В современном мире код часто выполняется не в том порядке, в котором он был написан в программе. Он часто переупорядочивается на уровне:

1. Компилятора байткода (в частности, *javac*)
2. Компилятора машинного кода (в частности, JIT компилятор *HotSpot C1/C2*). Например, среди компиляторов широко распространена такая оптимизация как Instruction scheduling
3. Процессора. Например, в мире процессоров широко распространены такие практики как Out*-*of*-*orderexecution, BranchPrediction*+*Speculation, Prefetching, а также многие другие

Также в современных процессорах каждое ядро имеет собственный локальный кэш, который не видим другим ядрам. Более того, записи могут удерживаться в регистрах процессора, а не сбрасываться в память. Это ведет к тому, что поток может не видеть изменений, сделанных из других потоков.

Все эти оптимизации делаются с целью повысить производительность программ:

1. Переупорядочивание необходимо для того, чтобы найти самый оптимальный путь к выполнению кода, учитывая стоимость выполнения процессорных инструкций. Например, процессор может инициировать загрузку значения из памяти заранее, даже если в порядке программы это чтение идет позднее. Операции чтения из памяти стоят дорого, поэтому эта оптимизация позволяет максимально эффективно утилизировать процессор, избежав простаивания, когда это чтение действительно понадобится
2. Чтение из регистра и кэша стоит сильно дешевле, чем чтение из памяти. Более того, локальный кэш необходим для того, чтобы ядра не простаивали в ожидании доступа к общему кэшу, а могли работать с кэшем независимо друг от друга

Есть хорошие новости, и плохие. Начнем с хорошей:

1. Java дает гарантию [as](https://en.wikipedia.org/wiki/As-if_rule)*[-](https://en.wikipedia.org/wiki/As-if_rule)*[if](https://en.wikipedia.org/wiki/As-if_rule)*[-](https://en.wikipedia.org/wiki/As-if_rule)*[serial](https://en.wikipedia.org/wiki/As-if_rule) выполнения кода — вне зависимости от используемой JDK итоговый результат выполнения будет не отличим от такого порядка, как если бы действия выполнялись действительно последовательно согласно порядку в коде
2. Процессоры тоже делают только такие переупорядочивания, которые не изменят итогового результата выполнения инструкций
3. Процессоры имеют Cache Coherence механизм, который гарантирует консистентность данных среди локальных кэшей: как только значение попадает в локальный кэш одного ядра, оно будет видно всем остальным ядрам

Примеры:

a = 5;

b = 7;

int r1 = a; /\* always 5 \*/

int r2 = b; /\* always 7 \*/

Плохие новости:

1. Java дает as*-*if*-*serial гарантию только для единственного треда в изоляции. Это означает, что в многопоточной программе при работе с shared данными мы можем не увидеть записи там, где полагаемся на порядок выполнения действий в коде другого треда. Другими словами, для первого треда в изоляции валидно переупорядочивать инструкции местами, если это не повлияет на *его* результат выполнения, но переупорядочивание может повлиять на *другие* треды
2. Процессор также дает гарантию только для единственного ядра в изоляции
3. Cache Coherence действительно гарантирует чтение актуальных значений, но пропагация записи происходит не мгновенно, а с некоторой задержкой

**public** **class** **MemoryReorderingExample** {

**private** **int** x;

**private** **int** y;

**public** **void** **T1**() {

x = 1;

**int** r1 = y;

}

**public** **void** **T2**() {

y = 1;

**int** r2 = x;

}

}

1. Обе записи идут до чтений, поэтому выполнение программы начинается или с записи x, или с записи y
2. Перед любым из чтений должна была произойти как минимум одна запись

JMM:  
Java имеет свою модель памяти под названием **Java Memory Model** (JMM). По умолчанию JMM разрешает любые переупорядочивания и не гарантирует видимости изменений. Однако при выполнении определенных условий нам гарантируется порядок действий, консистентный с порядком в коде, а также видимость всех изменений. Таким образом, JMM позволяет нам писать программы, которые будут полностью корректно работать среди множества различных имплементаций JDK и микро-архитектур процессоров, в то же время сохраняя преимущества оптимизаций.

Модель памяти:  
1. Разрешает выполнение различных оптимизаций компилятора, JVM или процессора

2. Строго закрепляет условия, при которых программа считается правильно синхронизированной, и закрепляет поведение правильно синхронизированных программ

3. Описывает отношение между высокоуровневым кодом и памятью

4. Является trade-off между строгостью исполнения кода и возможными оптимизациями

Volatile

Volatile предоставляет happens-before гарантию между записью и чтением из volatile переменной. Семантика volatile отличается от монитора только тем, что не предоставляет эксклюзивный доступ к критической секции.

**public** **class** **VolatileHappensBefore** {

**private** **int** x;

**private** **volatile** **boolean** initialized;

**public** **void** **writer**() {

x = 5; /\* W1 \*/

initialized = **true**; /\* W2 \*/

}

**public** **void** **reader**() {

**boolean** r1 = initialized; /\* R1 \*/

**if** (r1) {

**int** r2 = x; /\* R2, guaranteed to see 5 \*/

}

}

}

Final fields

JMM дает очень полезную гарантию порядка и видимости записей для final полей: если ссылка на создаваемый объект не утекла во время работы конструктора (так, что ее мог увидеть другой тред), то все остальные треды, которые увидели non-null ссылку на этот объект, гарантированно прочитают актуальные значения всех внутренних final полей объекта вне зависимости от того, была гонка при чтении ссылки или нет.

**public** **class** **Foo** {

**private** **final** **int** a; /\* always visible \*/

**public** **Foo**() {

**this**.a = 5;

}

}

**public** **class** **Bar** {

**private** **final** **int** b; /\* always visible \*/

**public** **Foo**() {

**this**.b = 7;

}

}

**public** **class** **DataHolder** {

**private** **final** Foo foo; /\* always visible \*/

**private** **final** **int** c; /\* always visible \*/

**private** Bar bar; /\* may not be visible \*/

**private** **int** d; /\* may not be visible \*/

**public** **DataHolder**() {

**this**.foo = **new** Foo();

**this**.bar = **new** Bar();

**this**.c = 9;

**this**.d = 10;

/\* StoreStore \*/

}

}