# Конспект лекции
# Модель памяти Java

## Цель и задачи лекции

Цель – изучить модель памяти Java.

Задачи:

1. Изучить особенности модели памяти Java
2. Понять порядок операций в многопоточном приложении

## План занятия

1. Особенности оригинальной модели памяти Java
2. Синхронизация потоков и видимость переменных
3. Порядок операций в многопоточном приложении
4. Отношение happens-before

## Модель памяти Java

Модель памяти Java описывает поведение потоков в среде исполнения Java. Модель памяти — часть семантики языка Java, и описывает, на что может и на что не должен рассчитывать программист, разрабатывающий ПО не для конкретной Java-машины, а для Java в целом.

Модель памяти описывает отношения между переменными в программе (полями экземпляров, статическими полями и элементами массива) и деталями низкого уровня их хранения и восстановления из памяти в реальной вычислительной системе. Объекты в конечном счете хранятся в памяти, но компилятор, среда выполнения, процессор или кэш могут обращаться довольно бесцеремонно с установкой временных характеристик для перемещаемых значений в заданные для переменных ячейки памяти или из них. Например, компилятор может выбрать оптимизацию переменной счетчика цикла, сохранив ее в регистре, или же кэш может отложить сброс нового значения переменной в основную память на более подходящее время. Все эти оптимизации направлены на повышение производительности и в общем прозрачны для пользователя, но в мультипроцессорных системах эти сложности могут иногда проявиться.

Модель памяти JMM позволяет компилятору и кэшу довольно свободно обращаться с порядком, в котором данные перемещаются между кэшем процессора (или регистром) и основной памятью, если только программист явно не попросит определенных гарантий видимости с помощью synchronized или volatile. Это значит, что при отсутствии синхронизации операции с памятью могут происходить в разном порядке с точки зрения различных потоков.

## Недостатки оригинальной модели памяти Java

В то время, как модель памяти JMM, заданная в главе 17 спецификации Java Language Specification, была амбициозной попыткой определить единообразную, межплатформенную модель памяти, у нее есть несколько незаметных, но существенных недостатков. Семантика synchronized и volatile была довольно такой запутанной, настолько, что многие хорошо подготовленные разработчики предпочитали иногда игнорировать правила, потому что написание соответствующим образом синхронизированного кода под старой моделью памяти было затруднительно.

Старая модель памяти JMM допускала некоторые удивительные и сбивающие с толку вещи, например появление в полях final не тех величин, которые были установлены в конструкторе (что могло превратить неизменяемые объекты в изменяемые), а также неожиданные результаты при переупорядочивании операций с памятью. Это также мешало некоторым формам оптимизаций программ во время компилирования, которые в иных случаях эффективны. Если вы прочли какие-либо статьи о проблеме блокировки с двойной проверкой (см. Ресурсы), вы помните, насколько запутанной может быть операция по переупорядочиванию, и насколько незаметные, но серьезные проблемы могут проникнуть в ваш код, когда вы не выполняете синхронизацию должным образом (или активно пытаетесь избежать синхронизации). Хуже того, многие неправильно синхронизированные программы оказываются работают правильно в некоторых ситуациях, например, при при небольшой загрузке, на однопроцессорных системах, или на процессорах с более сильными моделями памяти, чем это требуется JMM.

- Термин переупорядочивание используется для описания нескольких классов реальных и очевидных перераспределений операций с памятью:
- Компилятор может в качестве оптимизации свободно переупорядочивать определенные инструкции, если это не меняет семантику программы.
- Процессору позволяется исполнять операции не по порядку в некоторых обстоятельствах.

Кэшу, как правило, позволяется выполнять обратную запись переменных в основную память не в том порядке, в котором они были записаны программой.

Любые из этих условий могут привести к тому, что с точки зрения другого потока операции могут происходить не в том порядке, как это задано программой, и независимо от источника переупорядочивания, все считается моделью памяти эквивалентным.

## Синхронизация и видимость

Большинство программистов знают, что ключевое слово synchronized вызывает объект-мьютекс (взаимное исключение), которое мешает более чем одному потоку за раз входить в блок synchronized, защищенный данным монитором. Но синхронизация имеет еще один аспект: она устанавливает определенные правила видимости памяти, как указано в модели памяти Java. Это обеспечивает сброс кэшей при выхода из блока synchronized и объявление их недействительными при входе в него, таким образом, что значение, записанное одним потоком во время работы блока synchronized, может быть доступно любому другому потоку, выполняющему блок synchronized, защищенный тем же самым монитором. Это также ведет к тому, что компилятор не перемещает инструкции изнутри блока synchronized наружу (хотя он может в некоторых случаях перемещать

инструкции извне внутрь блока synchronized). Модель памяти Java этого не гарантирует в отсутствии синхронизации. Именно поэтому синхронизация (или ее младший собрат volatile) должны использоваться всякий раз, когда несколько потоков обращаются к одним и тем же переменным.

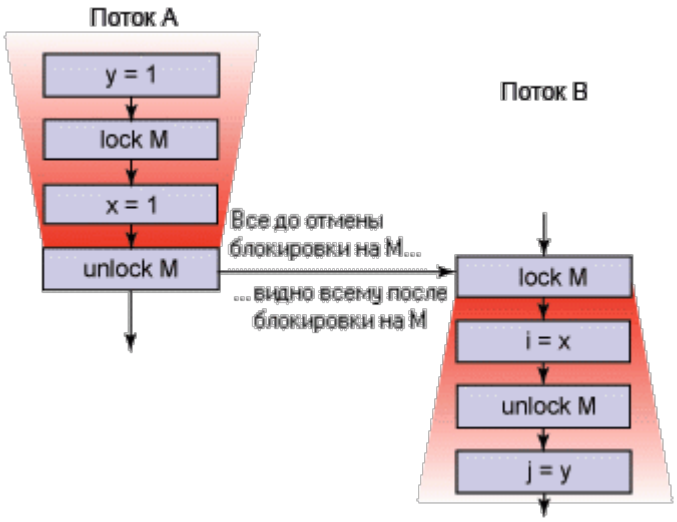
## Порядок операция

Действия, например чтение и запись переменных, распределяются по потоку в соответствии с тем, что называется "программным порядком", т. е. порядком, в котором они должны происходить в соответствии с семантикой программы. (Компилятор действительно может свободно и довольно бесцеремонно обращаться с программным порядком внутри потока, до тех пор, пока сохраняется квазипоследовательная семантика). Действия в других потоках совсем необязательно распределяются относительно друг друга - если вы начнете два потока и каждый из них выполните без синхронизации на любых общих мониторах или затрагивая любые общие переменные volatile, вы практически ничего не сможете предсказать об относительном порядке, в котором действия в одном потоке будут выполняться (или становиться видимыми третьему потоку) относительно действий в другом потоке.

Дополнительные гарантии переупорядочивания создаются, когда поток запускается, один поток присоединяется к другому, поток запрашивает или запускает монитор (входит или выходит из блока synchronized), или поток получает доступ к переменной volatile. Модель памяти Java описывает гарантии переупорядочивания, которые даются, когда программа использует синхронизацию или переменные volatile, чтобы координировать действия в многопоточной среде. Новая модель памяти JMM, если сказать проще, определяет порядок, который называется происходит-прежде (happens-before), который является частичным распределением всех действий внутри программы следующим образом:

- Каждое действие в потоке происходит-прежде каждого действия в этом потоке, которое идет позже в программном порядке
- Разблокировка монитора происходит-прежде каждой последующей блокировки того же монитора
- Запись в поле volatile происходит-прежде каждого последующего считывания того же самого volatile
- Вызов Thread.start() на поток происходит-прежде любых других действий в запущенном потоке
- Все действия в потоке происходят-прежде чем любой другой поток успешно возвращается из Thread.join() на этом потоке

Именно третье из этих правил, которое касается чтения и записи переменных volatile, является новым и решает проблему с примером. Так как запись volatile-переменной initialized происходит после инициализации configOptions, использование configOptions происходит после чтения initialized, а чтение initialized происходит после записи initialized, вы можете заключить, что инициализация configOptions потоком A происходит перед использованием configOptions потоком B. Поэтому configOptions и доступные через нее переменные будут видимы потоку B.




Использование синхронизации для гарантирования видимости записи в память по всем потокам.

## Atomicity

Хотя многие это знают, считаю необходимым напомнить, что на некоторых платформах некоторые операции записи могут оказаться неатомарными. То есть, пока идёт запись значения одним потоком, другой поток может увидеть какое-то промежуточное состояние. За примером далеко ходить не нужно — записи тех же long и double, если они не объявлены как volatile, не обязаны быть атомарными и на многих платформах записываются в две операции: старшие и младшие 32 бита отдельно.

## Visibility

В старой JMM у каждого из запущенных потоков был свой кеш (working memory), в котором хранились некоторые состояния объектов, которыми этот поток манипулировал. При некоторых условиях кеш синхронизировался с основной памятью (main memory), но тем не менее существенную часть времени значения в основной памяти и в кеше могли расходиться.

В новой модели памяти от такой концепции отказались, потому что то, где именно хранится значение, вообще никому не интересно. Важно лишь то, при каких условиях один поток видит изменения, выполненные другим потоком. Кроме того, железо и без того достаточно умно, чтобы что-то кешировать, складывать в регистры и вытворять прочие операции.

Важно отметить, что, в отличие от того же C++, «из воздуха» (out-of-thin-air) значения никогда не берутся: для любой переменной справедливо, что значение, наблюдаемое потоком, либо было ранее ей присвоено, либо является значением по умолчанию.

## Reordering

Также, инструкции в потоке могут быть переставлены местами, если не нарушают программный порядок. Одним из примечательных эффектов может оказаться то, что

действия, выполненные одним потоком, другой поток увидит в другом порядке. Эту фразу довольно сложно понять, просто прочитав, потому рассмотрим пример.

Пусть есть такой код:

```
public class ReorderingSample {
    boolean first = false;
    boolean second = false;
    void setValues() {
        first = true;
        second = true;
    }
    void checkValues() {
        while(!second);
        assert first;
    }
}
```

И в этом коде из одного потока вызывается метод checkValues, а из другого потока — setValues. Казалось бы, код должен выполняться без проблем, ведь полю second значение true присваивается позже, чем полю first, и потому когда (точнее, если) мы видим, что, второе поле истинно, то и первое тоже должно быть таким.

Хотя внутри одного потока об этом можно не беспокоиться, в многопоточной среде результаты операций, произведённых другими потоками, могут наблюдаться не в том порядке. Пусть класс Data в конструкторе выполняет какие-то не очень тривиальные вычисления и, главное, записывает какие-то значения в не final поля:

```
public class Data {
    String question;
    int answer;
    int maxAllowedValue;
    public Data() {
        this.answer = 42;
        this.question = reverseEngineer(this.answer);
        this.maxAllowedValue = 9000;
    }
}
```

Получится, что тот поток, который первый обнаружит, что data == null, выполнит следующие действия:

1. Выделит память под новый объект
2. Вызовет конструктор класса Data
3. Запишет значение 42 в поле answer класса Data
4. Запишет какую-то строку в поле question класса Data
5. Запишет значение 9000 в поле maxAllowedValue класса Data
6. Запишет только что созданный объект в поле data класса Keeper

Ничто не мешает другому потоку увидеть произошедшее в пункте 6 до того, как он увидит произошедшее в пунктах 3-5. В результате этот поток увидит объект в некорректном состоянии, когда его поля ещё не были установлены. Такое, разумеется, никого не устроит, и потому есть жёсткий набор правил, по которым оптимизатору/компилятору запрещено выполнять reordering.

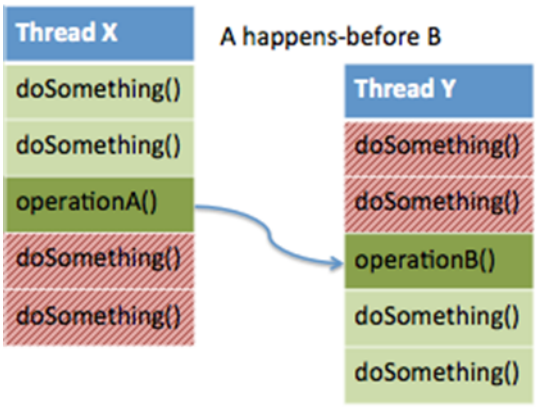
## Отношение happens-before

Все эти правила заданы с помощью так называемого отношения happens-before:

*Пусть есть поток X и поток Y (не обязательно отличающийся от потока X). И пусть есть операции A (выполняющаяся в потоке X) и B (выполняющаяся в потоке Y).*

В таком случае, A happens-before B означает, что все изменения, выполненные потоком X до момента операции A и изменения, которые повлекла эта операция, видны потоку Y в момент выполнения операции B и после выполнения этой операции.

Рассмотрим простейший случая, когда поток только один, то есть X и Y — одно и то же. Внутри одного потока, как мы уже говорили, никаких проблем нет, потому операции имеют по отношению к друг другу happens-before в соответствии с тем порядком, в котором они указаны в исходном коде (program order).



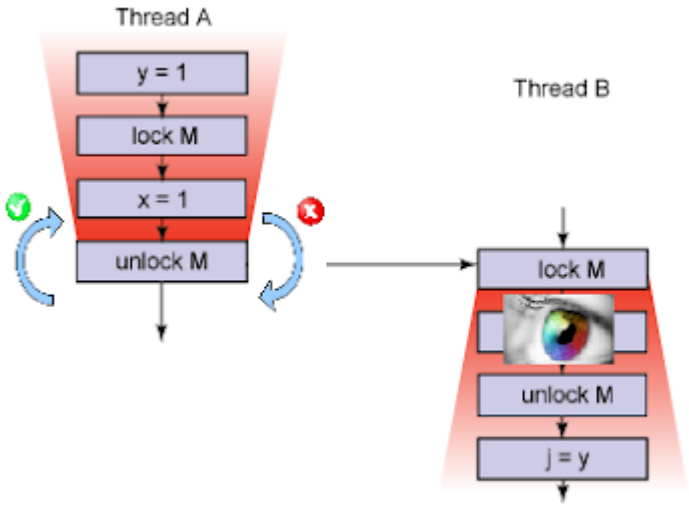

Здесь слева зелёным помечены те операции, которые гарантированно увидит поток Y, а красным — те, что может и не увидеть. Справа красным помечены те операции, при исполнении которых ещё могут быть не видны результаты выполнения зелёных операций слева, а зелёным — те, при исполнении которых уже всё будет видно. Важно заметить, что отношение happens-before транзитивно, то есть если A happens-before B и B happens-before C, то A happens-before C.

В момент отпускания монитора (записи в volatile переменную и дальше по списку) все регистры и локальные кэши процессора синхронизируются с основной памятью, а в момент последующего захвата лока (чтения volatile переменной и т.д.) процессор на котором выполняется второй поток инвалидирует свой кэш и зачитывает все последние данные из основной памяти. Почему же тогда надо обязательно синхронизироваться на один и тот же монитор? Да потому, что только в этом случае будет гарантироваться строгий порядок, т.е. второй поток гарантировано сбросит свой кэш, только после того как первый синхронизирует свой с основной памятью.

Отношение happens-before так же накладывает сильные ограничения на reordering. С точки зрения потока Y все операции произошедшие до точки happens-before в потоке X он может рассматривать как операции свершившиеся в своем собственном потоке. Т.е. никакого логического reordering по сравнению с прямым порядком в исходном коде с точки зрения потока Y быть не может.

Если взглянуть внимательнее на границу happens-before с точки зрения reordering для потока Y, то никакие операции располагающиеся выше границы happens-before в потоке X, не могут выполнится ниже границы happens-before в результате reordering, однако, операциям, находящимся ниже границы, разрешено выполнение до неё. Более

наглядно это изображено на рисунке.




## Операции, связанные отношением happens-before

Два действия могут быть упорядочены с помощью отношения произошло до. Если одно действие произошло до другого, то первое видимо вторым и при упорядочении находится перед ним.

Если у нас имеется два действия, x и y, мы записываем hb(x,y), чтобы указать, что x произошло до (happens-before) y.

- Если x и y представляет собой действия одного и того же потока и x находится до y

  в программном порядке, то hb(x,y).

- Имеется ребро произошло до от конца конструктора объекта до начала финализатора

  этого объекта.

- Если действие д; синхронизировано со следующим за ним действием y, то мы также

  имеем hb(x,y).

- Если hb(x,y) и hb(y,z), то hb(x,z).

Методы wait класса Object должны блокировать и разблокировать действия, связанные с ними; их отношения произошло до определяются связанными с ними действиями.

Следует заметить, что наличие отношения happens-before между двумя действиями не обязательно подразумевает, что они имеют место в данном порядке в реализации. Если переупорядочение дает результаты, согласующиеся с корректным выполнением, такое переупорядочение не является некорректным.

**Освобождение (releasing) монитора** happens-before заполучение (acquiring) того же самого монитора. Обратите внимание: именно освобождение, а не выход, то есть за безопасность при использовании wait можно не беспокоиться.

Как исправить данный пример? В данном случае всё очень просто: достаточно убрать внешнюю проверку и оставить синхронизацию как есть. Теперь второй поток гарантированно увидит все изменения, потому что он получит монитор только после того, как другой поток его отпустит. А так как он его не отпустит, пока всё не проинициализирует, мы увидем все изменения сразу, а не по отдельности:

```
public class Keeper {
    private Data data = null;
    public Data getData() {
        synchronized(this) {
            if(data == null) {
                data = new Data();
            }
        }
        return data;
    }
}
```

**Запись в volatile переменную** happens-before чтение из той же самой переменной.

Изменение исправляет некорректность, но возвращает того, кто написал изначальный код, туда, откуда он пришёл — к блокировке каждый раз. Спасти может ключевое слово volatile. Фактически, рассматриваемое утверждение (2) значит, что при чтении всего, что объявлено volatile, мы всегда будем получать актуальное значение. Кроме того, для volatile полей запись всегда (в т.ч. long и double) является атомарной операцией. Ещё один важный момент: если у вас есть volatile сущность, имеющая ссылки на другие сущности (например, массив, List или какой-нибудь ещё класс), то всегда «свежей» будет только ссылка на саму сущность, но не на всё, в неё входящее.

С использованием volatile исправить ситуацию можно так:

```
public class Keeper {
    private volatile Data data = null;
    public Data getData() {
        if(data == null) {
            synchronized(this) {
                if(data == null) {
                    data = new Data();
                }
            }
        }
        return data;
    }
}
```

Здесь по-прежнему есть блокировка, но только в случае, если data == null. Остальные случаи мы отсеиваем, используя volatile read. Корректность обеспечивается тем, что volatile store happens-beforevolatile read, и все операции, которые происходят в конструкторе, видны тому, кто читает значение поля.

**Запись значения в final-поле** (и, если это поле — ссылка, то ещё и всех переменных, достижимых из этого поля (dereference-chain)) при конструировании объекта happens-before запись этого объекта в какую-либо переменную, происходящая вне этого конструктора.

Это тоже выглядит довольно запутанно, но на самом деле суть проста: если есть объект, у которого есть final-поле, то этот объект можно будет использовать только после установки этого final-поля (и всего, на что это поле может ссылаться). Не стоит, впрочем, забывать, что если вы передадите из конструктора ссылку на конструируемый объект (т.е. this) наружу, то кто-то может увидеть ваш объект в недостроенном состоянии.

В примере достаточно сделать поле, запись в которое происходит последней, final, как всё магически заработает и без volatile и без синхронизации каждый раз:

```
public class Data {
    String question;
    int answer;
    final int maxAllowedValue;
    public Data() {
        this.answer = 42;
        this.question = reverseEngineer(this.answer);
        this.maxAllowedValue = 9000;
    }
    private String reverseEngineer(int answer) {
        return null;
    }
}
```

Кроме того, важно помнить, что поля бывают ещё и статические, а что инициализацию классов JVM гарантированно выполняет лишь один раз при первом обращении.

```
public class Singleton {
    private Singleton() {}
    private static class InstanceContainer {
        private static final Singleton instance = new Singleton();
    }
    public Singleton getInstance() {
        return InstanceContainer.instance;
    }
}
```

Из приведенных выше определений вытекает следующее.

- Разблокировка монитора происходит до каждой последующей блокировки монитора.
- Запись volatile-поля происходит до каждого последующего чтения этого поля.
- Вызов start() потока происходит до любого действия в запущенном потоке.
- Все действия в потоке происходят до того, как любой другой поток успешно вернется из вызова join() этого поток
- Инициализация по умолчанию любого объекта происходит до любых других действий (отличных от записи значений по умолчанию) программы.

## Литература и ссылки


1. Спецификация языка Java https://docs.oracle.com/javase/specs/jls/se8/html/index.html

2. Шилдт Г. Java 8. Полное руководство. 9-е издание. — М.: Вильямс, 2012. — 1377 с.
3. Эккель Б. Философия Java. 4-е полное изд. — СПб.: Питер, 2015. — 1168 с.
4. https://www.ibm.com/developerworks/ru/library/j-5things5/index.html
5. https://www.ibm.com/developerworks/ru/library/j-jtp10264/index.html
6. https://www.ibm.com/developerworks/ru/library/j-jtp07233/index.html
7. https://www.ibm.com/developerworks/ru/library/j-jtp02244/
8. https://www.ibm.com/developerworks/ru/library/j-jtp03304/index.html

## Вопросы для самоконтроля

1. В чем назначение модели памяти в Java?
2. Какие особенности у отношения happens-before?
3. Назовите основные свойства модели памяти и в чем их смысл.