***12. Каузальная модель памяти Java***

[***модель памяти Java***](https://ru.wikipedia.org/wiki/%D0%9C%D0%BE%D0%B4%D0%B5%D0%BB%D1%8C_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D0%B8_Java)

[***часы Лампорта***](http://neerc.ifmo.ru/wiki/index.php?title=%D0%9B%D0%BE%D0%B3%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B5_%D1%87%D0%B0%D1%81%D1%8B_%D0%9B%D0%B0%D0%BC%D0%BF%D0%BE%D1%80%D1%82%D0%B0)

[***JLS***](http://javacogito.net/index.php?title=%D0%A1%D0%BF%D0%B5%D1%86%D0%B8%D1%84%D0%B8%D0%BA%D0%B0%D1%86%D0%B8%D1%8F_%D0%B2%D0%B8%D1%80%D1%82%D1%83%D0%B0%D0%BB%D1%8C%D0%BD%D0%BE%D0%B9_%D0%BC%D0%B0%D1%88%D0%B8%D0%BD%D1%8B_Java#.D0.A1.D0.B8.D0.BD.D1.85.D1.80.D0.BE.D0.BD.D0.B8.D0.B7.D0.B0.D1.86.D0.B8.D1.8F) ***-***

* Физически результаты исполнения операций могут переупорядочиваться
  + Одновременным исполнением на разных процессорах
  + Суперскалярным процессором
  + Кешами памяти
  + Оптимизирующим компилятором
* Важен только порядок регистрации наблюдателем (см. логические часы Лампорта)
* На одной нити все операции с памятью выполняются в порядке исполнения.
* Для различных нитей все операции с памятью (кроме long и double) атомарные. ПОРЯДОК НЕ ОПРЕДЕЛЕН!
* До Java 1.5 упорядочивались только операции синхронизации
* С Java 1.5 каузальная модель памяти (JLS 3.0, 17.4)

*Синхронизованные операции*

Модификатор volatile накладывает некоторые дополнительные условия на чтение/запись переменной. Важно понять две вещи о volatile переменных:

1. Операции чтения/записи volatile переменной являются атомарными.
2. Результат операции записи значения в volatile переменную одним потоком, становится виден всем другим потокам, которые используют эту переменную для чтения из нее значения.
   * Volatile read/write
   * Monitor lock/unlock
   * Синтетические первая и последняя операции нити
   * Запуск исполнения и проверка завершения нити
   * Прерывание нити и обнаружение прерывания нити

* Существует строгий порядок всех синхронизованных операций в программе
  + согласован с порядком операций каждой нити

*Отношение синхронизации (synchronizes with)*

Quite often, though, a thread only modifies a shared variable when there are no concurrent readers or writers. In such cases, atomic operations are unnecessary. We just need a way to safely propagate modifications from one thread to another once they’re complete. That’s where the synchronizes-with relation comes in.

”Synchronizes-with” is a term invented by language designers to describe ways in which the memory effects of source-level operations – even non-atomic operations – are guaranteed to become visible to other threads. This is a desirable guarantee when writing lock-free code, since you can use it to avoid unwelcome surprises caused by memory reordering.

Отношение “синхронизации c”

* + M.unlock со всеми последующими M.lock
  + V.write со всеми последующими V.read
  + Запуск исполнения нити с первой операцией нити
  + Инициализация значения по умолчанию с первой операцией всех нитей
  + Последняя операция нити с обнаружением завершения нити (Thread.isAlive, Thread.Join)
  + Прерывание нити с обнаружением прерывания нити (InterruptedException, Thread.interrupted, Thread.isInterrupted)

*Отношение предшествования (happens before)*

* Пусть *x,* *y – операции.* Будем писать *hb*(*x*, *y*) для обозначения *x* предшествует *y.*
* *hb – частичный порядок операций*
* *hb*(*x*, *y*) если
  + *x,* *y* исполняются на одной нити и *x* встречается перед *y* в естественном порядке исполнения
  + x – последняя операция конструктора объекта, y – первая операция Object.finalize() того же объекта
  + *x* синхронизована с *y*
  + *hb*(*x*, z) и *hb*(z, y) – транзитивность
  + запись x предшествует чтению у по семантике *final* поля (JLS 3.0, 17.5)
* ВНИМАНИЕ! Если для z не определен порядок относительно x и y, то во время исполнения z порядок исполнения x и у неизвестен.

*Пример (double check)*

Правильно

class A {

volatile R r;

R getR() {

if (r == null) {

synchronized {

if (r == null){

r = new R();

}

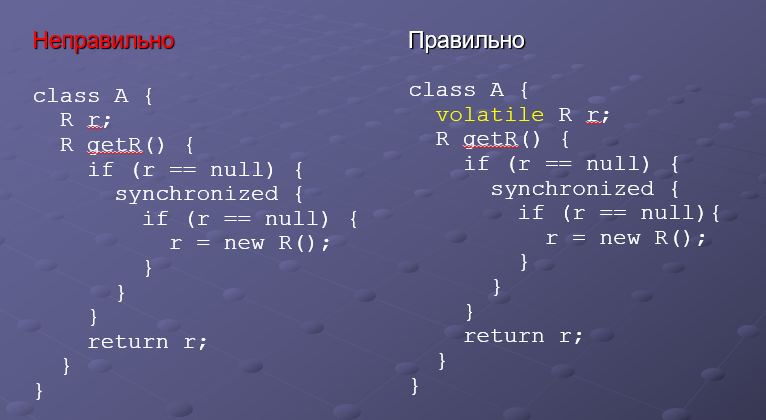
}

}

return r;

}

}



*Примеры упорядоченности*

