Оглавление

Термины и сокращения

1.	JDBC — Java Database Connectivity
2.	
3.	ORM — Object Relational Mapping
4.	
5.	JPA — Java Persistency API
6.	
7.	Hibernate — популярная библиотека для ORM,
9.	implements JPA
10.	
11.	DAO — Data Access Object

JDBC — это название библиотеки, которая входит в состав стандартной библиотеки. Здесь собраны все нужные для работы с базами классы и интерфейсы. Можно сказать, что это некоторая API для Java-программистов и разработчиков баз данных.

ORM: Object — объекты (Java-программистов), Relational — таблицы (база данных), Mapping (перекладывание одного в другое).

JPA является тоже частью стандартной библиотеки, содержащей набор интерфейсов, классов и *аннотаций*. В отличии от JDBC, JPA — это API между Java-программистами и разработчиками ORM решений.

Hibernate — одно из самых популярных ORM решений, использует JPA.

DAO — это объект, позволяющий либо получить доступ к данным, либо записать данные в базу.

Инструменты

```
https://dev.mysql.com/downloads/

MySQL Community Server
MySQL Workbench
MySQL Connector Connector/J

<dependency>
    <groupId>mysql</groupId>
    <artefactId>mysql-connector-java
            </artefactId>
    <version>5.1.35</version>
<dependency>
```

JDBC

```
JDBC — API для работы с базами из приложения
на Java

Предназначена для работы с реляционными
базами данных

Предоставляет методы для получения
и обновления данных

Не зависит от конкретного типа базы

Java Application -> JDBC API -> JDBC Driver
Manager ->(1)
(1) -> JDBC Driver -> Oracle
(1) -> JDBC Driver -> SQL Server
(1) -> JDBC Driver -> ODBC Data Source
```

JDBC API

```
Connection
```

```
2.  Объект отвечает за соединение с базой и режим
3.  работы с ней
4.
5.  Statement
6.  Объект представляет выражение обращения
7.  к базе
8.
9.  ResaultSet
10. Объект с результатом запроса, который вернула
11. база
```

JDBC Driver Manager

```
1.  Название класса драйвера
2.  com.mysql.jdbc.Driver
3.
3.  Создание объекта драйвера при помощи
4.  reflection:
5.  (Driver) Class.forName("com.mysql.jdbc
6.          .Driver").newInstance()
7.
8.  java.sql.DriverManager — класс хелпер
9.  для работы с драйверами
10.
11. Регистрируем драйвер:
12. DriverManager.registerDriver(driver);
```

Connection

```
1.  Driver driver = (Driver) Class.forName(
2.          'com.mysql.jdbc.Driver')
3.          .newInstance();
4.  DriverManager.registerDriver(driver);
5.
6.  StringBuilder url = new StringBuilder();
7.
8.  url
9.  .append('jdbc:mysql://') // db type
10. .append('localhost:') // host name
11. .append('3306/') // port
12. .append('bd_example?') // db name
```

```
13.  .append('user-tully&') // login
14.  .append('password-tully'); // password
15.
16.  // URL: "jdbc:mysql://localhost:3306/
17.  // lecture_db?user-tully&password-tully"
18.
19.  Connection connection = DriverManager
20.          .getConnection(url.toString());
```

Statements

```
 1.  JDBC позволяет создавать и выполнять запросы
 2.  к базе
 3.  Update statements: CREATE, DELETE, INSERT...
 4.  Query statements: SELECT
 5.
 6.  Интерфейс
 7.  Statement
 8.  PreparedStatement
 9.  CallableStatement
10.
11.  Query statements возвращают ResultSet
12.  Update statements возвращают число измененных
13.  строк
```

PreparedStatement. В отличии от Statement не придется каждый раз создавать новый Statement, а можно сделать шаблон запроса и исполнять его, подставляя в него конкретные значения.

CallableStatement нужен для работы с хранимыми процедурами.

ResaulrSet

```
 1.  Объект, содержащий результаты запроса
 2.  (внутри таблица)
 3.
```

```
Перемещение по строкам
next()
previous()
isLast()

Доступ к полям текущей строки
По имени колонки: getBoolean(String name),
getLong(String name) ...
По индексу колонки: getBoolean(int index),
getLong(int index) ...
```

close()

```
close()

Все обращения надо закрывать вручную
resultSet.close()
statement.close()
connection.close()

Или использовать AutoCloseable
```

3.1.10 Транзакции

```
Transactions

По умолчанию autocommit после выполнения
каждого statement

void setAutoCommit(boolean autoCommit) —
вкл/выкл автокоммита

void commit()

void rollback()
```

Картинка

Рассказ про работу с базами и с Java будет не полным, если не упомянуть еще транзакции и Prepared Statements.

Транзакции, как следует из названия, — это действие, которое могут быть выполнены либо целиком, атомарно, либо не выполнены совсем. То есть если мы какие-то изменения должны сделать транзакционно, то это означает, что мы либо все эти действия применяем, либо у нас, если мы отменяем эти действия, то так же отменяем все полностью. Например, это может быть обмен предметами ваших пользователей друг с другом. Или продажа предметов одного пользователя другому. В этом случае ваши пользователи будут не рады, если по результатам обвала системы, например, у одного пользователя исчезнет предмет, а у другого он не появится. Или у кого-то появится 2 предмета, а у кого-то не появится. В любом случае, особенно там списаны деньги, а предмета не будет. При этом работа на изменение должна затрагивать сразу несколько таблиц в базе. То есть вам нужно не просто в одной базе чтоб переставить, а сразу в нескольких таблицах что-то поменять при этом нужно сделать так, чтобы это было надежно. Для этого и нужны транзакции. Сразу скажу, что транзакции должны быть поддержаны в базе. Что если база транзакцию не поддерживает, то никакими ухищрениями JCBC вы этого не добьетесь.

Поэтому я и призывал вас работать, по крайней мере в таких критически важных участках вашего приложения, именно с реляционными базами, потому что они транзакции поддерживают. Если смотреть на транзакционность с точки зрения нас, разработчиков программного обеспечения, использующих JDBC, то для нас важно понятие auto-commit. auto-commit — это режим, когда после каждого вашего statement на изменение база получает приказ на сохранение этого изменения на запись к себе. Либо залогировать как минимум у себя. По умолчанию auto-commit включен. То есть когда вы по умолчанию используете JDBC, пишите statement на изменение, после выполнение каждого statement результата будет сохранен в базу. Если нам нужна транзакционность, то нам такой вариант не подходит, потому что нам нужно сделать несколько обращений к базе, возможно, к нескольким разным таблицам. И только после того, как мы выполнили последнее, мы должны сказать а теперь все вместе сохрани. Поэтому мы должны у себя setAuto-commit выключить, то есть явным образом сказать auto-commit больше не использовать. Если мы это сделали, то нам явным образом нужно будем по результатам всех наших действий вызвать

явным образом commit(), то есть применить все изменения. Либо, если произошли какие-то проблемы в процессе исполнения запросов, то мы должны сделать явным образом rollback(). Если мы сами его явным образом не сделаем, то база сама его через некоторое время сделает. Кроме того, есть такое понятие как Savepoint, когда мы сделали некое количество …, сохранили, если нужно откатили к SavePoint, если нужно откатились в самое начало.

```
Transactions

public void execUpdate(Connection connection,
        String[] updates) {
    try {
        connection.setAutoCommit(false);
        for(String update: updates) {
            Statement stmt = connection
                    .createStatement();
            stmt.execute(update);
            stmt.close();
        }
        connection.commit();
    } catch (SQLException e)
        try {
            connection.rollback(true);
            connection.setAutoCommit(true);
        } catch (SQLException ignore) {}
}
```

В виде примера это выглядит следующим образом. Представьте, что у вас есть некий набор строк с update (4). Вы должны первым делом вы выключить у connection

setAutoCommit() (6). После этого вы можете в цикле создавать statement (8), выполнять statement (10) и закрывать их (11). Каждый раз на каждую строку. И по результатам после выхода из цикла взять и все вместе применить (13). Если по каким-то причинам у вас что-то пошло не так, вы можете у connection опять же вызвать rollback() (16), то есть все откатить. Вот и все. Выглядит оно странно в том виде, что нам надо каждый раз создавать новый statement на этот запрос. И чтобы этого не делать как раз следующий пример про Prepared Statements.

```
 1.  Prepared Statements
 2.
 3.  public void execUpdate(Connection connection,
 4.          Map<Integer, String> idToName) {
 5.      try {
 6.          String update = 'insert into
 7.                  users(id, user_name)
 8.                  values(?, ?)';
 9.          PreparedStatement stmt = connection
10.                  .prepareStatement(update);
11.
12.          for(Integer id: idToName.keySet()) {
13.              stmt.setInt(1, id);
14.              stmt.setString(2, idToName
15.                      .get(id));
16.              stmt.executeUpdate();
17.
18.          }
19.          stmt.close();
20.      } catch (SQLException e)
```

```
21.             e.printStackTrace();
22.     }
```

То есть с транзакциями разобрались, транзакции нужны для того, чтобы сохранять блог изменений атомарно, а Prepared Statements нам могу в этом случае помочь. Например, если нам нужно сохранить однотипные запросы целиком. Объединив этот пример и предыдущий, мы с вами получим следующее. Мы можем создать уже не statements, а Prepared Statements (9). Создается точно так же через обращение к connection и создание Prepared statements передачей в него строки с запросом. Но строка с запросом отличается от предыдущий тем, что в ней есть еще поля, отмеченные знаком вопроса (8). Те поля, которые будут заданы потом перед вызовом. В цикле мы можем проставлять поля (13-14). Опять же обратите внимание, как в базах, у нас здесь единичка и двойка. То есть поля, нумерация полей начинается с единицы. Потом мы можем вызвать execteUpdate() (16). В текущем примере после execteUpdate() сразу пойдет запись в базу. Если мы объединим этот пример с предыдущим, выключим комит, накидаем изменения, потом сделаем их целиком, то мы много что сэкономим. Мы сэкономим

на подготовке statement. У нас statements будет один. Мы не нужно каждый раз его закрывать. И на запись сразу пачку изменений в базу без необходимости сохранять каждый раз после каждого обращения.

Executor

```
 1.  Объект, который содержит методы для работы
 2.  с запросами
 3.
 4.  Обработка запроса на создание, вставку,
 5.  обновление и удаление
 6.  execUpdate(Connection connection,
 7.          String update)
 8.
 9.  Обработка запроса на получение данных
10.  execQuery(Connection connection,
11.          String query, ResultHandler handler)
12.
13.  public interface ResultHandler {
14.      void handler(ResultSet result)
15.              `throws SQLException;
16.  }
```

В каждом нашем действии работы с базой есть некая неизменяемая часть — это создание statement, стандартная работа с ним, исполнение statement и закрытие statement. Не зависимо от того, что вы хотите выполнить. Вы хотите извлечь какие-то определенные типы данных, определенные результаты из таблицы, или вы хотите поменять их — все равно эти

действия нам нужно делать. И нам эти действия нужно куда-то спрятать. Чтобы у нас были люди, ответственные за написание той части, которая работает с базой, и люди, которые допустим готовят непосредственно сами строки запросов и получают результат. И Executor — это та самая часть, которая общая для всех запросов какие бы они ни были. Когда вы начнете работать с базой, вы довольно быстро поймете, что вам приходится делать одно и те же очень часто. В каждой функции работой с базой вы производите одни и те же действия. Когда вы попытаетесь эти действия выделить вы столкнетесь с неким набором трудностей. Сейчас я попытаюсь эти трудности разрешить с помощью объяснения Executor. Во-первых, это класс. В этом классе у нас будет, мы можем создавать объекты этого класса. В нем можем хранить connection. Connection, как я уже говорил, не нужно создавать каждый раз при каждом запросе. Connection вы создали и держите. Если вы в момент выполнения запроса поняли что Connection нет, разорвалось почему-то, вы можете пересоздать Connection.
Но создавать Connection в каждом обращении плохо, это очень сильно ресурсоемко задача. Поэтому создаем либо один Connection, если у нас простое приложение,

либо несколько и организуем их в пул, если более сложное. Так вот, в Executor у нас уже есть Connection, мы его туда при создании Executor передали, мы его там храним. Либо мы можем Executor сделать статическим, тогда нам надо будет Connection нам каждый раз туда передавать. Так вот, как у меня здесь написано. Я здесь подразумеваю, что это статические функции, то есть в этом примере у меня Executor это некий хелпер. Значит в Executor у нас будет 2 метода execUpdate() и execQuery(). Разница у них в том, что от execUpdate() нам не нужно возвещаемое значение. То есть мы можем в execUpdate() возвращать количество измененных строк. То есть мы заранее знаем что именно нам вернет execUpdate(), он вернет Integer, количество измененных строк. Мы в него будем передавать Connection и будем передавать строку, содержащую запрос. С точки зрения программиста, использующего Executor, в Executor 2 метода. Один метод довольно прост, это execUpdate(). В него мы должны передавить Connection, в него мы должны передать SQL-запрос и на выходе получить количество измененных строк, либо исключение, если что-то пошло не так. С получением данных чуть сложнее, потому

что в query, который уходит в получение данных в зависимости от того, какую query мы передаем, мы получим разные результат. То есть в случае с update там все просто, просто Integer вернулся всегда и все. В случае с query там может вернуться что-то разное. Может вернуться одно значение, может вернуться строка, может вернуться лист строк. И нам нужно каким-то образом понимать как именно, какое именно значение из этих query должно вернуть, то есть смотрите, это Executor, в нем execQuery(), она одна функция. Не по execQuery() на каждый запрос, а она всего одна. Она на вход получает Connection, query и еще она получает ResaultHendler. Вот ResaultHendler — это передача функции в функцию, о которой я до этого рассказывал, то есть ResaultHendler — это объект класса, содержащего функцию, которая поработает с ResaultSet. Дальше на примерах я поясню как именно мы будем это делать. Заранее я сразу скажу, что все это придумано за тем, чтобы не возвращать из execQuery() ResaultSet. Если мы в принципе ну может показаться что это нормальное решение, взять из execQuery() вернуть ResaultSet. Но в этом случае мы поручаем тому, кто вызывает Executor, задачу закрытие

ResaultSet и закрытие statement.
Потому что если мы statement закрыли,
RecaultSet у нас будет тоже закрыт.
То есть получается execQuery() может нам
открыть запрос, запросить данные,
но не может закрыть его. И это очень
плохо, потому что это раскрытие деталей
работы с базой и надежда на то,
что вызывающая сторона выполнить все
закрывающие действия, кроме того у нас
в общем то, если мы возвращаем ResaultSet,
а statement не возвращает, то statement
окажется не закрытым. То есть плохая идея
возвращать ResaultSet из execQuery().
Те трудности, о которых я говорил,
с которыми вы встретитесь. Вот одна
из них. Что возражать из execQuery().
Сейчас мы попытаемся эту проблему решить.
С execUpdate() с ним это сильно просто.
С execQuery() сложнее. Тут появляется
ResaultHendler и еще потом я подключу сюда
шаблоны, чтобы показать как лучше всего
это место организовать. Итак, здесь
на этом слайде я привел интерфейса
ResaultHendler. У него будет одна
единственная функция hendle(). Эта функция
hendle() будет брать на вход ResaultSet.
То есть мы таким образом в execQuery()
передаем наш код, тот код, который на вход
получит ResaultSet.

3.1.8 Executor. execUpdate()

```
Update Statement

public static int execUpdate(Connection
        connection, String update) {
    Statement stmt = connection
            .createStatement();
    stmt.execute(update);
    int updated = stmt.getUpdateCount();
    stmt.close();
    return updated;
}

Примеры

execUpdate(connection, 'create table users
        (id bigint auto_increment, name
        varchar(256), primary key (id))');

execUpdate(connection, 'insert into users
        (name) values ('tully')');
```

Пока вернемся к execUpdate(), чтобы на примере посмотреть как execUpdate() работает. Дальше execQuery() тоже очень подробно разберем. Значит execUpdate(). С ним все относительно просто. Мы в нем должны создать statement, вызвав createStatement() у connection. Мы должны statement попросить выполнить update.

update это так самая строка с update, которая к нас пришла. Когда строка Execut прошла, то вы можете считать, что задача в базе выполнена. То есть мы до этого места дошли, эту строку запустили. И если мы перешли на следующую строку без исключений, то есть исключение никакое не вылетело, то все, update прошел. Здесь сразу хочу сказать и еще вернусь к этой теме чуть позже. У нас обращение к базе синхронный. То есть мы действительно ждем, пока обращение будет сделано. После того, как мы это обращение сделали, мы у statement можем спросить количество строк, которые были обновлены. Мы можем закрыть statement. На самом деле код тут будет чуть более сложным, потому что тут еще всевозможные исключения могут быть выкинуты. Я именно в реальных примерах, когда мы до разбора примеров в среде разработки разберемся, я еще раз обращу внимание, что просто. Это иллюстративный пример, просто закрыли и все хорошо. Но закрыть мы обязаны здесь. (9) строка должна быть либо выполнена явно, либо мы должны AutoCloseable здесь сделать решение. И после этого мы знаем, что нам нужно вернуть Integer, мы этот Integer возвращаем. После того, как вы эту часть сделали, у вас уже есть Executor, в нем

есть этот метод. Вы можете в своем приложении где угодно, где вы хотите работать с базой, взять его и позвать и сказать execUpdate, передать connection и передать строку с запросом.
В строке (15) будет создана таблица с двумя полями. И потом при желании вы можете другой запрос (19) отправить на insert. Какой хотите запрос: на удаление строк, на удаление таблиц. execUpdate с этим справится. Вам вернет результат. И все в этом месте у вас будет хорошо.

3.1.9 Типизация. Executor. execQuery()

```
 1. public interface TResultHandler<T> {
 2.     T handle(ResulteSet resultSet)
 3.             throws SQLException;
 4. }
 5.
 6. public class TExecutor {
 7.     public <T> T execQuery(Connection
 8.             connection, String query,
 9.             TResultHandler<T> handler)
10.             throws SQLException {
11.         Statement stmt =
12.                 connection.createStatement();
13.         stmt.execute(query);
14.         ResultSet result =
15.                 stmt.getResultSet();
16.         T value = handler.handle(result);
17.         result.close();
18.         stmt.close();
19.         return value;
20. }
```

Ну вот теперь самая сложна часть этой части урока. Нам с вами нужно не просто возвращать void из execQuery(). Нам нужно из execQuery() возвращать значение. Мы же не просто просили написать в лог результат. Нам же на самом деле нужно спросить Executor извлечь некие данные, передать в Executor строку с запросом и еще на выход от него получить значение, которое мы хотим. То есть мы хотим, чтобы execQuery() сам там себе внутри что-то сделал при работе с базой. Работал непосредственно с statement, открыл его, выполнил, закрыл. А нам на выход вернул нужное нам значение. Причем это должен быть не ResaultSet. И вот, чтобы эту задачу решить, нам с вами понадобиться дженерики. На слайде я привел пример того, как это можно организовать. Нам для этого с вами понадобиться типизация интерфейса. Первый пример (1) — это модифицированный ResultHandler, типизированный неким значением. Я переписал ResultHandler, добавил в него типизацию. И теперь handle() возвращает теперь не void, а тот самый тип, который типизирован сам handler. Создавая ResultHandler для разных типов, я захочу получить от handler разные значения. То есть я знаю, что я хочу получить на выход. И это, что я хочу

получить на выход, я как раз и передаю в качестве параметра типизации ResultHandler. Executor мне тоже нужно будет поменять. execQuery() у меня будет типизированная функция (7). Я здесь говорю эта будет работать с параметром. И эта функция будет возвращать тип, который я захочу задать при ее использовании. И этот же самый тип в точности этот же тип я буду передавать в handler (9), то есть я буду говорить handler у меня будет именно того типа, который будет возвращать execQuery(). Дальше по коду самого execQuery() практически все то же самое, что было в предыдущем пример. Мы создаем statement (11). Мы исполняем этот statement (13). Мы получаем ResaultSet (14), и мы передаем ResaultSet внутрь handler (15). Пока ничего не изменилось. Но внутри handler мы производим действия, которые приводят к сборке нужного нам результата, этого самого T value. То есть мы в handler описываем как именно собрать тип, который handler вернет в execQuery() и который в результате execQuery() вернет нам после вызова (19). То есть смотрите что у нас получилось. Мы подготовили statement. Этот statement должен собрать нам строку, то есть результат исполнения statement будет строка. Мы запрашиваем

базу, говорим данное имя … и в результате строка. Мы знаем, что это должна быть строка. Мы создаем execQuery(), типизированной строкой, создает ResultHandler, типизированной строкой. В handler мы собираем строку на основе данных из ResaultSet и эту строку возвращаем. Если нам с вами понадобиться от execQuery() не собрать строку, а собрать List строк. Допустим, мы говорим дай нам имена всех пользователей, которые вчера заходили днем на наш сервис. И это будет список. Список имен, список строк. Поэтому мы в execQuery() говорим: execQuery() ты будешь возвращать список, вот тебе handler, который хочет список. Типизированный списком. И в этом handler мы на основе ResaultSet список собираем и возвращаем.

```
 1. TExecutor execT = new TExecutor();
 2. String query = "selest user_name from users
 3.         where id 1";
 4. String name execT.execQuery(
 5.     connection,
 6.     query,
 7.     new TResultHandler<String>() {
 8.
 9.         public String handle(ResultSet
10.                 result) throws SQLExeption {
11.             result.next();
12.             return result.getString(
13.                     'user_name');
14.     }
```

```
15. });
16.
17. System.out.append('User: ' + name + '\n');
```

Пример, который иллюстрирует то, что я сейчас говорю, именно со строками. То есть смотрите. Мы создаем Executor (1). Логично было сюда connection положить. Либо не создавать его, использовать статический. Не важно сейчас. У нас есть Executor. У нас есть строка (2). Строка, которая запрашивает по id из пользователей имя. Так как мы знаем, что нам должны вернуть строку, мы говорим, что будет строка на выходе (4), Executor типизирован строкой. Handler (7), который мы передаем сюда, тоже типизирован строкой. Все эти типы данных должны совпадать. Если мы захотим поменять запрос так, чтобы запрос возвращал не строку, а допустим long, мы должны будем во всех этих местах, где сейчас стоит String, написать long. Таким образом мы переключим Executor со строк на другой тип данных. И в handler (9) мы, как и раньше, получаем ResaultSet. Мы уже знаем, что именно с ним нужно делать. Нам надо с него строку получить. То есть мы в (12) должны получить строку. Мы эту строку получаем и возвращаем. Все. То есть мы с вами по результатам этого подхода организовали работу с базой таким

образом что у нас появилось две функции. Одна execUpdate(), другая execQuery(). И больше нам функция для работы с базой не нужно. Мы меняем запросы. Создаем разные запросы по извлечению или по модификации данных и внутри execQuery происходит вся та черная работа по созданию statement, по обращению к базе, по закрытию statement. А мы можем просто execQuery вызывать и не беспокоиться о том, что внутри там написано. Просто мы должны ему передать нужный handler. Чтобы он нужные нам данные организовал в нужный нам тип данных. Если вернуться к теме про лямбды, которые мы с вами разбирали до этого особенности 8 Java и вы хотите уже у себя начинать использовать, то весь вот этот пример можно переписать на упрощенную форму записи.

```
1.  TExecutor execT = new TExecutor();
2.  String query = "selest user_name from users
3.          where id 1";
4.  String name execT.execQuery(
5.      connection,
6.      query,
7.      result -> {
8.          result.next();
9.          return result.getString('user_name');
10.     }
11. );
12.
```

```
13.  System.out.append('User: ' + name + '\n');
```

Здесь компилятор понимает тип значения handler сам. То есть я задаю строку (4) и больше эту строку нигде не упоминаю. О том, что конструкция (7-10) должна обработать ResaultSet и вернуть строку ясно только, если мы зайдем внутрь execQuery (4) и посмотрим, как именно она устроена. Давайте сейчас вернемся к предыдущему слайду, и у нас здесь есть то же самое место (7-14). (7-10) — это сокращенная форма записи, в которой я говорю, что result должен быть передан в качестве параметра в функцию. И сама эта функция (7-10) является параметром и содержит тело следующего вида, которое я здесь написал. Ничего страшного не будет, если вы будете использовать у себя в работе предыдущий пример, если он вам покажется проще. Если вы легко переключитесь на новый вариант или уже готовы с функциональными подходами, то используйте вот этот вариант.