Федеральное государственное автономное учреждение
высшего образования

Московский физико-технический институт
(национальный исследовательский университет)

J A V A

III СЕМЕСТР
Физтех-школа: *ФПМИ*
Направление: *СберТех*
Лектор: *Александр Маторин*

Автор: *Савелий Романов*

Долгопрудный, Осень 2022 год.

Содержание

1 Структуры данных

1.1 equals и hashCode

У Object есть такие методы equals и hashcode.

1.1.1 equals

public boolean equals(@Nullable Object obj)

Функция проверяет, равны ли два обьекта логически. По умолчанию просто сравнивают ссылки, но можно переопределить.

Образует отношение на ненулевых обьектах, то есть рефлексивное, симметричное и транзитивное отношение. При этом еще есть требование консистентности: несколько вызовов equals должны возвращать одно и то же, если у обьектов не модифицировались данные на которых работает equals.

Для ненулевого обьекта x выражение x.equals(null) должно быть false.

Если переобределяете equals, надо переопределить и hashCode

1.1.2 hashCode

public native int hashCode();

Функция нужна например для хеш таблиц, таких как HashTable.

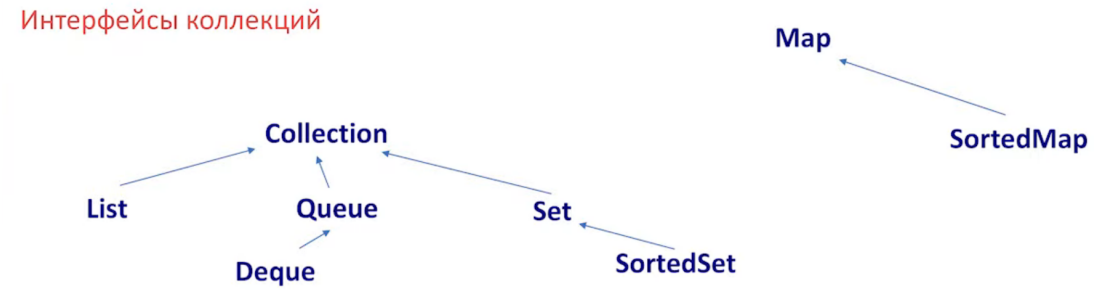
Контракт:

- Должна возвращать одно и то же значение если вызывать ее несколько раз на обьекте и при этом никакие данные на которых основывается equals не поменялись.
- Если equals говорит что обьекты одинаковые, то hashCode для них должен совпадать.

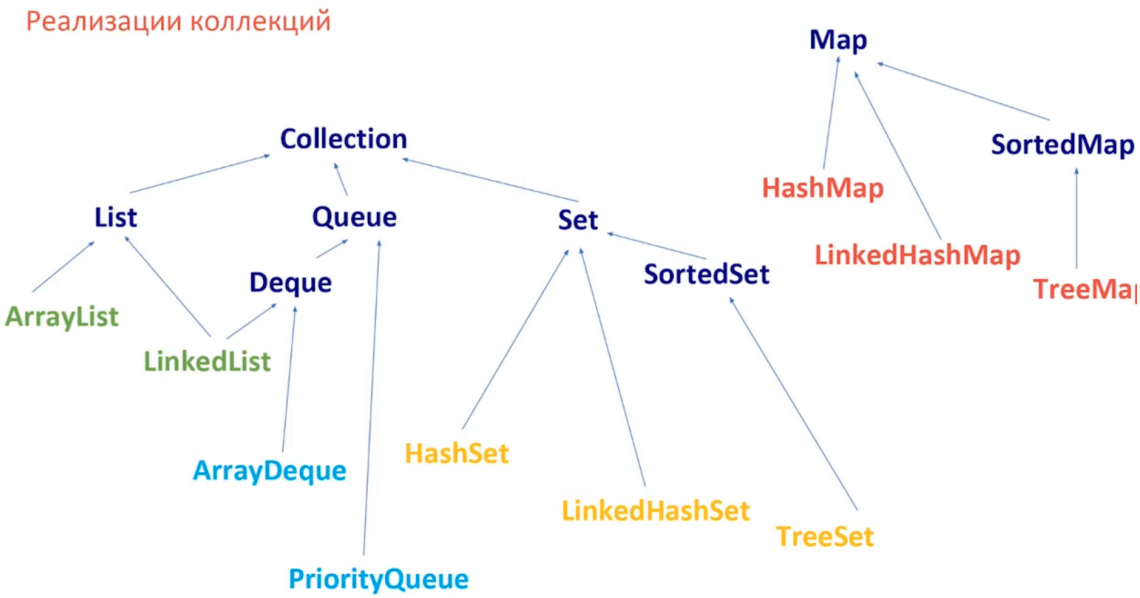
При этом если equals вернул false, то hashCode могут совпадать, это коллизия хешей.

Дефолтная реализация: раньше адрес в памяти, сейчас псевдослучайное число. Ее поменяли, потому что при небольшом размере heap-а адреса обьектов были близки друг к другу, а хотелось бы иметь равномерно распределение hashCode-ов. У каждого обьекта hashCode записывается в его заголовок. Если обьект не меняется, то и его hashCode не меняется.

1.2 интерфейсы коллекций



1.3 реализации коллекций



1.4 ArrayList

ArrayList

```
class ArrayList {
   Object[] elementData;
   int size;
}
```

```
List<String> list = new ArrayList<>(); // создается массив длины 0
```

Расширяется в 1.5 раза если capacity заполнилась полностью и места для добавления элемента больше нет.

Если массив уменьшается, capacity не уменьшается. Надо сделать trimToSize, если хочется ее уменьшить.

1.4.1 сложность

ArrayList – алгоритмическая сложность

```
list.get(i); //O(1)
list.add(value); //O(1) – амортизированная, O(n) - худшая
list.add(i, value); //O(n)
list.remove(i); //O(n)
list.contains(v); //O(n)
```

1.5 LinkedList

Двусвязный список. Прыгает по памяти, поэтому работает медленно.

1.6 Queue/Deque

Интерфейс добавляет методы для работы с началом и концом массива на O(1).

Реализация ArrayDeque — циклический буффер. Если заполнился, растим размер по формуле new-size = 2 * old-size + 2.

1.7 HashMap

HashMap

```
class HashMap {
   Node<K,V>[] table;
   int size;
}
```

```
static class Node<K,V> implements Map.Entry<K,V> {
   final int hash;
   final K key;
   V value;
   Node<K,V> next;
}
```

Используется цепочечная адресация. Сначала длина массива null, потом сразу 16. load-factor = 0.75, расширение в 2 раза.

Используем hashCode при вычислении хеша. Проверяем что нашли нужны ключ через

equals.

1.8 TreeMap

Используется сбалансированное двоичное дерево, красно-черное.

1.9 HashSet

HashSet просто используется HashMap.

1.10 TreeSet

TreeSet использует TreeMap.

1.11 LinkedHashMap

Тоже самое что и HashMap, только итерация по ключам происходит не в произвольном порядке, а в порядке их добавления. Появляется небольшое overhead на добавление в список, а остальные алгоритмы те же самые.

2 Generics

2.1 введение

Дженерики были созданы для того, чтобы избежать ситуации, когда код компилируется, а в рантайме падает в Exception. Не хочется в рантайме кастовать типы.

2.2 extends

List<? extends Number> это коллекция работающая только на чтения. Единственное что можно добавить это null. Поэтому нельзя сделать с ней elements.add(1). Когда мы оставляем только чтения, не может возникнуть исключение. Тут ? extends Number это либо Number, либо какое-то его подтип.

2.3 super

? super E — это какое-то класс, являющийся предкам класса E. В ? super E можно добавить любой подкласс E, то есть ? extends E.

Можно задавать ограничения, например, E extends Comparable.

2.4 producer extends, consumer super (PECS)

Думать в всех corner case-ах в дженериках каждый раз очень тяжело, поэтому придумали правило которое помогает понять как расставить extends и super.

Producer — то кто использует дженерики как типы возвращаемого значения.

Consumer — где дженерики используются как аргументы методов.

Для producer надо использовать ? extends E, для consumer ? super E.

Object & ...- хак для совместимости со старой Java.

3 Lamda. Stream API

3.1 декларативный vs императивный стиль

Декларативный стиль программирования обьясняет что мы хотим достичь, но не говорит как конкретно этого достичь.

В императивном стиле программирования мы перечисляем конкретные шаги до достижению цели.

```java
public List<Child> collectChildren(List<Parent> parents) {
    Set<Child> result = new LinkedHashSet<>();
    for (Parent parent : parents) {
        Child child = parent.getChild();
        if (child.getAge() >= 18) {
            result.add(child);
        }
    }

    return new ArrayList<>(result);
}
```

Java7

```java
public List<Child> collectChildren(List<Parent> parents) {
    return parents.stream()
            .map(Parent::getChild)
            .filter(child -> child.getAge() >= 18)
            .distinct()
            .collect(toList());
}
```

Java8

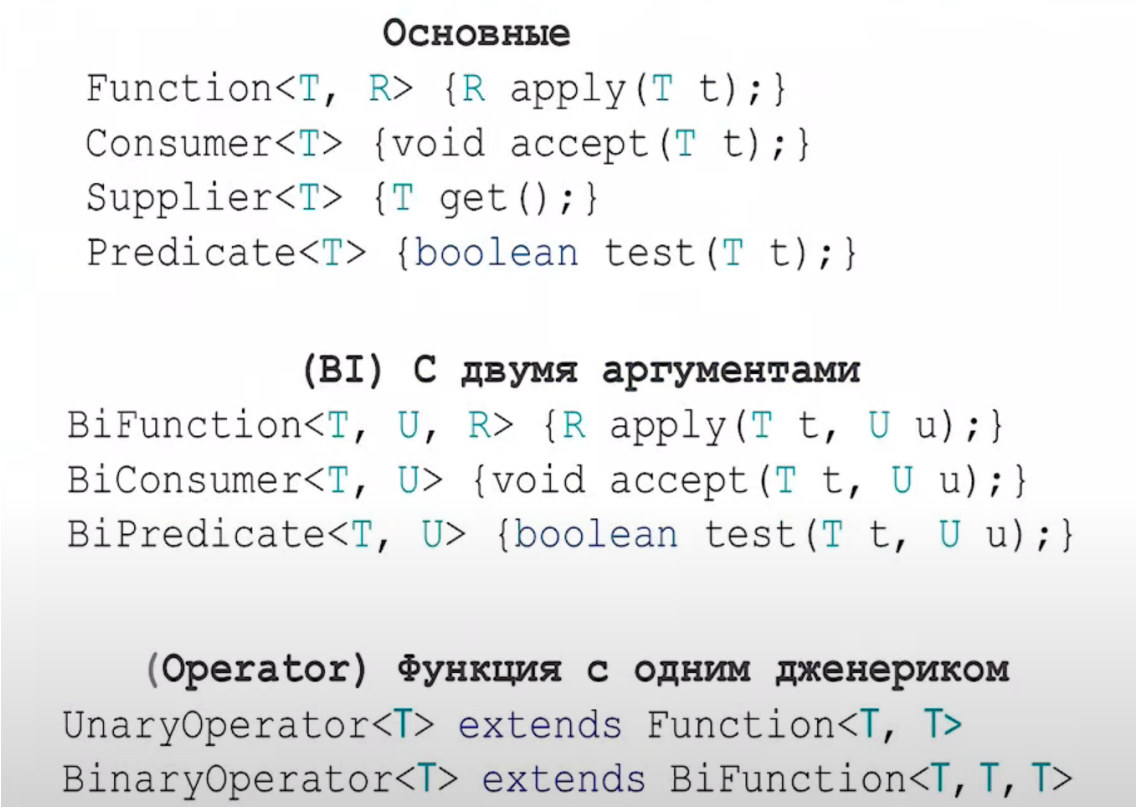
3.2 Lambda

Lambda можно использовать для реализации интерфейса с единственным методом. Это просто сокращенная запись.

```java
Comparator<Integer> comparator = (o1, o2) -> Integer.compare(o1, o2);
```

Захватывать можно только final или effetively final переменные. На захват полей нет такого ограничения.

3.3 java.util.function



3.4 функции первого класса

Язык программирования имеет **функции первого класса**, если он рассматривает функции как **обьекты первого класса** (поддерживает передачу функций в качестве аргументов другим функциям, возврат их как результат других функций, присваивание их переменным).

3.5 функции высшего порядка

Функция, принимающая в качестве аргументов другие функции и возвращающая другую функцию в качестве результата.

3.6 частичное применение функции

Процесс фиксации части аргументов функции, который создает функцию меньшей арности.

3.7 каррирование

Преобразование функции от многих аргументов в функцию, берущую свои аргументы по одному.

Преобразование функции от многих аргументов в функцию, берущую свои аргументы по одному

```
public int sum(int x, int y, int z) {
    return x + y + z;
}

public IntFunction<IntFunction<IntUnaryOperator>> currySum() {
    return x -> y -> z -> sum(x, y, z);
}
```

```
int sum = currySum().apply(1).apply(3).applyAsInt(2);
```

graphicx

4 Reflection

Получение информации о классе в рантайме

4.1 Class<?>

Как получить?
Class<Integer> c = Integer.class;
someObject.getClass();

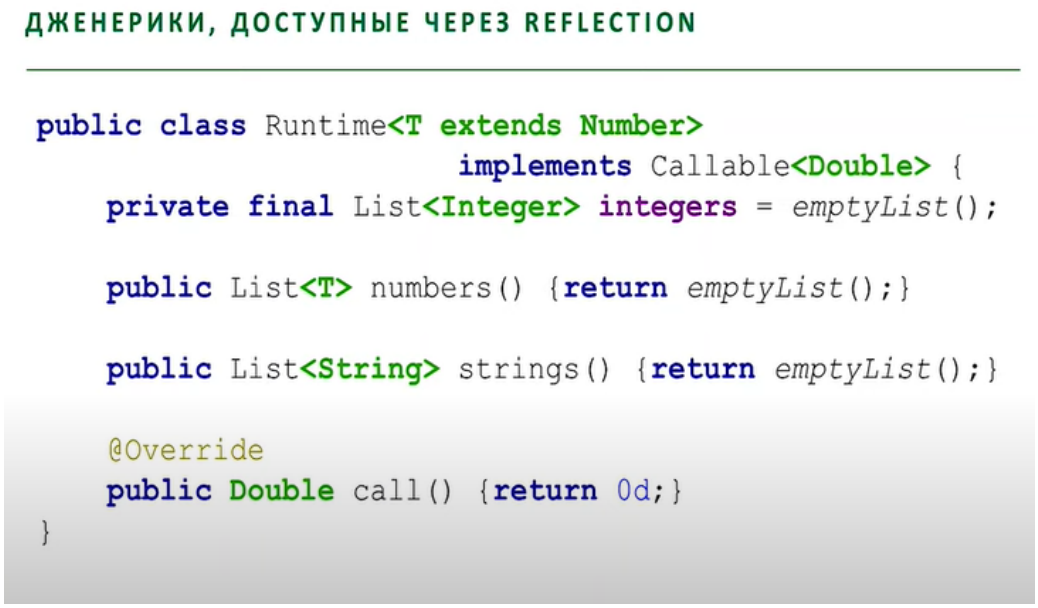
4.2 Методы класса Class

public Method[] getMethods(); - список public методов
public Method[] getDeclaredMethods(); - список методов самого класса
public Method getMethod(String name, Class<?> ...parametrTypes);
public Field[] getFields();
public Field[] getDeclaredFields();
public Field getField(String name);
public Field getDeclaredField(String name);

public native Class<? super T> getSuperclass() (Для Object вернет null)
Методы можно исполнять. Method m = ...; m.invoke(Object o, Args...)
Можно выполнять приватные методы, нужно поставить флаг m.setAccesible(true

4.3 Дженерики через Рефлекшн

Можно достать информацию о дженериках на уровне класса
Информация, чем параметризованы локальные объекты стирается
Пример



5 Аннотации

Добавление метаинформации в класс
Пример

```
@Deprecated
public int getHours() {
    return normalize().getHours();
}
```

Объявлять свои аннотации можно так
@Target(ElementType.Type - Enum обозначающий на что вешается аннотация)
@Retention(RetentionPolicy.Runtime)
public @interface Name {

...
(примитивный тип, String, Class, enum) name();
}

5.1 ElementType

Это Enum {TYPE, FIELD, METHOD, CONSTRUCTOR, LOCAL_VARIABLE, ANNOTATION_TYPE, PACKAGE, TYPE_PARAMETER, TYPE_USE, MODULE, RECORD_COMPONENT, PARAMETER}

5.2 Retention

Это Enum {Source, Class, Runtime}.
Source - информация на уровне исходного кода, её нет в конечном коде для java.
Class - аннотация компилируется, попадает в байт код, но в рантайме её нет
Runtime - В рантайме с помощью reflection можно получить к этой информации доступ
Пример
public class Person {
@validLenght(min = 4, max = 10)
private final String name;
}

5.3 Получение аннотации

У типа Field и Method есть методы isAnnotationField(Class<T> annotationClass) и getAnnotationField(Class<T> annotationClass)

6 Dynamic Proxy

Позволяет в рантайме перехватывать вызовы интерфейса и обрабатывать их. Прокси может быть любым интерфейсом. Синтаксис такой

```java
public class CacheHandler implements InvocationHandler {
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        return null;
    }
}
```

7 ClassLoader

7.1 Загрузка класса

JVM загружает класс при первом обращении к нему. Обращение к классу — создание обькта класса, обращение к статик методу или полю.

7.2 Где хранится

Загруженные классы хранятся в области памяти PermGen. Это специальная область для хранения классов.

7.3 ClassLoader

ClassLoader-ы это специальные классы, которые загружают другие классы. Они отличаются логикой поиска классов. Можно загрузить классы из базы данных, файла или другого места.

7.4 Стандартные ClassLoader-ы

- Bootstrap
- Extensions
- System

7.5 Bootstrap ClassLoader

Загружает классы JDK (String, Integer, List, ...). Является частью JVM. Единственный ClassLoader, написанный не на Java, а на Си.

7.6 Extensions ClassLoader

Почти нигде не используется.

7.7 System/Application ClassLoader

Загружает классы из classpath.

7.8 URL ClassLoader

Загружает файлы по сети или из файловой системы.

7.9 Методы ClassLoader

public Class<?> loadClass(String fullName) загружает класс по имени

protected Class<?> defineClass(byte[] b, ...) загружает класс по байтовому представлению. не надо писать логику формирования класса самому, достаточно получить байты откуда-то.

public ClassLoader getParent() получить ссылку на родительский ClassLoader.

7.10 выбор ClassLoader

Класс загружается тем ClassLoader, который используется для класса, внутри метода которого идет обращение. new Person() равносильно Main.class.getClassLoader().loadClass("Person")

7.11 Родители

Родитель System это Extensions. Родитель Extensions это Bootstrap. У Bootstrap нет родителя.

7.12 Модель делегирования

Каждый ClassLoader проверяет, не загружал ли он этот класс ранее. Если не загружал, он не делает это сам, а делегирует родителю.

Если не получается загрузить класс, ClassLoader делегирует загрузку своему потомку. Если никто не справляется загрузить, кидается ClassNotFoundException.

Initiating ClassLoader — тот кто начал загружать класс. Defining ClassLoader — тот кто реально загрузил.

Уникальность класса определяется парой из имени класса и его обьект Defining ClassLoader.

Если нет доступных ссылок на класс, на обьекты этого класса и нет ссылок на классслоадер загрузивший этот класс, то он может быть выгружен из памяти сборщиком мусора.

7.13 исключения

ClassNotFoundException — когда класса с таким именем нет. NoClassDefFoundError — когда класс был на этапе компиляции, но в рантайме не был найден. Например, его могли удалить из classpath. NoSuchMethodError — на этапе компиляции и в рантайме присутствовали разные версии класса. IllegalAccessError — загружена другая версия класса, у которой изменили область видимости. ClassCastException — попытка присвоения, когда классы загружены разными ClassLoader-ами.

7.14 иерархия

При загрузке класса происходит загрузка всех его суперклассов и интерфейсов.

Класс и его зависимости могут быть загружены разными классслоадерами. Если такое происходит, то не получится скастовать к родителю, потому что они загружены раздыми классслоадерами.

Может быть полезно написать свой ClassLoader, например если хочется использовать свою собственную модель делегации вместо стандартной.

8 Многопоточность и синхронизация

8.0.1 Thread

Заводим через new Thread, передаем туда Runnable. Запускаем через thread.start(), ждем завершения с помощью thread.join(). thread.stop() лучше не использовать, потому что может оставить данные в неконсистентном состоянии.

8.1 synchronized

Базово: ключевое слово, которое позволяет синхронизировать программу, не позволяя коду исполняться одновременно из нескольких потоков.

Если несколько потоков ждут на synchronized, то первым выполнится случайный из них, тут нет честности. Скорее наоборот, работает специально нечестно: если новый поток пришел и synchronized освободился, запустится новый. Это лучше с точки зрения эффективности, потому что у нового потока данные в кеше.

У каждого обьекта есть монитор. Монитор — это условно bool, который присущ каждому обьекту. При входе в synchronized секцию монитор обьекта захватывается, при выходе – отпускается.

Бывает три типа synchronized:

- на обычном методе
- на статическом методе
- отдельная секция

Если synchronized стоит на обычном методе, это равносильно synchronized (this) { ... }.

synchronized на статическом методе захватывает монитор класса этого метода. Например, synchronized (Main.class) { ... }.

У synchronized есть дополнительная семантика: все изменения под данным монитором будут видны другим потоком под тем же монитором. Это важно, потому что по умолчанию спецификация Java на гарантирует, что потом увидит изменения переменной, которые были сделаны в другом потоке.

8.2 volatile

Можно повесить на поле класса, это будет означать, что все потоки должны видеть самое актуальное значение этой переменной, если другие потоки его меняли. volatile работает чуть быстрее чем synchronized.

8.3 wait и notify

Чтобы не жечь CPU (не нагружать его на полную) когда надо просто дождаться.

Бывают concurrent реализации стандартных коллекций в пакете syncronized, которые позволяет работать с ними из нескольких потоков одновременно без дополнительной синхронизации.

У Object есть метод wait. Надо чтобы монитор обьекта на котором мы вызываем wait был захвачен. Поэтому надо делать wait в блоке synchronized.

Метод notify позволяет разбудить случайный поток, который ждет на мониторе. Метод notifyAll будет всех кто ждет.

Поток может быть пробужден в wait случайно (spurious wakeup) даже если его никто не разбудил, этот кейс надо правильно обрабатывать. wait всегда надо вызывать в цикле while, проверяя что нужный нам предикат выполнен.

8.4 lock

Lock поддерживает несколько условий для пробуждения одновременно. Так можно разделять кого вы хотите разбудить и на каком предикате вы ждете, чтобы не все работали с одним предикатом. Подробнее тут тут и тут.

8.5 semaphore

Semaphore позволяет нам сделать чтобы в потоке одновременно находилось не более n потоков. Это n мы сами задаем в конкрукторе семифора.

Полезно использовать try finally чтобы сделать unlock даже если в критической секции вылетело исключение.

9 Java memory model

Модель памяти отвечает за то будет ли один поток видеть изменения, сделанные в другом потоке.

9.1 reordering

reordering — перестановка инструкций компилятором, если это не меняет семантику в рамках одного потока.

9.2 happens-before

JMM оперирует терминами actions (read, write, lock, unlock).

JMM гарантирует что поток выполнивший action B увидит результат action A только если A happens-before B (A и B состоят в отношении happens-before). Это отношение транзитивно.

Отношение happens-before строится следующим образом:

- В рамках одного потока действия упорядочены с помощью happens-before.
- Разблокировка монитора happens-before каждый последующей блокировки того же самого монитора.
- Запись в volatile happens-before каждого последующего считывания того же самого volatile.
- Вызов Thread::start() happens-before любого действия в этом потоке
- Завершение потока T1 happens-before момента когда поток T2 определил что T1 завершился, вызвав T1.join().
- Вызов конструктора обьекта happens-before запуска финализации для него

9.3 reordering synchronized

Инструкции не могут выноситься наружу synchronized секции, иначе нарушится семантика synchronized. Однако могут залезать в synchronized снаружи (и сверху и снизу).

9.4 reordering volatile

9.4.1 volatile store

Инструкции могут переупорядочиваться так, что те что были после store окажутся раньше.

9.4.2 volatile read

Инструкции могут переупорядочиваться так, что те что были до read окажутся позже.

9.5 volatile

Гарантии volatile:

- Гарантии happens-before
- Гарантии с reordering
- Атомарные запись и чтение актуальных значений (даже для long и double).

При этим атомарность инкремента volatile не гарантирует. Это 3 действия.

9.6 safe publication idioms

p = new Point() ссылка из конструктора может присвоиться раньше, чем произойдут все инициализации внутри конструктора.

Возможные решения:

- инициализация в static инициализаторе
- сохранять ссылку на обьект в volatile
- сохранять ссылку в final

final поля инициализируются до возврата ссылки на обьект из конструктора. При этом гарантии распространяются также на случай, когда мы поле final, но его состояние не final, в контрукторе все еще это поле валидно проинициализируется.

final volatile не бывает.

10 Just-in-time компиляция (JIT)

Код на Java компилируется сначала в байткод, затем виртуальная машина Java исполняет этот байткод интерпретатором. Куски байткода компилируется в машинный код прямо во время исполнения, это и есть JIT компиляция.

Java компилирует код разными способами на основе собранной во время исполнения статистики, сохраняет скомпиленный в код кеше. Код кеш – специальная область памяти с собранным кодом размером примерно 250Мб.

10.1 inline

inlining — встраивание кода вызываемой функции в вызывающую, оптимизацию которую может делать JIT. Позволяет не делать лишние переходы.

10.2 dead code elimination

Если код не используется, уберем его.

10.3 замер времени работы

Надо хорошо думать когда пишете бенчмарки, чтобы узнать время работы куска кода. Учитывайте работу JIT, ClassLoader-ы. Для того чтобы делать точные замеры времени вам поможет библиотека JMH.

10.4 loop unrolling

Оптимизация развертывания циклов, когда вы вместо исполнения цикла просто несколько раз копируете код который был внутри цикла.

10.5 escape analysis

Если Java видит что обьект никуда не выходит за пределы метода, существует в рамках только этого метода, то Java не создает этот обьект. Вместо этого она инлайнит все такие обьекты. Поэтому не надо бояться создавать много обьектов, это не повлечет замедления работы программы.

10.6 lock coursing

Укрупнение блока синхронизации.

10.7 lock elimination

Для локальной переменной можно снять синхронизацию, потому что другой поток не сможет с ней работать.

10.8 biased lock

Если по статистике видно что лок захватывается всегда только одним обьектом, можно убрать тяжеловесную проверку для синхронизации, просто записать в хедере обьекта какой поток им владеет и в вместо синхронизации проверять что номер потока не поменялся.

10.9 inline interface

Если функция принимает на вход интерфейс, а не конкретную реализацию, то проблематично ее инлайнить. Поэтому Java смотрит на иерархию загруженных классов.

10.10 stacktrace

JIT помнит как изначально выглядел ваш код, чтобы после всех оптимизаций помнить как должен выглядеть stacktrace для оригинального кода. Поэтому бросание Exception работает не очень хорошо.

10.11 intrinsic

Заранее написанный на другом языке кусок кода. С помощью @HotSpotIntrinsicCandidate можно подменить реализацию на Java реализацией на другом языке. Такой код может работать быстрее.

10.12 Java Object Layout

Утилита JOL (Java Object Layout) позволяет посмотреть представление класса в памяти.

У каждого обьекта есть заголовок, где хранится:

- ссылка на Class
- native HashCode
- lock (в случае biased lock хранится номер потока который обычно захватывает)

Все поля обьектов в Java выровнены на 8 байт. То есть если поле занимает меньше 8 байт, оно все еще на деле занимает хотя бы 8 байт =)

10.13 compressed ops

Раз все обьекты и поля выровнены по 8 байт, то все указатели в двоичном представлении заканчиваются на три нуля. Поэтому указатели занимают 32 бита, а на деле адресуем памяти как с 35 битами. Поэтому при увеличении размера heap-а может потребоваться сильно больше памяти, это надо учитывать.