БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ и ИНФОРМАТИКИ
Кафедра многопроцессорных сетей и систем

Рафеенко Е.Д.

Web- программирование

Доступ к базам данных

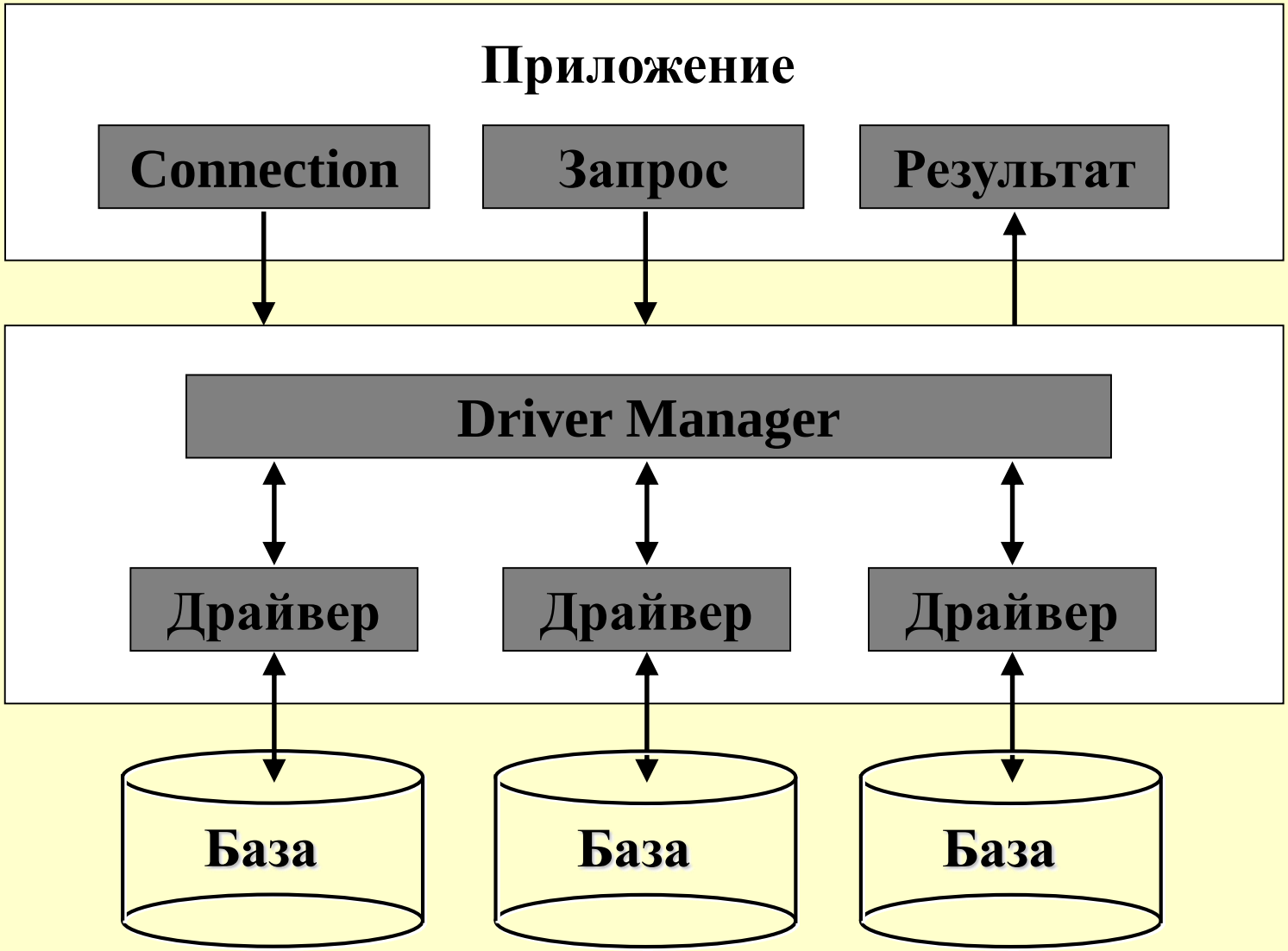
Содержание

Технология JDBC (Java Database Connectivity)

Технология JDBC

- JDBC – это стандартный прикладной интерфейс (API) языка Java для организации взаимодействия между приложением и СУБД.
  - JDBC является частью стандартной версии Java и находится в пакете **`java.sql`**
  - Дополнительная функциональность представлена в пакете **`javax.sql`**

Архитектура приложения, использующего JDBC

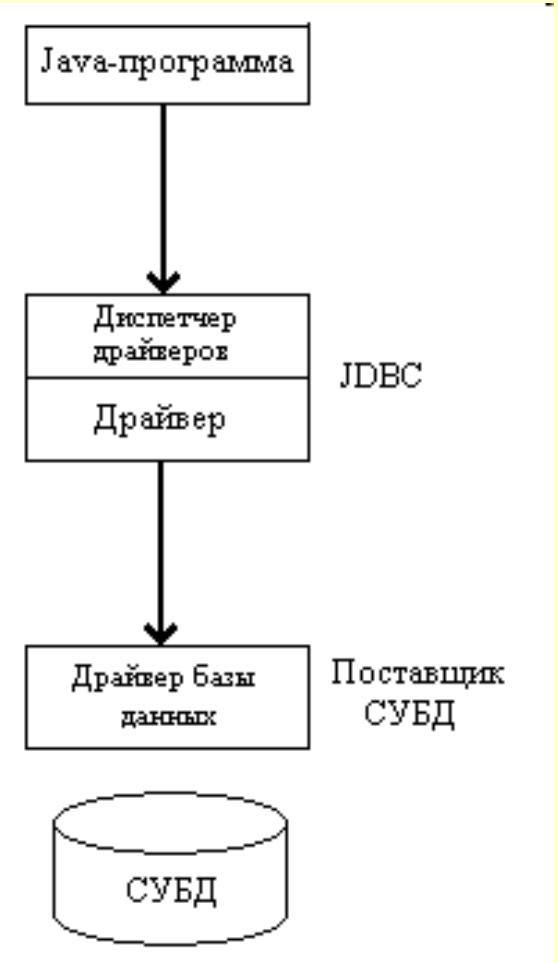


Концепция драйверов базы данных

JDBC основана на концепции драйверов, позволяющих получать соединение с базой данных по специально описанному URL.

Драйверы JDBC обычно создаются поставщиками СУБД.

Их работа заключается в обработке JDBC-подключений и команд, поступающих от Java-приложения, и в генерации машинно-зависимых вызовов по отношению к базе данных.



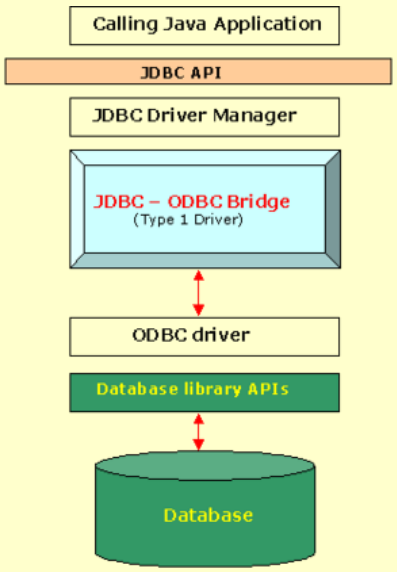
Типы драйверов

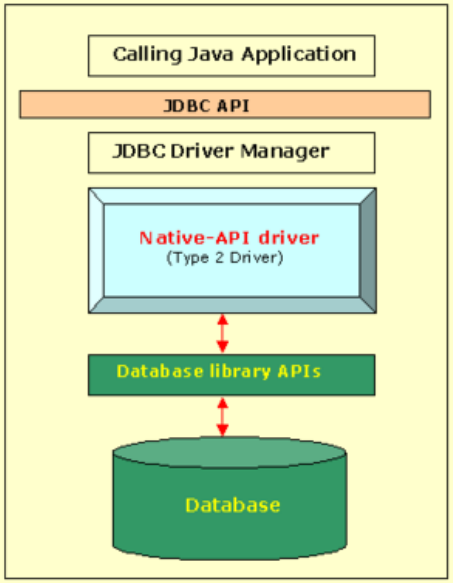
1 : "bridge" драйвер (использует прикладной интерфейс Open Database Connectivity (ODBC) для взаимодействия с БД.)

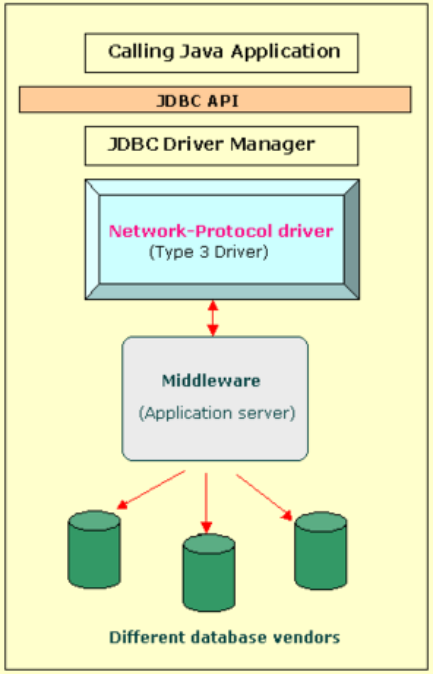
2 : драйвер использует native API для взаимодействия с БД.

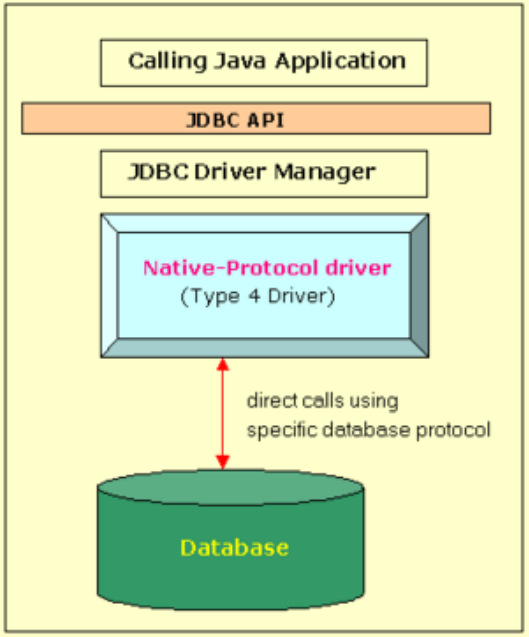
3 : драйвер использует сетевой протокол и middleware для взаимодействия с БД.

4 : драйвер использует Java. Работает напрямую с сервером БД.









Компоненты JDBC

- Driver Manager
  - предоставляет средства для управления набором драйверов баз данных
  - предназначен для выбора базы данных и создания соединения с БД.
- Драйвер
  - обеспечивает реализацию общих интерфейсов для конкретной СУБД и конкретных протоколов
- Соединение (Connection)
  - Сессия между приложением и драйвером базы данных

Компоненты JDBC (2)

- Запрос
  - SQL запрос на выборку или изменение данных
- Результат
  - Логическое множество строк и столбцов таблицы базы данных

Метаданные

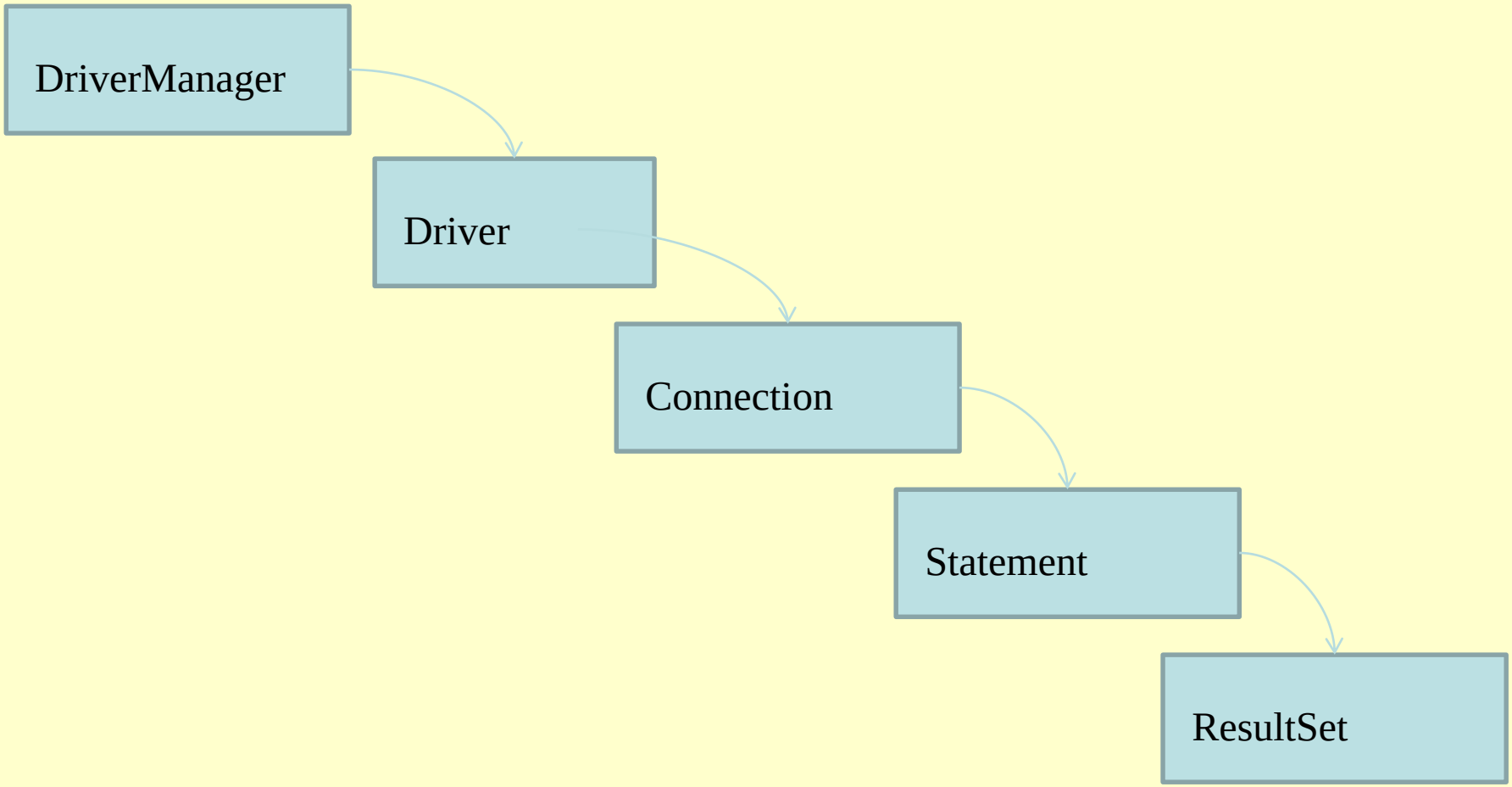
- Сведения о полученном результате и об используемой базе данных

Использование JDBC.

- Последовательность действий:

1. Загрузка класса драйвера базы данных.
2. Установка соединения с БД.
3. Создание объекта для передачи запросов.
4. Выполнение запроса.
5. Обработка результатов выполнения запроса.
6. Закрытие соединения.

Последовательность действий



Шаг 1: Загрузка класса драйвера базы данных

- Загрузка класса драйвера базы данных:
- в общем виде:

```
Class.forName([location of driver]);
```

- для MySQL:

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

- для JDBC-ODBC bridge (ex. MS Access) :

```
Class.forName("sun.Jdbc.odbc.jdbcodbcDriver ");
```

Шаг 2: Установка соединения с БД

- **Для установки соединения необходимо вызвать метод** getConnection() **класса** DriverManager.
- **В качестве параметров передаются:**
  - Тип, физическое месторасположение и имя БД;
  - Логин и пароль для доступа.

```
Connection cn = DriverManager.getConnection(
"jdbc:mysql://localhost/my_db", "login", "password");
```

- **В результате будет возвращен объект** Connection**, содержащий одно установленное соединение с БД** my_db**.**

Шаг 2: Установка соединения с БД

- Еще один способ соединения с базой данных возможен с использованием файла ресурсов database.properties, в котором хранятся, как правило, путь к БД, логин и пароль доступа.

- Например:

  url=jdbc:mysql://localhost/my_db?useUnicode=true

  characterEncoding=Cp1251

  driver=com.mysql.cj.jdbc.Driver

  user=root

  password=pass

Шаг 2: Установка соединения с БД

```
public Connection getConn() throws JDBCConnectionException {
    ResourceBundle resource = ResourceBundle.getBundle("database");
    String url = resource.getString("url");
    String driver = resource.getString("driver");
    String user = resource.getString("user");
    String pass = resource.getString("password");
    try {
            Class.forName(driver).newInstance();
    } catch (ClassNotFoundException e) {
       throw new JDBCConnectionException ("Драйвер не загружен!");
    } catch (InstantiationException e) {
            e.printStackTrace();
    } catch (IllegalAccessException e) {
            e.printStackTrace();
    }
   return DriverManager.getConnection(url, user, pass);
}
```

Шаг 3: Создание объекта для передачи запросов

- **Объект** Statement
  - используется для выполнения запросов и команд SQL, а также для установки некоторых ограничений на запросы;
  - один и тот же объект **Statement** может быть использован многократно для различных запросов.

```
Connection dbCon = DriverManager.getConnection(
 "jdbc:mysql://localhost/my_db","admin", "secret");
Statement stmt = dbCon.createStatement();
```

Шаг 4: Выполнение запроса

- **Метод executeQuery() выполняет предварительно созданный SQL запрос на выборку (SELECT).**
- **Результаты выполнения запроса помещаются в объект ResultSet.**

```
Connection dbCon =
  DriverManager.getConnection(
 "jdbc:mysql://localhost/my_db","admin",
"secret");

Statement stmt = dbCon.createStatement();

ResultSet rs = stmt.executeQuery(
  "SELECT first_name FROM employees");
```

Шаг 4: Выполнение запроса

- **Для** INSERT/UPDATE/DELETE **запросов используется метод** executeUpdate(), **который возвращает количество добавленных (измененных, удаленных) записей.**

```
Statement stmt = dbCon.createStatement();
int rowsAffected = stmt.executeUpdate(
  "UPDATE employees SET salary = salary*1.2");
```

Шаг 5: Обработка результатов выполнения запроса

- **Содержится в объекте** ResultSet
- **Методы:**

  boolean next()

  xxx getXxx(int columnNumber)

  xxx getXxx(String columnName)

  void close()

- **Итератор первоначально устанавливается в позицию перед первой строкой**
  - Необходимо вызвать **next()** для перемещения в позицию первой строки.
  - Когда строки закончатся, метод **next()** возвратит значение **false**.

Шаг 5: Обработка результатов выполнения запроса

Исходная таблица: Employees.

Id	FirstName	LastName	Address
1	Илья	Петров	ул. Кульман, 16-45
2	Николай	Иванов	ул. Гамарника, 46-120
3	Иван	Сидоров	ул. Гикало, 32-24

```
ResultSet rs = st.executeQuery("SELECT LastName + ' ' + FirstName AS FullName, Address FROM Employees");
```

В результате rs содержит:

FullName	Address
Илья Петров	ул. Кульман, 16-45
Николай Иванов	ул. Гамарника, 46-120
Иван Сидоров	ул. Гикало, 32-24

```
while(rs.next())
{
    System.out.println(
            rs.getString("FullName") +
"\t" +      rs.getString("Address"));
}
```

Необходимые сведения о структуре БД

- Существует целый ряд методов интерфейса ResultSetMetaData с помощью которых можно определить типы, свойства и количество столбцов БД.

```
ResultSet rs = stmt.executeQuery(
  "SELECT * FROM employees");
ResultSetMetaData rsm = rs.getMetaData();
int number = rsm.getColumnCount();
for (int i=0; i<number; i++) {
  System.out.println(rsm.getColumnName(i));
}
```

Шаг 6: Закрытие соединения

- **По окончании использования необходимо последовательно вызвать метод** close() **для объектов** ResultSet**,** Statement **и** Connection **для освобождения ресурсов.**

```
try {
  Connection conn = ...;
  Statement stmt = ...;
  ResultSet rset = stmt.executeQuery(...);
  ...
} finally
    // clean up
    rset.close();
    stmt.close();
    conn.close();
}
```

Объект SQLException

- Обязательно обрабатывайте исключительные ситуации (java.sql.SQLException).

```
try  {
  rset = stmt.executeQuery(
    "SELECT first_name, last_name FROM employee");
  } catch (SQLException sqlex) {
      ... // Обрабатываем ошибки
  } finally {
    // Освобождаем использованные ресурсы
    try {
      if (rset != null) rset.close();
    } catch (SQLException sqlex)  {
      ... // Обрабатываем ошибки при закрытии
    }
  ...
}
```

Рекомендации к индивидуальному заданию:

1. Конфигурацию базы хранить в property файле.
2. Отдельным классом реализовать работу с Connection.
3. Реализовать интерфейс для работы с DAO (основные операции: чтение, запись, удаление, поиск).

```
public class JdbcConnector {
    private Connection conn;
    public Connection getConnection() throws JDBCConnectionException {
        ConfigurationManager cfg = ConfigurationManager.getInstance();
        try {
            Class.forName(cfg.getDriverName);
            conn = DriverManager.getConnection(cfg.getURL(), cfg.getLogin(),
                cfg.getPassword());
        } catch (ClassNotFoundException e) {
          throw new JDBCConnectionException("Can't load database driver.", e);
        } catch (SQLException e) {
           throw new JDBCConnectionException("Can't connect to database.", e);
        }
        if(conn==null) {
            throw new JDBCConnectionException("Driver type is not correct in URL " +
      cfg.getProperty(ConfigurationManager.DB_URL) + ".");
        }
        return conn;
    }
...
```

Закрытие Connection

```
public void close() throws JDBCConnectionException {
        if (conn != null) {
            try {
                conn.close();
            } catch (SQLException e) {
 throw new JDBCConnectionException("Can't close connection", e);
            }
        }
    }

}
```

Использование DAO (Data Access Object)

Используйте Data Access Object (DAO) для абстрагирования и инкапсулирования доступа к источнику данных.

DAO управляет соединением с источником данных для получения и записи данных.

Пример DAO (Data Access Object)

```
public class DAO {
protected JdbcConnector cnr;
public DAO() throws DAOException
try
{
cnr = new JdbcConnector();
}
catch (JDBCConnectionException e){
throw new DAOException("Can't create JdbcConnector ", e);
}}
public JdbcConnector getJdbcConnector() {
return cnr;
}


…
}
```

```
public class DaoUser extends DAO{

public DaoUser() throws DAOException{
    super(); }

public User readUser(Integer id) throws DAOException{
User user=null;
try{
cnr.getConnection();
….
}catch(JDBCConnectionException e)
{int key= id; throw new DAOException("Can't obtain user", key, e); }
finally{
try {cnr.close();}
 catch (JDBCConnectionException e)
{throw new DAOException("Can't  close  conn", e); }
}
return user;
}
```

... //продолжение на следующем слайде

Пример DAO (Data Access Object)

```
public void createUser(User usr) throws DAOException{
  …
}

public void deleteUser () throws DAOException {
   …
 }
public void updateUser () throws DAOException {
   …
 }

…
}
```

Подготовленные запросы и хранимые процедуры

Интерфейсы PreparedStatement и CallableStatement

Подготовленные запросы и хранимые процедуры

- PreparedStatement
  - Предварительно готовится и хранится в объекте. Позволяет ускорить обмен информацией с БД.
- CallableStatement
  - Используется для выполнения хранимых процедур, созданных средствами самой СУБД.

PreparedStatement

- **Для компиляции SQL запроса используется метод** prepareStatement(String sql), **возвращающий объект** PreparedStatement.

- **Подстановка реальных значений происходит с помощью методов** setString(), setInt() **и подобных им.**

- **Выполнение запроса производится методами** executeUpdate(), executeQuery().

- **PreparedStatement-оператор предварительно откомпилирован, поэтому он выполняется быстрее обычных операторов ему соответствующих.**

PreparedStatement - Пример

```
class Rec {
        static void insert(PreparedStatement ps, int id, String name,
                String surname, int salary) throws SQLException {
                ps.setInt(1, id);
                ps.setString(2, name);
                ps.setString(3, surname);
                ps.setInt(4, salary);
        //      выполнение компилированного запроса
                ps.executeUpdate();
        }
}
…
Connection cn = null;
…
String sql = "INSERT INTO emp(id,name,surname,salary)
                                        VALUES(?,?,?,?)";
//      компиляция запроса
PreparedStatement ps = cn.prepareStatement(sql);
Rec.insert(ps, 2203, "Иван", "Петров", 230);
```

Prepared Statement

- PreparedStatement используется для:
  - Выполнения запроса с параметрами;
  - Улучшения производительности в случае частого использования запроса.

CallableStatement

- В терминологии JDBC, хранимая процедура - последовательность команд SQL, хранимых в БД и доступных любому пользователю этой СУБД. Механизм создания и настройки хранимых процедур зависит от конкретной базы данных.

- Интерфейс CallableStatement обеспечивает выполнение хранимых процедур

- Объект CallableStatement содержит команду вызова хранимой процедуры, а не саму хранимую процедуру.

- CallableStatement способен обрабатывать не только входные (**IN**) параметры, но и выходные (**OUT**) и смешанные (**INOUT**) параметры. Тип выходного параметра должен быть зарегистрирован методом registerOutParameter**()**.

- После установки входных и выходных параметров вызываются методы execute**()**, executeQuery**()** или executeUpdate**()**.

CallableStatement - пример

В БД существует хранимая процедура **getempname**, которая по уникальному для каждой записи в таблице **employee** числу SSN возвращает соответствующее ему имя:

```
CREATE PROCEDURE getempname (emp_ssn IN INT,
                             emp_name OUT VARCHAR) AS
BEGIN
        SELECT name INTO emp_name FROM employee
                WHERE SSN = EMP_SSN;
END
```

CallableStatement - Пример

Вызов данной процедуры из программы:

```
String SQL = "{call getempname (?,?)}";
CallableStatement cs = conn.prepareCall(SQL);
cs.setInt(1,822301);
//регистрация выходящего параметра
cs.registerOutParameter(2,java.sql.Types.VARCHAR);
cs.execute();
String empName = cs.getString(2);
System.out.println("Employee with SSN:" + ssn + " is " +
empName);
// Будет выведено: Employee with SSN:822301 is Spiridonov
```

Чтение автоматически сгенерированного ключа

- Для СУБД, которые поддерживают "auto increment" поля
  - Например MS SQL Server, MySQL, ...
  - JDBC имеет доступ к автоматически сгенерированным ключам

```
// добавляем запись...
int rowCount = stmt.executeUpdate(
  "INSERT INTO Messages(Msg) VALUES ('Test')",
  Statement.RETURN_GENERATED_KEYS);

// ... и получаем ключ
ResultSet rs = stmt.getGeneratedKeys();
rs.next();
long primaryKey = rs.getLong(1);
```

Batch-команды

Механизм batch-команд позволяет запускать на исполнение в БД массив запросов SQL вместе, как одну единицу.

```
con.setAutoCommit(false);
Statement stmt = con.createStatement();
stmt.addBatch("INSERT INTO employee VALUES (10, 'Joe ')");
stmt.addBatch("INSERT INTO location VALUES (260, 'Minsk')");
stmt.addBatch("INSERT INTO emp_dept VALUES (1000, 260)");
// submit a batch of update commands for execution
int[] updateCounts = stmt.executeBatch();
```

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