

EPAM Systems, RD Dep.

Конспект и раздаточный материал

# JAVA.SE.10 JDBC Fundamentals

REVISION HISTORY					
Ver.	Description of Change	Author	Date	Approved	
				Name	Effective Date
<1.0>	Первая версия	Игорь Блинов	<27.12.2011>		
<2.0>	Вторая версия. Конспект переделан под обновленное содержание материала модуля.	Ольга Смолякова	<18.08.2014>		

**Legal Notice**

This document contains privileged and/or confidential information and may not be disclosed, distributed or reproduced without the prior written permission of EPAM Systems.

## СОДЕРЖАНИЕ JAVA.SE.10 JDBC

1. Что такое JDBC
2. Модели доступа к БД
3. Компоненты JDBC
4. Типы драйверов
5. Использование JDBC
6. Загрузка драйвера базы данных
7. Установка связи с БД
8. Выполнение sql-запросов
9. Statement
10. ResultSet
11. PreparedStatement
12. CallableStatement
13. Batch-команды
14. Заккрытие ResultSet. Statement и Connection
15. Connection Pool
16. Data Access Object (DAO)
17. Транзакции и точки сохранения
18. Метаданные

### Legal Notice

This document contains privileged and/or confidential information and may not be disclosed, distributed or reproduced without the prior written permission of EPAM Systems.

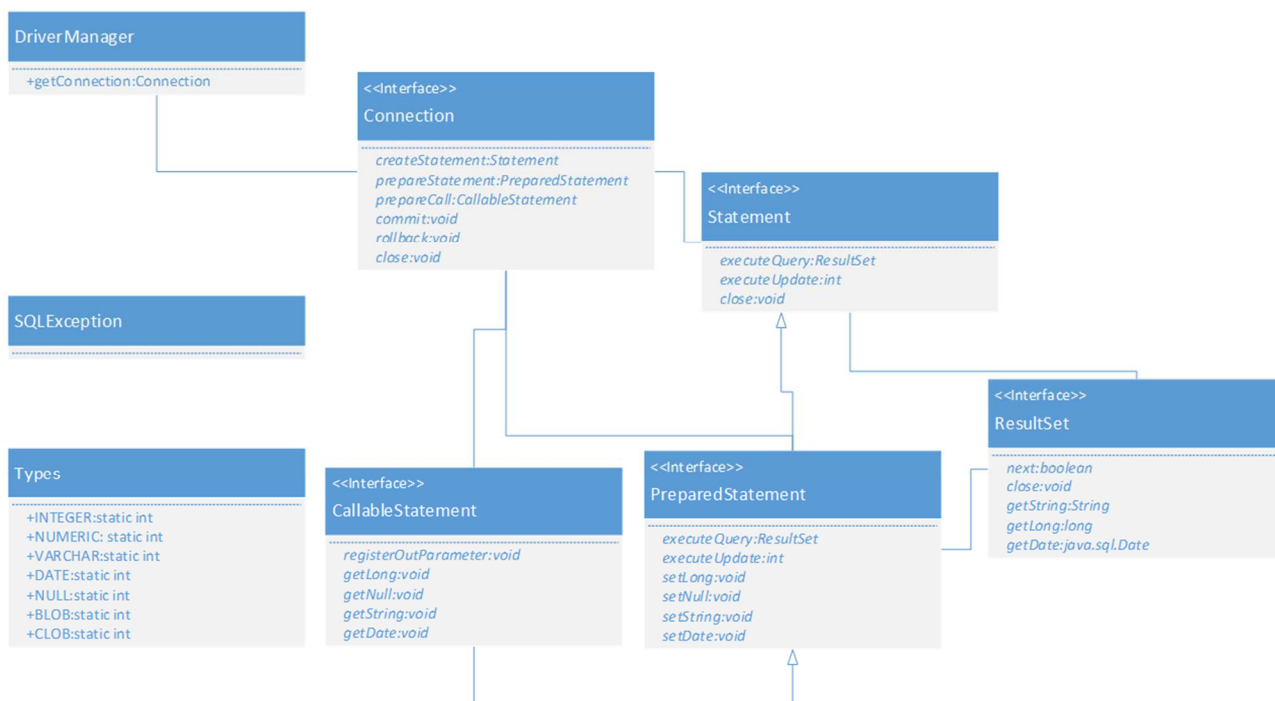
## Что такое JDBC



**JDBC** - это **прикладной программный интерфейс (API)** Java для выполнения **SQL-запросов**.

**JDBC** предоставляет стандартный **API** для разработчиков, использующих базы данных.

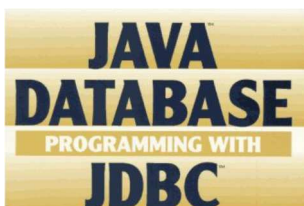
## Основные интерфейсы и классы JDBC



## Что может JDBC?

- Устанавливать соединение с БД
- Отсылать SQL-запросы
- Обработать результаты.

Использование JDBC API **избавляет от необходимости** для каждой СУБД (Informix, Oracle и т.д.) **писать свое приложение**. Достаточно написать одну единственную программу, использующую JDBC API, и эта программа сможет отсылать SQL-запросы к требуемой БД.



## Legal Notice

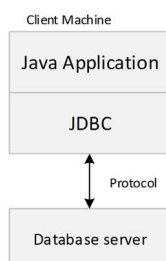
This document contains privileged and/or confidential information and may not be disclosed, distributed or reproduced without the prior written permission of EPAM Systems.

Версии JDBC.

Year	JDBC Version	JSR Specification	JDK Implementation
2014	JDBC 4.2	JSR 221	Java SE 8
2011	JDBC 4.1	JSR 221	Java SE 7
2006	JDBC 4.0	JSR 221	Java SE 6
2001	JDBC 3.0	JSR 54	JDK 1.4
1999	JDBC 2.1		JDK 1.2
1997	JDBC 1.2		JDK 1.1

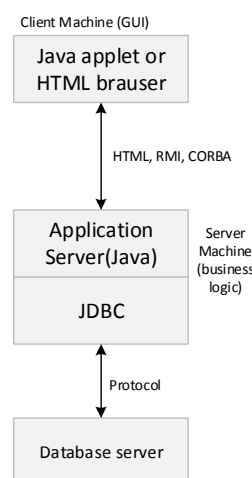
## Модели доступа к БД

JDBC использует двух- и трехзвенные модели для доступа к БД.



В двухзвенной модели приложение или апплет на языке Java обращается непосредственно к БД. В этом случае JDBC-драйвер "умеет" общаться с соответствующей СУБД. SQL-запросы отсылаются в СУБД, а результаты отсылаются обратно к пользователю. БД может находиться на другой машине, с которой пользователь соединяется по сети. И пользовательская машина, и сервер БД работают в конфигурации клиент/сервер. В качестве сети может выступать Интранет, соединяя между собой работников корпорации, или Интернет.

В трехзвенной модели команды поступают в т.н. сервис среднего звена, который отсылает SQL-выражения в БД. БД обрабатывает SQL, отсылая запросы в этот самый сервис, который затем возвращает результат конечному пользователю. Преимущества: контроль доступа и изменения, вносимые в корпоративную БД; программист может реализовать свой собственный предметно-ориентированный API, который транслируется средним звеном низкоуровневые SQL-запросы. Во многих случаях трехзвенная архитектура может увеличить производительность.



## 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

### Legal Notice

This document contains privileged and/or confidential information and may not be disclosed, distributed or reproduced without the prior written permission of EPAM Systems.

- **Соединение (Connection)**
  - Сессия между приложением и драйвером базы данных
- **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

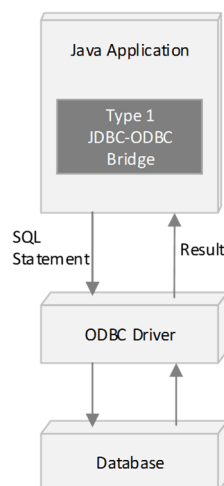
## Типы драйверов

1. Мост JDBC-ODBC
2. Нативный-API / частичный Java драйвер
3. Сетевой протокол / «чистый» Java драйвер
4. Нативный протокол / «чистый» Java драйвер

### JDBC-ODBC bridge драйвер. (драйвер типа 1)

Мост JDBC-ODBC драйвер конвертирует JDBC-запросы в запросы открытого интерфейса доступа к базам данных (ODBC). Мост JDBC-ODBC драйвер дает возможность Java-приложению использовать любую БД, которую поддерживает ODBC драйвер.

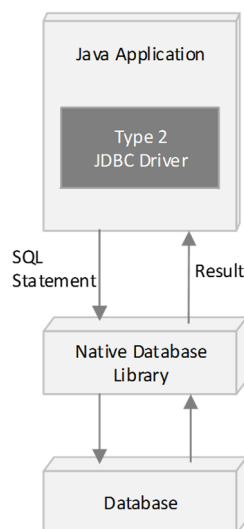
Чтобы использовать JDBC-ODBC bridge, необходимо установить драйвер ODBC на клиентском компьютере. Этот драйвер обычно используется в автономных приложениях.



### Native-API partly java driver (драйвер типа 2).

Использует собственные локальные библиотеки для доступа к базам данных, поставляемыми производителями баз данных. Драйвер преобразует запросы JDBC в собственные методы запросов, которые поступают на локальный собственный интерфейс уровня запроса Call Level Interface (CLI).

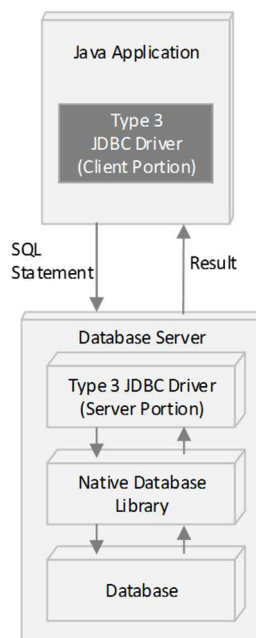
CLI должен быть загружен на клиентском компьютере. Обычно используется для сетевых приложений.



### JDBC-Net pure java driver (драйвер типа 3) –

состоит из клиентской и серверной части, клиентская часть содержит только функции, а серверная часть содержит методы Java, а также свои собственные. Java-приложение посылает запросы к JDBC-net pure Java драйверу клиентской части, который, в свою очередь переводит JDBC-запросы в запросы базы данных. Запросы к базе данных посылаются серверной части JDBC-Net Pure Java-драйвера, который пересылает запрос базе данных.

Это драйвер используется в веб-приложениях и при взаимодействии апплетов с БД.



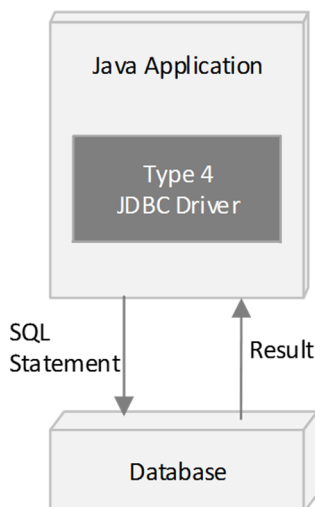
### Native-Protocol pure-java драйвер (драйвер 4-го типа) –

драйвер java, взаимодействует с БД, непосредственно использующей сетевые протоколы, специфицированные производителем. Не требует установки какой бы то ни было библиотек, специфицированной производителем.

#### Legal Notice

This document contains privileged and/or confidential information and may not be disclosed, distributed or reproduced without the prior written permission of EPAM Systems.

Обычно используется для корпоративных приложений.



## 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.

## Загрузка драйвера базы данных

- в общем виде:

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

- для MySQL:

**Class.forName("org.gjt.mm.mysql.Driver");**

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

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

Согласно принятому соглашению классы JDBC-драйверов регистрируют себя сами при помощи Driver Manager во время своей первой загрузки.

В общем драйверы JDBC можно зарегистрировать с помощью системных свойств Java или в программе на Java.

- Регистрация с помощью системных свойств:

**"-Djdbc.drivers=com.ibm.as400.access.AS400JDBCdriver"**

### Legal Notice

This document contains privileged and/or confidential information and may not be disclosed, distributed or reproduced without the prior written permission of EPAM Systems.

- Регистрация в программе на Java:

```
java.sql.DriverManager.registerDriver (new com.ibm.as400.access.AS400JDBCDriver());
```

Пользователь может пропустить этот управляющий уровень JDBC и *вызывать непосредственно методы класса `Driver`* для открытия соединения.

Это может быть нужным в тех редких случаях, когда *два или более драйвера могут обслужить заданный URL*, но пользователь хочет выбрать конкретный из них.

## Установление связи с БД

Объект **Connection** представляет собой **соединение с БД**. Сессия соединения **включает в себя выполняемые SQL-запросы и возвращаемые через соединение результаты**.

Приложение может открыть одно или более соединений с одной или несколькими БД.

Класс **DriverManager** содержит список зарегистрированных классов **Driver** и обеспечивает управление ими, и при вызове метода **getConnection** он проверяет каждый драйвер и ищет среди них тот, который "умеет" соединиться с БД, указанной в URL. Метод **connect()** драйвера использует этот URL для установления соединения.

Вызов метода

**DriverManager.getConnection(...)** -

стандартный способ получения соединения Методу передается строка, содержащая "URL". Класс **DriverManager** пытается найти драйвер, который может соединиться к БД с помощью данного URL.

```
...
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
public class ConnectToDBExample {
    public static void main(String[] args) {
        Connection con = null;
        try {
            Class.forName("org.gjt.mm.mysql.Driver");
            con = DriverManager.getConnection("jdbc:mysql://127.0.0.1/test", "root",
"123456");
            System.out.println("Соединение установлено.");
        } catch (ClassNotFoundException e) {...}
        } catch (SQLException e) {...}
        } finally {
            try {
                if (con != null) {con.close();}
            } catch (SQLException e) {...}
        } } }
```

## JDBC-URL (Uniform Resource Locator)

Стандартный синтаксис JDBC URL:

`jdbc:<subprotocol>:<subname>`

### Legal Notice

This document contains privileged and/or confidential information and may not be disclosed, distributed or reproduced without the prior written permission of EPAM Systems.



- **jdbc - протокол.** Протокол, используемый в JDBC-URL - всегда jdbc.
- **<subprotocol> (подпротокол)** - это имя драйвера или имя механизма соединения с БД.
- **<subname> (подимя)** - это идентификатор БД.

Разработчик драйвера **резервирует имя подпротокола** в **JDBC-URL**. Когда класс **DriverManager** "показывает" это имя своему списку зарегистрированных драйверов, и тот драйвер, который отвечает за этот подпротокол, должен "откликнуться" и установит соединение с БД.

Например, **odbc** зарезервирован за мостом **JDBC-ODBC**. Кто-нибудь другой, например, Miracle Corporation, может зарегистрировать в качестве подпротокола "**miracle**" для jdbc-драйвера, который соединяется с **СУБД Miracle**. При этом никто другой уже не сможет использовать это имя.

## Выполнение SQL-запросов

В JDBC есть три класса для отправления **SQL-запросов** в БД и три метода в интерфейсе **Connection** определяют экземпляры этих классов:

- **Statement** - создается методом *createStatement*. Объект Statement используется при простых SQL-запросах.
- **PreparedStatement** - создается методом *prepareStatement*. Подготовленные sql-запросы.
- **CallableStatement** - создается методом *prepareCall*. Объекты CallableStatement используются для выполнения т.н. хранимых процедур - именованных групп SQL-запросов, наподобие вызова подпрограммы.

## Statement

Метод **createStatement** используется для простых SQL-выражений (без параметров).

```
package _java._se._07._connect;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
public class ExecuteQueryToDBExample {
    public static void main(String[] args) {
        Connection con = null;
        Statement st = null;
        ResultSet rs = null;
        try {
            Class.forName("org.gjt.mm.mysql.Driver");
            con = DriverManager.getConnection("jdbc:mysql://127.0.0.1/test",
                                             "root", "123456");
            st = con.createStatement();
            rs = st.executeQuery("SELECT * FROM STUDENTS");
            while (rs.next()) {
                System.out.println(rs.getInt(1) + " " + rs.getString(2) + " " +
                                   rs.getInt(3));
            }
        } catch (ClassNotFoundException e) {
```

### Legal Notice

This document contains privileged and/or confidential information and may not be disclosed, distributed or reproduced without the prior written permission of EPAM Systems.

```

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

package _java._se._07._connect;
...
public class ExecuteUpdateToDBExample {
    public static void main(String[] args) {
        Connection con = null;
        Statement st = null;
        ResultSet rs = null;
        try {
            Class.forName("org.gjt.mm.mysql.Driver");
            con = DriverManager.getConnection(
                "jdbc:mysql://127.0.0.1/test","root", "123456");
            st = con.createStatement();
            int countRows = st.executeUpdate(
                "INSERT INTO students (name, id_group) VALUES (\\"Баба-Яра\\",123456)");
            System.out.println(countRows);
        } ...
    }
}

```

Метод **executeUpdate** возвращает количество строк, полученных в результате выполнения **SQL-команды**. может применяться для выполнения команд **INSERT**, **UPDATE** и **DELETE**, а также команд определения данных **CREATE TABLE** и **DROP TABLE**.

Для выполнения команды **SELECT** нужно использовать другой метод, а именно **executeQuery**.

Существует также универсальный метод **execute**, который может применяться для выполнения **произвольных SQL-команд**, но он используется в основном для интерактивного создания запросов.

## ResultSet

Метод **executeQuery** возвращает объект типа **ResultSet** с построчными результатами выполнения запроса.

```
ResultSet rs = stat.executeQuery("SELECT * FROM Books");
```

Для построчного анализа результатов выполнения запроса используется приведенный ниже цикл.

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

### Legal Notice

This document contains privileged and/or confidential information and may not be disclosed, distributed or reproduced without the prior written permission of EPAM Systems.

При обработке отдельной строки нужно с помощью специальных методов получить содержимое каждого столбца.

```
String isbn = rs.getString(1);
float price = rs.getDouble("Price");
```

Для каждого типа данных языка Java предусмотрен отдельный метод извлечения данных, например **getString** и **getDouble**. Каждый из них имеет два способа представления, основанных на числовом и строковом типе аргумента.

При использовании числового аргумента метод извлечет данные из столбца с указанным аргументом-номером. Например, метод **rs.getString(1)** возвратит значение из первого столбца текущей строки.

При использовании строкового аргумента метод извлечет данные из столбца с указанным аргументом-именем.

Например, метод **rs.getDouble("Price")** возвратит значение из столбца с именем **Price**. Первый способ на основе числового аргумента более эффективен, но строковые аргументы позволяют создать более читабельный и простой для сопровождения код.

Каждый метод извлечения данных выполняет преобразование типа, если указанный тип не соответствует фактическому типу. Например, метод **rs.getString("Price")** преобразует число с плавающей запятой из столбца **Price** в строку.

Каждый метод извлечения данных выполняет преобразование типа, если указанный тип не соответствует фактическому типу. JDBC отображение типов:

JDBC Type	Java Type	JDBC Type	Java Type
BIT	boolean	NUMERIC, DECIMAL	BigDecimal
TINYINT	byte	DATE	java.sql.Date
SMALLINT	short	TIME, TIMESTAMP	java.sql.Time, java.sql.Timestamp
INTEGER	int	CLOB	Clob
BIGINT	long	BLOB	Blob
REAL	float	ARRAY	java.sql.array
FLOAT, DOUBLE	double	DISTINCT	mapping of underlying type
BINARY, VARBINARY, LONGVARBINARY	Byte[]	STRUCT	Struct
NCHAR, NVARCHAR, LONGVARCHAR	String	REF	Ref

Для организации прокрутки результатов выполнения запроса необходимо получить объект Statement с помощью приведенного ниже способа.

```
Statement stat = conn.createStatement(type, concurrency);
```

Для предварительно подготовленного запроса нужно использовать следующий вызов.

```
PreparedStatement stat = conn.prepareStatement(command, type, concurrency);
```

Для организации прокрутки результатов выполнения запроса без возможности редактирования данных можно использовать следующую команду.

```
Statement stat = conn.createStatement(
    ResultSet.TYPE_SCROLL_INSENSITIVE,
    ResultSet.CONCUR_READ_ONLY);
```

Значения параметра type	
TYPE_FORWARD_ONLY	Без прокрутки
TYPE_SCROLL_INSENSITIVE	С прокруткой, но без учета изменений в базе данных
TYPE_SCROLL_SENSITIVE	С прокруткой и с учетом изменений в базе данных

Значения параметра concurrency	
CONCUR_READ_ONLY	Без редактирования
CONCUR_UPDATABLE	С редактированием и обновлением базы данных

Методы интерфейса ResultSet

Метод	Описание
boolean first()	Перемещает курсор к первой строке результирующего набора
boolean isFirst()	Определяет, указывает ли курсор на первую строку результирующего набора данных
boolean beforeFirst()	Перемещает курсор перед первой строкой результирующего набора данных
boolean last()	Перемещает курсор к последней строке результирующего набора данных

<code>boolean isLast()</code>	Определяет, установлен ли курсор на последнюю строку результирующего набора данных.
<code>int getRow()</code>	Номер строки, на которой установлен курсор
<code>boolean afterLast()</code>	Перемещает курсор после последней строки результирующего набора данных
<code>boolean isAfterLast()</code>	Определяет, находится ли курсор после последней строки результирующего набора данных
<code>boolean previous()</code>	Перемещает курсор на предыдущую строку результирующего набора данных
<code>boolean absolute(int i)</code>	Перемещает курсор к строке, номер которой указан как параметр
<code>boolean relative(int i)</code>	Перемещает курсор вперед или назад на количество строк, указанное в параметре, относительно текущего положения курсора

При попытке перемещения курсора за пределы имеющегося результата выполнения запроса он располагается либо после последней, либо перед первой записью в зависимости от направления перемещения.

Методы, используемые с обновляемым набором данных:

Метод	Описание
<code>void updateRow()</code>	Обновляет текущую строку объекта <code>ResultSet</code> и базовую таблицу базы данных
<code>void insertRow()</code>	Обновляет текущую строку объекта <code>ResultSet</code> и базовую таблицу базы данных
<code>void updateString()</code>	Обновляет специфицированный столбец, заданный строковым значением
<code>void updateInt()</code>	Обновляет специфицированный столбец, заданный целым значением

Не все запросы возвращают обновляемый набор результатов запроса. Например, запрос с соединением нескольких таблиц не всегда может быть обновляемым. Это всегда возможно только для запросов на основе одной таблицы или запросов на основе соединения нескольких таблиц по первичным ключам. Для проверки текущего режима работы рекомендуется использовать метод **getConcurrency** интерфейса **ResultSet**.

#### Legal Notice

This document contains privileged and/or confidential information and may not be disclosed, distributed or reproduced without the prior written permission of EPAM Systems.

```
String query = "SELECT * FROM Books";
ResultSet rs = stat.executeQuery(query);
while (rs.next()){
    if (...){
        double increase = ...
        double price = rs.getDouble("Price");
        rs.updateDouble("Price", price + increase);
        rs.updateRow();
    }
}
```

Методы **updateXxx** изменяют только отдельные значения в текущей строке в результатах выполнения запроса, а не в базе данных. Для обновления всех данных из отредактированной строки в базе данных нужно вызвать метод **updateRow**.

Для отмены обновлений из данной строки в базе данных можно использовать метод **cancelRowUpdates**.

## PreparedStatement

Метод **prepareStatement** используется для SQL-выражений с одним или более входным (IN-) параметром простых SQL-выражений, которые исполняются часто.

Для компиляции SQL запроса, в котором отсутствуют конкретные значения, 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()**.

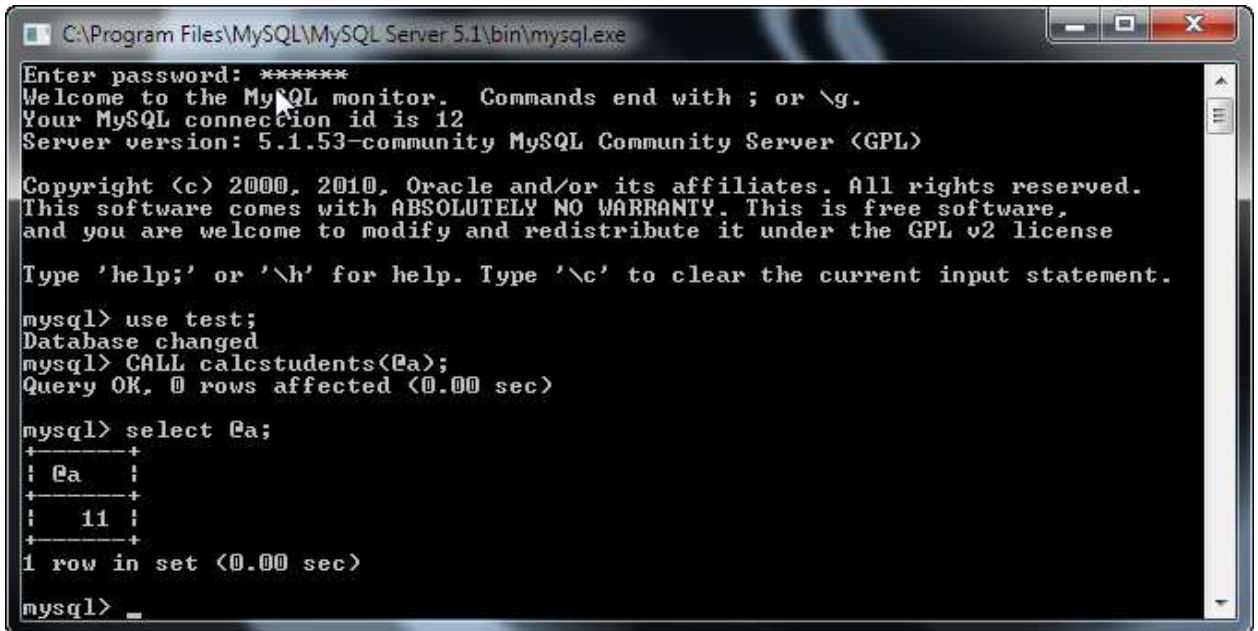
**PreparedStatement** - оператор предварительно откомпилирован, поэтому он выполняется быстрее обычных операторов ему соответствующих.

```
public class ExecutePreparedStatement {
    public static void main(String[] args) {
        Connection con = null;
        Statement st = null;
        ResultSet rs = null;
        try {
            Class.forName("org.gjt.mm.mysql.Driver");
            con = DriverManager.getConnection(
                "jdbc:mysql://127.0.0.1/test", "root", "123456");
            String sql = "INSERT INTO students(name,id_group) VALUES(?,?)";
            PreparedStatement ps = con.prepareStatement(sql);
            ps.setString(1, "Кукушнин");
            ps.setInt(2, 851001);
            ps.executeUpdate();
        } ...
    } ...
}
```

## CallableStatement

```
DELIMITER //
CREATE PROCEDURE calcstudents (OUT count INT)
BEGIN
```

```
SELECT COUNT(*) INTO count FROM students;
END;
```



```
C:\Program Files\MySQL\MySQL Server 5.1\bin\mysql.exe
Enter password: *****
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 12
Server version: 5.1.53-community MySQL Community Server (GPL)

Copyright (c) 2000, 2010, Oracle and/or its affiliates. All rights reserved.
This software comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to modify and redistribute it under the GPL v2 license

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> use test;
Database changed
mysql> CALL calcstudents(@a);
Query OK, 0 rows affected (0.00 sec)

mysql> select @a;
+-----+
| @a    |
+-----+
|      11      |
+-----+
1 row in set (0.00 sec)

mysql> _
```

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 выходящие (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()**.

Метод **prepareCall** используется для вызова хранимой процедуры.

```
Connection con = null;
CallableStatement cs = null;
ResultSet rs = null;
try {
    Class.forName("org.gjt.mm.mysql.Driver");
    con = DriverManager.getConnection(
        "jdbc:mysql://127.0.0.1/test","root", "123456");
    cs = con.prepareCall("{call calcstudents(?)}");
    cs.registerOutParameter(1, java.sql.Types.INTEGER);
    cs.execute();
    int empName = cs.getInt(1);
    System.out.println("Count students - " + empName);
} ...
```



## Batch-команды

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

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

```
st = con.createStatement();
st.addBatch("INSERT INTO students (name, id_group) VALUES (\\"Пушкин\\", 123456)");
st.addBatch("INSERT INTO students (name, id_group) VALUES (\\"Лермонтов\\", 123456)");
st.addBatch("INSERT INTO students (name, id_group) VALUES (\\"Ломоносов\\", 123456)");
// submit a batch of update commands for execution
int[] updateCounts = st.executeBatch();
```

## Закрытие ResultSet, Statement и Connection

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

## Connection Pool

db.properties

```
db.driver = oracle.jdbc.driver.OracleDriver
db.url = jdbc:oracle:thin:@127.0.0.1:1521:xe
db.user = hr
db.password = hr2
db.poolsize = 5
```

DBResourceManager.java

```
package _java._se._07_connectionpool;
import java.util.ResourceBundle;
public class DBResourceManager {
    private final static DBResourceManager instance = new DBResourceManager();

    private ResourceBundle bundle =
ResourceBundle.getBundle("_java._se._07._connectionpool.db");

    public static DBResourceManager getInstance() {
        return instance;
    }

    public String getValue(String key){
        return bundle.getString(key);
    }
}
```

DBParameter.java

```
package _java._se._07_connectionpool;
```

### Legal Notice

This document contains privileged and/or confidential information and may not be disclosed, distributed or reproduced without the prior written permission of EPAM Systems.



```
public final class DBParameter {
    private DBParameter() {}

    public static final String DB_DRIVER = "db.driver";
    public static final String DB_URL = "db.url";
    public static final String DB_USER = "db.user";
    public static final String DB_PASSWORD = "db.password";
    public static final String DB_POLL_SIZE = "db.poolsize";
}
```

### ConnectionPoolException.java

```
package _java._se._07_connectionpool;
public class ConnectionPoolException extends Exception {
    private static final long serialVersionUID = 1L;

    public ConnectionPoolException(String message, Exception e){
        super(message, e);
    }
}
```

### ConnectionPool.java

```
package _java._se._07_connectionpool;
import java.sql.Array;
import java.sql.Blob;
import java.sql.CallableStatement;
import java.sql.Clob;
import java.sql.Connection;
import java.sql.DatabaseMetaData;
import java.sql.DriverManager;
import java.sql.NClob;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.SQLClientInfoException;
import java.sql.SQLException;
import java.sql.SQLWarning;
import java.sql.XML;
import java.sql.Savepoint;
import java.sql.Statement;
import java.util.Locale;
import java.util.Map;
import java.util.Properties;
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.Executor;

public final class ConnectionPool {

    private BlockingQueue<Connection> connectionQueue;
    private BlockingQueue<Connection> givenAwayConQueue;

    private String driverName;
    private String url;
    private String user;
    private String password;
    private int poolSize;

    private ConnectionPool() {
        DBResourceManager dbResourceManager = DBResourceManager.getInstance();
        this.driverName = dbResourceManager.getValue(DBParameter.DB_DRIVER);
    }
}
```

### Legal Notice

This document contains privileged and/or confidential information and may not be disclosed, distributed or reproduced without the prior written permission of EPAM Systems.

```

        this.url = dbResourceManager.getValue(DBParameter.DB_URL);
        this.user = dbResourceManager.getValue(DBParameter.DB_USER);
        ;
        this.password = dbResourceManager.getValue(DBParameter.DB_PASSWORD);

        try {
            this.poolSize = Integer.parseInt(dbResourceManager
                .getValue(DBParameter.DB_POOL_SIZE));
        } catch (NumberFormatException e) {
            poolSize = 5;
        }
    }

    public void initPoolData() throws ConnectionPoolException {
        Locale.setDefault(Locale.ENGLISH);

        try {
            Class.forName(driverName);
            givenAwayConQueue = new ArrayBlockingQueue<Connection>(poolSize);
            connectionQueue = new ArrayBlockingQueue<Connection>(poolSize);
            for (int i = 0; i < poolSize; i++) {
                Connection connection = DriverManager.getConnection(url,
                    user,
                        password);
                PooledConnection pooledConnection = new PooledConnection(
                    connection);
                connectionQueue.add(pooledConnection);
            }
        } catch (SQLException e) {
            throw new ConnectionPoolException("SQLException in
                ConnectionPool",
                    e);
        } catch (ClassNotFoundException e) {
            throw new ConnectionPoolException(
                "Can't find database driver class", e);
        }
    }

    public void dispose() {
        clearConnectionQueue();
    }

    private void clearConnectionQueue() {
        try {
            closeConnectionsQueue(givenAwayConQueue);
            closeConnectionsQueue(connectionQueue);
        } catch (SQLException e) {
            // logger.log(Level.ERROR, "Error closing the connection.", e);
        }
    }

    public Connection takeConnection() throws ConnectionPoolException {
        Connection connection = null;
        try {
            connection = connectionQueue.take();
            givenAwayConQueue.add(connection);
        } catch (InterruptedException e) {
            throw new ConnectionPoolException(
                "Error connecting to the data source.", e);
        }
        return connection;
    }
}

```

**Legal Notice**

This document contains privileged and/or confidential information and may not be disclosed, distributed or reproduced without the prior written permission of EPAM Systems.

```

    public void closeConnection(Connection con, Statement st, ResultSet rs) {
        try {
            con.close();
        } catch (SQLException e) {
            // logger.log(Level.ERROR, "Connection isn't return to the
pool.");
        }

        try {
            rs.close();
        } catch (SQLException e) {
            // logger.log(Level.ERROR, "ResultSet isn't closed.");
        }

        try {
            st.close();
        } catch (SQLException e) {
            // logger.log(Level.ERROR, "Statement isn't closed.");
        }
    }

    public void closeConnection(Connection con, Statement st) {
        try {
            con.close();
        } catch (SQLException e) {
            // logger.log(Level.ERROR, "Connection isn't return to the
pool.");
        }

        try {
            st.close();
        } catch (SQLException e) {
            // logger.log(Level.ERROR, "Statement isn't closed.");
        }
    }

    private void closeConnectionsQueue(BlockingQueue<Connection> queue)
        throws SQLException {
        Connection connection;
        while ((connection = queue.poll()) != null) {
            if (!connection.getAutoCommit()) {
                connection.commit();
            }
            ((PooledConnection) connection).reallyClose();
        }
    }

    private class PooledConnection implements Connection {
        private Connection connection;

        public PooledConnection(Connection c) throws SQLException {
            this.connection = c;
            this.connection.setAutoCommit(true);
        }

        public void reallyClose() throws SQLException {
            connection.close();
        }

        @Override
        public void clearWarnings() throws SQLException {
            connection.clearWarnings();
        }
    }

```

```

    }

    @Override
    public void close() throws SQLException {
        if (connection.isClosed()) {
            throw new SQLException("Attempting to close closed
connection.");
        }

        if (connection.isReadOnly()) {
            connection.setReadOnly(false);
        }

        if (!givenAwayConQueue.remove(this)) {
            throw new SQLException(
                "Error deleting connection from the given away
connections pool.");
        }

        if (!connectionQueue.offer(this)) {
            throw new SQLException(
                "Error allocating connection in the pool.");
        }
    }

    @Override
    public void commit() throws SQLException {
        connection.commit();
    }

    @Override
    public Array createArrayOf(String typeName, Object[] elements)
        throws SQLException {
        return connection.createArrayOf(typeName, elements);
    }

    @Override
    public Blob createBlob() throws SQLException {
        return connection.createBlob();
    }

    @Override
    public Clob createClob() throws SQLException {
        return connection.createClob();
    }

    @Override
    public NClob createNClob() throws SQLException {
        return connection.createNClob();
    }

    @Override
    public SQLXML createSQLXML() throws SQLException {
        return connection.createSQLXML();
    }

    @Override
    public Statement createStatement() throws SQLException {
        return connection.createStatement();
    }

    @Override
    public Statement createStatement(int resultSetType,

```

**Legal Notice**

This document contains privileged and/or confidential information and may not be disclosed, distributed or reproduced without the prior written permission of EPAM Systems.

```
        int resultSetConcurrency) throws SQLException {
    return connection.createStatement(resultSetType,
        resultSetConcurrency);
}

@Override
public Statement createStatement(int resultSetType,
    int resultSetConcurrency, int resultSetHoldability)
    throws SQLException {
    return connection.createStatement(resultSetType,
        resultSetConcurrency, resultSetHoldability);
}

@Override
public Struct createStruct(String typeName, Object[] attributes)
    throws SQLException {
    return connection.createStruct(typeName, attributes);
}

@Override
public boolean getAutoCommit() throws SQLException {
    return connection.getAutoCommit();
}

@Override
public String getCatalog() throws SQLException {
    return connection.getCatalog();
}

@Override
public Properties getClientInfo() throws SQLException {
    return connection.getClientInfo();
}

@Override
public String getClientInfo(String name) throws SQLException {
    return connection.getClientInfo(name);
}

@Override
public int getHoldability() throws SQLException {
    return connection.getHoldability();
}

@Override
public DatabaseMetaData getMetaData() throws SQLException {
    return connection.getMetaData();
}

@Override
public int getTransactionIsolation() throws SQLException {
    return connection.getTransactionIsolation();
}

@Override
public Map<String, Class<?>> getTypeMap() throws SQLException {
    return connection.getTypeMap();
}

@Override
public SQLWarning getWarnings() throws SQLException {
    return connection.getWarnings();
}
```

```
@Override
public boolean isClosed() throws SQLException {
    return connection.isClosed();
}

@Override
public boolean isReadOnly() throws SQLException {
    return connection.isReadOnly();
}

@Override
public boolean isValid(int timeout) throws SQLException {
    return connection.isValid(timeout);
}

@Override
public String nativeSQL(String sql) throws SQLException {
    return connection.nativeSQL(sql);
}

@Override
public CallableStatement prepareCall(String sql) throws SQLException {
    return connection.prepareCall(sql);
}

@Override
public CallableStatement prepareCall(String sql, int resultSetType,
    int resultSetConcurrency) throws SQLException {
    return connection.prepareCall(sql, resultSetType,
        resultSetConcurrency);
}

@Override
public CallableStatement prepareCall(String sql, int resultSetType,
    int resultSetConcurrency, int resultSetHoldability)
    throws SQLException {
    return connection.prepareCall(sql, resultSetType,
        resultSetConcurrency, resultSetHoldability);
}

@Override
public PreparedStatement prepareStatement(String sql)
    throws SQLException {
    return connection.prepareStatement(sql);
}

@Override
public PreparedStatement prepareStatement(String sql,
    int autoGeneratedKeys) throws SQLException {
    return connection.prepareStatement(sql, autoGeneratedKeys);
}

@Override
public PreparedStatement prepareStatement(String sql,
    int[] columnIndexes) throws SQLException {
    return connection.prepareStatement(sql, columnIndexes);
}

@Override
public PreparedStatement prepareStatement(String sql,
    String[] columnNames) throws SQLException {
    return connection.prepareStatement(sql, columnNames);
}
```

**Legal Notice**

This document contains privileged and/or confidential information and may not be disclosed, distributed or reproduced without the prior written permission of EPAM Systems.

```
}

@Override
public PreparedStatement prepareStatement(String sql,
    int resultSetType, int resultSetConcurrency)
    throws SQLException {
    return connection.prepareStatement(sql, resultSetType,
        resultSetConcurrency);
}

@Override
public PreparedStatement prepareStatement(String sql,
    int resultSetType, int resultSetConcurrency,
    int resultSetHoldability) throws SQLException {
    return connection.prepareStatement(sql, resultSetType,
        resultSetConcurrency, resultSetHoldability);
}

@Override
public void rollback() throws SQLException {
    connection.rollback();
}

@Override
public void setAutoCommit(boolean autoCommit) throws SQLException {
    connection.setAutoCommit(autoCommit);
}

@Override
public void setCatalog(String catalog) throws SQLException {
    connection.setCatalog(catalog);
}

@Override
public void setClientInfo(String name, String value)
    throws SQLClientInfoException {
    connection.setClientInfo(name, value);
}

@Override
public void setHoldability(int holdability) throws SQLException {
    connection.setHoldability(holdability);
}

@Override
public void setReadOnly(boolean readOnly) throws SQLException {
    connection.setReadOnly(readOnly);
}

@Override
public Savepoint setSavepoint() throws SQLException {
    return connection.setSavepoint();
}

@Override
public Savepoint setSavepoint(String name) throws SQLException {
    return connection.setSavepoint(name);
}

@Override
public void setTransactionIsolation(int level) throws SQLException {
    connection.setTransactionIsolation(level);
}
```

**Legal Notice**

This document contains privileged and/or confidential information and may not be disclosed, distributed or reproduced without the prior written permission of EPAM Systems.

```
@Override
public boolean isWrapperFor(Class<?> iface) throws SQLException {
    return connection.isWrapperFor(iface);
}

@Override
public <T> T unwrap(Class<T> iface) throws SQLException {
    return connection.unwrap(iface);
}

@Override
public void abort(Executor arg0) throws SQLException {
    connection.abort(arg0);
}

@Override
public int getNetworkTimeout() throws SQLException {
    return connection.getNetworkTimeout();
}

@Override
public String getSchema() throws SQLException {
    return connection.getSchema();
}

@Override
public void releaseSavepoint(Savepoint arg0) throws SQLException {
    connection.releaseSavepoint(arg0);
}

@Override
public void rollback(Savepoint arg0) throws SQLException {
    connection.rollback(arg0);
}

@Override
public void setClientInfo(Properties arg0)
    throws SQLClientInfoException {
    connection.setClientInfo(arg0);
}

@Override
public void setNetworkTimeout(Executor arg0, int arg1)
    throws SQLException {
    connection.setNetworkTimeout(arg0, arg1);
}

@Override
public void setSchema(String arg0) throws SQLException {
    connection.setSchema(arg0);
}

@Override
public void setTypeMap(Map<String, Class<?>> arg0) throws SQLException {
    connection.setTypeMap(arg0);
}
}
```



## DATA ACCESS OBJECT (DAO)

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

Источником данных может быть реляционное хранилище (например, RDBMS), внешняя служба (например, B2B-биржа), репозиторий (LDAP-база данных), или бизнес-служба, обращение к которой осуществляется при помощи протокола CORBA Internet Inter-ORB Protocol (IIOP) или низкоуровневых сокетов.

Использующие DAO бизнес-компоненты работают с более простым интерфейсом, предоставляемым объектом DAO своим клиентам. **DAO полностью скрывает детали реализации** источника данных от клиентов.

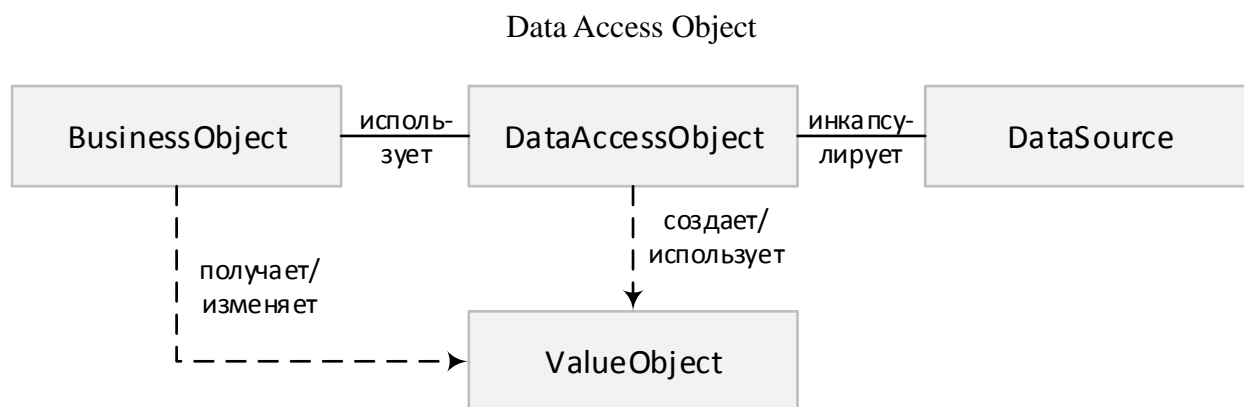
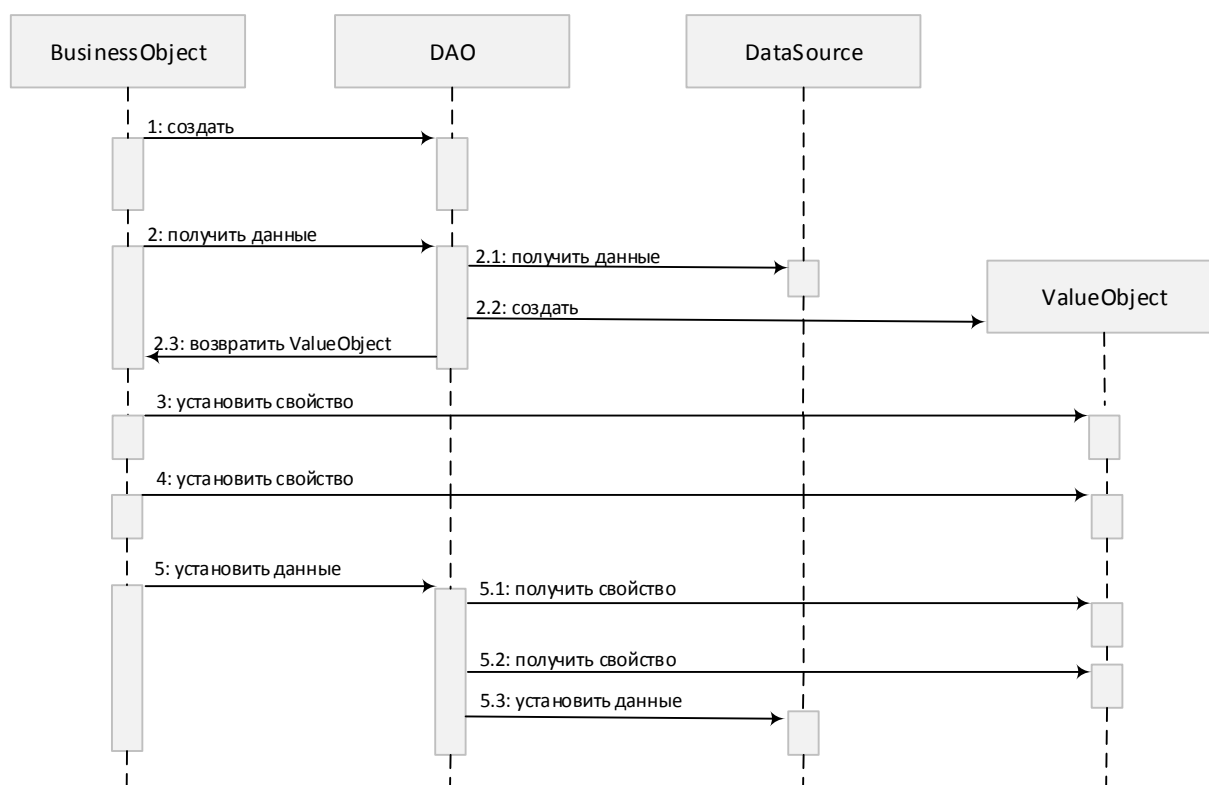


Диаграмма последовательности действий паттерна Data Access Object



### Legal Notice

This document contains privileged and/or confidential information and may not be disclosed, distributed or reproduced without the prior written permission of EPAM Systems.

### Диаграмма классов при применении стратегии **Factory for Data Access Objects**

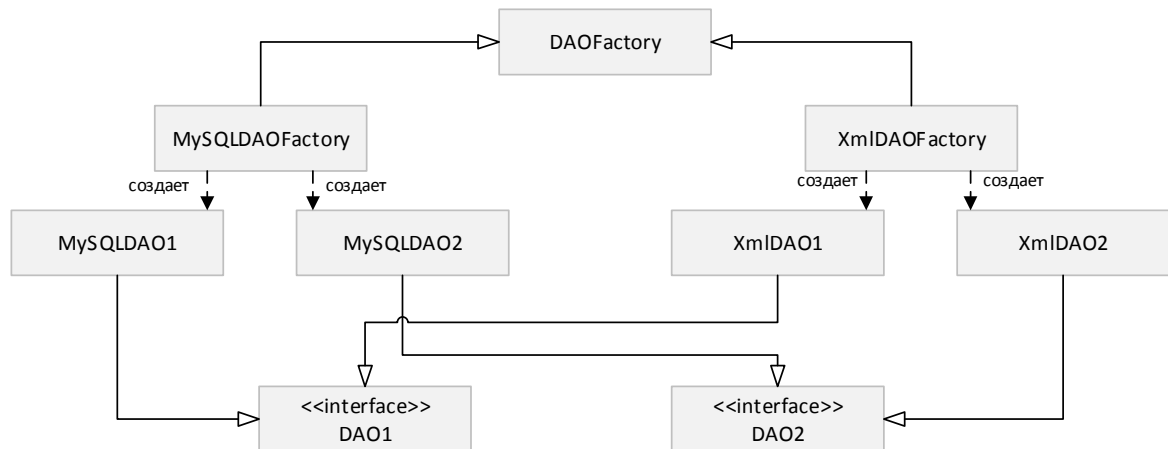
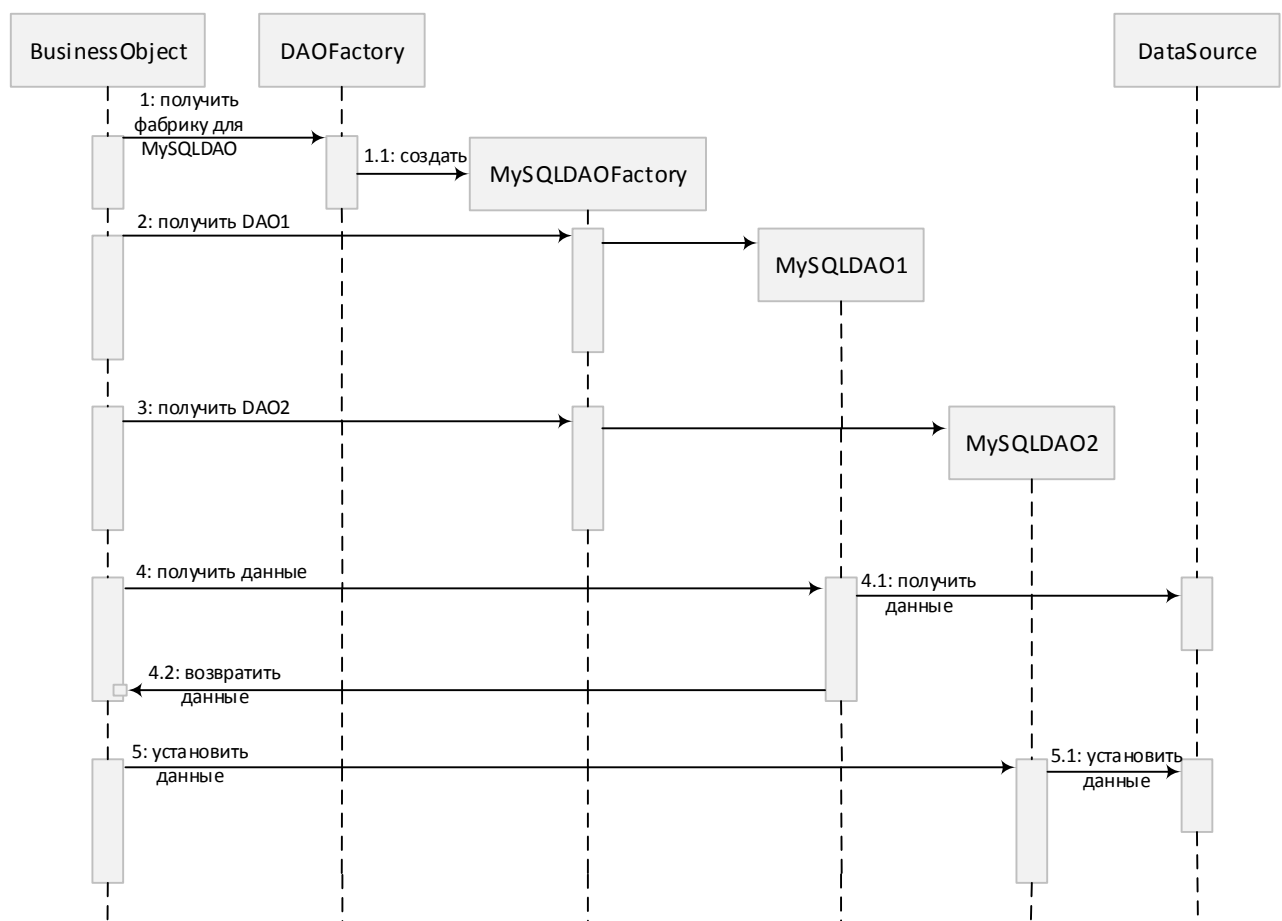


Диаграмма последовательности действий для стратегии Factory for Data Access Objects, использующей Abstract Factory.



## Legal Notice

This document contains privileged and/or confidential information and may not be disclosed, distributed or reproduced without the prior written permission of EPAM Systems.

## Транзакции и точки сохранения

Транзакция состоит из одного или более выражений (действий), которые после выполнения либо все **фиксируются (commit)**, либо все **откатываются назад (rollback)**.

Для работы с транзакциями используются методы

- **commit()**
- **rollback()**

При вызове метода **commit()** или **rollback()** текущая транзакция заканчивается и начинается другая.

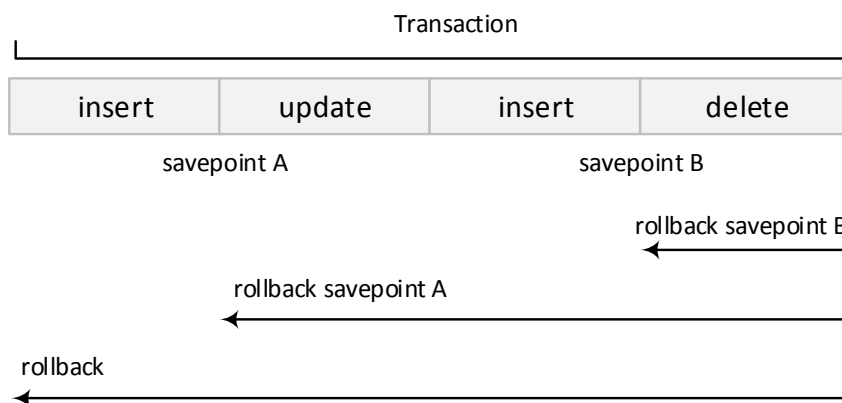
Каждое новое соединение по умолчанию находится в режиме автофиксации (**auto-commit**), что означает автоматическую фиксацию (**commit**) транзакции после каждого запроса. В этом случае транзакция состоит из одного запроса.

Если **auto-commit** запрещен, транзакция не заканчивается вплоть до явного вызова **commit** или **rollback**, включая, таким образом, все выражения, выполненные с момента последнего вызова **commit** или **rollback**. В этом случае все SQL-запросы в транзакции фиксируются или откатываются группой.

Метод фиксации **commit** делает окончательными все изменения в БД, сделанные SQL-выражением, и снимает также все блокировки, установленные транзакцией. Метод **rollback** проигнорирует, "отбракует" эти изменения.

Начиная с версии 3.0, JDBC поддерживает **точки сохранения**.

Интерфейс **Savepoint** позволяет разделить транзакцию на логические блоки, дающие возможность откатывать совершённые изменения не к последнему вызову **commit()**, а лишь к заранее установленной точке сохранения.



```
cn.setAutoCommit(false);
...
Statement st = cn.createStatement();
int rows = st.executeUpdate("INSERT INTO Employees " +
    "(FirstName, LastName) VALUES " + "('Игорь', 'Цветков')");
// Устанавливаем именнованную точку сохранения.
Savepoint svpt = cn.setSavepoint("NewEmp");
// ...
rows = st.executeUpdate("UPDATE Employees
    set Address = 'ул. Седых, 19-34' " +
    "WHERE LastName = 'Цветков'");
```

### Legal Notice

This document contains privileged and/or confidential information and may not be disclosed, distributed or reproduced without the prior written permission of EPAM Systems.

```
// ...
cn.rollback(svpt);
// ...
// Запись о работнике вставлена, но адрес не обновлен.
conn.commit();
```

Для транзакций существует несколько типов чтения:

- **Грязное чтение (dirty reads)** происходит, когда транзакциям разрешено видеть несохраненные изменения данных. Иными словами, изменения, сделанные в одной транзакции, видны вне ее до того, как она была сохранена. Если изменения не будут сохранены, то, вероятно, другие транзакции выполняли работу на основе некорректных данных;

- **Непроверяющееся чтение (nonrepeatable reads)** происходит, когда транзакция А читает строку, транзакция Б изменяет эту строку, транзакция А читает ту же строку и получает обновленные данные;

- **Фантомное чтение (phantom reads)** происходит, когда транзакция А считывает все строки, удовлетворяющие WHERE-условию, транзакция Б вставляет новую или удаляет одну из строк, которая удовлетворяет этому условию, транзакция А еще раз считывает все строки, удовлетворяющие WHERE-условию, уже вместе с новой строкой или недосчитавшись старой.

Интерфейс **DatabaseMetaData** предоставляет информацию об уровнях изолированности транзакций, которые поддерживаются данной СУБД.

Интерфейс **Connection** позволяет установить нужный уровень изолированности транзакций

```
cn.setTransactionIsolation(TRANSACTION_READ_UNCOMMITTED);
```

JDBC удовлетворяет четырем уровням изоляции транзакций, определенным в стандарте SQL:2003

Константы интерфейса Connection, определяющие уровень изоляции транзакции	Описание
<b>TRANSACTION_NONE</b>	информирует о том, что драйвер не поддерживает транзакции
<b>TRANSACTION_READ_UNCOMMITTED</b>	позволяет транзакциям видеть несохраненные изменения данных, что разрешает грязное, непроверяющееся и фантомное чтения
<b>TRANSACTION_READ_COMMITTED</b>	означает, что любое изменение, сделанное в транзакции, не видно вне неё, пока она не сохранена. Это предотвращает грязное чтение, но разрешает непроверяющееся и фантомное

<b>TRANSACTION_REPEATABLE_READ</b>	запрещает грязное и непроверяющееся, но фантомное чтение разрешено
<b>TRANSACTION_SERIALIZABLE</b>	определяет, что грязное, непроверяющееся и фантомное чтения запрещены

## Метаданные

В языке SQL данные о структуре базы данных и ее составных частей называются метаданными (metadata), чтобы их можно было отличить от основных данных.

Существуют метаданные двух типов: для **описания структуры базы данных** и **структуры результатов выполнения запроса**.

Доступ к этим дополнительным данным разработчики JDBC обеспечили через интерфейсы **ResultSetMetaData** и **DatabaseMetaData**.

Интерфейс **ResultSetMetaData** позволяет узнать:

- Число колонок в результирующем наборе.
- Является ли NULL допустимым значением в колонке.
- Метку, используемую для заголовка колонки.
- Имя заданной колонки.
- Таблицу, служащую источником данных для данной колонки.
- Тип данных колонки.

```
public class MetaDataResultSetExample {
    public static ResultSet executeSQL(){
        ...
        return rs;
    }
    public static void main(String[] args) throws SQLException {
        ResultSet rs = executeSQL();
        ResultSetMetaData meta = rs.getMetaData();

        int iColumnCount = meta.getColumnCount();

        for (int i = 1 ; i <= iColumnCount ; i++){
            System.out.println("Column Name: "+meta.getColumnName(i));
            System.out.println("Column Type: "+meta.getColumnType(i));
            System.out.println("Display Size: "+meta.getColumnDisplaySize(i));
            System.out.println("Precision: "+meta.getPrecision(i));
            System.out.println("Scale: "+meta.getScale(i) );
        }
    }
}
```

Получить объект **DatabaseMetaData** можно следующим образом:

```
DatabaseMetaData dbMetaData = cn.getMetaData();
```

В результате из полученного объекта **DatabaseMetaData** можно извлечь:

### Legal Notice

This document contains privileged and/or confidential information and may not be disclosed, distributed or reproduced without the prior written permission of EPAM Systems.

- название и версию СУБД методами **getDatabaseProductName()**, **getDatabaseProductVersion()**,
- название и версию драйвера - методами **getDriverName()**, **getDriverVersion()**,
- имя драйвера JDBC – методом **getDriverName()**,
- имя пользователя БД – методом **getUserName()**,
- местонахождение источника данных – методом **getURL()**

```
Connection con = null;
Statement st = null;
ResultSet rs = null;
DatabaseMetaData meta = null;
try {
    Class.forName("org.gjt.mm.mysql.Driver");
    con = DriverManager.getConnection(
        "jdbc:mysql://127.0.0.1/test", "root", "123456");
    ...
    meta = con.getMetaData();
    rs = meta.getTables(null, null, null, new String[]
        { "TABLE" });
    rs.next();
    System.out.println(rs.getString(3));
} ...
```