Java

Осень 2015, лектор: Полон Любви-И-Обожания

Авторы: Лиза Третьякова, Настя Старкова

Собрано: 31 августа 2016 г. 17:59

Оглавление

Зачёт с оценкой оценка = : 1. дз (0 .. 5) 2. кр (0 .. 5) 3. летучки (0 .. 5) 4. устный зачёт (2 .. 5) 5. антибонус с прошлого семестра (-1) (нечто лично-индивидуальное) 6. проект по Java (-2 .. +1) По всем пунктам не ниже тройки

Домашки будут практически каждое занятие Кр будут на практиках как обычно Летучки на лекциях относительно регулярно, некоторые с предупреждением, некоторые – без. Это намеренно, чтобы мы не спали вместо лекций.

Проект. Для сдачи нужно: – *ОЦЕНИВАЕТ АМ* доделать необходимый функционал(-1(не додел), 0(достаточно), 0.75(сделано больше, чем надо), оценивает АМ) – *ОЦЕНИВАЕТ ЧЕЛОВЕК* пройти код ревью (смотреть на sewiki; -1(не пройдено), 0(удовлетворительно), 0.5(порадовать руководителя)) Оценка – сумма оценок, каждаЯ не больше +1 Всё это надо сделать до 13 марта. 13 марта ревьюер выставляет итоговую оценку.

А в этом семестре нас ждут: Про стримы, Network in Java, Java GUI(JavaFX/Swing), Java New Input Output(- or 2), Annotations, JEE elements

Практики начнутся с разговоров о многопоточности. С момента сети уже состыкуются с лекциями.

Курс про сети будет отдельный, курс по JavaEE будет отдельный на 5 курсе, поэтому про них обзорно.

0.1. Collectors (Java8)

В стримах: куча объектов, обрабатываются потоковым образом, они ленивые; конвейер, по которому ползут элементы, операции бывают промежуточные(взяли элемент, обработали, вернули что-то, возможно другое) и терминальные(после них конфейер заканчивается). самые интересные операции: filter, map, flatMap.

- **Редукция** Пример – сумма. Результат – число. Начальное значение – ноль. При получении new элемента его н. прибавить к результату. Правило объединения двух частичных результатов в один.(нужно при распараллеливании)

  3 операции-правила надо задавать: определение контецнера, к-ый б. содержать итоговый результат; добавление элемента к контейнеру; совмещение двух частично наполненных контейнеров.

  Редукция основана на трёх операциях:

- Constructor: Supplier. Пример: `() -> new StringBuffer(); StringBuffer::new;`
- Accumulator: Fuction. `(StringBuffer sb, String s) -> sb.append(s); StringBuffer::append;` 2 аргумента, 1 функция: от первого параметра вызывается функция, все остальные параметры передаются ей в качестве аргументов
- Combiner: Function. `(StringBuffer sb1, StringBuffer sb2) -> sb1.append(sb2); StringBuffer`

Пример 0.1.1. Person.

```
1 //Collectors.joining(); -- синтаксический сахар, чтобы не писать вездесущий "new-append-ap
```

joining склеит все имеющееся без пробелов, с одним аргументом – разделитель, второй – приделывается в начало, третий – приделывается в конец.

Интерфейссс `Collector<T, A, R>` параметризуется тремя типами `Supplier<A> supplier BiConsumer<A, BinaryOperator<A> combiner Function<A, R> finisher` стрим типа T, хотим результат типа R, для этого заводим контейнер типа A и с помощью финишёра преобразуем его в R как-то Часто вместо A ставят ?, потому что он невидим пользователю, например. Наприемр, метод getCollector.

Возвращаемый тип – Collector<тип начала, пользователю_неважно_что, тип конечного результата>.

<Пример2> Constructor: () -> new ArrayList(); `ArrayList::new`; Acc: (list, el) -> list.add(el); `Collection::add`; Comb: (list1, list2) -> list1.addAll(list2); `Collection::addAll`; Это Collectors.toList(); Можно ArrayList заменить на HashSet – получим список уникальных фамилий Приветствуем Collectors.toSet(); А что-то они одинаковые – Collectors.toCollection(TreeSet::new); TreeSet – отсортированы. В Collectors есть 33 статических метода для всего, что обычно нужно. Познакомимся с groupingBy Collectors.groupingBy(Person::getAge); Определяет, что из чего отображается во что.

```
public static <T, K> //парам 2мя типами
Collector<T, ?, Map<K, List<T>>> groupingBy (
    Function<? super T, ? extends K> classifier
)
//по объекту получаем ключик джля него и под этим ключиком кладём в мапу
```

Но groupingBy умеет принимать больше, чем один параметр – два параметра.Мы же можем хотеть не список, а что-то другое – вот и второй параметр. Мап из возраста в, например, кол-во людей с таким возрастом. Так что давайте передадим вторым параметром ещё одному коллектор. В ашем конкретном примере – Collectors.counting(). Он превратит наши объекты ещё раз в стрим и из редуцирует во тчо-то новое. Коллектор обрабатывает список элементов, соответствующих конкретному ключу.

Из-за .counting() у нас результат в лонге – а если мы хотим int? Напишем Collectors.summingInt(e -> 1);//здесь 'e' – это person[для колектора, обрабатывающего объекты с одинаковым ключом].

Если хотим не список Person'ов, а список только их имён Collectors.mapping(Person::getLastName)

А теперь склеим все эти имена Collectors.mapping(//1st downstream collector Person::getLastName, Collectos.joining(") //2nd collector(ну, третий))

И самое главное: **всё это нагромождение работает ленивым образом, то есть каждый элемент пропускается через все возможные коллекторы сразу**.

В список фамилий, отсортированных по алфавиту.

0.1.1. Задачка

И-и-иии.. снова здравствуйте! Файл. Из него выдат 10 самых часто встречающихся там слов, отсортированных в порядке частоты встречаемости. Одной строчкой.

```
Files.lines(Paths.get("input.txt")).
map(s -> s.split("\\s+")). //стрим массивов слов
flatMap(Arrays::stream). //стрим слов
filter(s -> !s.isEmpty). //отсеили пустые слова-строчки
collect(Collectors.groupingBy(Function.identity()), Collectors.counting()). //мапа из слов
entrySet(). //сет из пар...
stream(). //... из которых мы сделали стрим
sorted((e2, e1) -> Long.compare(e1.getValue(), e2.getValue())). //отсортировали
limit(10). //оставили 10
//forEach(System.out::println) -- вывели и слова, и встречаемость
forEach(0 -> System.out.println(o.getKey()));//вывели только слова
```

Без стримов занимает не то чтобы меньше места. А работает если и быстрее, то только чуть-чуть. Стримы не сильно быстрее, потому что много обёрток – раз, наш пример довольно большой и существенное время на чтение уходит – два, и – три – при добавлении параллельных стримов всем станет хорошо. Основное преимущество стримов в том, что их проще осмыслить – это раз. А два – вот как раз параллельные стримы. Суббота: Экономика Java ОС ОС

В среду, после двух пар по формальным языкам – медосмотр.

0.2. Fork/Join

Fork – вилка, Join – присоединение. Идея "Разделяй и властвуй": решать маленькие подзадачи, из этих двух решений получить полное. И это хорошо работает, пока подзадачи не зависят друг от друга. Иначе мы будем тратить много времени и сил на разрешение зависимостей. Задача -> split на несколько -> fork[начались разные потоки] -> join[кончились раздельные потоки] -> merge обратно. Но это упрощённо: никто не говорил, что не бывает задач, которые надо делить не один раз, а больше. До какого момента делить? Пока подзадачи не становятся такого размера, чтобы их можно было быстро решить.

```
if(problem is small) {
    directly solve problem
} else {
    split problem into independent tasks
    fork new subtasks to solve each part
    join all subtasks
    compose result from subtasks
}
```

проект написан одним человеком → много претензий fork: был один процесс, мы его расклонировали на два(прям с копированием стека, у них идентичная(по содержанию) в начальный момент, но раздельная память). Здесь слово fork значит, что мы должны отпочковать нашу подзадачу, то есть разрешить ей выполняться в другом потоке. Здесь join ждет, пока задачка закончится, а потом возвращает нам результаты её выполнения.

Почему нельзя сделать так, чтобы каждая новая задача выполнялась в своём новом потоке? Представим себе двоичное дерево, в каждой ноде – число, хотим посчитать сумму чисел во всех вершинах. Поддерево задаётся корнем, подзадача – посчитать сумму в левом и правом поддереве. Когда мы будем стоять в листе, у нас будет куча потоков будет также запущено. То есть в худшем случае у нас будет столько потоков, сколько всего маленьких подзадач. Для n-ичного дерева получится $\frac{N^{L+1}-1}{N-1}$. Переключаться между ними будет долго, особенно если листьев много – в общем, работает "отвратительно"(c). Причём все потоки, соответствующие вершинам на пути от корня до данной ершине просто ожидают. ЭТо неэжффективно. Так вот идея состоит в том, чтобы join не занимала поток. Когда наш поток вызвал join у подзадачи, он не стоит, а выполняет другие подзадачи. Для этого мы и делали fork таким, чтобы задачи вычислялись независимо.

0.2.1. Классы

- ForkJoinClass: задачка должна уметь делиться и уметь обрабатываться нашей системой work-stealing – вместо ожидания тащим в рот всякую... утаскиваем себе несделанную задачку и сами делаем
- RecursiveAction – no returning value RecursiveTask – there is a returning value Аналогичны Runnable и Callable (пользуемся ими)
- ForkJoinPool – в нём вся логика распределения задач между потоками. execute не возвращает результат, invoke – возвращает

0.2.2. Балансировка задач

– какому потоку задавать решать какую задачку. Подвид задачи о расписании. Три подхода:

- арбитраж задач fork – добавлдение адач в очередь join потом берёт себе задачку из очереди У каждого потока есть ещё своя маленькая очередь, если там место кончается, он выкидывает задачку в общую очередь (это решает проблему, чтобы поток не лазил своими лапками в чужие задачки)

  Такой подход не гарантирует, что порядок выполнения будет наиболее правильным. В чем узкое место? Вариант с общей очередью требует от нас синхронизации у потоков, чтобы операции добавления и забирания задач были синхзронизованными – это узкое место

- work-dealing своя собственная очередь задач, но если она вдруг кончилась, то он отдаёт эту задачу другому потоку. Плох тем, что легко могут получиться перегруженные(почти полная очередь, но не переполнилась) и недогруженные потоки

  Это тупой варианит, его можно улучшить, чтобы нагшрузка была равномерной

- work-stealing У каждого своя очередь, но если он освободился, то тырит чужие задачки

в FJPool'е используется именно stealing. Ограничения:

- - с головой очереди может работать только владелец
- - из хвоста могут брать другие потоки

  задачи, положенные недавно, маленькие, поэтому их разумно скармливать другим потокам, чтобы они их щёлкали и дальше шли, поэтому они и берут из хвоста.

- - владелей всё равно может брать из хвоста Когда это может быть осмысленно? Асинхронный режим

Куда происходит submit внешних задач? Хочется поближе к началу очереди, потому что чем раньше мы разобьём, тем раньше у нас появится большее количество маленьких подзадач В общем, чтобы это делать, для внешних задач заводится отдельная очередь, у неё приоритет чуть выше, чем у старой очереди (но у подзадач всё авно есть свои приоритеты, чтобы мы не забивали на старые совсем)

Другое решение: случайным образом достаём/добавляем задачи в ту или иную очередь(одна – исключительно для внешних задач).

0.2.3. Демонстраци и сеанс чёрной магии

```
public interface Node {
    List<Node> getChildren();

    long
}
//базовая реализация строит дерево глуюбины 22
//первое, что можэно менять для ускорения:
то, с какого момента задачку считать маленькой;
форк и join задачек;
порядок создания и собирания результатов задач.
Цель: сделать так, чтобы работало быстрее, чем однопоточная версия
```

0.3. Parallel Streams

The awful true is that parallel streams works as frok-join pools. Все операции над параллельными стримами выполняются в этом FJP. И он один вообще один, на все параллельные стримы – то есть если у нас есть два, то они юзают один пул. Это плохо. Чтобы изюавиться от этой проблемы, пошли на хитрость: мы просили 4 потока, а они хадействовали пять. Это для тогоЮ, чтобы наш вызвавший поток – тоже вполне хороший и рабочий для этого FJP. Чтобы для каждой задачи был хотя бы один швой собственный поток.

Следствие: все операции над параллельными потоками – синхронные.

Проблема: при одновременной работе нескльких параллел стримов может категорически проседать производительность. Мы хотим, чтобы задачи с параллельным стримом использовали свой FJP

Проблемы общего FJP:

- не дай боже ввод-вывод – это не join, оно стоит и ничего не работает
- вложенные параллельные потоки – это долго
- boxing/unboxing – с ним примерно в два раза медленнее

Глава 1

Дивертисмент: ликбез про сети

Книжка: TCP/IP Illustrated Vol. 1 (1-2 chapters)

Чтобы отправить куда-то пакет — данные — нужно знать его адрес — IP-адрес. Состоит из четырёх байт (IPv4) — четырёх чисел: XXX.XXX.XXX.XXX Соответственно, у каждого компьютера должен быть свой IP-адрес, если он хочет работать в сети. Но давайте считать: порядка четырёх миллиардов адресов, при семи миллиардах человек, а на каждого человека ещё и не по одному компу, а по 2.5. Поэтому мы мечтаем перейти на IPv6, чтобы было побольше адресов (эти-то уже кончились давно, хотя планировались уникальными), но с ним как всегда: он почти везде поддерживается и почти везде не поддерживается.

Посмотрим:

```
10.*.*.*
192.168.*.*
127.*.*.*
172.X.*.*
```

эти IP-адреса запрещены для испольщования в интернете. Это сделано для локальных сетей: у почти всех компов в ней невалидные для интернета адреса и есть набор валидных, несколько штук. Так что для внешнего мира это всего те несколько компьютеров, через которые все остальные машины в их локальной сети выходят в интернет.

Пример 1.0.1. У студента Р. адрес есть, но невалидный, у `vk.com` есть нормальный адрес. *(это, как всегда, не совсем правда)* Как студенту Р. отправить данные во внешний мир? Можно отправить на точку доступа, а потом от ней — дальше. А как не во внешний мир, а, например, другому студенту — Г.? Кажется, что можно напрямую, если они в одной локальной сети. Принципиально есть два варианта: напрямую к другому пользователю или через место, через которое выходим в интернет.

Это место — точка доступа — нужна в том числе для безопасности.

Чтобы отправлять данные напрямую, вводится локальная сеть. Чтобы понять, что два устройства лежат в одной локальной сети, вводится пониятие маски сети. Два варианта записи:

- `192.168.65.13/24` (первые 24 единички)
- `255.255.254.0` (23 единички)

Компы находятся в одной локальной сети, если у них биты, соответствующие единичным битам маски, совпадают. `M & IP1 === M & IP2` (можно проксорить)

И ещё одна вещь — gateway — шлюз — IP-адрес компьютера в нашей(это обязательно!) локальной сети, через который будет осуществлено взаимодействие со внешним миром. Откуда шлюз знает, куда отправлять пакеты дальше? В самом простом варианте у шлюза есть такая же штука: он есть в ещё какой-то локальной сети (например, wi-fi и провод). И он понимает, что если мы

шлём на комп из какой-то его локальной сети, то напрямую, а иначе у него у самого есть шлюз, на который он дальше пересылает.

Одна проблема: студент Р. написал `vk.com`, а не `72.47.10.9` Чтобы это работало, есть специальная штука — `DNS`, `domain name service`. Это такая служба, основной задачей которой является ответ на вопрос: какой IP-адрес у компа с указанным именем. Но это надо у кого-то спросить, чтобы узнать — у extttDNS-server'а. Он необязательно лежит с нами в одной сети, но его IP-адрес нам надо щнать. Например, у `DNS-server`'а гугла адрес `8.8.8.8`

Т.о. сначала студент Р. отправил запрос к `DNS-server`'у, получает адрес `vk.com`, отправляет данные напрямую/через шлюс(зависит от локальности)

Как работает DNS? Ответ на вопрос: либо есть в кеше, либо спрашиваем у вышестоящего DNS-сервера. Есть корневые `DNS-server`'а, их 13 штук + резервные — что они знают? Довольно мало, он знает, какой комп знает всё про домены первого уровня `.ru`, `.com` Он, в свою очередь, знает, какой компьютер знает что-то про зону `vk.ru`(домен второго уровня)

Две модели общения при общении с `DNS-server`'ами:

1. Плюсы "по уепочке обратно": все закешировали ответ. Минусы: трафик.

2. Плюсы "обращаться напрямую к следующим": полегче с трафиком.

Раз в год проподится `DNS hack-day`.

Что сделать, чтобы завести "`vasya.com`": есть организация, отвечающая за домен "`.com`", ей надо заплатить, чтобы она узнала о том, что именно твой комп за этот домен отвечает. Ну, организация такая не одна: есть VeriSign — американская, есть InterNIC — кому она принадлежит? Американскому правительству, 50% акций + одна.

Если очень хочется пройти мимо провайдера — "хочу напрямую к облачку": есть точка подключения (штук 13 в Питере). За подклоючение к ним провайдеры платят денежку. Есть длинный кабель, втыкающийся в эти точки — большие маршрутизаторы, они принадлежат частным компаниям (RunNet, Ростелеком, ещё кто-то). У нас есть провод в Москву, в Хельсинки, спутниковая тарелка на ИТМО. Есть кабели по дну Атлантического океана, тарелки в Норвегии по тысяче терабит в секунду. Антенны в США. Кабели между материками. Кабели оптоволоконные.

На обоих концах этого кабеля стоит маршрутизатор и принадлежит США.

Любая страна — это собственник маршрутизаторов, стоящих на её границе. Великий китайский `firewall`, минуя их сервера выходить в интернет — в Китае это тстуденту Р.мный срок.

Возвращаемся к студенту Р. и `vk.com` Как-то чудно студент Р. получил ответ, пока чудно. Проблема: у студента Р. в браузере не одна вкладка.

Порт — это не идентификатор компьютера, это идентификатор [сетевого соединения нашего] приложения уже на нашем компьютере. То есть студент Р. невяно пишет `vk:com:80`. Из порта данные передаются в соответствующую вкладку.

А теперь о том, как студент Р. выходит с запрещённым IP-адресом в интернет. Студент Р. пишет запрос к `vk.com` на порт 80. В запросе от студента Р. написан его IP-адрес, порт его браузера для ответа, написан IP-адрес вконтакте и порт на вконтакте, куда студенту надо. Пакет с этими данными доходит до точки доступа. Если дальше начался интернет, то дальше этот пакет пропускать нельзя — там адрес плохой. Поэтому точка доступа делает преобразование и заменяет пакет на другой: IP-адрес точки доступа, порт на ней, тот же адрес и порт вконтактика. А кому передать ответ, точка доступа запомнит. Преобразование, которое сделала точка доступа — это NAT: network address translation. Резонный вопрос: сколько точка доступа держит? Сколько портов: от 0 до $(2^16 - 1) \to$ точка доступа не может одновременно поддержать больше 65K соединений. (на самом деле сильно меньше: не все порты для этого + памяти-то у точки доступа не так много) Запись обычно живёт не больше несколькихз десятков секунд: во-первых, вряд ли тебе уже через так долго ответят, во-вторых, записи вытесняются. Это причина, почему во многих общественных местах запрещены торренты: торренты открывают огромное количество соединений, точка доступа не выдержит.

<картинка с табличкой тут>

- В Application — это протоколы, на которых общаются друг с другом наши приложения. http — общение страничек. ssh — удалённый терминал. ...
- В Link — наши устройства физически общаются друг с другом: по беленькому проводу, по ви-фи.
- 2 промежуточных уровня.

  Зачем все эти 4 уровня сделаны? Пишем хром. Нам хочется не думать о том, как хардварно работает наш пользователь. Я приложение, я хочу абстрагироваться от непосредственной передачи данных по физическому носителю.
- IP-уровень. Здесь появляется абстракция "IP-адрес не зависящий от способа физической передачи данных по сети.
- Transport — вводится абстракция "порт". Чтобы приложения на одном IP-адресе разделядлись и получали свои данные.
- Уровень приложений — только наше приложение знает, как оно непосредственно передает данные.

И вся эта штука называется стек протоколов TCP/IP.

Почему стек? Приолжение хочет отправить данные → говорит компу "отправь туда-то"→ TCP добавляет в пакет шапочку(с нашим адресом и портами отправителя и получателя) и хвостик(контрольн сумма) → данные передаются на уровень ниже и запаковываются в пакет уровня "IP"→ в этом пакете приписывается IP-адрес отправителя и получателя → в зависимости от техноолоиии заворачивается в пакет link-level'a(пусть Ethernet) и отправляется в сеть.

Пусть эти два компа находились в одной локальной локальной сети. На линк уровень нам пришёл пакет. Но часто нам приходжит всё подряд. Поэтому вводится ещё и mac-адрес(вписывается на линк-уровне), адрес сетевой платы. По получении в первую очередь смотрим на мак-адрес и, если он наш, разворачиваем и передаём выше. Там проверяется, тот ли IP адрес. Если да, то разворачиваем и передаём на уровень выше. Посмотрели на порт, определили приложение, развернули, передали. Приложение может спросить, кто отправил mac, IP, ...

Вопрос: а mac-адрес — это не лишнее? Нет.

- Во-первых, он специфичен для Ethernet.
- Во-вторых, работаем чеерз шлюз. Если сошёлся мак-адрес, но не IP, то шлюз *заворачивает обратнро* и передает следующему шлюзу.

Если IP не интернетный ??? Разворачивание пакетов туда-сюда — довольно дорогая операция. 2 задача: threadPool: t.join(); – Thread t; – поэтому join запущен у потока, другой только ждать (а, а ещё это ничего не возвращает, а это v возвращает результат выполнения задачи)

FJP: t.join(); – RecursiveTask t; – join запущщен у задачи, поэтому поток работает

Задачки априори делящиеся, делятся и распределяются между потоками в пуле

3 задача: 1. любая задача такого вида должна начит=наться с проверки того, что она маленькая 2. разбить на кусочки, форкнуть каждый кусочек и дождаться выполнения

Не надо писать так: for(по всем подзадачам) t.fork(); t.join(); – неэффективно загрузились потоки: у нас процессится только одна задачка – насколько возможно, параллельно, но, тем не менее, большое количество веток простаивает, потому что ещё не форкнулось

4 задача: дебильные ошибки: -спутать параметры: первым – инициализация, то есть точечку создать надо было, например: Point::new
() -> return new Point(0, 0, 0) потом – combiner: (p1, p2) -> new Point(...)

-ошибка в формуле для центра масс
1 задача:

```
List<Integer> primes = new ArrayList<>();

IntStream.iterate(2, i -> i + 1).filter(i -> {
        for(int prime: primes) {
            if(i % prime == 0) {
                return false;
            }
        }
        return true;
    }).limit(100).forEach(primes::add);

//но не параллельно

//числа идут слева направо, проходят через ситечко уже нацеженных простых чисел
//и оседают в нём только в том случае, если ни на кого не делятся
```

List<Integer> → IntStream //(всегда; чтобы работало быстрее; это мы чтобы один раз unboxing сделать, а не туда-сюда разворачивать)

Обычный filter с проверкой на простоту, просто parallelStream – не меняется от этого **ничего**.

5 радачка: 1 способ: try-catch 2 способ: sc = new Scanner(str) sc.hasNextInt()

это как в "количестве уникальных слов"с пары Только вместо Identity оставляем лямбду, оставляющую от строчки первый символ

1.1. Возвращаемся к сетям

Напоминание: человек, роутер -> цепочка маршрутизаторов общение в интернете по http запрос на vk.com:80

app http transp tcp, udp формируем dns-запрос нашему dns-серверу, узнали IP-адрес vk network IP смотрится, павда ли 2 компьютера в одной локальной сети. Применяет маску, если нет, то напрямую нельзя – надо в шлюз(gate) link Ethernet mac-адрес шлюза, mac-адрес себя

Прилетел пакетик на шлюз. Мой пакетик? нет => выкидываем. да => передаём на уровень выше

UDP не гарантирует доставку данных, TCP – гарантирует

2 генерала стоят на двух холмах рядом с непристун=пной крепостью За каждым армия Но нужно обязательно обе армии напасть Они хотят договориться о времени штурма Связь? Лазутчики – но через город man-in-the-middle Ответа нет, невозможно договориться, если нет знания, которым обладают эти и только эти двое.

К чему задача? Нет способа гарантировать доставку данных без внешнего знания. Чем поможет это знание? С его помощью можно шифровать наши данные. От недоставки мы не застрахованы, но пошлём трёх человек с одинаковым сообщением – там разберутся.

То есть TCP ничего не гарантирует.

Зачем же придумали UDP? Например, передача видео по сети: если по дороге пропал один кадр – это фигня, это лучше, чем если доставятся все кадры, но с задержкой. Вторй вариант – торренты. Вместо того, чтобы скачивать файл из одного места, мы получаем табличку с пользователями, которые уже скачали, и уже у них по небольшому кусочку тырим. Там можно повторить запрос.

TCP – засчёт чего гарантии? Дополнительные вопросы-ответы. Каждый пакет нумеруется. 8 9 11... – тут мы просим переотправить нам 10. Если так и не получается, то шлём извиннения и

рвём соединение. То есть он очень медленный, а особенно в ситуации плохой среды – на несколько порядкой медленнее, чем udp. Также этот протокол поддерживает соединение. Официальненько Устанавливаем Соединение, периодически осведомляемся о здоровье-с противоположной партии, не изволили ли они, паче чаяния, соединение прервать.

Network ICMP Internet Control Message Protocol Что это? Начнём издалека.

Как устроен пакет IP кровня Network: TTL – jed time, когда пакет проходит через маршрутизатор, делается –, и если ноль, то пакет убивается Зачем? Если админ криворукий и у него пакеты ходятпо кругу между двумя маршрутизаторами

На точке доступа можно сравнить TTL пакетов, которые приходят с человека и тогда понять, выходит через него кто-то в интернет или нет: если есть различные TTl, то выходят.

ICMP позволяет получать сообщения об ошибках при работе с пакетами. Если кому-то по TTL пришлось выбросить пакет, то он обратно отошлёт сообщение о том, что он выброшен. Чтобы TCP понимал, как там у него дела у пакетиков отправленных. Если IP-пакет потерялся, то он отправляет ICMP-пакет об ошибке. Если тот потерялся, то ничего страшного. Если пакеты не приходят – соед=инение разорвалось.

traceroute – позволяет узнать IP-адреса всех маршрутизаторов по дороге к какому-то адресу. Как работает: Можем послать пакет с TTL = 1 на куда-нибудь, куда надо, на первом же маршрутизаторе пакет умер, а нам вернулось сообщение с ошибкой и ... IP-адресом маршрутизатора :) Но не совсем корректно: нет гарантии, что наши пакеты будут всегда идти одинаковым маршрутом.

Есть 2 типа ICMP-пакета: для ping-запроса и ping-ответа, а ещё один – об ошибке.

ICMP-пакеты – большая нагрузка для сети.

Типичная задача ICMP – синхронизация времени.

Задачка: есть 2 компа со своим локальным временем. Как им синхронизоваться? Один ICMP-запрос, в который мы вкладываем наше время отправления 1, на втором компе записывает время полчения2, время отправления [обратно] 2, мы, получив обратно, приписываем время получения 1 t1, t2, t3, t4 условно Как по ним понять дельту, на которую синхронизоваться? Что такое $(t2-t1)$: $(t4-t1)-(t3-t2)$ – суммарное время в пути, поделим на два – примерно в одну сторону По идее, если мы идеально синхронизованы, то $t1 +$ время_в_одну_сторону $= t2$. Поскольку не равно, то отсюда и дельта.

1.2. Презентации на следующей неделе

- Что делали – описание проекта (идеального), постановка задачи,
- мотивация(зачем делали и что хотелись решать)
- как сделано – про модули, которые есть в приложении, и то, как оно взаимодействует; здесь же про интересные моменты, которые встретились при решении
- продемонстрировать – реальная демонстрация/скриншоты/ролик
- выводы: что хотели, что получилось, что не получилось, что нового узнали в рамках этого проекта
- не забывать ссылки на репозиторий и собранный apk

жестко 7 минут вопросы после 7 минут
Раньше были проблемы:

1. не укладывались во время (надо порепетировать)
2. порепетировать перед незнающим человеком – чтобы понятно рассказывать

3. не рекламная презентация (не продать проект), а техническая: то есть не без доли саморекламы, но надо технически описать задачу, проблемы, их решения, что получилось и что не получилось

4. Александр Владимирович плохо относится к котикам

5. лучше следовать этой схеме (данной выше)

Это бонус, то есть можно дополнительно получить [0..1]

1.3. Ответы на тест

1. (Для чего используется протокол ICMP? Назовите побольше примеров использования?) Отправляются маршрутизатором, который по какой-то причине не может передать дальше пакет. А ещё ping, trace-rout, синхронизация времени.

2. (Что такое ARP? Как работает?) ARP – resolution. Какой mac-адрес по данному ip-адеусу. Широкоыещательный пакет,отвечет только комп с нужным ipc-адресом

3. (Что делают 13 корневых DNS-серверов?) перенаправляют на dns-сервера, отвеч за соотв домен первого уровня

4. (Что такое TTL? Для чего нужен?) TTL – число от 0 до 255, обознач время жизни покате. 0 – уничтожается. При проходе через маршрутизатор уменьшается на единичку Чтобы не ходили по кругу.

5. (Что такое порт? Зачем нужен? Как выбирается?) Порт – число от 0 до 65***, идентификатор сетевого соединения, чтобы пришедшие данные нужному приложению отправлять. Выбирается одним из трех вариантов: либо случайно, либо Порт 80 соответствует протоколу http. В линуксе есть специальный файлик Третий вариант: если мы пишем веб-сервер(отвечающий на http-запросы), то он может ручками выбрать (но это обычно только у серверных приложений так). Клиенту обычно не нужно, потому что ему редко что шлют извне.

   до 1024 порта в линуксе – для служебных приложений (надо запускать с правами root'а)

1.4. Опять какие-то уровни

1.4.1.

Можно писать приложения, которые общаются примерно на любом уровне Поэтому дальнейший рассказ будет про уровень app.

1.4.2.

Значит, если мы хотим писать приложение для работы по сети, нам надо выбрать протокол. Посмотрим на TCP.

Пусть наше приложение работает на порте 2078, а там (_там_!) есть порт 80. Оба шлют на один и тот же порт данные.

1.4.3. UDP

Сокет – это просто некоторый API для передачи сообщений. Бывают не только сетевые, но и межпроцессные (на одном компе), есть связанные с файлами – а мы сегодня говорим всё-таки про сетевые. В нашем, то есть са Программная абстракция, представляющая собой сетевое соединение (а оно задаётся IP-адресом и портом клиента и IP-адресом и портом сервера). В Java8 есть TCP и UDP сокеты, а также поддерживающие IPv4/6

```
//это по UDP -- самое простое ghjot dctuj? endth;lftncz
try (DatagramSocket s = new DatagramSocket()) {
    DatagramPacket //один пакет, который хотим передатьЮ надо указать данные, их длину и у
    s.send()
//тут порт клинета за нас выбрала ОС случайно
//ОС привязала нас автоматически ко всем сетевым интерфейсам имеющимся
//при отправке будет указан IP адрес того сетевого интерфейса, по которому пакет де-факто

//если мы сказали "отправь", то сначала пытаемся отправить по сетевому интерфейсу, в одной
//такой. Если нет, то по приоритету оставшихся.
//у интерфейсов есть приоритеты (выше у тех, кто подключён к интернету)
}
//мы не задаём свой порт, так как мы не сервер
//мы не задаём свой IP, потому что система должна знать + у нас, имеется в виду, только од
//
```

```
DatagramSocket(port) -- сервер будет ждать на этом порте
создали буфер
создали пакет, в котором мы ща один receive прочитаем не больше, чем сказали длины (оставше
receive -- блокирующая команда (программа останавливается и ждёт, пока кто-нибудь не пришлё
```

Как сервер узнаёт, от кого пакет? У пакета есть метод, возвращающий адрес клиента-отправителя.

обычно сервер – пассивен, то есть для него нормально сидеть на команде receive В ничего не делать.

1.4.4. TCP

Socket – это TCP сокет в джаве

```
Socket socket = new Socket("localhost", 11111);
//хост и номер порта: хост можно писать строчкой,
//можно прямо IP-адресом, можно свлй адрес и  порт прописать --
//если у нас несколько сетевых соединений (и вряд ли мы клиент в таком случае);
//можно вообще ничего не указывать, а привязать потом
\end}javacode{
При создании сокета сразу устанавливается соединение, и если это не удалось, то конструктор
В чём отличие от UDP -- у нас есть соединение, по которому мы можем передавать данные. Зато
Поэтому здесь мы используем стримы.
Надо помнить, что чтобы быть уверенным, что данные по output-stream'у точно отправились, на
Правда, не очень хорошо всё время делать flush() -- ну, головой думать надо.
Также есть и InputStream'ы. Это было про клиента. У сервера это сложнее.
Оборачивать InputStream  в BufferedStreamReader -- это хороший тон, быстрее будет.

\begin{javacode}
```

```
ServerSocket server = new ServerSocket(11111);//не обычный, потому что новые порты делает
Socket socket = server.accept();//принимает клиента, создаёт ему новый порт, а нма даёт об

InpurStream is = socket.getInputStream();
is.read(requestBytes);

OutputStream os = socket,getOutputStream();
os.write(responseBytes);
os.flush();
//но этот код обработает только одного клиента -- мало

while(true) {
    accept a connection;
    deal with it;
}

//тоже плохо -- однопоточно, только один клиент за раз
//что делать?

while(true) {
    accept a connection;
    create a thread to deal with the client;
}
//тут создаётся можно потоков, поэтому их лучше переиспользовать
//хотя Apache работает так, он на каждого клиента новый процесс создаёт
//но правильнее всего работать так

while(true) {
    accept a connection;
    create task which will deal with the client;
}
//лучше всего тут будет cached thread pool (может создавать дополнительные и убивать лишни
```

Существует ещё принципиально другая идеалогия – на callback'ах

1.5. Java NIO

New Input/Output Начиная с Java 1.0 С 4 версии произошло несколько улучшений: буферы, неблокирующий ввод-вывод, селекторы.

1.5.1. Каналы

Канал – это сущность, из которой можно считывать данные в буфер/пистаь в буфер – и только буфер, никак иначе. От стримов отличаются двунаправленностью, буфером и наличием асинхронности.

<надо кусочек про буферы стырить у Насти>

flip() — используется для переключения с записи в чтение $limit \neq capacity$ ну разве что тогда, когда там в конце у нас что-то важное записано и нам затирать не хочется.

Чтобы использовать буфр, его надо получить – статическим методом allocate в классах буферов. read – записаь из канала в буфер put – положить в буфер руками

write – записать в канал get – получаем один элемент из буфера
класс RandomAccessFile – из Java7.
buffer.mark() ... buffer.reset() – вернулись (position) туда, где в последний раз оставили маячок
Как сравнить два буфера? 3 условия:

1. они одного типа
2. в них должно оставаться одинаковое количество элементов (от position до limit)
3. эти оставшиеся элементы должны быть одинаковыми

1.5.2. Scatter / Gather

Чтение из канала не в один буфер, а в несколько; и читать не из одного буфера, а из нескольких.

1.5.3. Селекторы

Для чего всё это делалось? Пусть хотим приложение на большое число клиентов. Если мы позаводили потоков на клиентов, то они 99% времени будет ждать У нас на каждого пользователя канал заведён. Давайте заведём сущности – селекторы – которые будут обслуживать эти каналы (типа клиентов) и говорить, кого уже можно пообрабатывать. И они будут нам находить каналы, которые к чему-то готовы.

Создаём селектор, опять же, статическим методом. Неблокирующий режим – вернётся сразу, независимо от того, удалось ли сделать, что хотели, или не удалось.

Селектор умеет работать только с каналами в неблокирующем режиме. Подключаемся с помощью метода register, вызванного на канале, которому передаётся селектор и bit of interest. `OP\_READ` – готов, если из него моэно читать. `OP\_WRITE` – готов, если в него можно писать. Если хотим сразу несоклько, то через 'или'. Возвращает объект SelectionKey interest set – чем мы интересуемся ready set – набор того, к чему канал готов в данный момент сам канал, сам селектор attached object (optional)

.readyOps() – к чему канал сейчас готов или .isAcceptable & C^o

Attached object – что обычно прицепляют? То, что позхволить как-то идентифицировать канал (имя, id, socket)

выбор канала с помощью

- .select() – блокирует
- .selsct(timeout) – до таймаута
- .selectNow() – отдаёт результат сразу, но, возможно, NULL.

Возвращает число интересных каналов(когда возвращается? кому возвращается?)
selectedKeys
после вызова select формируется set из готовых каналов

1.6. Сетевые каналы

Три типа:

1. SocketChannel (на клиенте просто для себя ручками, на сервере – для клиентского сокета создания) Всё, как обычно Только с селекторами-то работаем в неблокирующем режиме, поэтому немножко по-другому: после .connect() нужно в цикле вручную ожидать, пока не произойдёт finishConnect, то есть мы не поймаем клиента write() – её надо вызывать в цикле, ничего

необычного read() – должна что-то прочитать и завершиться сразу, чтобы не_блокировать, поэтому надо внимательно следить за тем, сколько она считала.

2. ServerSocketChannel В неблокирующем режиме снова фигня какая-то. .accept() должна завершиться сразу, поэтому если в этот конкретный момент к нам никто не стучится, то она вернёт NULL и всё

3. DatagramChannel – для UDP-пакетиков канал

О следующем разе: как написать неблокирующий сервер, используя это всё. То есть обработка всех клиентов будет в одном потоке. На каждого будет один канал, подвешенный к серверу. Будем из них читать данные, но не будем сразу их куда-то записывать, а будем ждать, пока можно начать записывать (для каждого) Основная проблема неблокирующего сервера в том, что считать весь пришедший сетевой пакет нормально не моэем – могло прийти нецелое число пакетов и т.п., притом что мы ограничены в размерах буфера.

Вообще написать неблокирующий сервер сложно, потому что надо одновременно думать о многих вещах. Обычно высоконагруженные вещи пишутся на чем-то смешенном, когда селекторы формируют задачки в пул потоков.

Будет половина занятия на архитектуру неблокирующего сервера, а дальше будет про Java UI.

Глава 2

Графический интерфейс

Раньше было три графических библиотеки, теперь — четыре. AWT, Swing, SWT, JavaFX

- **AWT** — была платформозависимой. Все рисовашки делались средствами операционной системы. С одной стороны, всё выглядело очень привычно пользователю. Но были проблемы: на кнопочку на некоторых платформах можно повесить картинку, а на других — приложение упадёт. Компоненты, задействующие операционную систему — тяжеловесные.
- **Swing** — платформонезависимая, основана на легковесных компонентах (всё отрисовывается джавой). Вам как пользователю ОС выглядит инородно, зато на всех платформах одинаково. Но если всё-таки хочется привычного, то были введены стили — похожие на ОС'ные стили.
- Легковесные компоненты в виде Swing'а — это, конечно, хорошо, но долго, тяжеловесные ОС-компоненты быстрее. **SWT**. Swing завязан на AWT, а SWT полностью была переписана. Но интерфейс оказался не очень удобным, поэтому все больше Swing'ом пользуются.

В реальности больших графических проектов на Java написано всего пара: IDEA да Eclipse. Потому что графика — это ресурсоёмко и тяжело, лучше что-нибудь пошустрее, чем Java, использовать.

JavaFX позиционировалась как замена Swing'у. Имеется в виду, что она делает то же самое, но лучше и удобнее. Входит в стандартный пакет, начиная с седьмой Java'ы. То есть изучать осмысленно, но со Swing'а начнём.

2.1. Swing

Создавался давным-давно, когда IDE не были такими клёвыми, как сейчас, когда в них нельзя было ручками элементы таскать — всё programmatically. Окна верхнего уровня:

- Окно приложения — класс JFrame
- Диалоговое окно — класс JDialog
- Окно апплета — JApplet (Это кто-то в браузере O_o)
- Вложенное окно, которые внутри другого двигать можно — InternalFrame

```
1 import javax.swing.*;
2
3 public class testFrame {
4     public static void main(String[] args) {
5         // это можно и в конструкторе класса делать
```

```
        /* конструктор какой-нибудь */
        myWindow.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        myWindow.setSize(300, 400);
        myWindow.setVisible(true); // потому что иначе пользователю будет виден весь проце
    }
}

SwingUtilities.invokeLater
                // принимает на вход задачу. которую надо будет выполнить потом в другом по
                // отрисовка происходит в отдельном потоке, потому что может быть долго
            (  // лямбда с задачей
           ()-> {JFrame myWindow = new SimpleWindow2(); myWindow.setVisible(true);}
        )
```

Создание окна == наполнение компонентами + отображение

Программа не закончит выполняться, если мы закрыли окно. Если мы всё-таки так хотим, то надо использовать EXIT_ON_CLOSE(JFrame). DISPOSE_ON_CLOSE HIDE_ON_CLOSE

Стандартные диалоги Есть некоторый предустановленный набор диалогов, которые можно использовать (типа стандартных "да/нет/отмена"и т.п.)

Из чего сотстоит окно верхнего уровня: (пять слоёв, поговорим про четыре)

1. ContentPane – все графические компоненты
2. LayeredPane – чтобы задавать z-координату (глубину) компоненту
3. GlassPane – чтобы реагировать на события мышки для всего окна (а не над каким-то конкретным компонентом); или "лоадинг"делать, чтобы скрыть, пока не готово
4. RootPane – на нём всё это живёт; чтобы все три слоя разом на какую-то другую тройку заменить

ContentPane можно весь целиком на другой заменить.

КНопочка – класс JButton. 2 способа добавить:

```
JButton newButton = new JButton(...);
getContentPane().add(newButton);

vs

//синт сахар над ^^^
add(newButton);
```

Контейнеры, в которые можно добавлять другие компоненты:

- JPanel
- JFrame
- JDialog
- JScrollPane

2.2. Компоновщики

Задают правила, по которым компоненты будут отображаться на нашей панельке.

Все компоновщики — реализации интерфейса ”LAyoutManager досталось Swing’y от AWT (равно как и JFrame, то есть само окошко – не его содержимое – отрисовывается не средствами джавы, а ОС”ными)

- **FlowLayout** Всё в строчечку выкладывает `alignment` – горизонтальное выравнивание
- **BorderLayout** Стоит по умолчанию Можно расположить в центре, на севере, на юге, западе, востоке.
- **GridLayout** Имеет определённое количество столюцов и строк, которые мы последовательно заполняем
- **BoxLayout** `.createHorizontal//VerticalBox` — создаются статическими методами, два типа, как видно. Компоненты набиваются, как в стек. Есть Strut(распорка) и Glue(клей).

`getContentPane().add(new JButton(`WEST”), BorderLayout.WEST);”Если несколько кнопочек в центре создать, то они друг на друга наложатся.

PreferredSize – кнопка умеет сама рассчитывать свой размер (исзодя, в основном, из размеров текста).

```
new GridLayout(# солбцов, # строк, зазор по горизонтали между компонентами, зазор по ве
```

`panel.setLayout(null);` – тогда позиции всех компонентов надо задавать самостоятельно.

Давайте теперь ещё и красиво научимся делать, а не только как-нибудь.

2.2.1. Обрамления

Класс `Border`. `EmptyBorder`, `LineBorder`, `EtchedBorder`, `BevelBorder`, `MatteBorder`.

JToggleButton JCheckBox, JRadioButton Это всё кнопки

2.2.2. Компоненты ввода текста

JTextField JPasswordField JTextArea

2.2.3. Панель прокрутки JScrollPane

```
JScrollPane(Component?, политика горизонтальной строки прокрутки, п~ вепртикальной с.п.)
```

2.2.4. Ещё есть

JToolBar , JComboBox выпадающий список, JSlider ползунок, JTabbedPane вкладочки, JList можно выбрать какое-то количество позиций, JProgressBar

Можно вставлять кусочки html’я.

2.3. Слушатели

Вешаются на кнопку, по какому-то событию реагируют и выполняют какие-то действия

`.addMouseListener(...)` или `MouseAdapter` — у него заглушки на все основные методы реализованы.

`ComponentListener` — смена положения/размера `ActionListener` — универсальный, он слушает ровно одно — главное — событие (для каждого компонента оно своё). То, что ровно одно событие,

позволяет нам использовать лямбду для описания обработчика. При этом туда будет передан параметр `ActionEvent`, который позволит нам немножко лучше понять, по какому конкретно поводу нас вызвали.

2.4. Action

Действие — абстракция действия, которое можно произвести. (чтобы написать один раз и пихать потом везде) Интерфейс Action. Единственное, что реально надо переопределять — `actionPerforme` (совершить действие). Если к кнопке привязан `Action`, то на ней будет показан `LONG_FDESCRIPTION`, если к менюшке — то `SHORT_DESCRIPTION`.

То, что на несколько элементов наложен один и тот же `Action`, означает, что эффекты для них для всех общие (как и всякие ресурсы, типа текстов и картинок)

2.5. Swing и потоки

Если наш Listener делает какое-то очень долгое действие, то графический интерфейс зависнет — поэтому его надо вынести в отдельный поток, чтобы всё не вешать.

invokeLater invokeAndWait

Так что изменения в визуальной части компонентов пишите, пожалуйста, через `SwingUtilities` (откуда, собственно, это `invoke`'и родом).

2.6. Что на джаве сейчас пишут?

В основном, бекэнд и всякие серверные вещи. Потому что Джава не самый быстрый язык (нет, ладно, с Python успешно конкурирует). А вот с плюсами уже не очень. Поэтому осмысленно на ней писать большую сложную систему:

1. ООП + строгая типизация – важны в больших проектах
2. ООП -> удобная модульность
3. программы запускаются на Java-машине – контейнере, который умеет во всякие плюшки.

А писать десктопные приложения на джаве можно, но вряд ли оно будет настолько большое и сложное, чтобы не написать его на чём-то другом.

JavaEE – уабор соглашений к языку программирования. Она вся основана на *аннотациях*

2.7. Аннотации

Например, `@Override` — это хорошо контролирует ошибки: если написали её к методу, отсутствующему в родительскуом классе, то бужет ошибка компиляции.

Основная задача аннотаций – статическое расширение классов. Что значит расширение? Положить вместе с классом некоторую дополнительную статическую информацию, то есть менять её нельзя. Где её использовать? Во-первых, во время компиляции — для отлова ошибок. И наоборот, когда ошибки на самом деле не нужны.

А ещё это даёт дополнительную функциональность — генерация кода.

А ещё можно использовать прямо во время работы и смотреть, например, на наличие/отсутствие каких-либо аннотаций в классе.

2.7.1. How-to: как написать аннотацию

```
@Target(ElementType.TYPE) //определяет, к чему можно будет приписывать эту аннотацию; в эт
@Retention(RetentionPolicy.RUNTIME) // информацию о наличии этой аннотации в классе можно
                    // на этапе компиляции её видно не будет
public @interface Mammal { // "@interface" === "Джава, смотри, я пишу свою аннотацию!"
    String sound();
    int color() default 0xffffff; // цвет по умолчанию -- белый
}
```

Научимся это приписывать к чему-нибудь.

```
@Mammal(color = oxffa844, sound = ''uuuu'') // задаём полям значения
class Giraffe {
}
```

Retention:

1. `SOURCE` — сбрасывается при создании .class файла
2. `CLASS` — в .class хранится, используется в основном для класслоадеров (они работают с бинарными файлами, поэтому недоступность в runtime'е их мало волнует)
3. `RUNTIME`

`User.Permission` из примера – это enum.
Ми-ми-маленькая хитрость: в аннотациях можно делать и массивы аннотаций, да.

2.7.2. Как используем

Это информация, которая навешена на класс сверху — не лезет во внутреннюю реализацию.

2.7.3. Сериализация

Можно, конечно, написать класс "СериализаторЯблока". Но когда на вход валится куча объектов — это писать огромный `if`, разбирая все возможные варианты входа? Это же ужас.
А что предлагается:

```
@interface SerializedBy {
    Class<? extends Serializer> value();
}
// написать такую аннотацию, мы внутри кода сохраняем связь между объектом и классами, кот
```

2.7.4. Ещё немного аннотаций

- `@NotNull` — позволяет избежать внезапных `NullPointerException`.
- `@Nullable` — наоборот, здесь может всякое быть. Зачем эта аннотация нужна? Ну, скорее для определённости, что мы не забыли про это место, а подумали и уверены, что нам так надо.

2.7.5. Ещё примеры

Пусть мы пишем приложение, там есть сервер, в нём есть класс, отвечающий за связь с базой данных, — `DB`. Пусть он в качестве параметров конструктора должен что-нибудь получить — например, имя сервера и пароль. Откуда нам их взять? Ну, наверно, из какого-то конфига. А мы хотим это полем в нашем классе приложения. Очень не хотим возиться со всей этой инициализацией руками — то ли дело написать `@Inject`, дальше разберётся сама JVM, только настроить её надо.

```
Application {
    @Inject
    DB db;
}
```

2.7.6. Промежуточный итог

Оченб мощная вещь. Прям боевая магия. Про всё это в слеующих сериях.

2.8. Аннотации в compile-time'e

Проыессор — сущность, которая будет занимаьтя обработкой аннотаций в процессе компиляции.

Чтобы его написать, надо отнаследоваться от `AbstractProcessor`'а и переопределить в нём четыре метода.

1. `init()` Конструктор должен быть пустым, вся инициализация — здесь.
2. `process()` `environment`'ы в них дают доступ к некоторым утилитарным классам. Предоставляет некоторый сет из аннотаций.
3. `getSupportedAnnotationTypes()` Возвращает список строк — названий аннотаций, которые мы хотим обрабатывать.
4. `getSupportedSourceVersion()` Мы можем наш процессор использовать в разных проектах, и чтобы описать, начиная с каких версий джавы с ним можно работать. Можно возвращать `SourceVersion.latestSupported()`(при вызове возвращает текущую версию джавы, в которой мы сейчас компилируемся), можно — `SourceVersion.RELEASE_6`.

В Java8 вместо последних двух методов принято писать две аннотации

```
@SupportedSourceVersion(SourceVersion.latestSupported())
@SupportedAnnotationTypes({
    // Set of full qullified annotation type names
})
```

2.8.1. jar-ник

В него надо записать содержимое MyProcessor.class и javax.annotation.processing.Processor. Передать компилятору со специальном ключиком и всё сделается за нас (а в какой-нибудь среде разработки так и подавно).

2.8.2. Пример: пицца и тирамису

[Магазин пиццы] Это ужасный код, но мы на него сейчас посмотрим, потому что нам всё равно его сейчас генерировать. Ужасно здесь то, что если у нас, паче всякого чаяния, добавился ещё один вид еды, то нам придётся дописывать ещё один `if`.

[Вынесем фабрику в отдельный класс] И вот эту фабрику еды мы сейчас и будем генерировать.

[MealFactory] Давайте подумаем, что мы вообще должны делать? Нам надо найти все классы еды. Для этого надо знать, как класс зовут. Чтобы это делать, мы создадим собственную аннотацию `@Factory`.

[@Factory] Без type можно было и обойтись, но тогда мы сможем безболезненно ещё и чайниковые фабрики генерировать. Мы хотим, чтобы наш процессор нашёл все классы с аннотациями `@Factory` и типом `Meal.class` и генерировал нужный код.

Чтобы мы могли это делать, в классе должен быть публичный конструктор без параметров, не кидающий exception'ов, он должен быть отнаследован от своего type (Meal.class).

1. Нельзя интерфейсы и абстрактные классы
2. Публичный конструктор безх параметров и исключений
3. Нужна возможность привести аннотированный класс к "типу" из параметров
4. Классы с одним "типом" будут объединены в одну фабрику
5. `id` — это строка, должна быть уникальной для классов с одним *типом*

[Приступаем] Почему сет, а не коллекция? Потому что чтобы уникальные элементы были.

`@AutoService(Processor.class)` — автоматически генерирует манифест-файл. Да-да, и у неё есть свой процессор.

[Приступаем] В `init`'е сохраняем экземпляры всех четырех утилитарных классов в приватные поля.

[Init]

- Elements — Element. Это о чём вообще? AnnotationProcessor сканирует весь код и разбивает на некоторые части — называются *элементами*. Каждому элементу можно сопоставить некоторую сущность языка.

  Как элементы обходить? [Как обходить элементы] `TypeElement.getEnclosedElements()` — получить детей. `~Enclosing~` — получить родителя.

  НО это всё есть всего лишь описание элемента исходного кода, поэтому родительский класс, например, мы узнать не можем. `TypeMirror` — может.

- Type — для работы с TypeMirror. Позволяет получать класс `TypeElement`'а, прямо тот, который с точки зрения языка Java. Делается методом `element.asType()`.

- Filer — автоматически создаёт файлы в нужных папках (по конвенции, всё строго)

[Поиск аннотации @Factory] Код, который пробежится по всем классам, проаннотированным аннотацией @Factory.

`.getElementsAnnotatedWith(Factory.class)` — вернёт любые элементы, к которым приписана данная аннотация. Но мы должны проверить, что пользователь сделал всё правильго и приписал аннотацию к классу. [Проверим]

TypeElement — слишком широкое понятие (нам интерфейсы и абстрактные классы не подходят.) Поэтому так.

Если мы обнаружили, что пользователь всё-таки приписал аннотацию не туда, то джава, конечно, предлагает нам кинуть `Exception` — но нам нужна ошибка компиляции,а не исключение, чтобы

некорректно завершать компилятор и только среду разработки смущать. Что делать? Как генерировать сообщение об ошибке компиляции? Используем четвёртую сущность — `messager`. (функция `error()` в нашем коде) Компилятор всё поймёт, место — засчёт параметра `e`, имеющего тип `Element` (то есть "строчка кода")

Вообще, метод `process()` должен возвращать `Boolean`. Если он вернул `false`, то компиляция завершится. Но `AnnotationProcessor`'ам *крайне* не рекомендуется возвращать `false`, потому что кроме нас есть и другие процессоры, надо дать им тоже нормально отработать.

[Лирическое отсутупление] Чтобы нам дальше было удобно всё это обрабатывать, введём ещё две сущности — напишем ещё два класса для работы с классами, проаннотированными нашей аннотацией: куча_классов_сгруппированных_по_типу и один_класс_из_этой_кучи

[Монструозный слайдик O_o] Первая часть кода получает `id` (`TypeElement` → `getAnnotation` → `id`).

Вторая часть: просим у аннотации `type`. Но класс, который мы получили, мог быть ещё не откомпилирован, то что делать? Тогда строчка `annotation.type` выдаст нам `MirroredTypeException` (то есть класс ещё не откомипилирован) — в ответ на это мы берём у него `declaredTypeMirror` (типа "описание описания класса") (плохо пользоваться `TypeMirror`'ом, долго это, поэтому чтараемся избегать, по возможности). Точнее, это ещё не самое долгое — его мажорирует `asElement()`, потому что он идёт в файлы с кучей неоткомпилированного кода и среди него находит нужный нам класс.

`FactoryGroupedClasses` — кучка классов с одинаковым `type`'ом.

[Возвращаемся к Processor] Ntgthm? rjulf vs ghjdthbkb? xnj fyyjnfwbz приписал аннотацию именно к классу, мы можем безбоязненно кастовать `(TypeElement) annotatedElement`

[Что мы проверяем] Метод `isValid()` проверяет наши тербования.

Reminder: `factoryClasses` — мапа из типа в кучки.

Всё, мы все кучки разложили, теперь можно генерировать. Надо пробежаться по всем кучкам и сказать "сгенерь код".

[Code generation] [Как сгенерировать код?] Хоть строчки конкатенировать, хоть утилитки использовать — например, здесь используется библиотека `JavaWriter` (пристально смотрим на переменную `jw`).

[Processing Rounds] Мы молодцы, конечно. Но компилировать-то это кто будет? Поэтому говорят, что компиляция происходит в несколько раундов. Помимо новых файлов на втором этапе компиляции мы будем компилировать те файлы, которые не смогли скомпилировать на первом этапе. Новые раунды будут до тех пор, пока набор файлов меняется. Для нашего процессора каждый раунд — вызов метода `process` (`init` — только один раз). Поэтому надо очищать в `process`'е всё, что нам не потребуется на следующем этапе — например, чтобы не генерировать снова `MealFactory` (генерировать два одинаковых файла запрещено, так что это кончится ошибкой компиляции).

2.8.3. Как ещё?

Можно изменять существующие классы на этапе компиляции. Это сложно, но библиотечки есть, компилятор обманываем. А ещё мы можем ещё и не исходные, а уже откомпилированные java-файлы генерировать (например, запустить джавовский компилятор отдельно; так мы наш компилятор и обманем).

Глава 3

Java EE

3.1. О чём это?

Набор спецификаций, которые реализуют различные контейнеры. Потому что программа может выполняться не только в JVM, но и в чем-то другом – они, контейнеры, сущности, в каком-то смысле прослойка, расширяющая наши возможности. А контейнер выполняется на JVM.

Обычно мы компилируем `.class` и скармливаем JVM'не. А контейнеру можно и артефакт `JAR` или `WAR`(для запуска вашей программы внутри веб-контейнера – он как веб-сервер, написанный на джаве) скормить, чтобы тот его запустил. `JAR` — это не только заархивированные(zip) `.class`-файлы, но можно и доп. служебную информацию, например, какой класс можно запустить, если стартует весь jar целиком, какие-нибудь настройки и т.д.

3.1.1. JCP

Java Communication Process – определяют, как Java будет дальше развиваться. Запрос на новую функциональность называется JSR, для этой фигни разрабатывается спецификация, затем пишется базовая реализация и набор тестов, которым должны соответствовать сторонние реализации, которые хотят соответствовать этой спецификации.

Текущая версия JavaEE отстаёт от JavaSE на одну версию.

3.2. DI & CDI

(Context) Dependency Injection. Технологии, объединяющая в себе несколько спецификаций.

Пусть у нас есть библиотека и генератор штрих-кодов. Короче, приделать штрих-коды со стороны нужно в книжки в библиотеке – и очень хочется не думать об этом, а написать `@Inject` и чтобы оно "само сделалось". Только это "само— контейнер, именно он за нас это делает. Видя эту аннотацию, он понимает, что в данный класс нужно инжектировать что-то – идёт, это что-то создаёт и вставляет. Это довольно сложно. И всё это на лету во время работы.

`POJO` — Plain Old Java Object. Наши вот все такие: `new` для создания экземпляра (а можно заметить, при `@Inject` никакого `new` не было – там по-другому)

Базовая реализация CDI — WELD (от компании JBoss).

"Доп. настройки"из артефакта – для DI это должен обязательно иметься `beans.xml`.

Что можно инъектировать? Не любой же. Чтобы был не абстрактный, чтобы был публичный, чтобы конструктор по умолчанию был (конструктор с параметрами может быть, но эти параметры тогда тоже должны инъектироваться).

Инжект от метода – все параметры у него надо инъектировать.

Квалификатор. Например, `@Default`: сообщает CDI, что если вдруг кто-то захочет инжектировать, а у нас несколько реализаций – показываем, какую именно, если подходит любая (недефолтную тоже можно указать, но это явно делается).

Чем `@Inject` вообще лучше **new**? Во-первых, мы ничего не переписывали, но в другое место приписали `@Default` — и вот уже поведение библиотеки изменилось без изменения её кода. Более того, мы можем этого добиться, вообще не меняя и не перекомпилируя код – просто подправить `beans.xml`.

3.2.1. Квалификаторы

Идёт вместе с инжектом и помогает определить конкретный класс, который надо инжектировать. Это тоже некоторая аннотация, к которой нужно приписать аннотацию `@Qualifier`(пустая). Можно писать свои, соответственно.

Говорят, что "CDI обеспечивает строгую типизацию".

Если хотим квалификатор с параметрами

```
@Qualifier
@Retention(RUNTIME)
@Target({FIELD, TYPE, METHOD})
...
```

Множественные квалификаторы — инжектировать с несколькими ограничениями.

3.2.2. Альтернативы

Пусть мы хотим какие-то заглушки инжектировать весто нормальных классов – нам нужна альтернатива дефолтному классу, который обычно инжектится.

Соответственно, при обычном запуске будет инъектировать нормальные классы, а с определёнными настройками — альтернативы. (делается в xml-файле)

3.3. Производители данных

Хотим, чтобы на место поля подставлялся результат работы какого-то метода.

Пусть у нас есть метод, выступающий в качестве фабрики экземпляров компонентов. Будем искать нужное значение по результатам сущностей, помеченных `@Produces` (и, может, какие-нибудь дополнительные)

Константе в JEE можно распространить на всё приложение — если где-то ещё напишем `@Inject @Same` то автоматически значение туда подставится.

Если у класса будут вызываться любые методы, то на самом деле будет вызываться тот, который помечен `@AroundInvoke`(он как перехватчик, все вызовы через него проходят), а уже оттуда – они (скармливается перехватчику, а тот уже решает, будет вызывать или нет). Отличный способ залогировать всё, что происходит внутри класса.

3.4. Декораторы

(как раз с помощью инжектов)