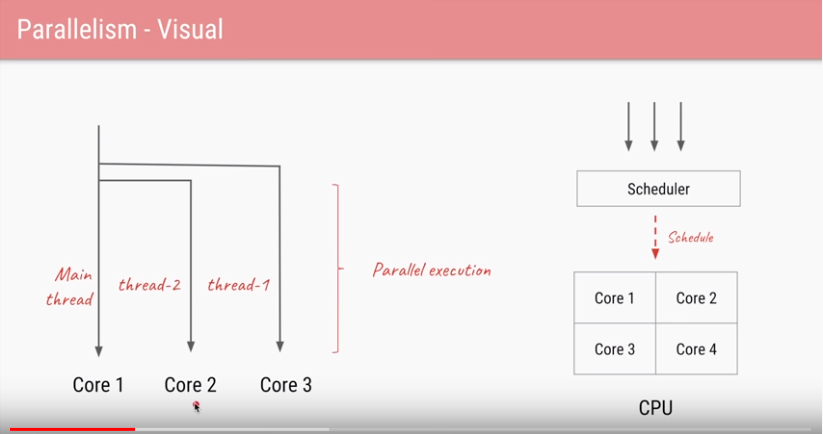
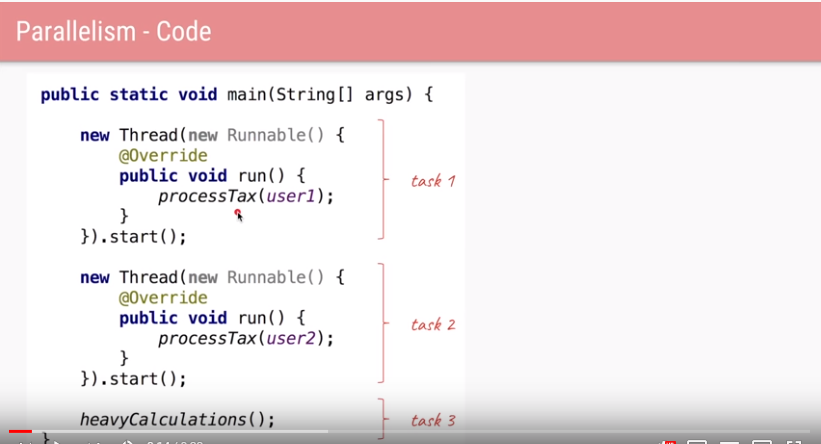
Java Concurrency

Parallel vs Concurrency

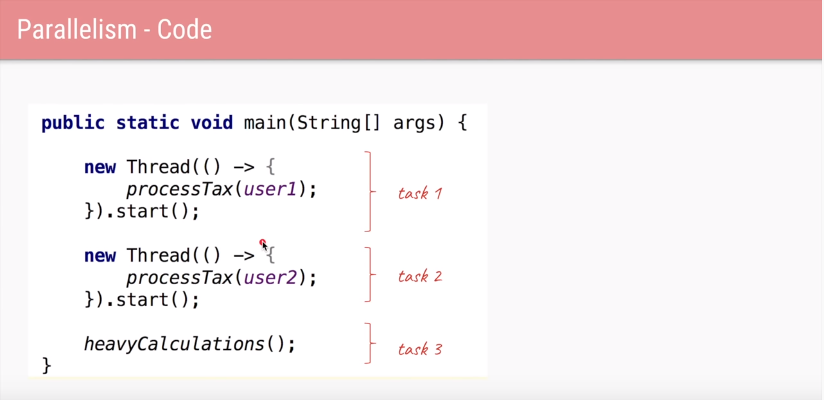
Parallelism

Parallelism is about doing lot of things at once





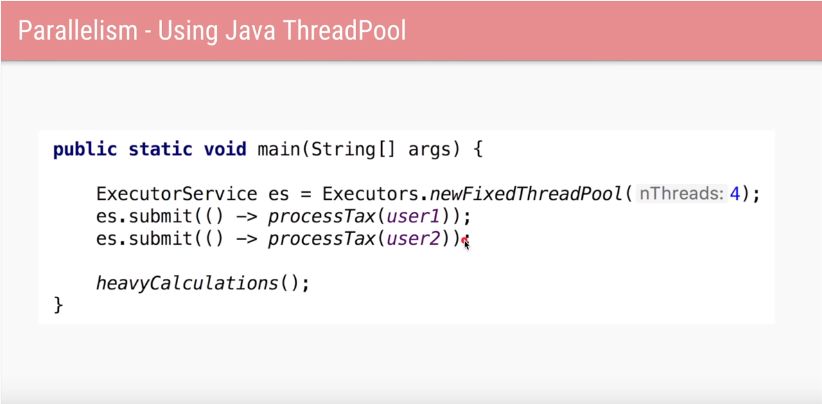
In Java 8.0 we can simplify the above code as shown below

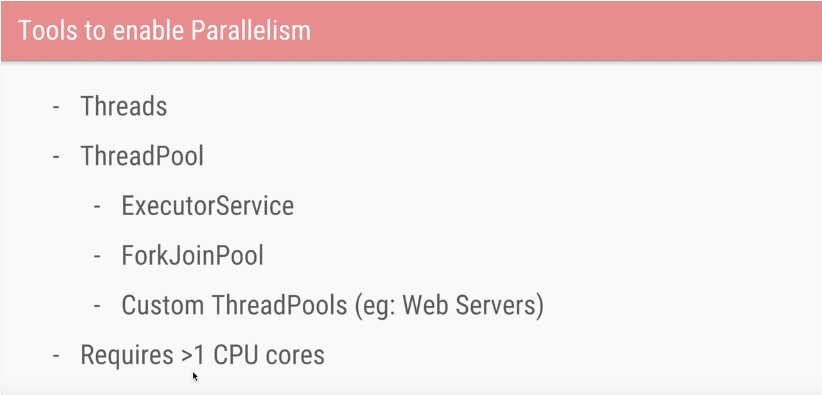


“Parallelism is about doing lot of things at once

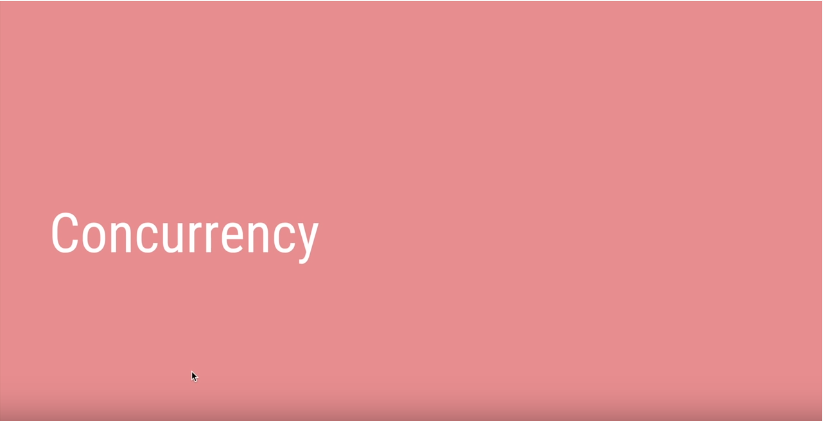


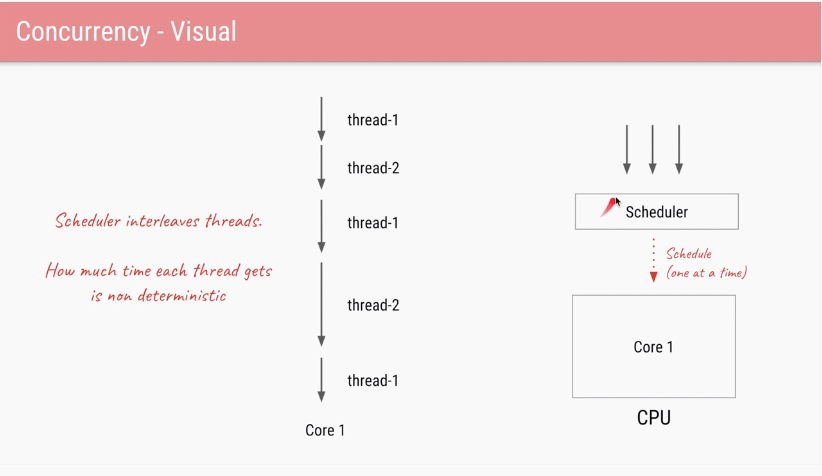
We can also use Java ThreadPool to create multiple threads

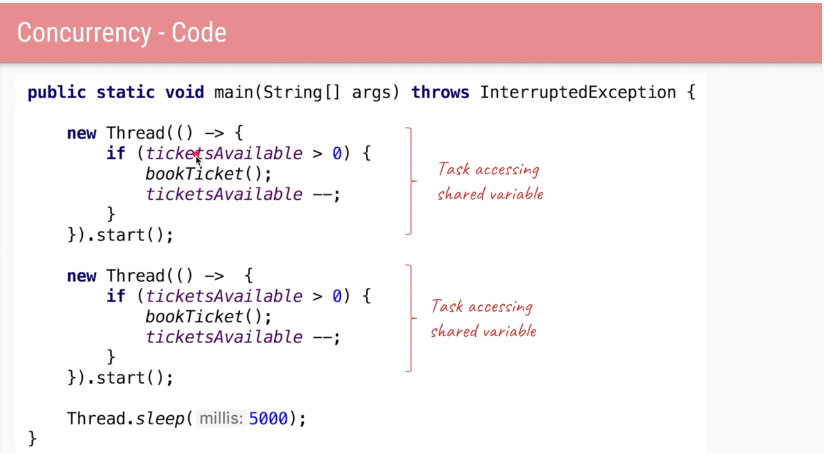


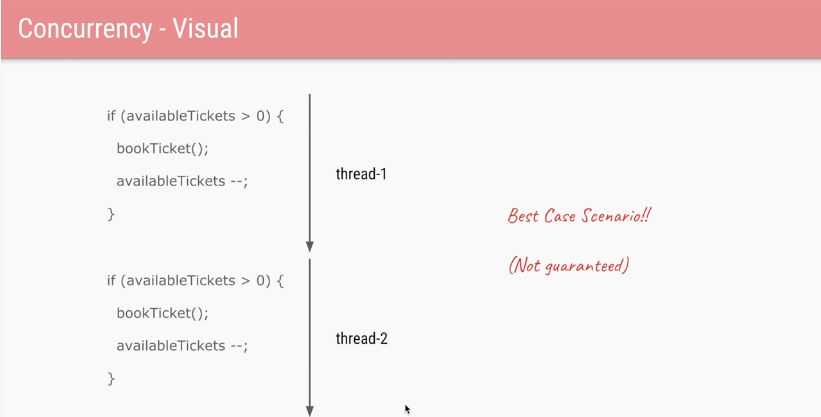


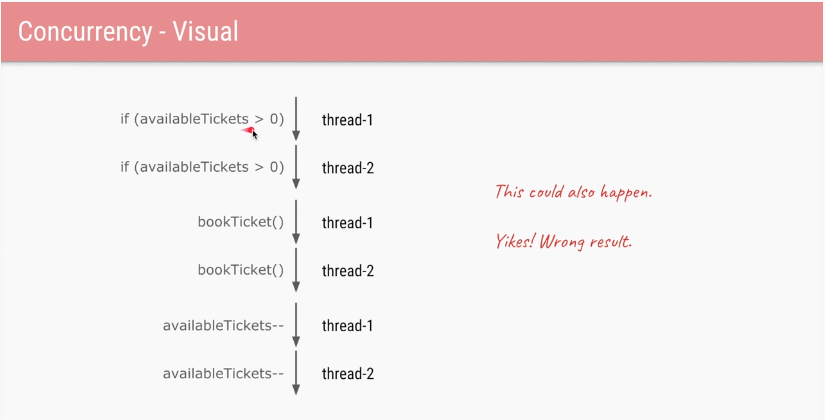
Concurrency

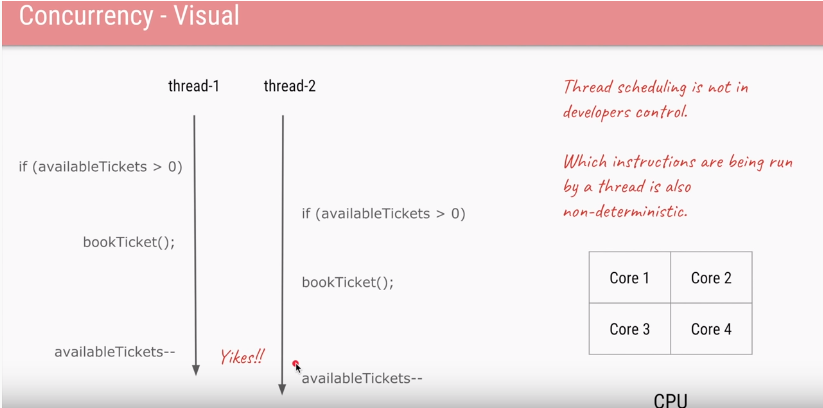




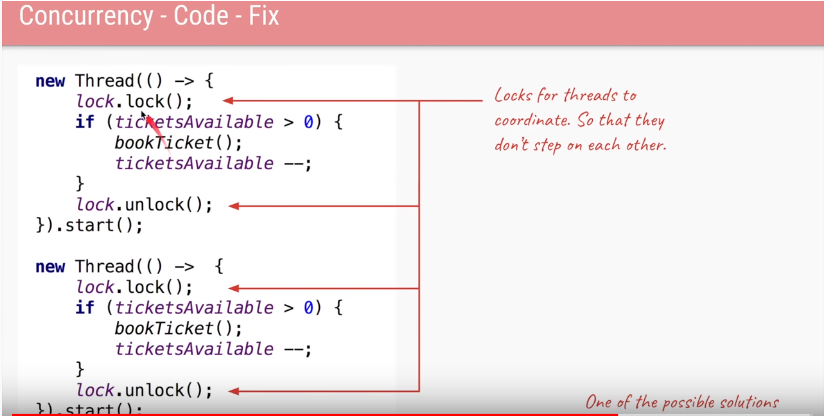


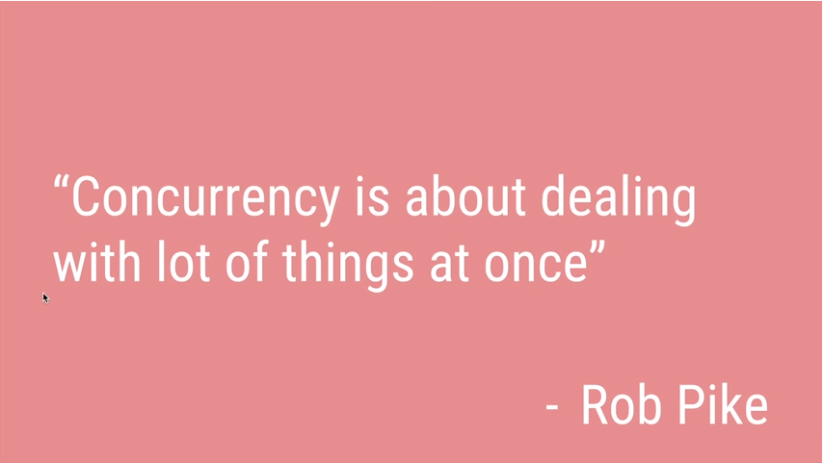




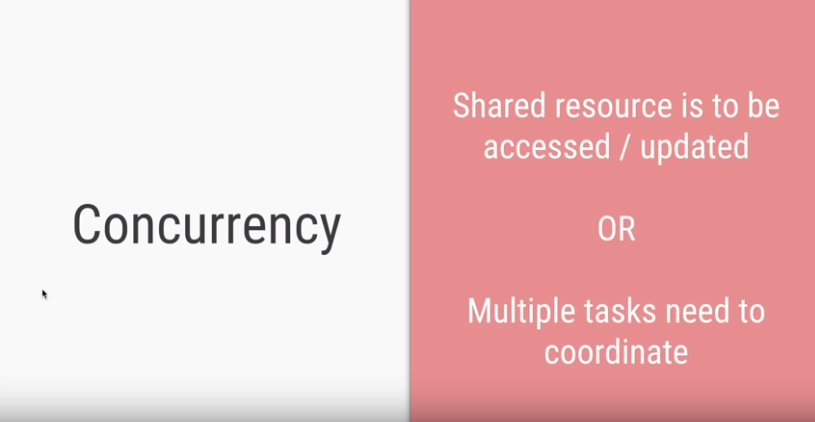


To fix this the above problem one of the possible solution is introduce the locks

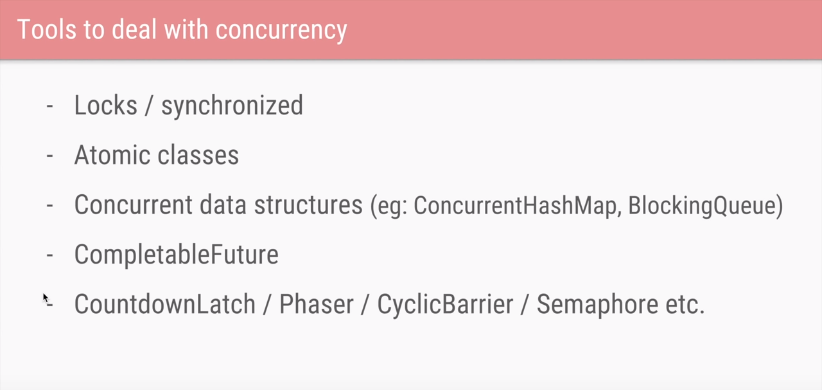




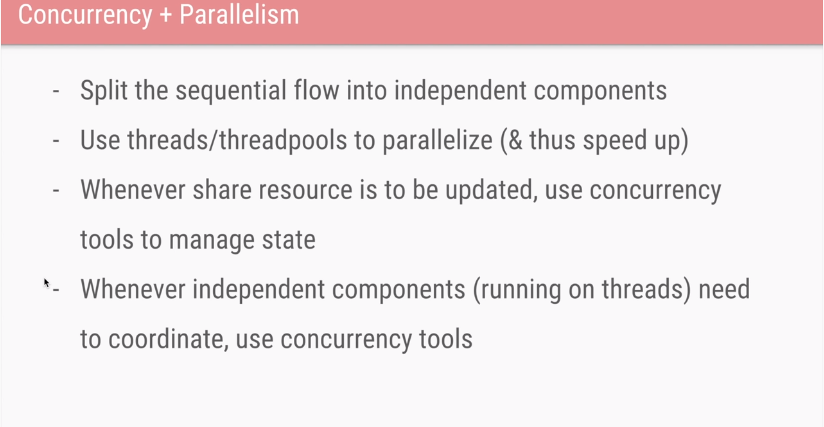
Concurrency is applied when



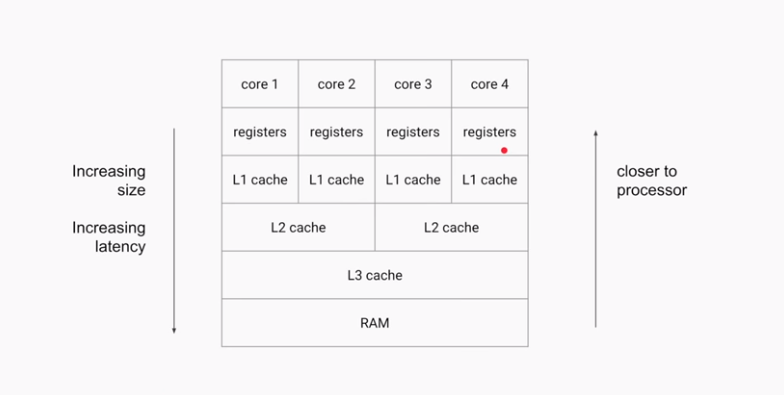
In Java there are lot of tool to deal with concurrency



Parallelism is about doing lot of things at once whereas Concurrent is about dealing with lot of things happing at once

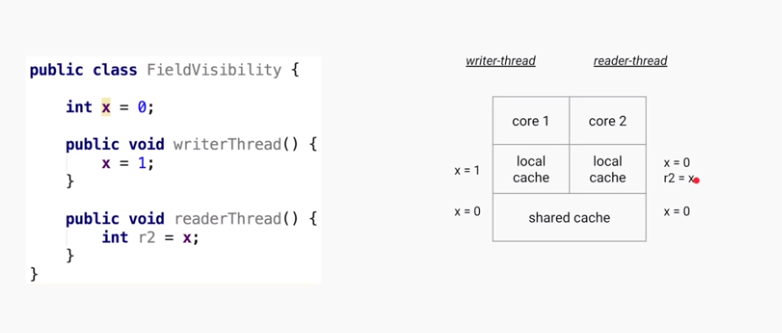


Java Memory Model

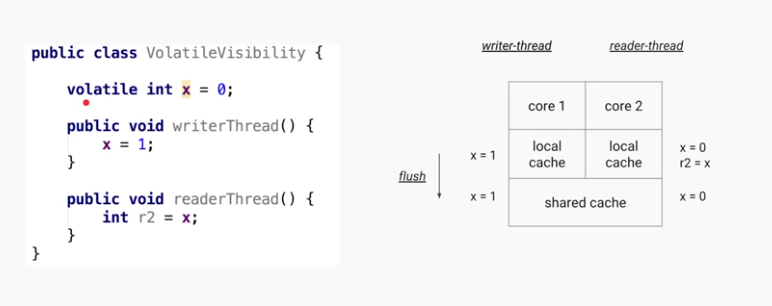


Field Visibility

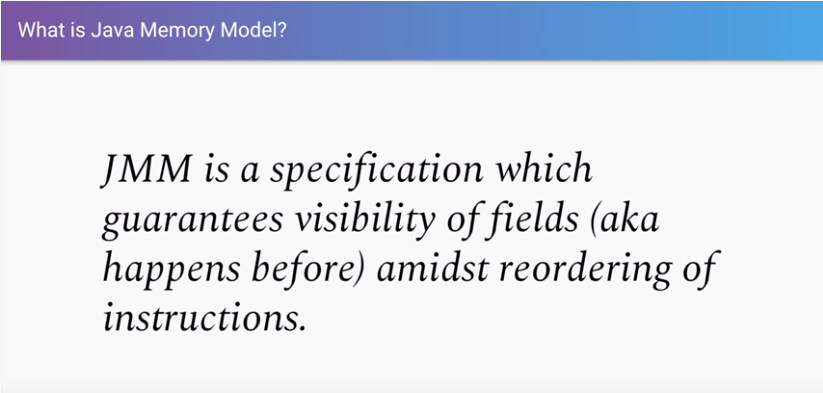
Assume that FieldVisibility object created and share between two threads. One thread is writing x =1 and if at same time another thread accessing the x, there is no guarantee that the second thread will get the updated value. Because variable x is not visible to thread 2 in true sense.



To overcome that java provides volatile key word so that if a volatile variable set to new value JVM ensures that it will immediately flushes from local cache to shared cache /RAM .Hence other thread always gets the updated value

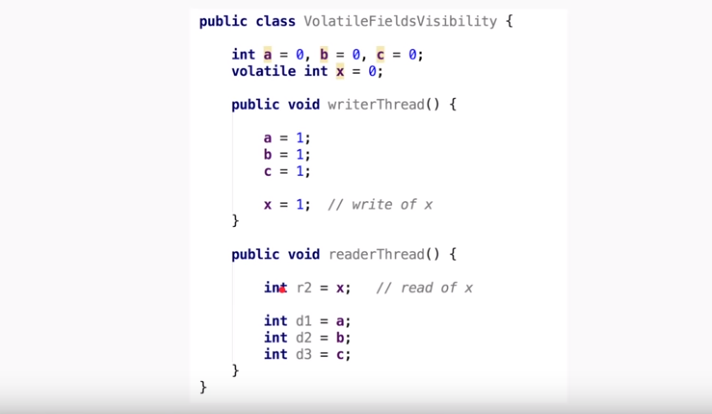




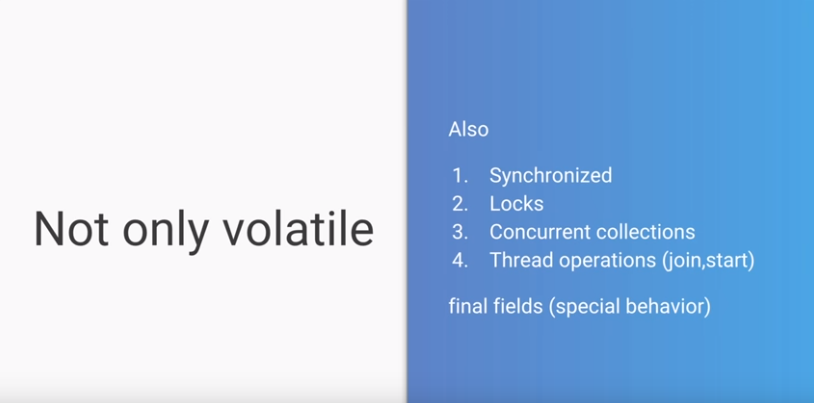


Happens Before relationship

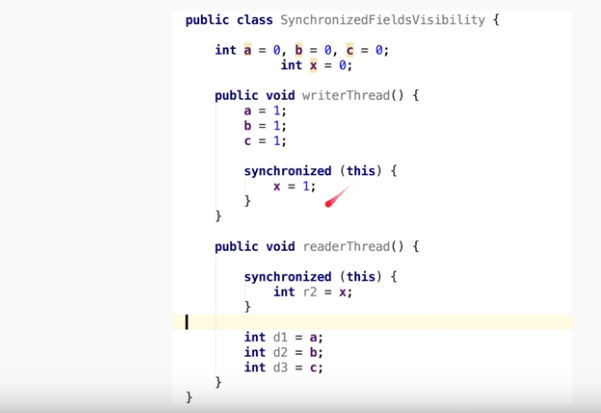
Any fields which are written to before write x those values updated and visible to any other threads after it has read the value of x. What ever happened to before the variable x should be visible after read the x. That is it is called Happens Before relationship.

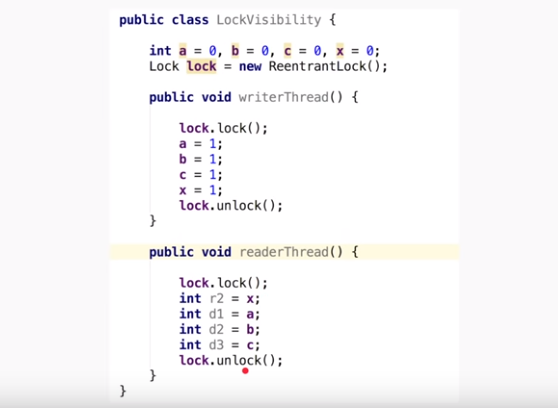


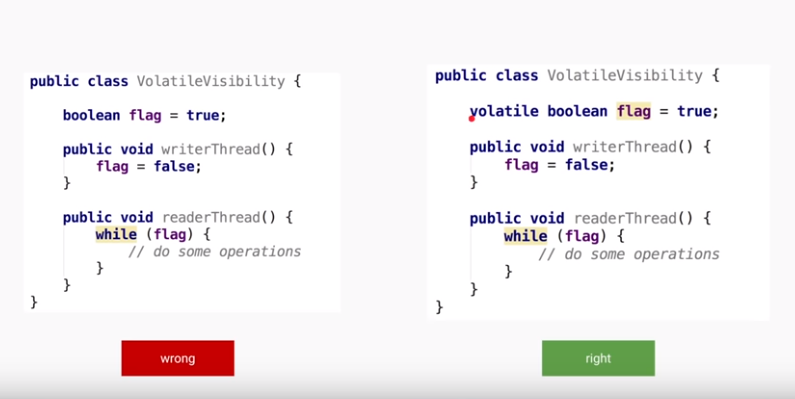
In fact this happens before relationship concept is not only applicable for volatile keywords it is also applicable for synchronized/Locks/Concurrent collections/Thread operations. Any fields which are written to before write x those values updated and visible to any other threads after it has read the value of x. What ever happened to before the synchronized keyword/block, x should be visible after read the x. That is it is called Happens Before relationship. The caveat is if use the synchronized keyword on different objects it will not works.

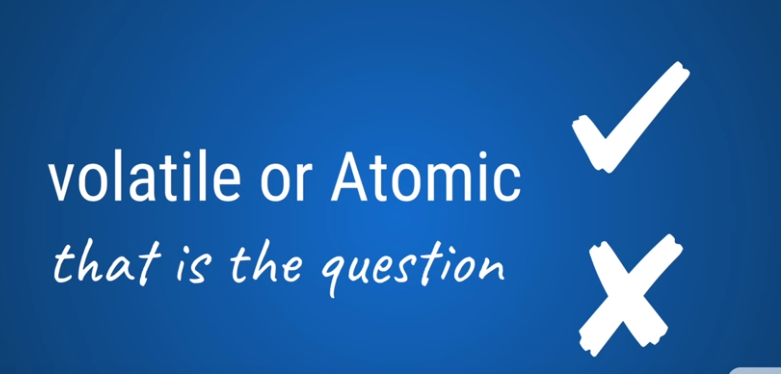


It also the same happens before relationship for synchronized blocks if







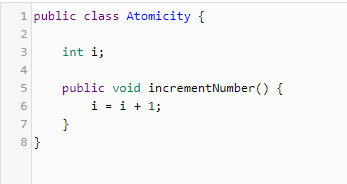


What is Atomic Variable?

Before understanding atomic variables let’s understand what do we mean by atomic or atomicity.

Atomic operations are those operations that ALWAYS execute together. Either all of them execute together, or none of them executes. If an operation is atomic, then it cannot be partially complete, either it will be complete, or not start at all, but will not be incomplete.

Consider below example



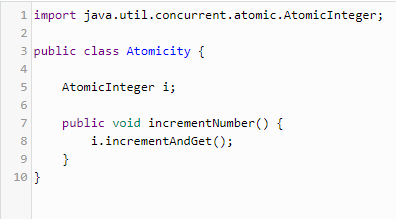
At first glance, this might seem like an atomic operation, but it is actually not. This one-liner actually consists of three operations.

* Read operation, where the value of i is read.
* Modify operation, where a new value is being calculated (i + 1).
* Write operation, where the new value is written to the variable i.

Atomic Variables

Atomic variables come to the rescue in these types of situations. Atomic variables allow us to perform atomic operations on the variables.

Consider the following example



The above example shows how an AtomicInt can be used to update the value atomically. What incrementAndGet() does is atomically increment the value by 1, and then returns the updated value. If the program is now run in a multi-threaded environment, supposing with 2 threads, then the end result will always be i being equal to 2. This is because no matter which thread gets to the incrementAndGet() method first, since it is an atomic operation, the thread will update the value of i to 1, and only then another thread will be able to access or update it, which will make the value of i to 2, thus giving us the correct result.

The most commonly used atomic classes are – AtomicInt, AtomicLong, AtomicBoolean, and AtomicReference. All of these provide atomic operations for the respective classes. AtomicReference can be used for just any type of object.