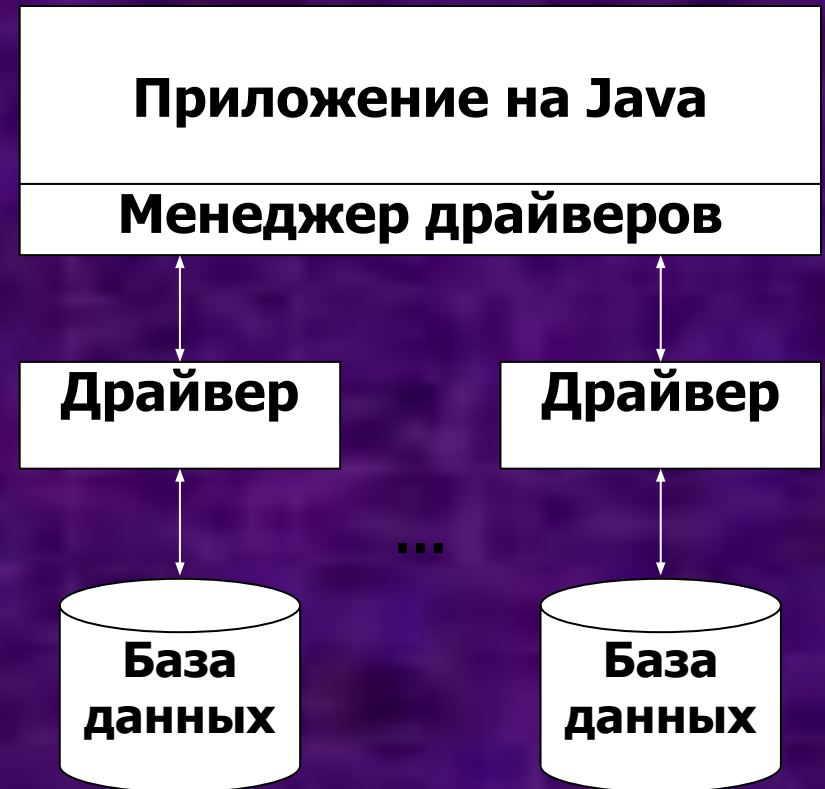


Что такое JDBC

- **JDBC (Java DataBase Connectivity)** - это платформенно-независимая технология, позволяющая из программы на Java получить доступ к любой SQL-совместимой базе данных
- **JDBC** построен по **драйверной архитектуре**, типичной для универсальных систем доступа к данным
- В настоящий момент действует стандарт **JDBC 3.0**
- Главное **достоинство JDBC** — тесное взаимодействие с другими Java-технологиями в рамках создания распределенных систем. В первую очередь это касается JNDI (Java Naming and Directory Interface) и JTS (Java Transaction Service)

Структура JDBC

- В задачу **менеджера драйверов** входит присоединение Java-приложений к требуемому драйверу JDBC
- **Драйвер** поддерживает обмен данными между приложением и базой данных



Принципы построения JDBC

- С точки зрения разработчика можно считать, что JDBC состоит из двух основных частей:
 - **JDBC API**, который содержит набор классов и интерфейсов, определяющих Java – ориентированный доступ к базам данных. Эти классы и методы объявлены в двух пакетах (package) `java.sql` и `javax.sql`
 - **JDBC-драйвера**, специфического для каждой базы данных (или других источников данных) JDBC превращает (тем или иным способом) вызовы уровня JDBC API в "родные" команды того или иного сервера баз данных

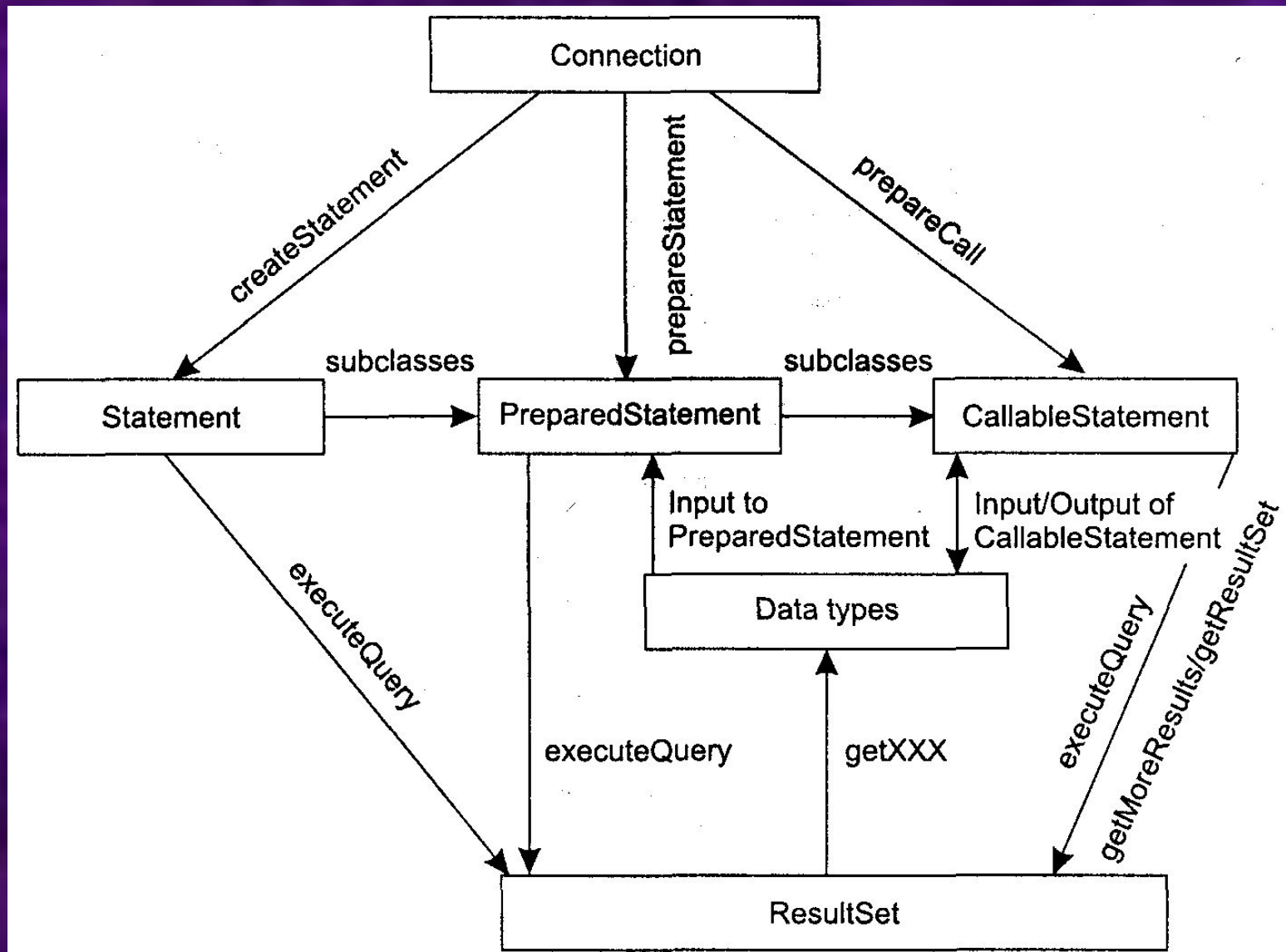
Основные классы и интерфейсы JDBC

- **java.sql.DriverManager** - позволяет загрузить и зарегистрировать необходимый JDBC-драйвер, а затем получить соединение с базой данных
- **javax.sql.DataSource** - предназначен для решения примерно тех же задач, что и DriverManager, но гораздо более удобным и универсальным образом. В JDBC 2.1 появились интерфейсы **javax.sql.ConnectionPoolDataSource** и **javax.sql.XADataSource**, которые обеспечивают поддержку пула соединений, в том числе таких соединений, которые сопоставлены с внешними (по отношению к серверу базы данных) транзакциями
- **java.sql.Connection** - обеспечивает формирование запросов к источнику данных и управление транзакциями. Предусмотрены также интерфейсы **javax.sql.PooledConnection** (логическое соединение с БД из пула соединений) и **javax.sql.XAConnection** (логическое соединение с БД из пула, сопоставленное с внешней транзакцией)

Основные классы и интерфейсы JDBC

- `java.sql.Statement` , `java.sql.PreparedStatement` и `java.sql.CallableStatement` - эти интерфейсы позволяют отправить запрос к источнику данных. Различные виды интерфейсов применяются в зависимости от того, используются ли в запросе параметры или нет и является ли запрос обращением к хранимой процедуре реляционной базы данных.
- `java.sql.ResultSet` - объявляет методы, которые позволяют перемещаться по набору данных, возвращаемых оператором `SELECT`, и считывать значения отдельных полей в текущей записи
- `java.sql.ResultSetMetaData` - позволяет получить информацию о структуре набора данных: количество полей, их названия, тип и т. д.
- `java.sql.DatabaseMetaData` Здесь объявлено большое количество методов, позволяющих получить информацию о структуре самого источника данных

Схема взаимодействия классов JDBC



Типы данных JDBC

- JDBC поддерживает **взаимное отображение между типами данных**, характерных для использования SQL, и их Java-аналогами.
- В таблице приведены соответствия между ними (каким образом JDBC-типы отображаются на Java)

JDBC Type	Java Type
CHAR	String
VARCHAR	String
LONGVARCHAR	String
NUMERIC	java.math.BigDecimal
DECIMAL	java.math.BigDecimal
BIT	Boolean
BOOLEAN	Boolean
TINYINT	Byte
SMALLINT	Short
INTEGER	Int
BIGINT	Long
REAL	Float
REF	Ref
...	...
JAVAOBJECT	Соответствующий класс Java

Группы драйверов JDBC

- **Драйверы JDBC 1.0.** Этот драйвер не поддерживает интерфейс DataSource. Все соединения, которые могут быть получены через этот драйвер (с помощью вызова метода `DriverManager.getConnection()`), являются *физическими соединениями с базой данных*.
- **Драйверы JDBC 2.0.** Эти драйверы реализуют интерфейс `javax.sql.DataSource` и, как правило, `ConnectionPoolDataSource`, то есть предоставляют возможность получения и управления *логическими соединениями*, что позволяет значительно эффективнее использовать ресурсы сервера. Но такие драйверы не обеспечивают поддержки распределенных двухфазных транзакций (этот режим необходим, например, при использовании распределенных баз данных).
- **Драйверы JDBC XA.** XA - это протокол, с помощью которого "общаются" друг с другом так называемые *менеджеры транзакций* и *менеджеры ресурсов*. Протокол XA является частью модели объектных транзакций DTP (Distributed Transaction Process) консорциума X/Open. Такие драйверы реализуют интерфейс `javax.sql.XADataSource`, что позволяет управлять логическими соединениями, сопоставленными с контекстами транзакций.

Типы драйверов JDBC

- **Мост JDBC-ODBC** + драйвер ODBC: Этот мост использует интерфейс JDBC для доступа к драйверам ODBC.
- **Наполовину-Java драйверы** - этот тип драйверов преобразуют вызовы JDBC в вызовы клиентского API для Oracle, Sybase, Informix, DB2 и т.д. Этот тип драйверов требует, чтобы на каждой клиентской станции был установлен некоторый исполняемый (двоичный) не java код.
- **Сетевые JDBC-драйверы**, которые написаны на Java и транслируют JDBC API вызовы в СУБД-независимый сетевой протокол, который затем транслируется **сервером** в СУБД-зависимый протокол. Этот сервер служит посредником между всеми своими Java-клиентами и различными СУБД.
 - Например **pg73jdbc3.jar (Драйвер для PostgreSQL)**
- **Драйверы СУБД**, которые полностью написаны на Java и сами, без дополнительного промежуточного сервера, способны "общаться" по сети с сервером БД, реализуя тот или иной сетевой протокол
 - Например: **MM.MySQL Driver**

Инсталляция драйверов JDBC

- Драйверы 3 и 4 типов написаны на Java, поэтому соответствующие классы должны быть доступны и путь к ним прописан в переменной среды CLASSPATH
- При использовании моста JDBC-ODBC должен быть создан источник данных DSN
- В JDBC описанием базы данных служит ее **URL-адрес**, который строится по образцу описателя ресурсов Интернет и имеет вид:
 - **протокол:подпротокол:информация_для_соединения**
- В качестве **протокола** всегда указывается "jdbc".
- **Подпротокол** идентифицирует драйвер JDBC. Это может быть db2 - для СУБД DB-2, mysql - для СУБД MySQL, odbc - для драйвера-моста и т.п.
- Информация для соединения зависит от выбранного драйвера и может включать имя хоста, номер порта, имя источника данных ODBC, параметры соединения и другие элементы.
- Пример: **String url = "jdbc:mysql://localhost/mentor";**

Регистрация драйверов

- Регистрация драйверов осуществляется классом **DriverManager**.
- Задача класса **DriverManager** - обеспечить поиск нужного JDBC-драйвера среди всех доступных при поступлении запроса клиента, который содержит URL нужной базы данных.
- Класс **DriverManager** хранит список объектов типа **Driver**, которые зарегистрировались с помощью вызова **DriverManager.registerDriver**.
- Класс драйвера загружается с помощью вызова
 - **Class.forName("имя класса драйвера").newInstance()**.
- В случае работы с **MySQL** нужно:
 - Добавить библиотеку с драйвером для MySQL (**mm.mysql-2.0.14-bin.jar**) в **CLASSPATH** проекта.
 - Осуществить загрузку драйвера, вызвав метод **Class.forName("org.gjt.mm.mysql.Driver").newInstance()**;

Регистрация драйвера MySQL

- Здесь приведена часть исходного кода класса `org.gjt.mm.mysql.Driver`

```
public class Driver implements java.sql.Driver {  
    ...  
    static {  
        try {  
            java.sql.DriverManager.registerDriver(new Driver());  
        } catch (java.sql.SQLException E) {  
            throw new RuntimeException("Can't register driver!");  
        }  
  
        if (debug) {  
            Debug.trace("ALL");  
        }  
    }  
    ...  
}
```

Установка соединения с БД

- Чтобы получить доступ из программы к данным базы необходимо:
 - зарегистрировать драйвер
 - задать URL и создать объект соединения при помощи одного из статических методов **getConnection()** класса DriverManager
- **Пример.** Установить соединение с базой данных MySQL "mentor"

```
Class.forName("org.gjt.mm.mysql.Driver").newInstance();  
String url = "jdbc:mysql://localhost/mentor";  
Connection con = DriverManager.getConnection(url, "root", "1");
```

- Второй и третий параметры метода **getConnection()** означают **ИМЯ ПОЛЬЗОВАТЕЛЯ** и **пароль**

Создание ConnectionFactory

```
public class ConnectionFactory {  
    private static final String DB_USER_NAME="root";  
    private static final String DB_USER_PASS="1";  
    private static final String DB_HOST="localhost";  
    private static final String DB_DATABASE="mybase";  
  
    static{  
        try{  
            Class.forName("org.gjt.mm.mysql.Driver").newInstance();  
        } catch(Exception e){  
            e.printStackTrace();  
        }  
    }  
  
    public static Connection getMySQLConnection() throws SQLException{  
        String url = "jdbc:mysql://"+ DB_HOST + "/" + DB_DATABASE;  
        return DriverManager.getConnection(url, DB_USER_NAME,  
            DB_USER_PASS);  
    }  
}
```


Выполнение SQL-команд

- Для выполнения запросов к БД в Java используются три интерфейса:
 - **Statement** - для операторов SQL без параметров
 - **PreparedStatement** - для операторов SQL с параметрами и часто выполняемых операторов
 - **CallableStatement** - для исполнения хранимых в базе процедур
- Интерфейсы **PreparedStatement** и **CallableStatement** расширяют интерфейс **Statement**, поэтому имеют все его методы
- Объекты-носители интерфейсов создаются при помощи методов объекта **Connection**
 - **createStatement()** возвращает объект **Statement**
 - **prepareStatement()** возвращает объект **PreparedStatement**
 - **prepareCall()** возвращает объект **CallableStatement**

Методы выполнения SQL-команд

- Интерфейс **Statement** предоставляет три различных метода выполнения SQL-команд:
 - **executeQuery()** - для запросов, результатом которых является один единственный набор значений, таких как запросов SELECT. Метод возвращает набор данных, полученный из базы
 - **executeUpdate()** - для выполнения операторов INSERT, UPDATE или DELETE, а также для операторов DDL (Data Definition Language). Метод возвращает целое число, показывающее, сколько строк данных было модифицировано
 - **execute()** – исполняет SQL-команды, которые могут возвращать различные результаты. Например, может использоваться для операции CREATE TABLE, и т.д.

Пример запроса данных

```
try {
    Connection con = ConnectionFactory.getMySQLConnection();
    Statement st = con.createStatement();
    // Выполнить SQL-оператор изменения данных
    String sql = "INSERT INTO tutor (id_tutor, name) VALUES (123, 'Иванов')";
    int n = st.executeUpdate(sql);
    // Выполнить SQL-оператор выборки данных
    sql = "SELECT * FROM student";
    ResultSet rs = st.executeQuery(sql);
    // Закрываем ресурсы !!!!!!!!!!!!!!!
    rs.close();
    st.close();
    con.close();
}
catch (ClassNotFoundException ex) { ...
}
catch (SQLException ex) { ...
}
```


Правильное закрытие ресурсов

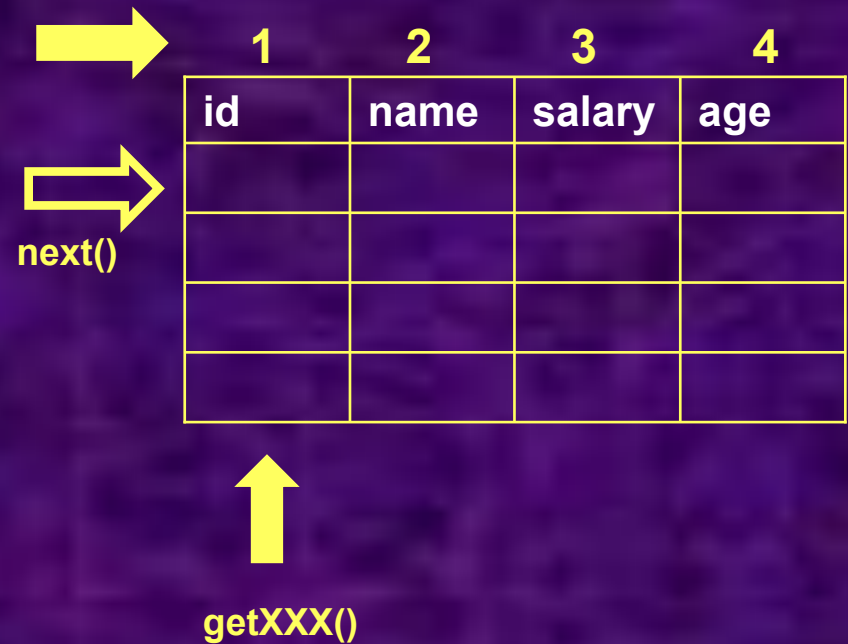
```
Connection con = null;
try{
    con = ConnectionFactory.getMySQLConnection();
    Statement st = con.createStatement();
    ResultSet rs = st.executeQuery("SELECT * FROM user");

    // doing something

    rs.close();
    st.close();
} catch(SQLException e){
    e.printStackTrace();
} finally{
    if (con != null){
        try{
            con.close();
        } catch(SQLException e){
            e.printStackTrace();
        }
    }
}
```

Наборы данных

- Метод `executeQuery()` возвращает объект с интерфейсом `ResultSet`, который хранит в себе результат запроса к базе данных
- В наборе данных есть **курсор**, который может указывать на одну из строк таблицы, эта строка называется текущей.
 - Курсор перемещается по строкам при помощи метода `next()`.
- Сразу после получения набора данных его **курсор находится перед первой строкой**. Чтобы сделать первую строку текущей надо вызвать метод `next()`
- Поля текущей записи (колонки таблицы) доступны программе при помощи методов интерфейса `ResultSet`: `getInt()`, `getFloat()`, `getString()`, `getDate()` и им подобных.



The diagram illustrates the state of a database cursor. A table with 4 columns (id, name, salary, age) and 5 rows is shown. A thick black arrow points to the first row, and a thinner black arrow labeled 'next()' points to the second row, indicating the cursor's current position. Below the table, an upward-pointing black arrow labeled 'getXXX()' indicates the retrieval of data from the current row.

1	2	3	4
id	name	salary	age

getXXX()

Пример получения данных

// Получить набор данных

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

```
ResultSet rs = st.executeQuery("SELECT * FROM Student");
```

// Распечатать набор данных

```
while (rs.next()) {
```

 // Напечатать значения в текущей строке.

```
    int id_student = rs.getInt("id_student");
```

```
    String name = rs.getString("name");
```

```
    float average = rs.getFloat("average");
```

```
    System.out.println("Строка = " + id_student + " " + name + " " +  
    average);
```

```
}
```

// Закрыть объект оператора

```
st.close();
```


Значение NULL

- Некоторые поля таблиц в базах данных могут **не иметь значения**.
 - Те методы **ResultSet.getXxx()**, которые преобразуют данные в **объектный тип**, возвращают в этом случае значение **null**.
 - Те методы, которые преобразуют данные в **простые типы**, возвращают **0**
- Чтобы отличить не инициализированные поля от полей с нулевым значением, можно воспользоваться методом **ResultSet.isNull()**

Большие значения

- Некоторые поля таблиц способны хранить очень большие значения (тексты, двоичные объекты, изображения и т.п.)
- Если программа не готова принять большое значение сразу, например, методом getString() или getObject(), она может сделать это частями: для этого есть такие методы как
 - `getBinaryStream()` – поток байтов
 - `getAsciiStream()` – поток однобайтовых ASCII-символов
 - `getCharacterStream()` – поток Unicode-символов

```
Statement st = con.createStatement();
ResultSet rs = st.executeQuery("SELECT biography FROM Student WHERE
    id_student=123");
// Теперь считываем колонку результатов частями по 4 К:
byte[] buff = new byte[4096];
while (rs.next()) {
    InputStream fin = rs.getAsciiStream("biography");
    for (;;) {
        int size = fin.read(buff);
        if (size == -1) break;
        System.out.write(buff, 0, size);    }
    }
st.close();
```

Интерфейс PreparedStatement

- Экземпляры **PreparedStatement** сохраняют скомпилированные SQL-выражения.
- Особенностью SQL-выражений в PreparedStatement является то, что они могут иметь параметры
 - Параметризованное выражение содержит знаки вопроса в своем тексте.
 - Например: **"SELECT name from persons where age=?"**
 - Перед выполнением запроса значение каждого вопросительного знака явно устанавливается методами **setXxx()**
 - Например: **ps.setInt(1, 30);**
- Использование PreparedStatement приводит к **более быстрому выполнению запросов** при их многократном вызове с различными параметрами

Пример использования PreparedStatement

// Создать и откомпилировать запрос с параметрами

```
PreparedStatement st = con.prepareStatement("SELECT * FROM  
student WHERE id_student > ? AND name = ? ");
```

// Установить значение параметров

```
st.setInt(1, 100);  
st.setString(2, "ОЛЕГ");
```

// Выполнить прекомпилированный запрос

```
ResultSet rs = st.executeQuery();
```

Интерфейс CallableStatement

- Интерфейс **CallableStatement** используется, когда необходимо обратиться к хранимой процедуре.
- Работа с интерфейсом **CallableStatement** усложняется несколькими обстоятельствами:
 - Хранимые процедуры сильно **отличаются** (с точки зрения **синтаксиса** их вызова) для **различных серверов баз данных**
 - Хранимые процедуры **отличаются друг от друга** тем, какие **результаты они могут возвращать** и как именно выполняется возврат.
 - Например, для некоторых серверов (точнее, их JDBC-драйверов) хранимые процедуры могут возвращать **только out- (и inout-)** параметры,
 - Для других серверов процедура возвращает out- и inout-параметры как **поля в наборе записей** (типа ResultSet).
 - Третьи поддерживают **и тот, и другой подход**
- При работе с интерфейсом **CallableStatement** (который является производным от интерфейса **PreparedStatement**) широко используется **escape-синтаксис**, и программист задает команду обращения к процедуре либо с помощью вызова метода **executeQuery()**, либо с помощью **executeUpdate()**.
- В любом случае при работе с конкретным JDBC-драйвером нужно изучить, каким образом этот драйвер обеспечивает взаимодействие с хранимыми процедурами

Пример вызова хранимой процедуры для Interbase

- Эта хранимая процедура имеет единственный входной аргумент типа integer, а в качестве выходного результата тоже возвращает целое число
- Синтаксис вызова хранимой процедуры и способ трактовки out-аргументов являются **специфическими для используемого JDBC-драйвера**

```
// Задание SQL-выражения вызова хранимой процедуры
// Синтаксис является специфическим для Interbase
CallableStatement pstmt = connection.prepareCall(
    "execute procedure getUniqueValue (?");
// Escape-синтаксис (о чем говорит использование фигурных скобок)
// Можно использовать и тот. и другой способ задания выражения
CallableStatement pstmt = connection.prepareCall
    ("{"call getUniqueValue (?)}");
// Настройка параметров и обращение к процедуре
pstmt.setInt (1, 1);
ResultSet rs =pstmt.executeQuery();
```


Работа с метаданными

- Информация о метаданных (то есть о самой структуре, а не о содержимом базы данных) бывает двух видов:
 - метаданные набора данных (ResultSet)
 - метаданные источников данных
- В первом случае для представления метаданных используется интерфейс **ResultSetMetaData**, который может быть получен следующим образом:

```
ResultSet rs = stmt.executeQuery(sqlString);  
ResultSetMetaData rsmd = rs.getMetaData();
```
- Во втором случае нужно использовать интерфейс **DatabaseMetaData**. Он доступен с помощью метода **getMetadata()** интерфейса **Connection**:

```
DatabaseMetaData dbmd = connection.getMetadata();
```
- **DatabaseMetaData** содержит методы получения информации о структуре таблиц, атрибутах столбцов, о поддержке СУБД различных функций, и т.д.

Получение значения autoincrement-поля в mysql

- Иногда возникает необходимость сразу после выполнения операции **INSERT** узнать значение ключа только что добавленной записи
- В случае использования **autoincrement-ключа** это возможно с помощью вызова Mysql-функции **last_insert_id()**.
- Следующий пример демонстрирует получение добавленного ключа:

```
Connection con = ConnectionFactory.getMySQLConnection();
Statement st = con.createStatement();
st.executeUpdate("INSERT INTO keytest (name, num) VALUES ('vasya', '1')");
ResultSet rs = st.executeQuery("SELECT last_insert_id()");
if (rs.next()){
    int key = rs.getInt(1);
    System.out.println(key);
} else{
    System.out.println("Error getting key");
}
```

Работа с транзакциями

- Речь идет о транзакциях, управление которыми полностью берет на себя JDBC-совместимый источник данных (как правило, SQL-сервер)
- В транзакции участвует только один источник данных, и транзакция завершается в однофазном режиме с помощью вызова методов **commit ()** или **rollback()** на уровне конкретного соединения с БД .
- Все методы, необходимые для управления транзакциями, объявлены в интерфейсе **java.sql.Connection**.
 - Это означает, что не может существовать локальной транзакции, в которой участвует несколько соединений.

Работа с транзакциями

- Транзакцию начинает сам сервер, когда она необходима для выполнения тех или иных действий, например исполнения SQL-команды INSERT.
- Завершение транзакции возможно как явное, так и неявное (с помощью вызова метода `commit()` или `rollback()`).
 - Режим завершения транзакции (явное или неявное) задается с помощью метода `setAutoCommit()`.

```
Connection connection = myDataSource.getConnection();  
Connection.setAutoCommit (false);  
// Операции с БД  
connection.commit();
```

Работа с пулом соединений

- Применение пулов соединений позволяет повысить производительность приложения за счет переиспользования объектов-соединений с БД
- Для доступа к пулу соединений, как правило, используется служба JNDI.

Конфигурация пула при использовании сервера Tomcat

- В каталоге `$CATALINA_HOME/common/lib` должны находиться файлы
 - Jakarta-Commons DBCP 1.0
 - Jakarta-Commons Collections 2.0
 - Jakarta-Commons Pool 1.0

- В конфигурации контекста должны быть указаны элементы

```
<Resource name="jdbc/mysqlIDs" auth="Container" type="javax.sql.DataSource"/>
```

```
<ResourceParams name="jdbc/mysqlIDs">
```

```
  <parameter>
```

```
    <name>factory</name>
```

```
    <value>org.apache.commons.dbcp.BasicDataSourceFactory</value>
```

```
  </parameter>
```

```
  ...
```

```
</ResourceParams>
```

- Полный список параметров см. на http://lab253.iyc.kture/library/j2ee/doc/pool/The_Tomcat_4_JNDI_Datasource_HOW-TO.htm

Конфигурация пула при использовании сервера Tomcat

- В файле **web.xml** должно быть указано

```
<resource-ref>  
  <description>DB Connection</description>  
  <res-ref-name>jdbc/TestDB</res-ref-name>  
  <res-type>javax.sql.DataSource</res-type>  
  <res-auth>Container</res-auth>  
</resource-ref>
```

- Получение объекта **Connection** из пула

```
Context initContext = new InitialContext();  
Context envContext = (Context)initContext.lookup("java:/comp/env");  
DataSource ds = (DataSource)envContext.lookup("jdbc/myoracle");  
Connection conn = ds.getConnection();
```

Предотвращение потери соединений при использовании пула

- В случае, если соединение с БД по каким-то причинам не было закрыто, по умолчанию оно **не возвращается в пул**.
- Если в пуле не остается свободных соединений, приложение может **зависнуть, ожидая закрытия соединения**.
- Чтобы избежать этого, при использовании пула в элементе **<ResourceParams>** нужно добавить следующие параметры:
 - **removeAbandoned** - true
 - **removeAbandonedTimeout** — интервал, по истечении которого будет возвращено незакрытое соединение
 - **logAbandoned** - true