ЛАБОРАТОРНАЯ РАБОТА №5. JDBC

Цель: изучить интерфейс JDBC; научиться создавать приложения с доступом к БД (базе данных).

5.1 Теоретические сведения

JDBC (Java DataBase Connectivity) – стандартный прикладной интерфейс языка Java для организации взаимодействия между приложением и системой управления базами данных (СУБД). Взаимодействие осуществляется с помощью драйверов JDBC, обеспечивающих реализацию общих интерфейсов для конкретных СУБД и конкретных протоколов [1].

JDBC предоставляет интерфейс для разработчиков, использующих различные СУБД. С помощью JDBC отсылаются SQL-запросы только к реляционным БД, для которых существуют драйверы, предоставляющие способ общения с реальным сервером базы данных [1, 2].

Далее приводится последовательность действий для выполнения первого запроса.

Подключение библиотеки с классом-драйвером БД

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

mysql-connector-java-[номер версии]-**bin.jar**.

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 4.0 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:

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

В результате возвращен объект Connection и имеет место 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

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

или регистрировать драйвер

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

В большинстве случаев в этом нет необходимости, т. к. экземпляр драйвера загружается автоматически при попытке получения соединения с БД [2].

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 [2, 3].

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 строка запроса помещается в метод executeUpdate().

Обработка результатов выполнения запроса производится методами интерфейса ResultSet, самыми распространенными из которых являются next(), first(), previous(), last(), группа методов по доступу к информации вида getString(int pos), а также аналогичные методы, начинающиеся с getТип(int

pos)(getInt(int pos), getFloat(int pos) и др.) и updateТип(). 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 [2, 4].

Закрытие соединения statemen:

```
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 [4].

Рассмотрим пример использование 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 [2, 4].

Создание простого соединения и простого запроса

Воспользуемся всеми предыдущими инструкциями и создадим пользовательскую БД с именем testphones и одной таблицей PHONEBOOK. Таблица

должна содержать три поля: числовое (первичный ключ) IDPHONEBOOK, символьное LASTNAME, числовое PHONE и несколько занесенных записей (таблица 5.1).

Таблица 5.1 – Пример таблицы из БД

IDPHONEBOOK	LASTNAME	PHONE
1	Иванов	9379992
2	Петров	8415141

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:

```
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 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 phone-
                                book");
                                ArrayList 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 если
                *  произошла ошибка 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, т. к. незакрытое или «провисшее» соединение снижает быстродействие системы. Класс Abonent, используемый приложением для хранения информации, извлеченной из БД, выглядит очень просто:

```
        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;
        }
}

        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=" + last-
        name + "]";
    }
    }
```

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:

```
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 [4]. 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 [4]:

```
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().

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

Хранимая процедура – это в общем случае именованная последовательность команд SQL, рассматриваемая как единое целое и выполняющаяся в адресном пространстве процессов СУБД, которые 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 [2–4]. 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 заключается в том, что CallableStatement способен обрабатывать не только входные (IN), но и выходные (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 следующего 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, 'Ивано')");
st.addBatch("INSERT INTO location VALUES (260, 'Minsk')");
st.addBatch("INSERT INTO student_department VALUES (500, 130)");
// 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 getТип()/setТип() для каждого из его полей и необходимо внести их значения в БД. Многократное использование методов execute() или executeUpdate()

становится неэффективным, и в данном случае лучше использовать схему batch-команд:

```
try {
ArrayList abonents = new ArrayList<>(); // заполнение списка
PreparedStatement statement = con.prepareStatement("INSERT INTO phone-book 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();
}
```

5.2 Содержание отчета

1 Титульный лист.
2 Цель работы.
3 Краткие теоретические сведения.
4 Результаты выполнения практического задания.
5 Выводы.

5.3 Задания к лабораторной работе №5

Используя интерфейс JDBC, совместно с СУБД MySQL создать приложение согласно варианту:

1 Приложение «Справочник телефонных номеров».
2 Приложение для учета товаров на складе.
3 Приложение для учета билетов в кинотеатре.
4 Приложение для учета билетов автовокзала.
5 Приложение учета заявок по ремонту бытовой техники.
6 Приложение учета заказов в интернет-магазине.
7 Приложение для учета успеваемости студентов.
8 Приложение «Электронный отдел кадров».
9 Приложение «Расписание движения поездов».
10 Приложение для учета оказываемых предприятием услуг.
11 Приложение для учета книг в библиотеке.
12 Приложение для учета курсов валют.
13 Приложение для учета футбольных матчей.

14 Приложение для учета выигрышных лотерейных билетов.
15 Приложение для учета результатов тестирования.

Контрольные вопросы

1 Что такое JDBC?
2 Что такое драйвер базы данных?
3 Как выполнить запрос к базе данных?
4 Что называют параметром запроса?
5 С помощью каких интерфейсов можно получить список таблиц, определить типы, свойства и количество столбцов БД?
6 Какой метод возвращает имя указанного столбца в БД?
7 Какой метод возвращает местонахождение источника данных?
8 Для чего используются интерфейс PreparedStatement?
9 Что такое хранимая процедура?
10 Что позволяет использовать интерфейс CallableStatement?