Оглавление

3.2.1 Аннотации

Картинка

Добрый день. Давайте начнем вторую часть занятия по базам данных. И если в первой мы с вами разбирали базовые возможности, которые вообще нам предоставлены в языке библиотеки для того, чтобы работать с базами, то в этой половине мы с вами разберем более высокоуровневые подходы, то есть разберем то, что делает нашу работу с базами удобнее и скрывает от нас детали внутренней реализации, то есть если прошлый раз мы с вами разобрали Executor, то сейчас этот Executor от нас будет спрятан внутри библиотеки, которую мы будем использовать. Начну я не совсем с баз данных, начну с аннотаций, потому что работа очень многих сторонних библиотек, работы которых основана на анализе вашего кода. Эта работа основана на восприятии аннотаций.

```
1.  Аннотации — метаданные в коде
2.
3.  Содержат данные о программе, не являясь
4.  частью программы
5.  Не влияют непосредственно на работу
6.  приложения
7.  Влияют только на ту функциональность, которая
8.  их обрабатывает
9.
```

10. 11.	Могут влиять на работу компилятора, инструментов и библиотек "Decorating" or "wrapping" классы в runtime

Для начало разберем что такое аннотации. Я думаю, все из вас знают, что код можно подписать комментарием, то есть вы можете в любом месте своего кода поставить два слеша и написать что-нибудь поясняющее. При желании можно написать приложение, которое проанализирует ваш код и найдет все те места, которые вы пометили как комментарий. Если вы, как человек, который пишет комментарии, и человек, который пишет анализатор, договоритесь, что какие-то виды ваших комментариев должны быть каким-то образом интерпретированы, то программа, которая будет разбирать ваш код и искать комментарии, сможет этот ваш код поменять на основе ваших комментариев. То есть, например, вы можете создать функцию, у функции перечислить параметры, которые в функцию нужно передавать, и поставить перед переменной внутри функции комментарий, означающий что это переменная не должна быть null. А та часть, которая будет ваш код обрабатывать, другое приложение, она сможет найти этот ваш комментарий, понять что здесь должен быть

не нулевая переменная и удалить
комментарий, а вместо него в начало
функции вставить проверку,
что действительно переменная не null,
а если null — выкинуть это все.
Или какое-то исключение, какое
договоритесь. То есть вы таким образом
получили возможность подписать свой код,
расширить его возможности
перед компиляцией или если у вас есть
возможность перенести эти комментарии
в runtime, то уже как-то отреагировать
на них в runtime. Это и есть аннотации.
То есть аннотации — это такой удобный
способ подписать некий функционал
дополнительно, то есть подписать ваш код,
сказать что тому, кому интересно, может
увидеть здесь еще дополнительные
возможности. Аннотации также как
и комментарии, которые я упоминал,
в чем-то дополнительны к коду. За редким
исключением наличие или отсутствие
аннотации на компиляцию не влияет. Только
есть некий набор аннотаций, который
компилятору известны, специально для него
написаны. Все остальные, тем более
самописные аннотации, наличие
или их отсутствие в коде также как наличие
и отсутствия комментариев на компиляцию
не влияют. И в общем-то по большой части

непосредственно на работу тоже не влияют, то есть вы можете подписать аннотациями, такими метакомментариями, свой код, запустить его, убрать, собрать запустить в общем-то можно работать и так. Таким образом, добавляя аннотации, вы добавляете возможность другим библиотекам, либо вашим собственным, либо написанными кем-то еще, разобрать ваш код в тот момент, когда вы его пишите, в тот момент, когда идет компиляция, и уже в runtime, если аннотации остаются в runtime. То есть вы таким образом передаете некую информацию, дополнительную к коду, тому, кому эта информация может быть интересна. Если она ему не интересна, он ее пропускает, если интересна, он каким-то образом может на нее реагировать. И аннотации, в отличии от комментариев, могут повлиять на работу компилятора, то есть компилятор может обратить внимание на те или иные аннотации. Влияет на инструменты, которыми вы пользуетесь. Например, на среду разработки, то есть вы пишите код, а среда разработки следит за тем, есть ли аннотации на том, что вы написали, и что вы стороннее используете. Например, среда разработки может сказать вам, что эту функцию лучше не использовать, потому что

разработчики библиотеки, из которой вы ее используете, считают, что эта функция уже устарела. То есть в процессе написания и в процессе компиляции, и еще в runtime, когда вы уже собрали приложение, передали его в Java-машину, она его запустила, вы в своем приложении обратились к некой библиотеке, библиотека обратила внимание на ваш код и проанализировала его. Например, на ваши объекты. То есть вы ей говорите вот мои объекты или классы моих объектов, а библиотека уже начинает смотреть что же там за классы и видит в них аннотации. Получается такая некая дополнительная обертка для того, кому это нужно.

```
Примеры аннотаций

Аннотация на класс
@Deprecated
class oldClass{...}

Аннотация на метод
@Test
public void unitTestSomething(){...}

Аннотация на поле
@Nullable
private Object object;

Аннотация на переменную
pubic int getUserId(@NotNull User user){...}
```

Если перейти уже к практической части, то так могут выглядеть аннотации. Аннотации можно прицепить к классу. То есть вид аннотации. У вас есть нечто, на что вы хотите повесить аннотацию, вы пишите собаку, название аннотации, может в нее какие-то параметры. Здесь у меня в примерах параметров нет. oldClass, который здесь представлен в самом начале, я на него повесил аннотацию @Deprecated. И это означает, что среда разработки если увидит, что я пытаюсь использовать этот класс, она мне скажет, что лучше его не использовать, потому что он @Deprecated. Кроме как аннотации на класс, мы можем повесить аннотацию на метод. Опять же вспоминайте пример с комментарием. Представьте, что я просто взял и написал комментарий Deprecated oldClass. И среда разработки точно так же, как она увидела аннотацию, мы с ней можем договориться, что она будет видеть мои комментарии. Получается нам не удобно, если делать все комментариями, то все в куче получается и комментарии для человека и комментарии для среды, и для компилятора. Поэтому решили, что давайте отдельно удобнее сделаем возможность подписать. И эта удобная возможность подписать — это и есть как раз

аннотации. Кроме класса можно повесить на функцию. В данном случае функция тестовая. И эту аннотацию @Test увидит библиотека, которую вы запустите, чтобы протестировать свое приложение. Она просто посмотрит, что у класса есть методы, у этих методов есть аннотация @Test, а это значит, что можно эти функции запустить. В них произойдут какие-то изменения, посмотреть на результаты, если результаты хорошие, то тест прошел. Можно повесить аннотацию на поле, то есть представьте, что у меня класс, в классе есть поле с именем object класса Object. И я могу сказать, что в этом классе этот object будет @Nullable. То есть будьте все осторожны, когда вы пытаетесь к нему обратиться, проверяйте на null, потому что я могу в конструкторе его не создать, а надеяться на то, что кто-нибудь когда-нибудь вызовет сеттер и проставит его. И можно повесить аннотацию на переменную. То есть если у вас есть функция, у этой функции есть список параметров, вы список параметров можете вставить в аннотацию. Пример, с которого я начал, @NotNull — это аннотация означает что пользователь не должен быть null.

3.2.2 Аннотации

```
@NotNull

Ссылка не может быть null
Для программиста — указание на то,
что на null можно не проверять
Для среды разработки — подсветка присваивания
null
В runtime — исключение в момент присваивания
null
```

Теперь давайте разберем как этот самый @NotNull у нас вообще работает. По задумке @NotNull нужен нам для того, чтобы показать, с одной стороны, разработчику, операционной системе, среде разработки и компилятору, что поле не null. То есть у вас есть класс, в классе есть поле. Вы прицепляете к нему аннотацию @NotNull. Таким образом вы всем, кто смотрит на код и человеку, и машине подсказываете, что это поле по задумке null не может быть. Точно так же может быть @NotNull тип возвращаемого значения. То есть функция обязана поработать и по результатам своей работы вернуть не null поле. Или это может быть переменная, в функцию которой вы перекладываете. И тот человек, который пишет функцию, он всем, кто эту функцию будет вызывать, подсказывает, что при вызове этой функции в качестве параметра null передавать нельзя. Задумка понятна. Вернусь к самому первому примеру.

В данном случае использование аннотации похоже на использование комментария. В том только отличие, что у нас есть встроенный в компилятор и среду разработки проверка того, что то, что мы написали, оно действительно @NotNull. Некий такой получился оговоренный комментарий. Мы договорились со средой разработкой, договорились с компилятором, что если @NotNull, значит @NotNull. Как это может работать? То есть я везде расставил @NotNull. И я уверен, что у меня в функцию null не попадет. Но как это подкреплено в языке? В среде, в принципе, понятно. Среда разработки может статически проанализировать ваш код и увидеть, что я передаю в @NotNull функцию, и мне подсветить это место и сказать, что тут возможны ошибки. Компилятор в принципе тоже может понять, то есть он собирает также статически проанализирует. Но нам надо то на самом деле не только при разработке, и при компиляции. Нам еще хорошо бы в runtime все это дело анализировать. Вдруг если я допустил ошибки при сборке, то есть пропустил эти сообщения от среды разработки, от компилятора, оно соберется, кстати. Плюсы еще если я использую какую-то библиотеку, а в библиотеки у меня уже

собранное. То мне нужно бы в runtime тоже проверять, а в runtime проверять дорого.

1.	Как работает @NotNull
2.	
3.	Annotation preprocessors
4.	Плагины для компилятора
5.	Разбирают аннотации
6.	Могут создать дополнительный код (в том числе
7.	новые классы)
8.	Можно написать свои плагины

И поэтому есть на стадии компиляции такая стадия, в которую мы запускаем Annotation preprocessors, то есть это плагины для компилятора. Их работа пробежать по коду, который вы собираете, проанализировать аннотации и для известных им аннотаций выполнить некие действия. То есть они могут неким образом модифицировать ваш код, то есть получается в момент сборки ваш код может быть поправлен preprocessors неким образом, который вы указали в аннотациях. В нашем случае, в случае @NotNull, annotation preprocessors пробегут по коду, увидят везде, где @NotNull. @NotNull можно будет при желании будет убрать, можно будет и оставить. Будет ли он в runtime в байт-коде потом я имею в виду. В том месте везде, где @NotNull взять и вставить проверки, что допустим везде, где идет присваивание переменной, присваивание идет

не @NotNull если переменный класс. Функция перед тем, как вернуть значение, может проверить null или не null возвращаемое значение. С параметром все вообще просто, просто в самое начало тело функции preprocessors может вставит проверку а null ли переменная, если переменная null, то взять и пропарсить в этом месте. Вы можете писать свои собственные плагины. Точно так же, как вы можете писать свои собственные аннотации. Об этом я буквально сейчас расскажу. Например, вы можете придумать аннотацию, которая будет проверять, что строка, которую вы передаете в функцию, не пустая. То есть стандартной аннотации на это нет. Но никто не мешает вам ее сделать. Вы пишите аннотацию, называете ее @NotEmpty, прицепляете эту аннотацию к строке, может быть в классе, может быть при передаваемых параметрах, и пишите плагин, который позволяет при компиляции проверить везде, где такие аннотации есть, поменять место, где эти аннотации, на некий код, который будет встроен во все те места, где вы работаете с этой строкой. И тогда получается вместо того, чтобы вам везде дублировать свой один и тот код проверки на пустую не пустую строку, вы получаете возможность просто

расставлять удобную аннотацию везде короткую @NotEmpty. И вам удобно будет, и разработчикам другим. И можете среду каким-то образом тоже через плагин настроить. И компилятору будет понятно, что с этим кодом делать. Пока что бонусы от аннотации на стадии разработки и компиляции очевидны. Какие есть еще бонусы — в следующих занятиях.

3.2.3 Аннотации

```
Синтаксис аннотаций

Не может наследовать или быть базовым классом
Не содержит конструкторов и полей
Может содержать методы без переменных,
которые работают как поля

@interface CreateBy {
    String author();
    String date();
}
@CreateBy (author = "tully", date =
        "01.01.2014")
public class MyClass {...}

@CreateBy (author = "tully", date =
        "01.19.2014")
public static void main(String args[])
```

Я уже говорил, что вы можете создать свои собственные аннотации. И самое время посмотреть, как это делать. На этом слайде внизу пример кода самой аннотации и использовании аннотации. Для того,

чтобы создать свою собственную аннотацию, вы в отдельном файле задаете интерфейс с названием аннотации. Единственная разница по сравнению с обычным интерфейсом в том, что перед интерфейсом вы еще должны поставить собаку (8). То есть разработчики аннотации решили ничего нового не придумывать, сказали пусть будет собака, потом слово interface и название интерфейса. Выглядит все как обычный класс или обычный интерфейс, только вот еще собака. Никакого наследования в аннотации быть не может, ни сама не может наследовать, ни какого-то другого, не нужно ему никакие конструкторы, никакие поля, в общем так же как у интерфейса. Но может содержать свои методы. Здесь начинаются странности. Всего я таких странностей знаю в Java два места: аннотации и Input/Output. До Input/Output еще дойдем, а в случае с аннотациями смотрите. В аннотации решили добавить поля. Но аннотация — это некая надстройка над интерфейсом. В интерфейсе полей нет, кроме статических. И чтобы добавить в интерфейс поля, решили что а давайте мы их напишем как будто они методы, но работать они будут как поля. То есть синтаксис (9) означает, что у меня есть аннотация, внутри аннотации есть поле.

И поле это с именем author. Написали, давайте использовать. И мы можем куда хотим на классы, на методы, на поля ставить свою аннотацию, написать @CreateBy (12) и перечислять поля, то есть в скобочках после аннотации я могу перечислить поля через запятую и значения, заполняющие эти поля. В данном примере я создал аннотацию, которую можно повесить на некий участок кода. Аннотация подписывает того, кто и когда создал этот участок кода, то есть, например я написал некий класс, пометил его аннотаций, передал туда имя и дату. Дальше я хотел бы, чтобы эту информацию можно было получить в runtime, то есть, например, я запускаю код и хочу, чтобы запущенное приложение мне выдало список того, кто когда что сделал. Такая задача. И сделать это можно следующим образом.

```
 1.  Обработка аннотации в runtime
 2.
 3.  Class myClass = MyClass.class;
 4.  Method method = myClass.getMethod("main");
 5.
 6.  CreateBy annotationC = myClass.getAnnotation(
 7.          CreateBy.class);
 8.  CteateBy annotetionM = method.getAnnotation(
 9.          CreateBy.class);
10.
11.  log.info("Autor of the class: " +
12.          annotationC.author());
```

```
13.  log.info("Date of Writing the class: " +
14.          annotationC.data());
15.  log.info("Author of the method: " +
16.          annotationM.author());
17.  log.info("Date of Writing the method: " +
18.          annotationM.data());
```

У нас в runtime есть возможность получить доступ к аннотации. Предыдущий мой рассказ об аннотации был о том, что можно статическим анализом увидеть аннотации и повлиять как-то на код. Кроме этого, есть еще возможность к аннотациям обратиться в runtime. Вы помните, что есть в Java понятие Class (3), то есть это класс, представляющий объект класса. И я в предыдущем примере уже создал класс с названием MyClass. Присваивание в этой строке (3) означает следующее. Я в runtime говорю дай мне пожалуйста в этом месте в переменную myClass запишем ссылку на объект, в объекте в этом будет информация о классе, то есть эта запись означает я хочу получить ссылку на объект с информацией о классе. И кроме того, когда я эту ссылку на объект получил, я могу у этого класса попросить а дай мне метод. Я знаю, что у тебя есть метод main(), дай мне пожалуйста ссылку на метод, то есть Method method (4) — это тоже библиотечный класс такой же как,

простите, класс. И я могу получить ссылку на объект, представляющий метод. То есть объекты, приставляющие классы, в них хранится информация о классах. Что значит хранится информация? Значит она каким-то образом представлена. Если у меня у касса есть несколько методов, то логично эту информацию представить в виде массива методов. Здесь именно это и происходит. У меня есть MyClass, в нем есть метод main() и я получаю ссылку на объект, который в runtime представляет этот самый main(). Я думаю вы уже догадались зачем мне это нужно. И у класса, и у метода я могу спросить аннотации. В runtime данные об этом есть. Эта информация записана в классе, записана в объекте метода. Поэтому я могу просто сказать дай мне аннотацию (6, 8). Либо я могу сказать дай мне все аннотации. Такой вызов тоже есть. И после этого я, получив аннотации, могу взять и в логе прописать все то, что получил. Я могу проанализировать все свои классы, со всеми своими методами в runtime и узнать нужную мне информацию. Мы с вами встретимся уже в следующей части занятий с тестированием, с Unit-тесты. Unit-тесты тесты мы будем запускать, используя JUnit. И одна из аннотаций, которая нам поможет в этом будет аннотация

@Test. Я ее сегодня даже показывал. И эта аннотация подскажет библиотеки Junit, что есть некий набор классов, в классах есть методы, эти методы, помеченные аннотацией @Test нужно запустить как тесты. То есть произойдет в точности то, что я здесь показал, но только не по отношению к моим собственным аннотациям, а по отношению к стандартным, библиотечным аннотациям @Test. Что такое будет @Test? @Test — это будет обычный класс, в классе будет некий набор методов, часть из них будет помечена аннотацией @Test. И когда я захочу запустить тестирование, я ничего дополнительного в класс добавлять не буду. Просто библиотека, которая захочет протестировать, посмотрит список методов, в точности так же, как на слайде. И увидит где какие методы с аннотацией @Test и запустит их.

3.2.4 Аннотации

```
1.  Виды аннотаций
2.
3.  RetentionPolice.SOURSE — аннотации
4.  присутствуют только в коде
5.  Информация для компилятора
6.  Информация для инструментов IDE
7.
8.  RetentionPolice.CLASS — сохранены в .class,
9.  но не доступны в runtime
```

```
10.  Доступны при анализе byte кода
11.  Не доступны через reflection
12.
13.  RetentionPolice.RUNTIME — сохранены в .class
14.  и доступны в runtime
15.  Аннотации можно получать из getClass()
16.  через reflection
```

Я уже несколько раз говорил вам, что аннотации можно проанализировать во время написания кода, среда разработки может проанализировать. В принципе, вы сами, когда видите какую-то аннотацию, уже тоже можете понимать в чем здесь дело, что происходит именно в этом участке кода. Анализ человека, анализ машины, анализ среды разработки, анализ компилятора и еще возможность получить доступ в runtime. Чтобы каким-то образом это все упорядочить придумали виды аннотаций, то есть три стандартных вида аннотаций. По тому, откуда они видны, то есть их решили так разделить в три группы. И назвали все это RetentionPolice. Это enum, у него три значения SOURCE, CLASS и RUNTIME. (3) означает, что аннотация будет доступна только при анализе исходного кода. В байт коде ее не будет совсем. То есть она просто туда не попадет. Если она вам там не нужна, если вы не собираетесь аннотацию получать в процессе работы приложения, то делайте (3) и все.

Есть (8). Это стандартный по умолчанию RetentionPolice. Это означает, что запись об аннотации попадет в байт-код, но в runtime доступна не будет опять же. То есть запись о ней будет, но получить ее через рефлекшн у класса будет невозможно. Дискуссия у нас была о том, зачем эта вообще аннотация CLASS нужна, сошлись на том, что в общем полезно будет для безопасности. Вы подписываете свой код, в байт-коде он есть. Но на производительность, на работу он не влияет.

И есть еще (12). Аннотация этого типа означает, что вы где угодно можете к этой аннотации получить доступ. Вы можете естественно ее увидеть в исходниках. Она есть в байт-коде, и она будет еще в run time. Через reflection можно получить класс или метод и спросить дай аннотации и там все (12) у вас списки будут. Теперь вопрос внимательным слушателям разделам про аннотации. На лекциях я в этом месте даже делаю некую паузу. Вам предлагаю перед тем, как я перейду на следующий слайд, тоже остановить видео и предложить свое решение. Есть у нас аннотации, мы можем написать новые аннотации, у нас уже есть

библиотечная аннотации, и они разбиты в три группы. И нам нужно при создании своей аннотации, либо обратившись к исходнику уже существующей аннотации, понять какого она типа, то есть нам нужно указать какого типа аннотация. Вы пишите аннотацию, вам нужно указать какого она типа. Как бы вы указали тип аннотации. Здесь такая пауза. И если вы этой паузой воспользовались, подумали, у вас есть свое решение, решение в действительности следующее.

```
1.  Аннотации аннотаций
2.
3.  @Retention — аннотация для указания типа
4.  аннотации
5.
6.  @Retention(RetentionPolice.RUNTIME)
7.  @interface CreateBy {
8.      String author();
9.      String date();
10. }
```

RetentionPolice мы передаем через аннотацию, то есть аннотация нужна для того, чтобы подписать функциональность. У нас есть аннотация, мы хотим ее подписать, мы для нее используем аннотацию. Пример того, как мы ее используем. То есть у нас есть аннотация @Retention. Она на вход берет параметр RetentionPolice. Если переменная

всего одна в аннотации, то не нужно указывать ее имя, просто пишите ее значение. Это как здесь, то есть @Retention, у нее есть одно поле, видимо retention. И мы передаем RetentionPolice.RUNTIME.
Сама RetentionPolice.RUNTIME она RetentionPolice.RUNTIME. Так что все нормально. Там бесконечный цикл не появляется. CreateBy в моем случае будет видна во время исполнения приложения тоже, потому что я ей указал RetentionPolice.RUNTIME.

3.2.5 ORM

Вернемся к разбору особенностей работы с базой данных из Java. И перед тем, как перейти к библиотеке, про которую я и хочу вам рассказать, остановимся на том что такое ORM. Некоторые слова я уже говорил в начале занятия об этом. Но мне хотелось бы подробнее это место еще обговорить, чтобы было понятно какие бонусы мы получаем с вами.

1.	Object-Relation Mapping
2.	
3.	Связь между объектной моделью программы
4.	и реляционной моделью базы
5.	
6.	Примеры
7.	

8.	Метод DAO принимают объекты и сохраняют их
9.	в базе
10.	Методы DAO возвращают объекты собирая их
	на основе запросов

Картинка

У нас есть приложение, написанное с использование объектов с одной стороны и базы данных с другой стороны. У базы данных нет объектов. Там таблицы. У нас в приложении нет таблиц, у нас объекты. ORM — это то, что позволяет нам видеть из приложения базу как будто там объекты, а для базы приложение поставляет запросы к реляционным схемам. То есть ORM перекладывает объекты в записи в таблицах и на основе записей из таблиц может собрать объекты. И для этого мы будем элементы ORM, о которых я сейчас рассказываю, это DAO и DataSet. DAO — это Data Access Object. DataSet — это набор данных. DAO — это центральный элемент ORM, потому что задача DAO взять от вас объекты и превратить его в запрос к базе, либо предоставить разработчику функцию, при вызове которой внутри себя DAO создаст запрос к базе, а разработчику отдаст объекты, то есть это некий дополнительный слой абстракции, позволяющий всем тем, кто работает с базой вместо того, чтобы писать запросы

производить вызов методов у объектов. То есть вы хотите получить некую запись от базы, вы результат этого запроса оформляете как DataSet, запрашиваете DAO дай мне данные уже не просто в виде таблицы, не просто в виде ResaltSet, о котором мы раньше говорил, а дай мне данные в виде списка DataSet и получаете на выход объекты, с которыми потом каким-то образом работать. Либо у вас есть уже объекты, и вы передаете их в DAO и DAO их превращает в записи.

```
1.  Java Persistence API
2.
3.  Package javax.persistence
4.
5.  Аннотации
6.  @Entity: объект класса можно «переложить»
7.  в таблицу
8.  @Table: связывает класс и таблицу
9.  @Id: поле является первичным ключом в таблице
10. @Column: связывает поле и колонку в таблице
```

Почему мы до этого разбирали аннотации. Потому что, чтобы все это работало, в Java есть раздел javax.persistence. Это часть стандартной библиотеки, то есть разработчики стандартной библиотеки позаботились о том, чтобы предоставить возможность разработчикам ORM решений и пользователям ORM решений использовать единую систему аннотаций. На этом слайде я

привел основные аннотации из этого списка. Если вы хотите, чтобы какой-то объект был понятен ORM системе как объект, который можно переложить в таблицу, вы обозначаете его как @Entity, потом аннотации @Table можете передать его конкретной таблице и разметить поля, которые в этом классе есть, пометить их как колонкой или как специальная колонка с аннотацией id, если вы хотите, чтобы эта колонка была первичным ключом, то есть вы не работаете с базой запросами, вы работаете с базой через создание классов со специальной разметкой. Сообщаете библиотеке о том, что у вас есть эти классы и при запросе дай мне данные из таблицы вы на выход получаете объекты этих классов.

3.2.6 DataSet

```
1.  DataSet
2.
3.  DataSet — объект, содержащий данные одной
4.  строки таблицы
5.  На каждую таблицу свой DataSet
6.  Извлечение и вставку данных удобно проводить
7.  через DataSet
8.  В терминах JPA DataSet это Entity
```

Подробнее разберем что такое DataSet и что такое DAO. ORM подход к работе с базой он кроме бонусов, которые сразу вам предоставляет, что вы видите

результаты запросов как объекты
и отправляете на запрос тоже объекты,
он дает некие ограничения. Ограничения
в том смысле, что без ORM вы можете
спросить какой-то участок таблицы,
ограничив его набором нужных вам колонок
вплоть до того, чтобы извлечь одну
колонку. Получить на выход некую
абстрактную структуру данных типа таблицы
или как ResaltSet сделан и кастомным
образом с ней поработать. В случае ORM
не принято писать на каждый особенный тип
запроса свой класс, то есть обычно делают
таблицу в базе с некоторым количеством
колонок, 10 колонок. И для каждой таблицы
создают один класс, представляющий одну
строку в таблице. В этом классе будут
перечислены все поля, если у нас там
10 колонок, то в DataSet будет 10 полей
с именами, совпадающими с теми,
что в таблице. И с типами, совпадающими
теми, что в таблице. То есть получается
что когда вы запрашиваете дай мне записи.
Вам нужно только дни рождения
пользователей. В этой таблице у вас и дни
рождения, и последнее время входа
в систему, и может быть очки какие-нибудь
еще набранные, имена. Вы все равно
запрашиваете целиком все строки, то есть
вы говорите дай мне запись пользователей,

вам на самом деле нужно только имя пользователя или только день его рождения, но тем не менее вы запрашиваете целиком всю строку. То есть это некая избыточность, но при проектировании идут на эту жертву, потому что в целом работать становится удобнее. И запрашивают у базы из таблиц целиком строки. То есть мне нужен пользователь, зная его id запрошу целиком строку и эту строку получу в виде DataSet, то есть почему так получилось. Нам нужно при запросе из приложения получить на выход объект. Объект — это некая структура данных с заранее заданным количеством полей. В Java по крайней мере так. Если вы в каждом запросе будете менять формат выдачи данных, то вам нужно будет каждый раз новый этот Set придумывать. Так никто не делает. Поэтому давайте просто такое ограничение сразу при использовании ORM. DataSet — это объект, в классе которого перечислены поля и эти поля соответствуют данным одной строки таблицы. Соответствуют точнее одной строке таблицы, каждое поле соответствует своей какой-то колонке. На каждую таблицу вы заводите DataSet. Если вам нужно что-то у таблицы спросить, вы у объекта DAO, который сейчас будем обсуждать, спрашиваете дай мне DataSet,

соответствующий пользователю, которого мы ищем. И тогда, если нам нужно новую вставку сделать в таблицу, то мы создаем DataSet, заполняем все поля, которые нам нужны, кроме генерируемых таблицей, если такие есть, и передаем в DAO объект DataSet и говорим сохранить. И она ее сохраняет. То есть мы при работе с базой из приложения вызовем объекты DAO, либо получаем от них DataSet, либо передаем в них DataSet. Это удобно, потому что мы работаем с базой через объекты, но минус в таком подходе в том, что нам приходиться как-то ограничивать себя. Мы не можем в DAO прописать какой-то любой абстрактный запрос, который не пойми что нам вернет. Он обязан нам вернуть либо DataSet, либо набор DataSet, либо ничего, если ничего не нашел.

```
DataSet

public class UsersDataSet {
    private long id;
    private String name;

    public UsersDataSet(long id,
            String name) {
        this.id = id;
        this.name = name;
    }

```

```
13.     public UsersDataSet(String name) {
14.         this.id = -1;
15.         this.name = name;
16.     }
17.
18.     public String getName() {
19.         return name;
20.     }
21.
22.     public long getId() {
23.         return id;
24.     }
25. }
```

Как может выглядеть DataSet. Это обычный класс, в нем список полей. И в нем еще должен быть конструктор, который позволит нам этот DataSet создать и гетеры у этих полей. Ничего особенно сложного. Чтобы ORM системы понимали DataSet, его нужно будет разметить аннотациями, но об этом чуть позже в разделе про hibernate я покажу просто как этот DataSet нужно разметить, чтобы hibernate его понял. Пока что мы просто считаем, что у нас есть просто объект, и каждому объекту соответствует запись в таблице.

```
1.  DAO
2.  Объект доступа к данным
3.  Шаблон проектирования, скрывающий детали
4.  работы с базой
5.  Обычно один DAO на одну таблицу
6.  Высокоуровневый доступ к данным через DataSet
7.
```

```
 8.  Варианты операций над базой:
 9.  Вставки строки — добавление DataSet
10.  Поиск строки по ключу — возврат DataSet
11.  Поиск строки по признаку — возврат
12.  List<DataSet>
13.  Удаление строки
```

DAO в свою очередь — это тоже объект. Тот, который скрывает от нас работу с базой. Вообще говоря, DAO не подразумевает именно работу с базой. Это сокрытие работы с данными. Объект, позволяющий получить доступ к данным. А что там внутри таблицы на самом деле для DAO не так важно. Обычно, когда говорят DAO, подразумевают работу именно с базами. У класса этого объекта есть набор методов, который позволяет нам обращать к этим методам передать туда DataSet, либо получить от туда DataSet. То есть мы можем написать методы. Например, метод, который на вход получает id пользователя, а на выход дает DataSet пользователя. Или метод, который на вход получает временной интервал с какого по какой момент времени, а на выход возвращает List<DataSet> всех пользователей, который в этот момент заходили в приложение. Обычно DAO делает отдельные таблицы. На самом деле это не обязательно. В том приложении, о котором я вам рассказываю, я одному DAO

завожу на таблицу. Часто под DAO подразумевают еще более высокого уровня систему, которая вообще ко всем таблицам закрывает доступ. Например, что можно сделать с базой — можно вставлять строки, то есть добавлять DataSet, можно искать по ключу, возвращать, искать по признакам и можно удалять ее. И выглядеть это может следующим образом.

```
 1. interface UsersDAO
 2.
 3. public interface UsersDAO {
 4.
 5.     UsersDataSet get(long id)
 6.             throws SQLException;
 7.
 8.     UsersDataSet getByName(String name)
 9.             throws SQLExeption;
10.
11.     void add(UsersDataSet dataSet)
12.             throws SQLExeption;
13.
14.     void dalete(long id) throws SQLExeption;
15. }
```

Я здесь привел интерфейс. Не саму реализацию, а просто список функций. Представим, что у нас есть таблица с названием Users. Мы для этой таблицы заводим UsersDataSet и UsersDAO. И в UsersDAO перечисляем методы, которые позволят нам работать с таблицей через создание или добавку DataSet. У меня приложение некое работает. Пользователь

хочет авторизоваться, присылает свой логин и пароль. Я должен запросить базу, чтобы этот логин пароль извлечь. Что я делаю. Я обращаюсь к UsersDAO и говорю getByName() и передаю туда имя, то есть логин, который мне пользователь передал. На выход я получаю UsersDataSet. В UsersDataSet у меня описано имя, время регистрации, пароль, емайл и еще какие-то поля. Я проверяю, что пароли совпадают. И говорю, что пользователь авторизовался. После того, как этот блок работы с базой написан, дальнейшая работа с базой сводится не к написанию запросов, а к обращению к обертке, к объекту, который скрывает от нас непосредственно уже работу с базой через запрос.

3.2.7 Hibernate

Перейдем к содержательной части занятия. Мы всячески готовились до этого к тому, чтобы понять как работает и из чего состоит библиотека Hibernate. Библиотека Hibernate — это реализация принципов ORM. Плюс еще некие дополнительные особенности, о которых я сейчас вам и расскажу.

```
1.  ORM библиотека для Java
2.
3.  <dependencies>
```

```
    <dependency>
        <groupId>mysql</groupId>
        <artefactId>mysql connector java
                </artefactId>
        <version>5.1.35</version>
    </dependency>
    <dependency>
        <groupId>org.hibernate</groupId>
        <artefactId>hibernate core
                </artefactId>
        <version>4.3.8.Final</version>
    </dependency>
</dependencies>
```

Перед тем как начать Hibernate использовать нам надо с вами его выкачать. То есть у нас, во-первых, уже должна быть зависимость в pom файле на драйвер к базе, с которой вы собираетесь работать.
И заодно еще туда надо будет добавить зависимость на библиотеку. Еще раз обращаю ваше внимание, что проверяйте версии. Приложения развиваются, библиотеки тоже, поэтому версии у вас могут быть новые, другие. Единственное, что я со своей стороны хочу сказать именно про эти версии, что с этими версиями я проверил, с ними работает. С теми, которые у вас будут, у вас может и не заработать. Если что вдруг читайте почему или возвращайтесь к моим версиям. Первое, что надо сделать перед тем как с Hibernate работать — это выкачать сам Hibernate себе

через постановку зависимостей <dependencies> …

```
Configuration

Configuration configuration = new org
        .hibernae.cfn.Configuration();
configuration.setProperty(propertyName,
        propertyValue);
```

propertyName	propertyValue
hibernate.dialect	org/hibernate.dialect .MySQLDialect
hibernate.commection .driver_class	com.mysql.jdbc.Driver
hibernate.connection .url	jdbc:mysql:// localhost:3306/ db_example
hibernate.connection .username	tully
hibernate.connection .password	tully
hibernate.show_sql	true
hibernate.hbm2ddl.auto	update

После того как вы его выкачаете вам нужно его настроить. Настройка похожа на настройку работы базы через jdbc. То есть вам тоже нужно будет включить некий набор параметров. Если вы вернетесь к тому месту, где мы конфигурировали работу с JDBC, увидите, что многие из них совпадают. В случае с hibernate нам нужно создать объект конфигурации. Здесь я выписал специально полный путь конфигурации (3), потому что в библиотеке Java и вообще в тех

библиотеках, которые вы скачиваете, очень много классов с названием конфигурация. Чтобы на всякий случай не запутаться, я вам написал полное название класса конфигурации. И после того, как вы его у себя его создадите, в него нужно прописать property, чтобы дать понять hibernate с чем и как именно мы будем работать. И вы можете посмотреть на список параметров. Часть из этих параметров вам должно быть уже знакома, то есть понятно что нет смысла особенно пояснять что такое URL, потому что URL мы уже разобрали, username, userparol и драйвер тоже уже более менее должно быть понятно. (10-17) — с ними все понятно, они точно такие же как у JDBC, никакой разницы нет. Но есть еще здесь три поля дополнительных, которые являются особенностью hibernate. Первый из них — диалект. Мы с вами уже указали драйвер. То есть, казалось бы, драйвер мы уже передали, hibernate знает, что мы работаем с MySQL, зачем еще нам диалект? Диалект — это подсказка hibernate как именно формировать SQL запросы. Несмотря на то, что SQL стандартный, должен быть всеми базами одинаково понимаем, между разными базами все таки есть разница в том, как именно SQL-запросы писать. Диалект —

это подсказка hibernate как именно работать с той базой, которую передаем. Я не проверял, но по всей видимости диалектов работы с MySQL может быть несколько, иначе было бы бессмысленно передавать драйвер и еще заодно диалект. Тем не менее MySQL диалект как в том виде, в котором он здесь представлен точно есть, вы можете просто зайти в список диалектов и посмотреть что там происходит. Если вы его не укажете, то hibernate будет писать некий сайв тип запросов, стараясь никак не использовать особенности базы, потому что он не знает какая там, то есть никак не оптимизирует свои запросы, никак не использовать особенности, если он не знает как это делать. Дальше у hibernate есть параметр show_sql. Это очень удобный параметр для отладки. В примере у меня проставлен true, вы можете у себя тоже проставить true в процессе написания. Во время работы лучше убрать false, потому что если у вас выставлен true у вас в логах будут записи о том, какие именно запросы к базе делает hibernate. Посмотрите, это очень интересно на самом деле, то есть hibernate скрывает от вас запрос. Вы ему поставляете объекты, он эти объекты превращает в запросы, отправляет в базу. И наоборот вы у него

спрашиваете объекты, он понимает по тому, что вы спрашиваете какой запрос сделать, делает запросы, собирает данные, получает тот же самый ResaltSet. Собирает на основе ResaltSet объекты и вам возвращает, собственно, ORM. Задача такая. Кроме того, он может вам в лог при записях еще писать SQL-запросы, который он для этих запросов вам создал, то есть вы увидите какие запросы создают машина для того, чтобы сделать те запросы объектные, которые вы к ней делаете. И последний параметр здесь hbm2ddl.auto — это очень интересный параметр. Про него отдельный целый слайд.

```
1.  Hibernate.hbm2ddl.auto
2.
3.  Автоматически создает или проверяет схему
4.  базы при создании SessionFactory
5.
6.  validate: проверяет схему не внося изменений
7.
8.  update: обновляет схему, если находит
9.  различия
10.
11. create: пересоздает схему
12.
13. create drop: уничтожает схему при закрытии
14. SessionFactory
```

Значит hibernate при старте вы сообщаете о своих DataSet, то есть вы написали приложение, создали DataSet,

разметили DataSet, покажу пример ниже, и сообщаете hibernate вот у меня есть некий набор DataSet. На основе этих DataSet Hibernate понимает какие именно таблицы этим DataSet соответствует. Каждому DataSet соответствует своя таблица. Каждому полю в DataSet соответствует колонка. И Hibernate всю эту информацию от вас получает, то есть вы ему сообщаете классы, в классы помечены аннотации, мы вспоминаем что такое аннотации, понимаем что hibernate анализирует ваши классы, извлекает аннотации, по аннотациям понимает как работать с этими классами. И более того он на основе этих классов понимает какая структура таблиц должна быть в базе. Так вот перед тем как начать работать у hibernate есть возможность проверить а существующая в базе схема соответствует ли тем DataSet, которые вы в hibernate передали. Потому что если она не соответствует, то он запросы сделает, но база их не выполнит. Потому что там структура может быть другая. Либо в базе может быть вообще пусто. То есть базу вы создали, таблиц в ней нет. Вы сообщаете в hibernate вот у меня есть некий набор DataSet, каждому набору DataSet таблица, hibernate смотрит в базу, таблиц нет.

Так вот, у hibernate есть некий набор реакций на разные ситуации соответствия или несоответствия DataSet, которые вы ему дали, и таблиц, которые он нашел в базе. На этом слайде я выписал возможные варианты поведения hibernate. И именно их мы задаем параметром hbm2ddl.auto. Первый — это самый простой способ validate. Если hibernate работает в режиме validate, то при старте, при создании коннекта к базе hibernate проверит есть ли в базе все нужные таблицы и вообще правильная ли вообще зависимость между этими таблицами, все ли там соответствует тем DataSet, которые hibernate получил на вход перед установки коннекта, от вас он получил на вход. Если схема соответствует, то с ней дальше работает. Если не соответствует, то выкидывает вам исключения, говорит не будет с ней работать. Следующий вариант его поведения — это update. Все то же самое, что в случае с validate, только в том случае, если он находит отличие, например, нет таблицы, или видит, что в DataSet, который ему пришел, полей стало больше. Разработчик добавил еще несколько полей, а в таблице полей нет, то hibernate попытается эту схему улучшить, модифицировать так, чтобы схема

в базе соответствовала классам объектам в коде. То есть он добавит необходимые колонки в таблицу или создаст их, если их там нет. Вот это режим update и есть еще два режима create и create-drop. Create означает, что не важно что там было в базе, при присоединении мы полностью все пересоздаем. Там все дропаем, ничего там не остается и создаем заново. То есть первым делом при установке соединения hibernate смотрит, если там что-нибудь, если там что-нибудь, он все удаляет. Создает новую схему, полностью соответствующей тем DataSet, которым передали в него. И create-drop — это не просто пересоздать, а еще и почистить за собой, если при нормальном завершении приложения никаких исключительных ситуаций не случилось. При нормальном завершении приложения почистить там все за собой, просто взять и все убрать.

3.2.8 Session Factory

```
1.  Session Factory
2.
3.  Фабрика, которая создает сессии
4.
5.  Одна фабрика на поток
6.  Одна сессия на запрос
```

Картинка

После того, как вы настроили hibernate, разобрались с тем, как он будет создавать вам схему, установили соединение, вам нужно начать работать с hibernate, точнее даже вы еще не установили соединение, вы пока все настроили и проверили, что все может работать. И дальше вам нужно создать Session Factory. Как следует из названия — это фабрика, задача которой будет производить вам сессии. А сессия — это юнит, позволяющий вам делать запросы к базе. Создание Session Factory оно тяжелое, то есть процесс, требующий неких затрат. И не нужно пересоздавать сессию каждый раз. То есть по-хорошему сессию у нас должна быть одна фабрика на поток вашего приложения. В сессии, в которой фабрика будет создавать, они легкие, создание их потокобезопасное. И вы можете их создавать на каждый запрос, который вам нужен, то есть Session Factory такая большая фабрика по производству сессии, сессии относительно легкие. Задачи к базе, которые из Session Factory можно производить, и к сессии обращаться непосредственно уже к базе.

```
1.  Session Factory
2.
3.  StandartServiceRegistryBuilder builder =
```

```
 4.          new StandartServiceRegistryBuilder();
 5. builder.applySetting(configuration
 6.          .getProperties());
 7. ServiceRegistry serviceRegistry =
 8.          builder.buildServiceRegistry();
 9.
10. SessionFactory sessionFactory = configuration
11.          .buildSessionFactory(
12.          serviceRegistry);
13.
14. Session session = sessionFactory
15.          .openSession();
16. Transaction transaction = session
17.          .biginTransaction();
18.
19. System.out.append(transaction
20.          .getLocalStatus().toString());
21.
22. Session.close();
23. sessionFactory.close();
```

Раньше инициализация фабрики в версиях hibernate была проще, теперь нужно создать много особенных билдеров.
Есть ServiceRegistryBuilder, нужен сделать. Потом у регистриБилдера нужно попросить ServiceRegistry, создать sessionFactory, построить sessionFactory на основе конфигурации и ServiceRegistry. То есть (3-10) я предлагаю вам скопировать себе в то место, где вы будете с hibernate работать. Те, кому интересно, разберите его подробнее. Зачем же именно вот так было сделано. Раньше можно было sessionFactory по конфигурации сделать.

Потом решили, что так deprecated. Решили, что (3-10) лучше. Больше абстракции, больше фабрик, больше возможностей, модификации. Наверное, они это хотели сделать, разработчики. После того, как вы sessionFactory сделали, вы можете у нее запросить openSession() (14). Вы у сессии можете начать транзакции (16). В моем примере я ничего особенного с базой не делал, я просто данные транзакции запрашиваю. По-хорошему в (21) должны быть запросы на извлечение, вставку. Я покажу пример как именно это будет выглядеть. И после того, как вы со всем этим поработали, вы можете закрыть сессию (22). Если вы закончили работать и с фабрикой тоже, то закрыть и фабрику тоже (23).

```
 1.  Аннотации для DataSet
 2.
 3.  import javax.persistence.*;
 4.
 5.  @Entity
 6.  @Table(name = 'users')
 7.  public class UserDataService {
 8.      @Id
 9.      @Column(name='id');
10.      @GeneratedValue(strategy = GenerationType
11.              .IDENTITY)
12.      private long id;
13.
14.      @Column(name = 'name')
15.      private String name;
16.     ...
```

```
17.  }
18.
19.  Перед созданием SessionFactory:
20.  Configuration.addAnnotatedClass(UserDataSet
21.          .class);
```

В этом же разделе я еще раз хочу вернуться к аннотациям и к DataSet. Помните, у нас была DataSet, UserDataSet. (7) — это он же. Это класс, в котором мы перечислившем поля, соответствующим колонкам таблицы. То есть у нас есть таблица с пользователем и мы ее в соответствии ставим класс. Чтобы передать его в hibernate, мы должны перед созданием фабрики в конфигурацию передать все классы всех DataSet, которых мы хотим использовать. И перед тем, как его передать в коде разметить. Мы должны из javax.persistence его разметить как @Entity (5), сказать, что какой таблице он соответствует, отметить колонки, которые есть в @Entity специальной аннотацией и в моем примере еще есть поле id и стратегия как это id генерировать. Если у вас несколько разных DataSet, вы все эти DataSet должны передать в конфигурацию через вызов addAnnotatedClass() (20). После этого создаете уже фабрику и можете обращаться к базе.

3.2.9 Session

```
1.  Session
2.
3.  Основной интерфейс между приложением
4.  и библиотекой
5.  Время жизни сессии соответствует времени
6.  жизни транзакции
7.  Задача сессии — работа с объектами
8.  проанатированными как @Entity
```

Session Factory мы сделали. Для этого мы сконфигурировали, мы установили соединение, в момент создания SessionFactory соединение с базой есть. Нам нужно к базе написать запросы. И вот запросы происходят через сессии. Это некий аналог statement или же JDBC. Если так проводить параллели — не совсем точно, но можно так сказать. Там у вас был connection, вы у него получили statement. Здесь у вас Session Factory, вы получаете сессию. На каждый запрос вы создаете по сессии. То есть работа с сессиями можно организовать через транзакции. То есть вы начинаете транзакцию сессии, заканчиваете транзакцию сессии. Задача сессии — это получать, точнее в коде в объект сессии мы передадим запрос, а в момент исполнения этого запроса в сессию придет ответ. И ответ нам будет сформирован уже в виде объекта, содержащего @Entity.

Если вы вспомните Executor, то вот здесь Executor спрятан внутри сессии.

```
 1.  Session
 2.
 3.  public void save(UserDataSet dataset) {
 4.      Session session = sessionFactory
 5.              .openSession();
 6.      Transaction trx =
 7.              session.beginTransaction();
 8.      session.save(dataSet);
 9.      trx.commit();
10.      session.close();
11.  }
12.
13.  public UserDataSet read(long id) {
14.      Session session = sessionFactory.
15.              openSession();
16.      return (UserDataSet)session.load(
17.              UserDataSet.class, id);
18.  }
```

Например, DAO, которое работает с UserDataSet, может выглядеть следующим образом. Вы можете в ней создавать сессию (4), начинать транзакцию (6), передавать в сессию просто вызовом у сессии метода save() (8) объект DataSet. То есть hibernate сам понимает, что это именно UserDataSet. Ему не нужно сообщать класс. Класс у него уже есть. Вы в конфигурации этот класс ему передали. Вы просто передаете объект. И говорите транзакцнКоммит (9). В этот момент, в save() уже полетит запрос к базе.

А в транзакшнКоммит этот запрос будет уже сохранен самой базой. И после этого вы закрываете сессию (10). То есть таким образом в этом участке кода ни строки не записано про запрос. В запрос, который будет отправлен к базе, он полностью на совести hibernate. Вы в DAO передаете объект. Вы в сессию передаете этот объект, в функцию save() сессии. Внутри функции save() hibernate берет классы в объекты, понимает какая этому классу соседствует таблица. На основе данных полей, объекты которого мы передали, формирует запрос на вставку данных в эту таблицу и отправляет этот запрос в базу.
И примерно то же самое происходит при чтении. При чтении мы допустим хотим извлечь запись по идентификатору. У нас в DataSet есть id. Мы хотим его извлечь. Здесь мы точно так же создаем сессию. Здесь нам уже никакие транзакции не нужны, потому что мы же извлекаем данные.
И единственное что мы вызываем у сессии load(), но в этот load() мы должны передать класс, который мы хотим прочитать. Что здесь происходит.
Мы создали сессию, в сессию передали класс, который мы хотим прочитать, объекты, которые мы хотим прочитать.
И идентификатор записи таблицы,

по которому мы хотим искать эту запись. В функции load() hibernate, получив на вход класс, ищет какая таблица соответствует этому классу, обращается к этой таблицы с запросом, в котором указан идентификатор, получает ResaltSet. ResaltSet разбирает ResaltSet, превращая его в UserDataSet. В наш же тип, не в стандартный, в наш собственный и нам его возвращает. Мы потом из UserDataSet можем его уже передать дальше в наше приложение, где хотим использовать. Не написав ни одного sql запроса здесь, мы получили возможность сохранять и читать данные. Естественно, возникает вопрос как быть в случае с запросами более сложными. Например, поиск не по id, а поиск по имени. Или селект какого-то набора данных. Об этом в следящей части в HQL.

3.2.10 HQL

Продолжаем тему запросов к базе через hibernate. Самое время рассказать про HQL. То есть это такой язык запросов, с помощью которого мы формируем задачу для hibernate. И если SQL — это язык, в котором мы запросы формируем строкой, то в HQL мы запросы формируем объектами, то есть мы формируем запрос, составляя запрос вызовами методами разных объектов.

```
 1. HQL
 2.
 3. Hibernate Query Language
 4. «Язык» написания запросов «к hibernate»
 5.
 6. Вместо написания запросов к базе — описание
 7. запроса вызовами методов сессии
 8.
 9. Ссылка
10. Ссылка
```

Давайте посмотрим как это происходит. Вместо того, чтобы писать запросы к базе напрямую, мы можем составить в коде последовательность вызовов методов объектов так, чтобы hibernate понял, что именно мы от него хотим. Вот это как раз и есть HQL. Я сейчас сделаю только краткий обзор основных возможностей. За деталями обращайтесь к документации. Этот минимум, который я нашел. Дальше по ключевым словам вы сможете найти уже все, что вам нужно будет. У hibernate есть режим, когда вы делаете запросы над строкой. То есть вы говорите хочу обратиться к базе числам SQL. Делать стоит в случае крайней необходимости, потому что ни диалекты вы таким образом не используете, ни кеширования hibernate, ничего. То есть могут некие странные … эффекты, если вы пишите прямые SQL запросы вместо того, чтобы использовать hibernate язык.

```
HQL вставка

public void save(UserDataSet dataset) {
    Session session = sessionFactory
            .openSession();
    Transaction trx = session
            .beginTransaction();
    session.save(dataSet);
    trx.commit();
    session.close();
}
```

Пример с сохранением я вам уже показывал. Вот он еще раз. И если вам нужно сохранить DataSet, то вы передаете этот dataSet в функцию session save(). И hibernate сам понимает, как ее сохранять. То есть это один из самых простых примеров. Что именно и как сохранять hibernate понимает внутри себя, то есть save() (8) — это уже пример HQL. Потому что мы обратились к базе, сказали вставить данные через вызов метода save() сессии и передачи туда нужного нам dataSet.

```
HQL чтение

public UserDataSet read(long id) {
    Session session = sessionFactory
            .openSession();
    return (UserDataSet) session.load(
            UserDataSet.class, id);
}
```

Если мы хотим прочитать DataSet, то мы должны точно так же сформировать запрос через вызов метода сессии. Но в этом случае метод уже будет load(). Мы в него передадим класс того, что мы хотим загрузить и идентификатор того, где искать (7). То есть это тоже пример HQL.
И обратите внимание, что на выход (6) вам сессия вернет объект просто object.
Тип базовый для всех объектов, всех классов. И вы сами должны его скастить к тому, что вы передали (7).
По-хорошему hibernate должен вторым … первым параметром в load() понимать, то есть надо было бы, мне кажется, здесь бы хорошо подошла типизация. То есть функция load() должна была брать на вход класс определенного типа и понимать какого именно типа она вам должна вернуть параметры. Но библиотека hibernate не типизирована. То есть это некий минус ее. Не знаю почему, может быть какие-то исторические предпосылки к этому есть.

```
1.  HQL поиск по ключу
2.
3.  public UserDataSet readByname(String name) {
4.      Session session = sessionFactory
5.              .openSession();
6.      Criteria criteria = session
7.              .createCriteria(
8.              UserDataSet.class);
9.      return (UserDataSet) criteria.add(
```

```
10.                 Restrictions.eq('name', name))
11.                 .uniqueResult();
12.     }
```

Более сложный пример. Представим, что вам нужно найти не просто запись по id. В hibernate, кстати, вот думаю хорошо будет сейчас об этом сказать. Hibernate требует, чтобы у всех таблиц был первичный ключ BigInteger идентификатор. Именно имея у каждой записи этот идентификатор. Почему он еще должен быть автокриментый. hibernate позволяет удобно по id находить записи без использования того поиска, который я сейчас вам покажу в случае, если вам нужно найти не по id, а по какому-то другому полю, не важно есть ли на нем индекс или нет на нем индекса. Задача. Нам нужно найти userDataSet в таблице, которой этому usetDataSet соответствует по имени. Предположим, что у нас в userDataSet id и имя. И мы хотим не по id найти, а найти по имени. Мы так же, как в предыдущих примерах, создаем сессию (4). То есть в этом смысле ничего у нас не поменялось, но в добавок мы теперь создаем критерий. Это как раз один из элементов HQL. Language, состоящий из объектов и обращений методов к этим объектов. То есть вы запрос формируете объектами.

Что такое критерий? Критерий — это вы запрашиваете у сессии (6) дайте мне критерию для класса (8). И после этого вы в критерии можете добавлять некие сужения (9), рестришены на то, что искать. В данном случае я говорю Restrictions проверь, чтобы поле name было равно тому значению, которое я передаю. Обратите внимание вот это name. Здесь на самом деле очень интересно. name — это не имя колонки в базе. Если бы это было имя колонки в базе, то произошло бы некое нарушение абстракции. Вам бы нужно было знать структуры колонок в базе для того, чтобы написать запрос. А по задумке вы не обязаны это знать. Поэтому это name — это не колонка в базе так называется, это название поля, в котором лежит имя внутри userDateSet. Обычно название поля совпадает с названием колонки. Поэтому часто, если вы думаете, что здесь название колонки пишите, у вас будет работать, но не всегда. Может быть такая ситуация, когда поле в dataSet отличается от названия поля в колонке. И в этом случае надо помнить, что здесь при формировании запроса нужно указывать имя поля, а не имя колонки. Вернемся к примерам. Вы задаете Restrictions (10) и в него передаете имя поля и значение

поля в userDataSet, которое нужно искать. И после этого вы подразумеваете, что ответ будет всего один, поэтому пишите (11). И после выполнения этого (11) вам на выход hibernate снова вернет собранный userDataSet, либо null, если он ничего не найдет.

```
 1.  HQL получение всех записей
 2.
 3.  public List<UserDataSet> readAll() {
 4.      Session session = sessionFactory
 5.              .openSession();
 6.      Criteria criteria = session
 7.              .createCriteria(
 8.              UserDataSer.class);
 9.      return (List<UserDataSet>) criteria
10.              .list();
11.  }
```

И в заключении, если вам нужно найти не один элемент, а набор элементов. То вы можете точно так же создать критерий (6) и спросить List (9). Если вернуться к предыдущему примеру, взять из этого предыдущего примера добавление Restrictions, то вы ее же можете добавить в этот пример, но только вместо equals поставить, например, in и передать массив, в котором искать среди каких элементов искать. Тогда вы возьмете не весь список. То есть вот в текущем виде она вам вернет вообще все значения, какие найдет. То есть все колонки, для каждой колонки

из таблицы, для всех какие найдет в этой таблице, она вам сформирует UserDataSet, запишет его в лист и вернет. Можно сюда же в этот критерий добавлять сужения на извлечение данных. Можно с помощью HQL устанавливать порядок выдачи.
Можно считать количество элементов.
Можно задавать лимит по запросу. Это уже просто дополнительные вещи. Если они вам понадобиться, то вы сможете сами в документации о них почитать.

Видеоинструкция