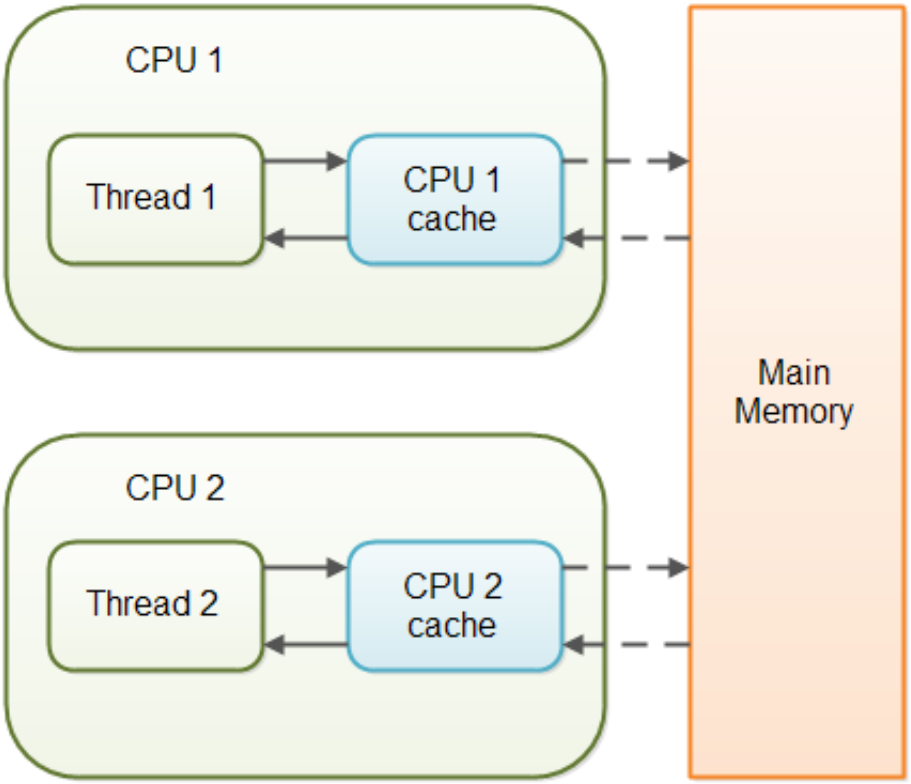
volatile is probably the less known/understood/documented keyword in Java. volatile keyword can be used in variables to indicate compiler and JVM that always read its value from main memory and follow happens-before relationship on visibility of volatile variable among multiple thread.



The following shows a basic example where volatile is required

VolatileTest.java:

```
----------------------------------------
public class VolatileTest {
private static final Logger LOGGER = MyLoggerFactory.getSimplestLogger();

private static volatile int MY_INT = 0;

public static void main(String[] args) {
new ChangeListener().start();
new ChangeMaker().start();
}


static class ChangeListener extends Thread {
@Override
```

```
public void run() {
int local_value = MY_INT;
while ( local_value < 5){
if( local_value!= MY_INT){
LOGGER.log(Level.INFO,"Got Change for MY_INT : {0}", MY_INT);
local_value= MY_INT;
}
}
}
}
```

```
static class ChangeMaker extends Thread{
@Override
public void run() {

int local_value = MY_INT;
while (MY_INT <5){
LOGGER.log(Level.INFO, "Incrementing MY_INT to {0}", local_value+1);
MY_INT = ++local_value;
try {
Thread.sleep(500);
} catch (InterruptedException e) { e.printStackTrace(); }
}
}
}
}
```

With the volatile keyword the output is :

```
Incrementing MY_INT to 1
Got Change for MY_INT : 1
Incrementing MY_INT to 2
Got Change for MY_INT : 2
Incrementing MY_INT to 3
Got Change for MY_INT : 3
Incrementing MY_INT to 4
Got Change for MY_INT : 4
Incrementing MY_INT to 5
Got Change for MY_INT : 5
```

Without the volatile keyword the output is :

Incrementing MY_INT to 1
Incrementing MY_INT to 2
Incrementing MY_INT to 3
Incrementing MY_INT to 4
Incrementing MY_INT to 5
.....And the change listener loop infinitely...

Each thread has its own stack, and so its own copy of variables it can access. When the thread is created, it copies the value of all accessible variables in its own memory. The volatile keyword is used to say to the jvm "Warning, this variable may be modified in an other Thread". Without this keyword the JVM is free to make some optimizations, like never refreshing those local copies in some threads. The volatile force the thread to update the original variable for each variable. The volatile keyword could be used on every kind of variable, either primitive or objects!

When is volatile Enough?

If two threads are both reading and writing to a shared variable, then using the volatile keyword for that is not enough. You need to use synchronization in that case to guarantee that the reading and writing of the variable is atomic.

But in case one thread reads and writes the value of a volatile variable, and other threads only read the variable, then the reading threads are guaranteed to see the latest value written to the volatile variable. Without making the variable volatile, this would not be guaranteed.

Performance Considerations of volatile:

Reading and writing of volatile variables causes the variable to be read or written to main memory. Reading from and writing to main memory is more expensive than accessing the CPU cache. Accessing volatile variables also prevent instruction reordering which is a normal performance enhancement technique. Thus, you should only use volatile variables when you really need to enforce visibility of variables.