ЛАБОРАТОРНАЯ РАБОТА №6.
ПО ПРЕДМЕТУ
ПО ПРЕДМЕТУ «ПРОГРАММИРОВАНИЕ СЕТЕВЫХ ПРИЛОЖЕНИЙ»
ТЕМА: «РАБОТА С БАЗАМИ ДАННЫХ В JAVA»

1-40 05 01 «Информационные системы и технологии (в бизнес-менеджменте и промышленной безопасности)»

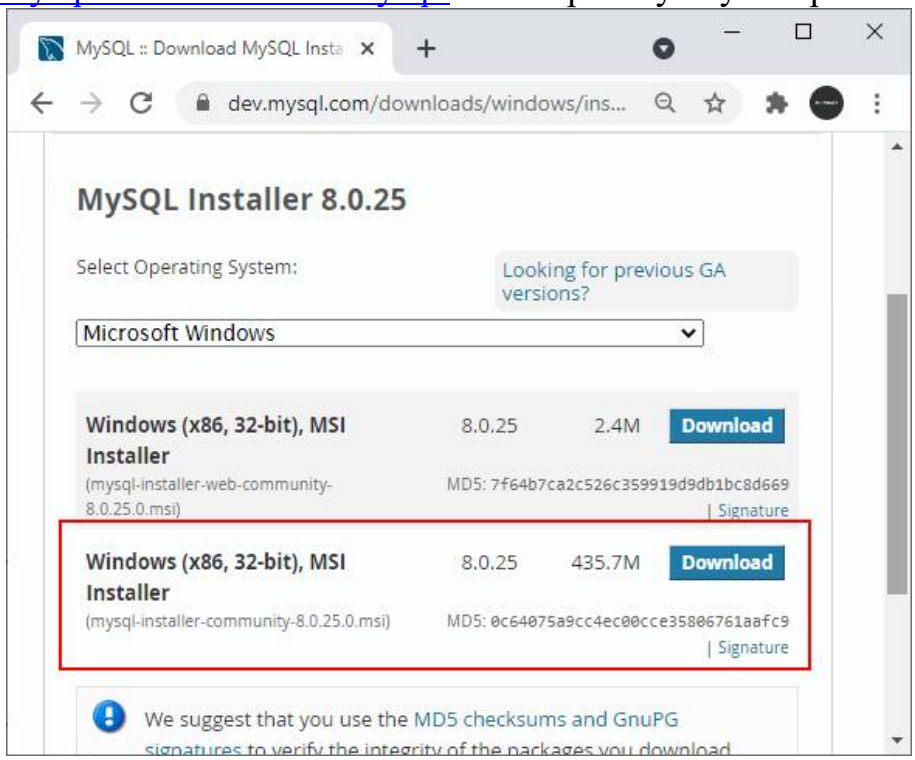
ЦЕЛЬ: освоение навыков использования интерфейса JDBC для доступа к реляционным базам данных из программ, разрабатываемых с использованием платформы Java.

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 драйверы для наиболее распространенных СУБД, таких как *MySQL* и *PostgreSQL*.

РАБОТА С БД И ИСПОЛЬЗОВАНИЕМ JAVA

Скачать СУБД можно с официальный сайта https://www.mysql.com/.

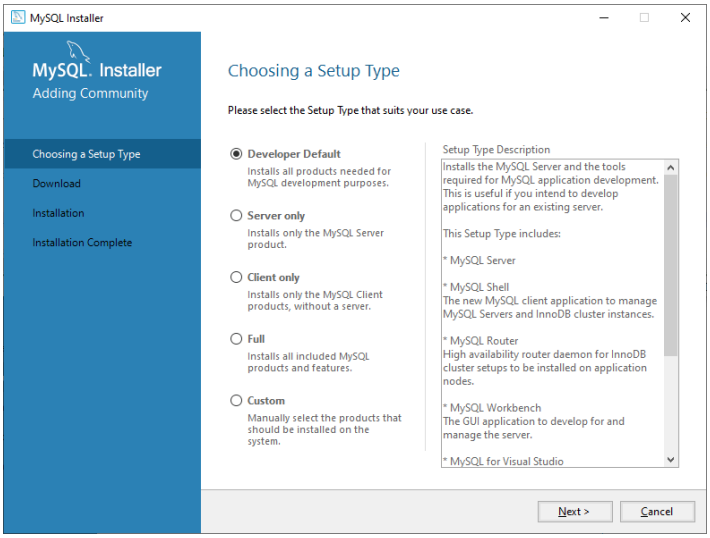
Для установки MySQL загрузим дистрибутив по адресу https://dev.mysql.com/downloads/mysql/ и выберем нужную версию.



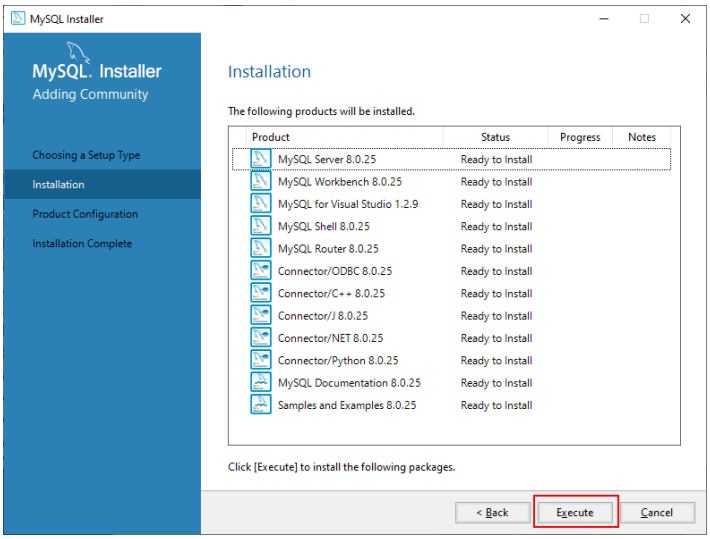
Выберите необходимый вариант загрузки. Дальше может быть предложено авторизироваться в системе с помощью учетной записи *Oracle*. Можно пропустить этот этап и нажать на ссылку "No thanks, just start my download.".

В начале установки будет предложено выбрать тип установки. Выберем тип *Developer Default*, которого будет достаточно для базовых нужд, и нажмем на кнопку Next.

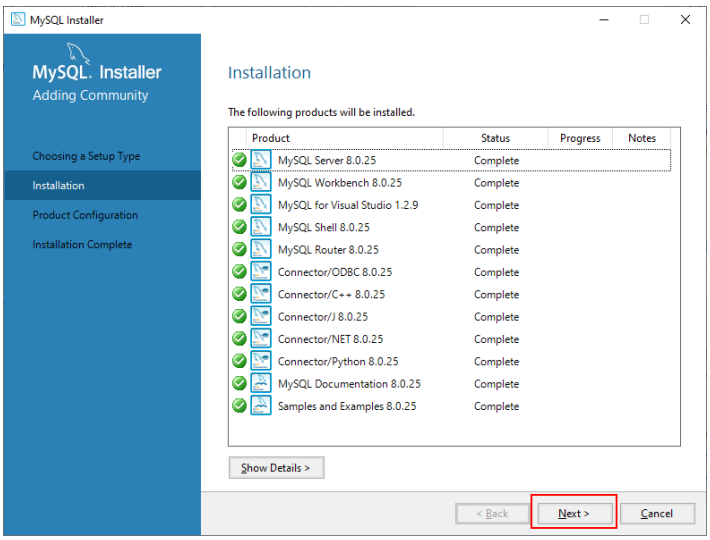
Сначала будет предложено выбрать тип установки. Выберем тип **Developer Default**, которого хватит для базовых нужд, и нажмем на кнопку Next.



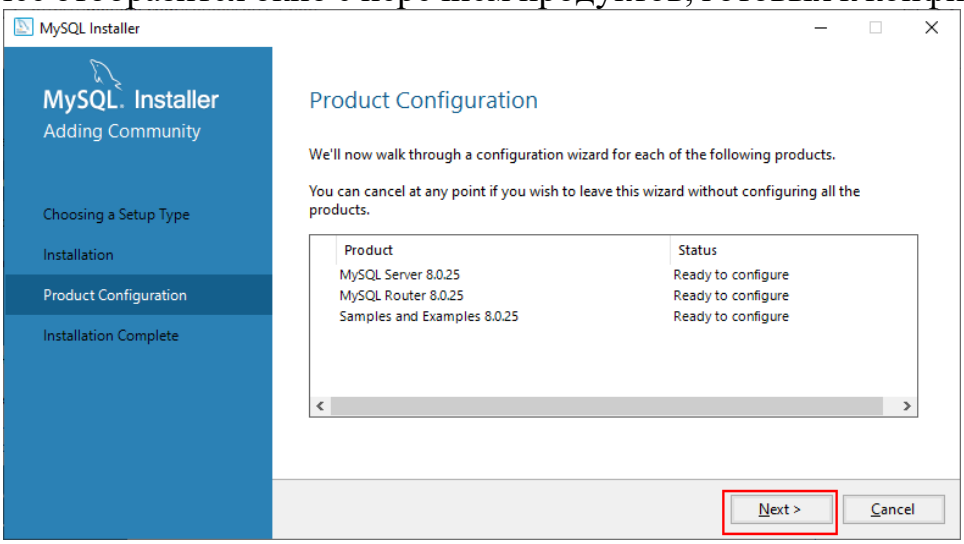
Затем на этапе установки инсталлятор отобразит весь список устанавливаемых компонентов.



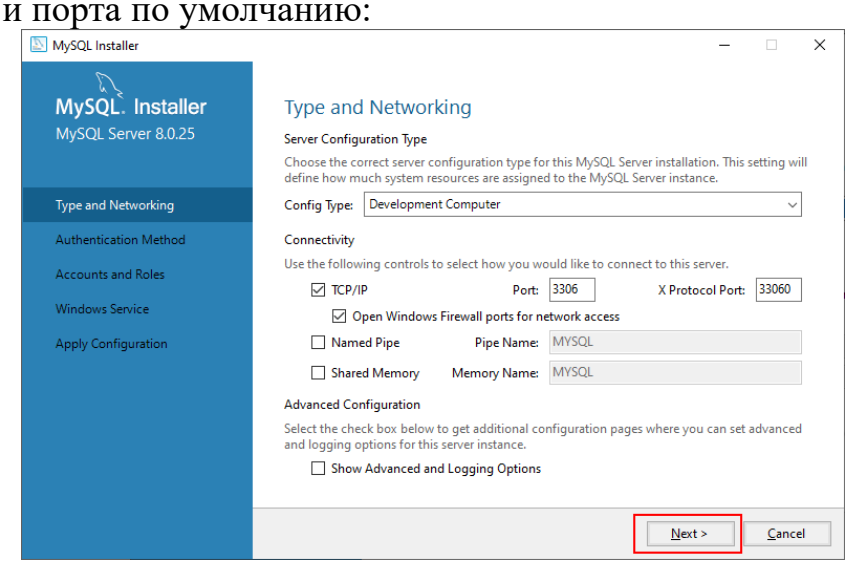
Чтобы выполнить установку всех компонентов, нажмем кнопку Execute. После того, как все компоненты будут установлены, нажмем кнопку Next.



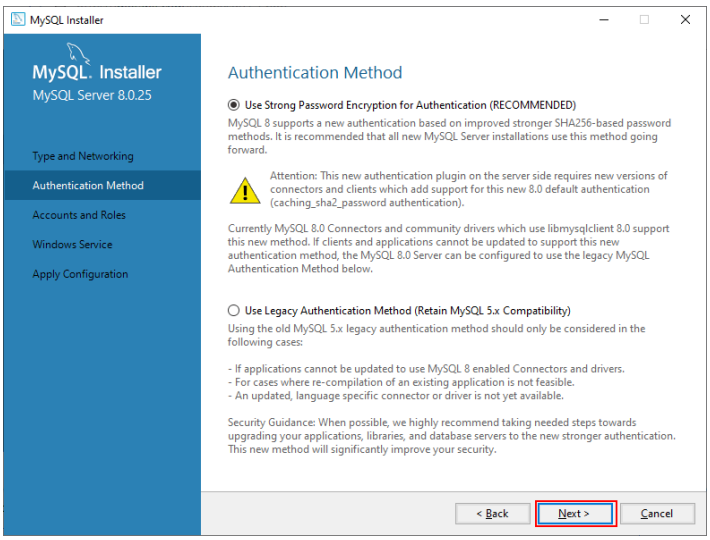
Далее отобразится окно с перечнем продуктов, готовых к конфигурации.



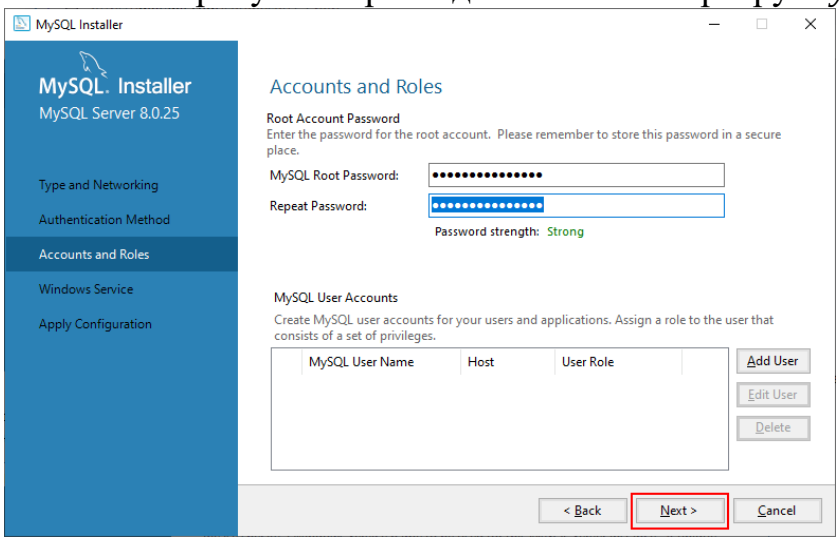
Нажмем на кнопку Next и далее будет предложено установить ряд конфигурационных настроек сервера MySQL. В частности для подключения будет применяться протокол TCP/IP и порт 3306. Оставим все эти настройки соединения и порта по умолчанию:



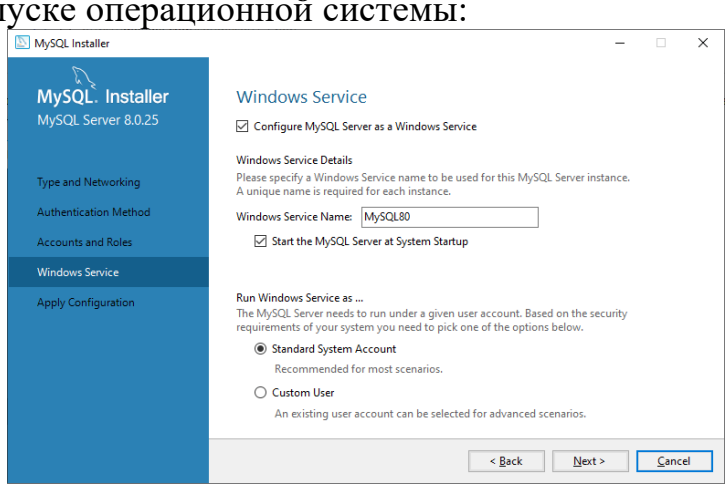
На следующем шаге будет предложено установить метод аутентификации. Оставим настройки по умолчанию:



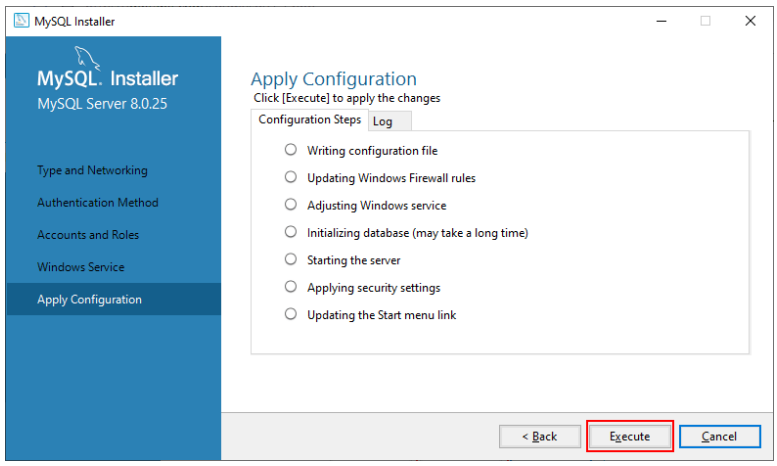
На следующем окне программы установки укажем пароль, и запомним его, так как он потом потребуется при подключении к серверу *MySQL*:



Следующий набор конфигураций, который также оставим по умолчанию, указывает, что сервер будет запускаться в качестве службы *Windows* при запуске операционной системы:

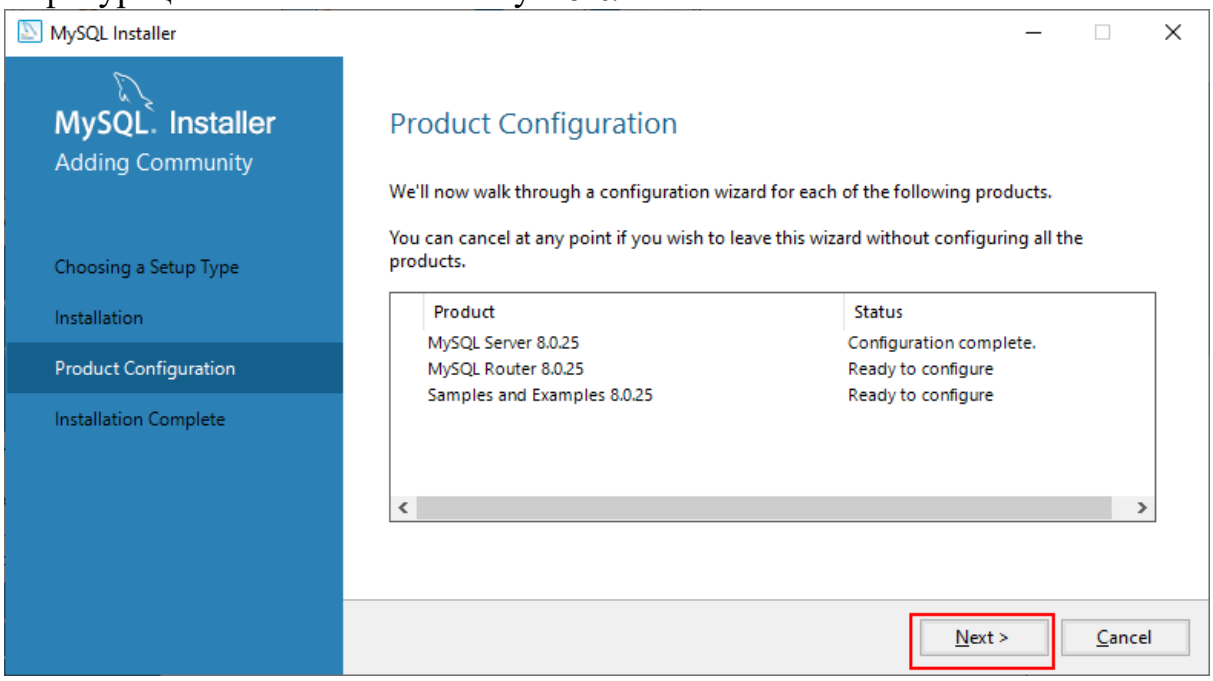


На следующем экране необходимо применить все ранее установленные конфигурационные настройки, нажав на кнопку *Execute*:

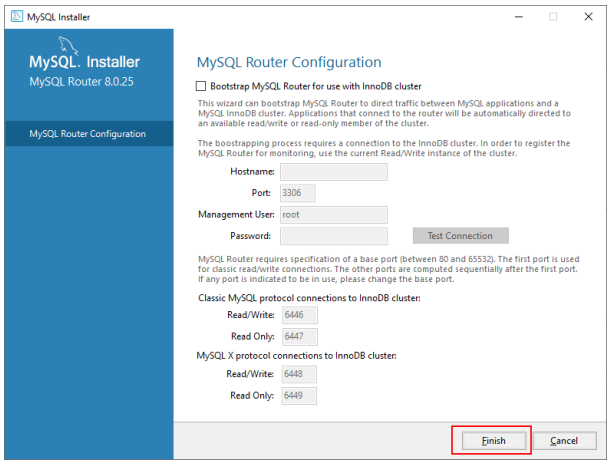


После применения конфигурационных настроек сервер *MySQL* будет полностью установлен и сконфигурирован, нажмем на кнопку *Finish*.

Далее опять отобразится окно с перечнем продуктов, готовых к конфигурации. Нажмем на кнопку *Next*.

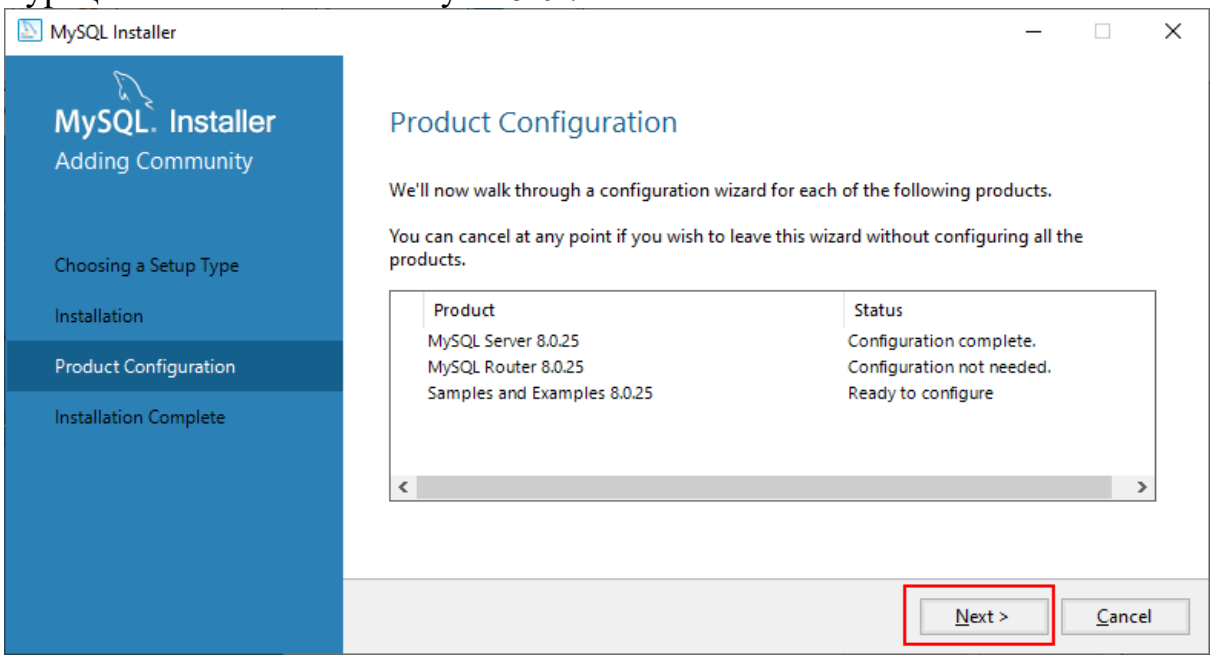


Будет предложено установить конфигурацию для второго продукта *MySQL Router*.

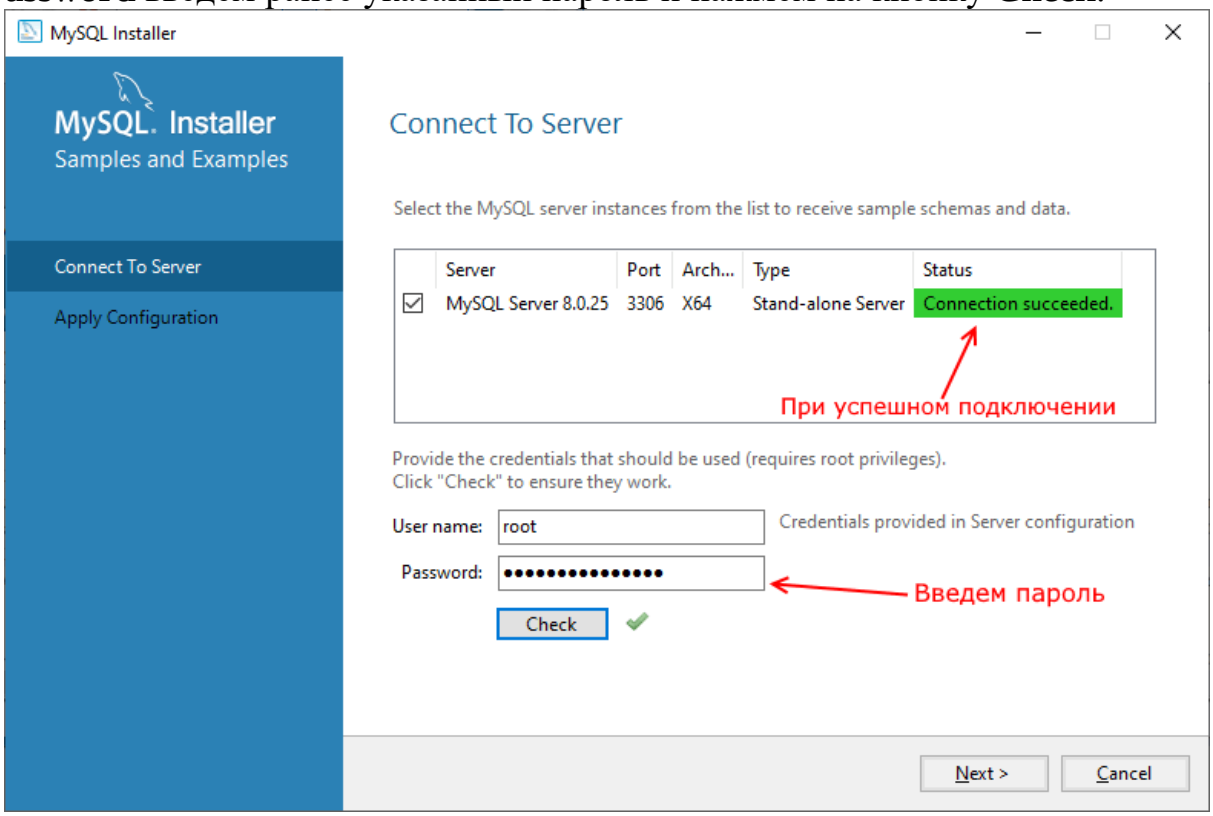


Ничего не будем менять, оставив все настройки по умолчанию, и нажмем на кнопку "*Finish*".

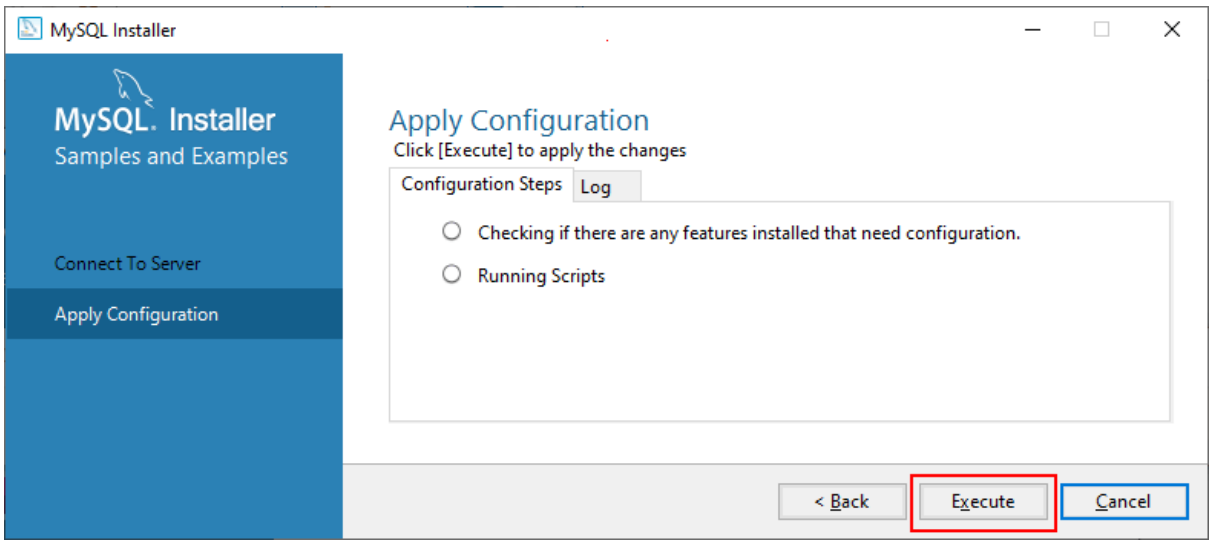
Далее опять отобразится окно с перечнем продуктов, готовых к конфигурации. Нажмем на кнопку "*Next*".



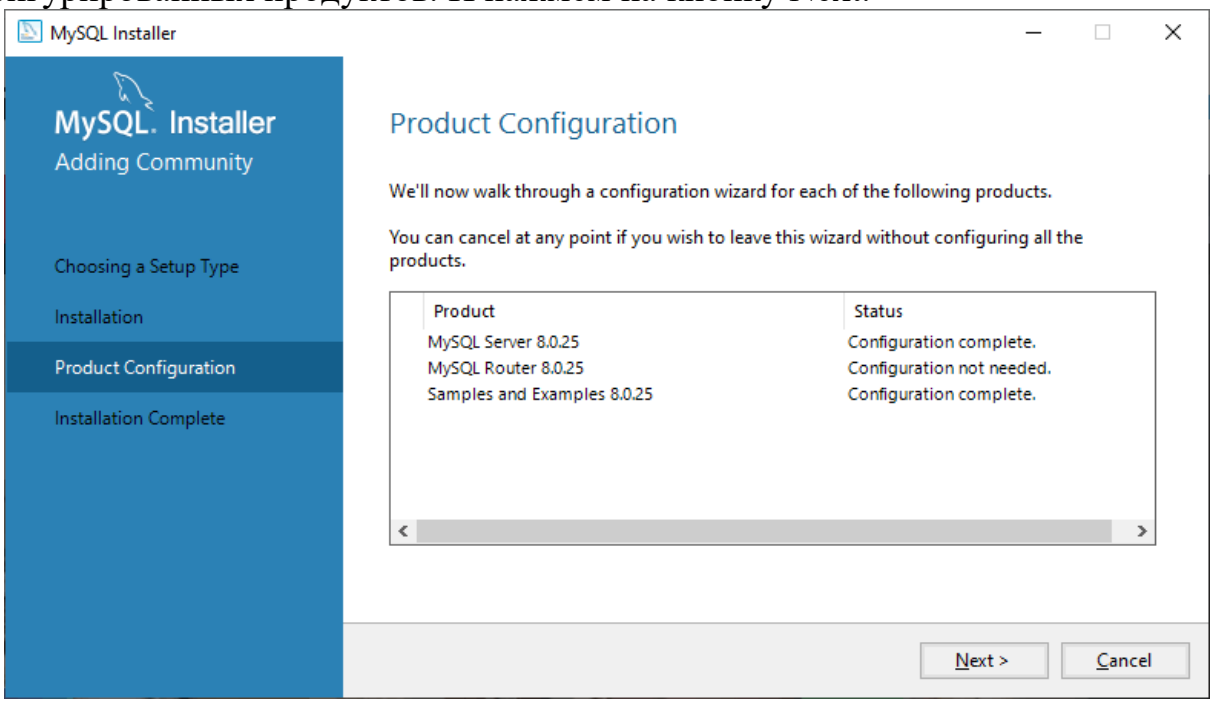
Далее будет предложено установить конфигурацию для третьего продукта ***Samples and Examples*** (Примеры работы с MySQL). В частности, надо будет указать экземпляр сервера *MySQL* для получения примеров для работы с *MySql*. Установленный экземпляр будет автоматически отмечен в списке. Кроме того, предлагает протестировать подключения. В поле **Password** введем ранее указанный пароль и нажмем на кнопку **Check**:



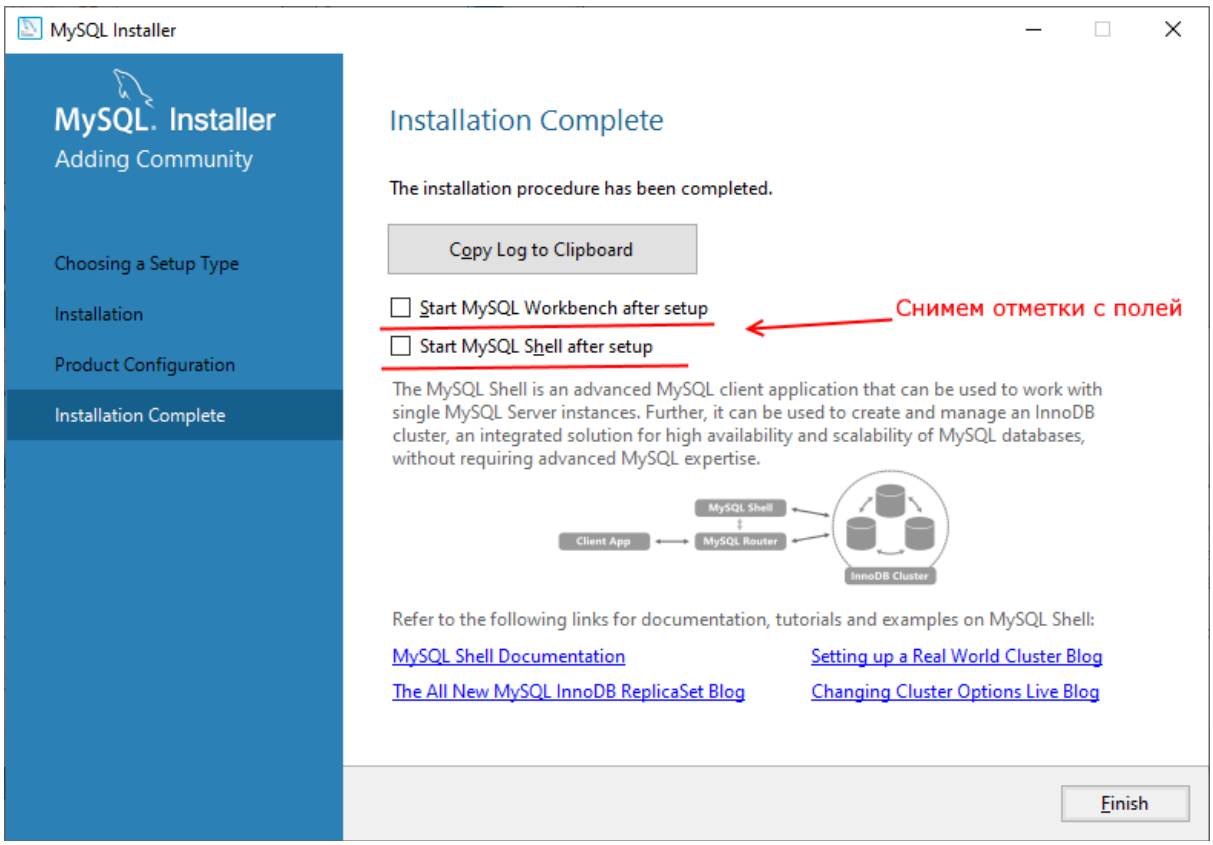
Пи успешном подключении к *MySQL* отобразится выделенная зеленым цветом надпись **Connection succeeded**. Нажмем на кнопку Next. И на последнем окне необходимо будет применить конфигурацию.



Далее мы опять увидим окно с перечнем установленных и сконфигурированных продуктов. И нажмем на кнопку Next.



На последнем экране мы увидим два отмеченных поля: **Start MySQL Workbench after setup** и **Start MySQL Shell after setup**. Эти поля позволяют запустить графический и консольный клиенты для управления сервером MySQL. Снимем отметки с этих полей, поскольку пока мы не собираемся запускать соответствующие программы.



И нажмем на кнопку Finish. MySQL полностью установлен, сконфигурирован и запущен.

УСТАНОВЛЕНИЕ СОЕДИНЕНИЯ

Если приложению необходимо выполнить запрос к базе данных, то первым его шагом должно быть установление соединения с базой данных. Это можно сделать следующим образом:

```
String url = "jdbc:mysql://localhost/tutorial", user = "root",
password = "qwerty";
try (Connection c = DriverManager.getConnection(url, user,
password)) {
    // Использовать соединение с базой данных
} catch (SQLException e) {
    throw new RuntimeException(e);
}
```

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. 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 соединения. 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.

Другим способом создания соединения 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 источник данных самостоятельно, а полагаться на то, что упомянутый контейнер сам в нужный момент инъецирует (*inject*) полностью готовый к использованию источник данных в приложение.

Иногда для успешной установки соединения надо указать дополнительные параметры помимо имени пользователя и пароля. Для этого можно использовать другую (более общую) форму метода *getConnection()*:

```
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 connection = DriverManager.getConnection(url,
p)) {
    // Использовать соединение с базой данных
} catch (SQLException e) {
    throw new RuntimeException(e);
}
```

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 *IDE* для этого достаточно 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 разработки и не был создан 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 системном свойстве *jdbc.drivers*. 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*. Этот метод

при необходимости загружает и инициализирует указанный класс. Обязательной частью инициализации класса драйвера является его регистрация в менеджере драйверов.

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 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. 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).

ВЫПОЛНЕНИЕ ЗАПРОСОВ

Соединение не может использоваться непосредственно для выполнения запросов. Вместо этого для выполнения запросов прикладная программа должна создать с использованием соединения оператор.

```
try(Connection connection = DriverManager.getConnection(url,p)){
    // Использовать соединение с базой данных
    // в переменной connection хранится 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 = connection.createStatement()) {
        s.execute(sql);
    }
} catch (SQLException e) {
    throw new RuntimeException(e);
}
```

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 необходимое число раз вызвать метод *execute()*, либо использовать пакетный режим выполнения.

```
// 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.

РЕЗУЛЬТИРУЮЩИЕ МНОЖЕСТВО

Если запрос возвращает множество строк, то для их выборки прикладная программа должна использовать специальный объект, называемый в *JDBC* результирующим множеством. Множество строк не является множеством (*set*) в строго математическом смысле, это — последовательность (*sequence*). Дело в том, что, во-первых, множество строк может содержать повторяющиеся строки а, во-вторых, порядок строк в множестве строк не обязательно произвольный. 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()) {
        int id = rs.getInt("st_id");
        String name = rs.getString("st_name");
        // обработать значения id и name
    }
}
```

Для выполнения запроса и получения возвращаемого им результирующего множества используется метод *executeQuery()*. 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 ссылочных типов. Чтобы отличить нулевое и ложное значение от пустого значения можно использовать метод *wasNull()*; этот метод не имеет параметров и возвращает значение *true*, если последнее по времени получаемое значение было пустым. 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()*.

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

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

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 connection хранится ранее установленное
соединение
String sql = "INSERT INTO student(st_name) VALUES(?)";
try (PreparedStatement ps = connection.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);
        // обработать полученный id
    }
}
```

При использовании подготовленного оператора элементы пакета различаются между собой только значениями параметров. Кроме пакетного режима этот пример показывает, как выбрать сгенерированные ключи. 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*).

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.

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 connection хранится ранее созданное соединение
int bestStudentId;
String sql = "{call get_best_student_id(?)}";
try (CallableStatement cs = connection.prepareCall(sql)) {
```

```
    cs.registerOutParameter("st_id", Types.INTEGER);
    cs.execute();
    bestStudentId = cs.getInt("st_id");
}
```

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). **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.

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 connection хранится ранее установленное
соединение
String sql = "SELECT * FROM student ORDER BY st_name";
try (Statement s =
connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_READ_ONLY)) {
  try  (ResultSet rs = s.executeQuery(sql)){// Попытаться
выбрать строки с 11 по 20 try
        if (rs.absolute(11)) {
            for (int i = 0; i < 10; ++i) {
                // обработать выбранную строку
                if (!rs.next()) {
                    break;
                }
            }
        }
    }
}
```

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.

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).

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 connection хранится ранее установленное
соединение
String sql = "SELECT * FROM student ORDER BY st_name" + " FOR
UPDATE";
try (Statement s =
connection.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();
                }
            }
        }
}
```

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();
    }
}
```

Вставка новых строк 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).

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 connection хранится ранее созданное соединение
    try (PreparedStatement ps =
connection.prepareStatement(sql)) {
        ps.setInt(1, 1001);
        ps.setString(2, "О чем-то");
        ps.setCharacterStream(3, r);
        ps.executeUpdate();
    }
} catch (FileNotFoundException e) {
    throw new RuntimeException(e);
} catch (IOException e) {
    throw new RuntimeException(e);
}
```

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.

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

Третий способ работы с большими объектами — это использование объектов, реализующих интерфейсы *Clob* и *Blob* языка программирования *Java*. Как это сделать демонстрирует следующий пример.

```
// в переменной rs хранится результирующее множество(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)));
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
} finally {
    clob.free();
}
```

Для получения 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.

```
String sql = "INSERT INTO article(a_id,a_title,a_content)" + "
VALUES(?, ?, ?)";
try (PreparedStatement ps = connection.prepareStatement(sql)) {
    ps.setInt(1, 2001);
    ps.setString(2, "Некоторое название");
    Clob clob = connection.createClob();
    try {
        try (Writer w = clob.setCharacterStream(1)) {
            // используя w записать необходимые данные
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
        ps.setClob(3, clob);
        ps.executeUpdate();
    } finally {
        clob.free();
    }
}
```

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 необходимости 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 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()*.

Ниже приведена последовательность действий для работы с БД приложений, написанных с использованием платформы *Java*.

1. Загрузить класс(ы), реализующие необходимые драйверы

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

2. Установить соединение с БД, используя загруженный драйвер

```
DriverManager.getConnection("jdbc:mysql://localhost:3306/univers
ity?characterEncoding=UTF-
8&useSSL=false&zeroDateTimeBehavior=CONVERT_TO_NULL&serverTimezo
ne=GMT&allowPublicKeyRetrieval=true")
```

3. Создать объект(ы) для исполнения SQL-команд

```
connection.createStatement();
```

4. Выполнить необходимые SQL-команды

```
stmt.executeUpdate("Delete From MyTable Where Id=1");
stmt.executeQuery("Select * From MyTable");
```

5. Обработать полученные таблицы

```
result.getString("fieldName");
```

6. Закрыть открытые соединения

```
connection.close();
```

Рассмотрим создание приложения на *Java*, реализующее двухзвенную архитектуру (приложение – база данных). Приложение позволяет добавлять информацию в базу данных.

```
//Пример №1. Организация взаимодействия приложения двухзвенной
архитектуры.
import javax.swing.*;
import javax.swing.event.ListSelectionEvent;
import javax.swing.event.ListSelectionListener;
import javax.swing.table.DefaultTableModel;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;
class DBClass extends JFrame {
    JTable table1;
    JScrollPane sp;
    static String nazv, cena, naim;
    static int massa;
    Statement sq;
    ListSelectionModel lsm;
    DefaultTableModel dtm;
    String vibor_naim;
    Connection db;
    String colheads[] = {"Наименование", "Название", "Цена",
"Масса"};
    static Object dataConditer[][];
    JTextField tf1, tf2, tf3, tf4;
    void showData() {
        try {
            String sq_str = "SELECT * FROM Assortiment";
            ResultSet rs = sq.executeQuery(sq_str);
            while (rs.next()) {
                nazv = rs.getString("Nazvanie");
                cena = rs.getString("Cena");
                naim = rs.getString("Naimenovanie");
                massa = rs.getShort("Massa");
                Object addingData[] = {naim, nazv, cena, massa};
                dtm.addRow(addingData);
                System.out.println(naim + nazv + massa + cena);
            }
        } catch (Exception e) {
        }
    }
    public DBClass(String Title) {
        super(Title);
        setLayout(null);
        dtm = new DefaultTableModel(dataConditer, colheads);
        table1 = new JTable(dtm);
        JScrollPane sp = new JScrollPane(table1,
```

```
ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED,
ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED);
        sp.setBounds(50, 5, 350, 250);
        add(sp);
        JButton btn1 = new JButton("Добавить");
        btn1.setBounds(100, 370, 100, 30);
        add(btn1);
        btn1.addActionListener(new ActionListenerClass1());
        tf1 = new JTextField("0");
        tf1.setBounds(55, 280, 80, 30);
        add(tf1);
        tf2 = new JTextField("0");
        tf2.setBounds(140, 280, 80, 30);
        add(tf2);
        tf3 = new JTextField("0");
        tf3.setBounds(225, 280, 80, 30);
        add(tf3);
        tf4 = new JTextField("0");
        tf4.setBounds(310, 280, 80, 30);
        add(tf4);
        lsm = table1.getSelectionModel();
        lsm.addListSelectionListener(new
SelectionListenerClass());
        this.setSize(500, 450);
        this.setVisible(true);
        try {
            Class.forName("com.mysql.jdbc.Driver");//получение
класса драйвера
            db =
DriverManager.getConnection("jdbc:mysql://localhost:3306/condite
r", "root", "");
            sq = db.createStatement();
            String sq_str = "SELECT * FROM Assortiment";
            ResultSet rs = sq.executeQuery(sq_str);
            while (rs.next()) {
                nazv = rs.getString("Nazvanie");
                cena = rs.getString("Cena");
                naim = rs.getString("Naimenovanie");
                massa = rs.getShort("Massa");
                Object addingData[] = {naim, nazv, cena, massa};
                dtm.addRow(addingData);
            }
        } catch (Exception e) {
        }
    }
    class SelectionListenerClass implements
ListSelectionListener {
        public void valueChanged(ListSelectionEvent lse) {
            Object obj =
table1.getValueAt(table1.getSelectedRow(),
table1.getSelectedColumn());

tf1.setText(table1.getValueAt(table1.getSelectedRow(),
```

```
0).toString());

tf2.setText(table1.getValueAt(table1.getSelectedRow(),
1).toString());

tf3.setText(table1.getValueAt(table1.getSelectedRow(),
2).toString());

tf4.setText(table1.getValueAt(table1.getSelectedRow(),
3).toString());
            vibor_naim = tf1.getText();
        }    }
    class ActionListenerClass1 implements ActionListener
{//кнопка Добавить
        public void actionPerformed(ActionEvent ae) {
            try {
                naim = tf1.getText();
                nazv = tf2.getText();
                cena = tf3.getText();
                massa = Integer.parseInt(tf4.getText());
                String sq_str_insert = "INSERT INTO Assortiment
VALUES ('" + naim + "','" + nazv + "','" +
                        cena + "'," + massa + ")";
                int rs_update2 =
sq.executeUpdate(sq_str_insert);
                Object addingData[] = {naim, nazv, massa, cena};
                dtm.addRow(addingData);
            } catch (Exception e) {
            }
        }
    }
    public static void main(String args[]) {
// объявление переменных
        DBClass classObj = new DBClass("Frame");
    }
}
```

Следующий пример демонстрирует работу с БД в трехзвенной архитектуре. Вся информация хранится в таблице базы данных. Клиент взаимодействует с сервером по протоколу TCP. Клиент посылает запрос на сервер на добавление данных в таблицу. Приложение сервера подключается к базе, реализует SQL запрос на добавление. Обновленные данные таблицы посылаются сервером клиенту. Клиент видит результат добавления данных.

```
//Пример 2. Реализации трёхзвенной архитектуры. Код клиентского
приложения
import javax.swing.*;
import javax.swing.table.DefaultTableModel;
import javax.swing.table.TableModel;
import javax.swing.table.TableRowSorter;
import java.awt.*;
import java.awt.event.ActionEvent;
```

```
import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.InetAddress;
import java.net.Socket;
import java.sql.Connection;
import java.sql.Statement;

public class Client extends JFrame{
    Socket sock;
    InputStream is;
    OutputStream os;
    String message;
    int a=0;
    JTable table1;JScrollPane sp;
    static String name, type, firma,price,date;
    Statement sq;
    DefaultTableModel DTM;
    String vibor_naim;
    Connection db;
    String colheads[]={"Наименование", "Тип",
"Фирма","Цена","Дата"};
    static Object dataConditer[][];
    byte clientMessage[]=new byte[5000];
    byte bytemessage[] = new byte[5000];
    byte bytemessage2[] = new byte[100];
    JTextField tf1,tf2,tf3,tf4,tf5;
    public Client(String Title){
        super(Title);
        setLayout(null);
        DTM=new DefaultTableModel(dataConditer,colheads);
        table1=new JTable(DTM);
        RowSorter<TableModel> sorter=new
TableRowSorter<TableModel>(DTM);
        table1.setRowSorter(sorter);
        table1.setBackground(Color.getHSBColor(159, 216, 234));
        JScrollPane sp=new
JScrollPane(table1,ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEE
DED,ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED);
        sp.setBounds(50,5,350,250);
        add(sp);
        JButton btn1=new JButton("Добавить");
        JButton btn2=new JButton("Отключить");
        btn1.setBounds(84,320,125,30);
        add(btn1);
        btn1.addActionListener(new ActionListenerClass1());
        btn2.setBounds(239,320,125,30);
        btn1.setBackground(Color.getHSBColor(72,150,180));
        btn2.setBackground(Color.getHSBColor(72,150,180));
        add(btn2);
        btn2.addActionListener(new ActionListenerClass2());
```

```
        tf1=new JTextField("0");
        tf1.setBounds(50,280,68,25);
        add(tf1);
        tf2=new JTextField("0");
        tf2.setBounds(120,280,68,25);
        add(tf2);
        tf3=new JTextField("0");
        tf3.setBounds(190,280,68,25 );
        add(tf3);
        tf4=new JTextField("0");
        tf4.setBounds(260,280,68,25 );
        add(tf4);
        tf5=new JTextField("0");
        tf5.setBounds(330,280,68,25 );
        add(tf5);
        this.setSize(500,450);
        this.setVisible(true);
        addWindowListener(new WindowClose());
        try{
            sock=new
Socket(InetAddress.getByName("localhost"),1024);
            is=sock.getInputStream();
            os=sock.getOutputStream();
            is.read(clientMessage);
            String tempString=new
String(clientMessage,0,clientMessage.length);
            for (int i=0; i<tempString.length(); i++)
            {
                if (tempString.charAt(i) == '/')
                    a++;
            }
            String arrStr[] = tempString.split("/");
            for (int r=0; r<=a;r=r+5)
            {
                Object addingData[]=
{arrStr[r],arrStr[r+1],arrStr[r+2],arrStr[r+3],arrStr[r+4]};
                DTM.addRow(addingData);
            }
        }
        catch(Exception e){}
    }
    class ActionListenerClass1 implements ActionListener{
        public void actionPerformed(ActionEvent ae)
        {
            try{
                name=tf1.getText();
                type=tf2.getText();
                firma=tf3.getText();
                price=tf4.getText();
                date=tf5.getText();
                Object
addingData[]={name,type,firma,price,date};
                DTM.addRow(addingData);
```

```
                message = (name+ "/" + type +"/" +firma+"/"+
price + "/" +date+ "/");
                bytemessage=message.getBytes();
                os.write(bytemessage);
            }
            catch(Exception e){}
        }
    }
    class ActionListenerClass2 implements ActionListener{
        public void actionPerformed(ActionEvent ae)
        {
            try{
                String h ="End";
                bytemessage2=h.getBytes();
                os.write(bytemessage2);
            }
            catch(Exception e){}
        }
    }
    public static void main(String args[]){
        Client classObj=new Client("Client");
    }
    class WindowClose extends WindowAdapter {
        public void windowClosing(WindowEvent we) {
            try{
                String h ="End";
                bytemessage2=h.getBytes();
                os.write(bytemessage2);
            }
            catch(Exception e){}
            System.exit(0);
        }}}
//Код серверного приложения
import javax.swing.*;
import javax.swing.table.DefaultTableModel;
import java.awt.*;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.ServerSocket;
import java.net.Socket;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;
public class Server extends JFrame {
    ServerSocket sock;
    InputStream is;
    OutputStream os;
    String st;
    String stroka;
    JTable table1;
```

```
    JScrollPane sp;
    static String name, newname, type, newtype, firma, newfirma,
price, newprice, date, newdate;
    Statement sq;
    DefaultTableModel DTM;
    String vibor_naim;
    Connection db;
    String colheads[] = {"Наименование", "Тип", "Фирма", "Цена",
"Дата"};
    static Object dataConditer[][];
    byte bytemessage[] = new byte[10000];
    public Server(String Title) {
        super(Title);
        setLayout(null);
        DTM = new DefaultTableModel(dataConditer, colheads);
        table1 = new JTable(DTM);
        table1.setBackground(Color.getHSBColor(159, 216, 234));
        JScrollPane sp = new JScrollPane(table1,
ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED,
ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED);
        sp.setBounds(50, 5, 350, 250);
        add(sp);
        this.setSize(500, 450);
        this.setVisible(true);
        addWindowListener(new Server.WindowClose());
        try {
            Class.forName("com.mysql.jdbc.Driver");//получение
класса драйвера
            db =
DriverManager.getConnection("jdbc:mysql://localhost:3306/dogovor
", "root", "");
            sq = db.createStatement();
            StringBuffer x = new StringBuffer();
            String sq_str = "SELECT * FROM dogovor";
            ResultSet rs = sq.executeQuery(sq_str);
            while (rs.next()) {
                name = rs.getString("Name");
                type = rs.getString("Type");
                firma = rs.getString("Firma");
                price = rs.getString("Price");
                date = rs.getString("Date");
                Object addingData[] = {name, type, firma, price,
date};
                DTM.addRow(addingData);
                st = (name + "/" + type + "/" + firma + "/" +
price + "/" + date + "/");
                x.append(st);
            }
            stroka = x.toString();
            sock = new ServerSocket(1024);
            while (true) {
                Socket client = sock.accept();
                is = client.getInputStream();
```

```
                os = client.getOutputStream();
                bytemessage = stroka.getBytes();
                os.write(bytemessage);
                boolean flag = true;
                while (flag == true) {
                    byte readmessage[] = new byte[10000];
                    is.read(readmessage);
                    String tempString = new String(readmessage,
0, readmessage.length);
                    if (tempString.trim().compareTo("End") ==0){
                        flag = false;
                    } else {
                        stroka = stroka + tempString;
                        String arrStr[] = tempString.split("/");
                        newname = arrStr[0];
                        newtype = arrStr[1];
                        newfirma = arrStr[2];
                        newprice = arrStr[3];
                        newdate = arrStr[4];
                        Object addingData[] = {newname, newtype,
newfirma, newprice, newdate};
                        DTM.addRow(addingData);
                        String sq_str_insert = "INSERT INTO
dogovor  VALUES ('" + newname + "','" + newtype + "','" +
                                newfirma + "','" + newprice +
"','" + newdate + "')";
                        int rs_update2 =
sq.executeUpdate(sq_str_insert);
                    }
                }
                is.close();
                os.close();
                client.close();
            }
        } catch (Exception e) {
        }
    }
    public static void main(String args[]) {
        Server classObj = new Server("Server");
    }
    class WindowClose extends WindowAdapter {
        public void windowClosing(WindowEvent we) {
            System.exit(0);
        }
    }}
```

КОНТРОЛЬНЫЕ ВОПРОСЫ К ЛАБОРАТОРНОЙ РАБОТЕ:

1. Что такое драйвер базы данных? Для чего используется менеджер драйверов?
2. Для чего необходима регистрация драйвера и как она выполняется?
3. Что называется строкой соединения с базой данных?
4. Как задать значения дополнительных параметров соединения?

5. Как выполнить запрос к базе данных?

6. Что называется пакетным режимом выполнения запросов? Что такое результирующее множество?

7. Как нумеруются столбцы результирующего множества?

8. Что называют курсором?

9. Как извлечь значение из результирующего множества?

10. Как проверить, было ли извлеченное значение пустым?

11. Что называют параметром запроса? Как задать значение параметра запроса?

12. Как задать, что значение параметра является пустым?

13. Как выполнить запрос, содержащий параметры?

14. Как использовать пакетный режим с запросом, содержащим параметры?

15. Как извлечь значения ключей, сгенерированных в результате выполнения запроса?

16. Как выполнить хранимую процедуру?

17. Что такое escape-последовательность?

18. Какой курсор называют прокручиваемым? Какой курсор называют обновляемым?

19. Какие методы перемещения курсора поддерживаются?

20. Как с помощью обновляемого курсора можно вставлять, изменять и обновлять строки в таблице базы данных.

21. Как работать с большими объектами с использованием стандартных потоков ввода-вывода?

22. Как работать с большими объектами с использованием специальных интерфейсов определенных в JDBC?

ПОРЯДОК ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ РАБОТЫ:

1. Изучить теоретические сведения, полученные на лекции и лабораторной работе, ознакомиться с соответствующими материалами литературных источников.

2. Ответить на контрольные вопросы лабораторной работы.

3. Разработать алгоритм программы по индивидуальному заданию.

4. Написать, отладить и проверить корректность работы созданной программы.

5. Написать электронный отчет по выполненной лабораторной работе.

Отчет должен быть оформлен по стандарту БГУИР (Стандарт предприятия СТП 01-2017 "Дипломные проекты (работы). Общие требования"**) и иметь следующую структуру:**

1. титульный лист

2. цель выполнения лабораторной работы

3. теоретические сведения по лабораторной работе

4. формулировка индивидуального задания

5. весь код решения индивидуального задания, разбитый на необходимые типы файлов

6. скриншоты выполнения индивидуального задания

7. диаграмму созданных классов в нотации UML
8. выводы по лабораторной работе

В КАЖДОМ ЗАДАНИИ НЕОБХОДИМО РАЗРАБОТАТЬ GUI-ПРИЛОЖЕНИЕ НА ЯЗЫКЕ ПРОГРАММИРОВАНИЯ JAVA С ИСПОЛЬЗОВАНИЕМ БИБЛИОТЕК SWING ИЛИ JAVAFX В ТРЕХ (ПРОТОКОЛ TCP/IP) ИЛИ ДВУХЗВЕННОЙ АРХИТЕКТУРЕ. АВТОМАТИЗИРОВАТЬ GRUD ОПЕРАЦИИ ПО ПРЕДМЕТНОЙ ОБЛАСТИ ИНДИВИДУАЛЬНОГО ЗАДАНИЯ. ДАННЫЕ ПРИЛОЖЕНИЯ ДОЛЖНЫ ХРАНИТЬСЯ В БАЗЕ ДАННЫХ. БАЗА ДАННЫХ СОСТОИТ МИНИМУМ ИЗ ОДНОЙ ТАБЛИЦЫ. ТАБЛИЦА СОДЕРЖИТ МИНИМУМ 6 ПОЛЕЙ.

ВАРИАНТЫ ИНДИВИДУАЛЬНЫХ ЗАДАНИЙ К ЛАБОРАТОРНОЙ РАБОТЕ:

№ варианта	Предметная область
1.	студенты ВУЗа
2.	компьютерные игры
3.	каталог книг
4.	тестирование знаний студентов
5.	транспортная техника
6.	web-приложения
7.	медицинские работники
8.	каталог книг
9.	банковские операции
10.	строительная техника
11.	мобильные приложения
12.	банковские сотрудники
13.	печатная продукция
14.	бронирование авиабилетов
15.	продажа и покупка недвижимости
16.	компьютерные сети
17.	аналитика данных
18.	ассортимент услуг
19.	бронирование авиабилетов
20.	бытовая техника
21.	участники спортивных мероприятий
22.	тестирование приложений
23.	издательство печатной продукции
24.	документооборот
25.	сотрудники ИТ-организации
26.	розничная продажа товаров и услуг
27.	тестирование по английскому языку
28.	работники строительной организации
29.	сотрудники библиотеки
30.	сельскохозяйственная продукция