Java

Осень 2015, лектор: Полон Любви-И-Обожания

Авторы: Лиза Третьякова, Настя Старкова

Собрано: 11 февраля 2016 г. 12:04

Оглавление

Глава 1

Введение

1.1. Информация про курсовой проект по Java

- Должен быть написан под Android
- Должен использовать возможности андроид: геопозиционирование, смс-ки, функции мобильника, touch-screen...
- Тема должна иметь хоть какую-то связь с реальностью, а не быть высосанной из пальца
- Можно делать парой, если совсем масштабно — можно трое, но это придётся обосновать
- Сдача проекта будет трёхэтапной:
    1. Защита архитектуры приложения Код им в этот момент безынтересен. Происходит примерно в октябре.
    2. Proof of concept Необходимо представить некое приложение, демонстрирующее функционал и возможности проекта. (это может быть заглушка на заглушке). Происходит примерно в ноябре.
    3. Сдача приложения. Происходит ближе к зачётной неделе.

Нужная для написания кода информация будет приходить по ходу семестра, но можно и бежать впереди паровоза или мучить практиков (ответы на эти наши вопросы — это одна из их задач >:3)

Сейчас надо думать над проектом и делиться на команды. Тему проекта надо будет согласовать с Антоном Михайловичем.

При желании тема может касаться архитектуры ЭВМ, которую нам также будут читать в этом семестре.

1.2. От авторов

Данный конспект рекомендуется употреблять в качестве вспомогательной добавки к Слайдам, любое количество раз в день по необходимости, утром, вечером, днём или ночью. При повышенной впечатлительности — натощак ;)

1.3. Зачем мы учим Java?

- В проге на C++ сложно найти ошибку, а если используется ещё и оптимизирующий компилятор, то и вообще. Java создавалась с учётом ошибок C++, на ней должно быть проще писать.

- Но ещё есть C#, и его тоже можно изучать. C# на Java похож. Но он сложнее. И он основывался на Java и её ошибках при создании,но там есть микрософтовская гигантомания. Зная Java, его можно спокойно выучить.

- Основная плюшечка Java не кроссплатформенность (компиляторы C++ есть правда на все платформы, а вот Java-машитны — не на все), а память. Вручную необходимо лишь выделять память, а всю работу по освобождению берёт на себя виртуальная машина. Это более безопасно (Java-машину можно ограничивать в правах), и это ещё и своеобразная песочница — если что, сломается машина, а не вся операционная система, как могло быть с C++.

- При этом главный минус — производительность.

  На самой первой Java-машине все работало в 10 раз медленнее, чем на C++, не считая время на запуск. Сейчас это не так — сейчас есть just-in-time компиляция(и адаптивная jit-компиляция). Машина умная и длогадывается, что некоторые куски кода не используются. Например

```
j = 0;
for(int i = 0; i < 1000; i++) {
    if(j > 0) {
        /*...*/
    }
}
```

  Этот цикл выполнится пару раз, а потом на него радостно забьют.

- Ещё один плюс — очень широкая стандартная библиотека, хорошая библиотека для работы с многопоточностью и кросс-платформа для отрисовки графики, чего нет в C++.

1.4. Напоследок

Есть 6 версий Java. Мы говорим про стандартную SE. ME сейчас не используется(она для старых мобильников). EE — серверные возможности. Card — современная для исполнения на совсем маленьких устройствах.

Принципиально это всё один язык, но различаются стандартные библиотеки и оптимизации в компиляторе.

Разных виртуальных машин тоже очень много. Самые известные: oracle java и openJdk (принята в качестве стандарта, она с седьмой версии). IBM J9 продвигается как серверная версия.

Глава 2

Язык Java

2.1. Некоторый синтаксис

Жестко закреплённые и принятые соглашения по code style — code conventions: если написать иначе, то работать будет, но все скажут „фуууу”.

Основные правила, которые надо помнить:

1. Названия всех классов — с большой буквы (и CamelCase)
2. Все функции, методы и поля пишутся с маленькой буквы
3. Названия всех констант пишутся полностью большими буквами (и, да, разделяются подчёркиваниями)
4. `this` пишутся только там, где это необходимо, чтобы скомпилировалось.

Комментарии есть, они как в C++. Но есть у них ещё одна функция — `javadoc`. Если текст комментария начинается со значка `*`, то документацию можно будет генерить автоматически.

В программе должен быть класс, у которого будет метод

```
public static void main(String[] args)
```

где `args` — массив строк, если при запуске передаются какие-то параметры.

Ещё одно кардинальное отличие от C++: там массив — это кусок памяти, попиленный на кусочки. А в Java это объект, у которого есть поле — `length`, чтобы можно было узнать длину своего массива и не схлопотать выход за его пределы и т.п.

По умолчанию оно само сделает код возврата 0.

Есть плюшка: другой цикл `for` — `foreach`:

```
    for(String str: args) {
        /*...*/
    }
//String ”--- тип переменной
//args ”--- коллекция, которую переменная пробегает
```

При использовании `javac`, если нужны какие-то нестандартные классы, надо править `-classpath`.

Запустить свою программу(содержащую класс): `java ClassName`. Важно, что расширение писать не надо.

2.2. Package и иже с ним

`package` — пока что это для нас „как namespace“. В Java принято, чтобы все классы находились внутри каких-то пакетов, названия их строго регламентированы.

Пока `public` пишем к каждому методу отдельно.

У строки для определения длины используется не поле, как у массива, а метод `.length()`. `.charAt(i)` — это потому что нельзя `[]`

2.3. Наследование

Синтаксис отличается от синтаксиса C++: заместо двоеточия и сопровождающей его компании ключевых слов и названий классов используется `extends` и название класса после него. При этом множественного наследования нет. Поля заводятся аналогично C++, только модификатор доступа надо писать.

Единственный метод, название которого пишется с большой буквы — конструктор, потому что его название совпадает с наванием класса.

`virtual` мы не пишем, потому что все методы — `virtual`. Если надо вызвать родительский — пишем `super.methodName(...)`.

```
1 return result*num;
```

Не пишем `this.num`, потому что code style.

2.4. import

`import` используется, если мы хотим импортировать классы (чтобы не писать их полное название).

```
1 import ru.spbau.kuznetsov.test01.parser.Parser;
```

Полезно писать целиком, а не со звёздочкой, чтобы избежать конфликтов имён.

2.5. Объекты

Объекты создаются только одним способом — на heap'е с помощью `new`.

В Java совсем нет указателей. Нет амперсандов. Всё, что есть — ссылки. И работать с ними как со ссылками. (с точки зрения плюсов аналогично механизму `smart_pointer`'ов)

2.6. Примитивные типы

Все примитивные типы в Java имеют строго регламентированный размер. И все они — `signed`. Все без исключения остальные типы — ссылочные (и пишутся с большой буквы). Объекты этих типов передаются по ссылке. Все эти ссылочные типы — классы. И все они наследуются от `java.lang.Objec` у которого есть следующие методы:

`String toString()`: то есть абсолютно все классы умеют приводить себя к строке

`boolean equals(Object obj)`: дефолтно два объекта равны, если у них одинаковый адрес в памяти.

`int hashCode()`: хеш-функция, для равных объектов должна быть равна.

2.6.1. Отличия примитивов от плюсовых

1. Нет автоматического приведения никаких других типов к `boolean`.

   `if(x = 3) {...}` не работает, потому что `if(3) {...}` не работает, так как нет автоматического приведения 3(числа) к `boolean`.

   О логических операциях:

   - `&&`, `||` — ленивые операторы.
   - `&`, `|` — полные операторы

   Для `boolean` есть обёртка `Boolean`. Из полезного в ней:

   `boolean parseBoolean(String)`: по ''true''/''false'' выдаёт нормальный `boolean`.

   `String toString(boolean)`: в обратную сторону.

2. Тип `char` занимает два байта и он беззнаковый. И это номер символа в Юникоде. Его класс-обёртка — `Character` — его методы, вроде, понятные.

3. Четыре целочисленных типа. Можно писать число через подчёркивание — чтобы числа большие читались лучше (`100_100_100_107` вместо `1e9 + 7`).

   Целочисленное деление здесь — это взятие неполного частного.

   Переполнение, как и в C++, не исключительная ситуация.

   В обёртках полезного немного, но есть

   - `MIN_VALUE`
   - `MAX_VALUE`
   - `.toString(typename)`
   - `.parseTypename(String)`

4. Вещественных типов два. Деление на ноль не исключительная ситуация.

5. Преобразование типов бывает явное и неявное.

   При округлении просто отбрасывается дробная часть.

   Пример на то, как (не)работает неявное приведение:

```
byte a = ...;
byte b = ...;
a = a + b; // не компилится, потому что
           // автоматически привелось к int'у
a = (byte)(a + b);
```

2.6.2. Из принципиальных отличий от примитивов C++: Autoboxing

Есть возможность автоматически заводить переменные типа обёртки (с пятой Java).
Зачем нужны такие переменные?

- Передавать переменную как объект (то есть не по значению).
- (важно) Запрещено создавать коллекции примитивов. То есть если бы такого преобразования не было, было бы тяжело. Но примитивные типы всё равно нужны, так как занимают меньше места и хранятся на стеке.

2.6.3. Важные замечания: комментарии к примерам на слайдах.

Чем там единичка так принципиально от тысячи отличается? Объекты сравниваются по адресу в памяти. И Java-машина кеширует 256 маленьких чисел, какие — зависит от настроек Java-машины, на это лучше не полагаться. То есть чтобы сравнивать обекты, надо использовать метод `equals()`.

Если переопределять `equals()`, то и `hashCode()` тоже надо переопределять.

2.6.4. Передача аргументов

Примитивные типы передаются по значению, все остальные — по ссылке. Так почему в примере все равно осталась единичка?

Потому что нельзя объект поменять, можно поменять его внутреннее значение (все дружно помахали ручкой механизму указателей). Класс `Integer` поменять нельзя, и никак нельзя поменять его внутреннее значение. То есть swap для `Integer`'а написать не получится.

Глава 3

Классы и ООП

С прошлой лекции: всё, как в плюсах, только шапочка другая. И swap не написать. У всех примитивных типов строго определённый всегда одинаковый размер. Есть обёртки для каждого примитивного типа.

3.0.5. Классы

Если значение не указано, то поле инициализируется значением по умолчанию. Для ссылок — `NULL`, примитивных типов — 0, `boolean` — `false`

3.0.6. Методы

Есть статический полиморфизм.

3.0.7. Конструкторы

Если конструктора в классе нет, то генерится конструктор по умолчанию – ничего не делает и параметров не принимает.

3.0.8. Деструкторы

Их нет. Есть метод `finalize()`, он будет вызван у объекта перед тем, как его удалит сборщик мусора. Удалять надо те объекты, до которых нельзя добраться из стека вызовов. А что если `finalize()` добавляет ссылку на объект в некий глобальный список, то есть до него становится можно добраться. Java-машина его всё равно удалит, и это способ её завалить. Это одна из причин, по которой `finalize()` сейчас в состоянии `deprecated`, только для обратной совместимости, пользоваться им нельзя, да и не нужен он нам в Java. Помимо того, что не факт, что ваш объект сборщиком мусора вообще будет когда-то удалён – то есть `finalize()` не будет вызван вообще.

3.0.9. Создание экземпляра

Через слово `new`. Кроме двух случаев:

- `Integer a = 5;`
- `String str = ''...'';`

3.0.10. Наследование

`extends`; В отличие от плюсов здесь есть только один тип наследования — публичное наследование. При этом множественного наследования нет. Все классы имеют единую иерархию, во главе которой стоит `java.lang.Object`

3.0.11. Конструктор в наследнике

Слово `super`, если нужнол что-нибудь от родителя
`this()` — вызов своего конструктора.

3.0.12. Интерфейсы

В плюсах некоторым аналогом абстрактных классов. Всё в интерфейсе — публичное. После `implements` можно писать сколько угодно нужных интерфейсов, которые будут реализованы в этом классе.

3.0.13. Абстрактные классы

Точный аналог абстрактынх классов в плюсах. Нельзя создать объект этого класса. Некоторые методы могут быть не реализованы.

Отличие от интерфейса: абстрактный класс может хранить состояние, там нет нормальных полей, есть только методы. И не все методы в абстрактном классе публичны.

3.0.14. java.lang.Object

- `toString()`
- `equals()`
- `hashCode()`
- `getClass()`
- `wait()`
- `notify()`
- `notifyAll()` — для многопоточного программирования, как и notify
- `finalize()`
- `clone()`

3.1. Модификаторы

Задание, которое чисто случайно можно схлопотать на зачёте, если АМ захочется: придумайте самую длинную цепочку модификаторов, которую реально можно ипользовать.

- `public`
- `private`
- `protected`

- по умолчанию(отсутствие модификатора)

  Зачем нужен пакетный режим доступа? Потому что в Java нет **friend**-классов. Правило хорошего тона: не оставлять поля без модификатора без очень жёсткой необходимости.

3.1.1. final

- **final** для поля – операция присваивания может быть проведена ровно один раз.

  Почему аккуратнее со ссылками? Потому что ссылку-то на неё мы поменять не можем, а вот внутреннее состояние — пожалуйста.

- У метода это позволяло (до пятой версии) убрать виртуальность от метода и соптимизировать программу. А потом ввели JIT-компиляцию, и с тех пор оно само всё понимает.

- У классов. Зачем запрещать наследоваться от класса?

```
private final int i = 0;//константа времени компиляции
public static final int N = 10;//самая настоящая константа, которая никогда не меняется и
private fincal int j;//у всех объектов может быть разной, но в рамках одного объекта ”---
```

3.1.2. Инициализация

Значение по умолчанию есть только для полей, а для переменных в функции, например, — нет.

3.1.3. Инициализация полей

В Java принято `все` переменные примитивного типа инициализировать руками, чтобы избежать глупых ошибок.

3.1.4. Статичекие секции

А статические секции инициализации запускаются при первой встрече названия класса в коде(при первом обращении). секция инициализации *обычно* создаются до выполнения конструктора, выполняется, соответственно, только один раз.

Раньше можно было написать

```
class A {
    static {
        System.out.print(”Hello”);
        System.exit(0);
    }
}
//Потом сделать
//java A
```

И оно бы напечатало что-то, несмотря на отсутствие метода `main`(он его не успевал поискать). Но только до Java7(теперь всё равно сначала ищет `main`, а потом идёт всё выполнять).

3.2. Ягодки. Ээээ, исключения

EOL — exception oriented language(о Java). В плюсах исключения ”ущербные чутка”.

3.2.1. Причины ошибок

- Проверяемое исключение — вы обязаны его обработать, иначе не скомпилится.
- Runtime exception — непроверяемые исключения.
- `getStackTrace()` — показывает, в каком месте исполнения программы произошло исключение.

`radix` — основание системы счисления
Параметром конструктора исключения передаётся текст. **`throws`** — знак того, что метод может кидать исключение.

```
1  public class Main {
2
3      public static void foo(int a, String s) throws IOException {
4          if(s == null) {
5              throw new IOException("Aaaaagh! Bird!");//обрабатываемое: обяз. слово throws и
6              //throw new NuberFormatException(...);//необрабатываемое исключение
7          }
8          System.out.ptinln(s.roLowerCase() + " " + a);
9      }
10
11     public static void bar(String s) {
12         foo(3, s);//либо приписать throws к методу bar, либо обработать это исключение
13     }
14
15     public static void main(String[] args) {
16         foo(5, null);//приписывать throws к main'у "--- это катастрофически плохая идея :)
17     }
18 }
```

3.2.2. Перехват

Внутри **`try`** — код, который может сгенерить исключение, внутри **`catch`** — обработчик
Есть ещё одна секция — **`finally`**. Она выполняется в любом случае, если надо — после **`catch`**, если пробрасывается на другой уровень — до проброса.

3.3. Применение исключений в коде

Правило обработки исключений: исключение надо обрабатывать там, где у вас достаточно данных, чтобы его обработать.

3.3.1. Управление на исключениях

Во-первых, исключения — это небыстро. Во-вторых, затираются настоящие ошибки — мы просто не отличим их от нащего стандартного способа выйти из цикла.

3.3.2. Запись в лог

Для себя — норм, но пользователю как-то неинтересно это читать.

3.3.3. Перехват базовых исключений

Все наследники отловятся туда же — вряд ли мы хотим всё одинаково обрабатывать.

3.3.4. Пользовательские исключения

Принято хотя бы один свой конструктор(от строки-причины писать).

3.3.5. Обёртывание исключений

Правильно — генерировать своё новое исключение, заворачивая в него старое. (чтобы обрабатывающий код мог узнать первоначальную причину исключения)

3.3.6. Может ли конструктор кидать исключение

Правило такое: если конструктор недоделан, то переменная становится null.

```
1 ExampleClass b = new ExampleClass(...);
```

Глава 4

Interface

Немного ключевых слов:

- `public`
- `static` — единственный для всех экземпляров. После 8 Java стало можно делать статические методы в интерфейсе.
- `final` — значение(поля, переменной) задаётся только один раз. Поэтому создание полей в интерфейсе — это такой способ задать константы, глобальные константы.

4.1. String

- Нет гарантии, что это массив(зависит от реализации java-машины).
- Неизменяемы.

  Зачем? При копировании кусочков можно со стопроцентной уверенностью сказать, что нам достаточно двух ссылок, а не полноценной копии — экономим память.

  Но с Java.7.43 подстрока всё-таки создаётся отдельно, потому что там есть потенциальные утечки памяти.

Что можно делать со строками:

- Запросить длину.
- `charAt()` — взять символ на определённом месте, работает константу.
- `toCharArray()` — амортизированно константа: один раз эта строчка реально копируется, потом ссылочки наводятся. Реальная копия создаётся только при попытке поменять массив — до этого там *что-то*

`indexOf()` — первое вхождение подстроки. `lastIndexOf()` — последнее вхождение подстроки.

4.1.1. Конкатенация

'+' — можно складывать только примитивные типы, обёрточные типы и, вот, строчки.

Кажется, плюсик долго работает(если только у компилятора нет возможности пооптимизировать — работу с константными строками, например). Утверждается, что это плохой и долгий код:

```
for(...)
    str = str + 'A';
//эта штука работает не совсем за линию, потому что там, как в векторе, но
//у нас захламляется память: был str-> |||||||
//                                     |||||||A
//                           str теперь ссылается на вторую строку, а первый объект надо ч
```

А `StringBuilder()` сработает за суммарную длину всех строк, которые надо склеить — сначала посчитает, потом склеит. Если надо 28 захардкоженных строковых литералов сложить, то надо использовать плюсик, потому что эти литералы ещё при компиляции сольются вместе – сопти-мизируются. Для склейки двух строк `concat` и плюсик — одно и то же, так что `concat` можно не использовать

4.2. Regexp'ы

Есть из коробки.

Глава 5

Generics

В Java нет шаблонов.

Для чего всё это. `ArrayList` — что-то типа вектора в плюсахъ.

Как выглядит цепочка наследования для объекта `(Object)2`: 2 → Integer → Object

Чем этот код нехорош? В него можно положить всё, начиная от крокодила до апельсина. Это было до пятой версии(пользоваться было невозмонжо). А после пятой весрсии появились дженерики. Что же это? Пусть то, что у нас там лежит, мы всегда хотим приводить к строке. Но у нас всё сломается, `integer` к `String`'у не приводится. Пишем <`String`>. При попытке добавить всё, что угодно, кроме `String`'а, мы получаем ошибку компиляции.

Как это должно быть устроено внутри? Параметризуемся `automobile`. Изначально мы можем туда положить автомобильчик и наследников. Если автомобильчик → `Object`, то это ж надо помнить, что положили, чтобы не `Object` обратно выдавать.

На этапе компиляции `Generic` гарантирует, что ничего лишнего мы не положим, а на выходе получим то, что укажем в <>.

5.1. Как работает

`Generic` компилируется только один раз. Вся информация про `T` стирается "к чёртовой матери"(c) и "условно"всё заменяется на `Object`. Проблема: в `runtime` мы не знаем, от какого типа мы создались, то есть внутри `Generic`-класса мы не можем создать объекты типа `T`. Инфа стирается и приписывается только к `add()` и `get()`.

Нельзя выполнять преобразования типа

```
1 ArrayList<String> = ArrayList<Object>
```

Два случая(и оба плохие >:3):

1. `Integer`'у присваивается `Object`.

   Плохо, если в `Object` добавим строчку, то потом из `Integer`'а попытаемся это достать и получим ошибку

2. `Object`'у присваивается `Integer` Попробуем добавить в `Object` строчку, но сразу получим ошибку

`ArrayList` — список, реализованный на массиве(он реализует интерфейс `List`). `List` — интерфейс, просто список.

5.2. Преобразование типов

Так писать можно, но выдаётся `warning`. В плюсах на `warning` можно забить, а в Java так не принято, надо без них, поэтому нужно ручками писать `SupressWarnings(unchecked")"`.

5.3. Примеры испо... Проблемы

1. "Итератор — это такая лягушка, которая прыгает между листиками"(c)

   Почему от `Integer` не работает? Потому что запрещено приравнивание разных типов. `Collection<?` вот это мы называем "более элегантный подход чем в C++. То есть "мне всё равно, какую коллекцию я получаю на вход". Только команда `next()` будет выдавать `Object`, потому что она не знает ничего. Такой код принимает на вход абсолютно любую коллекцию.

2. Мы хотим передавать не абы что, а только `Shape` или его наследников – пожалуйста:

```
1 ...<? extends Shape>...
```

   Тут итератор будет возвращать — самый высокий возможный тип — `Shape`

3. Итог: параметризуем метод.

4. В Java нельзя сравнивать объекты – нельзя перегрузить операторы больше-меньше. Надо реализовывать интерфейс `Comparable` (он параметризованный). `class Integer implements Comparab` `class Integer implements Comparable<Object>`

   `<? super T>` — любой класс-родитель T. До этого не работало, потому что кружочек был `Comparable` не от кружочка, а от чего-то выше него — от фигуры. Это адекватно, просто надо это явно указать.

5. `get()` — выдаётся `Object`, и это точно, а там может быть всё что угодно. Это ошибка компиляции.

   `typeCapture`

   Компилятор проверит, что везде тип `T`, всё совпадает и мы молодцы, и только после этого всё стирает и приводит к `Object`'у.

Есть в Java золотое правило программированрия: если ваш тип данных является `Generic-типом`, то везде, где нужно что-то параметризовывать, оно должно быть параметризовано наиболее общим типом данных.

Отличие от плюсов:

1. `T tmp = new T();` писать нельзя
2. Если мы подразумеваем, что где-то что-то есть, то надо писать `extends`.
3. Дся информация о типах стирается.

5.3.1. ЕЩё один пример

```
1 class Fruit {}
2
3 class Apple extends Fruit{}
4
5 class Antonovka extends Apple {}
```

```
6
7  public class Main {
8      public static void foo(ArrayList<? extends Apple> l) {
9  //         l.add(new Apple()); не компилится -- вдруг это был список антоновки?
10 //можно положить _только_ одну вещь -- null, он имеет все типы на свете
11 //бессмысленно, беспощадно, но можно
12         Apple a = l.get(0);//доставать яблоки можно
13     }
14     public static void bar(ArrayList<? super Apple> l) {
15 //сюда можно положить Apple или его наследников
16 //а доставать можно наиболее общий тип для всех этих super -- Object
17         l.add(new Apple());
18         l.add(new Antonovka());
19         l.get(0);
20 //на самом деле информация стирается не всегда до Object'а, а до наибольшего общего типа
21     }
22     public static void main(String[] args) {
23
24     }
25 }
```

Глава 6

Collections frameworks

Коллекция – неупорядоченный контейнер, причём в двух смыслах: нет заданного порядка обхода и нет на элементах сравнения.

Проверка элемента на вхождение в коллекцию производится по `equals()`.

Почему `Object`? В нём определён `equals()` Почему `Collection<?>`? Потому что нам всё равно, потому что в `equals()` можно передать любой `Object`.

`remove` удаляет только один объект.

`unsupportedOperationException` — например, если попытались сделать remove в немодифицируемой коллекции.

`Scanner` – позволяет работать не с битами, а уже с объектами.

Операций у коллекции мало. `Iterator` написан в интерфейсе коллекции, это тоже интерфейс. Параметризован тем же типом, что и коллекция. Это такая стрелочка, которая ходит между элементами. Команда `remove()` удалит тот элемент, который мы только что перепрыгнули.

`Iterator` кидает два исключения: при достижении конца коллекции и второе. По договорённости итерторами нельзя пользоваться, если коллекция изменяется. Это сделано по нескольким причинам:

1. Избежать инвалидации.
2. Итератор может быть не только обёрткой указателя, а `next()` может работать неконстантное время. Из-за этого бывает удобно сразу упорядоченно выгрузить коллекцию в массив. И вот там порядок сильно нарушится при изменении коллекции, и итератор "очень сильно"инвалидируется при добавлении/удалении.

   При этом `remove()` делать можно, потому что он про внутреннюю структуру итератора что-то знает.

В примере применения итераторов `final` — это просто для поддержки штанов, ничего более. Просто `element` всё равно будет ссылочкой на элемент коллекции, поэтому элемент внутри коллекции мы так(через =) не поменяем в любом случае, значит присваивание нам скорее не нужно.

Интерфейс `Iterable` — те, у кого можно брать итератор. У них можно делать укороченный, синтаксически сладкий `for`.

В примере плохо, что у нас после последнего слова тоже будет запятая. Лучше исправлять это методом "первое слово + for[запятая-слово] а не делать ещё `if` с `hasNext()`, чтобы было меньше проверок.

Нельзя внутри коллекции создавать объекты типа T. Внутри коллекции нельзя создать массив типа T. Поэтому нет метода `T[] toArray();`. Возвращается тот массив, который коллекции передали, заполненный её элементами.

В коллекции всего много. Поэтому нам сделали класс `AbstractCollection`. `remove()` писать необязательно. Почти всегда в своих коллекциях мы его писать не будем. Гораздо проще реализовать 4(3) метода, чем всю ту кучу. Через них всё остальное можно реализовать.

Какие разные виды коллекций мы знаем?

6.0.2. Множество(Set)

Каждый элемент встречается только один раз. Но ещё же есть всякие пересечения-объединения.

У нас появляются важные требования к `equals()`. Хотя с чего бы? Зачем мы сюда тащим всю эту математику про отношения сюда?

- `point.equals(colorPoint)` — координаты совпадают →вернётся `true`.
- `colorPoint.equals(point)` — цвета не совпадают →вернётся `false`.

В итоге нарушилась симметричность — неудобно.

Надо что-то делать! Ладно, будем пытаться приводить к `coloPoint`'у, и если не получается, то вызывать `super.equals()` Всё, теперь-то мы молодцы? Нет: `point`, `colorPoint1` и `colorPoint2` — нарушается транзитивность, потому что при сравнении с первой все результаты будут `true`, а при сравнении второй с третьей — `false`

Варианты решения: послойное сравнение.

```
this.getClass().equals(that.getClass());
```

То есть для равенства мы потребуем совпадение типов.

`canEqual()` — в слайде на `Point` vs `ColorPoint` в любом порядке даёт `false`. Какой профит? А мы могли бы написать просто

```
if(o.isInstanceOf(ColorPoint)) {
    //сравнивай, как Point, просто по координатам
}
```

Hashset & LinkedHashset

`LinkedHashset` – заодно хранится списочек в порядке добавления.

Согласованность с `equals()` — стрелочки в обратную сторону нет, потому что коллизии.

6.0.3. Списки

`List` – наследник коллекции, `Generic`-класс.

Всякие поиски принимают `Object` →сравнивается по `equals()`.

Постоянное `get(i)` компилируется в то же самое, что и проход итератором, но это в `ArrayList`'е. А вообще в `get` может быть какая-то логика, так что в общем случае лучше писать итератор.

`AbstractList`, конечно, тоже есть. Но реализовывать нужно чуть-чуть другой набор методов. При таких методах можно не реализовывать свой итератор, потому что он его возьмёт из `get()`'а, но лучше реализовать, потому что `get()` может работать неконстантное время.

`AbstractSequentalList` – это тот, в котором выгоднее писать итератор, то есть где `get()` долгий.

6.0.4. Очереди и деки

Элементы из очереди выдаются в некотором порядке. Приоритеты задаём мы в своей реализации. У очереди может быть ограниченный размер. Она вправе не принимать какой-то элемент(например, потому что память закончилась) (тогда будет брошено исключение IllegalStateException)

В очереди запрещено хранить null. Методы, не бросающие исключения, делают то же самое, что и бросающие, но возвращают null, если требуемого элемента нет(поэтому-то его и нельзя хранить).

Куда же без `AbstractQueue`. Надо реализовывать неброские[в плане исключений неброские] методы, потому что исключения – недешёвая операция.

6.1. Отображения(mapping)

Это не наследник коллекции, потому что его параметризуем двумя... параметрами

6.2. Упорядоченные коллекции

Упорядоченные в плане "есть порядок на множестве то есть их можно сравнивать между собой.

Зачем нужны и `Comparable`, и `Comparator`? Мы можем хотеть несколько компараторов, а `equalsTo()` у нас единственный.

- `SortedSet` — `set`, на элементах которого задан порядок.
- `TreeSet` — обычно реализоввано как красно-черное дерево с некоторыми модификациями.
- `NavigableSet`

`Map`'ы тоже разные

- `SortedMap`
- `TreeMap`
- `NavigableMap`

6.3. Алгоритмы

Класс `Collections` — в нём лежат все эти утилитные методы. `Collection` — это интерфейс.

Операция `shuffle()` применяется только к спискам, потому что перемешиваются номера.

В сортировке испольщуется не `MergeSort`, и даже не совсем `TeamSort` — всё зависит от типа объектов и размера коллекции

Лирическое отступление: по аналогии есть класс `Arrays`. Класс, содержащий все те же самые утилитарные методы, но для массивов.

6.4. Вывод

Коллекции бывают упорядоченные и неупорядоченные.

Ещё. В чём разница между двумя кодами?

```
ArrayList<> l = new ArrayList();
for(int i = 0; i < l.size(); i++) {
    System.out.println(l.get(i));
}
```

и

```
List<> l = new ArrayList<>();
for(a: l) {
    System.out.println(a);
}
```

Правило такое: нужно использовать что-то наиболее общее, если возможно. Если же мы используем что-то специфическое, то максимально точно указывать тип.

6.5. Текущие объявления

Скоро будет кр. Лучше дома попробовать реализовать какую-нибудь коллекцию, типа BST.

С 9го числа будет три лекции подряд про андроид – пятница-среда-пятница. К 21му надо сдать архитектуру практику – его надо спросить про то чего он от нас хочет.

Глава 7

Внутренние(inner) и вложенные(nested) классы

Nested class может добираться до статических методов и полей внешнего класса, не связан ни с каким его конкретным экземпляром.

Inner class'ы — это:

- basic(или "базовые классы")
- local(или "локальные классы")
- анонимные классы

Объект этого класса не может существовать без объекта внешнего класса. (Например: VectorIterator не может существовать без конкретного Vector'а) Создание происходит при участии объекта внешнего класса следующим образом: `objectOfOuterClass.new InnerClassName();`. Если "objectOfOuterClas то подразумевается `this.new InnerClassName();`.

7.1. Basic классы

Посмотрим на пример.

```
1 public class Selector {//outer class
2     private class SequenceSelected {...}
3     //private -- никто извне не может создать экземпляр этого класса
4     //нет слова static, значит класс привязан к конкретному объекту, значит это inner клас
5 }
```

При этом, так как во внутреннем классе обязательно есть неявная ссылка на объект внешнего класса, внутреннему классу доступны все поля и методы внешнего класса, в том числе приватные.

```
1 public Selector selector() {//возвращаем Selector, так как SequenceSelector имеет модифика
2     return new SequenceSelector();
3 }
```

Во что это всё компилируется? Под каждый класс создастся свой файлик:`Sequence.class`, `Sequence$Selector.class`

7.2. Nested классы

Статический вложенный класс.

```
public class Outer {
    public static class Inner {//лучше бы назвать его nested, ну да ладно
        public static void main(String[] args) {//то есть этот класс можно запустить
            foo();
            Example04 t = new Example04;
            t.boo();//и взяли вызвали приватный метод
        }
    }//это удобно для тестирования приложения:
    //запускать java Example04$Inner для тестирования,
    //а при распространении(мы распространяем только class-файлы) можно просто удалить фай
    /*...*/
}
```

Обратите внимание на слово **static**: вложенный класс не привязан в конкретному объекту. Если бы не было **static**, был бы это внутренний класс. Вот и разница.

Интересный вопрос: а зачем так вообще писать? Сокрытие реализации. (Многоугольники, сделаем точку nested — пользователю точки не понадобятся).

Также к таким классам принято писать модификаторы доступа **public**/**private** (**protected** — это какая-то вещь в себе).

7.3. Анонимные классы

```
events[i] = new Event() {
    /*...*/
}//и хоть каждый раз -- разный класс
//и используется только здесь, больше не нужен нигде
```

Компилируется в EventTest$1.class Но воспользоваться этим файлом ручками мы не можем. И создавать классы с именем-числом тоже нельзя.

`getEvent()` производит нам каждый раз новый класс. Анонимные классы могут использовать поля-то любые, а вот переменные, которые внутри метода, можно использовать, только если они **final**. Почему? Пусть можно было бы. Создали мы в **for**'е один класс с `i`, потом другой. `i` изменилась. Надо её менять в первом классе или нет? Непонятно, вопрос договорённости. Поэтому пусть лучше сразу **final** будет.

Анонимные классы используются много где. Характерный пример — сортировка.

```
class MyComparator implements Comparator<Integer> {
    public int compare(Integer a1, Integer a2) {
        return a2 - a1;
    }
}

public class ArrayTest {
    public static void main(String[] args) {
        Integer[] arr = new Integer[100];
        Random r = new Random();
        for(int i = 0; i < arr.length(); i++) {
```

```
            arr[i] = r.nextInt(10000);
        }
/*
        Arrays.sort(arr);
у этой сортировки всё очень хорошо, пока нам не потребовался нестандартный компаратор
        Arrays.sort(arr, new MyComparator);
это долго, нудно и отвратительно
        Arrays.sort(arr, new Comparator<Integer>() {
у анонимного класса нет конструктора(даже имени нет, чего уж там)
есть секция инициализации
            {
                System.out.println("Hello, world!");
            }

            public int compare(Integer a1, Integer a2) {
                return a2 - a1;
            }
        });
*/
        Arrays.sort(arr, (a1, a2) -> a2-a1);
        /*
        lambda -- это синтаксический сахар с Java8.
        у интерфейса компаратор всего один метод, и его мы описываем.
        Даже типы не пишем, потому что ?компилятор? их сам знает
        */
    }
}
```

7.4. Локальные(local) классы

Пишутся внутри метода. Область видимости этим методом и ограничена.

7.5. Пример: Singleton

```
public class Singleton {
    private Singleton() {}//приватный конструктор

    private static class SingletonHolder {
        public static Singleton instance = new Singleton();//можно, так как внутренний кла
    }

    public static Singleton getInstance() {//статический метод, его от класса вызовем
        return SingletonHolder.instance;//поле можем взять, потому что оно статическое, од
    }
} //это Singleton с ленивой инициализацией

class Singleton2 {//внутри одного файла может быть только один public class
    private Singleton2() {

```

```
16      }
17
18      private static Singleton2 instance = new Singleton2();//это поле проинициализируется,
19      //оно может быть раньше нашей реальной необходимости
20      //а SingletonHolder вызоветсся, только когда его из getInstance попросят, то есть экзем
21      //понадобится
22
23      public static Singleton2 getInstance() {
24          return instance;
25      }
26      //можно ещё какой-нибудь foo() приписать
27  }
```

7.6. Пример: FabricMethod

У нас есть `Product`'ы и `Creator`. `Creator` — абстрактный класс, у которого только один метод — `getProduct()`. Почему он абстрактный? Потому что `Product`'ы могут быть разные.

Для `ConcreteProduct`'ов — `ConcreteCreator`'ы. В `ConcreteProductB` запихали его `ConcreteCreatorB`, который с помощью анонимного класса создаёт `Product`'ы. Причём этот `Creator` — `static`, так что он ровно один, и добираться до него легко, так что этот вриант предпочтительнее.

7.7. Ещё один хитрый пример

```
1   public interface Node {
2       public Node getNext();
3
4       public final static Node NULL = new Node() {//final static -- константа
5           //реализация по умолчанию
6           public Node getNext() {
7               return Node.NULL;//по определению NULL ссылается сам на себя в односвязном спи
8           }
9       };
10  }
```

7.8. CurrentSummary

Типичный пример использования

- внутреннего класса — итератор
- анонимного класса — сортировка
- вложенного класса — спрятать часть кода + тестирование

7.9. Множественное наследование

Но вот вопрос: как в Java эмулировать множественное наследование? Вот есть у нас класс `UpdateEvent`, в нём есть какая-то начинка:

- Метод `fireEvent`(пробегает по списочку и применяет к каждому элементу какой-то `update`).
- Сам списочек `event`'ов, которые надо обновить.
- Конструктор.

А ещё у нас есть классс `DrawEvent`, у которого начинка, в сущности, такая же, только он каждый элемент списочка рисует, а не обновляет.

Хотим себе класс, который умеет и то, и то. Технически нам нужен класс, который умеет приводиться и к первому, и ко воторму + все их фичи тоже умеет + полиморфизм.

```
class X {
    private class A2 extends A {

    }

    private class B2 extends B {

    }

    private A a = new A2();//"привестись к типу" ---> вернуть, соответственно, a или b.
    private B b = new B2();//получаем через них полиморфизм: fierEvent в этих классах дела

    public A toA() {return a;}
}
```

7.10. Заметки на полях(планы на ближайшие лекции)

В следующий раз кр на практике. Базовые вещи, generic'и, коллекции

Презенташка может помочь с домашкой. Сдавать можно на Java8

После следующей лекции будет рассказ Миши Кринкина про приложение на android На практике будет подразумеваться, что мы начнём своё писать и будем задавать свои вопросы.

На кр можно пользоваться всем, чем угодно – ноутбуком. Надо будет написать код.

Глава 8

Reflection

8.1. Основное

Пакеты `java.lang` и `java.lang.Reflect`. Есть класс `Class<T>`, дающий информацию о типе: все его методы, поля, родители.

Получить можно разными способами:

- Используя метод `getClass()`: `Class<String> c` =a".getClass();//вернёт Class"
- `Integer.TYPE` эквивалетнен `int.class`

`forName` позволяет получать классы по имени. Удобно используется в плагинах (нашёл себе в папке кучу файликов и все проверил, класс ли, и, может, использовать как-то).

```
//это комментарии к кусочку примера из лекции

Initable.class//подгружается java-машиной, 47 захардкожено.
//А чтобы выдать staticFinal2,
//сначала его надо вычислить -- и это есть до этого выполнить
//статическую секцию инициализацию.
```

Имя класса:

- Каноническое — с прописанным пакетом.
- Полное — с внешним классом через точку, если таковой есть.
- Простое — ну, просто название :)

Ещё несколько полезных функций:

- `isAssignableFrom()` — является ли вторая штука нашим наследником.
- `isInstance()` — можно ли объект привести к нашему классу. `instanceof` работает быстрее, потому что на этапе компиляции компилятору может быть известна информация об `x` и сработает быстрее (иначе всё точно так же через рефлексию)

8.1.1. О том, как и где можно использовать cast(), а где нужно

```
//foo -- статический метод и в классе A, и в классе B
A x = new B();
x.foo(); //статические методы не виртуальны, поэтому вызовется из класса A
((B)x).foo();//работает только если B известен
//если B неизвестен
Class bClass;
(bClass.cast(x)).foo();
```

`Замечание в тему:` в отличие от `isInstance`, == даёт `true` только в случае иточно совпадения типов сравниваемых объектов. Из соответствующего примера с лекции можно заметить, что `Class` является синглтоном.

8.2. Применение

Реально Reflection используют, чтобы добраться до внутренностей какого-то класса.

- Modifiers `getModifiers` возвращает массив констант(`ABSTRACT`, `FINAL` и т.д.).
- Fields
  - `getFields()` вернёт все публичные поля, в том числе и родительские.
  - `getDeclaredFields()` вернёт все поля вне зависимости от модификатора доступа, но не из родителей. То есть мы реально имеем доступ к любым приватным данным.
  - `getType()` — `returns Class`
  - `get(Object)` — тот объект, у которого мы хотим вытащить значение поля. То есть мы знаем имя поля и хотим к нему обратиться, но у какого объекта... Вот в `get()` и передадим, у какого объекта.

  `B.class.getField(y").get(b);//так можно написать"`
- Methods
  - `getMethod(name, Class... Parameters)` — к имени нужно ещё добавлять принимаемые типы.
  - `invoke()` — позволяет вызвать в том числе и приватный метод. Если метод `static`, то в качестве `object` можно передать `null`(благо нам всё равно, что именно передавать).
- Constructor
  - Вместо `B.class.getConstructor().newInstance();` можно, если хочется просто конструктора по умолчанию, написать `B.class.newInstance()`

Но нельзя просто так безнаказанно вызывать и использовать приватные методы, поля и т.п. >:3 Если мы попытаемся, то вылетит исключение. Но его, конечно, можно обойти: сделать `setAccessible(bo` пометить, что мы правда хотим нарушить инкапсуляцию.

`field.setAccessible(true)` — по этой переменной теперь можно будет добираться до соответствующего поля в любом случае. В то же время если у нас будет ещё одна такая же переменная `field2`, в которой `setAccessible()` не сделан, то через неё это сделать будет нельзя.

Если метод ожидает увидеть на вход примитивы, то в `invoke()` передаём ему мы обёртки. И как мы это делаем? Массивом `Object`'ов.

8.3. Массивы

Зачем ему методы `get` и `set`? Почему не `[]` делать? `newInstance()` возвращает `Object[]`.

8.4. ClassLoader

Описывает механизм загрузки классов. Можно сделать свой — кастомная загрузка класса в java-машину. (можно, например, хранить свои классы в базе данных/в интернете и подкачивать их оттуда)

`ClassLoader`'ы образуют дерево(иерархическое). Если какой-то `ClassLoader` из этой иерархии просят загрузить класс `X`, то он сначала обежит поспрашивает потомков, не грузил ли кто из них, и если нет, догрузит сам. Если в поддереве уже загружено несколько, то вернёт первый найденный. Если кто-то изпредков уже грузил `X`, то нам его уже грузить нельзя.

8.5. Выводы

Типичными примерами использования `Reflection`'а являются

- Сериализация(требуется передать все данные объекта или класса, невзирая на модификаторы доступа).
- Система плагинов(возможность работы с любыми классами, получая их имена автоматически благодаря `forName`).

Глава 9

Android

activityintentBroadCastReceiverих нужно указывать в android-манифесте PendingIntent – тот, кто его распространяет, даёт свои права на выполнение чего-то приложению, которому этот intent посылает

Context – базовый класс, предоставляющий доступ ко всем андроидовским ресурсам. Activity является его наследником. BroadCastInten - нет, но ему надо передавать контекст, пототму что чего-то ему там нужно

Для продолжительных действий Bra=oadCastReceiver не подходит, нам нужен service(прописывается в манифесте). Добавились два permission'a: INTERNET и NETWORK_STATE.

Создаём интент и указываем, кого хотим запустить. Интент – такой способ общаться между процессами. IntentService – скармливает нам интенты из очереди по одному. Отлично, чтобы тихонечко работать в фоновом режиме. Location – получаем из intent'a. Кастить надо к чему-то тому, потому что иначе вылетит exception. URL собирается каким-то андроидовским API. connetion – получается от url'a.

```
1 connection.setRequestMethod(''GET'');//мы хотим получать данные
```

Дальше получаем строчку через InputStream. ЧИтаем его. В примере – JSON, а не XML, просто потому что, проще ему так. Правда, он даже не парсит его.

Полезно подумать об оффлайн-работе. Получить результат по сети и сохранить его в базу данных. provider – наш content-provider class Contract – android требует описать структуру базы данных в отдельном классе сначала. Названия колонок, типы записей, URI CONTENT_TYPE – тип набора записей CONTENT_ITEM_TYPE – тип одной записи implements BaseColumns – ну, так надо

DBHelper – мы через него создаём базу данных использует SQLite, как видно createtable – всякие имена, ещё чего-то

Иногда БД надо менять, поэтому есть ещё поле 'DATABASE_VERSION' — если текущая версия не соответствует версии, которую сейчас ожидает приложение, то вызовется метод onUpgrade().

WheatherProvider: итилитарные методы всякие bulkInsert – вставить не по одмону, а сразу пачкой getType – единственный метод, использующий поля типа в Contract'e. sUriMatcher – проверяет конкретный URI относительно шаблона. Мы по результату будем понимать тип запроса и что нам возвращать.

Почему мы нигде connection в базе данных не закрываем? "ОС сама всё потом освободит"В поздних версия появился метод для дебага, закрывающий connection – но только для дебага.

Об изменениях базы данных неплохо бы оповестить интересующихся. Метод NotifyChange, которому передаётся URI. Чтобы сделать его доступным извне, нужно в манифесте надо прописать android.exported="true". Если у нас false, то не очень понятно, зачем городить отдельный contentProvider – но пусть будет, нам должны были его показать. А так можно было и напрямую к базе данных пообращаться.

Весь UI в большинстве приложений распихан по фрагментам. Model, в котором лежат данные, по которым строится UI, и View, который рисует, разделены Adapter – через него происходит доступ к данным, которые надо отображать, он оповещает, кого нужно, и тыкает View, когда нужно CursurAdapter – если нужен доступ к базе данных newView: parent – куда её положить; у нас некуда Layout описывает расположение рисуемых элементов. Поле androi:id – штука, по которой потом можно будет потом данный элемент найти в программе. Имя туда какое-нибудь запишем. LayoutInflater – получает созданную нами XML-описание и строит по ней дерево объектов newView - создаёт новый view. Можем положить туда в приватное поле какие-то даннные, если нужно вместе с ней хранить, через метод setTag. bindView – заполняет вьюшку данными(новую не всегда надо создавать, а чтобы переиспользовать старую, её нужно дать данные)

CursorLoader – стандартный для загрузки данных из базы данных. swapCursor освобождает старый курсор и заполняет его новыми переданными данными(если надо дропнуть курсор – закончились мы, например – можно передать null). Так Adapter можно оповещать об изменениях в данных, чтобы он передавал изменения во ”вьюхи”.

Если хочется добавить файлы, layout’ы. Их чего-то по пять штук добавляется... Суффиксы – размер экрана и плотность пикселей; портретная/альбомнаяя ориентация экрана; локализация; ...

Глава 10

Потоки

☒Потоки можно воспринимать как стек вызовов (каждый поток имеет стек вызовов).
В любой нормальной операционной системе есть планировщик потоков.
У JVM свой планировщик потоков (не зависит от планировщика ОС).
Есть потоки-демоны, а также пользовательские потоки (JVM завершает программу, когда завершились все пользовательские потоки, при этом демоны завершаются вне звисимости от того, что они делали)

10.1. Создание потоков

Сущности:

- Класс поток — Thread (создание потоков и работа с ними)
- Интерфейс Runnable (поток сам по себе Runnable) - описывает то, что запускается в потоке (имеет один метод run)

Создание потока как объекта класса Thread.

```
Thread t = new Thread(new Runnable() {
    public void run() {
            System.out.println(''Hello'');
        // Тут один поток выполняет одну задачу, мы его не можем переиспользовать,
        //а создание потока - очень дорогая операция, поэтому так лучше не делать
    }
});
```

Чтобы запустить поток - команда `t.start();`
Нужно не путать `t.start()` и `t.run`: первое — это создание потока, а `run()` — просто вызов метода.

```
NameRunnable nr = new NameRunnable(); // Создали новую задачу
```

Создаем три потока, которым в качестве параметра передаем эту задачу.
Мы делаем start и между потоками управление переходит как-то. А если сделать run, то они будут все дружно по очереди работать в main.
Конструктор потока: можно передать задачу(а можно не передавать), можно задать имя(можно не задавать).

10.2. Свойства потока

- `daemon` Правда ли поток — демон.
- **Приоритет** Влияет на то, насколько часто ему дают выполняться на процессоре.
- **Состояние потока** Самое главное его свойство.
  - **new** Создан, не запущен.
  - **runnable** Запущен(но может ещё не выполняться).
  - **running** Активное выполнение(переключением управляет планировщик потоков).
  - **dead** Закончился.
  - **waiting/blocking** Это `blocked` $\vee$ `waiting` $\vee$ `timed\_waiting`

`getState()` — возвращает состояние потока.

10.3. Взаимодействие потоков

Самое простое и типичное действие — команда `sleep`, заснуть. Поток прерывает своё выполнение на время, *не меньшее* чем то, которое мы попросили. `Thread.sleep()`(только милли-/наносекунды) или `TimeUnit.<единица измерения>.sleep()`(всякие более долгие времена).

10.4. yield

`yield()` — символ того, что нас можно прерывать пока. Полезность? Пусть есть два потока, которым лучше закончиться примерно одновременно. Тогда, чтобы планировщик потоков сделал(дал) примерно по одинаковому процессорному времени обоим(примерно равномерно), `yield()` и присутствует сегодня здесь. Осторожно: не рекомендуется использовать, если нет особо тонких мест в производительности.

10.5. Приоритеты

По умолчанию приоритет равен 5. Почти никакие ОС не поддерживают прям 10 приоритетов. В винде чуть поменьше, в Linux чуть побольше, в маке примерно столько же. Поэтому рекомендуется пользоваться именно этими тремя константами: уж три-то приоритета в любой ОС найдётся.

10.6. блокировка другого потока

Методы `sleep` и `yield` — статические. Нельзя попросить уснуть другой поток.

10.7. ожидание окончания потока

`join()` просит наш поток остановиться и дождаться, пока выполнится другой поток. `t.join()` заставляет текущий поток дождаться окончания потока `t`. Применение? Программа, считающая интеграл, должна дождаться, пока посчитаются интегралы на каждом кусочке.

`joib(millis)` – это делается для того, чтобы программа не зависала.

Следующая штука приведёт к dead-lock'у

t1 t2 t2.join() t1.join()

Если же при этом указать millis, то deadlock'а не произойдёт.

10.8. прерывание потока

`interrupt()` является лишь рекомендация для другого потока, чтобы он шёл лесом

Исключение `interrupted` кидается, если поток прервали, когда он чего-то ждал/бездействовал(`sleep` `join`). Это способ явно показать, что его не нормально разбудили, а `interrupt`.

Глава 11

Синхронизация кода

Всё вышесказанное малоприменимо в реальной жизни. Реально мы используем одни и те же данные. И ладно бы мы их только читали – но так мы же их ещё и пишем. long a = 1 long – это 8 байт. Java-машина за 1 раз изменяет only четыре байта. И не дай бог первые 4 отредактировала первая программа, а в это время вторые 4 – вторая. Чтобы это нормальнло работало, есть синхзронизация кода. А также понятия блокировок и критических секций.

Одной и той же блокировкой одновременно может владеть only один поток.

syncronized(o) оба потока должны синхронизироваться по одному и тому же объекту. И нельзя синхронизироваться по примитивным типам. Обычно синхронизируются по тем объектам, которые изменяют.

Куда ещё можно писать syncronized? Методам. Эквивалентно "почти потому что первый шустрее Синхронизация тогда происходит по `.this`

Если у нас в классе есть 2 метода – getValue и incValue, – то их нужно делать синхронизованными. Если inc в какой-то момент начал, но не закончил, то get вернёт не то, а другой inc прибавит не к тому

Если у нас меняется что-то маленькое, что можно за 1 раз увеличить, то всё равно синхронизировать надо. Для inc'а ясно, потому что иначе у нас оба могут увеличить случайно старое значение вместо последовательного старое -> новое -> итоговое Но и get тоже надо. Но вызов – это тоже не атомарная операция, поэтому у нас между вызовом get'а и возвращением им значения значение могло подрасти. Но есть ещё и вторая причина, но АМ скажет о ней чуть дальше.

syncronized для статического метода синхронизируется по классу. Никакие два статических синхронизованных метода класса не могут работать одновременно.

11.1. примерчики

произовдитель-потребитель: с syncronized медленно, а без него точно нельзя. Или надо научиться засыпать вместо активного ожидания. По этому поводу просим любить и жаловать...

Глава 12

Мониторы и условия

Если было несколько потоков, которые ждали, то непонятно, кто тот единственный счастливчик, которому прилетит одиночный `notify()`. (На деле выбирается случайным образом)

Команда wait() блокировку отдаёт

```
syncronized (o2) {
    o2.wait();//теперь другие могут занять o2
                    syncronized(o2) {
                        ...
                        o2.notify();//мы тыкнули wait(), теперь он пристальным взглядом за
                        ...         //только мы отпустим блокировку, тут же заберёт её обра
                        ...         %разбудили древнее зло, блин))
                    }
    ...
    ...
}
```

вот так это выглядит

wait() нужно вызывать в while'е, чтобы там всякие такие спонтанные пробуждения побороть.

12.1. Java memory model

В контексте многопоточности.

12.1.1. Три свойства какие-то

- Атомарность Не все операции атомарны. Причём не только с **`long`** и **`double`**, а и с другими типами при некоторых операций.
- Видимость Чтобы спокойно программировать, мы хотим, чтобы переменные всегда были корректны – это видимость. Она гарантируется ровно в одном из трёх случаев:
  1. После изменений поток 1 освободил блокировку, которую захватил поток 2
  2. После изменения поток 1 создал поток 2
  3. Поток 2 дождался потока 1. Тогда ему видны все изменения, сделанные первым потоком. Это самый правильный способ.

  Если мы не делаем ничто из вышеперечисленного, то потоки могут увидеть любой возможный вариант развития событий и изменения переменных.

```
int a = 0;
int b = 0;

//T1
a = 1;
b = 2;

//возможные значения пары (a, b):
//(0, 0), (1, 0), (1, 2)
//(0, 2)!
//Потоки выполняются на разных ядрах процессора. Для каждого потока бужет свой кусочек
//Чтобы изменения оказались в оперативной памяти должно пройти некоторое время, и в ка
//наши переменные будут туда переведены -- неизвестно.
```

join() происходит через synchronized(по какому-то мифически-мистическому объекту типа потока).

- Упорядоченность Гарантируется, что в итоге программист получит ровно то, что задумал. Но порядок – не гарантирован: компилятор(а у нас два: тот, который в байткод, и JIT-компилятор) может в рамках оптимизации кода переставлять инструкции.

12.1.2. Volatile-переменные

Иногда мы не хотим про всю вышеперечисленную штуку думать. Для этого есть volatile-переменные: все операции с ними атомарны, а работают они с общей памятью.

Надо помнить одну вещь: volatile – это плохо. Ну, то есть долго – операции с общей памятью всё-таки. Поэтому прежде чем их использовать(не зря же они всё-таки есть), надо хорошо подумать(о производительности, например).

12.1.3. Про Singleton

Если нет синхронизации, может получиться несколько helper'ов – плохо в singleton'е Если она есть, то долго(после того, как хелпер создан, потоки все равно строго по очереди вызывают getHelper, хотя уже смело могут все вместе). Double-Checked Locking: изначально не работает. Потому что при **new** сначала выделяется память, значение ссылки инициадизируется и только потом запускается конструктор. А мы уже решим, что можно пользоваться – ну и получим объект в невалидном состоянии. Что нам нужно, чтобы конструктор выполнялся аттомарно? Переменную helper сделать volatile – тогда конструктор будет атомарным. В реальности в данном конкретном случае можно использовать **final**: они точно являются **volatile**. Надо только понимать, что **final** запрещает менять дальше.

Есть же ещё один вариант singleton'а, с неленивой инициализацией, — он многопоточный. Но тут мы не можем нормально отреагировать на исключение конструктора — не обернёшь же инициализацию поля в **try** `cathch`. Можно написать статическую секцию инициализации, и уже внутри неё **try-catch**-секцию. А можно и просто статический для инициализации поля.

12.2. Блокировки и условия

Мы не можем брать synchronized в одном методе, а отдавать в другом. И ещё его не запихать в if для проверки, занято или нет.

Специально для нас в java.utils.Concurrent есть интерфейс `Lock`.

- lock()
- lockInterruptibly() – если вам очень надо, а блокировка занята, то отберёт у другого потока блокировку и вызовет у негоinterrupt
- tryLock()
- unlock()

Но это интерфейс. А нам нужна реализация — это ReentrantLock. isFair() — ”честность”блокировки: честная – это когда первым lock получает тот, кто первым за ним обратился. Это работает сильно медленнее, чем нечестный lock.

Разница между абстрактными классами и интерфейсамми стёрлась. Осталось, пожалуй, только то, что интерфейсы не могут хранить состояние (у них все поля – это public static final)

Если есть compose, то первой применяется функция, переданная параметром: T(B(x)) Статическим метод быть не может, так как юзаем apply, а реализация ну хоть ты тресни практически всегда одинаковая. Теперь это не беда – запишем метод в интерфейс с модификаором default

Что делать, если мы реализуем несколько интерфейсов и у них есть одинаковые методы по умолчанию. Становится непонятно, какой метод использовать. Тогда такой код не компилируется – Java это гарантирует. Как с этим бороться? Примерно так же, как в C++ – явно указать, какой метод изволите использовать. Написать руками.

Функциональный интерфейс — ровно(!) один метод можно/нужно реализовывать ручками. Абстрактные классы с ровно одним абстрактым методом тоже считаются функциональными интерфейсами ф этой идеологии.

Зачем нам все это? Мы хотим лямбды.

В Java[8] это просто синтаксический сахар: чтобы не городить много ненужного синтаксиса мы реализуем сразу метод и покороче. Прикомпиляции автоматически оборачивается в создание нового класса и т.д., и т.п. А если в методе ещё и команда одна, то — ооооо! — можно так и вообще фигурные скобочки не писать!

Если параметры в лямбде — ничего, то надо написать пустые скобочки.

Ссылки на методы: если захотелось уже готовый метод куда-то передать.

Можно пробежаться по списку и посоздавать себе юзеров. А так ссылка на конструктор обычно редко нужна.

Рекомендуется уметься решать: только с помощью stream’ов выдать по файлу первые 10 слов (по количеству вхождений) с приписанными количествами

Лямбды являются замыканиями: мы когда их используем, то обращаемся к чему-то вне нашего контекста. Лямбда неявно тащит вместе с собой ссылки на все внешние объекты, которые в ней используются.Effective final — это название технологии. Если объект такой, то — плюшка восьмой Java — final можно было не писать.

Примитивные типы до 8 Java всегда хранились на стеке. А теперь из-за замыканий, если какой-то несчастный int потребовался лямбде, то будет сохранён в куче — просто потому что все необходимые ей для замыкания элементы лямбда хранит в куче.

- **optional** – потому что нет указателей Optional<String> o = ... String t = o.orElse(”test”);

  аналогичнго

  String t; if(x == null) t = ”test”; else t = x;

- **Сравнения** .thenComparing

  аналогично

  (Student o1, Student o2) if(o1.getAge() == o2.getAge()) return o1.getName().compareTo(o2.getName()) if(... > ...) return 1; return -1;

12.3. Stream

Это `generic`-классы. ”Стримы”.

Операции порождающие(конвейерные: порождают поток), завершающие(выдают значение, а не поток; почти все промеж операции – ленивые. вычисляют ровно столько элементов потока, сколько от них попросит след операция

если мы хотим findFirst, то все предыдущие сделают ровно столько, сколько ей, последней, потребуется)

12.4. Разбор летучки

- A a = null; try a = new A(); finally if(a != null) a.close();//но если в finally выкинется исключение, то оно забьёт первое, выкинутое в try. //Это плохо, try-with-resources делает не так: он выбрасывает первое, а второе делает как deprecated //чтобы ссылка не сохранялась

  надо так:

  Throwable t = null; try A a = null; try a = new A(); ... catch (Throwable e) t = e; throw e; finally ... catch (Throwable e) if(e != t) t.addSuppressed(e); throw t;

- Inner и Nested/ Все внутренние невяно внутри себя сохраняют ссылку на экземпляр внешнего класса. => имеют доступ к полям внешнего класса, в т.ч. и приватным Исп в ситуациях, когда нужен класс, имеющий доступ к внутреннему содержимому класса(например, итератор) Вложенные – не имеют ссылки, поэтому джоступ только к static (в т.ч.и private) тестирование и сокрытие кода(например, в BST класс node – мы не хотим, чтобы им кто-то пользовалсядругая ситуаци: несколько классов с итерфейсом итератор(ну не итератор, но что-нибудь типа того, чего нужно несколько штук, но чтобы не нужен был доступ во внешний класс)

- lock vs synchronized: 1. много условий condition vs один wait 2. получение блокировки по набору объектов3. брать и отдавать блокировку не в одном и том же куске кода 4. можно проверить, можно ли взять блокировку 5. можно насильно захватить блокировку и, что важно, есть разные политики выдачи блокировки: read-write & Co

- equals – есть в презентации

- Error – ошибка джава-машины. Единственный содерж способ обработки – писание в лог. Exception обязательно нужно обрабатывать; вызван пользователем – надо обрабатывать(файл не открылся) RuntimeException – необяз; вызван программистом(его ошибка: nullPointerException)

- volatile – любое изменение переменной приводит к автоматическому изменению её в общей памяти(медленно). Атомарность некоторых операций(например, конструктора: не volatile сначала создаётся ссылка, потом вызывается инкремент-декремент для volatile примитивных типов)

- ? super Apple: классть в него любые Apple и наследников, доставать только Object ? extends Apple: класть только null, потому что там можнт быть несколько слоёв наслдования и мы случайно моем положить более верхние в список более нижних

12.5. Команды над stream’ами

Стримы бывают двух типов: объектов и примитивных типов. Методы с ними бывают тоже двух типов: терминальные/конечные и промежуточные.

Промежуточные – возвращают всё ещё стрим. filter, skip(пропустить N первых элементов), distinct, map(тип может поменяться;ыункция не должна возвращать void; кладёт результат работы функции, которую выполнил), peek(берёт объект, выполняет функцию, кладёт объект(ту же самую ссылку) обратно), limit, sorted, если хотим получить конвенйер примитивных типов, то не можем использовать обычным map – исп mapToInt и ко, flatMap принимает команду, которая из объекта делает поток

Терминальные операции findFirst – берёт первое яблоко, подходящее под условие и останавливает контейнер; @return Optional(внутри – null, если ничего не нашлось) findAny – @returns Optional; отличается от первого поведением на параллельных потоках collect – собирает объекты из потока и что-то с ними делает count – min, max – без параметра – естественным образом, с параметром – надо передать функцию сравнения? forEach – аналог peek, но она терминальная. Не гарантирует порядок ыполнения команд в том порядке, в котором элеименты лежат в стриме, при паралдлельном испольнении. Для этого есть forEach

12.6. Пример

Сумма квадратов чисел, оканч на 5

Второй пример. Есть словарь. Есть к нему буфферед реадер. Есть ещё пафс, путь в файлово системе. list vs walk: walk – рекуурсивный. .chars() устрочки – поток символов(типа Int*thumb up*)

Третий пример. Генерируем случайные числа. closedRange включает правую границу тоже. concat доклеивает второй в конец первого Потоки можно делать из массивов или просто из нгабора элементов

Пример сложнее Из всех цифр, которые встречаются в числах, больших ста, берём четвёртую и пятую.

Для потока int'ов findFirst вернёт первый ненулевой

В min передаём компаратор в новом виде: comparing(как получить то, по чему сравнивать; то, чем сравнивать)

reduce – левая свёртка

Слудеющий пример считает факториал числа. Но он большой и не влезает в Int'ы, поэтому нам приходится джелать BigInteger. Но чтобы 'nj преобразовать, мы делаем mapToObj/ В чем прелесть? Можно же и с помощью for'а написать. Ответ: как только напишем слово parallel, это станет быстрее – выяисление факториала можно распараллелить.

Последние два примера. Или три. Проверка на то, является ли строчка хитрым палиндромом: выкидываем все небуквы и нецифры, toLowerCase'им их и поверяем. Зачем потоки? код поэлегантнее. appendCodePoint – это чтобы добавлять к stringBuilder'у не строчку, а символ.

Обратите внимание: порядок, в котором мы пишем операции, может существенно влиять на производительность.(попробуйте поперемещать sort туда-сюда)

А вот теперь действительно последние два примера. filter не обязательно ничего не делает. Обратите внимание на порядок выполнения. Элементы последовательно проходят черезх весь конвейрер. Это хорошо, потому что обработается только то реальное количество элементов, которые нужны. Ну, sorted да, sorted выковыривает – ну да с ним всё и так хитро

Почему, когда мы поменяли местами filter В sorted, у нас исчез вывод команды sorted? Потому тчо один элемент остался.