301AA - Advanced Programming

Lecturer: **Andrea Corradini**
andrea@di.unipi.it
http://pages.di.unipi.it/corradini/

***AP-22**: The Java Memory Model*

Memory model

- A **memory model** for multithreaded systems specifies how memory actions (e.g., reads and writes) in a program will appear to execute to the programmer, and specifically, which value each read of a memory location may return.
- Every hardware and software interface of a system that admits **multithreaded access to shared memory** requires a memory model.
- The model determines the transformations that the system (*compiler*, *virtual machine*, or *hardware*) can apply to a program.
  - For example, given a program in machine language, the memory model for the machine language / hardware interface will determine the optimizations the hardware can perform.

Memory model (2)

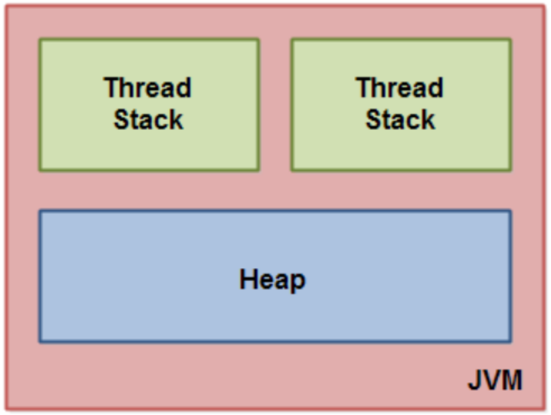
- For a high-level programming language such as **Java**, the memory model determines
  - the transformations the compiler may apply to a program when producing bytecode,
  - the transformations that a virtual machine may apply to bytecode when producing native code, and
  - the optimizations that hardware may perform on the native code.
- The model also impacts the programmer; the transformations it allows (or disallows) determine the possible outcomes of a program, which in turn determines which design patterns for communicating between threads are legal.
- Without a well-specified memory model for a programming language, it is impossible to know what the **legal results** are for a program in that language.

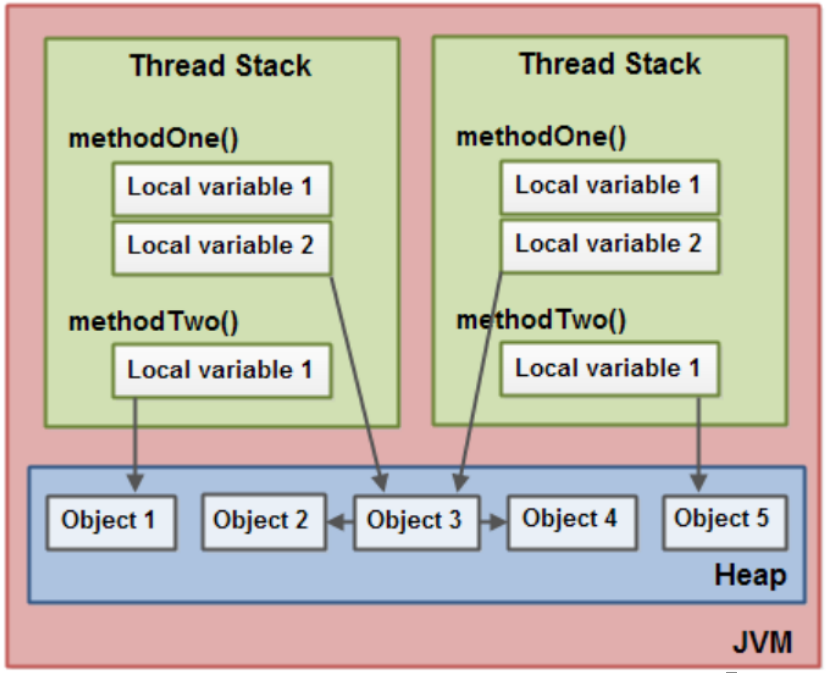
JMM - Java Memory Model

- New version with Java 5 (2004)
- Specifies legal behaviour of multithreaded programs
- Provides standard guarantees for **correctly synchronized programs**: *sequential consistency* of *data-race-free* programs
- For **incorrectly synchronized program** the behaviour is bounded by a well-defined notion of causality
- The causality constraints are strong enough to respect the **safety and security properties of Java** and weak enough to allow standard compiler and hardware optimizations.
- This was one of the first memory models for high level programming languages.
  - The one of C++ was introduced with C++11.

JVM Runtime Data Areas

- Local variables of methods are allocated on thread stack (primitive types)
- Local variables cannot be accessed by other threads
- Objects are allocated on the heap
- Only objects (and their fields) can be shared among threads

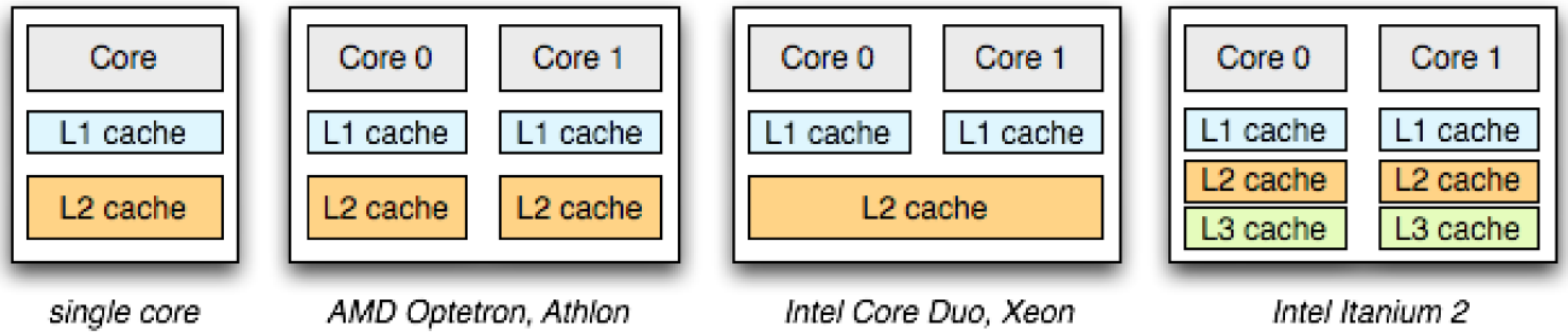




Memory Hierarchy

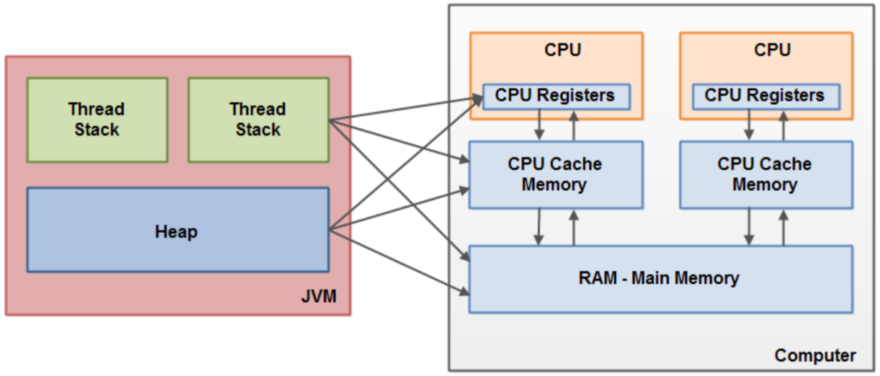
- In modern architectures, memory is **stratified** into different levels, ranging from mass memory (hard disk) to CPU registers, passing through different levels of cache
  - this stratification is called **memory hierarchy**
- Some layers, such as registers and first cache layers, are separated between different cores
- Other layers, such as RAM, are shared between cores

For example, this is a part of the memory hierarchy in some processors:



JVM data areas and Memory Hierarchy

- They are orthogonal:



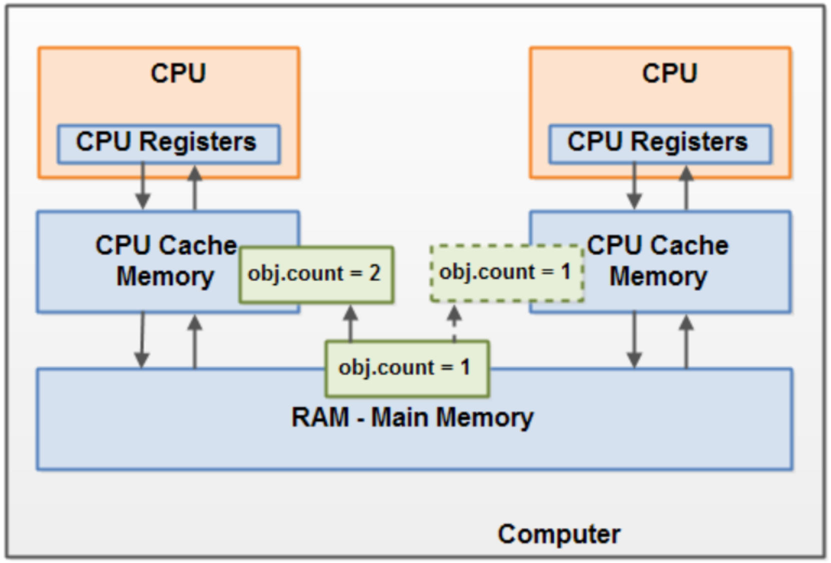
- Access to shared variables, possibly in different memory areas, leads to two main problems:

  Visibility of updates & **Data races**

- Java provides two mechanism to synchronize accesses to shared memory: the **volatile** modifier and **synchronized methods / blocks**
- The JMM takes them into account explicitly

Visibility of updates of shared objects

- Due to caches and to policies of “flushing”



- Possible solution: the **volatile** modifier

Visibility problems

What is the output of this program?

```java
public class Loop {
    static boolean done;
    static int n;
    public static void main(String args[]) {
        Thread t = new Thread() {
            public void run() {
                n = 42;
                try {
                    sleep(1000);
                } catch (InterruptedException e) {
                    return;
                }
                done = true;
                System.out.println("Done");
            }};
        t.start();
        while (!done) {
        }
        System.out.println(n);
}}
```

```
                      done = false
                      n = 0

Thread 1 (main):            Thread 2:
----------------------------------------------------------
while (!done) { };          n = 42;
System.out.println(n);      sleep(1000);
                            done = true;
                            System.out.println("Fatto");
```

- Note that the two threads share the variables **n** and **done**, but do not use any synchronization mechanism.
- The *while* loop of thread 1 can behave like an infinite loop, even if thread 2 after waiting one second executes **done = true**.
- This is because, in the absence of synchronization, there is no guarantee as to when the write to the **done** variable made by thread 2 will be visible to thread 1.
- The JMM allows this behavior.
- To avoid it, the variable **done** can be declared **volatile.**

The **volatile** modifier

- **volatile** is a modifier that can only be applied to fields of a class
- Intuitively, volatile declares that that field can be accessed or modified by multiple threads
  - A volatile field cannot be **final**: it would be meaningless
- The JMM guarantees (see later) that the write of a volatile variable is visible when it is read
- An implementation should guarantee that the new value is flushed from the cache to the RAM

```java
public class Loop {
    static volatile boolean done;
    static int n;

    public static void main(String args[]) {
        Thread t = new Thread() {
            public void run() {
                n = 42;
                try {
                    sleep(1000);
                } catch (InterruptedException e) {
                    return;
                }
                done = true;
                System.out.println("Fatto");
            }
        };
        t.start();
        while (!done) {
        }
        System.out.println(n);
    }
}
```

Each reading of the **done** variable made by the main thread makes visible the changes to **done** made by the other thread

Therefore, the main thread always terminates.

Data Races

- Simplifying, there is a **data race** if two threads can execute two **conflicting** actions on a shared variable in any order
  - conflicting: at least one is a write
- Depending on the execution interleaving, different values can be obtained, leading to logical inconsistencies
- **volatile** does not help: **synchronized** needed

```
static Counter sharedCounter = new Counter();
public static void main(String... args)
  throws InterruptedException{
Runnable r = () -> {
      for (int i = 1; i <= 10000; i++) {
            sharedCounter.incr();
        }};
    Thread t1 = new Thread(r);
    Thread t2 = new Thread(r);
    t1.start(); t2.start();
    t1.join();  t2.join();
    System.out.println(sharedCounter.getCount());
}  // prints values smaller than 10000
```

```
class Counter {
    int count;
    public void incr() {
        count++;
    }
    public int getCount() {
        return count;
    }
}
```

Mutual exclusion using monitors

- Every Java object has a monitor, offering lock/unlock
- Only one thread at a time can hold the lock
- "**synchronized**" methods or blocks take the lock at start and release it at the end
- Thus, synchronized methods / blocks are in mutual exclusion
- Declaring the **incr()** method synchronized guarantees that the output of the previous program is always 20000

```
class Counter {
    int count;
    public synchronized void incr() {
        count++;
    }
    public int getCount() {
        return count;
    }
}
```

Back to the Java Memory Model

- The Java Memory Model has no explicit global ordering of all actions by time, consistent with each thread’s perception of time, and has no global store.
- Instead, executions are described in terms of memory related actions, partial orders on these actions, and a visibility function that assigns a write action to each read action.

Actions of the JMM

Actions are memory-related operations.

Each action a has a thread $T(a)$ and a kind:

- Volatile read of $v \in \mathbb{L}$
- Volatile write of $v \in \mathbb{L}$
- Lock on monitor $m \in \mathbb{M}$
- Unlock of monitor $m \in \mathbb{M}$
- Normal read from $v \in \mathbb{L}$
- Normal write to $v \in \mathbb{L}$
- External action

"Synchronization actions" are in red

- The JMM dictates rules to decide if a set of actions is a legal execution of a program

Program order & sequential consistency

- An execution of a single-threaded program fixes a total order $\leq_{po}$ on its actions, called **program order**
- For a multi-threaded program, the program order is the union of those of its threads: it does not relate actions of different threads
- An execution of a multi-threaded program is **sequentially consistent** if there is a total order of its actions consistent with the program order (and such that each read of *v* has the value of the last write to *v*)
- For data-race-free multi-threaded programs, the JMM guarantees that only sequential consistent executions are legal

Guarantees of the JMM

The JMM has been designed to make three guarantees:

1. A promise for programmers: **sequential consistency must be sacrificed** to allow optimisations, but it will still hold for data race free programs. This is the data race free (DRF) guarantee.
2. A promise for security: even for programs with data races, values should not appear “out of thin air”, preventing unintended information leakage.
3. A promise for compilers: common hardware and software optimizations should be allowed as far as possible without violating the first two requirements.

The complexity of the new JMM is justified by the goal of ensuring guarantees 2 **even for non-data-race-free** programs. The previous approach considered such program erroneous, with an unspecified semantics.

Why is sequential consistency too strong?

```
// Thread1
int r1;
r1 = B;
A = 1;
```

```
// Thread2
int r2;
r2 = A;
B = 1;
```

- Consider the following two threads (A == B == 0 initially):
- Which values can take r1 and r2?

  r1 = 0, r2 = 1 if Thread1 executes first

  r1 = 1, r2 = 0 if Thread2 executes first

  r1 = 0, r2 = 0 if scheduler stops one thread in the middle

  r1 = 1, r2 = 1 ???? Yes, in some scenarios
- Note that the last execution is not sequential consistent: still, the JMM allows it for the sake of guarantee 3, as it does not conflict with 1 (because the program has data-races)
- Conceptually, in the absence of synchronization, the compiler/JVM/CPU is allowed to reorder the instructions, typically to improve performance, as long as this reordering is irrelevant from the point of view **of the single thread**. Indeed, if the order of the instructions of Thread 1 and/or Thread 2 is reversed, the result **r1 = 1, r2 = 1** becomes possible.

“out-of-thin-air” behaviours

```
// Thread1
r1 = x;
y = r1;
```

```
// Thread2
r2 = y;
x = r2;
```

- x == y == 0 initially.
- Can we obtain r1 == r2 == 42 at the end?
- Not really, but in a future aggressive, speculative evaluation…
- We say that 42 comes “out-of-thin-air”
- Even if a program contains data races, there must be some security guarantees. This is an unwanted behavior: if a value does not occur anywhere in the program, it should not be read in any execution of the program.
- The out-of-thin-air behaviours could cause security leaks, because references to objects from possibly confidential parts of program could suddenly appear as a result of a self-justifying data race.
- **The JMM forbids such an execution.**

Synchronization order, synchronizes-with and happens-before order

- Each execution of a program is associated with a **synchronization order** $\leq_{so}$, which is a total order over all synchronization actions satisfying:
  - consistency with program order
  - read to a volatile variable v returns the value of the write to v that is ordered last before the read by the synchronization order
- $a \leq_{sw} b$ is read "action a **synchronizes-with** action b". This holds if $a \leq_{so} b$ and
  - a unlocks a monitor and b locks it
  - a writes a volatile variable and b reads it
- Relation happens-before $\leq_{hb}$ is the transitive closure of the program order $\leq_{po}$ and of the synchronizes-with relation $\leq_{sw}$

Data Races

- Two accesses x and y form a **data race** if they are from different threads, they conflict, and they are not ordered by happens-before in a sequential consistent execution
- A program is said to be **correctly synchronized** or **data-race-free** if and only if all sequentially consistent executions of the program are free of data races
- The first requirement for the Java model is to ensure sequential consistency for correctly synchronized or data-race-free programs.
- Programmers then need only worry about code transformations having an impact on their programs' results if those programs contain data races.

Executions, formally

$E = (P, A, \leq_{po}, \leq_{so}, W, V, \leq_{sw}, \leq_{hb})$, where

- P is a **program**
- A is a **set of actions**
- $\leq_{po}$ **program order**, total on actions of each thread
- $\leq_{so}$ **synchronization order**, total on synchronization actions in A
- W - a **write-seen function**, which for each read r in A, gives $W(r)$, the write action seen by r in E.
- V - a **value-written function**, which for each write w in A, gives $V\ (w)$, the value written by w in E.
- $\leq_{sw}$, **synchronizes-with** partial order
- $\leq_{hb}$ **happen-before** partial order

Well-formed executions

$E = (P, A, \leq_{po}, \leq_{so}, W, V, \leq_{sw}, \leq_{hb})$, is well-formed if

- Each read of a variable x sees a write to x. All reads and writes of volatile variables are volatile actions.
- Synchronization order is consistent with program order and mutual exclusion.
- The execution obeys inter-thread consistency.
- The execution obeys intra-thread and happens-before consistency (each read of v sees the last preceding write to v)

Legal executions

- Legal executions are built iteratively. In each iteration, it commits a set of memory actions; actions can be committed if they occur in some well-behaved execution that also contains the actions committed in previous iterations.
- A careful definition of “well-behaved executions” ensures that the appropriate executions are prohibited (e.g. those creating values *out-of-thin-air*) while standard compiler transformations are allowed.

Causality requirements for executions

A well-formed execution $E = (P, A, \leq_{po}, \leq_{so}, W, V, \leq_{sw}, \leq_{hb})$ is validated by **committing** actions in A. If all actions of A are committed, E is legal.

- There must exist a sequence of subsets of A

$$C_0 \subset C_1 \subset \cdots \subset C_n = A$$

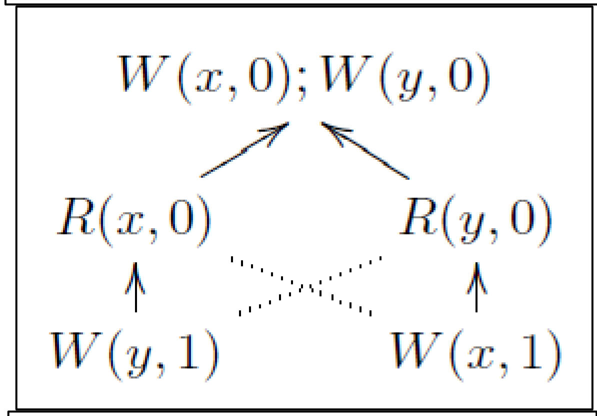
and one $\{E_i\}_{i \leq n}$ of well-formed executions such that each E_i "witnesses" the actions in C_i

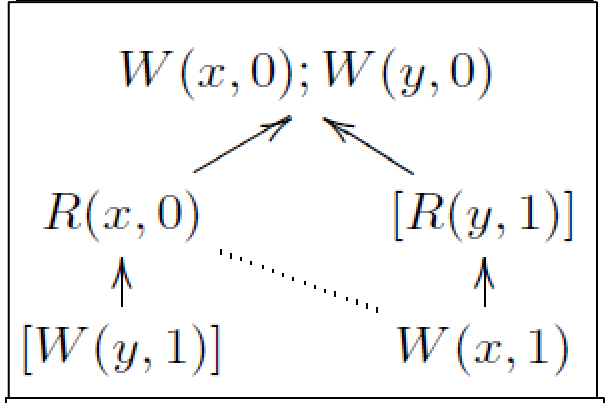
- Complex definition articulated in 9 points. See either the JLS Chapter 7.4 or [Manson05]

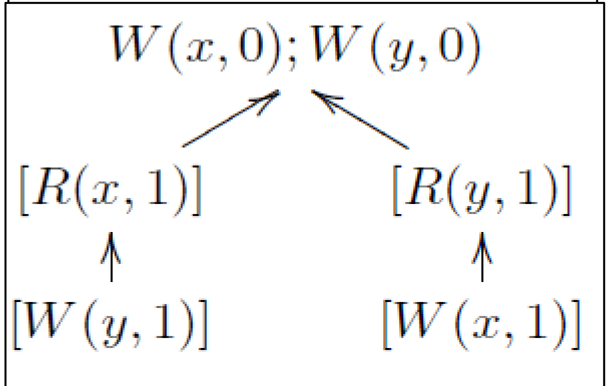
The intuition

- Start with the possible sequential consistent executions of the program
- Identify the data races
- Choose how to resolve one (or some) of them ("commit")
- Start again with executions, using the committed choices

```
initially x = y = 0
r1 := x    | r2 := y
y  := 1    | x  := 1
```







Properties of the formal JMM

The formal model allows to prove results like (see [Manson05]

- “Reordering two adiacent ‘independent’ statements in a program is a ‘legal’ program transformation”
- “Correctly synchronized programs exhibit only sequentially consistent behaviors”
- Correctness of other program transformations like
  - Redundant synchronizations can be removed
  - Volatile fields of thread local objects can be treated as normal fields

JMM from the programmer perspective

- As shown, the JMM describes which executions of a Java program are legal with respect to memory accesses
- Even disregarding the technicalities of the model, its impact can be translated to a set of useful rules for Java programmers
- Rules are of three types:

– **Atomicity**	Which operations are naturally atomic?
– **Visibility**	When does a memory write become visible to other threads?
– **Reordering**	In what order can the operations be rearranged?

Atomicity rules

An operation is **atomic** if, from the point of view of any other thread, its effects are seen in full, or not at all (but never "half")

Which operations are naturally atomic (even in the absence of mutual exclusion mechanisms)?

The JLS, together with the JMM, guarantee that:

1. Reading and writing variables of primitive type (excluding the long and double types) and of reference type are atomic operations
2. Reading and writing **volatile** variables are atomic operations

Note:

- Modification of a **long** variable can occur in two distinct operations, which separately modify the 32 most significant bits and the 32 least significant bits
- These two operations may be interrupted by the scheduler

Examples of atomicity

```
int x, y;
long n;
volatile long m;
Object a, b;
volatile Object c, d;
```

Given these variables, are the following assignments atomic or not?

```
1. x = 8;
2. x = y;

3. n = 0x1122334455667788; //long constant expressed in hexadecimal

4. m = 0x1122334455667788;
5. m++;

6. a = null;
7. a = b;
8. c = d;
```

Rule for visibility

- In absence of synchronization, the operations (writes to memory) performed by a thread can remain hidden from other threads **indefinitely**
- In particular, some operations may remain hidden, and others may be visible

Visibility is guaranteed by the following operations:

1. Acquiring a **monitor** (i.e., entering a synchronized method or block) makes visible the operations performed by the last thread that owned that monitor, up until the moment it released it
2. Reading the value of a **volatile** variable makes visible the operations carried out by the last thread that modified that variable, up to the moment in which it modified it
3. **Invoking t.start()** makes visible to the new thread t all the operations carried out by the calling thread, up to the invocation of start
4. Returning from a **t.join() invocation** makes visible all the operations carried out by thread t until its termination

A comparison between synchronized and volatile

Both **synchronized** and **volatile** offer guarantees of atomicity and visibility

However, volatile modifier makes only **a single write** to the variable in question **atomic**

Even if **a** and **b** are both volatile, "**a = b**" is not atomic

Similarly, "**n++**" is not atomic even if **n** is volatile

A synchronized block or method is the only option to make a sequence of instructions mutually atomic

The volatile modifier is indicated if visibility of changes is required, but not mutual atomicity

Reordering Rules

- As said, the compiler, the JVM and the CPU can reorder any sequence of instructions as long as the result does not change for the single thread.
- The **synchronized** and **volatile** constructs reduce the possibility of reordering.
- Let's consider two subsequent instructions having no dependencies from a single thread perspective:

  x1

  x2
- In which cases can **x1** and **x2** be executed in reverse order?
- We must distinguish three categories of instructions:
  1. **Readings of a volatile** variable, or **start of a synchronized** block or method
  2. **Writes of a volatile** variable, or **end of a synchronized** block or method
  3. All the others ("**normal**" instructions)

Can instructions **x1** and **x2** be swapped?

Type of x2 / Type of x1	Normal	Volatile read / Synchronized start	Volatile write / Synchronized end
Normal	Yes	Yes	No
Volatile read / Synchronized start	No	No	No
Volatile write / Synchronized end	Yes	No	No

Thus:

1. Normal instructions can always be interchanged
2. Normal instructions can be brought into a synchronized block
3. The normal instructions that precede the reading of a volatile can be moved after the reading
4. The normal instructions that follow the writing of a volatile can be moved before the writing

References

The reading material for the JMM is:

- [Manson05] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In POPL ’05: 378–391

Additional material:

- Jaroslav Sevcík, David Aspinall: On Validity of Program Transformations in the Java Memory Model. ECOOP 2008: 27-51
- JLS Chapter 7 – Section 7.4 – Memory Model