Глава 12

JDBC

Пространство — иллюзия,
дисковое пространство — тем более.

Драйверы, соединения и запросы

API JDBC (Java DataBase Connectivity) — стандартный прикладной интерфейс языка Java для организации взаимодействия между приложением и СУБД. Взаимодействие осуществляется с помощью драйверов JDBC, обеспечивающих реализацию общих интерфейсов для конкретных СУБД и конкретных протоколов. В JDBC определяются четыре типа драйверов:

1. Драйвер, использующий другой прикладной интерфейс взаимодействия с СУБД, в частности, ODBC (так называемый JDBC-ODBC — мост). Стандартный драйвер первого типа **sun.jdbc.odbc.JdbcOdbcDriver** входит в JDK.
2. Драйвер, работающий через внешние native библиотеки клиента СУБД.
3. Драйвер, работающий по сетевому и независимому от СУБД протоколу с промежуточным Java-сервером, который, в свою очередь, подключается к нужной СУБД.
4. Сетевой драйвер, работающий напрямую с нужной СУБД и не требующий установки native-библиотек.

Предпочтение естественным образом отдается второму типу, однако если приложение выполняется на машине, на которой не предполагается установка клиента СУБД, то выбор производится между третьим и четвертым типами. Причем четвертый тип работает напрямую с СУБД по ее протоколу, поэтому можно предположить, что драйвер четвертого типа будет более эффективным по сравнению с третьим типом с точки зрения производительности. Первый же тип, как правило, используется редко, т. е. в тех случаях, когда у СУБД нет своего драйвера JDBC, но присутствует драйвер ODBC.

JDBC предоставляет интерфейс для разработчиков, использующих различные СУБД. С помощью JDBC отсылаются SQL-запросы только к реляционным базам данных (БД), для которых существуют драйверы, знающие способ общения с реальным сервером базы данных.

Последовательность действий для выполнения первого запроса.

1. *Подключение библиотеки с классом-драйвером базы данных.*

Дополнительно требуется подключить к проекту библиотеку, содержащую драйвер, поместив ее предварительно в папку **/lib** приложения.

mysql-connector-java-[*номер версии***]-bin.jar** для СУБД MySQL,
ojdbc[*номер версии*].**jar** для СУБД Oracle.

2. *Установка соединения с БД.*

Для установки соединения с БД вызывается статический метод **getConnection()** класса **java.sql.DriverManager**. В качестве параметров методу передаются URL базы данных, логин пользователя БД и пароль доступа. Загрузка класса драйвера базы данных при отсутствии ссылки на экземпляр этого класса в JDBC 4.1 происходит автоматически при установке соединения экземпляром **DriverManager**. Метод возвращает объект **Connection**. URL базы данных, состоящий из типа и адреса физического расположения БД, может создаваться в виде отдельной строки или извлекаться из файла ресурсов. Соответственно:

```
Connection cn = DriverManager.getConnection("jdbc:mysql://localhost:3306/testphones",
                                                                    "root", "pass");
Connection cn = DriverManager.getConnection("jdbc:oracle:thin:@//localhost:1521:testphones",
                                                                    "system", "pass");
```

В результате будет возвращен объект **Connection** и будет одно установленное соединение с БД с именем **testphones**. Класс **DriverManager** предоставляет средства для управления набором драйверов баз данных. С помощью метода **getDrivers()** можно получить список всех доступных драйверов.

До появления JDBC 4.0 объект драйвера СУБД нужно было создавать явно с помощью вызова соответственно:

```
Class.forName("com.mysql.jdbc.Driver");
Class.forName("oracle.jdbc.OracleDriver");
```

или зарегистрировать драйвер

```
DriverManager.registerDriver(new com.mysql.jdbc.Driver());
DriverManager.registerDriver(new oracle.jdbc.OracleDriver());
```

В большинстве случаев в этом нет необходимости, так как экземпляр драйвера загружается автоматически при попытке получения соединения с БД.

3. *Создание объекта для передачи запросов.*

После создания объекта **Connection** и установки соединения можно начинать работу с БД с помощью операторов SQL. Для выполнения запросов применяется объект **Statement**, создаваемый вызовом метода **createStatement()** класса **Connection**.

```
Statement st = cn.createStatement();
```

Объект класса **Statement** используется для выполнения SQL-запроса без его предварительной подготовки. Могут применяться также объекты классов **PreparedStatement** и **CallableStatement** для выполнения подготовленных запросов и хранимых процедур.

4. *Выполнение запроса.*

Созданные объекты можно использовать для выполнения запроса SQL, передавая его в один из методов **execute(String sql)**, **executeBatch()**, **executeQuery(String sql)** или **executeUpdate(String sql)**. Результаты выполнения запроса помещаются в объект **ResultSet**:

```
/* выборка всех данных таблицы phonebook */
ResultSet rs = st.executeQuery("SELECT * FROM phonebook");
```

Для добавления, удаления или изменения информации в таблице запрос помещается в метод **executeUpdate()**.

5. *Обработка результатов выполнения запроса* производится методами интерфейса **ResultSet**, где самыми распространенными являются **next(), first(), previous(), last()** для навигации по строками таблицы результатов и группа методов по доступу к информации вида **getString(int pos)**, а также аналогичные методы, начинающиеся с **get***Тип***(int pos)** (**getInt(int pos)**, **getFloat(int pos)** и др.) и **update***Тип***()**. Среди них следует выделить методы **getClob(int pos)** и **getBlob(int pos)**, позволяющие извлекать из полей таблицы специфические объекты (Character Large Object, Binary Large Object), которые могут быть, например, графическими или архивными файлами. Эффективным способом извлечения значения поля из таблицы ответа является обращение к этому полю по его имени в строке результатов методами типа **int getInt(String columnLabel)**, **String getString(String columnLabel)**, **Object getObject(String columnLabel)** и подобными им. Интерфейс располагает большим числом методов по доступу к таблице результатов, поэтому рекомендуется изучить его достаточно тщательно.

При первом вызове метода **next()** указатель перемещается на таблицу результатов выборки в позицию первой строки таблицы ответа. Когда строки закончатся, метод возвратит значение **false**.

6. *Закрытие соединения, statement*

```
st.close(); // закрывает также и ResultSet
cn.close();
```

После того, как база больше не нужна, соединение закрывается. Для того, чтобы правильно пользоваться приведенными методами, программисту требуется знать типы полей БД. В распределенных системах это знание предполагается изначально. В Java 7 для объектов-ресурсов, требующих закрытия, реализована технология try with resources.

СУБД MySQL

СУБД MySQL совместима с JDBC и будет применяться для создания учебных баз данных. Версия СУБД может быть загружена с сайта **www.mysql.com**. Для корректной установки необходимо следовать инструкциям мастера. В процессе установки следует создать администратора СУБД с именем **root** и паролем, например, **pass**. Если планируется разворачивать реально работающее приложение, необходимо исключить тривиальных пользователей сервера БД (иначе злоумышленники могут получить полный доступ к БД). Для запуска следует использовать команду из папки **/mysql/bin:**

```
mysqld-nt -standalone
```

Если не появится сообщение об ошибке, то СУБД MySQL запущена. Для создания базы данных и ее таблиц используются команды языка SQL.

Простое соединение и простой запрос

Теперь следует воспользоваться всеми предыдущими инструкциями и создать пользовательскую БД с именем **testphones** и одной таблицей **PHONEBOOK**. Таблица должна содержать три поля: числовое (первичный ключ) — **IDPHONEBOOK**, символьное — **LASTNAME** и числовое — **PHONE** и несколько занесенных записей.

IDPHONEBOOK	LASTNAME	PHONE
1	Каптур	7756544
2	Artukevich	6861880

При создании таблицы следует задавать кодировку UTF-8, поддерживающую хранение символов кириллицы.

Приложение, осуществляющее простейший запрос на выбор всей информации из таблицы, выглядит следующим образом.

/ # 1 # простое соединение с БД и простой запрос # SimpleJDBCRunner.java */*

```
package by.bsu.data.main;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.Properties;
import by.bsu.data.subject.Abonent;
```

```
public class SimpleJDBCRunner {
          public static void main(String[ ] args) {
                    String url = "jdbc:mysql://localhost:3306/testphones";
                    Properties prop = new Properties();
                    prop.put("user", "root");
                    prop.put("password", "pass");
                    prop.put("autoReconnect", "true");
                    prop.put("characterEncoding", "UTF-8");
                    prop.put("useUnicode", "true");
                    Connection cn = null;
                    DriverManager.registerDriver(new com.mysql.jdbc.Driver());
                    try { // 1 блок
                              cn = DriverManager.getConnection(url, prop);
                              Statement st = null;
                              try { // 2 блок
                                        st = cn.createStatement();
                                        ResultSet rs = null;
                                        try { // 3 блок
                                            rs = st.executeQuery("SELECT * FROM phonebook");
                                            ArrayList<Abonent> lst = new ArrayList<>();
                                                  while (rs.next()) {
                                                            int id = rs.getInt(1);
                                                            int phone = rs.getInt(3);
                                                            String name = rs.getString(2);
                                                            lst.add(new Abonent(id, phone, name));
                                                  }
                                                  if (lst.size() > 0) {
                                                            System.out.println(lst);
                                                  } else {
                                                            System.out.println("Not found");
                                                  }
                                        } finally { // для 3-го блока try
                                                  /*
                                                   * закрыть ResultSet, если он был открыт
                                                   * или ошибка произошла во время
                                                   * чтения из него данных
                                                   */
                                                  if (rs != null) { // был ли создан ResultSet
                                                            rs.close();
                                                  } else {
                                                            System.err.println(
                                                            "ошибка во время чтения из БД");
                                                  }
                                        }
                              } finally {
                                        /*
                                         * закрыть Statement, если он был открыт или ошибка
                                         * произошла во время создания Statement
                                         */
                                        if (st != null) { // для 2-го блока try
```

```
                                        st.close();
                                } else {
                                        System.err.println("Statement не создан");
                                }
                        }
                } catch (SQLException e) { // для 1-го блока try
                        System.err.println("DB connection error: " + e);
                                        /*
                                        * вывод сообщения о всех SQLException
                                        */
                } finally {
                        /*
                         * закрыть Connection, если он был открыт
                        */
                        if (cn != null) {
                                try {
                                      cn.close();
                                } catch (SQLException e) {
                                      System.err.println("Connection close error: " + e);
                                }
                        }
                }
        }
}
```

В несложном приложении достаточно контролировать закрытие соединения, так как незакрытое или «провисшее» соединение снижает быстродействие системы.

Класс **Abonent**, используемый приложением для хранения информации, извлеченной из БД, выглядит очень просто:

/ # 2 # класс с информацией # Entity.java # Abonent.java */*

```
package by.bsu.data.subject;
import java.io.Serializable;
public abstract class Entity implements Serializable, Cloneable {
          private int id;
          public Entity() {
          }
          public Entity(int id) {
                    this.id = id;
          }
          public int getId() {
                    return id;
          }
          public void setId(int id) {
                    this.id = id;
          }
}
```

```
package by.bsu.data.subject;
public class Abonent extends Entity {
          private int phone;
          private String lastname;
          public Abonent() {
          }
          public Abonent(int id, int phone, String lastname) {
                    super(id);
                    this.phone = phone;
                    this.lastname = lastname;
          }
          public int getPhone() {
                    return phone;
          }
          public void setPhone(int phone) {
                    this.phone = phone;
          }
          public String getLastname() {
                    return lastname;
          }
          public void setLastname(String lastname) {
                    this.lastname = lastname;
          }
          @Override
          public String toString() {
                    return "Abonent [id=" + id + ", phone=" + phone +
                    ", lastname=" + lastname + "]";
          }
}
```

Параметры соединения можно задавать несколькими способами: с помощью прямой передачи значений в коде класса, а также с помощью файлов **properties** или **xml**. Окончательный выбор производится в зависимости от конфигурации проекта.

Чтение параметров соединения с базой данных и получение соединения следует вынести в отдельный класс. Класс **ConnectorDB** использует файл ресурсов **database.properties**, в котором хранятся, как правило, параметры подключения к БД, такие, как логин и пароль доступа. Например:

```
db.driver   = com.mysql.jdbc.Driver
db.user     = root
db.password = pass
db.poolsize = 32
db.url  = jdbc:mysql://localhost:3306/testphones
db.useUnicode = true
db.encoding = UTF-8
```

Код класса **ConnectorDB** выглядит следующим образом:

/ # 3 # установка соединения с БД # ConnectorDB.java */*

```
package by.bsu.data.connect;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.ResourceBundle;
public class ConnectorDB {
          public static Connection getConnection() throws SQLException {
                    ResourceBundle resource = ResourceBundle.getBundle("database");
                    String url = resource.getString("db.url");
                    String user = resource.getString("db.user");
                    String pass = resource.getString("db.password");
                    return DriverManager.getConnection(url, user, pass);
          }
}
```

В таком случае получение соединения с БД сведется к вызову

```
Connection cn = ConnectorDB.getConnection();
```

Метаданные

Существует целый ряд методов интерфейсов **ResultSetMetaData** и **DatabaseMetaData** для интроспекции объектов. С помощью этих методов можно получить список таблиц, определить типы, свойства и количество столбцов БД. Для строк подобных методов нет.

Получить объект **ResultSetMetaData** можно следующим образом:

```
ResultSetMetaData rsMetaData = rs.getMetaData();
```

Некоторые методы интерфейса **ResultSetMetaData**:

int getColumnCount() — возвращает число столбцов набора результатов объекта **ResultSet**;

String getColumnName(int column) — возвращает имя указанного столбца объекта **ResultSet**;

int getColumnType(int column) — возвращает тип данных указанного столбца объекта **ResultSet** и т. д.

Получить объект **DatabaseMetaData** можно следующим образом:

```
DatabaseMetaData dbMetaData = cn.getMetaData();
```

Некоторые методы весьма обширного интерфейса **DatabaseMetaData**:

String getDatabaseProductName() — возвращает название СУБД;

String getDatabaseProductVersion() — возвращает номер версии СУБД;

String getDriverName() — возвращает имя драйвера JDBC;

String getUserName() — возвращает имя пользователя БД;

String getURL() — возвращает местонахождение источника данных;

ResultSet getTables() — возвращает набор типов таблиц, доступных для данной БД, и т. д.

Подготовленные запросы и хранимые процедуры

Для представления запросов существует еще два типа объектов **PreparedStatement** и **CallableStatement**. Объекты первого типа используются при выполнении часто повторяющихся запросов SQL. Такой оператор предварительно готовится и хранится в объекте, что ускоряет обмен информацией с базой данных при многократном выполнении однотипных запросов. Второй интерфейс используется для выполнения хранимых процедур, созданных средствами самой СУБД.

При использовании **PreparedStatement** невозможен sql injection attacks. То есть если существует возможность передачи в запрос информации в виде строки, то следует использовать для выполнения такого запроса объект **PreparedStatement**.

Для подготовки SQL-запроса, в котором отсутствуют конкретные параметры, используется метод **prepareStatement(String sql)** интерфейса **Connection**, возвращающий объект **PreparedStatement**.

```
String sql = "INSERT INTO phonebook(idphonebook, lastname, phone) VALUES(?, ?, ?)";
PreparedStatement ps = cn.prepareStatement(sql);
```

Установка входных значений конкретных параметров этого объекта производится с помощью методов **setString(int index, String x)**, **setInt(int index, int x)** и подобных им, после чего и осуществляется непосредственное выполнение запроса методами **int executeUpdate()**, **ResultSet executeQuery()**.

/ # 4 # подготовка запроса на добавление информации # DataBaseHelper.java */*

```
package by.bsu.data.connect;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import by.bsu.data.subject.Abonent;
public class DataBaseHelper {
          private final static String SQL_INSERT =
                         "INSERT INTO phonebook(idphonebook, lastname, phone ) VALUES(?,?,?)";
          private Connection connect;
                    public DataBaseHelper() throws SQLException {
                    connect = ConnectorDB.getConnection();
          }
          public PreparedStatement getPreparedStatement(){
                    PreparedStatement ps = null;
                    try {
                              ps = connect.prepareStatement(SQL_INSERT);
                    } catch (SQLException e) {
```

```
                    e.printStackTrace();
                }
                return ps;
        }
        public boolean insertAbonent(PreparedStatement ps, Abonent ab) {
                boolean flag = false;
                try {
                    ps.setInt(1, ab.getId());
                    ps.setString(2, ab.getName());
                    ps.setInt(3, ab.getPhone());
                    ps.executeUpdate();
                    flag = true;
                } catch (SQLException e) {
                    e.printStackTrace();
                }
                return flag;
        }
        public void closeStatement(PreparedStatement ps) {
                if (ps != null) {
                    try {
                        ps.close();
                    } catch (SQLException e) {
                        e.printStackTrace();
                    }
                }
        }
}
```

Так как данный оператор предварительно подготовлен, то он выполняется быстрее обычных операторов, ему соответствующих. Оценить преимущества во времени можно, выполнив большое число повторяемых запросов с предварительной подготовкой запроса и без нее.

/ # 5 # добавление нескольких записей в БД # PreparedJDBCRunner.java */*

```
package by.bsu.data.main;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.util.ArrayList;
import by.bsu.data.connect.DataBaseHelper;
import by.bsu.data.subject.Abonent;
public class PreparedJDBCRunner {
        public static void main(String[] args) {
                ArrayList<Abonent> list = new ArrayList<Abonent>() {
                    {
                       add(new Abonent(87, 1658468, "Кожух Дмитрий"));
                       add(new Abonent(51, 8866711, "Буйкевич Александр"));
                    }
                };
                DataBaseHelper helper = null;
```

```
                    PreparedStatement statement = null;
                    try {
                              helper = new DataBaseHelper();
                              statement = helper.getPreparedStatement();
                              for (Abonent abonent : list) {
                                        helper.insertAbonent(statement, abonent);
                              }
                    } catch (SQLException e) {
                              e.printStackTrace();
                    } finally {
                              helper.closeStatement(statement);
                    }
          }
}
```

Интерфейс **CallableStatement** расширяет возможности интерфейса **PreparedStatement** и обеспечивает выполнение хранимых процедур.

Хранимая процедура — это, в общем случае, именованная последовательность команд SQL, рассматриваемых как единое целое и выполняющаяся в адресном пространстве процессов СУБД, который можно вызвать извне (в зависимости от политики доступа используемой СУБД). В данном случае хранимая процедура будет рассматриваться в более узком смысле как последовательность команд SQL, хранимых в БД и доступных любому пользователю этой СУБД. Механизм создания и настройки хранимых процедур зависит от конкретной базы данных. Для создания объекта **CallableStatement** вызывается метод **prepareCall()** объекта **Connection**.

Интерфейс **CallableStatement** позволяет исполнять хранимые процедуры, которые находятся непосредственно в БД. Одна из особенностей этого процесса в том, что **CallableStatement** способен обрабатывать не только входные (**IN**) параметры, но и выходящие (**OUT**) и смешанные (**INOUT**) параметры. Тип выходного параметра должен быть зарегистрирован с помощью метода **registerOutParameter()**. После установки входных и выходных параметров вызываются методы **execute()**, **executeQuery()** или **executeUpdate()**.

Пусть в БД существует хранимая процедура **getlastname**, которая по уникальному номеру телефона для каждой записи в таблице **phonebook** будет возвращать соответствующее ему имя:

```
CREATE PROCEDURE getlastname (p_phone IN INT, p_lastname OUT VARCHAR) AS
BEGIN
SELECT lastname INTO p_lastname FROM phonebook WHERE phone = p_phone;
END
```

Тогда для получения имени через вызов данной процедуры необходимо исполнить java-код вида:

```
final String SQL = "{call getlastname (?, ?)}";
CallableStatement cs = cn.prepareCall(SQL);
```

```
// передача значения входного параметра
cs.setInt(1, 1658468);
// регистрация возвращаемого параметра
cs.registerOutParameter(2, java.sql.Types.VARCHAR);
cs.execute();
String lastName = cs.getString(2);
```

В JDBC также существует механизм batch-команд, который позволяет запускать на исполнение в БД массив запросов SQL вместе, как одну единицу.

```
// turn off autocommit
cn.setAutoCommit(false);
Statement st = con.createStatement();
st.addBatch("INSERT INTO phonebook VALUES (55, 5642032, 'Гончаров')");
st.addBatch("INSERT INTO location VALUES (260, 'Minsk')");
st.addBatch("INSERT INTO student_department VALUES (1000, 260)");
// submit a batch of update commands for execution
int[ ] updateCounts = stmt.executeBatch();
```

Если используется объект **PreparedStatement**, batch-команда состоит из параметризованного SQL-запроса и ассоциируемого с ним множества параметров.

Метод **executeBatch()** интерфейса **PreparedStatement** возвращает массив чисел, причем каждое характеризует число строк, которые были изменены конкретным запросом из batch-команды.

Пусть существует список объектов типа **Abonent** со стандартным набором методов **get***Tun***()/set***Tun***()** для каждого из его полей, и необходимо внести их значения в БД. Многократное выполнение методов **execute()** или **executeUpdate()** становится неэффективным, и в данном случае лучше использовать схему batch-команд:

```
try {
   ArrayList<Abonent> abonents = new ArrayList<>(); // заполнение списка
   PreparedStatement statement = con.prepareStatement("INSERT INTO phonebook VALUES(?,?,?)");
   for (Abonent abonent : abonents) {
              statement.setInt(0,  abonent.getId());
              statement.setInt(1,  abonent.getPhone());
              statement.setString(2,  abonent.getLastname());
              statement.addBatch();
   }
updateCounts = statement.executeBatch();
} catch (BatchUpdateException e) {
          e.printStackTrace();
}
```

Транзакции

При проектировании распределенных систем часто возникают ситуации, когда сбой в системе или какой-либо ее периферийной части может привести к потере информации или к финансовым потерям. Простейшим примером может служить пример с перечислением денег с одного счета на другой. Если сбой произошел в тот момент, когда операция снятия денег с одного счета уже произведена, а операция зачисления на другой счет еще не произведена, то система, допускающая такие ситуации, должна быть признана не отвечающей требованиям заказчика. Или должны выполняться обе операции, или не выполняться вовсе. Такие две операции трактуют как одну и называют транзакцией.

Транзакция, или деловая операция, определяется как единица работы, обладающая свойствами ACID:

- Атомарность — две или более операций выполняются все или не выполняется ни одна. Успешно завершенные транзакции фиксируются, в случае неудачного завершения происходит откат всей транзакции.
- Согласованность — при возникновении сбоя система возвращается в состояние до начала неудавшейся транзакции. Если транзакция завершается успешно, то проверка согласованности удостоверяется в успешном завершении всех операций транзакции.
- Изолированность — во время выполнения транзакции все объекты-сущности, участвующие в ней, должны быть синхронизированы.
- Долговечность — все изменения, произведенные с данными во время транзакции, сохраняются, например, в базе данных. Это позволяет восстанавливать систему.

Для фиксации результатов работы SQL-операторов, логически выполняемых в рамках некоторой транзакции, используется SQL-оператор COMMIT. В API JDBC эта операция выполняется по умолчанию после каждого вызова методов **executeQuery()** и **executeUpdate()**. Если же необходимо сгруппировать запросы и только после этого выполнить операцию COMMIT, сначала вызывается метод **setAutoCommit(boolean param)** интерфейса **Connection** с параметром **false**, в результате выполнения которого текущее соединение с БД переходит в режим неавтоматического подтверждения операций. После этого выполнение любого запроса на изменение информации в таблицах базы данных не приведет к необратимым последствиям, пока операция COMMIT не будет выполнена непосредственно. Подтверждает выполнение SQL-запросов метод **commit()** интерфейса **Connection**, в результате действия которого все изменения таблицы производятся как одно логическое действие. Если же транзакция не выполнена, то методом **rollback()** отменяются действия всех запросов SQL, начиная от последнего вызова **commit()**. В следующем примере информация добавляется в таблицу в режиме действия транзакции, подтвердить

или отменить действия которой можно, снимая или добавляя комментарий в строках вызова методов **commit()** и **rollback()**.

/ # 6 # транзакция по переводу денег со счета на счет # SingletonEngine.java */*

```
package com.epam.logic;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
public class SingletonEngine {
   private Connection connectionTo;
   private Connection connectionFrom;
   private static SingletonEngine instance = null;
   public synchronized static SingletonEngine getInstance() {
             if (instance == null) {
                       instance = new SingletonEngine();
                       instance.getConnectionTo();
                       instance.getConnectionFrom();
             }
             return instance;
   }
   private Connection getConnectionFrom() {
             try {
                       Class.forName("com.mysql.jdbc.Driver");
                       connectionFrom = DriverManager.getConnection(
                                 "jdbc:mysql://localhost:3306/testFrom", "root", "pass");
                       connectionFrom.setAutoCommit(false);
             } catch (SQLException e) {
                       System.err.println("SQLException: " + e.getMessage()
                       + "SQLState: " + e.getSQLState());
             } catch (ClassNotFoundException ex) {
                       System.out.println("Driver not found");
             }
             return connectionFrom;
   }
   private Connection getConnectionTo() {
             final String connectToAdress = "jdbc:mysql://10.162.4.151:3306/testTo";
             try {
                       Class.forName("com.mysql.jdbc.Driver");
                       connectionTo  =  DriverManager.getConnection( connectToAdress, "root", "pass");
                       connectionTo.setAutoCommit(false);
             } catch (SQLException e) {
                       System.err.println("SQLException: " + e.getMessage()
                       + "SQLState: " + e.getSQLState());
             } catch (ClassNotFoundException e) {
                       System.err.println("Driver not found");
```

```
            }
            return connectionTo;
    }
    public void transfer(String summa) throws SQLException {
            Statement stFrom = null;
            Statement stTo = null;
            try {
                    int sum = Integer.parseInt(summa);
                    if (sum <= 0) {
                            throw new NumberFormatException("less or equals zero");
                    }
                    stFrom = connectionFrom.createStatement();
                    stTo = connectionTo.createStatement();
                    // транзакция из 4-х запросов
                    ResultSet rsFrom =
                            stFrom.executeQuery("SELECT balance from table_from");
                    ResultSet rsTo =
                            stTo.executeQuery("SELECT balance from table_to");
                            int accountFrom = 0;
                    while (rsFrom.next()) {
                            accountFrom = rsFrom.getInt(1);
                    }
                    int resultFrom= 0;
                    if (accountFrom >= sum) {
                            resultFrom = accountFrom - sum;
                    } else {
                            throw new SQLException("Invalid balance");
                    }
                    int accountTo = 0;
                    while (rsTo.next()) {
                            accountTo = rsTo.getInt(1);
                    }
                    int resultTo = accountTo + sum;
                    stFrom.executeUpdate(
                            "UPDATE table_from SET balance=" + resultFrom);
                    stTo.executeUpdate("UPDATE table_to SET balance=" + resultTo);
                    // завершение транзакции
                    connectionFrom.commit();
                    connectionTo.commit();
                    System.out.println("remaining on :" + resultFrom + " rub");
            } catch (SQLException e) {
                    System.err.println("SQLState: " + e.getSQLState()
                    + "Error Message: " + e.getMessage());
                    // откат транзакции при ошибке
                    connectionFrom.rollback();
                    connectionTo.rollback();
            } catch (NumberFormatException e) {
                    System.err.println("Invalid summa: " + summa);
            } finally {
                    if (stFrom != null) {
```

```
                                        try {
                                                stFrom.close();
                                        } catch (SQLException ex) {
                                                ex.printStackTrace();
                                        }
                                }
                                if (stTo != null) {
                                        try {
                                                stTo.close();
                                        } catch (SQLException ex) {
                                                ex.printStackTrace();
                                        }
                                }
                        }
                }
}
```

Для транзакций существует несколько типов чтения:

- грязное чтение (dirty reads) происходит, когда транзакциям разрешено видеть несохраненные изменения данных. Иными словами, изменения, сделанные в одной транзакции, видны вне ее до того, как она была сохранена. Если изменения не будут сохранены, то, вероятно, другие транзакции выполняли работу на основе некорректных данных;
- неповторяющееся чтение (nonrepeatable reads) происходит, когда транзакция А читает строку, транзакция Б изменяет эту строку, транзакция А читает ту же строку и получает обновленные данные;
- фантомное чтение (phantom reads) происходит, когда транзакция А считывает все строки, удовлетворяющие **WHERE**-условию, транзакция Б вставляет новую или удаляет одну из строк, которая удовлетворяет этому условию, транзакция А еще раз считывает все строки, удовлетворяющие **WHERE**-условию, уже вместе с новой строкой или недосчитавшись старой.

JDBC удовлетворяет уровням изоляции транзакций, определенным в стандарте SQL:2003.

Уровни изоляции транзакций определены в виде констант интерфейса **Connection** (по возрастанию уровня ограничения):

- **TRANSACTION_NONE** — информирует о том, что драйвер не поддерживает транзакции;
- **TRANSACTION_READ_UNCOMMITTED** — позволяет транзакциям видеть несохраненные изменения данных, что разрешает грязное, неповторяющееся и фантомное чтения;
- **TRANSACTION_READ_COMMITTED** — означает, что любое изменение, сделанное в транзакции, не видно вне ее, пока она не сохранена. Это предотвращает грязное чтение, но разрешает неповторяющееся и фантомное;
- **TRANSACTION_REPEATABLE_READ** — запрещает грязное и неповторяющееся чтение, но фантомное разрешено;

- **TRANSACTION_SERIALIZABLE** — определяет, что грязное, неповторяющееся и фантомное чтения запрещены.

Метод **boolean supportsTransactionIsolationLevel(int level)** интерфейса **DatabaseMetaData** определяет, поддерживается ли заданный уровень изоляции транзакций.

В свою очередь, методы интерфейса **Connection** определяют доступ к уровню изоляции:

int getTransactionIsolation() — возвращает текущий уровень изоляции;

void setTransactionIsolation(int level) — устанавливает необходимый уровень.

Точки сохранения

Точки сохранения, представляемые классом **java.sql.Savepoint**, дают дополнительный контроль над транзакциями, привязывая изменения СУБД к конкретной точке в области транзакции. Установкой точки сохранения обозначается логическая точка внутри транзакции, которая может быть использована для отката данных. Таким образом, если произойдет ошибка, можно вызвать метод **rollback(Savepoint point)** для отмены всех изменений, которые были сделаны после точки сохранения. Метод **boolean supportsSavepoints()** интерфейса **DatabaseMetaData** используется для того, чтобы определить, поддерживает ли точки сохранения драйвер JDBC и сама СУБД. Методы **setSavepoint(String name)** и **setSavepoint()** возвращают объект **Savepoint** интерфейса **Connection** и используются для установки именованой или неименованой точки сохранения во время текущей транзакции. При этом новая транзакция будет начата, если в момент вызова **setSavepoint()** не будет активной транзакции.

Data Access Object

При создании информационной системы выявляются некоторые слои, которые отвечают за взаимодействие различных частей приложения. Связь с базой данных является важной частью любой системы, поэтому всегда выделяется часть кода, ответственная за передачу запросов в БД и обработку полученных от нее ответов. Общее определение шаблона Data Access Object трактует его как прослойку между приложением и СУБД. DAO абстрагирует бизнес-сущности системы и отражает их на записи в БД. DAO определяет общие способы использования соединения с БД, моменты его открытия и закрытия или извлечения и возвращения в пул. В общем случае DAO можно определять таким образом, чтобы была возможность подмены одной модели базы

данных другой. Например: реляционную заменить на объектную или, что проще, MySQL на Oracle. В практическом программировании такие глобальные задачи ставятся крайне редко, поэтому будет приведено несколько способов организации взаимодействия с БД, отличающихся уровнем использования коннекта к БД и организацией работы с бизнес-сущностями.

Вершина иерархии DAO представляет собой класс или интерфейс с описанием общих методов, которые будут использоваться при взаимодействии с таблицей или группой таблиц. Как правило, это методы выбора, поиска сущности по признаку, добавление, удаление и замена информации.

/ # 7 # общие методы взаимодействия с моделью данных # AbstractDAO.java */*

```
package by.bsu.simpledao;
import java.sql.Connection;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.List;
import by.bsu.subject.Entity;
public abstract class AbstractDAO <K, T extends Entity> {
          public abstract List<T> findAll();
          public abstract T findEntityById(K id);
          public abstract boolean delete(K id);
          public abstract boolean delete(T entity);
          public abstract boolean create(T entity);
          public abstract T update(T entity);
}
```

Набор методов может варьироваться в зависимости от логики приложения. Параметр **K** описывает, как правило, ключ в таблице, так как редкая таблица, содержащая описание сущности, обходится без первичного колюча. Параметр **Entity** определяет общую бизнес-сущность, от которой наследуются все бизнес-сущности системы. Класс может содержать также методы возвращения соединения в пул или его закрытия, а также закрытия экземпляра **Statement** в виде:

```
public void close(Statement st) {
                    try {
                              if (st != null) {
                                        st.close();
                              }
                    } catch (SQLException e) {
                              // лог о невозможности закрытия Statement
                    }
          }
public void close(Connection connection) {
                    try {
                              if (connection != null) {
                                        connection.close();
```

```
                    }
                } catch (SQLException e) {
                    // генерация исключения, т.к. нарушается работа пула
                }
        }
```

DAO. Уровень метода

Реализация DAO для конкретного бизнес-объекта имеет шанс выглядеть следующим образом. Часть методов может остаться нереализованной, кроме того, могут добавляться собственные методы, определить которые в более общем классе невозможно из-за узкой области применения. В данном случае это метод **Abonent findAbonentByLastName(String name)**.

Реализация на уровне метода предполагает использование соединения для выполнения единственного запроса, т. е. соединение будет получено из пула в начале работы метода и возвращено по его окончании, что в общем случае не является экономным решением.

/ # 8 # конкретная реализация взаимодействия с моделью данных # AbonentDAO.java */*

```
package by.bsu.simpledao;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.*;
import by.bsu.pool.ConnectionPool;
import by.bsu.subject.Abonent;
public class AbonentDAO extends AbstractDAO <Integer, Abonent> {
          public static final String SQL_SELECT_ALL_ABONENTS = "SELECT * FROM phonebook";
          public static final String SQL_SELECT_ABONENT_BY_LASTNAME =
                    "SELECT idphonebook,phone FROM phonebook WHERE lastname=?";
          @Override
          public List<Abonent> findAll() {
                    List<Abonent> abonents = new ArrayList<>();
                    Connection cn = null;
                    Statement st = null;
                    try {
                              cn = ConnectionPool.getConnection();
                              st = cn.createStatement();
                              ResultSet resultSet =
                                        st.executeQuery(SQL_SELECT_ALL_ABONENTS);

                              while (resultSet.next()) {
                                        Abonent abonent = new Abonent();
                                        abonent.setId(resultSet.getInt("idphonebook"));
```

```
                    abonent.setPhone(resultSet.getInt("phone"));
                    abonent.setLastName(resultSet.getString("lastname"));
                    abonents.add(abonent);
                }
        } catch (SQLException e) {
                System.err.println("SQL exception (request or table failed): " + e);
        } finally {
                close(st);
                // код возвращения экземпляра Connection в пул
        }
        return abonents;
}
@Override
public Abonent findEntityById(Integer id) {
        throw new UnsupportedOperationException();
}
@Override
public boolean delete(Integer id) {
        throw new UnsupportedOperationException();
}
@Override
public Abonent create(Abonent entity) {
        throw new UnsupportedOperationException();
}
@Override
public Abonent update(Abonent entity) {
        throw new UnsupportedOperationException();
}
// собственный метод DAO
public Abonent findAbonentByLastName(String name) {
        Abonent abonent = new Abonent();
        Connection cn = null;
        PreparedStatement st = null;
        try {
                cn = ConnectionPool.getConnection();
                st =
                cn.prepareStatement(SQL_SELECT_ABONENT_BY_LASTNAME);
                st.setString(1, name);
                ResultSet resultSet =st.executeQuery();
                resultSet.next();
                        abonent.setId(resultSet.getInt("idphonebook"));
                        abonent.setPhone(resultSet.getInt("phone"));
                        abonent.setLastName(name);
        } catch (SQLException e) {
                System.err.println("SQL exception (request or table failed): " + e);
        } finally {
                close(st);
                // код возвращения экземпляра Connection в пул
        }
        return abonent;
```

```
        }
        @Override
        public boolean delete(Abonent entity) {
                throw new UnsupportedOperationException();
        }
}
```

SQL-запросы размещаются в статических константах класса. В редких случаях запросы могут храниться вне системы в **xml** или **properties** файлах.

В данном примере использовался пул соединений, конфигурация которого задается в файле конфигурации **context.xml**.

/ # 9 # стандартный пул соединений # ConnectionPool.java */*

```
package by.bsu.pool;
import java.sql.Connection;
import java.sql.SQLException;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.sql.DataSource;
public class ConnectionPool {
    private static final String DATASOURCE_NAME = "jdbc/testphones";
    private static DataSource dataSource;
    static {
        try {
            Context initContext = new InitialContext();
            Context envContext = (Context) initContext.lookup("java:/comp/env");
            dataSource = (DataSource) envContext.lookup(DATASOURCE_NAME);
        } catch (NamingException e) {
            e.printStackTrace();
        }
    }
    private ConnectionPool() { }
    public static Connection getConnection() throws SQLException {
        Connection connection = dataSource.getConnection();
        return connection;
    }
// метод возвращения Connection в пул
}
```

Особенности конфигурирования и использования стандартного пула соединений приведены в конце главы 16.

DAO. Уровень класса

Реализация на уровне класса подразумевает использование одного коннекта к базе данных для вызова нескольких методов конкретного DAO класса. В этом

случае вершина иерархии DAO в качестве поля может объявлять сам коннект к СУБД или его оболочку, кроме стандартного набора методов, например:

/ # 10 # DAO с полем Connection # AbstractDAO.java */*

```
package by.bsu.simpledao;
import java.sql.Statement;
import by.bsu.action.WrapperConnector;
public abstract class AbstractDAO {
          protected WrapperConnector connector;
          // методы добавления, поиска, замены, удаления
          // методы закрытия коннекта и Statement
          public void close() {
                    connector.closeConnection();
          }
          protected void closeStatement(Statement statement) {
                    connector.closeStatement(statement);
          }
}
```

Реализация конкретного DAO при таком построении взаимодействия никогда в методе не должна закрывать соединение. Соединение закрывается в той части бизнес-логики, откуда выполняются обращения к DAO.

/ # 11 # DAO уровне класса # AbonentDAO.java */*

```
package by.bsu.simpledao;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.*;
import by.bsu.action.WrapperConnector;
import by.bsu.subject.Abonent;
public class AbonentDAO extends AbstractDAO {
    public static final String SQL_SELECT_ALL_ABONENTS = "SELECT * FROM phonebook";
    public AbonentDAO() {                          "SELECT * FROM phonebook";
              this.connector = new WrapperConnector();
    }
    public List<Abonent> findAll() {
              List<Abonent> abonents = new ArrayList<>();
              Statement st = null;
              try {
                        st  =  connector.getStatement();
                        ResultSet  resultSet = st.executeQuery(SQL_SELECT_ALL_ABONENTS);
                        while (resultSet.next()) {
                                  Abonent  abonent  = new  Abonent();
                                  abonent.setId(resultSet.getInt("idphonebook"));
                                  abonent.setPhone(resultSet.getInt("phone"));
```

```
                                    abonent.setLastName(resultSet.getString("lastname"));
                                    abonents.add(abonent);
                            }
        } catch  (SQLException e) {
                  System.err.println("SQL exception (request or table failed): " + e);
        }  finally {
                   this.closeStatement(st);
        }
        return abonents;
  }
  // другие методы
}
```

Соединение с базой данных инициирует конструктор DAO, либо получает его из пула. В методе остаются возможности по созданию экземпляра **Statement** для выполнения запросов и его закрытию. В данной реализации использовался класс-обертка для соединения, инкапсулирующий процесс создания соединения и упрощающий его использование. Такой подход при организации пула соединений из экземпляров классов-оберток резко затрудняет попадание в пул «диких» соединений, созданных программистом в обход пула.

/ # 12 # класс-оболочка соединения # WrapperConnector.java */*

```
package by.bsu.action;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;
import java.sql.SQLException;
import java.util.MissingResourceException;
import java.util.Properties;
import java.util.ResourceBundle;
public class WrapperConnector {
     private Connection connection;
     public WrapperConnector() {
               try {
                        ResourceBundle resource = ResourceBundle.getBundle("resource.database");
                        String url = resource.getString("url");
                        String user = resource.getString("user");
                        String pass = resource.getString("password");
                        Properties  prop  = new  Properties();
                        prop.put("user",  user);
                        prop.put("password",  pass);
                        connection  =  DriverManager.getConnection(url, prop);
               } catch  (MissingResourceException e) {
                          System.err.println("properties file is missing " + e);
               } catch  (SQLException e) {
                          System.err.println("not obtained connection " + e);
               }
```

```
        }
        public Statement getStatement() throws SQLException {
                if (connection != null) {
                        Statement statement = connection.createStatement();
                        if (statement != null) {
                                return statement;
                        }
                }
                throw new SQLException("connection or statement is null");
        }
        public void closeStatement(Statement statement) {
                if (statement != null) {
                        try {
                                statement.close();
                        } catch (SQLException e) {
                                System.err.println("statement is null " + e);
                        }
                }
        }
        public void closeConnection() {
                if (connection != null) {
                        try {
                                connection.close();
                        } catch (SQLException e) {
                                System.err.println(" wrong connection" + e);
                        }
                }
        }
        // другие необходимые делегированные методы интерфейса Connection
}
```

DAO. Уровень логики

На практике чаще всего возникает необходимость при выполнении запроса пользователя обращаться сразу к нескольким ветвям DAO и использовать при этом единственное соединение с БД. В этом случае соединение с БД создается или извлекается из пула до создания экземпляров DAO, а закрывается, соответственно, после выполнения всех обращений к БД.

/ # 13 # DAO с полем Connection без прямой инициализации # AbstractDAO.java */*

```
package by.bsu.simpledao;
import java.sql.Connection;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.List;
import by.bsu.subject.Entity;
```

```
public abstract class AbstractDAO <T extends Entity> {
          protected Connection connection;
          public AbstractDAO(Connection connection) {
                    this.connection = connection;
          }
          public abstract List<T> findAll();
          public abstract T findEntityById(int id);
          public abstract boolean delete(int id);
          public abstract boolean delete(T entity);
          public abstract boolean create(T entity);
          public abstract T update(T entity);
          public void close(Statement st) {
                    try {
                              if (st != null) {
                                        st.close();
                              }
                    } catch (SQLException e) {
                              // лог о невозможности закрытия Statement
                    }
          }
}
```

/ # 14 # примерные реализации DAO # AbonentDAO.java # PaymentDAO.java */*

```
package by.bsu.simpledao;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.*;
import by.bsu.subject.Abonent;
public class AbonentDAO extends AbstractDAO <Abonent> {
          public AbonentDAO(Connection connection) {
                    super(connection);
          }
          // реализации методов
}
package by.bsu.simpledao;
import java.sql.Connection;
import java.util.List;
import by.bsu.subject.Payment;
public class PaymentDAO extends AbstractDAO <Payment> {
          public PaymentDAO(Connection connection) {
                    super(connection);
          }
          // реализации методов
}
```

где **Payment** представляет класс-сущность в виде подкласса класса **Entity**.

Схематически применение этого подхода можно увидеть в методе **doLogic()** класса **SomeLogic**.

/ # 15 # использование DAO на уровне логики # SomeLogic.java */*

```
package by.bsu.logic;
import java.sql.Connection;
import java.sql.SQLException;
import by.bsu.pool.ConnectionPool;
import by.bsu.simpledao.AbonentDAO;
import by.bsu.simpledao.PaymentDAO;
import by.bsu.subject.Abonent;
import by.bsu.subject.Payment;
public class SomeLogic {
          public void doLogic(int id) throws SQLException {
                    // 1. создание-получение соединения
                    Connection conn = ConnectionPool.getConnection();
                    // 2. открытие транзакции
                    conn.setAutoCommit(false);
                    // 3. инициализация необходимых экземпляров DAO
                    AbonentDAO abonentDAO = new AbonentDAO(conn);
                    PaymentDAO paymentDAO = new PaymentDAO(conn);
                    // 4. выполнение запросов
                    abonentDAO.findAll();
                    paymentDAO.findEntityById(id);
                    paymentDAO.delete(id);
                    // 5. закрытие транзакции
                    conn.commit();
                    // 6. закрытие-возвращение соединения
                    ConnectionPool.close(conn);
          }
}
```

Задания к главе 12

Вариант А

В каждом из заданий необходимо выполнить следующие действия:

- организацию соединения с базой данных вынести в отдельный класс, метод которого возвращает соединение;
- создать БД. Привести таблицы к одной из нормированных форм;
- создать класс для выполнения запросов на извлечение информации из БД с использованием компилированных запросов;
- создать класс на модификацию информации.

1. **Файловая система.** В БД хранится информация о дереве каталогов файловой системы — каталоги, подкаталоги, файлы.
   Для каталогов необходимо хранить:

— родительский каталог;
— название.

Для файлов необходимо хранить:
— родительский каталог;
— название;
— место, занимаемое на диске.

- Определить полный путь заданного файла (каталога).
- Подсчитать количество файлов в заданном каталоге, включая вложенные файлы и каталоги.
- Подсчитать место, занимаемое на диске содержимым заданного каталога.
- Найти в базе файлы по заданной маске с выдачей полного пути.
- Переместить файлы и подкаталоги из одного каталога в другой.
- Удалить файлы и каталоги заданного каталога.

2. **Видеотека.** В БД хранится информация о домашней видеотеке: фильмы, актеры, режиссеры.

   Для фильмов необходимо хранить:
   — название;
   — имена актеров;
   — дату выхода;
   — страну, в которой выпущен фильм.

   Для актеров и режиссеров необходимо хранить:
   — ФИО;
   — дату рождения.

   - Найти все фильмы, вышедшие на экран в текущем и прошлом году.
   - Вывести информацию об актерах, снимавшихся в заданном фильме.
   - Вывести информацию об актерах, снимавшихся как минимум в *N* фильмах.
   - Вывести информацию об актерах, которые были режиссерами хотя бы одного из фильмов.
   - Удалить все фильмы, дата выхода которых была более заданного числа лет назад.

3. **Расписание занятий.** В БД хранится информация о преподавателях и проводимых ими занятиях.

   Для предметов необходимо хранить:
   — название;
   — время проведения (день недели);
   — аудитории, в которых проводятся занятия.

   Для преподавателей необходимо хранить:
   — ФИО;
   — предметы, которые он ведет;
   — количество пар в неделю по каждому предмету;
   — количество студентов, занимающихся на каждой паре.

- Вывести информацию о преподавателях, работающих в заданный день недели в заданной аудитории.
- Вывести информацию о преподавателях, которые не ведут занятия в заданный день недели.
- Вывести дни недели, в которых проводится заданное количество занятий.
- Вывести дни недели, в которых занято заданное количество аудиторий.
- Перенести первые занятия заданных дней недели на последнее место.

4. **Письма.** В БД хранится информация о письмах и отправляющих их людях. Для людей необходимо хранить:
   — ФИО;
   — дату рождения.
   Для писем необходимо хранить:
   — отправителя;
   — получателя;
   — тему письма;
   — текст письма;
   — дату отправки.
   - Найти пользователя, длина писем которого наименьшая.
   - Вывести информацию о пользователях, а также количестве полученных и отправленных ими письмах.
   - Вывести информацию о пользователях, которые получили хотя бы одно сообщение с заданной темой.
   - Вывести информацию о пользователях, которые не получали сообщения с заданной темой.
   - Направить письмо заданного человека с заданной темой всем адресатам.
5. **Сувениры.** В БД хранится информация о сувенирах и их производителях. Для сувениров необходимо хранить:
   — название;
   — реквизиты производителя;
   — дату выпуска;
   — цену.
   Для производителей необходимо хранить:
   — название;
   — страну.
   - Вывести информацию о сувенирах заданного производителя.
   - Вывести информацию о сувенирах, произведенных в заданной стране.
   - Вывести информацию о производителях, чьи цены на сувениры меньше заданной.
   - Вывести информацию о производителях заданного сувенира, произведенного в заданном году.
   - Удалить заданного производителя и его сувениры.

6. **Заказ.** В БД хранится информация о заказах магазина и товарах в них.
   Для заказа необходимо хранить:
   — номер заказа;
   — товары в заказе;
   — дату поступления.
   Для товаров в заказе необходимо хранить:
   — товар;
   — количество.
   Для товара необходимо хранить:
   — название;
   — описание;
   — цену.
   - Вывести полную информацию о заданном заказе.
   - Вывести номера заказов, сумма которых не превосходит заданную и количество различных товаров равно заданному.
   - Вывести номера заказов, содержащих заданный товар.
   - Вывести номера заказов, не содержащих заданный товар и поступивших в течение текущего дня.
   - Сформировать новый заказ, состоящий из товаров, заказанных в текущий день.
   - Удалить все заказы, в которых присутствует заданное количество заданного товара.
7. **Продукция.** В БД хранится информация о продукции компании.
   Для продукции необходимо хранить:
   — название;
   — группу продукции (телефоны, телевизоры и др.);
   — описание;
   — дату выпуска;
   — значения параметров.
   Для групп продукции необходимо хранить:
   — название;
   — перечень групп параметров (размеры и др.).
   Для групп параметров необходимо хранить:
   — название;
   — перечень параметров.
   Для параметров необходимо хранить:
   — название;
   — единицу измерения.
   - Вывести перечень параметров для заданной группы продукции.
   - Вывести перечень продукции, не содержащий заданного параметра.
   - Вывести информацию о продукции для заданной группы.
   - Вывести информацию о продукции и всех ее параметрах со значениями.

- Удалить из базы продукцию, содержащую заданные параметры.
- Переместить группу параметров из одной группы товаров в другую.

8. **Погода.** В БД хранится информация о погоде в различных регионах.
   Для погоды необходимо хранить:
   — регион;
   — дату;
   — температуру;
   — осадки.
   Для регионов необходимо хранить:
   — название;
   — площадь;
   — тип жителей.
   Для типов жителей необходимо хранить:
   — название;
   — язык общения.
   - Вывести сведения о погоде в заданном регионе.
   - Вывести даты, когда в заданном регионе шел снег и температура была ниже заданной отрицательной.
   - Вывести информацию о погоде за прошедшую неделю в регионах, жители которых общаются на заданном языке.
   - Вывести среднюю температуру за прошедшую неделю в регионах с площадью больше заданной.
9. **Магазин часов.** В БД хранится информация о часах, продающихся в магазине.
   Для часов необходимо хранить:
   — марку;
   — тип (кварцевые, механические);
   — цену;
   — количество;
   — реквизиты производителя.
   Для производителей необходимо хранить:
   — название;
   — страна.
   - Вывести марки заданного типа часов.
   - Вывести информацию о механических часах, цена на которые не превышает заданную.
   - Вывести марки часов, изготовленных в заданной стране.
   - Вывести производителей, общая сумма часов которых в магазине не превышает заданную.
10. **Города.** В БД хранится информация о городах и их жителях.
    Для городов необходимо хранить:
    — название;
    — год основания;

— площадь;
— количество населения для каждого типа жителей.
Для типов жителей необходимо хранить:
— город проживания;
— название;
— язык общения.
- Вывести информацию обо всех жителях заданного города, разговаривающих на заданном языке.
- Вывести информацию обо всех городах, в которых проживают жители выбранного типа.
- Вывести информацию о городе с заданным количеством населения и всех типах жителей, в нем проживающих.
- Вывести информацию о самом древнем типе жителей.

11. **Планеты.** В БД хранится информация о планетах, их спутниках и галактиках.
Для планет необходимо хранить:
— название;
— радиус;
— температуру ядра;
— наличие атмосферы;
— наличие жизни;
— спутники.
Для спутников необходимо хранить:
— название;
— радиус;
— расстояние до планеты.
Для галактик необходимо хранить:
— название;
— планеты.
- Вывести информацию обо всех планетах, на которых присутствует жизнь, и их спутниках в заданной галактике.
- Вывести информацию о планетах и их спутниках, имеющих наименьший радиус и наибольшее количество спутников.
- Вывести информацию о планете, галактике, в которой она находится, и ее спутниках, имеющей максимальное количество спутников, но с наименьшим общим объемом этих спутников.
- Найти галактику, сумма ядерных температур планет которой наибольшая.

12. **Точки.** В БД хранится некоторое конечное множество точек с их координатами.
- Вывести точку из множества, наиболее приближенную к заданной.
- Вывести точку из множества, наиболее удаленную от заданной.
- Вывести точки из множества, лежащие на одной прямой с заданной прямой.

13. **Треугольники.** В БД хранятся треугольники и координаты их точек на плоскости.
    - Вывести треугольник, площадь которого наиболее приближена к заданной.
    - Вывести треугольники, сумма площадей которых наиболее приближена к заданной.
    - Вывести треугольники, которые помещаются в окружность заданного радиуса.
14. **Словарь.** В БД хранится англо-русский словарь, в котором для одного английского слова может быть указано несколько его значений и наоборот. Со стороны клиента вводятся последовательно английские (русские) слова. Для каждого из них вывести на консоль все русские (английские) значения слова.
15. **Словари.** В двух различных базах данных хранятся два словаря: русско-белорусский и белорусско-русский. Клиент вводит слово и выбирает язык. Вывести перевод этого слова.
16. **Стихотворения.** В БД хранятся несколько стихотворений с указанием автора и года создания. Для хранения стихотворений использовать объекты типа Blob. Клиент выбирает автора и критерий поиска.
    - В каком из стихотворений больше всего восклицательных предложений?
    - В каком из стихотворений меньше всего повествовательных предложений?
    - Есть ли среди стихотворений сонеты и сколько их?
17. **Четырехугольники.** В БД хранятся координаты вершин выпуклых четырехугольников на плоскости.
    - Вывести координаты вершин параллелограммов.
    - Вывести координаты вершин трапеций.
18. **Треугольники.** В БД хранятся координаты вершин треугольников на плоскости.
    - Вывести все равнобедренные треугольники.
    - Вывести все равносторонние треугольники.
    - Вывести все прямоугольные треугольники.
    - Вывести все тупоугольные треугольники с площадью больше заданной.

Вариант В

Для заданий варианта В гл. 4 создать базу данных для хранения информации. Определить класс для организации соединения (пула соединений). Создать классы для выполнения соответствующих заданию запросов в БД.