
}

Output:
Produced: 0
Queue is empty Consumer is waiting , size: 0
Produced: 1
Consumed: 0
Produced: 2
Produced: 3
Produced: 4
Produced: 5
Queue is full Producer is waiting , size: 4
Consumed: 1
Produced: 6
Queue is full Producer is waiting , size: 4
Consumed: 2
Consumed: 3
Consumed: 4
Consumed: 5
Consumed: 6
Queue is empty Consumer is waiting , size: 0
```

That's all on How to solve producer consumer problem in Java using wait and notify method. I still think that using`BlockingQueue` to implement producer consumer design pattern is much better because of its simplicity

and concise code. At the same time this problem is an excellent exercise to understand concept of wait and notify method in Java.

Difference between Callable and Runnable in Java - Interview question

Difference between `Callable` and `Runnable` interface in Java is one of the interesting question from my list of Top 15 Java multi-threading questions, and it's also very popular in various Java Interviews. `Callable` interface is newer than `Runnable` interface and added on Java 5 release along with other major changes e.g. Generics, Enum, Static imports and variable argument method. Though both `Callable` and `Runnable` interface are designed to represent task, which can be executed by any thread, there is some significant difference between them. In my opinion, *major difference between Callable and Runnable interface* is that `Callable` can return result of operationperformed inside `call()` method, which was one of the limitation with `Runnable` interface. Another significant difference between `Runnable` and `Callable` interface is ability to throw checked exception. `Callable` interface can throw checked exception because it's call method throws Exception. By the way sometime this question is also asked as follow-up question of another classic difference between Runnable and Thread in Java. Commonly`FutureTask` is used along with `Callable` to get result of asynchronous computation task performed in `call()` method.

Callable vs Runnable interface

As I explained major differences between `Callable` and`Runnable` interface in last section. Sometime this question is also asked as difference between `call()` and `run()` method in Java. All the points discussed here is equally related to that question as well. Let's see them in point format for better understanding :

1) `Runnable` interface is older than `Callable`, there from JDK 1.0, while Callable is added on Java 5.0.

2) `Runnable` interface has `run()` method to define task while `Callable` interface uses `call()` method for task definition.

3) `run()` method does not return any value, it's return type is void while call method returns value. Callable interface is a generic parameterized interface and Type of value is provided, when instance of Callable implementation is created.

4) Another difference on run and call method is that run method can not throw checked exception, while call method can throwchecked exception in Java.

That's all on Difference between Callable and Runnable interface in Java or difference between `call()` and `run()` method. Both are very useful interface from core Java and good understanding of where to use `Runnable` and `Callable` is must for any good Java developer. In next article we will see example of `Callable` interface along with `FutureTask` to learn How to use`Callable` interface in Java

Difference between Callable and Runnable in Java - Interview question

Difference between `Callable` and `Runnable` interface in Java is one of the interesting question from my list of Top 15 Java multi-threading questions, and it's also very popular in various Java Interviews. `Callable` interface is newer than `Runnable` interface and added on Java 5 release along with other major changes e.g. Generics, Enum, Static imports and variable argument method. Though both `Callable` and `Runnable` interface are designed to represent task, which can be executed by any thread, there is some significant difference between them. In my opinion, *major difference between Callable and Runnable interface* is that `Callable` can return result of operationperformed inside `call()` method, which was one of the limitation with `Runnable` interface. Another significant difference between `Runnable` and `Callable` interface is ability to throw checked exception. `Callable` interface can throw checked exception because it's call method throws Exception. By the way sometime this question is also asked as follow-up question of another classic difference between Runnable and Thread in Java. Commonly`FutureTask` is used along with `Callable` to get result of asynchronous computation task performed in `call()` method.

Callable vs Runnable interface

As I explained major differences between `Callable` and`Runnable` interface in last section. Sometime this question is also asked as difference between `call()` and `run()` method in Java. All the points discussed here is equally related to that question as well. Let's see them in point format for better understanding :

1) `Runnable` interface is older than `Callable`, there from JDK 1.0, while Callable is added on Java 5.0.

2) `Runnable` interface has `run()` method to define task while `Callable` interface uses `call()` method for task definition.

3) `run()` method does not return any value, it's return type is void while call method returns value. Callable interface is a generic parameterized interface and Type of value is provided, when instance of Callable implementation is created.

4) Another difference on run and call method is that run method can not throw checked exception, while call method can throwchecked exception in Java.

That's all on Difference between Callable and Runnable interface in Java or difference between `call()` and `run()` method. Both are very useful interface from core Java and good understanding of where to use `Runnable` and `Callable` is must for any good Java developer. In next article we will see example of `Callable` interface along with `FutureTask` to learn How to use`Callable` interface in Java

Difference between start and run method in Thread - Java Tutorial

Why do one call `start` method of thread if `start()` calls `run()` in turn" or "What is difference by calling `start`() over `run()` method in java thread" are two widely popular beginner level multi-threading interview question. When a Java programmer `start` learning Thread, first thing he learns is to implement thread either overriding `run()` method of Thread class or implementing Runnable interface and than calling `start()` method on thread, but with some experience he finds that `start`() method calls `run()` method internally either by looking API documentation or just poking around, but many of us just don't care at that time until its been asked in Java Interview. In this Java tutorial we will see What is difference between calling `start()` method and `run()` method for starting Thread in Java.

This article is in continuation of my earlier post on Java multi-threading e.g. Difference between Runnable and Thread in Java and How to solve Producer Consumer problem in Java using `BlockingQueue`. If you haven't read them already you may find them interesting and useful.

Difference between `start` and `run` in Java Thread

So what is difference between `start` and `run method`? Main difference is that when program calls `start`() method a new Thread is created and code inside `run()` method is executed in new Thread while if you call `run()` method directly no new Thread is created and code inside `run()` will execute on current Thread. Most of the time calling `run()` is bug or programming mistake because caller has intention of calling `start`() to create new thread and this error can be detect by many static code coverage tools like findbugs. If you want to perform time consuming task than always call `start`() method otherwise your main thread will stuck while performing time consuming task if you call `run()` method directly. Another difference between start vs run in Java thread is that you can not call `start()` method twice on thread object. once started, second call of start() will throw `IllegalStateException` in Java while you can call `run()` method twice.

Code Example of `start` vs `run` method

Here is a simple code example which prints name of Thread which executes `run()` method of Runnable task. Its clear that if you call `start()` method a new Thread executes Runnable task while if you directly call `run()` method task, current thread which is main in this case will execute the task.

```
public class StartVsRunCall{

    public static void main(String args[]) {

        //creating two threads for start and run method call
        Thread startThread = new Thread(new Task("start"));
        Thread runThread = new Thread(new Task("run"));

        startThread.start(); //calling start method of Thread - will execute in new
Thread
        runThread.run();  //calling run method of Thread - will execute in current
Thread

    }


    /*
     * Simple Runnable implementation
     */
    private static class Task implements Runnable{
        private String caller;

        public Task(String caller){
            this.caller = caller;
        }

        @Override
        public void run() {
            System.out.println("Caller: "+ caller + " and code on this Thread is
executed by : " + Thread.currentThread().getName());

        }
    }
}



Output:

Caller: start and code on this Thread is executed by : Thread-0
Caller: run and code on this Thread is executed by : main
```

In Summary only difference between `start`() and `run()` method in Thread is that `start` creates new thread while run doesn't create any thread and simply execute in current thread like a normal method call.

How CountDownLatch works in Java

Now we know What is `CountDownLatch` in Java, its time to find out How `CountDownLatch` works in Java. CountDownLatch works in latch principle, main thread will wait until Gate is open. One thread waits for n number of threads specified while creating `CountDownLatch` in Java. Any thread, usually main thread of application, which calls `CountDownLatch.await()` will wait until count reaches zero or its `interrupted` by another Thread. All other thread are required to do count down by calling `CountDownLatch.countDown()` once they are completed or ready to the job. as soon as count reaches zero, Thread awaiting starts running. One of the disadvantage of `CountDownLatch` is that its not reusable once count reaches to zero you can not use `CountDownLatch` any more, but don't worry Java concurrency API has another concurrent utility called `CyclicBarrier` for such requirements.

CountDownLatch Exmaple in Java

In this section we will see a full featured real world example of using *CountDownLatch in Java*. In following CountDownLatch example, Java program requires 3 services namely `CacheService`, `AlertService` and `ValidationService` to be started and ready before application can handle any request and this is achieved by using `CountDownLatch` in Java.

```
import java.util.Date;
import java.util.concurrent.CountDownLatch;
import java.util.logging.Level;
import java.util.logging.Logger;

/**
 * Java program to demonstrate How to use CountDownLatch in Java. CountDownLatch is

 * useful if you want to start main processing thread once its dependency is
completed

 * as illustrated in this CountDownLatch Example
 *
 * @author Javin Paul
 */
public class CountDownLatchDemo {

    public static void main(String args[]) {
       final CountDownLatch latch = new CountDownLatch(3);
       Thread cacheService = new Thread(new Service("CacheService", 1000, latch));
       Thread alertService = new Thread(new Service("AlertService", 1000, latch));
       Thread validationService = new Thread(new Service("ValidationService", 1000,
latch));

       cacheService.start(); //separate thread will initialize CacheService
       alertService.start(); //another thread for AlertService initialization
       validationService.start();

       // application should not start processing any thread until all service is
up

       // and ready to do there job.
       // Countdown latch is idle choice here, main thread will start with count 3

       // and wait until count reaches zero. each thread once up and read will do

       // a count down. this will ensure that main thread is not started processing

       // until all services is up.

       //count is 3 since we have 3 Threads (Services)

       try{
            latch.await();  //main thread is waiting on CountDownLatch to finish
            System.out.println("All services are up, Application is starting now");
       }catch(InterruptedException ie){
           ie.printStackTrace();
       }

    }

}
```

```
/**
 * Service class which will be executed by Thread using CountDownLatch
synchronizer.
 */
class Service implements Runnable{
    private final String name;
    private final int timeToStart;
    private final CountDownLatch latch;

    public Service(String name, int timeToStart, CountDownLatch latch){
        this.name = name;
        this.timeToStart = timeToStart;
        this.latch = latch;
    }

    @Override
    public void run() {
        try {
            Thread.sleep(timeToStart);
        } catch (InterruptedException ex) {
            Logger.getLogger(Service.class.getName()).log(Level.SEVERE, null, ex);
        }
        System.out.println( name + " is Up");
        latch.countDown(); //reduce count of CountDownLatch by 1
    }

}

Output:
ValidationService is Up
AlertService is Up
CacheService is Up
All services are up, Application is starting now
```

By looking at output of this CountDownLatch example in Java, you can see that Application is not started until all services started by individual Threads are completed.

When should we use `CountDownLatch` in Java :

Use `CountDownLatch` when one of Thread like main thread, require to wait for one or more thread to complete, before its start doing processing. Classical example of using `CountDownLatch` in Java is any server side core Java application which uses services architecture, where multiple services is provided by multiple threads and application can not start processing until all services have started successfully as shown in our `CountDownLatch` example.

CountDownLatch in Java – Things to remember

Few points about Java CountDownLatch which is worth remembering:

1) You can not reuse `CountDownLatch` once count is reaches to zero, this is the main difference between CountDownLatch and CyclicBarrier, which is frequently asked in core Java interviews and multi-threading interviews.

2) Main Thread wait on Latch by calling `CountDownLatch.await()` method while other thread calls `CountDownLatch.countDown()` to inform that they have completed.

What is CyclicBarrier Example in Java 5 - Concurrency Tutorial

What is `CyclicBarrier` in Java

CyclicBarrier in Java is a synchronizer introduced in JDK 5 on java.util.Concurrent package along with other concurrent utility like Counting Semaphore, BlockingQueue, ConcurrentHashMap etc. CyclicBarrier is similar to CountDownLatch which we have seen in last article What is CountDownLatch in Java and allows multiple threads to wait for each other (barrier) before proceeding. Difference between CountDownLatch and CyclicBarrier is a also very popular multi-threading interview question in Java. CyclicBarrier is a natural requirement for concurrent program because it can be used to perform final part of task once individual tasks are completed. All threads which wait for each other to reach barrier are called parties, CyclicBarrier is initialized with number of parties to be wait and threads wait for each other by calling CyclicBarrier.await() method which is a blocking method in Java and blocks until all Thread or parties call await(). In general calling await() is shout out that Thread is waiting on barrier. await() is a blocking call but can be timed out or Interrupted by other thread. In this Java concurrency tutorial we will see *What is CyclicBarrier in Java* and an example of CyclicBarrier on which three Threads will wait for each other before proceeding further.

Difference between CountDownLatch and CyclicBarrier in Java

In our last article we have see how CountDownLatch can be used to implement multiple threads waiting for each other. If you look at CyclicBarrier it also the does the same thing but there is a different you can not reuse CountDownLatch once count reaches zero while you can reuse CyclicBarrier by calling reset() method which resets Barrier to its initial State. What it implies that CountDownLatch is good for one time event like application start-up time and CyclicBarrier can be used to in case of recurrent event e.g. concurrently calculating solution of big problem etc. If you like to learn more about threading and concurrency in Java you can also check my post on When to use Volatile variable in Java and How Synchronization works in Java.

CyclicBarrier in Java - Example

Now we know what is CyclicBarrier in Java and it's time to see example of CyclicBarrier in Java. Here is a simple example of CyclicBarrier in Java on which we initialize CyclicBarrier with 3 parties, means in order to cross barrier, 3 thread needs to call await() method. each thread calls await method in short duration but they don't proceed until all 3 threads reached barrier, once all thread reach barrier, barrier gets broker and each thread started there execution from that point. Its much clear with the output of following example of CyclicBarrier in Java:

```
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;
import java.util.logging.Level;
import java.util.logging.Logger;

/**
 * Java program to demonstrate how to use CyclicBarrier in Java. CyclicBarrier is a

 * new Concurrency Utility added in Java 5 Concurrent package.

 *
 * @author Javin Paul
 */
public class CyclicBarrierExample {

    //Runnable task for each thread
    private static class Task implements Runnable {

        private CyclicBarrier barrier;

        public Task(CyclicBarrier barrier) {
            this.barrier = barrier;
        }
```

```
        @Override
        public void run() {
            try {
                System.out.println(Thread.currentThread().getName() + " is waiting
on barrier");
                barrier.await();
                System.out.println(Thread.currentThread().getName() + " has crossed
the barrier");
            } catch (InterruptedException ex) {
                Logger.getLogger(CyclicBarrierExample.class.getName()).log(Level.SE
VERE, null, ex);
            } catch (BrokenBarrierException ex) {
                Logger.getLogger(CyclicBarrierExample.class.getName()).log(Level.SE
VERE, null, ex);
            }
        }
    }

    public static void main(String args[]) {

        //creating CyclicBarrier with 3 parties i.e. 3 Threads needs to call
await()
        final CyclicBarrier cb = new CyclicBarrier(3, new Runnable(){
            @Override
            public void run(){
                //This task will be executed once all thread reaches barrier
                System.out.println("All parties are arrived at barrier, lets
play");
            }
        });

        //starting each of thread
        Thread t1 = new Thread(new Task(cb), "Thread 1");
        Thread t2 = new Thread(new Task(cb), "Thread 2");
        Thread t3 = new Thread(new Task(cb), "Thread 3");

        t1.start();
        t2.start();
        t3.start();

    }
}

Output:
Thread 1 is waiting on barrier
Thread 3 is waiting on barrier
Thread 2 is waiting on barrier
All parties are arrived at barrier, lets play
Thread 3 has crossed the barrier
Thread 1 has crossed the barrier
Thread 2 has crossed the barrier
```

When to use `CyclicBarrier` in Java

Given the nature of `CyclicBarrier` it can be very handy to implement map reduce kind of task similar to fork-join framework of Java 7, where a big task is broker down into smaller pieces and to complete the task you need output from individual small task e.g. to count population of India you can have 4 threads which counts population from North, South, East and West and once complete they can wait for each other, When last thread completed there task, Main thread or any other thread can add result from each zone and print total population. You can use `CyclicBarrier` in Java :

1) To implement multi player game which can not begin until all player has joined.

2) Perform lengthy calculation by breaking it into smaller individual tasks, In general to implement Map reduce technique.

Important point of `CyclicBarrier` in Java

1. `CyclicBarrier` can perform a completion task once all thread reaches to barrier, This can be provided while creating `CyclicBarrier`.

2. If `CyclicBarrier` is initialized with 3 parties means 3 thread needs to call await method to break the barrier.

3. Thread will block on `await()` until all parties reaches to barrier, another thread interrupt or `await` timed out.

4. If another thread interrupt the thread which is waiting on barrier it will throw `BrokernBarrierException` as shown below:

```
java.util.concurrent.BrokenBarrierException
        at java.util.concurrent.CyclicBarrier.dowait(CyclicBarrier.java:172)
        at java.util.concurrent.CyclicBarrier.await(CyclicBarrier.java:327)
```

5.`CyclicBarrier`.reset() put `Barrier` on its initial state, other thread which is waiting or not yet reached barrier will terminate with `java.util.concurrent.BrokenBarrierException`.

That's all on What is `CyclicBarrier` in Java , When to use CyclicBarrier in Java and a Simple Example of How to use CyclicBarrier in Java . We have also seen difference between `CountDownLatch` and `CyclicBarrier` in Java and got some idea where we can use `CyclicBarrier` in Java Concurrent code.