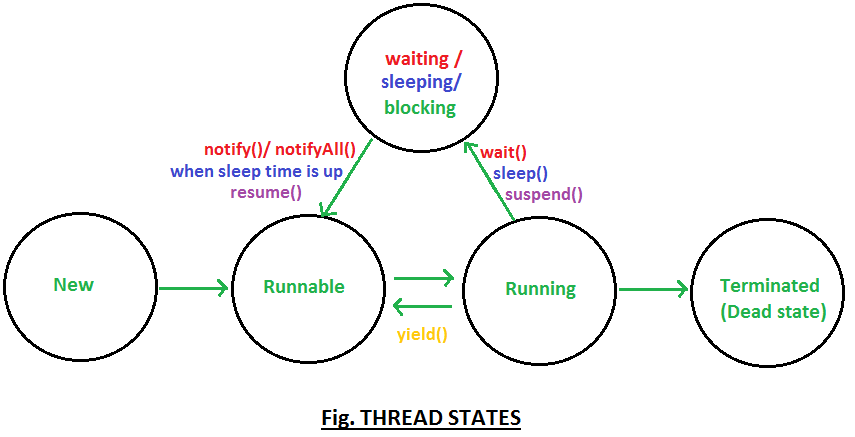
**Thread Life Cycle.**



**What is Thread in Java?**

A thread, in the context of Java, is the path followed when executing a program. All Java programs have at least one thread, known as the main thread, which is created by the JVM at the program’s start, when the main() method is invoked with the main thread. In Java, creating a thread is accomplished by implementing an interface and extending a class. Every Java thread is created and controlled by the java.lang.Thread class.

By employing multiple threads you can speed up CPU bound task. For example, if one thread takes 100 milliseconds to do a job, you can use 10 thread to reduce that task into 10 milliseconds. Java provides excellent support for multithreading at the language level, and it's also one of the strong selling points.

**Difference between Process and Thread**

1. Process is heavy weight or resource intensive. Thread is light weight, taking lesser resources than a process.

2. Process switching needs interaction with operating system. Thread switching does not need to interact with operating system.

3. In multiple processing environments, each process executes the same code but has its own memory and file resources. All threads can share same set of open files, child processes.

4. If one process is blocked, then no other process can execute until the first process is unblocked. While one thread is blocked and waiting, a second thread in the same task can run.

5. Multiple processes without using threads use more resources. Multiple threaded processes use fewer resources.

6. In multiple processes each process operates independently of the others. One thread can read, write or change another thread's data.

**What are the benefits of multi-threaded programming?**

Better resource utilization.

Simpler program design in some situations.

More responsive programs.

**Difference between "Implements Runnable" And "Extends Thread".**

|  |  |  |
| --- | --- | --- |
|  | **Runnable** | **Thread** |
| Inheritance option | extends any java class | No |
| Reusability | Yes | No |
| Object Oriented Design | Good, allows composition | Bad |
| Loosely Coupled | Yes | No |
| Function Overhead | No | Yes |

**Difference between sleep() and wait().**

|  |  |  |
| --- | --- | --- |
|  | **sleep()** | **wait()** |
| Class belongs | java.lang.Thread | java.lang.Object |
| Context | Called from any context | Only synchronized context |
| Locking | Does not release the lock for specified time or until interrupt. | Releases the lock |
| Wake up Condition | When time expires or due to interruption | Awake by call to notify() or notifyAll() method |
| Execution | Execution of sleep will pause the current running thread not the object on which it is called. | Thread wait() continues till a specific condition holds true |
| **Usage** | for time-synchronization | for multi-thread-synchronization |

**What is difference between user Thread and daemon Thread?**

JVM waits for user threads to finish their work. It will not exit until all user threads finish their work.

JVM will not wait for daemon threads to finish their work. It will exit as soon as all user threads finish their work.

**User** threads are foreground threads.

Daemon threads are background threads.

**User** threads are high priority threads.

Daemon threads are low priority threads.

**User** threads are created by the application.

Daemon threads, in most of time, are created by the JVM.

**User** threads are mainly designed to do some specific task.

Daemon threads are designed to support the user threads.

JVM will not force the user threads to terminate. It will wait for user threads to terminate themselves.

JVM will force the daemon threads to terminate if all user threads have finished their work.

**How can we create a Thread in Java?**

There are two ways of implementing threading in Java  
1) By extending java.lang.Thread class, or  
2) By implementing java.lang.Runnable interface.

**What is the difference between start() and run() method of Thread class?**

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](http://javarevisited.blogspot.com/2012/02/java-mistake-1-using-float-and-double.html) because caller has intention of calling start() to create new thread and this error can be detect by many static code coverage tools like find bugs. If you want to perform time consuming task than always call start() method otherwise your [main thread](http://javarevisited.blogspot.com/2011/12/main-public-static-java-void-method-why.html) 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 **cannot call start() method twice**on thread object. Once started, second call of start() will throwIllegalStateException in Java while you can call run() method twice.

**What does the join() method do ?**

The join () method waits for a thread to die. In other words, it causes the currently running threads to stop executing until the thread it joins with completes its task.

**What is volatile variable in Java?**

volatile is a special modifier, which can only be used with instance variables. In concurrent Java programs, changes made by multiple threads on instance variables is not visible to other in absence of any synchronizers e.g. synchronized keyword or locks. Volatile variable guarantees that a write will happen before any subsequent read: as stated: "volatile variable rule" in previous question. Read this answer to learn more about volatile variable and when to use them.

**Why Thread sleep() and yield() methods are static?**

Thread sleep() and yield() methods work on the currently executing thread. So there is no point in invoking these methods on some other threads that are in wait state. That’s why these methods are made static so that when this method is called statically, it works on the current executing thread and avoid confusion to the programmers who might think that they can invoke these methods on some non-running threads.

**What is the difference between start and run method in Java Thread?**

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 find bugs. 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 cannot 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.

**Why wait and notify method are called from synchronized block?**

A wait() only makes sense when there is also a notify(), so it's always about communication between threads, and that needs synchronization to work correctly. One could argue that this should be implicit, but that would not really help, for the following reason:

Semantically, you never just wait(). You need some condition to be satisfied, and if it is not, you wait until it is. So what you really do is

if(!condition){

wait();

}

But the condition is being set by a separate thread, so in order to have this work correctly you need synchronization.

A couple more things wrong with it, where just because your thread quit waiting doesn't mean the condition you are looking for is true:

You can get spurious wakeups (meaning that a thread can wake up from waiting without ever having received a notification), or

The condition can get set, but a third thread makes the condition false again by the time the waiting thread wakes up (and reacquires the monitor).

To deal with these cases what you really need is always some variation of this:

synchronized(lock){

while(!condition){

lock.wait();

}

}

Better yet, don't mess with the synchronization primitives at all and work with the abstractions offered in the java.util.concurrent packages.

**What is thread-safety?**

The concept of thread safe is simply that the program state (fields/objects/variables) behaves correctly when multiple simultaneous threads are using a resource. The term can be applied to a section of code, a method, a class, a library, an application, etc. The basic meaning remains the same across these scopes, but what is required to be thread safe can vary widely.

It's perhaps more useful to look at what it means to not be thread safe. Code is not thread safe when it performs an operation that relies on the underlying state not being changed by another thread without guaranteeing that the state cannot be changed by another thread.

The simplest means of making something thread safe is to only access the state from a single thread. Since the stack, and thus all local variables, are within the scope of a single thread, code that doesn't access object fields (either directly or indirectly) and only accesses local variables, is inherently thread safe. Additionally, if no other thread has a reference to an object, then the state of any instance (non-static) fields in the object are inherently thread safe. This condition is true for the majority of code, which is why most code doesn't need to take any extra measures in order to be thread safe.

**Difference between Callable and Runnable in Java.**

1) The 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. The Callable interface is a [generic parameterized interface](http://javarevisited.blogspot.sg/2012/08/how-to-write-parametrized-class-method-Generic-example.html) and Type of value is provided when an instance of Callable implementation is created.  
4) Another difference on run and call method is that run method cannot [throw](http://java67.blogspot.sg/2012/10/difference-between-throw-vs-throws-in.html) checked exception while call method can throw checked exception in Java.

**How to stop a thread in Java?**

I always said that Java provides rich APIs for everything but ironically Java doesn't provide a sure shot way of stopping thread. There was some control methods in JDK 1.0 e.g. stop(), suspend() and resume() which was deprecated in later releases due to potential deadlock threats, from then Java API designers has not made any effort to provide a consistent, thread-safe and elegant way to stop threads. Programmers mainly rely on the fact that thread stops automatically as soon as they finish execution of run() or call() method. To manually stop, programmers either take advantage of volatile boolean variable and check in every iteration if run method has loops or interrupt threads to abruptly cancel tasks. See this tutorial for sample code of stopping thread in Java.

**What is the difference between notify and notifyAll in Java?**

1. First and main difference between notify() and notifyAll() method is that, if multiple threads is waiting on any lock in Java, notify method send notification to only one of waiting thread while notifyAll informs all threads waiting on that lock.

2. If you use notify method, It's not guaranteed that, which thread will be informed, but if you use notifyAll since all thread will be notified, they will compete for lock and the lucky thread which gets lock will continue. In a way, notifyAll method is safer because it sends notification to all threads, so if any thread misses the notification, there are other threads to do the job, while in the case of notify() method if the notified thread misses the notification then it could create subtle, hard to debug issues.

3. Some people argue that using notifyAll can drain more CPU cycles than notify itself but if you really want to sure that your notification doesn't get wasted by any reason, use notifyAll. Since wait method is called from the loop and they check condition even after waking up, calling notifyAll won't lead any side effect, instead it ensures that notification is not dropped.

**Why wait, notify and notifyAll are not inside thread class?**

|  |
| --- |
| I am just having hard time to understand concept behind putting wait() in object class For this questions sake consider as if wait() and notifyAll() are in thread class  In the Java language, you wait() on a particular instance of an Object -- a monitor assigned to that object to be precise. If you want to send a signal to one thread that is waiting on that specific object instance then you call notify() on that object. If you want to send a signal to all threads that are waiting on that object instance, you use notifyAll() on that object.  If wait() and notify() were on the Thread instead then each thread would have to know the status of every other thread. How would thread1 know that thread2 was waiting for access to a particular resource? If thread1 needed to call thread2.notify() it would have to somehow find out that thread2 was waiting. There would need to be some mechanism for threads to register the resources or actions that they need so others could signal them when stuff was ready or available.  In Java, the object itself is the entity that is shared between threads which allows them to communicate with each other. The threads have no specific knowledge of each other and they can run asynchronously. They run and they lock, wait, and notify on the object that they want to get access to. They have no knowledge of other threads and don't need to know their status. They don't need to know that it is thread2 which is waiting for the resource -- they just notify on the resource and whomever it is that is waiting (if anyone) will be notified.  In Java, we then use lock objects as synchronization, mutex, and communication points between threads. We synchronize on a lock object to get mutex access to an important code block and to synchronize memory. We wait on a lock object if we are waiting for some condition to change -- some resource to become available. We notify on an object if we want to awaken sleeping threads. |

**What is ThreadLocal variable in Java?**

ThreadLocal variables are special kind of variable available to Java programmer. Just like instance variable is per instance, ThreadLocal variable is per thread. It's a nice way to achieve thread-safety of expensive-to-create objects, for example you can make SimpleDateFormat thread-safe using ThreadLocal. Since that class is expensive, its not good to use it in local scope, which requires separate instance on each invocation. By providing each thread their own copy, you shoot two birds with one arrow. First, you reduce number of instance of expensive object by reusing fixed number of instances, and Second, you achieve thread-safety without paying cost of synchronization or immutability. Another good example of thread local variable is ThreadLocalRandom class, which reduces number of instances of expensive-to-create Random object in multi-threading environment. See this answer to learn more about thread local variables in Java.

**What is thread pool? Why should you thread pool in Java?**

Creating thread is expensive in terms of time and resource. If you create thread at time of request processing it will slow down your response time, also there is only a limited number of threads a process can create. To avoid both of these issues, a pool of thread is created when application starts-up and threads are reused for request processing. This pool of thread is known as "thread pool" and threads are known as worker thread. From JDK 1.5 release, Java API provides Executor framework, which allows you to create different types of thread pools e.g. single thread pool, which process one task at a time, fixed thread pool (a pool of fixed number of threads) or cached thread pool (an expandable thread pool suitable for applications with many short lived tasks). See this article to learn more about thread pools in Java to prepare detailed answer of this question.

**How do you avoid deadlock in Java? Write Code?**

Deadlock is a condition in which two threads wait for each other to take action which allows them to move further. It's a serious issue because when it happen your program hangs and doesn't do the task it is intended for. In order for deadlock to happen, following four conditions must be true:

**Mutual Exclusion** : At least one resource must be held in a non-shareable mode. Only one process can use the resource at any given instant of time.

**Hold and Wait**: A process is currently holding, at least, one resource and requesting additional resources which are being held by other processes.

**No Pre-emption**: The operating system must not de-allocate resources once they have been allocated; they must be released by the holding process voluntarily.

**Circular Wait**: A process must be waiting for a resource which is being held by another process, which in turn is waiting for the first process to release the resource.

The easiest way to avoid deadlock is to prevent Circular wait, and this can be done by acquiring locks in a particular order and releasing them in reverse order so that a thread can only proceed to acquire a lock if it held the other one.

**How do you check if a Thread holds a lock or not? (answer)**

I didn't even know that you can check if a Thread already holds lock before this question hits me in a telephonic round of Java interview. There is a method called holdsLock() on java.lang.Thread, it returns true if and only if the current thread holds the monitor lock on the specified object. You can also check this article for the more detailed answer.

**What does yield method of Thread class do?**

Yield method is one way to request current thread to relinquish CPU so that other thread can get a chance to execute. Yield is a static method and only guarantees that current thread will relinquish the CPU but doesn't say anything about which other thread will get CPU. Its possible for the same thread to get CPU back and start its execution again. See this article to learn more about yield method and to answer this question better.

**What happens if a thread throws an Exception inside synchronized block?**

This is one more tricky question for average Java programmer, if he can bring the fact about whether lock is released or not is a key indicator of his understanding. To answer this question, no matter how you exist synchronized block, either normally by finishing execution or abruptly by throwing exception, thread releases the lock it acquired while entering that synchronized block. This is actually one of the reasons I like synchronized block over lock interface, which requires explicit attention to release lock, generally this is achieved by releasing the lock in a finally block.

**Explain Synchronization and Multithreading.**

Traditionally all our programs run only on a single thread (main thread) and the thread completes when the sequential execution of code is over. Java permits Multi-Threading, where we can have more than one threads, each executing an independent module of its own to gain parallelization in the code, making it efficient. Then comes the problem of shared resource. When we have multiple threads in our program, and we have a particular variable or piece of code which works on a sensitive data. And different threads may access the code at a same time, therefore interfering with each other’s operation, and giving dirty values to the program. Java has a *synchronized keyword* and a *synchronized block* to avoid this issue. Any method marked synchronized becomes thread safe and can be accessed only by one thread at a time; similarly any block of code inside the synchronized block becomes thread safe and only one thread can be inside that block at a time.

**Difference between synchronized block and method in Java.**

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 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.

**Explain Synchronization.**

Synchronized is the keyword applicable for methods and blocks but not for classes and variables.

 If a method or block declared as the synchronized then at a time only one Thread is allow to execute that method or block on the given object.

 The main advantage of synchronized keyword is we can resolve date inconsistency problems.

 But the main disadvantage of synchronized keyword is it increases waiting time of the Thread and effects performance of the system.

 Hence if there is no specific requirement then never recommended to use synchronized keyword.

 Internally synchronization concept is implemented by using lock concept.

 Every object in java has a unique lock. Whenever we are using synchronized keyword

then only lock concept will come into the picture.

 If a Thread wants to execute any synchronized method on the given object 1st it has to get the lock of that object. Once a Thread got the lock of that object then it’s allow to execute any synchronized method on that object. If the synchronized method execution completes then automatically Thread releases lock.

 While a Thread executing any synchronized method the remaining Threads are not allowed execute any synchronized method on that object simultaneously. But remaining Threads are allowed to execute any non-synchronized method simultaneously. [lock concept is implemented based on object but not based on method].

**What is Daemon thread?**

A daemon thread is a thread that does not prevent the JVM from exiting when the program finishes but the thread is still running. An example for a daemon thread is the garbage collection.

You can use the setDaemon(boolean) method to change the Thread daemon properties before the thread starts.

**Priority of a Thread (Thread Priority):**

Each thread have a priority. Priorities are represented by a number between 1 and 10. In most cases, thread schedular schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.

**3 constants defiend in Thread class:**

public static int MIN\_PRIORITY

public static int NORM\_PRIORITY

public static int MAX\_PRIORITY

Default priority of a thread is 5 (NORM\_PRIORITY). The value of MIN\_PRIORITY is 1 and the value of MAX\_PRIORITY is 10.