МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное
учреждение высшего профессионального образования
«Юго-Западный государственный университет»
(ЮЗГУ)

Кафедра информационных систем и технологий

УТВЕРЖДАЮ
Проректор по учебной работе
_____ О. Г. Локтионова
«____»_____ 2013 г.

Программирование приложений баз данных на языке Java с использованием интерфейса JDBC

Методические указания по выполнению лабораторной работы по курсам «Управление данными» и «Базы данных» для студентов, обучающихся по направлениям подготовки 230400.62 «Информационные системы и технологии» и 231000.62 «Программная инженерия»

Курск 2013

УДК 004.42, 004.65

Составитель М. В. Бородин

Рецензент
Доктор технических наук, профессор О. И. Атакищев

Программирование приложений баз данных на языке Java с использованием интерфейса JDBC : методические указания по выполнению лабораторной работы / Юго-Зап. гос. ун-т; сост. М. В. Бородин. Курск, 2013. 46 с., ил.: 2, библиогр.: 4.

В методических указаниях описывается интерфейс JDBC и его использование в приложения баз данных, разрабатываемых для платформы Java. Описание сопровождается примерами. Приводится список контрольных вопросов и вариантов заданий. Предназначены для студентов направлений подготовки 230400.62 и 231000.62.

Текст печатается в авторской редакции

Подписано в печать . Формат 60 × 84 1/16.
Усл. печ. л. . Уч.-изд. л. . Тираж 100 экз. Заказ . Бесплатно.
Юго-Западный государственный университет.
305040, г. Курск, ул. 50 лет Октября, 94

1. Цель

Целью настоящей лабораторной работы является освоение студентами навыков использования интерфейса JDBC для доступа к реляционным базам данных из программ, разрабатываемых для платформы Java.

2. Задание

В соответствии с индивидуальным вариантом задания, полученным от преподавателя, требуется:

- определить набор таблиц и представлений базы данных, необходимых прикладной программе;
- определить порядок взаимодействия прикладной программы с базой данных;
- разработать запросы к базе данных на языке SQL;
- разработать структуры данных для представления запрашиваемых данных в памяти;
- реализовать прикладную программу на языке программирования Java.

3. Содержание отчета

Отчет о выполнении работы должен включать в себя:

- титульный лист;
- вариант задания;
- схему базы данных или той ее части, которая используется прикладной программой;
- текст запросов на языке SQL;
- исходный текст программы;
- тесты и описание процедуры тестирования.

4. Теоретические сведения

JDBC (Java Database Connectivity) — это стандартный интерфейс, предоставляемые платформой Java прикладным программам для доступа к реляционным базам данных. JDBC не привязан к какой-либо конкретной СУБД, но может использоваться с любой СУБД при наличии соответствующего драйвера. JDBC входит JDK и не нуждается в отдельной установке; также в состав JDK входят драй-

веры для наиболее распространенных СУБД, таких как MySQL и PostgreSQL.

4.1. Установление соединения

Если приложению необходимо выполнить запрос к базе данных, то первым его шагом должно быть установление соединения с базой данных. Следующий пример показывает, что это можно сделать так…

```
String url = "jdbc:mysql://localhost/tutorial",
       user = "root", password = "qwerty";
try (Connection c = DriverManager.getConnection(url, user,
        password)) {
    // Использовать соединение c
}
```

или так…

```
String url = "jdbc:mysql://localhost/tutorial",
       user = "root", password = "qwerty";
Connection c = DriverManager.getConnection(url, user,
        password);
try {
    // Использовать соединение c
} finally { c.close(); }
```

Как видно из примера для того, чтобы установить соединение с базой данных используется статический метод getConnection, определенный в классе DriverManager. Класс DriverManager представляет менеджер драйверов и его использование является одним из двух основных способов установления соединения.[1] В примере методу getConnection передается три параметра: идентификатор соединения, имя пользователя и пароль.

Идентификатор соединения задает используемый драйвер, а также информацию, необходимую драйверу для установки соеди-

[1] Другим способом является использование источника данных — объекта, реализующего интерфейс DataSource. Этот способ обычно применяется в приложениях, выполняемых в Web-контейнере либо в EJB-контейнере, поскольку в этом случае приложение может не создавать источника данных самостоятельно, а полагаться на то, что упомянутый контейнер сам в нужный момент инъецирует (inject) полностью готовый к использованию источник данных в приложение.

нения. Состав и формат представления этой информации зависит от драйвера; как правило, она включает в себя сетевое имя компьютера, на котором запущен сервер базы данных, а также имя базы данных. Отметим, что именно драйвер отвечает как за установку соединения, так и за все дальнейшее взаимодействие с базой данных; функция менеджера драйверов только в том, чтобы найти подходящий драйвер.

Иногда для успешной установки соединения надо указать дополнительные параметры помимо имени пользователя и пароля. Для этого можно использовать другую (более общую) форму метода getConnection, показанную в следующем примере…

```
Properties p = new Properties();
p.put("user", "root");
p.put("password", "qwerty");
p.put("useUnicode", "true");
p.put("characterEncoding", "utf8");
String url = "jdbc:mysql://localhost/tutorial";
try (Connection c = DriverManager.getConnection(url, p)) {
    // Использовать соединение c
}
```

Из примера видно, что при установке соединения можно задать параметры связанные с кодировкой символов. Полный набор допустимых параметров зависит от драйвера, хотя параметры user и password должны поддерживаться всеми драйверами.

Для того, чтобы менеджер драйверов смог найти драйвер, последний должен быть зарегистрирован. Если драйвер соответствует спецификации JDBC версии 4.0, то все, что необходимо — это наличие класса драйвера в пути к классам (в среде NetBeans для этого достаточно в окне Projects выбрать нужный проект, а в нем — папку Libraries; в контекстном меню этой папки надо выбрать Add Library, после чего выбрать библиотеку, содержащую драйвер в появившемся диалоговом окне; если нужной библиотеки нет — так будет в том случае, если драйвер не поставляется вместе со средой NetBeans и вы не создали для него библиотеки самостоятельно — то в контекстном меню папки Libraries надо выбрать Add JAR/Folder, после чего откроется окно выбора файла). Если же драйвер соответствует более ранней спецификации JDBC нежели версии 4.0, то в этом случае класс драйвера надо указать в систем-

ном свойстве jdbc.drivers (в среде NetBeans для этого можно воспользоваться окном свойств проекта). При необходимости в свойстве jdbc.drivers можно перечислить несколько классов драйверов, разделяя их двоеточием. Также приложение может само зарегистрировать драйвер, для чего достаточно просто загрузить его, воспользовавшись статическим методом forName класса Class.[2]

После того, как приложение выполнит все необходимые запросы к базе данных, оно обязано закрыть соединение. Это можно сделать, как вызвав для объекта соединения метод close, так и воспользовавшись оператором автоматического освобождения ресурсов (второй способ предпочтительнее, однако доступен только начиная с версии 1.7 языка Java и применим только, если ссылка на объект соединения хранится в локальной переменной). Нужно постоянно помнить, что несвоевременное закрытие ненужных соединений приводит к расходу ресурсов (в том числе таких дорогостоящих, как сетевые соединения); это может сделать невозможным создание новых соединений. Вообще говоря, соединение следует рассматривать как короткоживущий объект, который прикладная программа создает только в тот момент, когда возникает необходимость выполнить запрос к базе данных, и удаляет спустя небольшое время после этого (допустимо кэширование на непродолжительное время).

4.2. Выполнение запросов

Соединение не может использоваться непосредственно для выполнения запросов. Вместо этого для выполнения запросов прикладная программа должна создать с использованием соединения оператор. Как сделать это и как выполнить простейший запрос на определение схемы данных показывает следующий пример…

```
// в переменной c хранится ранее созданное соединение
String sql = "CREATE TABLE student ("
           + " st_id INT NOT NULL AUTO_INCREMENT,"
           + " st_name VARCHAR(50) NOT NULL,"
           + " PRIMARY KEY (st_id))";
try (Statement s = c.createStatement()) {
```

[2] Этот метод при необходимости загружает и инициализирует указанный класс. Обязательной частью инициализации класса драйвера является его регистрация в менеджере драйверов.

```
    s.execute();
}
```

Как видно из примера, оператор создается методом createStatement; после этого выполнить запрос можно, воспользовавшись методом execute, которому в виде строки передается текст запроса. Подобно соединениям ненужные операторы подлежат обязательному закрытию прикладной программой.

С использованием одного оператора можно выполнить несколько запросов. В этом случае можно либо просто нужно число раз вызвать метод execute, либо использовать пакетный режим выполнения. Как это сделать показывает следующий пример…

```
// в переменной s хранится ранее созданный оператор
s.addBatch("ALTER TABLE student ADD st_address
VARCHAR(50)");
s.addBatch("ALTER TABLE student ADD st_phone VARCHAR(50)");
s.executeBatch();
```

При выполнении в пакетном режиме прикладная программа сначала добавляет запросы в пакет, используя метод addBatch, после чего выполняет все содержимое пакета, вызывая метод executeBatch. Использование пакетного режима может сократить коммуникационные издержки при выполнении большого количества запросов.

4.3. Результирующие множество

Если запрос возвращает множество строк,[3] то для их выборки прикладная программа должна использовать специальный объект, называемый в JDBC результирующим множеством. Как это сделать показывается в следующем примере…

```
// в переменной s хранится ранее созданный оператор
String sql = "SELECT * FROM student";
try (ResultSet rs = s.executeQuery(sql)) {
    while (rs.next()) {
```

[3] Множество строк не является множеством (set) в строго математическом смысле; скорее это — последовательность (sequence). Дело в том, что, во-первых, множество строк может содержать повторяющиеся строки а, во-вторых, порядок строк во множестве строк вовсе не обязательно произвольный. Тем не менее, мы будем использовать термин «множество» в силу сложившейся традиции.

```
            int id = rs.getInt("st_id");
            String name = rs.getString("st_name");
            // как-то обработать значения id и name
        }
}
```

Как видно из примера, для выполнения запроса и получения возвращаемого им результирующего множества используется метод executeQuery. Отметим, что прикладная программа может вызывать метод executeQuery только в том случае, когда заранее известно, что запрос вернет множество строк; в противном случае надо сначала вызвать метод execute для выполнения запроса, а затем — метод getResultSet для получения результирующего множества; если результирующего множества нет, то метод getResultSet возвращает значение null.

С каждым результирующим множеством неявно связан курсор, позиция которого может находиться либо на одной из строк результирующего множества либо перед первой строкой либо после последней. Первоначально курсор находится перед первой строкой. Для передвижения курсора к следующей позиции используется метод next; в случае успешного перемещения (т. е., если курсор не оказался после последней строки) новая строка становится текущей, а метод возвращает значение true.

Хотя результирующее множество может содержать много строк, только значения, находящиеся в текущей строке доступны прикладной программе. В зависимости от предполагаемого типа значения, следует использовать соответствующий метод получения значения: getInt — для целых чисел, getDouble — для вещественных чисел, getString — текстовых строк и так далее; однако, строгое соответствие между типом значения в строке и методом, использующимся для его получения не всегда обязательно: например, метод getString можно использовать для получения значений, числовых типов, поскольку всякое число может быть преобразовано в текстовую строку.

Если некоторое значение в строке результирующего множества пусто, то при попытке его получения соответствующий метод возвращает значение по умолчанию: 0 — для числовых типов, false — для логического типа и null — для объектных типов. Чтобы отличить нулевое и ложное значение от пустого значения можно ис-

пользовать метод wasNull; этот метод не имеет параметров и возвращает значение true, если последнее по времени получаемое значение было пустым.[4]

4.4. Использование параметров в запросах

Если один и тот же запрос необходимо выполнить несколько раз или же, если запрос должен содержать параметры, следует использовать подготовленные операторы. Как это сделать показано в следующем примере…

```
// в переменной c хранится ранее установленное соединение
String sql = "UPDATE student SET st_name = ?"
           + " WHERE st_name = ?";
try (PreparedStatement ps = c.prepareStatement(sql)) {
    ps.setString(1, "Сидорова А.");
    ps.setString(2, "Петрова А.");
    int n = ps.executeUpdate();
}
```

Как видно из примера при использовании параметров текст запроса задается непосредственно при создании оператора, а не при вызове методов execute, executeUpdate и executeQuery, как это имело место выше. После создания объекта оператора и перед его выполнением прикладная программа должно задать значения всех параметров используя для этого методы setString, setInt и т. п. В качестве первого параметра эти методы принимают порядковый номер параметра запроса, соответствующий положению маркера параметра (вопросительного знака) в тексте запроса. Нумерация параметров запроса начинается с единицы; именованные параметры JDBC не поддерживает.

Если значение параметра должно быть пустым надо использовать метод setNull (просто пропустить такой параметр нельзя). Метод setNull принимает два параметра: первый — это номер параметра запроса (как у ранее рассмотренных методов), а второй — код типа данных языка SQL. Допустимые коды типов являются константами (статическими неизменяемыми полями) класса Types;

---

[4] Представьте себе, что каждый вызов метода получения значения не только возвращает значение, но и присваивает некоторой скрытой переменной значение true или false в зависимости от того, было ли получаемое значение пустым; значение этой скрытой переменной и возвращает метод wasNull.

среди них есть VARCHAR, NVARCHAR, INTEGER, NUMERIC, DECIMAL, DATE и т. п.

Заканчивая рассматривать пример заметим, что метод executeUpdate возвращает количество строк, обновленных, вставленных или удаленных соответствующих оператором модификации данных.

Подготавливаемые операторы, так же как и обычные, поддерживают пакетный режим. Как пользоваться им показывает следующий пример…

```
// в переменной c хранится ранее установленное соединение
String sql = "INSERT INTO student(st_name) VALUES(?)";
try (PreparedStatement ps = c.prepareStatement(sql,
        Statement.RETURN_GENERATED_KEYS)) {
    ps.setString(1, "Иванов И.");
    ps.addBatch();
    ps.setString(1, "Петров П.");
    ps.addBatch();
    ps.executeBatch();
    ResultSet rs = ps.getGeneratedKeys();
    while (rs.next()) {
        int id = rs.getInt(1);
        // сделать что-то с id
    }
}
```

Как видно из примера при использовании подготовленного оператора элементы пакета различаются между собой только значениями параметров. Кроме пакетного режима этот пример показывает, как выбрать сгенерированные ключи.[5] Во-первых, необходимость выборки сгенерированных ключей нужно явно указать еще при создании оператора, для чего используется второй (необязательный) параметр метода prepareStatement. Во-вторых, после выполнения оператора для выборки ключей следует воспользоваться методом getGeneratedKeys. Отметим, что драйвер может и не поддерживать возможность выборки сгенерированных ключей.

[5] Предполагается, что в таблице student столбец первичного ключа объявлен с использованием GENERATED ALWAYS AS IDENTITY или какой-либо другой аналогичной конструкции (в случае MySQL — это AUTO_INCREMENT).

Если запрос состоит в выполнении хранимой функции либо хранимой процедуры, имеющей выходные параметры, то для выполнения такого запроса надо использовать не подготовленный, а вызываемый оператор. Как это сделать показывает следующий пример…

```
// в переменной c хранится ранее созданное соединение
int bestStudentId;
String sql = "{call get_best_student_id(?)}";
try (CallableStatement cs = c.prepareCall(sql)) {
    cs.registerOutParameter("st_id", Types.INTEGER);
    cs.execute();
    bestStudentId = cs.getInt("st_id");
}
```

Как видно из примера перед выполнением запроса прикладная программа должна зарегистрировать в вызываемом операторе все выходные параметры, указав при этом их тип данных. После выполнения запроса значения выходных параметров можно получить, используя для этого такие методы как getInt и getString. В том случае если значение выходного параметра может быть пустым, проверить это можно с помощью метода wasNull (как и в случае результирующего множества перед использованием этого метода необходимо попытаться получить значение соответствующего параметра одним из предназначенных для этого методов). Отметим, что в отличие от подготавливаемого оператора выполняемый оператор позволяет указывать параметры, как по порядковым номерам, так и по названиям.

Приведенный пример демонстрирует использование в тексте запроса escape-последовательности — JDBC поддерживает те же escape-последовательности, что и ODBC. Разумеется, в JDBC escape-последовательности можно использовать не только с вызываемыми, но также и подготовленными и обычными операторами.

4.5. Прокручиваемые и обновляемые курсоры

Последовательные перебор всех строк начиная с первой не является единственно возможным способом выборки. Часто драйвер поддерживает курсоры с прокруткой. Как воспользоваться этой возможностью показано в следующем примере…

```
// в переменной c хранится ранее установленное соединение
```

```
String sql = "SELECT * FROM student ORDER BY st_name";
try (Statement s = c.createStatement(
        ResultSet.TYPE_SCROLL_INSENSITIVE,
        ResultSet.CONCUR_READ_ONLY)) {
    try (ResultSet rs = s.executeQuery(sql)) {
        // Попытаться выбрать строки с 11 по 20
        if (rs.absolute(11)) {
            for (int i = 0; i < 10; ++i) {
                // Что-то сделать с выбранной строкой
                if (!rs.next()) {
                    break;
                }
            }
        }
    }
}
```

Как видно из примера для перемещения курсора на нужную строку результирующего множества можно использовать метод absolute, которому передается номер строки, причем нумерация строк начинается с 1 (0 соответствует позиции перед первой строкой). Методу absolute можно передавать отрицательные номера; в этом случае –1 означает последнюю строку, –2 — предпоследнюю и т. д.

Помимо методов absolute и next для перемещения курсора можно использовать методы previous (на предыдущую строку), first (на первую строку), last (на последнюю строку), beforeFirst (в позицию перед первой строкой), afterLast (в позицию после последней строки), а также метод relative, который позволяет переместить курсор на указанное количество строк относительно текущей строки; подобно методу absolute метод relative имеет один целочисленный параметр, причем отрицательные значения параметра соответствуют перемещению курсора к началу множества.

Отметим, что если приложению необходим прокручиваемый курсор, то это должно быть указано еще при создании соответствующего оператора, для чего используется первый параметр метода createStatement либо второй параметр метода prepareStatement. Помимо значения TYPE_SCROLL_INSENSITIVE, означающего нечувствительный прокручиваемый курсор, возможны также значения TYPE_SCROLL_SENSITIVE (чувствительный прокручиваемый) и TYPE_FORWARD_ONLY (без возможности прокрутки — вариант по умолчанию).

Заметим, что прокручиваемые курсоры не должны использоваться в качестве альтернативы предложениям OFFSET и FETCH FIRST оператора SELECT. Дело в том, что при использовании прокручиваемых курсоров большинство драйверов кэшируют в памяти прикладной программы все результирующее множество (впрочем, многие драйверы делают это вне зависимости от вида курсора). Поскольку следует максимально сокращать нагрузку на сервер и сетевой трафик, надо стремиться к минимизации результирующих множеств, что, в свою очередь предполагает отбор необходимых строк непосредственно на сервере (на основе присутствующих в запросе критериев) и, тем самым, освобождение последнего от затрат по передаче клиенту заведомо не нужных тому строк.

Помимо прокрутки строк результирующего множества в JDBC есть возможность модификации этого множества. Как это сделать показано в следующем примере…

```
// в переменной c хранится ранее установленное соединение
String sql = "SELECT * FROM student ORDER BY st_name"
           + " FOR UPDATE";
try (Statement s = c.createStatement(
        ResultSet.TYPE_SCROLL_SENSITIVE,
        ResultSet.CONCUR_UPDATABLE)) {
    try (ResultSet rs = s.executeQuery(sql)) {
        String oldName = "Петрова А.";
        String newName = "Иванова А.";
        while (rs.next()) {
            if (oldName.equals(rs.getString("st_name"))) {
                rs.updateString("st_name", newName);
                rs.updateRow();
            }
        }
    }
}
```

Как видно из примера модификация строки результирующего множества выполняется в несколько шагов. Сначала с использованием методов updateString, updateInt и подобных им задаются новые значения столбцов, которые сохраняются в памяти; затем, после задания всех новых значений производится запись обновленной строки в базу данных, для чего используется метод updateRow. Методы обновления значений переводят текущую строку в состояние обновления (если она уже не находилась в этом состоянии).

Возврат из состояния обновления происходит при вызове метода updateRow (при этом изменения значений сохраняются), а также при вызове метода cancelRowUpdates или при переводе курсора на другую строку результирующего множества (в этом случае изменения значений отменяются).

Следующий пример показывает, что надо сделать, чтобы вставить в результирующее множество новые строки…

```
// в переменной s хранится оператор,
// поддерживающий обновление
String sql = "SELECT * FROM student FOR UPDATE";
try (ResultSet rs = s.executeQuery(sql)) {
    rs.moveToInsertRow();
    try {
        rs.updateString("st_name", "Иванов И.");
        rs.insertRow();
        rs.updateString("st_name", "Петров П.");
        rs.insertRow();
    } finally { rs.moveToCurrentRow(); }
}
```

Как видно из примера вставка новых строк также выполняется в несколько шагов. Сначала с использованием метода moveToInsertRow производится перевод курсора на т. н. вставляемую строку, причем перед этим текущее положение курсора автоматически запоминается. Вставляемая строка — это специальная временная строка, не входящая в число строк результирующего множества и служащая своего рода шаблоном для новых строк. После перехода на вставляемую строку производится ее заполнение с использованием методов обновления значения (т. е. методов updateString, updateInt и т. п.). После заполнения заполненную строку фактически вставляют в таблицу базы данных, для чего используется метод insertRow; после вставки вставляемая строка автоматически очищается. Метод insertRow оставляет курсор на вставляемой строке, что может использоваться для того, чтобы вставить еще одну строку или несколько строк. После того, как все необходимые строки вставлены курсор следует снова установить на ту строку результирующего множества, на которой он находился до перехода на вставляемую строку; для этого используется метод moveToCurrentRow.

Для удаления текущей строки из результирующего множества используется метод deleteRow, а для повторного считывания текущей строки из базы данных — метод refreshRow; последнее может оказаться полезным в том случае, когда текущая строка в базе данных изменилась (это может произойти в результате действий как данной, так и какой-либо другой транзакции, если используется низкий уровень изолированности).

Вообще, основное назначение рассмотренных данном разделе методов — упрощение привязки к данным в пользовательском интерфейсе (как в Delphi). Впрочем, в Java никогда не было (и, видимо, не будет) стандартных элементов управления с привязкой непосредственно к базе данных. Современный подход к проектированию приложений предполагает использование моделей (парадигма Модель–Представление–Контроллер). В приложениях баз данных модель играет роль посредника между представлением и базой данных (однако назначение моделей не сводится только к этому) и, стало быть, необходимость в развитой функциональности на уровне результирующего множества отпадает.

Все же упомянутые методы могут оказаться очень полезны при разработке всякого рода прототипов, когда все остальные критерии могут быть принесены в жертву быстроте получения (кое-как) работающих форм, единственная цель которых продемонстрировать, как будет в будущем выглядеть законченное приложение.

4.6. Управление транзакциями

Соединения используются не только для создания операторов — в интерфейсе Connection также определены методы для управления транзакциями. По умолчанию каждое соединение находится в режиме автоматического подтверждения транзакций; это означает, что выполнение каждого оператора автоматически помещается в отдельную транзакцию. Разумеется, такой режим избавляет прикладную программу от необходимости явно управлять транзакциями, но вместе с этим не дает возможности выполнить несколько операторов в рамках одной транзакции. Чтобы отключить режим автоматического подтверждения транзакций надо вызвать метод setAutoCommit, передав ему в качестве параметра значение false. После этого для разграничения транзакций прикладная программа

должна вызывать методы commit и rollback, предназначенные соответственно для подтверждения и отката текущей транзакции.

Необходимо помнить, что инициатором отката транзакции может быть не только прикладная программа, но и сервер базы данных. Последнее может случиться, например, при обнаружении взаимной блокировки. Если транзакция откатывается по инициативе сервера базы данных, то соответствующее обращение к базе данных в прикладной программе приводит к выбросу исключения (этим исключением вовсе не обязано быть исключение SQLTransactionRollbackException; например, если откат транзакции вызван нарушением ограничения целостности, то будет выброшено исключение SQLIntegrityConstraintViolationException).

Кроме непосредственного управления транзакциями прикладная программа может изменять характеристики транзакций. Чтобы задать уровень изоляции транзакции можно вызвать метод setTransactionIsolation, передав ему в качестве параметра уровень изолированности, представленный одной из констант, определенных в интерфейсе Connection: TRANSACTION_READ_UNCOMMITTED — неподтвержденное чтение, TRANSACTION_READ_COMMITTED — подтвержденное чтение, TRANSACTION_REPEATABLE_READ — повторяемое чтение и TRANSACTION_SERIALIZABLE — упорядочивание. С целью снижения нагрузки на сервер базы данных прикладная программа должна всегда устанавливать наименьший приемлемый для данной транзакции уровень изолированности. Также можно перевести транзакцию в режим «только для чтения»; для этого надо вызвать метод setReadOnly, передав ему значение true. Использование транзакций в режиме «только для чтения» может помочь серверу базы данных лучше оптимизировать выполнение запросов и, тем самым, повысить производительность.

Изменение характеристик транзакции должно производиться либо сразу же после установления соединения либо после подтверждения предыдущей транзакции. Действуют изменения характеристик транзакции на все последующие транзакции данного соединения (вплоть до следующего изменения).

4.7. Работа с большими объектами

Данные потенциального большого размер, хранящиеся в столбцах таблицы базы данных, называют большими объектами. Большие объекты бывают символьным и двоичными. В JDBC поддерживается три способа работы с большим объектами.

Первый способ — это, в случае символьного объекта, использовать ранее упоминавшиеся методы getString и setString, а в случае двоичного — методы getBytes и setBytes, представляющие объект в виде массива байтов. Разумеется, при использовании этого способа весь объект сразу передается прикладной программе и полностью помещается в память. Вместе с тем, этот способ обещает наибольшую производительность и может рассматриваться как способ по умолчанию.

Второй способ — это использование потоков ввода-вывода. Как это сделать демонстрирует следующий пример…

```
String sql = "INSERT INTO article(a_id,a_title,a_content)"
          + " VALUES(?, ?, ?)";
try (Reader r = new FileReader("article.txt")) {
    // в переменной с хранится ранее созданное соединение
    try (PreparedStatement ps = c.prepareStatement(sql)) {
        ps.setInt(1, 1001);
        ps.setString(2, "О чем-то");
        ps.setCharacterStream(3, r);
        ps.executeUpdate();
    }
}
```

Как видно из примера для задания большого объекта используется метод setCharacterStream, вторым параметром которого является символьный поток. В данном случае поток имеет своим источником файл article.txt; однако, вообще можно использовать любой объект, реализующий интерфейс Reader (например, в Web-приложении поток может представлять тело HTTP-запроса; более того используя канал приложение может динамически генерировать содержимое потока). Метод setCharacterStream имеет необязательный третий параметр — это целое число, задающее максимальное количество символов, которое может быть считано из потока. Метод setCharacterStream применяется для символьных объектов; для двоичных объектов предназначен метод setBinaryStream,

работающий с двоичным потоком, представленным интерфейсом InputStream. При использовании методов setCharacterStream и setBinaryStream следует помнить, что обязанность закрытия потоков, передаваемых в лежит на приложении.

Потоки можно использовать не только для задания, но и для получения больших объектов. Для этого в интерфейсе ResultSet определены методы getCharacterStream и getBinaryStream; единственным параметром этих методов является номер или название столбца результирующего множества. При использовании методов getCharacterStream и getBinaryStream следует учитывать, что каждый следующий вызов одного из этих методов автоматически закрывает поток, возвращенный предыдущим вызовом; также потоки, возвращаемые методами getCharacterStream и getBinaryStream автоматически закрываются при переходе к другой строке и при закрытии набора строк.[6]

Третий способ работы с большими объектами — это использование объектов, реализующих интерфейсы Clob и Blob.[7] Как это сделать демонстрирует следующий пример…

```
// в переменной rs хранится результирующее множество
// (т. е. ResultSet)
Clob clob = rs.getClob("a_content");
try {
    // запись большого объекта в файл блоками по 1K
    final int m = 1024;
    long i = 1, n = clob.length();
    String pathname = String.format("art-%04d.txt",
            rs.getInt("a_id"));
    try (Writer w = new FileWriter(pathname)) {
        for (long j = i + m; j < n; j += m) {
            w.write(clob.getSubString(i, m));
            i = j;
        }
        w.write(clob.getSubString(i, (int) (n - i + 1)));
    }
} finally { clob.free(); }
```

[6] Строго говоря, драйвер не обязан немедленно закрывать поток во всех этих случаях — в принципе, драйвер может закрыть поток тогда, когда ему это будет удобно; однако, прикладная программа ни в коем случае не должна рассчитывать на такое поведение драйвера.

[7] В последнем случае имеются в виду объекты языка программирования Java.

Как видно из примера для получения упомянутого объекта, реализующего интерфейс Clob, используется метод getClob, параметром которого должен быть номер или название соответствующего столбца. Для получения размера большого объекта в интерфейсе Clob предусмотрен метод length, а для выборки подстроки (т. е. части большого объекта) — метод getSubString. Параметрами метода getSubString является начальная позиция подстроки и ее размер, причем позиции нумеруются начиная с 1, а не с 0. Помимо метода getSubString в интерфейсе Clob для получения данных предусмотрен метод getCharacterStream; у этого метода две формы: без параметров и с двумя параметрами аналогичными параметрам метода getSubString.

Кроме выборки интерфейс Clob позволяет искать вхождение заданной подстроки; для этого используется метод position, параметрами которого являются искомая подстрока и позиция, с которой следует начать поиск. В случае успеха метод position возвращает найденную позицию, а при неудаче — –1.

С помощью интерфейса Clob можно изменять большой объект. Как это сделать показано в следующем примере…

```
// в переменной c хранится ранее созданное соединение
String sql = "INSERT INTO article(a_id,a_title,a_content)"
           + " VALUES(?, ?, ?)";
try (PreparedStatement ps = c.prepareStatement(sql)) {
    ps.setInt(1, 2001);
    ps.setString(2, "Некоторое название");
    Clob clob = c.createClob();
    try {
        try (Writer w = clob.setCharacterStream(1)) {
            // используя w записать что-нибудь подходящее
        }
        ps.setClob(3, clob);
        ps.executeUpdate();
    } finally { clob.free(); }
}
```

Как видно из примера для создания нового (временного) большого объекта, следует использовать метод createClob. После этого для записи в него можно использовать выходной символьный поток, возвращаемый методом setCharacterStream, параметром которого является номер позиции, начиная с которой будет производиться запись. Подобно файлу в процессе записи большой объект при не-

обходимости увеличивается. Кроме метода setCharacterStream для записи большого объекта можно использовать метод setString, принимающий два параметра, первый из которых задает позицию, а второй — записываемую строку. Помимо записи данных допускается также удаление, для чего используется метод truncate, единственный параметр которого задает нужную длину большого объекта, которая должна быть меньше текущей.

Методы, изменяющие содержимое, можно применять не только к тем большим объектам, которые созданы методом createClob, но также и к тем, которые связаны с результирующим множеством и возвращаются методом getClob. В последнем случае, однако, необходимо, чтобы соответствующий оператор SELECT содержал предложение FOR UPDATE, накладывающее блокировку на выбираемые данные.

Важным преимуществом методов getClob и createClob является то, что возвращаемые или объекты можно использовать вплоть до конца транзакции.

Аналогом интерфейса Clob, используемым при работе большими двоичными объектами является интерфейс Blob. Вместо строк в интерфейсе Blob используются массивы байтов, для работы с которыми вместо методов getSubString и setString предусмотрены методы getBytes и setBytes. Также интерфейс Blob поддерживает работу с потоками, для чего можно использовать методы getBinaryStream и setBinaryStream.

5. Практический пример

Предположим, что необходимо разработать приложение для учета полученных счетов-фактур.

5.1. Схема данных

Ради сокращения примера примем ряд упрощений по сравнению с действующей формой счета фактуры, а также откажемся от использования справочника товаров. Выберем СУБД MySQL и будем использовать следующую весьма упрощенную схему данных:

```
-- счет-фактура
CREATE TABLE invoice (
  id INT NOT NULL AUTO_INCREMENT, -- идентификатор
  ver INT NOT NULL DEFAULT 1, -- версия
```

```
  num VARCHAR(10) NOT NULL, -- номер счета-фактуры
  idate DATE NOT NULL, -- дата счета-фактуры
  sname VARCHAR(50) NOT NULL, -- наименование продавца
  saddr VARCHAR(90) NOT NULL, -- адрес продавца
  total DECIMAL NOT NULL DEFAULT 0, -- итоговая сумма
  PRIMARY KEY (id)
);
-- строка счета-фактуры с информацией о товаре
CREATE TABLE invoice_line (
  id INT NOT NULL, -- идентификатор
  ord INT NOT NULL, -- порядковый номер
  prod VARCHAR(100) NOT NULL, -- описание товара
  qty DECIMAL NOT NULL, -- количество
  price DECIMAL NOT NULL, -- цена
  PRIMARY KEY (id, ord)
);
ALTER TABLE invoice_line
  ADD FOREIGN KEY (id) REFERENCES invoice(id
    ON DELETE CASCADE;
```

Примечание: будет считать, что значения в столбце total обновляются триггером, исходный код которого здесь опустим.

Для предотвращения потерянных обновлений (lost update) при выполнении прикладных транзакций будет использоваться оптимистическая блокировка на основе версий (столбец ver в таблице invoice).

5.2. Исходный код

Далее приведем исходный код приложения с комментариями (ради уменьшения объема комментарии выполнены с отступлением от стандарта JavaDoc; студентам, выполняющим лабораторные работы рекомендуется придерживаться этого стандарта). По соображениям экономии объема из приводимо исходного кода будет исключены некоторые классы, связанные с реализацией пользовательского интерфейса; предполагается, что студентам будет нетрудно воспроизвести их на основе снимков экрана, которые будут приведены в следующем разделе. Также не будут приведены некоторые тривиальные классы, реализация которых совершенно очевидна из того, как они используются.

1. Файл DataConfiguration.java

```
package invoices;
```

```
import java.io.*;
import java.sql.*;
import java.util.Properties;
// инкапсулирует установку соединения с БД на основе
// параметров, загружаемых из файла, находящегося
// в домашнем каталоге пользователя
public class DataConfiguration {
  private static DataConfiguration user;
  private final String url;
  private final Properties info;
  private DataConfiguration(String url, Properties info) {
    this.url = url;
    this.info = info;
  }
  public Connection getConnection() throws SQLException {
    return DriverManager.getConnection(url, info);
  }
  public String getUrl() {
    return url;
  }
  public Properties getInfo() {
    return info;
  }
  public static synchronized DataConfiguration user()
          throws DataException {
    if (null == user) {
      user = createUser();
    }
    return user;
  }
  private static DataConfiguration createUser()
          throws DataException {
    Properties p = new Properties();
    try {
      System.out.println(System.getProperty("user.home"));
      File file = new File(System.getProperty("user.home"),
              ".invoices.properties");
      try (Reader reader = new FileReader(file)) {
        p.load(reader);
      }
    } catch (FileNotFoundException ex) {
      throw new DataException("Ask your administrator to"
              + " configure this application.", ex);
    } catch (IOException ex) {
      throw new DataException(ex);
    }
    Properties info = new Properties();
```

```
    info.setProperty("user", p.getProperty("user"));
    info.setProperty("password",p.getProperty("password"));
    info.setProperty("characterEncoding", "utf-8");
    return new DataConfiguration(
            String.format("jdbc:mysql://%s:%s/%s",
            p.getProperty("host"),
            p.getProperty("port"),
            p.getProperty("database")), info);
  }
}
```

2. Файл DataEnvironment.java

```
package invoices;
import java.awt.event.*;
import java.sql.*;
import javax.swing.Timer;
// инкапсулирует управление соединение с БД: для снижения
// нагрузки на сервер БД в случае неактивности пользователя
// в течении 5 минут соединение автоматически закрывается
public class DataEnvironment {
  public static final int DELAY = 5 * 60 * 1000;
  private Connection connection;
  private final Timer timer;
  private static ThreadLocal<DataEnvironment> current;
  private DataEnvironment() {
    timer = new Timer(DELAY, new ActionListener() {
      @Override
      @SuppressWarnings("CallToThreadDumpStack")
      public void actionPerformed(ActionEvent e) {
        try {
          connection.close();
          connection = null;
        } catch (SQLException ex) {
          ex.printStackTrace();
        }
      }
    });
    timer.setRepeats(false);
  }
  public static DataEnvironment current() {
    return current.get();
  }
  public static void setCurrent(
          DataEnvironment newCurrent) {
    current.set(newCurrent);
  }
```

```
  public <T> T withinTransaction(DataQuery<T> query)
          throws DataException, SQLException {
    Connection c = getConnection();
    c.setAutoCommit(false);
    boolean succeed = false;
    try {
      T result = query.perform(c);
      succeed = true;
      return result;
    } finally {
      if (succeed) {
        c.commit();
      } else {
        c.rollback();
      }
    }
  }
  private Connection getConnection() throws SQLException,
          DataException {
    if (null == connection) {
      connection =
          DataConfiguration.user().getConnection();
      timer.restart();
    }
    return connection;
  }
  static {
    current = new ThreadLocal<DataEnvironment>() {
      @Override
      protected DataEnvironment initialValue() {
        return new DataEnvironment();
      }
    };
  }
}
```

3. Файл DataQuery.java

```
package invoices;
import java.sql.*;
// используется для реализации «инверсии управления»
// при обращении к БД
public interface DataQuery<T> {
  T perform(Connection connection) throws DataException,
          SQLException;
}
```

4. Файл InvoiceModel.java

```
package invoices;
import java.math.BigDecimal;
import java.util.*;
import org.jdesktop.observablecollections.*;
// полная информации о счете-фактуре,
// используемая в диалоговом окне редактирования
public class InvoiceModel extends AbstractBean {
  public static final String PROP_NUMBER = "number";
  public static final String PROP_DATE = "date";
  public static final String PROP_TOTAL = "total";
  private int id;
  private int version;
  private String number = "";
  private Date date = new Date(System.currentTimeMillis());
  private final ContactModel seller;
  private final ObservableList<InvoiceLineModel> lines;
  private BigDecimal total = BigDecimal.ZERO;
  public InvoiceModel() {
    seller = new ContactModel();
    lines = ObservableCollections.observableList(
            new ArrayList<InvoiceLineModel>());
    lines.addObservableListListener(new LinesListener());
  }
  public int getId() {
    return id;
  }
  public void setId(int newId) {
    id = newId;
  }
  public int getVersion() {
    return version;
  }
  public void setVersion(int newVersion) {
    version = newVersion;
  }
  public String getNumber() {
    return number;
  }
  public void setNumber(String newNumber) {
    if (!Objects.equals(newNumber, number)) {
      String oldNumber = number;
      number = newNumber;
      firePropertyChange(PROP_NUMBER, oldNumber,
              newNumber);
    }
```

```
}
public Date getDate() {
  return (Date) date.clone();
}
public void setDate(Date newDate) {
  if (!Objects.equals(newDate, date)) {
    Date oldDate = date;
    date = (Date) newDate.clone();
    firePropertyChange(PROP_DATE, oldDate,
            newDate);
  }
}
public ContactModel getSeller() {
  return seller;
}
public ObservableList<InvoiceLineModel> getLines() {
  return lines;
}
public BigDecimal getTotal() {
  return total;
}
public InvoiceProxy createProxy() {
  return InvoiceProxy.createBuilder().id(id)
          .version(version).number(number).date(date)
          .seller(seller.getName()).total(total)
          .build();
}
void lineAmountChanged(InvoiceLineModel line,
        BigDecimal oldAmount) {
  setTotal(total.add(line.getAmount()
          .subtract(oldAmount)));
}
private void setTotal(BigDecimal newTotal) {
  if (!Objects.equals(newTotal, total)) {
    BigDecimal oldTotal = total;
    total = newTotal;
    firePropertyChange(PROP_TOTAL, oldTotal, newTotal);
  }
}
private void lineAdded(int lineIndex) {
  InvoiceLineModel line = lines.get(lineIndex);
  line.setParent(this);
  setTotal(total.add(line.getAmount()));
}
private void lineRemoved(Object lineElement) {
  InvoiceLineModel line = (InvoiceLineModel) lineElement;
  setTotal(total.subtract(line.getAmount()));
```

```
    line.setParent(null);
  }
  private class LinesListener
          implements ObservableListListener {
    @Override
    public void listElementsAdded(ObservableList list,
            int index, int length) {
      for (int i = 0; i < length; ++i) {
        lineAdded(index + i);
      }
    }
    @Override
    public void listElementsRemoved(ObservableList list,
            int index, List oldElements) {
      for (int i = 0; i < oldElements.size(); ++i) {
        lineRemoved(oldElements.get(i));
      }
    }
    @Override
    public void listElementReplaced(ObservableList list,
            int index, Object oldElement) {
      lineRemoved(oldElement);
      lineAdded(index);
    }
    @Override
    public void listElementPropertyChanged(
            ObservableList list, int index) {
      throw new RuntimeException("Should never happen.");
    }
  }
}
```

5. Файл InvoiceLineModel.java

```
package invoices;
import java.math.BigDecimal;
import java.util.Objects;
// строка счета-фактуры
public class InvoiceLineModel extends AbstractBean {
  public static final String PROP_PRODUCT = "product";
  public static final String PROP_QUANTITY = "quantity";
  public static final String PROP_PRICE = "price";
  public static final String PROP_AMOUNT = "amount";
  private InvoiceModel parent;
  private String product = "";
  private BigDecimal quantity = BigDecimal.ZERO;
  private BigDecimal price = BigDecimal.ZERO;
```

```
private BigDecimal amount;
public InvoiceLineModel() {
  amount = computeAmount();
}
public InvoiceModel getParent() {
  return parent;
}
public String getProduct() {
  return product;
}
public void setProduct(String newProduct) {
  if (!Objects.equals(newProduct, product)) {
    String oldFreight = product;
    product = newProduct;
    firePropertyChange(PROP_PRODUCT, oldFreight,
            newProduct);
  }
}
public BigDecimal getQuantity() {
  return quantity;
}
public void setQuantity(BigDecimal newQuantity) {
  if (!Objects.equals(newQuantity, quantity)) {
    BigDecimal oldQuantity = quantity;
    quantity = newQuantity;
    firePropertyChange(PROP_QUANTITY, oldQuantity,
            newQuantity);
    setAmount(computeAmount());
  }
}
public BigDecimal getPrice() {
  return price;
}
public void setPrice(BigDecimal newPrice) {
  if (!Objects.equals(newPrice, price)) {
    BigDecimal oldPrice = price;
    price = newPrice;
    firePropertyChange(PROP_PRICE, oldPrice, newPrice);
    setAmount(computeAmount());
  }
}
public BigDecimal getAmount() {
  return amount;
}
void setParent(InvoiceModel newParent) {
  parent = newParent;
}
```

```
  private void setAmount(BigDecimal newAmount) {
    if (!Objects.equals(newAmount, amount)) {
      BigDecimal oldAmount = amount;
      amount = newAmount;
      firePropertyChange(PROP_AMOUNT, oldAmount,
              newAmount);
      if (null != parent) {
        parent.lineAmountChanged(this, oldAmount);
      }
    }
  }
  private BigDecimal computeAmount() {
    return quantity.multiply(price);
  }
}
```

6. Файл ContactModel.java

```
package invoices;
import java.util.Objects;
// контактная информация —
// используется для представления информации о покупателе
public class ContactModel extends AbstractBean {
  public static final String PROP_NAME = "name";
  public static final String PROP_ADDRESS = "address";
  private String name = "";
  private String address = "";
  public String getName() {
    return name;
  }
  public void setName(String newName) {
    if (!Objects.equals(newName, name)) {
      String oldName = name;
      name = newName;
      firePropertyChange(PROP_NAME, oldName,
              newName);
    }
  }
  public String getAddress() {
    return address;
  }
  public void setAddress(String newAddress) {
    if (!Objects.equals(newAddress, address)) {
      String oldAddress = address;
      address = newAddress;
      firePropertyChange(PROP_ADDRESS, oldAddress,
              newAddress);
```

```
    }
  }
}
```

7. Файл InvoiceProxy.java

```
package invoices;
import java.math.BigDecimal;
import java.util.Date;
// сокращенная информации о счете-фактуре,
// используемая в главном окне
public class InvoiceProxy {
  private final int id;
  private final int version;
  private final String number;
  private final Date date;
  private final String seller;
  private final BigDecimal total;
  private InvoiceProxy(Builder builder) {
    id = builder.id;
    version = builder.version;
    number = builder.number;
    date = builder.date;
    seller = builder.seller;
    total = builder.total;
  }
  public int getId() {
    return id;
  }
  public int getVersion() {
    return version;
  }
  public String getNumber() {
    return number;
  }
  public Date getDate() {
    return date;
  }
  public String getSeller() {
    return seller;
  }
  public BigDecimal getTotal() {
    return total;
  }
  public static Builder createBuilder() {
    return new Builder();
  }
```

```
  public static class Builder {
    private int id;
    private int version;
    private String number;
    private Date date;
    private String seller;
    private BigDecimal total;
    public Builder id(int id) {
      this.id = id;
      return this;
    }
    public Builder version(int version) {
      this.version = version;
      return this;
    }
    public Builder number(String number) {
      this.number = number;
      return this;
    }
    public Builder date(Date date) {
      this.date = (Date) date.clone();
      return this;
    }
    public Builder seller(String seller) {
      this.seller = seller;
      return this;
    }
    public Builder total(BigDecimal total) {
      this.total = total;
      return this;
    }
    public InvoiceProxy build() {
      return new InvoiceProxy(this);
    }
  }
}
```

8. Файл

```
package invoices;
import java.sql.*;
import java.sql.Date;
import java.util.*;
// запросы на языке SQL
public class TransactionScript {
  private TransactionScript() {
  }
```

```
// ищет счета-фактуры за заданный период
public static List<InvoiceProxy> findInvoices(Period p,
        Connection c) throws SQLException {
  List<InvoiceProxy> proxies = new ArrayList<>();
  String sql = "SELECT * FROM invoice"
          + " WHERE idate BETWEEN ? AND ?"
          + " ORDER BY idate";
  try (PreparedStatement ps = c.prepareStatement(sql)) {
    ps.setDate(1, new Date(p.getStartDate().getTime()));
    ps.setDate(2, new Date(p.getEndDate().getTime()));
    ResultSet rs = ps.executeQuery();
    while (rs.next()) {
      proxies.add(invoiceProxyFromRow(rs, ""));
    }
  }
  return proxies;
}
// ищет счет-фактуру по идентификатору
public static InvoiceModel loadInvoice(InvoiceProxy p,
        Connection c) throws SQLException,
        DataException {
  String sql = "SELECT * FROM invoice WHERE id = ?";
  try (PreparedStatement ps = c.prepareStatement(sql)) {
    ps.setInt(1, p.getId());
    ResultSet rs = ps.executeQuery();
    if (!rs.next()) {
      throw new DataException("Not found.");
    }
    InvoiceModel m = fromRow(rs, "", new InvoiceModel());
    m.getLines().addAll(loadLineModels(p.getId(), c));
    return m;
  }
}
// сохраняет новый счет-фактуру
public static void save(InvoiceModel m, Connection c)
        throws SQLException, DataException {
  assert m.getId() <= 0;
  String sql = "INSERT INTO"
          + " invoice(num, idate, sname, saddr)"
          + " VALUES(?, ?, ?, ?)";
  try (PreparedStatement ps = c.prepareStatement(sql,
          PreparedStatement.RETURN_GENERATED_KEYS)) {
    setParameters(ps, 1, m);
    if (1 != ps.executeUpdate()) {
      throw new DataException("Insertion failed.");
    }
    ResultSet rs = ps.getGeneratedKeys();
```

```
    if (!rs.next()) {
      throw new DataException("Insertion failed.");
    }
    int id = rs.getInt(1);
    save(m.getLines(), id, c);
    m.setId(id);
  }
}
// обновляет ранее созданный счет-фактуру
public static void update(InvoiceModel m, Connection c)
        throws SQLException, DataException {
  assert m.getId() > 0;
  String sql = "UPDATE invoice SET num = ?, idate = ?"
          + ", sname = ?, saddr = ?, ver = ver + 1"
          + " WHERE id = ? AND ver = ?";
  try (PreparedStatement ps = c.prepareStatement(sql)) {
    int i = setParameters(ps, 1, m);
    ps.setInt(i++, m.getId());
    ps.setInt(i++, m.getVersion());
    if (1 != ps.executeUpdate()) {
      throw new DataException("Update failed.");
    }
    deleteInvoiceLines(m.getId(), c);
    save(m.getLines(), m.getId(), c);
    m.setVersion(m.getVersion() + 1);
  }
}
// удаляет счет-фактуру
public static void delete(InvoiceProxy p, Connection c)
        throws SQLException, DataException {
  String sql = "DELETE FROM invoice"
          + " WHERE id = ? AND ver = ?";
  try (PreparedStatement ps = c.prepareStatement(sql)) {
    ps.setInt(1, p.getId());
    ps.setInt(2, p.getVersion());
    if (1 != ps.executeUpdate()) {
      throw new DataException("Deletion failed.");
    }
  }
}
private static List<InvoiceLineModel> loadLineModels(
        int id, Connection c)
        throws SQLException, DataException {
  String sql = "SELECT * FROM invoice_line"
          + " WHERE id = ? ORDER BY ord";
  try (PreparedStatement ps = c.prepareStatement(sql)) {
    ps.setInt(1, id);
```

```
    ResultSet rs = ps.executeQuery();
    List<InvoiceLineModel> m = new ArrayList<>();
    while (rs.next()) {
      m.add(fromRow(rs, "", new InvoiceLineModel()));
    }
    return m;
  }
}
private static void save(List<InvoiceLineModel> m,
        int id, Connection c)
        throws SQLException, DataException {
  String sql = "INSERT INTO"
          + " invoice_line(id, ord, prod, qty, price)"
          + " VALUES(?, ?, ?, ?, ?)";
  c.setAutoCommit(false);
  try (PreparedStatement ps = c.prepareStatement(sql)) {
    for (int i = 0; i < m.size(); ++i) {
      ps.setInt(1, id);
      ps.setInt(2, i);
      setParameters(ps, 3, m.get(i));
      ps.addBatch();
    }
    int[] counts = ps.executeBatch();
    for (int i = 0; i < counts.length; ++i) {
      if (1 != counts[i]) {
        throw new DataException("Insertion failed.");
      }
    }
  }
}
private static void deleteInvoiceLines(int id,
        Connection c) throws SQLException {
  String sql = "DELETE FROM invoice_line WHERE id = ?";
  try (PreparedStatement ps = c.prepareStatement(sql)) {
    ps.setInt(1, id);
    ps.executeUpdate();
  }
}
private static InvoiceProxy invoiceProxyFromRow(
        ResultSet rs, String p) throws SQLException {
  return InvoiceProxy.createBuilder()
          .id(rs.getInt(p + "id"))
          .version(rs.getInt(p + "ver"))
          .number(rs.getString(p + "num"))
          .date(rs.getDate(p + "idate"))
          .seller(rs.getString(p + "sname"))
          .total(rs.getBigDecimal(p + "total"))
```

```
            .build();
    }
    private static InvoiceModel fromRow(ResultSet rs,
            String p, InvoiceModel model)
            throws SQLException {
      model.setId(rs.getInt(p + "id"));
      model.setVersion(rs.getInt(p + "ver"));
      model.setNumber(rs.getString(p + "num"));
      model.setDate(rs.getDate(p + "idate"));
      fromRow(rs, "s", model.getSeller());
      return model;
    }
    private static InvoiceLineModel fromRow(ResultSet rs,
            String p, InvoiceLineModel m)
            throws SQLException {
      m.setProduct(rs.getString(p + "prod"));
      m.setQuantity(rs.getBigDecimal(p + "qty"));
      m.setPrice(rs.getBigDecimal(p + "price"));
      return m;
    }
    private static void fromRow(ResultSet rs, String p,
            ContactModel m) throws SQLException {
      m.setName(rs.getString(p + "name"));
      m.setAddress(rs.getString(p + "addr"));
    }
    private static int setParameters(PreparedStatement ps,
            int index, InvoiceModel m) throws SQLException {
      ps.setString(index++, m.getNumber());
      ps.setDate(index++, new Date(m.getDate().getTime()));
      index = setParameters(ps, index, m.getSeller());
      return index;
    }
    private static int setParameters(PreparedStatement ps,
            int i, InvoiceLineModel m) throws SQLException {
      ps.setString(i++, m.getProduct());
      ps.setBigDecimal(i++, m.getQuantity());
      ps.setBigDecimal(i++, m.getPrice());
      return i;
    }
    private static int setParameters(PreparedStatement ps,
            int i, ContactModel m) throws SQLException {
      ps.setString(i++, m.getName());
      ps.setString(i++, m.getAddress());
      return i;
    }
}
```

9. Файл ApplicationService.java

```
package invoices;
// действия, доступные контроллерам
public interface ApplicationService {
  boolean editInvoice(InvoiceModel model);
  Period editPeriod(Period period);
  void reportException(Exception exception);
}
```

10. Файл InvoiceListController.java

```
package invoices;
import java.sql.Connection;
import java.util.*;
import java.sql.SQLException;
import org.jdesktop.observablecollections.*;
// контроллер главного окна
public class InvoiceListController extends AbstractBean {
  public static final String PROP_PERIOD = "period";
  public static final String PROP_COUNT = "count";
  private final DataEnvironment environment;
  private Period period;
  private final ObservableList<InvoiceProxy> proxies;
  private int count;
  private InvoiceProxy selectedProxy;
  private ApplicationService service;
  public InvoiceListController() {
    this(DataEnvironment.current());
  }
  public InvoiceListController(
          DataEnvironment environment) {
    this.environment = environment;
    Date date = new Date();
    period = new Period(date, date);
    this.proxies = ObservableCollections.observableList(
            new ArrayList<InvoiceProxy>());
  }
  public Period getPeriod() {
    return period;
  }
  public void setPeriod(Period newPeriod) {
    if (!Objects.equals(newPeriod, period)) {
      Period oldPeriod = period;
      period = newPeriod;
      firePropertyChange(PROP_PERIOD, oldPeriod,
              newPeriod);
      refresh();
```

```
  }
}
public InvoiceProxy getSelectedProxy() {
  return selectedProxy;
}
public void setSelectedProxy(
        InvoiceProxy newSelectedProxy) {
  selectedProxy = newSelectedProxy;
}
public ObservableList<InvoiceProxy> getProxies() {
  return proxies;
}
public int getCount() {
  return count;
}
public ApplicationService getService() {
  return service;
}
public void setService(ApplicationService newService) {
  service = newService;
}
// вызывается при нажатии на кнопку «Добавить»
public void add() {
  try {
    InvoiceModel model = new InvoiceModel();
    if (!service.editInvoice(model)) {
      return;
    }
    doAdd(model);
    proxies.add(model.createProxy());
  } catch (SQLException | DataException ex) {
    service.reportException(ex);
  }
  setCount(proxies.size());
}
// вызывается при нажатии на кнопку «Редактировать»
public void edit() {
  if (null == selectedProxy) {
    return;
  }
  try {
    InvoiceModel model = doLoad(selectedProxy);
    if (!service.editInvoice(model)) {
      return;
    }
    doUpdate(model);
    InvoiceProxy proxy = model.createProxy();
```

```
      proxies.set(proxies.indexOf(selectedProxy), proxy);
      selectedProxy = proxy;
    } catch (SQLException | DataException ex) {
      service.reportException(ex);
    }
  }
  // вызывается при нажатии на кнопку «Удалить»
  public void delete() {
    if (null == selectedProxy) {
      return;
    }
    try {
      doDelete(selectedProxy);
    } catch (DataException | SQLException ex) {
      service.reportException(ex);
    }
    proxies.remove(selectedProxy);
    setCount(proxies.size());
  }
  // вызывается при нажатии на кнопку «Обновить»
  public void refresh() {
    proxies.clear();
    try {
      proxies.addAll(doFind());
    } catch (SQLException | DataException ex) {
      service.reportException(ex);
    }
    setCount(proxies.size());
  }
  // вызывается при нажатии на кнопку «Период...»
  public void choosePeriod() {
    Period newPeriod = service.editPeriod(period);
    if (null == newPeriod) {
      return;
    }
    setPeriod(newPeriod);
  }
  private void setCount(int newCount) {
    if (newCount != count) {
      int oldCount = count;
      count = newCount;
      firePropertyChange(PROP_COUNT, oldCount, newCount);
    }
  }
  private List<InvoiceProxy> doFind()
          throws SQLException, DataException {
    return environment.withinTransaction(
```

```
            new DataQuery<List<InvoiceProxy>>() {
      @Override
      public List<InvoiceProxy> perform(Connection c)
              throws DataException, SQLException {
        return TransactionScript.findInvoices(period, c);
      }
    });
}
private void doAdd(final InvoiceModel model)
          throws SQLException, DataException {
  environment.withinTransaction(new DataQuery<Void>() {
      @Override
      public Void perform(Connection c)
              throws DataException, SQLException {
        TransactionScript.save(model, c);
        return null;
      }
    });
}
private InvoiceModel doLoad(final InvoiceProxy proxy)
          throws DataException, SQLException {
  return environment.withinTransaction(
            new DataQuery<InvoiceModel>() {
      @Override
      public InvoiceModel perform(Connection c)
              throws DataException, SQLException {
        return TransactionScript.loadInvoice(proxy, c);
      }
    });
}
private void doUpdate(final InvoiceModel model)
          throws SQLException, DataException {
  environment.withinTransaction(new DataQuery<Void>() {
      @Override
      public Void perform(Connection c)
              throws DataException, SQLException {
        TransactionScript.update(model, c);
        return null;
      }
    });
}
private void doDelete(final InvoiceProxy proxy)
          throws DataException, SQLException {
  environment.withinTransaction(new DataQuery<Void>() {
      @Override
      public Void perform(Connection c)
              throws DataException, SQLException {
```

```
        TransactionScript.delete(proxy, c);
        return null;
      }
    });
  }
}
```

11. Файл InvoiceController.java

```
package invoices;
// контроллер диалогового окна редактирования счета-фактуры
public class InvoiceController extends AbstractBean {
  public static final String PROP_MODEL = "model";
  public static final String PROP_SELECTED_LINE_MODEL =
          "selectedLineModel";
  private final InvoiceModel model;
  private InvoiceLineModel selectedLineModel;
  public InvoiceController() {
    this(new InvoiceModel());
  }
  public InvoiceController(InvoiceModel model) {
    this.model = model;
  }
  public InvoiceModel getModel() {
    return model;
  }
  public InvoiceLineModel getSelectedLineModel() {
    return selectedLineModel;
  }
  public void setSelectedLineModel(
          InvoiceLineModel newSelectedLineModel) {
    selectedLineModel = newSelectedLineModel;
  }
  // вызывается при нажатии на кнопку «Добавить»
  public void addLine() {
    getModel().getLines().add(new InvoiceLineModel());
  }
  // вызывается при нажатии на кнопку «Удалить»
  public void deleteLine() {
    if (null != selectedLineModel) {
      getModel().getLines().remove(selectedLineModel);
    }
  }
}
```

12. Файл Period.java

```
package invoices;
```

```
import java.util.*;
// период во времени
public class Period {
  private final Date start;
  private final Date end;
  public Period(Date start, Date end) {
    assert !start.after(end);
    this.start = (Date) start.clone();
    this.end = (Date) end.clone();
  }
  public Date getStartDate() {
    return start;
  }
  public Date getEndDate() {
    return end;
  }
  @Override
  public int hashCode() {
    int hash = 5;
    hash = 71 * hash + Objects.hashCode(start);
    hash = 71 * hash + Objects.hashCode(end);
    return hash;
  }
  @Override
  public boolean equals(Object other) {
    return this == other || null != other
            && other instanceof Period
            && contentEquals((Period) other);
  }
  private boolean contentEquals(Period other) {
    return Objects.equals(start, other.start)
            && Objects.equals(end, other.end);
  }
}
```

13. Файл PeriodController.java

```
package invoices;
import java.util.*;
// контроллер диалогового окна редактирования периода
public class PeriodController extends AbstractBean {
  public static final String PROP_START_DATE = "startDate";
  public static final String PROP_END_DATE = "endDate";
  private Date startDate;
  private Date endDate;
  public PeriodController() {
    startDate = new Date();
```

```
    endDate = new Date();
  }
  public Period toPeriod() {
    return new Period(startDate, endDate);
  }
  public void applyPeriod(Period period) {
    setStartDate(period.getStartDate());
    setEndDate(period.getEndDate());
  }
  public Date getStartDate() {
    return (Date) startDate.clone();
  }
  public void setStartDate(Date newStartDate) {
    if (!Objects.equals(newStartDate, startDate)) {
      Date oldStartDate = startDate;
      startDate = newStartDate;
      firePropertyChange(PROP_START_DATE, oldStartDate,
              newStartDate);
    }
  }
  public Date getEndDate() {
    return (Date) endDate.clone();
  }
  public void setEndDate(Date newEndDate) {
    if (!Objects.equals(newEndDate, endDate)) {
      Date oldEndDate = endDate;
      endDate = newEndDate;
      firePropertyChange(PROP_END_DATE, oldEndDate,
              newEndDate);
    }
  }
  public void set(Period period) {
    setStartDate(period.getStartDate());
    setEndDate(period.getEndDate());
  }
}
```

5.3. Интерфейс пользователя

Благодаря тому, что контроллеры и модели строго следуют спецификации JavaBeans, а для хранения совокупностей объектов используют наблюдаемые (observable) коллекции, становится возможно построение пользовательского интерфейса на основе привязки компонентов (beans binding). При использовании привязки компонентов связи между элементами управления и свойствами

объектов (моделей и контроллеров) задается в редакторе, а написание кода в основном сводится к вызовам соответствующих методов контроллера в обработчиках нажатия на кнопку.

Внешний вид интерфейса пользователя представлен на рис. 1 и 2

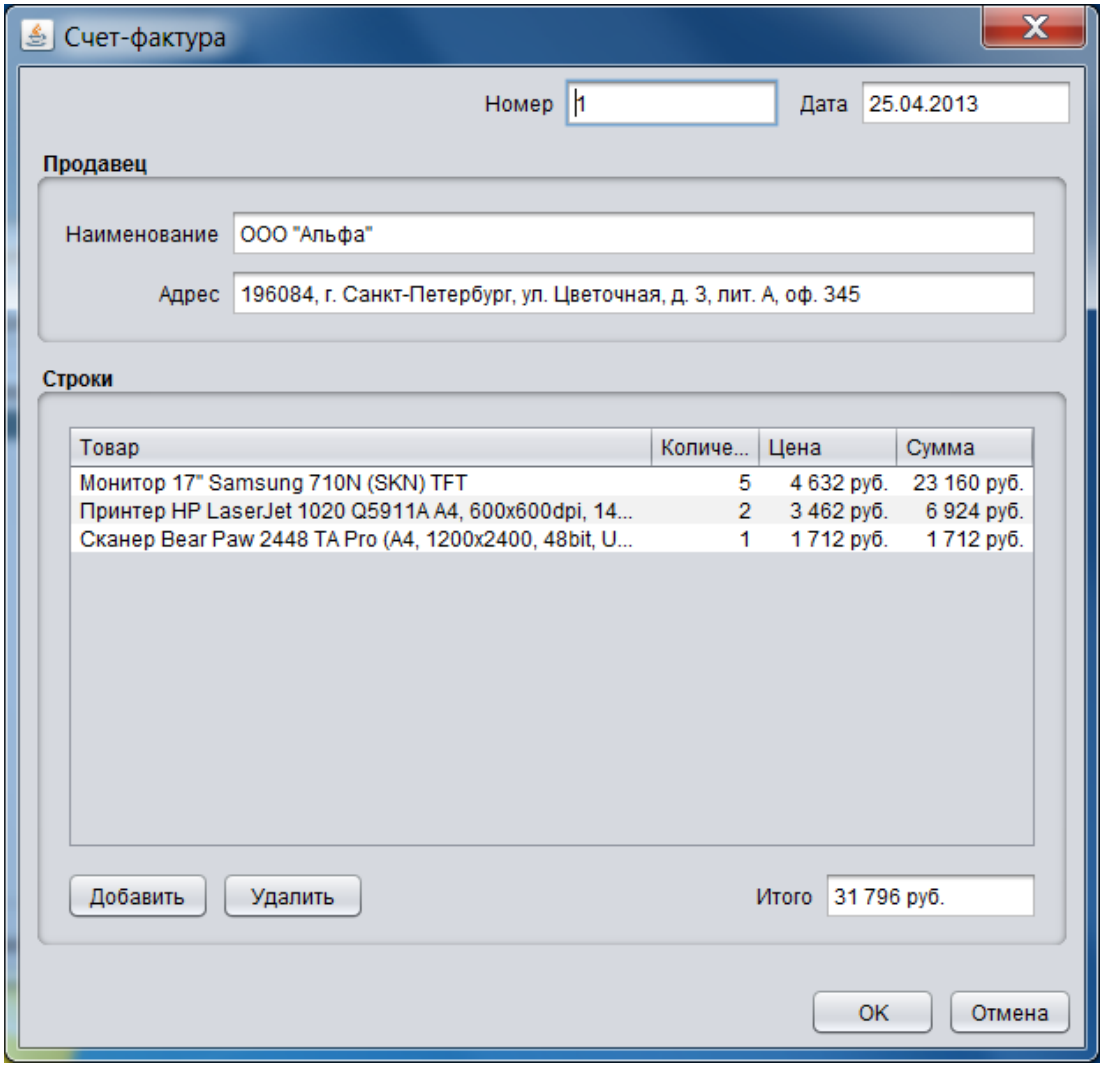

Рис. 1. Диалоговое окно редактирования счета-фактуры

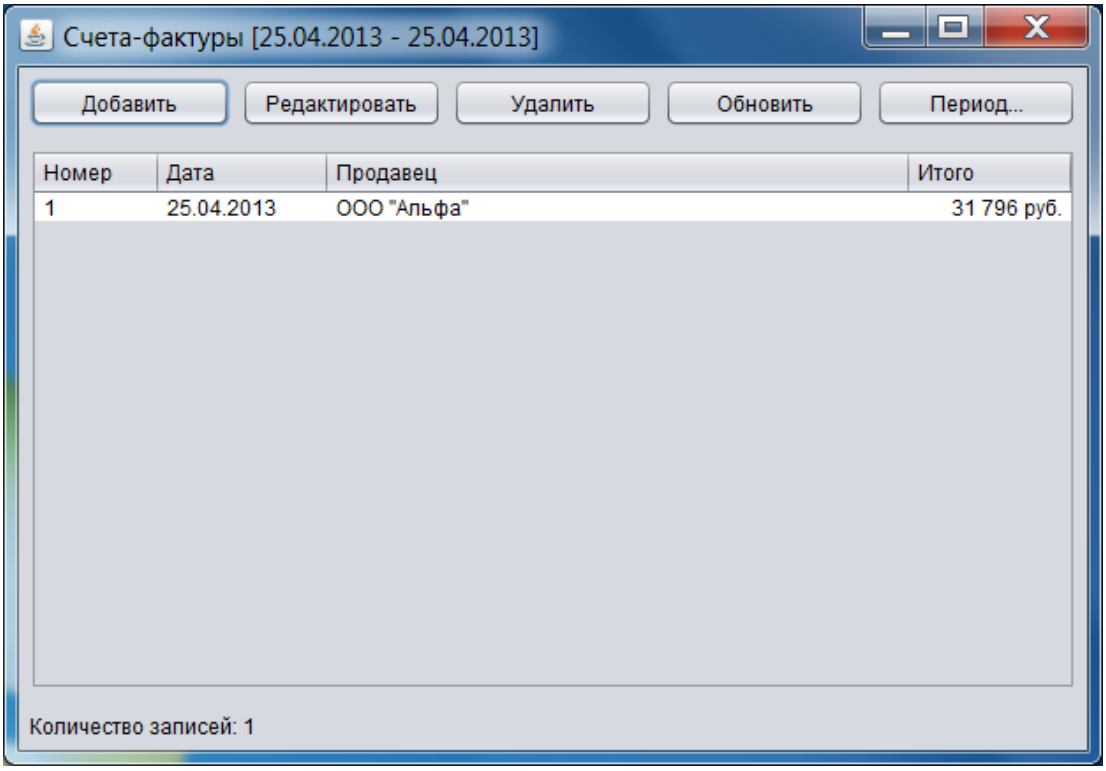

Рис. 2. Главное окно

6. Контрольные вопросы

1. Что такое драйвер базы данных?
2. Для чего используется менеджер драйверов?
3. Для чего необходима регистрация драйвера и как она выполняется?
4. Что называется строкой соединения с базой данных?
5. Как задать значения дополнительных параметров соединения?
6. Как выполнить запрос к базе данных?
7. Что называется пакетным режимом выполнения запросов?
8. Что такое результирующее множество?
9. Как нумеруются столбцы результирующего множества?
10. Что называют курсором?
11. Как извлечь значение из результирующего множества?
12. Как проверить, было ли извлеченное значение пустым?
13. Что называют параметром запроса?
14. Как задать значение параметра запроса?
15. Как задать, что значение параметра является пустым?
16. Как выполнить запрос, содержащий параметры?

17. Как использовать пакетный режим с запросам, содержащим параметры?
18. Как извлечь значения ключей, сгенерированных в результате выполнения запроса?
19. Как выполнить хранимую процедуру?
20. Что такое escape-последовательность?
21. Какой курсор называют прокручиваемым?
22. Какие методы перемещения курсора поддерживаются?
23. Какой курсор называют обновляемым?
24. Как с помощью обновляемого курсора можно вставлять, изменять и обновлять строки в таблице базы данных.
25. Какие режимы подтверждения транзакций поддерживаются?
26. Как задать параметры транзакции?
27. Как работать с большими объектами с использованием стандартных потоков ввода-вывода?
28. Как работать с большими объектами с использованием специальных интерфейсов определенных в JDBC?

7. Варианты заданий

1. Приложение для учета заказов.
2. Приложение для учета проданных билетов для кинотеатра.
3. Приложение для учета проданных билетов для авиакомпании.
4. Приложение для учета проданных туристических путевок.
5. Приложение для учета приема пациентов для медицинского учреждения.
6. Приложение для учета задержаний для отделения полиции.
7. Электронная касса для розничной торговли.
8. Электронная бухгалтерия (синтетический учет).
9. Электронный отдел кадров.
10. Электронный деканат (расписание занятий).
11. Электронный деканат (фактически проведенные занятия).
12. Электронный деканат (расписание экзаменов и зачетов).
13. Электронный деканат (ведомости и экзаменационные листки).

8. Список литературы

1. Дейтел Х. М., Дейтел П. Дж., Сантри С. И. Технологии программирования на Java 2. Книга 2. Распределенные приложения — М.: ООО «Бином-Пресс», 2003. — 464 с.
2. Библиотека профессионала. Java 2. Том 2. Тонкости программирования. — М.: Издательский дом «Вильямс», 2002. — 1120 с.
3. Соломон М., Мориссо-Леруа Н., Басу Дж. Oracle. Программирование на языке Java. — М.: Издательство «Лори», 2010. — 512 с.
4. Хабибуллин И. Создание распределенных приложений на Java 2. — СПб.: БХВ-Петербург, 2002. — 704 с.