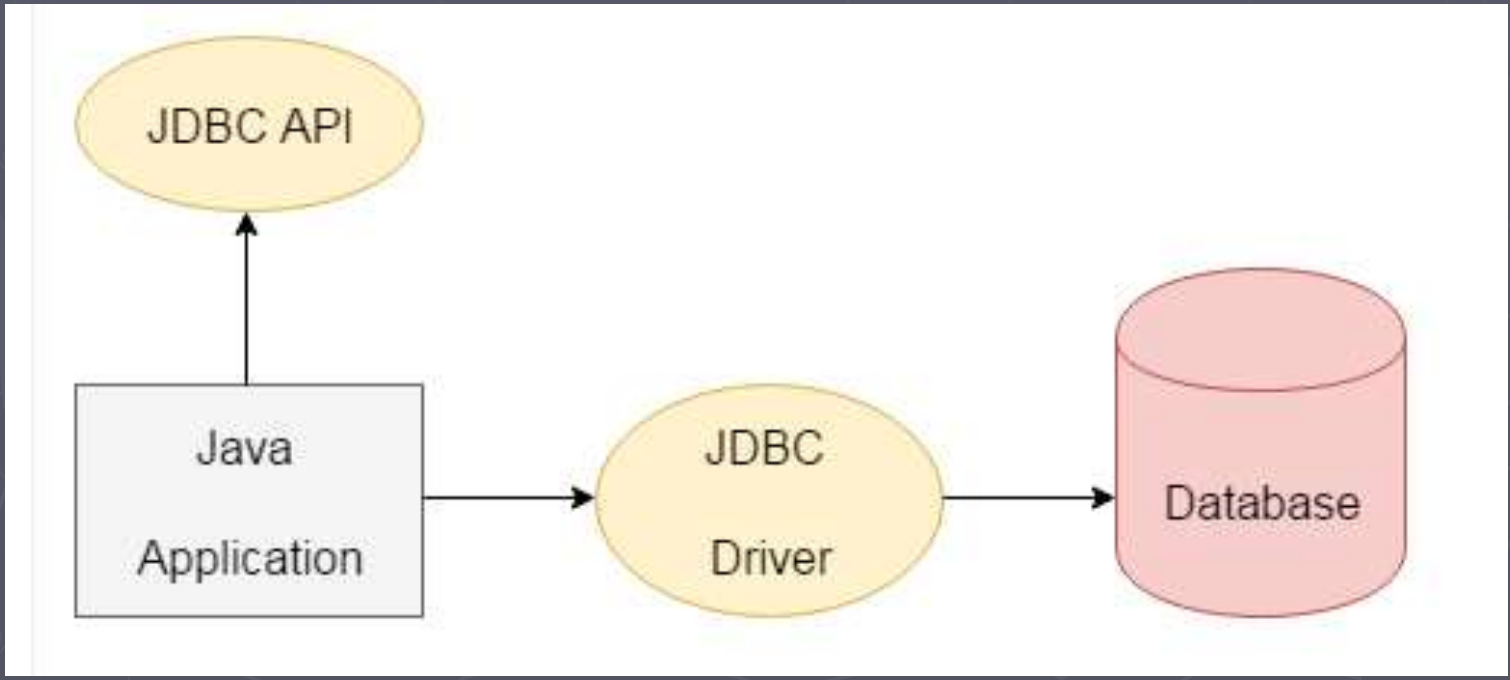
API JDBC

- API JDBC (Java DataBase Connectivity)



Типы драйверов

- 1) Драйвер, использующий другой прикладной интерфейс взаимодействия с СУБД (JDBC-ODBC) JDBC-ODBC bridge driver
- 2*) Драйвер, работающий через внешние native библиотеки клиента СУБД* Native-API driver (partially java driver)
- 3) Драйвер, работающий по сетевому и независимому от СУБД протоколу с промежуточным Java-сервером . Network Protocol driver (fully java driver)
- 4) Сетевой драйвер, работающий напрямую с нужной СУБД Thin driver (fully java driver)

- JDBC-ODBC bridge driver

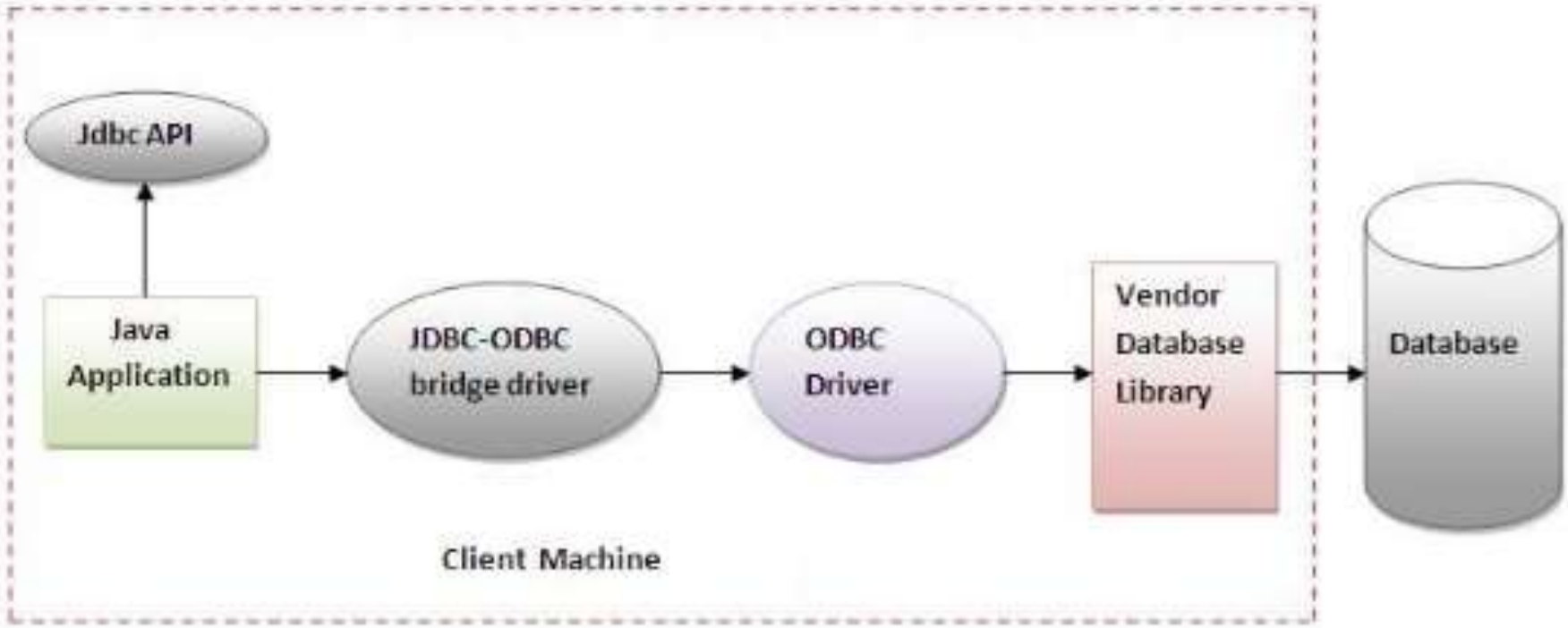
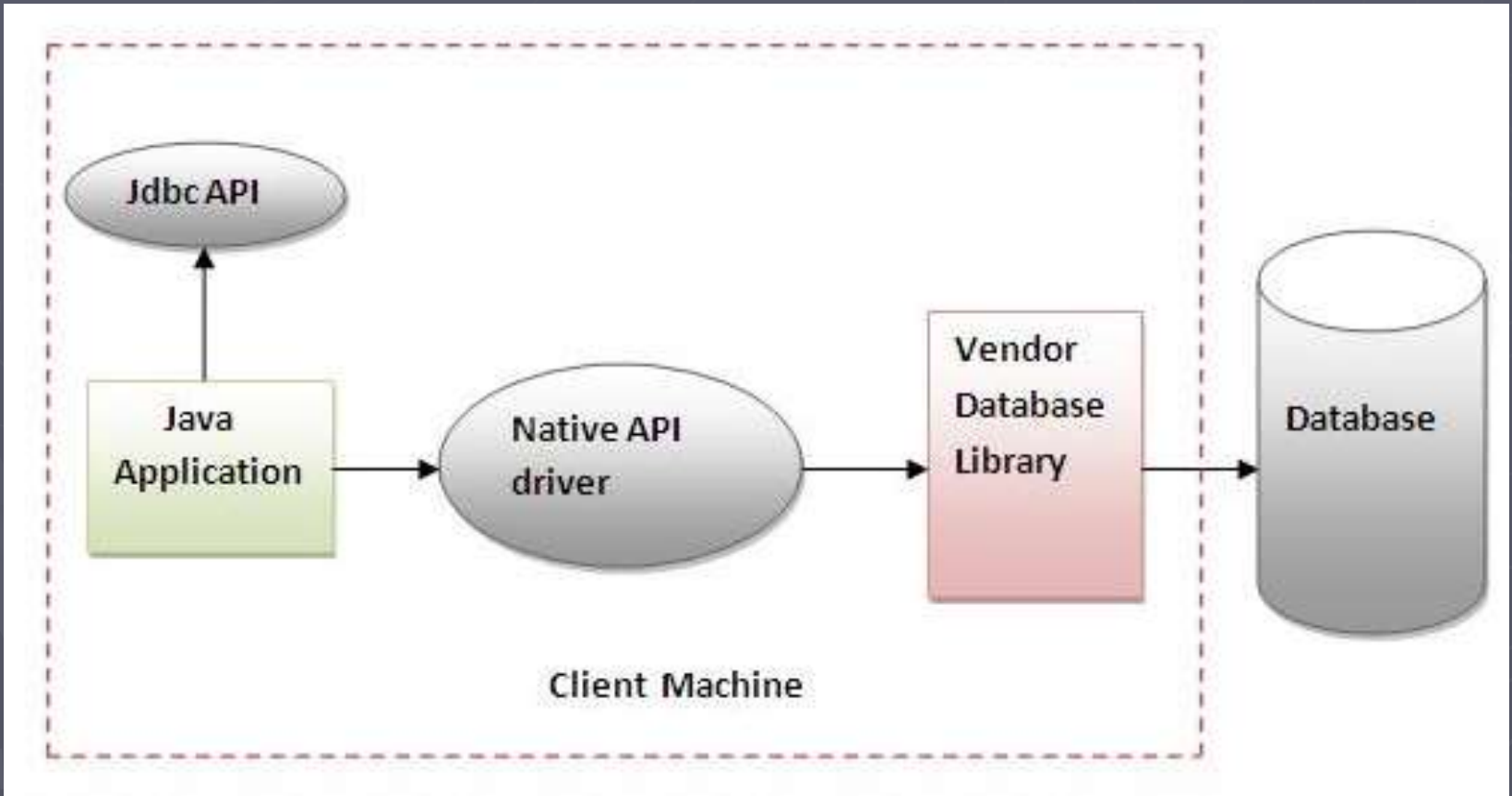


Figure- JDBC-ODBC Bridge Driver

- Native-API driver



- Network Protocol driver

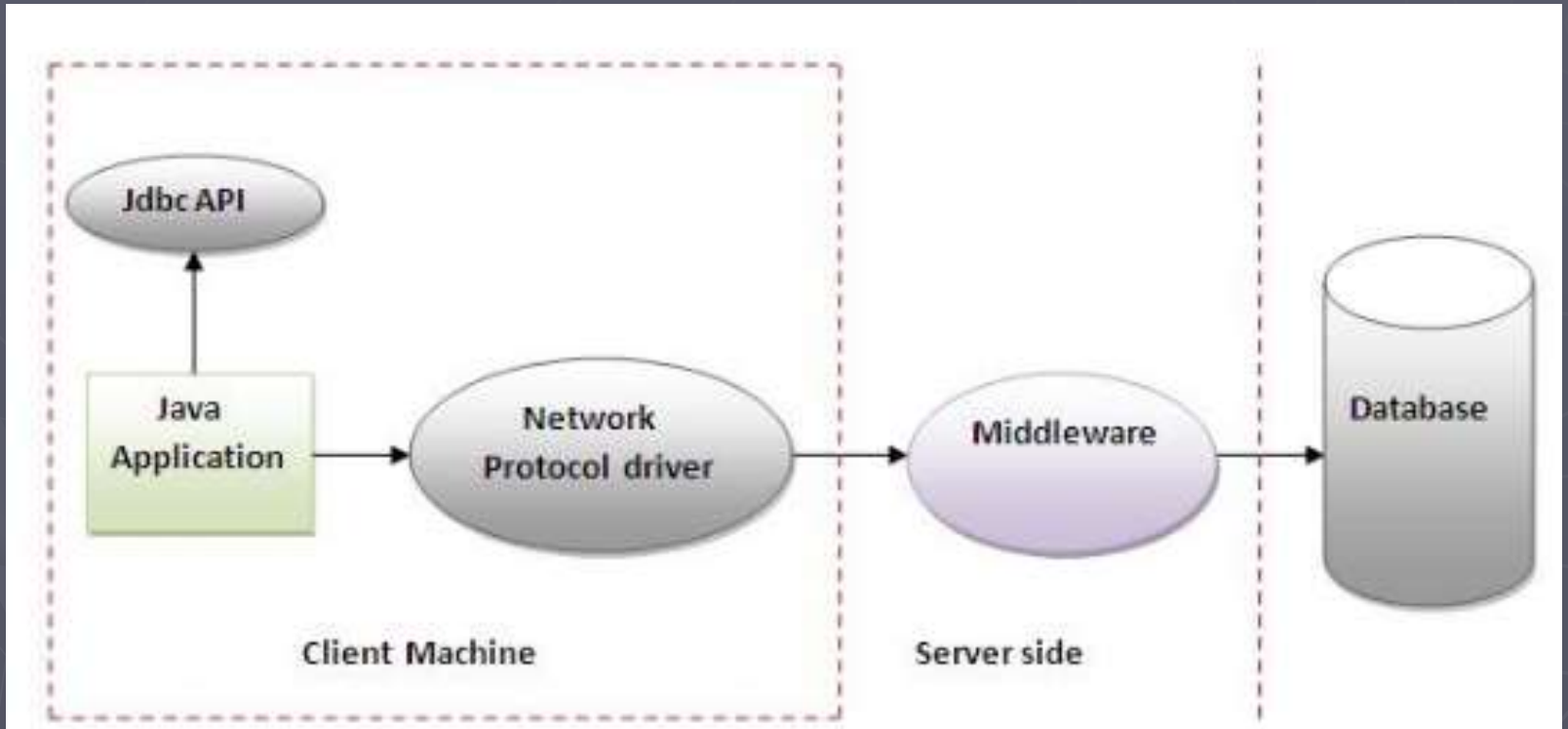
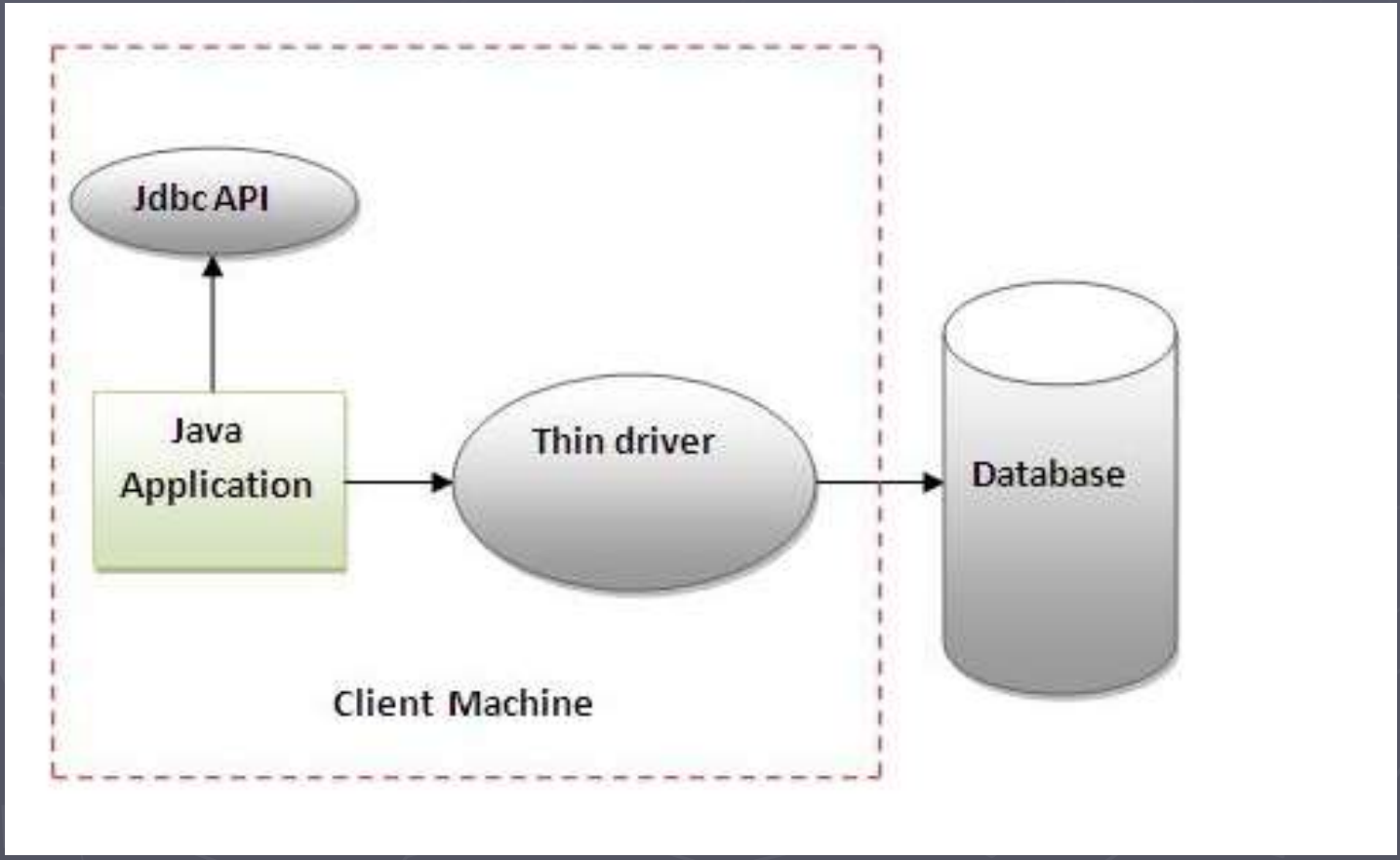


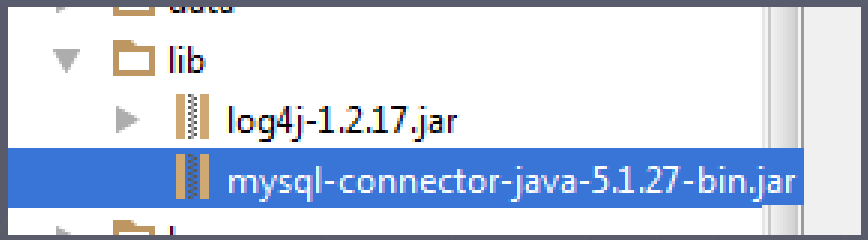
Figure- Network Protocol Driver

- Thin driver

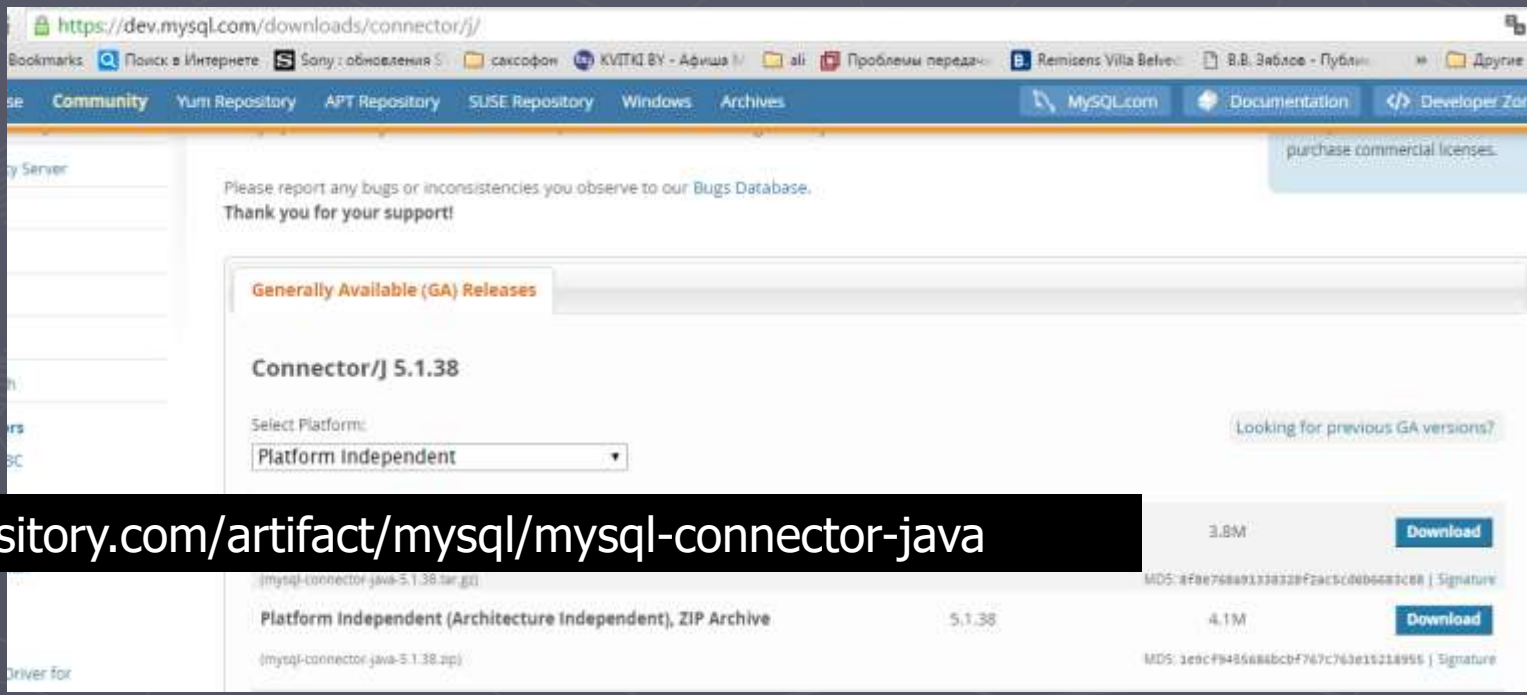

Jdbc API
Java Application
Thin driver
Database
Client Machine

1. Подключение драйвера базы данных

/lib



- mysql-connector-java-[номер версии]-bin.jar
- ojdbc[номер версии].jar



https://mvnrepository.com/artifact/mysql/mysql-connector-java

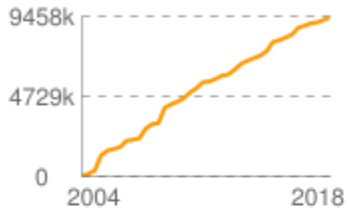
Защищено | https://mvnrepository.com/artifact/mysql/mysql-connector-java

MVNREPOSITORY

Search for groups, artifacts, categories

Search

Indexed Artifacts (9.46M)



Popular Categories

- Aspect Oriented
- Actor Frameworks
- Application Metrics
- Build Tools
- Bytecode Libraries
- Command Line Parsers
- Cache Implementations
- Cloud Computing
- Code Analyzers
- Collections
- Configuration Libraries
- Core Utilities
- Date and Time Utilities
- Dependency Injection

Home » mysql » mysql-connector-java

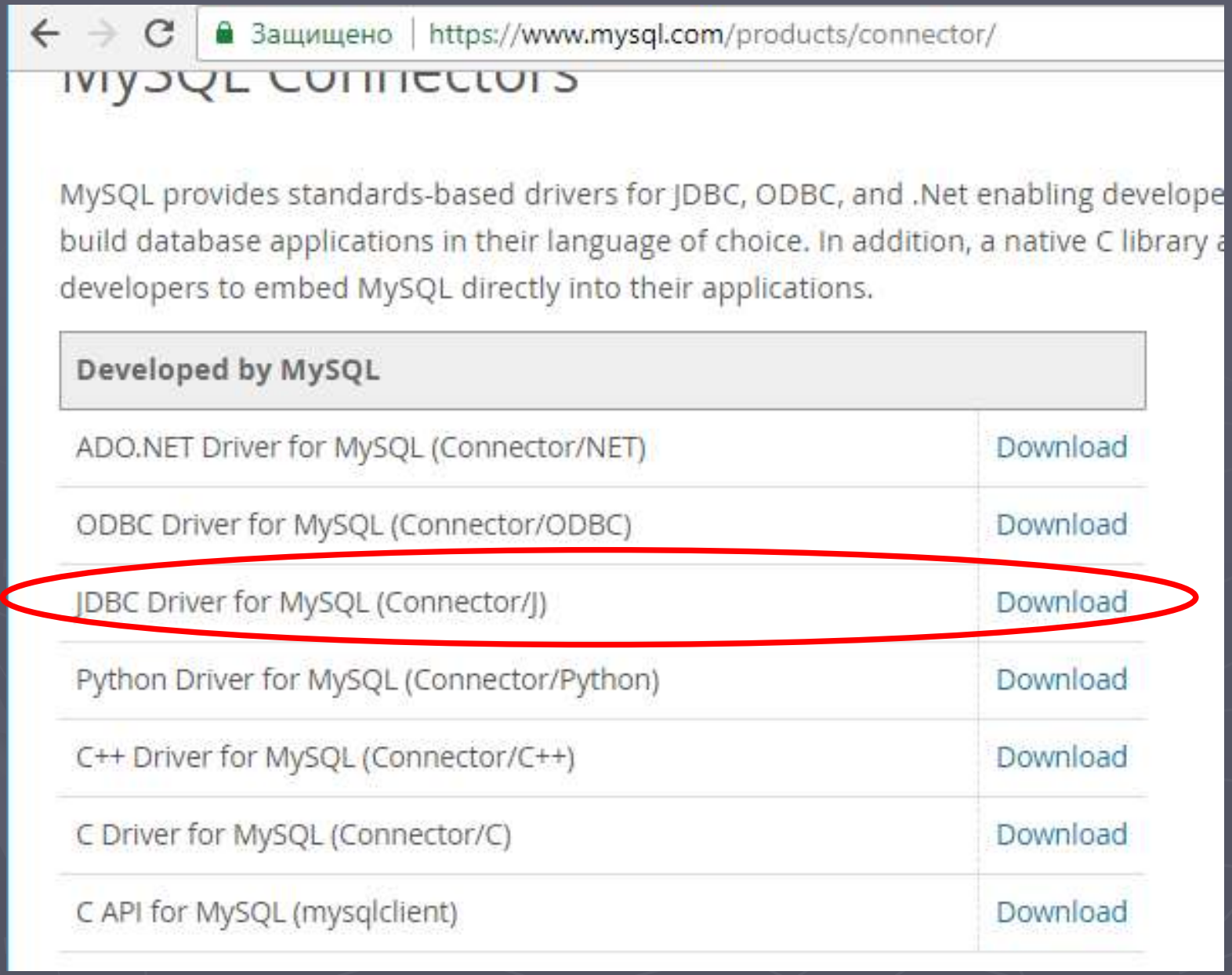
MySQL Connector/J

JDBC Type 4 driver for MySQL

Categories	MySQL Drivers
Tags	mysql database connector driver
Used By	2,426 artifacts

Central (67) | Jahia (1)

	Version	Repository	Usages	Da
8.0.x	8.0.11	Central	1	(Apr, 201
	8.0.9-rc	Central	0	(Jan, 201
	8.0.8-dmr	Central	12	(Sep, 201
	8.0.7-dmr	Central	6	(Jun, 201
6.0.x	6.0.6	Central	113	(Feb, 201
	6.0.5	Central	48	(Oct, 201
	6.0.4	Central	16	(Aug, 201
	6.0.3	Central	21	(Jun, 201


Защищено
https://www.mysql.com/products/connector/
MySQL Connectors
MySQL provides standards-based drivers for JDBC, ODBC, and .Net enabling develope
build database applications in their language of choice. In addition, a native C library a
developers to embed MySQL directly into their applications.
Developed by MySQL
ADO.NET Driver for MySQL (Connector/NET)
Download
ODBC Driver for MySQL (Connector/ODBC)
Download
JDBC Driver for MySQL (Connector/J)
Download
Python Driver for MySQL (Connector/Python)
Download
C++ Driver for MySQL (Connector/C++)
Download
C Driver for MySQL (Connector/C)
Download
C API for MySQL (mysqlclient)
Download

2. Установка соединения с БД

java.sql.DriverManager

```
// DriverManager.registerDriver(new com.mysql.jdbc.Driver());
//JDBC 4.0
Connection cnn = DriverManager.getConnection
("jdbc:mysql://localhost:3306/test","root",  "root");
```

3. Создание объекта для передачи запросов

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

PreparedStatement

CallableStatement

4. Выполнение запроса

- execute(String sql)
- executeBatch()
- executeQuery(String sql)
- executeUpdate(String sql)

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

5. Обработка результатов выполнения запроса

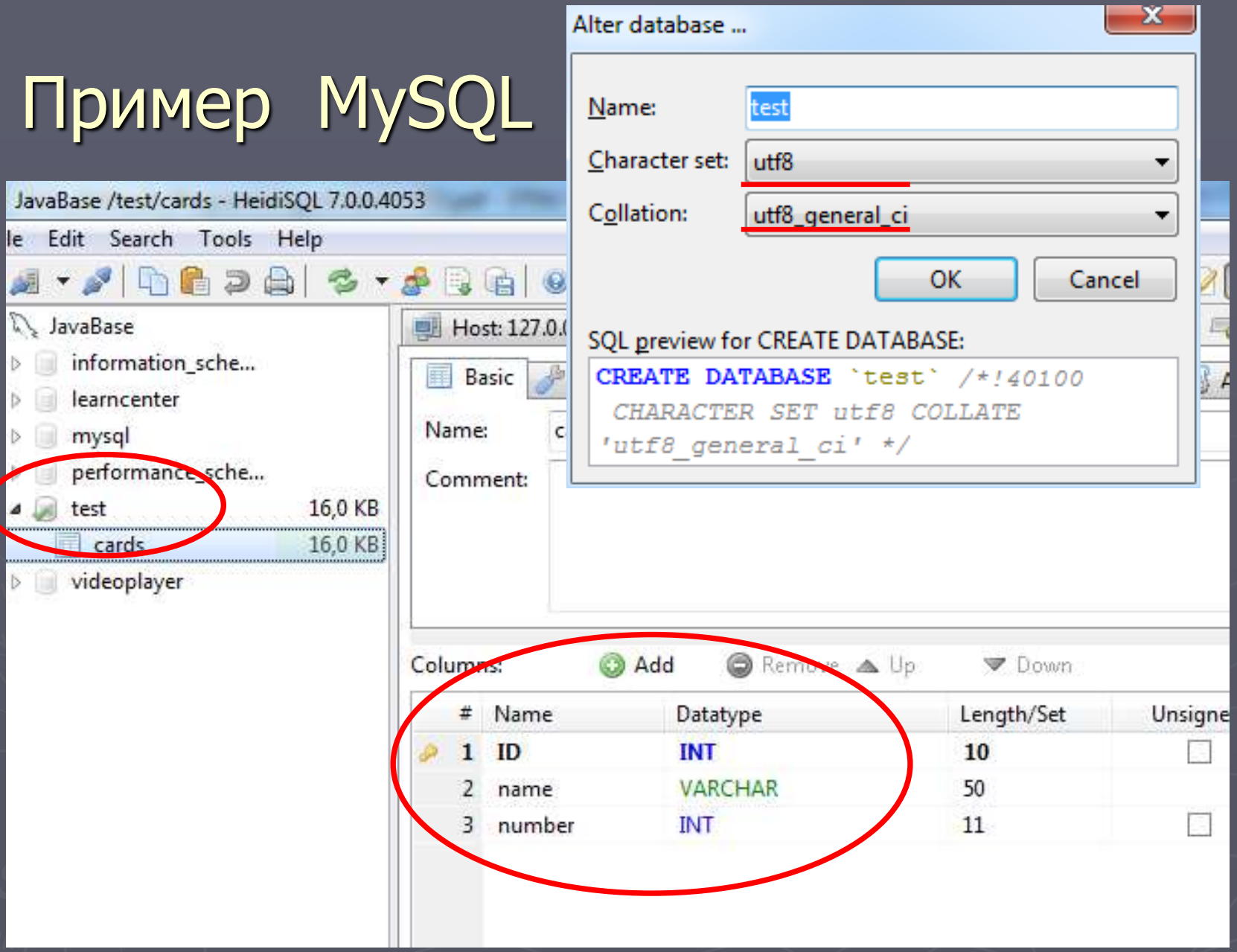
интерфейс ResultSet

- next(), first(), previous(), last()
- getString(int pos),
- get*Тип*(int pos) -
  getInt(int pos), getFloat(int pos)
  - getClob(int pos) и getBlob(int pos),
- update*Тип*()
- int getInt(String columnLabel),
  String getString(String columnLabel),
  Object getObject(String columnLabel)

6. Закрытие соединения, statement

```
st.close(); // закрывает также и ResultSet
cn.close();
```

Пример MySQL



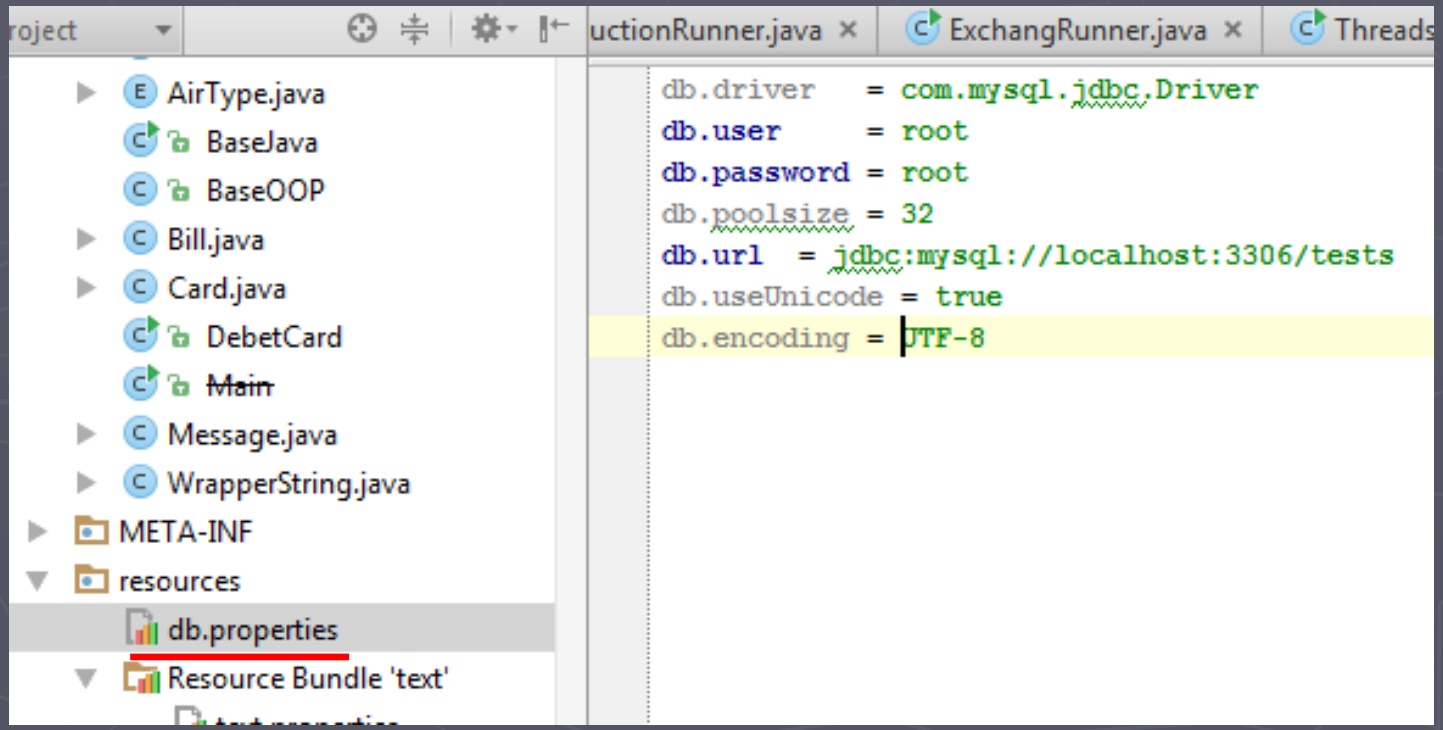
```
class Card{
    public Card(int id, String name, int number) {
        this.id = id;
        this.name = name;
        this.number = number;
    }

    public void setId(int id) { this.id = id; }

    public void setName(String name) { this.name = name; }

    public void setNumber(int number) { this.number = number; }

    public int getId() { return id; }

    public String getName() { return name; }

    public int getNumber() { return number; }

    private  int id;
    private  String name;
    private  int number;
}
```

```
public class SimpleJDBC {
    public static void main(String[] args) {
        String url = "jdbc:mysql://127.0.0.1:3306/test";
        Properties prop = new Properties();
        prop.put("user", "root");
        prop.put("password", "root");
        prop.put("autoReconnect", "true");
        prop.put("characterEncoding", "UTF-8");
        prop.put("useUnicode", "true");
        Connection cn = null;

        try { // 1 блок
            cn = DriverManager.getConnection(url, prop);
             Statement st = null;
            try { // 2 блок
                st = cn.createStatement();
            ResultSet rs = null;
            try { // 3 блок
                rs = st.executeQuery("SELECT * FROM cards");
                ArrayList<Card> lst = new ArrayList<>();
                while (rs.next()) {
                    int id = rs.getInt(1);
                    int number = rs.getInt(3);
                    String name = rs.getString(2);
                    lst.add(new Card(id, name, number));
                }
                if (lst.size() > 0) {
                    System.out.println(lst);
                } else {
                    System.out.println("Not found");
```

```
"C:\Program ...
[Card{id=1, name='Ivanov',
 number=4523}, Card{id=2,
 name='Perov', number=12345}]

Process finished with exit code 0
```

Параметры соединения

- с помощью прямой передачи значений в коде класса – плохо
- properties
- xml



```
class ConnectorDB {
    public static Connection getConnection() throws SQLException {
        ResourceBundle resource =
                    ResourceBundle.getBundle("db");

        String url = resource.getString("db.url");
        String user = resource.getString("db.user");
        String pass = resource.getString("db.password");

       return DriverManager.getConnection(url, user, pass);
    }
}
```

```
cn = ConnectorDB.getConnection();
```

Метаданные

Интроспекция объектов - методы, которые позволяют получить список таблиц, определить типы, свойства и количество столбцов БД и т.д.

- интерфейсы *ResultSetMetaData* и *DatabaseMetaData*

```
ResultSet rs = ...;
    ResultSetMetaData rsMetaData = rs.getMetaData();
```

int getColumnCount()

String getColumnName(int column)

int getColumnType(int column)

```
    cn = DriverManager.getConnection(url, prop);

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

String getDatabaseProductName() возвращает название

String getDatabaseProductVersion() возвращает номер версии СУБД

String getDriverName() имя драйвера JDBC

String getUserName() имя пользователя БД

String getURL() местонахождение источника данных

ResultSet getTables() набор типов таблиц

Подготовленные запросы и хранимые процедуры

- PreparedStatement - используется для часто повторяющихся запросов SQL

Такой оператор предварительно готовится и хранится в объекте

- Ускоряет обмен информацией с базой данных при многократном выполнении однотипных запросов
- невозможен sql injection attacks

подготовка SQL-запроса

```
String sql = "INSERT INTO cards(id, name, number)
                                VALUES(?, ?, ?)";
PreparedStatement ps = cn.prepareStatement(sql);
```

- CallableStatement - для выполнения хранимых процедур, созданных средствами самой СУБД.

```
class DBHelp {
   private final static String SQL_INSERT =
           "INSERT INTO cards(id, name, number ) VALUES(?,?,?)";
   private Connection connect;

   public DBHelp() throws SQLException {
       connect = ConnectorDB.getConnection();
   }
   public PreparedStatement getPreparedStatement(){
       PreparedStatement ps = null;
       try {
           ps  =  connect.prepareStatement(SQL_INSERT);
       }  catch (SQLException e) {
           e.printStackTrace();
       }
       return ps;
   }
```

Готовим запрос

```
public boolean insertCard(PreparedStatement ps, Card cr) {
        boolean flag = false;
        try {
            ps.setInt(1,  cr.getId());
            ps.setString(2,  cr.getName());
            ps.setInt(3,  cr.getNumber());
            ps.executeUpdate();
            flag  =  true;
        }  catch (SQLException e) {
            e.printStackTrace();
        }
        return flag;
    }
    public void closeStatement(PreparedStatement ps) {
        if (ps != null) {
            try {
                ps.close();
            }  catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Выполняем запрос

Установка входных значений конкретных параметров

```
public class DBInsertRunner {

    public static void main(String[] args) {


        ArrayList<Card> list = new ArrayList<Card>() {
            {
                add(new Card(8, "Feldman", 234));
                add(new Card(3, "Poter", 574));
            }
        };
        DBHelp helper = null;

        PreparedStatement statement = null;
        try {
            helper = new DBHelp();
            statement = helper.getPreparedStatement();
            for (Card a : list) {
                helper.insertCard(statement, a);
            }
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
                        helper.closeStatement(statement);
        }
    }
```

Элементы для вставки

Host: 127.0.0.1 | Database: test

test.cards: 4 rows total (approximately)

ID	name	number
1	Ivanov	4523
2	Perov	12345
3	Poter	574
8	Feldman	234

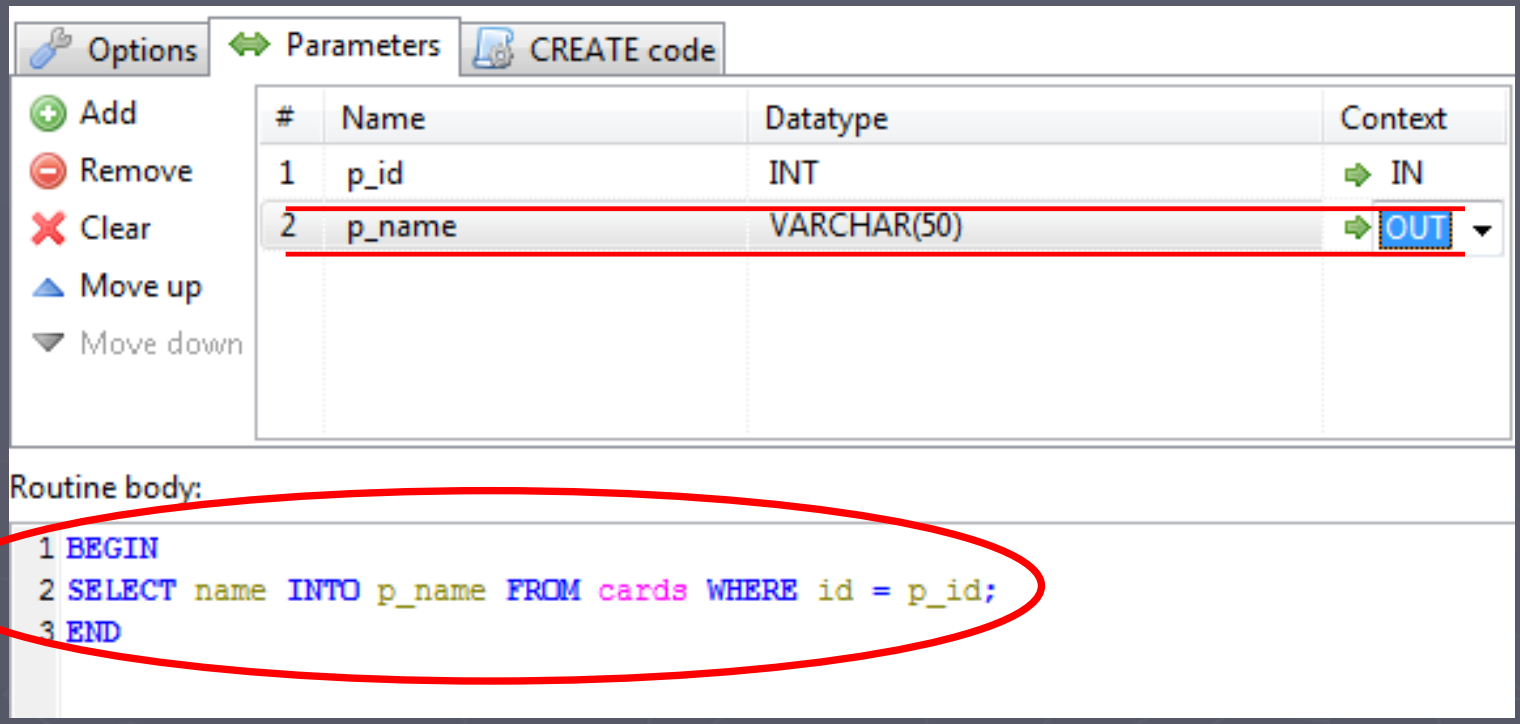
предварительно подготовлен и выполняется быстрее обычных операторов

Интерфейс CallableStatement

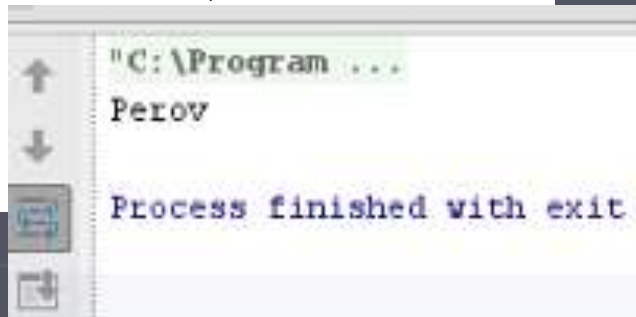
обеспечивает выполнение хранимых процедур

Именованная последовательность команд SQL, рассматриваемых как единое целое и выполняющаяся в адресном пространстве процессов СУБД, который можно вызвать извне

- 1) регистрируются входные и выходные параметры
  - registerOutParameter()
- 2) Выполняется
  - execute(),
  - executeQuery()
  - executeUpdate()




```
final String SQL = "{call getname (?, ?)}";
CallableStatement cs = helper.getConnect().prepareCall(SQL);
// передача значения входного параметра
cs.setInt(1, 2);
// регистрация возвращаемого параметра
cs.registerOutParameter(2, java.sql.Types.VARCHAR);
cs.execute();
String RName = cs.getString(2);
System.out.println(RName);
```



запуск batch-команд

- запускает на исполнение в БД массив запросов SQL вместе, как одну единицу

Заменяет многократное выполнение методов execute() или executeUpdate()

переходит в режим неавтоматического подтверждения операций

```
// выключить autocommit
helper.getConnect().setAutoCommit(false);
Statement st =  helper.getConnect().createStatement();
st.addBatch("INSERT INTO cards VALUES (10, 'Vavilov',76)");
st.addBatch("INSERT INTO cards VALUES (11, 'Samal',23)");
st.addBatch("INSERT INTO cards VALUES (12, 'Gucu',1)");
// выполнить batch команд обнавлений
int[ ] updateCounts = st.executeBatch();
```

возвращает массив чисел, которые характеризуют число строк, которые были изменены конкретным запросом

Транзакции

единица работы, обладающая свойствами ACID (Атомарность, Согласованность, Изолированность, Долговечность)

интерфейс Connection

commit() - подтверждает выполнение SQL-запросов

В JDBC commit выполняется по умолчанию после каждого вызова методов executeQuery() и executeUpdate()

rollback() - отменяет действие всех запросов SQL, начиная от последнего вызова commit()

Задача: Перевести деньги (оплатить)

```
int sum = 1234567;

Connection connectionFrom  =  DriverManager.getConnection(
        "jdbc:mysql://localhost:3306/testFrom", "root", "root");
connectionFrom.setAutoCommit(false);

Connection     connectionTo = DriverManager.getConnection(
        "jdbc:mysql://10.162.4.151:3306/testTo", "root", "root");
    connectionTo.setAutoCommit(false);

try {

    if (sum <= 0) {
        throw new NumberFormatException("less or equals zero");
    }
    Statement   stFrom  =  connectionFrom.createStatement();
    Statement   stTo  =  connectionTo.createStatement();
```

текущее соединение с БД переходит в режим неавтоматического подтверждения операций

выполнение запроса на изменение не приведет к необратимым последствиям, пока COMMIT не будет выполнен

Посмотреть балансы на счетах

```
// транзакция
ResultSet rsFrom =
        stFrom.executeQuery("SELECT balance from table_from");
ResultSet  rsTo  =
        stTo.executeQuery("SELECT balance from table_to");
int accountFrom = 0;
while (rsFrom.next()) {
    accountFrom  =  rsFrom.getInt(1);
}
int resultFrom= 0;
if (accountFrom >= sum) {
    resultFrom = accountFrom - sum;
}  else {
    throw new SQLException("Invalid balance");
}
```

Рассчитать сумму остатка

```
            int accountTo = 0;
            while (rsTo.next()) {
                accountTo = rsTo.getInt(1);
            }
            int resultTo = accountTo + sum;
            stFrom.executeUpdate(
                    "UPDATE table_from SET balance=" + resultFrom);
            stTo.executeUpdate("UPDATE table_to SET balance=" + resultTo);

        // завершение транзакции
            connectionFrom.commit();
            connectionTo.commit();
            System.out.println("remaining on :" + resultFrom + " dollars");
        }

          catch (SQLException e) {
            System.err.println("SQLState: " + e.getSQLState()
                    + "Error Message: " + e.getMessage());
            // откат транзакции при ошибке
            connectionFrom.rollback();
            connectionTo.rollback();

}
```

Рассчитать сумму после зачисления

все изменения таблицы производятся как одно действие

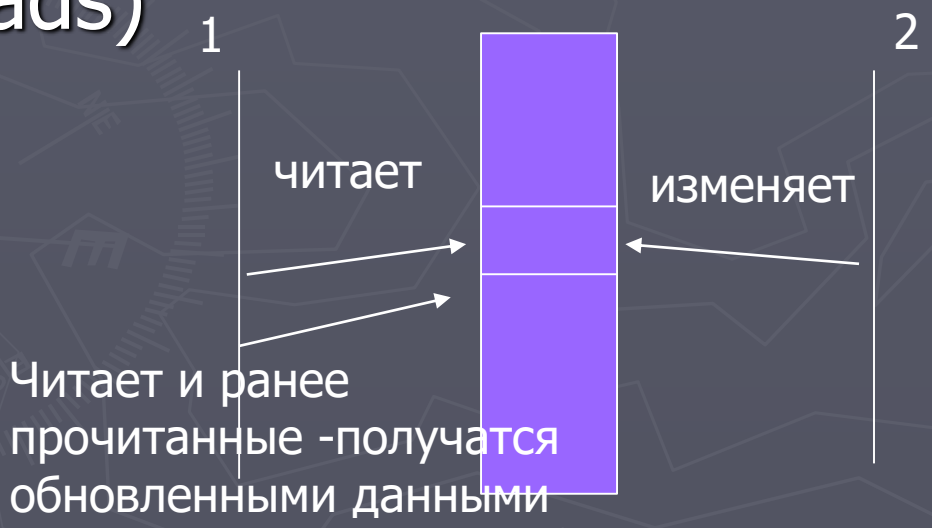
отменяются действия всех запросов, начиная от последнего вызова commit()

Типы чтения при транзакциях

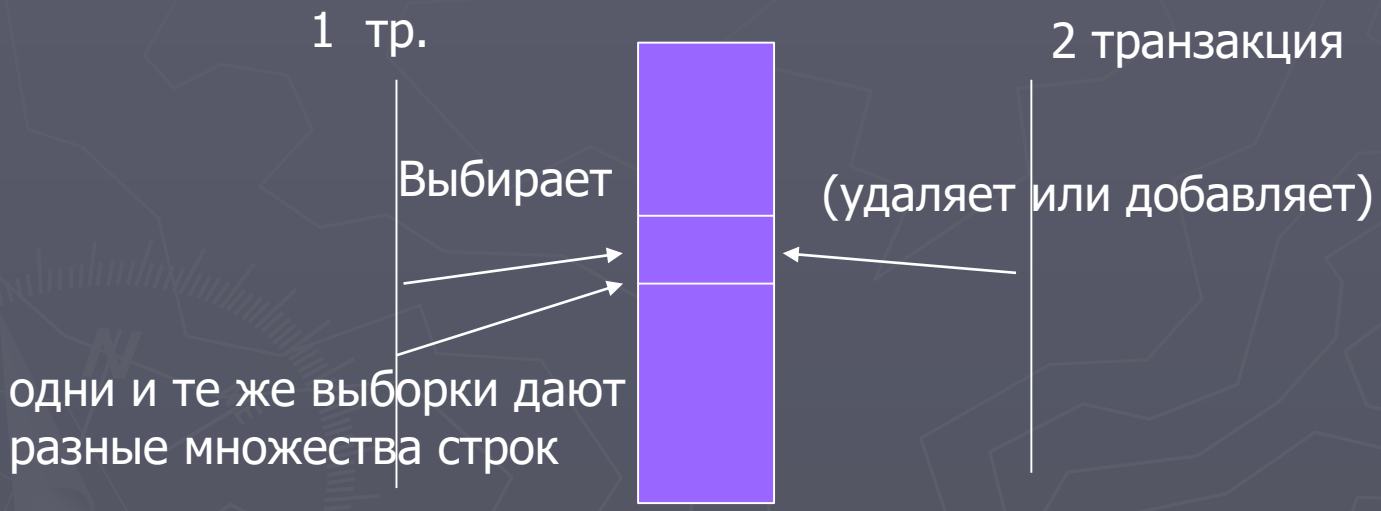
- грязное чтение (dirty reads)
  - изменения, сделанные в одной транзакции, видны вне ее до того, как она была сохранена

Может быть выполнение на основе некорректных данных

- неповторяющееся чтение (nonrepeatable reads)



- фантомное чтение (phantom reads)



Уровни изоляции транзакции

- TRANSACTION_NONE драйвер не поддерживает транзакции
- TRANSACTION_READ_UNCOMMITTED — разрешает грязное, неповторяющееся и фантомное чтения;
- TRANSACTION_READ_COMMITTED —

предотвращает грязное чтение, но разрешает неповторяющееся и фантомное;

- TRANSACTION_REPEATABLE_READ — запрещает грязное и неповторяющееся чтение, но фантомное разрешено;
- TRANSACTION_SERIALIZABLE — определяет, что грязное, неповторяющееся и фантомное чтения запрещены

boolean supportsTransactionIsolationLevel(int level)

определяет, поддерживается ли заданный уровень изоляции транзакций.

int getTransactionIsolation()

текущий уровень изоляции

void setTransactionIsolation(int level)

устанавливает необходимый уровень

Точки сохранения

Назначение - дополнительный контроль над транзакциями, привязывая изменения СУБД к конкретной точке в области транзакции

Класс java.sql.Savepoint

логическая точка внутри транзакции, которая может быть использована для отката данных

```
rollback(Savepoint point)
boolean  supportsSavepoints()

Savepoint  setSavepoint(String name)
Savepoint  setSavepoint()
```

поддерживает ли точки сохранения драйвер JDBC и СУБД

установка именованной или неименованой точки сохранения во время текущей транзакции

шаблон Data Access Object

прослойка между приложением и СУБД - часть кода, ответственная за передачу запросов в БД и обработку полученных от нее ответов

DAO абстрагирует бизнес сущности системы и отражает их на записи в БД.

- определяет способы использования соединения с БД (открытие и закрытие или извлечения и возвращения в пул)
- Назначение : возможность подмены одной модели базы данных другой

Реализация DAO на уровне метода

ключ в таблице

общую бизнес-сущность

```
public abstract class AbstractDAO <K, T extends Entity> {
    public abstract List<T> findAll();
    public abstract T findEntityById(K id);
    public abstract boolean delete(K id);
    public abstract boolean delete(T entity);
    public abstract boolean create(T entity);
    public abstract T update(T entity);
}
```

Вершина иерархии DAO - класс или интерфейс с описанием общих методов (выбор, поиск сущности по признаку, добавление, удаление и замена информации)

```
class CardtDAO extends AbstractDAO <Integer, Card> {}
```

Реализация DAO на уровне класса

Предполагает использование одного коннекта к базе данных для вызова нескольких методов конкретного DAO класса

```
public abstract class AbstractDAO {
    protected Connection connector;
    // методы добавления, поиска, замены, удаления
// методы закрытия коннекта и Statement
    public void close() {
        connector.close();
    }
    protected void closeStatement(Statement statement) {
        statement.close();
    }
}
```

в методе не должны закрываться соединения

Соединение с базой данных инициирует конструктор DAO

```
class WrapperConnector {
   private Connection connection;
   public WrapperConnector() {
       try {
           ResourceBundle resource = ResourceBundle.getBundle("db.properties");
           String url = resource.getString("url");
           String user = resource.getString("user");
           String pass = resource.getString("password");
           Properties prop   =  new Properties();
           prop.put("user", user);
           prop.put("password", pass);
           connection = DriverManager.getConnection(url, prop);
          }  catch   (MissingResourceException e) {
           System.err.println("properties file is missing " + e);}
       catch   (SQLException e) {
           System.err.println("not obtained connection " + e);
       }
   }
   public Statement getStatement() throws SQLException {
       if (connection != null) {
           Statement statement = connection.createStatement();
           if (statement != null) {
               return statement;
           }
       }
       throw new SQLException("connection or statement is null");
   }
```

```
public void closeStatement(Statement statement) {
        if (statement != null) {
            try {
                statement.close();
            }  catch (SQLException e) {
                System.err.println("statement is null " + e);
            }
        }
    }
    public void closeConnection() {
        if (connection != null) {
            try {
                connection.close();
            }  catch (SQLException e) {
                System.err.println(" wrong connection" + e);
            }
        }
    }
// другие необходимые делегированные методы интерфейса Connection
}
```

DAO уровень логики

при выполнении запроса пользователя обращение сразу к нескольким ветвям DAO и использовать при этом единственное соединение с БД

```
public class SomeLogic {
    public void doLogic(int id) throws SQLException {
// 1. создание-получение соединения
        Connection  conn = ConnectionPool.getConnection();
// 2. открытие транзакции
        conn.setAutoCommit(false);
// 3. инициализация необходимых экземпляров DAO
        AbonentDAO abonentDAO = new AbonentDAO(conn);
        PaymentDAO paymentDAO = new PaymentDAO(conn);
// 4. выполнение запросов
        abonentDAO.findAll();
        paymentDAO.findEntityById(id);
        paymentDAO.delete(id);
//  5.  закрытие транзакции
        conn.commit();
// 6. закрытие-возвращение соединения
        ConnectionPool.close(conn);
    }
}
```

Альтернативы

- ORM (object-relational mapping) – Hibernate
- ORM - JPA (Java Persistentse API)