Volatile keyword

A volatile keyword is a field modifier that ensures that the object can be used by multiple threads at the same time without having any problem. volatile is one good way of ensuring that the Java program is thread-safe. a volatile keyword can be used as an alternative way of achieving Thread Safety in Java.

Volatile variables have the visibility features of synchronized but not the atomicity features. The values of the volatile variable will never be cached and all writes and reads will be done to and from the main memory.

Atomic Variable

Using an atomic variable is another way to achieve thread-safety in java. When variables are shared by multiple threads, the atomic variable ensures that threads don't crash into each other.

An atomic variable ensure consistence of data. There are different type of atomic variable except for double and float.

CAS for atomic : **compareAndSet(oldvalue,newValue)**

Race condition

Definition:

A race-condition is present when a program reads the value of a variable/field and needs then to perform an operation that depends on that value (e.g. check-then-act or read-modify-write).

- **Check then act:** it's a problem that concern decision instructions, they took decisions based on possibly old values, example if condition.
- **Read modify write**: it's a problem that concern edit instructions, it updated date based on not updated data, example var++
- ***Compound actions:*** *they are sequences of instructions that must be execute atomically to maintain the integrity, so be* ***thread-safe****.*

Synchronized block:

It has two parts. A reference to an object (our choice, even "new Object( )") used as a lock and a code block to be protected by the block. The lock is automatically activated when accessing and automatically deactivated when exiting. Works also with exception throws.

Synchronized method:

It used the object in which the method its written as a lock. Its like using the synchronized block with "this". In case of static methods, the class object it's used as our lock.

Explicit locks:

We instantiate a lock from ReentrantLock. We call .lock( ) from our lock, then we create a try-finally codeblock where we put our operation, and in the finally we .unlock( ) the lock. In this way we could also share a lock, or unlock / unlock in other methods instead where we are making the operations.

Lock vs RW-Locks

With the RW-Lock you tell the OS to permit multiple READINGS simultaneously and lock unlock-lock when a write need to be done. The normal lock instead permit only a single operation (reading or writing) for each locking time.

One writer- many readers:

- volatile variable
- use thread confinement, local variable to save shared value, we perform actions on locals either for readers(avoid check-then-act) and writer.

This in the constructor is dangerous, so create object then call a method of the final object that call this.

Immutable class has **final fields** and **synchronized** method to edit/perform actions on mutable objects, they return new object with new value if you want edit data.

Publishing an object means making it accessible to the code outside the scope of the class (or the method, in case of local variables), that owns it in a specific moment.

Safe publication of an object means that the reference to the object and its state are made accessible simultaneously (in an atomic way) => this can only be done by using synchronization tools. Visibility problems must be avoided!

Thread-confined objects: are accessed (read and write) only by the owning thread.

Shared read-only objects: can be read concurrently without synchronization. Include immutable objects as well as effectively immutable objects.

Shared thread-safe objects: perform synchronization internally. Can be accessed (read and write) concurrently without synchronization in the client code.

Shared objects protected by locks: can only be accessed (read and write) when holding the associated lock.

All mutable objects must be safely published. Synchronization is required on both sides: the publisher, as well as the recipient of the publication.
there are synchronized collections, because normal version of collections are not thread safe.
We can find these type of collections with:

```
List synchList = Collections.synchronizedList(new ArrayList());
```

they have poor performance, **poor concurrency**.

Client side locking

```
synchronized (myList) {
        Interator<Integer> it=myList.iterator();
        while(it.hasNext())
                doSomething(it.next)
}
```

to use synchronized collection use client side locking if you use iterator, Synchronized list can give a lot of problems an alternative could be **Immutable collections.**

Immutable collections

they are unmodificable collections are similar to immutable objects (but are not completely immutable).
It provide read-only views of collection objects.
This collections don't guarantee correct memory visibility. If modified the read-only views should be safely republished or outdated values have to be tolerated.

```
List<String> list = List.of("One", "Two", "Three", "Four");
  Map<String, String> map = Map.of("One", "1", "Two", "2", "Three", "3");
  Map<String, String> map = Map.ofEntries(
                              new AbstractMap.SimpleEntry<>("One", "1"),
                              new AbstractMap.SimpleEntry<>("Two", "2"),
                              new AbstractMap.SimpleEntry<>("Three", "3"));
```

Immutable collections don't accept null values
Immutable collection are not the solution of all problems.
If use ofXXX(....immutable objects...) method we can create immutable collection, remove method not supported, you can't modify the list.
To create an immutable collection use Collection class static methods, example: Collections.*unmodifiableList*(lst);
Unmodifiable collections are collection useful for read only and for confinement of a collection in one writer many reader approach.
Can't be modified!

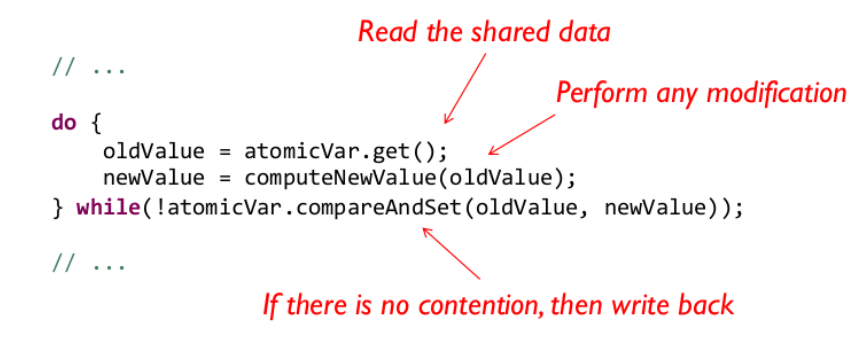
Concurrent collections

these collections are designed to support concurrent access by multiple thread

- ConcurrentHashMap - replaces the synchronized HashMaps
- ConcurrentSkipListMap - replaces the synchronized TreeMap and other synchronized SortedMaps
- CopyOnWriteArrayList - replacement for the various versions of synchronized Lists, for situations where traversing (read) is the predominant operation
- CopyOnWriteArraySet - replacement for the various versions of synchronized Set, for situations where traversing (read) is the predominant operation

get put contains are slow operations, use cas to modify for atomicity.
AtomicLongArray is an array with atomicity, there are atomic arrays for every type, for object use AtomicReferenceArray.

IF I USE CAS USE ALWAYS THE FOLLOWING CODE BLOCK!!! ALSO FOR CONCURRENT HASHMAP IN REPLACE METHOD PUTTING OLD AND NEW VALUE



Shared variables passed as parameter is a good choice create a clone of the object and not use the reference, (ABA problem).
In the concurrenthasmap merge method is needs to **unify** old and new value to existing key, needs lambda (old,new)→{....}.
through composition and delegation we can use normal collection in thread-safe (synchronized methods that delagate)