Java.SE.10

JAVA DATABASE CONNECTIVITY

Author: **Olga Smolyakova**
Oracle Certified Java 6 Programmer
Olga_Smolyakova@epam.com

Содержание

ЧТО ТАКОЕ JDBC

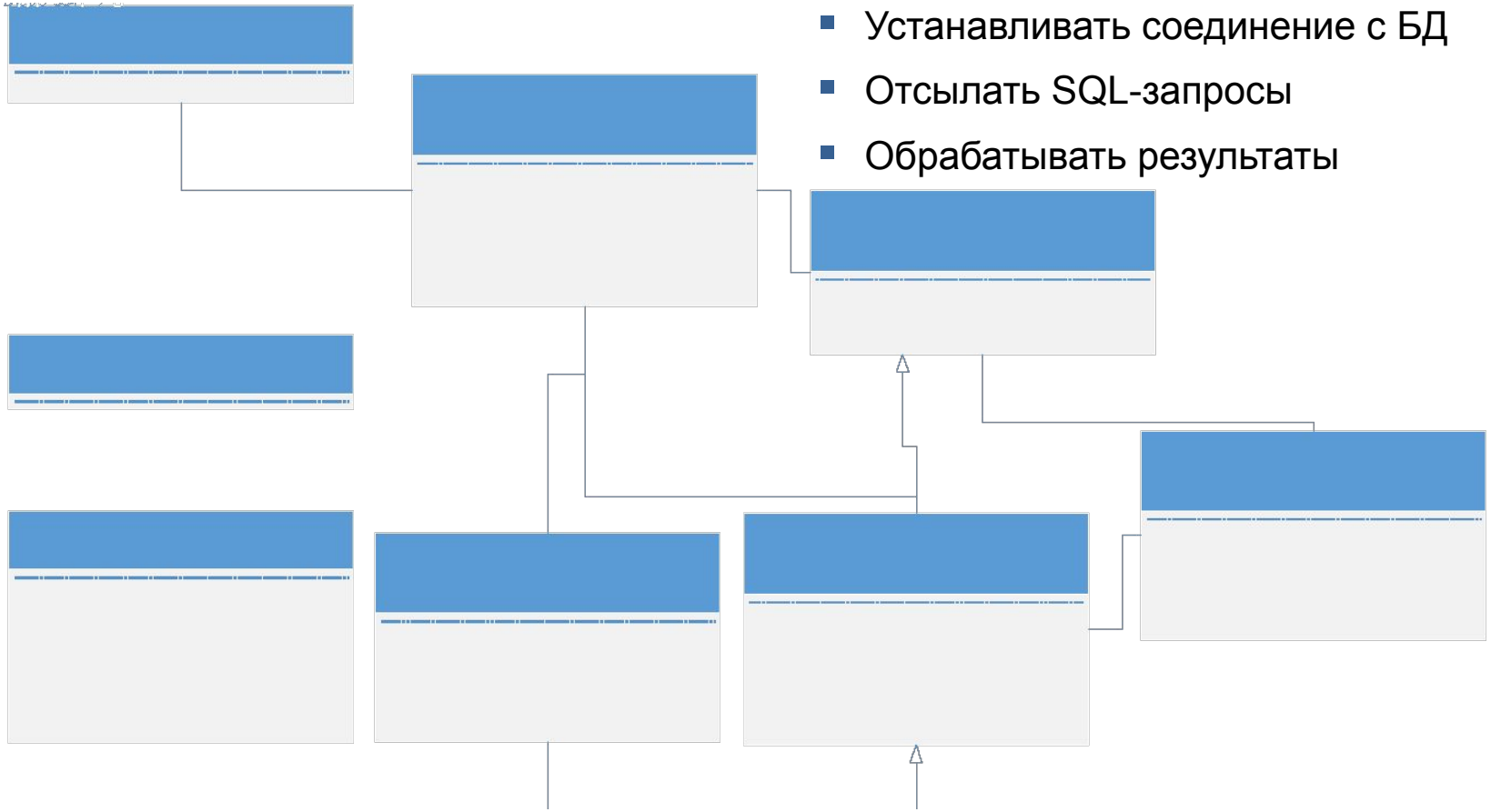
Что такое JDBC

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

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

Что такое JDBC

Основные интерфейсы и классы JDBC



Что может JDBC?

- Устанавливать соединение с БД
- Отсылать SQL-запросы
- Обрабатывать результаты

Что такое JDBC

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.

Что такое JDBC

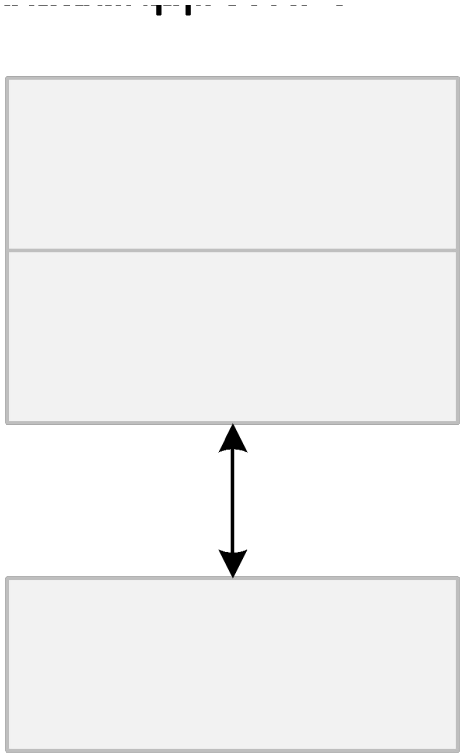
Версии JDBC.

Year	JDBC Version	JSR Specification	JDK Implementation
2017	JDBC 4.3	JSR 221	Java SE 9
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

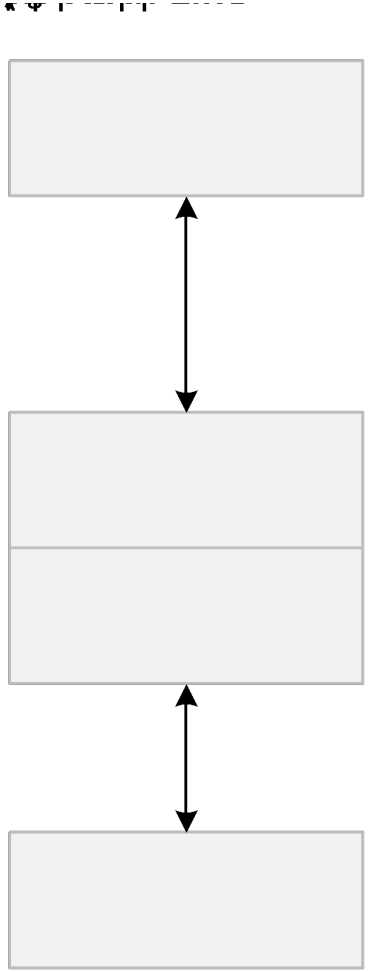
Модели доступа к БД

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



В двухзвенной модели приложение или апплет на языке Java обращается непосредсвенно к БД. В этом случае JDBC-драйвер "умеет" общаться с соответствующей СУБД. SQL-запросы отсылаются в СУБД, а результаты отсылаются обратно к пользователю.

Модели доступа к БД



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.

ТИПЫ ДРАЙВЕРОВ

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

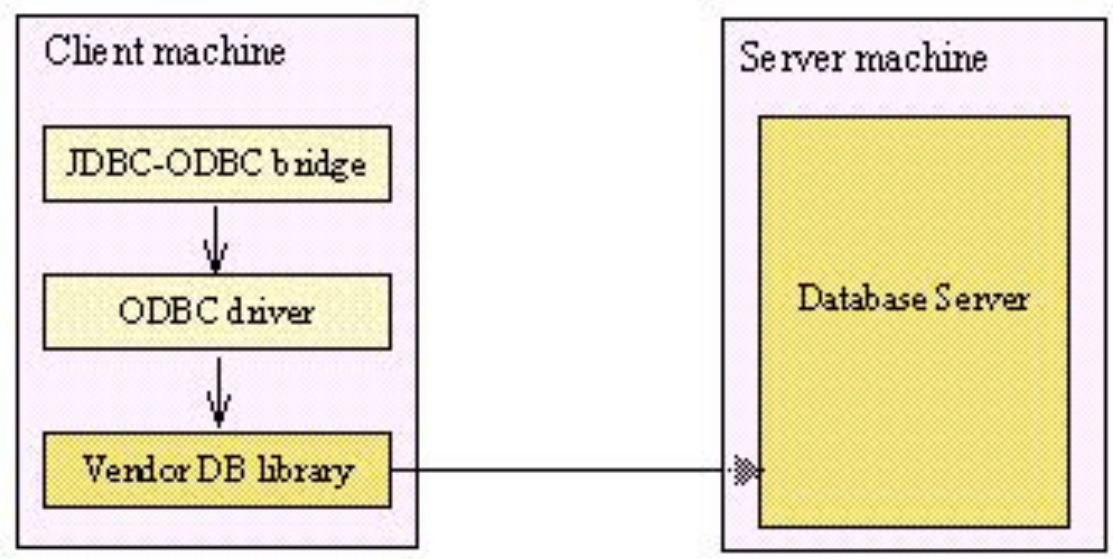
- **Запрос**
  - SQL запрос на выборку или изменение данных
- **Результат**
  - Логическое множество строк и столбцов таблицы базы данных
- **Метаданные**
  - Сведения о полученном результате и об используемой базе данных

Типы драйверов

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

Мост JDBC-ODBC

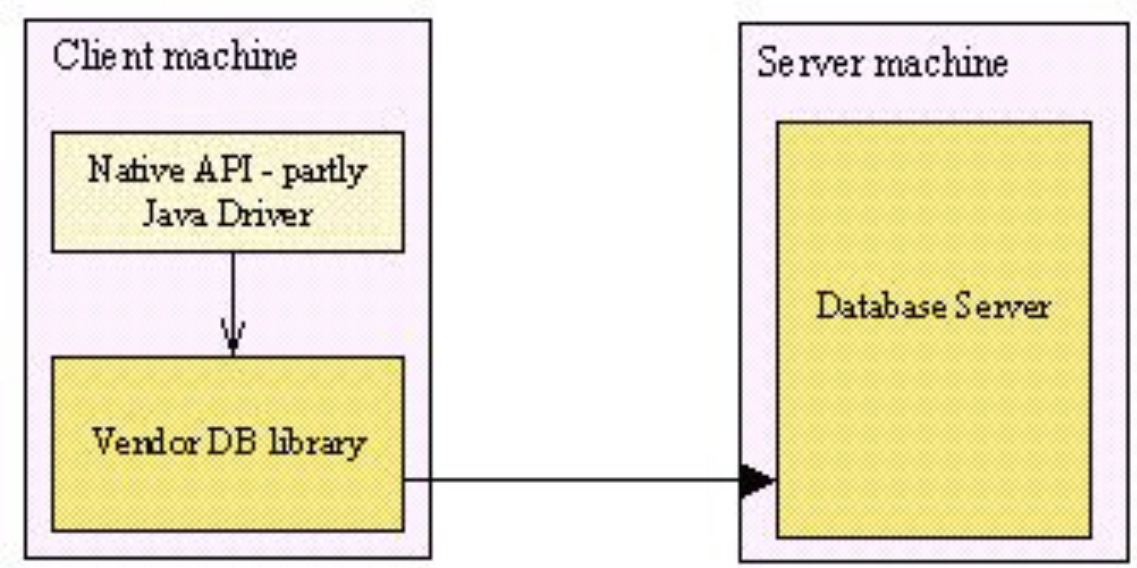
Драйверы 1-го типа транслируют все вызовы JDBC в вызовы ODBC (Open Database Connectivity), с пересылкой всех данных в ODBC драйвер.



Типы драйверов

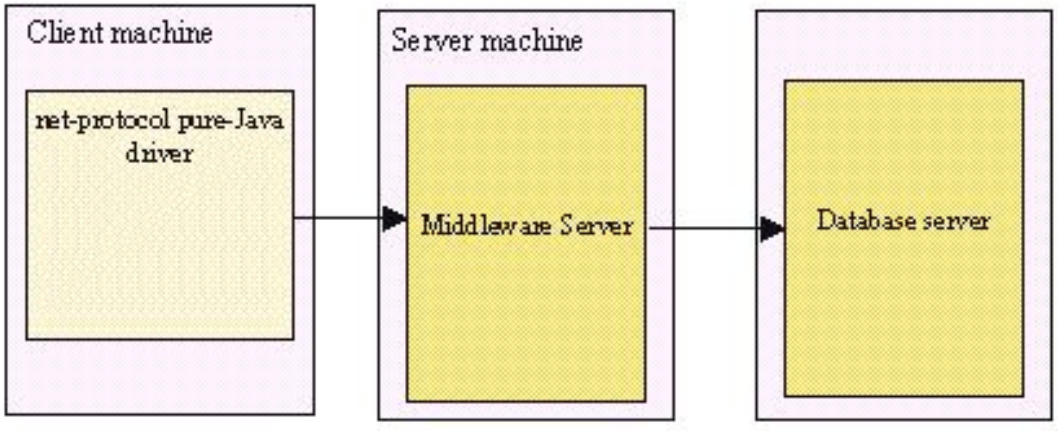
Нативный-API/частичный Java драйвер

JDBC драйвер 2-го типа - нативный-API/частичный Java драйвер – переводит вызовы JDBC в вызовы специфичные к СУБД таких как например SQL Server, Informix, Oracle или Sybase.



Сетевой протокол/«чистый» Java драйвер

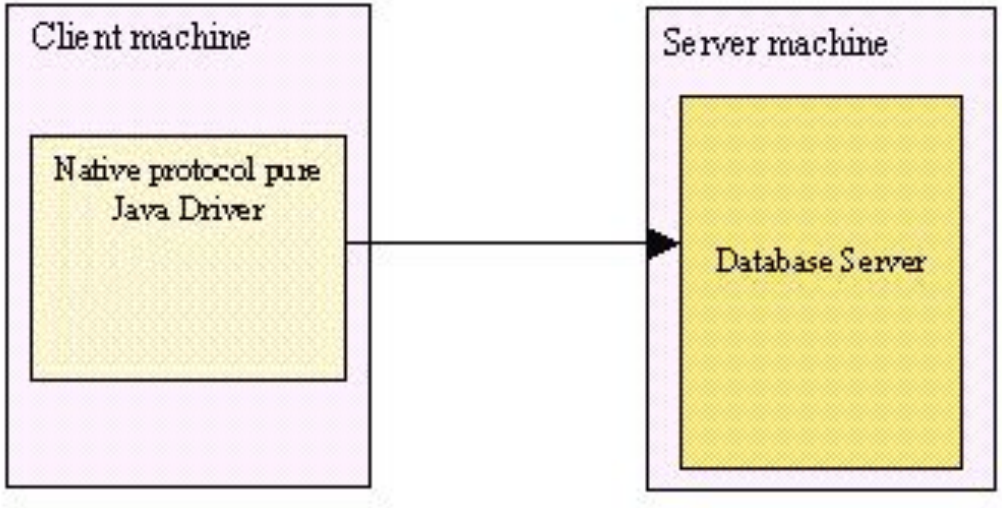
JDBC драйвер 3 типа – сетевой протокол/«чистый» Java драйвер – использует трехуровневую архитектуру, где вызовы JDBC посылаются на сервер приложений, далее этот сервер транслирует вызовы (явно или косвенно) в вызовы специфичного к СУБД нативного интерфейса для дальнейшего обращения к базе данных.



Типы драйверов

Нативный протокол/«чистый» Java драйвер

Нативный протокол/«чистый» Java драйвер (JDBC драйвер 4-го типа) конвертирует вызовы JDBC в специфический протокол вендора СУБД, так что клиентские приложения могут напрямую обращаться с сервером базы данных.



ИСПОЛЬЗОВАНИЕ JDBC

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

Загрузка драйвера базы данных

Загрузка класса 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.

Загрузка драйвера базы данных

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"
```

- Регистрация в программе на Java:

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

Загрузка драйвера базы данных

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

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" соединятся с БД, указанной в URL. Метод **connect**() драйвера использует этот URL для установления соединения.

Установление связи с БД

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**.

Установление связи с БД

```
…
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) {…}
    }  }  }
```

Установление связи с БД

JDBC-URL (Uniform Resource Locator)

Стандартный синтаксис JDBC URL:

```
jdbc:<subprotocol>:<subname>
```

- **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.

Установление связи с 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

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

Statement

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

```
…
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();
```

Statement

```
  rs = st.executeQuery("SELECT * FROM STUDENTS");
  while (rs.next()) {
    System.out.println(rs.getInt(1) + " " + rs.getString(2) + ""
+ rs.getInt(3));
  }
  } catch (ClassNotFoundException e) {…
  } catch (SQLException e) {…
  } finally {
    try {
        if (rs != null ){rs.close();}
        if (st != null) {st.close();}
        if (con != null) {con.close();}
    } catch (SQLException e) {
        …
  }
}}}
```

Statement

```
…
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-Яга\ ",123456)" );
    System.out.println(countRows);
} …
```

Statement

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

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())
{
    //обработка строки
}
```

ResultSet

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**.

ResultSet

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);
```

ResultSet

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

ResultSet

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
boolean isLast()	Определяет, установлен ли курсор на последнюю строку результирующего набора данных.
int getRow()	Номер строки, на которой установлен курсор

ResultSet

Методы интерфейса ResultSet

Метод	Описание
boolean afterLast()	Перемещает курсор после последней строки результирующего набора данных
boolean isAfterLast()	Определяет, находится ли курсов после последней строки результирующего набора данных
boolean previous()	Перемещает курсор на предыдущую строку результирующего набора данных
boolean absolute(int i)	Перемещает курсор к строке, номер которой указан как параметр
boolean relative(int i)	Перемещает курсор вперед или назад на количество строк, указанное в параметре, относительно текущего положения курсора

ResultSet

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

ResultSet

Методы, используемые с обновляемым набором данных:

Метод	Описание
void updateRow()	Обновляет текущую строку объекта ResultSet и базовую таблицу базы данных
void insertRow()	Обновляет текущую строку объекта ResultSet и базовую таблицу базы данных
void updateString()	Обновляет специфицированный столбец, заданный строковым значением
void updateInt()	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**.

ResultSet

```
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**.

PREPARED STATEMENT

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.

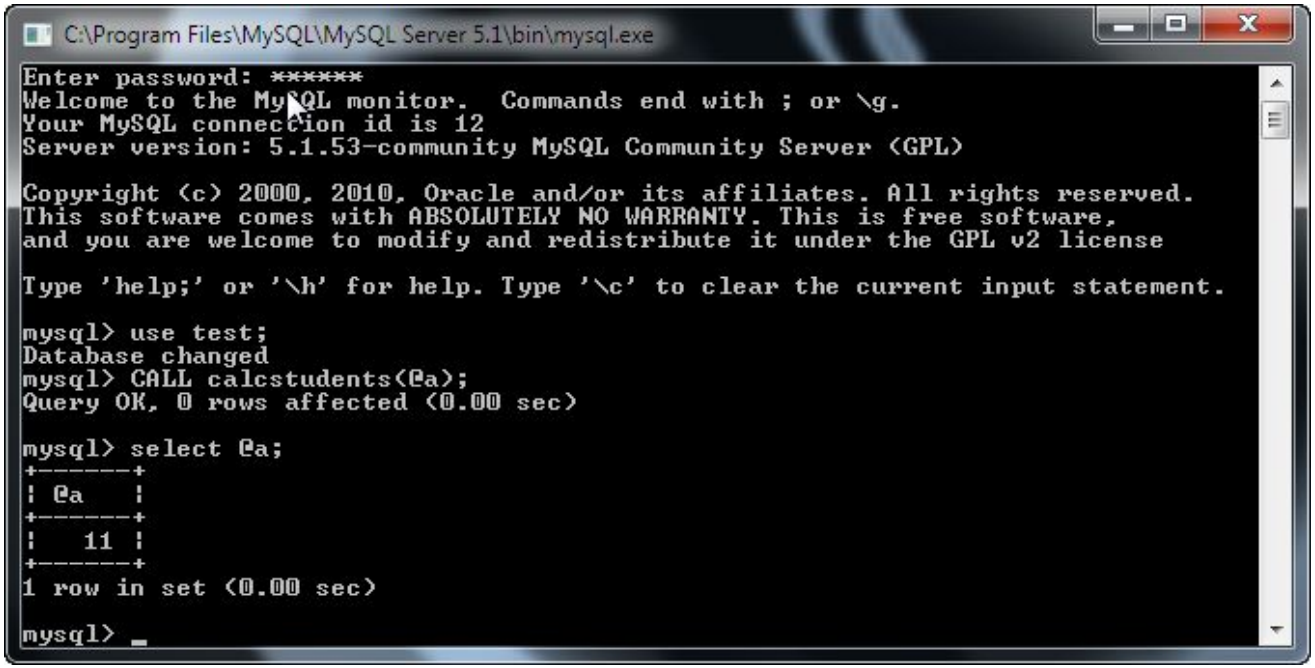
PreparedStatement

```
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, "Кукушкин");
      ps.setInt(2, 851001);
      ps.executeUpdate();
    } …
  }…
```

CALLABLE STATEMENT

CallableStatement

```
DELIMITER //
CREATE PROCEDURE calcstudents (OUT count INT)
BEGIN
SELECT COUNT(*) INTO count FROM students;
END;
```



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

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.

CallableStatement

```
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

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();
```

ЗАКТРЫТИЕ RESULTSET, STATEMENT И CONNECTION

Закрытие ResultSet, Statement и Connection

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

CONECTION POOL

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
```

Connection Pool

```
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);
    }

}
```

Connection Pool

```
public final class 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";
}
```

```
public class ConnectionPoolException extends Exception {
     private static final long serialVersionUID = 1L;

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

Connection Pool

```
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 dbResourseManager = DBResourceManager.getInstance();
  this.driverName = dbResourseManager.getValue(DBParameter.DB_DRIVER);
  this.url = dbResourseManager.getValue(DBParameter.DB_URL);
  this.user = dbResourseManager.getValue(DBParameter.DB_USER);

  this.password = dbResourseManager.getValue(DBParameter.DB_PASSWORD);

  try {
     this.poolSize = Integer.parseInt(dbResourseManager
          .getValue(DBParameter.DB_POLL_SIZE));
  } catch (NumberFormatException e) {
     poolSize = 5;
  }
}
```

Connection Pool

```
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);
  }
}
```

Connection Pool

```
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;
}
```

Connection Pool

```
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.");
      }
}
```

Connection Pool

```
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();
     }
```

Connection Pool

```
@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();
}

…
}
}
```

DATA ACCESS OBJECT (DAO)

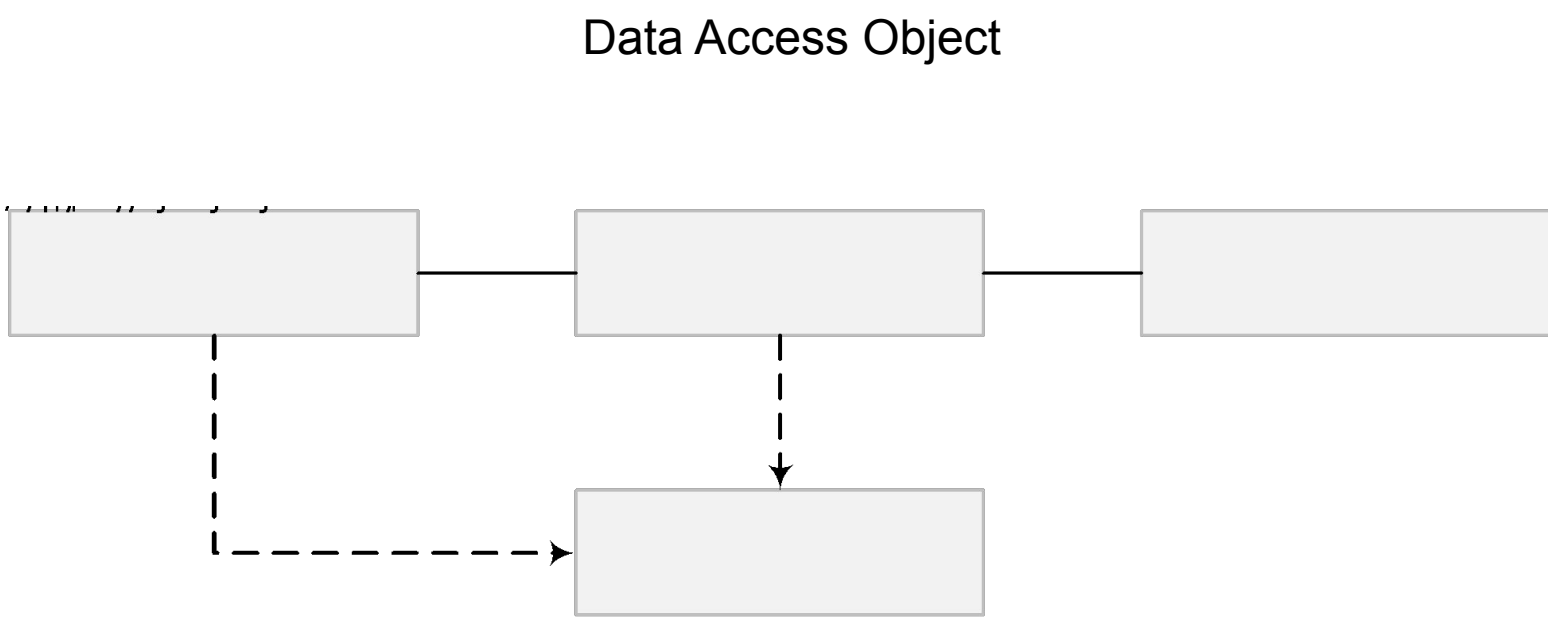
Data Access Object

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.

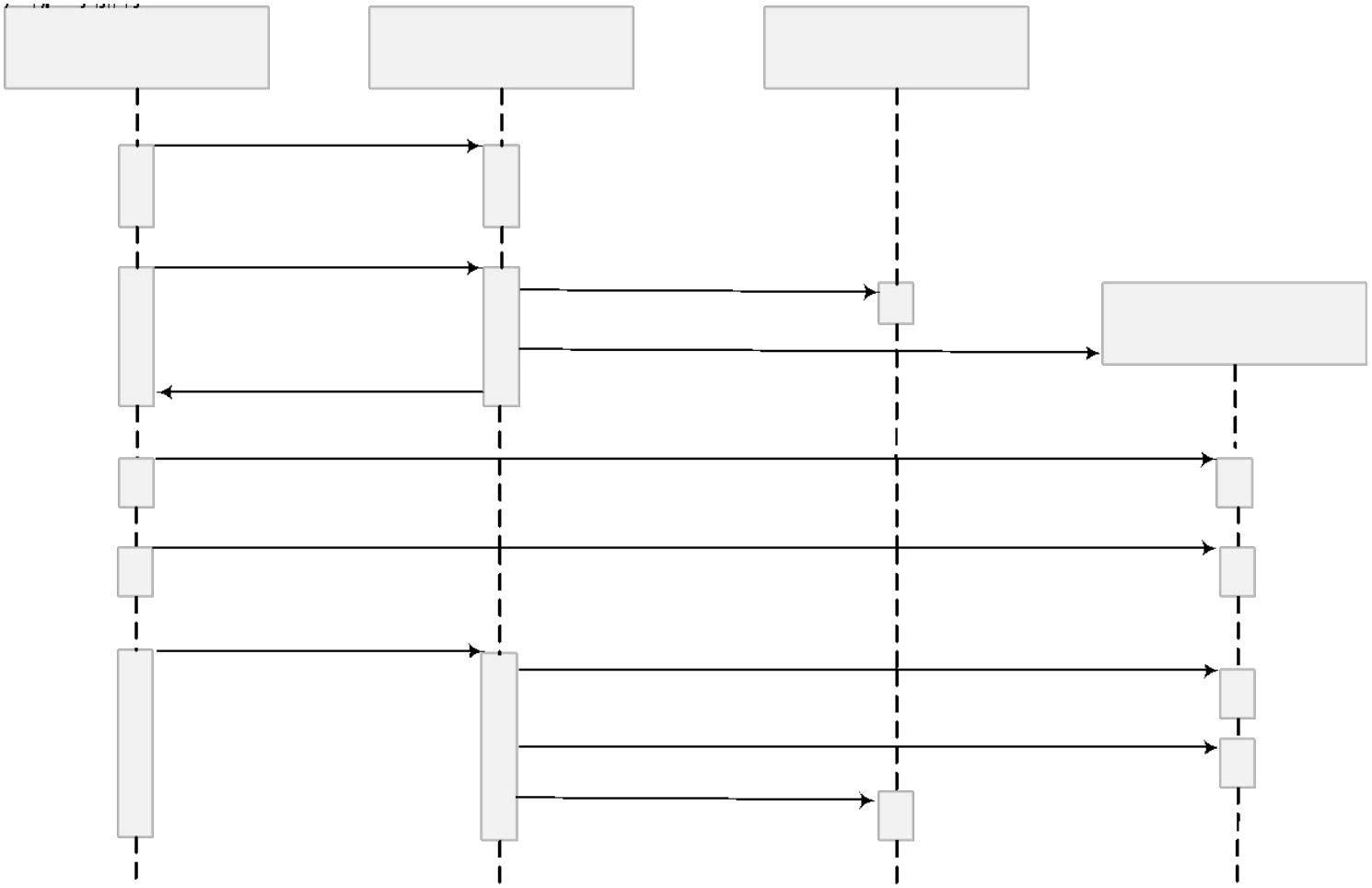
Использующие DAO бизнес-компоненты работают с более простым интерфейсом, предоставляемым объектом DAO своим клиентам. **DAO полностью скрывает детали реализации** источника данных от клиентов.

Data Access Object



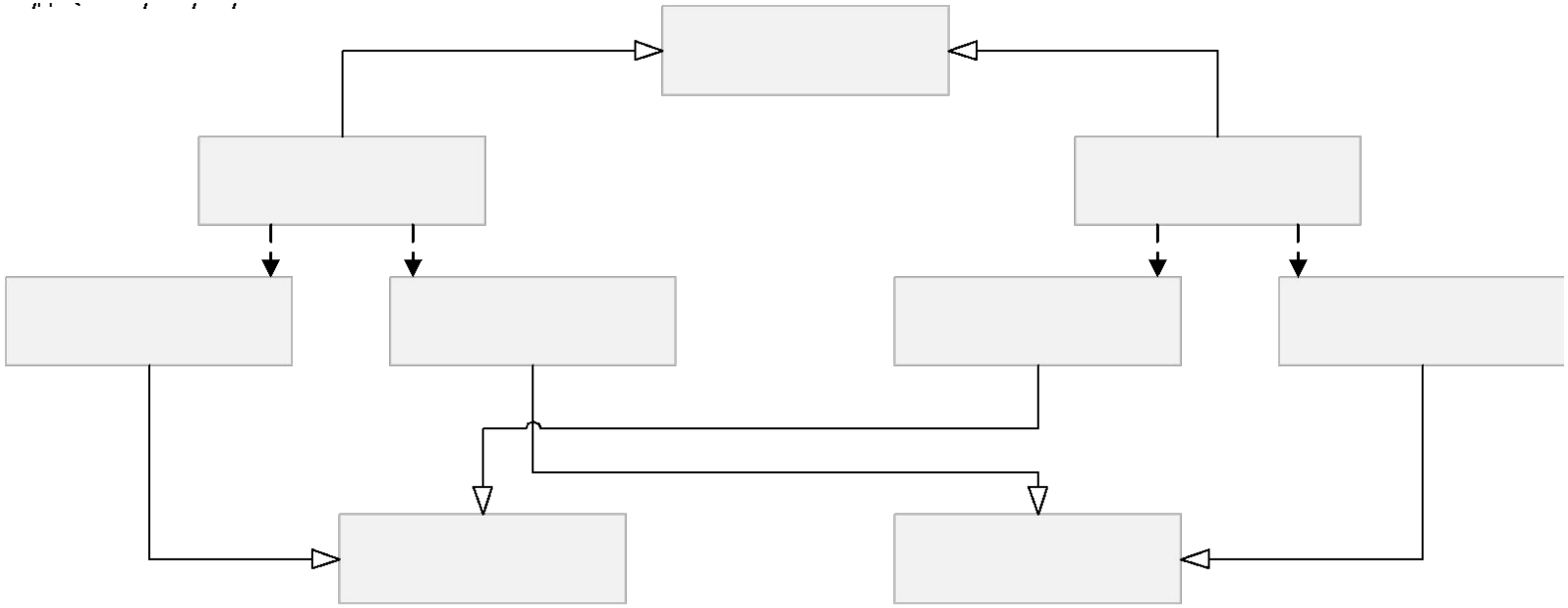
Data Access Object

Диаграмма последовательности действий паттерна Data Access Object



Data Access Object

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



Data Access Object

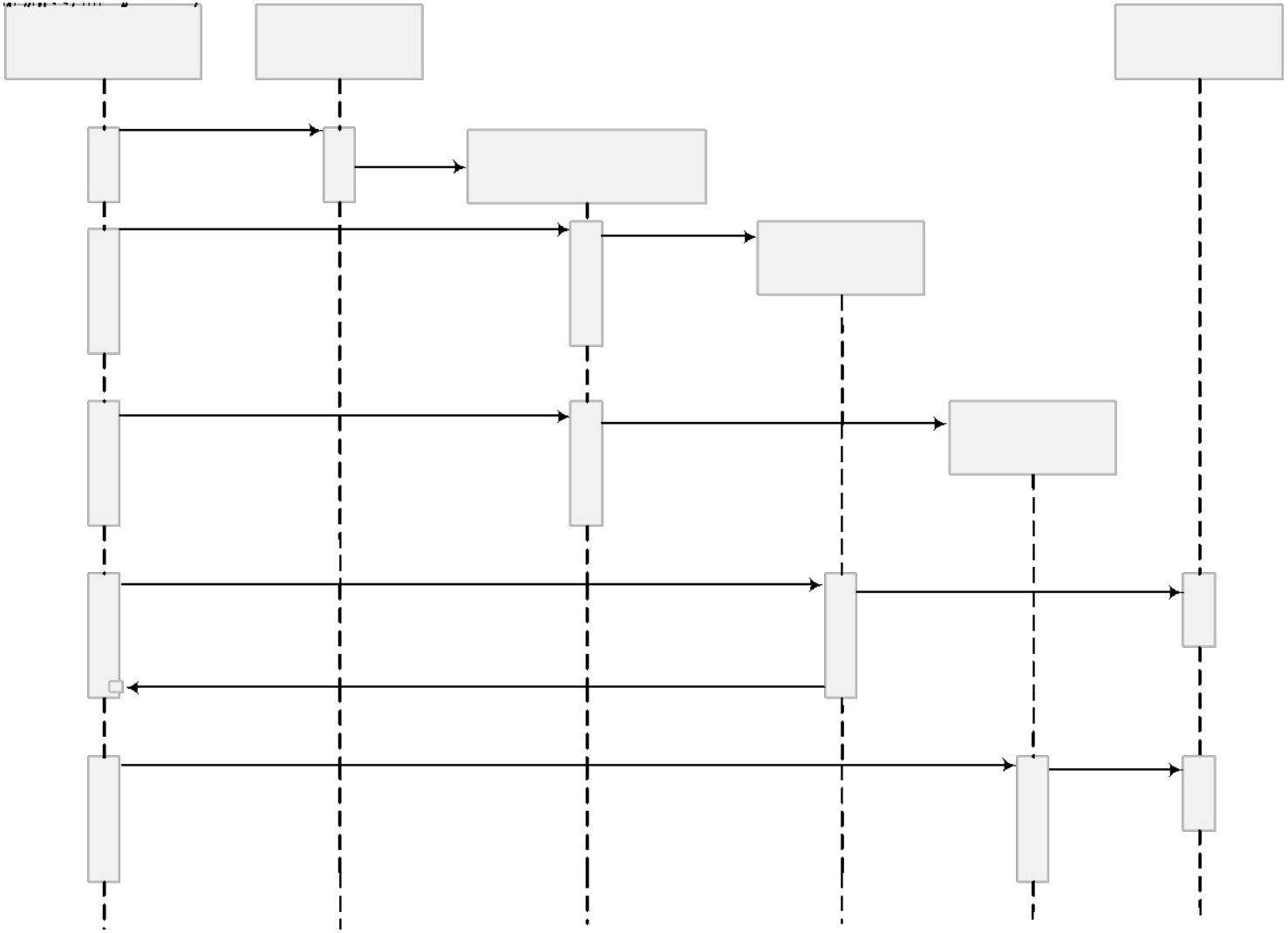


Диаграмма последовательности действий для стратегии Factory for Data Access Objects, использующей Abstract Factory.

ТРАНЗАКЦИИ И ТОЧКИ СОХРАНЕНИЯ

Транзакции и точки сохранения

Транзакция состоит из одного или более выражений (действий), которые поле 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.

Транзакции и точки сохранения

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.

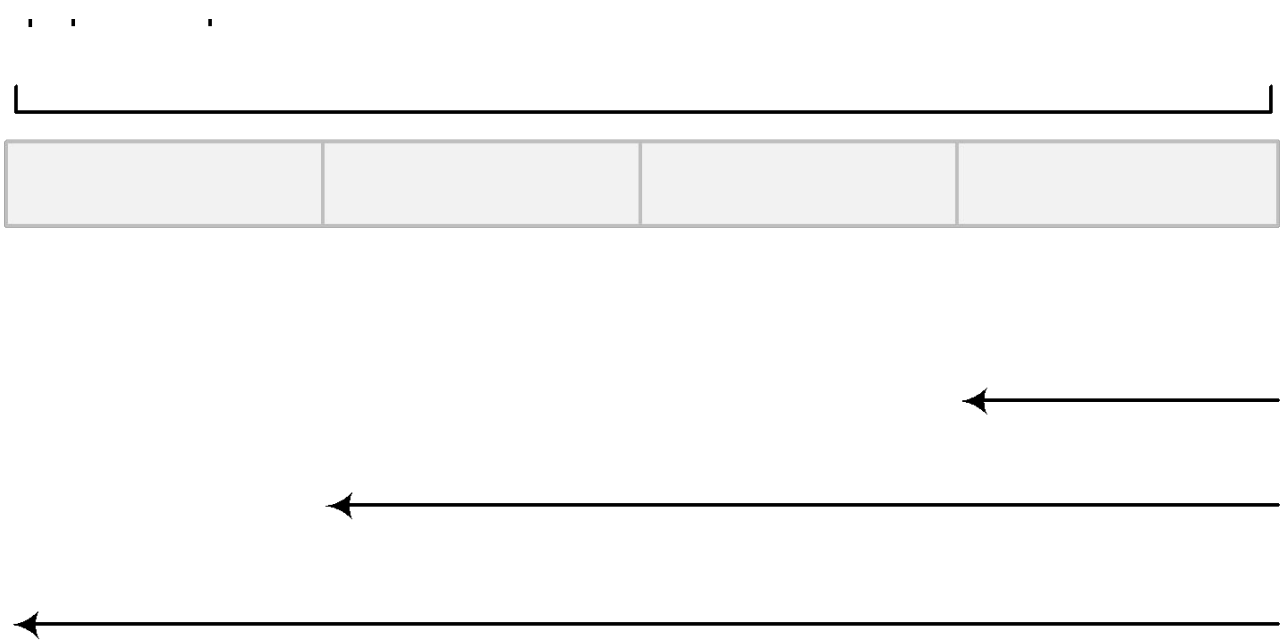
Если **auto-commit** запрещен, транзакция не заканчивается вплоть до явного вызова **commit** или **rollback**, включая, таким образом, все выражения, выполненные с момента последнего вызова **commit** или **rollback**. В этом случае все SQL-запросы в транзакции фиксируются или откатываются группой.

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

Транзакции и точки сохранения

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.



Транзакции и точки сохранения

```
cn.setAutoCommit(false);
…
Statement st = cn.createStatement();
int rows = st.executeUpdate("INSERT INTO Employees " +
        "(FirstName, LastName) VALUES " + "('Игорь', 'Цветков')");
// Устанавливаем именованную точку сохранения.
Savepoint svpt = cn.setSavepoint("NewEmp");

// ...
rows = st.executeUpdate("UPDATE Employees
    set Address = 'ул. Седых, 19-34' " +
     "WHERE LastName = 'Цветков'");

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

МЕТАДАННЫЕ

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.**

Метаданные

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.

Метаданные

```
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) );
  }
}
}
```

Метаданные

Получить объект **DatabaseMetaData** можно следующим образом:

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

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

- 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**()

Метаданные

```
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));
} …
```

СПАСИБО ЗА ВНИМАНИЕ!

ВОПРОСЫ?

Java.SE.10

JDBC fundamentals

Author: **Olga Smolyakova**
Oracle Certified Java 6 Programmer
Olga_Smolyakova@epam.com