

# Работа с базами данных. Часть II

Croc Java School

---

**JDBC**



## Java Database Connectivity

JDBC — API, определяющий интерфейсы доступа к реляционным базам данных с целью извлечения и модификации данных таблиц (отношений).

В Java SE не входят реализации JDBC. Для работы с конкретной СУБД требуется включить **JDBC-драйвер** этой СУБД в качестве зависимости приложения.



## JDBC

JDBC API (Java) + JDBC драйвер (сторонняя библиотека)



## JDBC API

Задача JDBC API — предоставить набор интерфейсов и классов для выполнения SQL запросов к реляционным базам данных.



## JDBC драйвер

Задача JDBC драйвера — преобразовать методы приложения Java в протокол, специфичный для конкретной СУБД.

---

# JDBC API: Соединения



## Создание соединения

Любой запрос к базе данных выполняется в рамках соединения, которое предоставляет JDBC драйвер СУБД.

```
String connectionUrl = "jdbc:h2:tcp://localhost/~/test";  
Connection connection = DriverManager.getConnection(connectionUrl,  
    "sa", "");
```





## Строка подключения

По строке подключения DriverManager определяет, драйвер какой СУБД необходимо использовать.

```
jdbc:h2:tcp://localhost/~test
```

- Интерфейс доступа: JDBC
- Тип СУБД: H2 Database
- Хост: localhost
- Название бд: test



## Загрузка класса драйвера

Перед использованием класса драйвера к конкретной СУБД его необходимо загрузить в память JVM. Сделать это можно несколькими способами.

Инстанцировать экземпляр класса драйвера:

```
Driver driver = new org.h2.Driver();
```

Загрузить класс динамически с помощью вызова Class.forName:

```
Class.forName("org.h2.Driver");
```



## Заккрытие соединения

После окончания работы с соединением, его необходимо закрывать. Это позволит драйверу СУБД освободить ресурсы.

Connection реализует интерфейс `AutoCloseable`, поэтому можно пользоваться конструкцией `try with resources`.

```
try (Connection connection = DriverManager.getConnection(connectionUrl)) {  
    // выполнение запроса  
}
```



## Пулы соединений

Открытие нового соединения к базе данных — “тяжелая” операция. Часто в работе с базами данных применяется техника “пулов соединений”.

При закрытии соединения оно не уничтожается, но помечается как свободное, что позволяет предоставить его по следующему запросу без необходимости заново конструировать запрос.



## Транзакции

По умолчанию все запросы в рамках соединения выполняются в режиме auto-commit. Для перехода в режим ручного управления транзакциями его необходимо выключать.

```
connection.setAutoCommit(false);  
try {  
    // statements  
    connection.commit();  
} catch (SQLException e) {  
    connection.rollback();  
}
```



## Метаданные

```
DatabaseMetaData meta = connection.getMetaData();
```

```
ResultSet getTables(String catalog,  
    String schemaPattern,  
    String tableNamePattern,  
    String[] types)  
throws SQLException;
```

---

# JDBC API: Запросы



## Запросы

Выражения соответствуют запросам к базам данных.

Базовый класс для выражений: `java.sql.Statement`. Основные методы выполнения запросов:

```
ResultSet executeQuery(String sql) throws SQLException;
```

```
boolean execute(String sql) throws SQLException;
```

```
ResultSet getResultSet() throws SQLException;
```

```
int executeUpdate(String sql) throws SQLException;
```





## Пример. Выборка всех фигур

```
try (Statement statement = connection.createStatement()) {
    boolean hasResult = statement.execute("SELECT * FROM Figure");
    if (hasResult) {
        try (ResultSet result = statement.getResultSet()) {
            while (result.next()) {
                int id = result.getInt("id");
                String shape = result.getString("shape");
                String color = result.getString("color");
            }
        }
    }
}
```



## Пример. Выборка всех фигур (executeQuery)

```
try (Statement statement = connection.createStatement()) {  
    try (ResultSet result = statement  
        .executeQuery("SELECT * FROM Figure")) {  
        while (result.next()) {  
            int id = result.getInt("id");  
            String shape = result.getString("shape");  
            String color = result.getString("color");  
        }  
    }  
}
```



## Один Statement — один запрос

В каждый момент времени один Statement связан только с одним результатом выполнения. Вызов любого из методов `executeX` автоматически закрывает предыдущий `ResultSet`.



## Параметризированные запросы

Класс `java.sql.PreparedStatement` позволяет определить выражение, пригодное для **повторного использования** с различными наборами значений параметров. Для задания параметров в тексте запросов используются **маркеры**: знаки вопроса.

```
SELECT *  
FROM Figure f  
WHERE f.shape = ? AND f.color = ?
```

Параметры в таких запросах адресуются порядковым номером маркера в строке запроса (начиная с 1).



## Параметризированные запросы

```
SELECT *  
FROM Figure f  
WHERE f.shape = ? AND f.color = ?
```

В этом запросе два параметра:

- Форма фигуры с порядковым номером 1.
- Цвет с порядковым номером 2.

## Пример. Выборка фигур с заданным цветом

```
String color = "красный";
String sql = "SELECT * FROM Figure WHERE color = ?";
try (PreparedStatement statement = connection.prepareStatement(sql)) {
    statement.setString(1, color);
    try (ResultSet result = statement.executeQuery()) {
        while (result.next()) {
            int id = result.getInt("id");
            String shape = result.getString("shape");
        }
    }
}
```



## Пример. Изменение цвета заданной фигуры

```
int id = 1;
String color = "зеленый";
String sql = "UPDATE Figure SET color = ? WHERE id = ?";

try (PreparedStatement statement = connection.prepareStatement(sql)) {
    statement.setString(1, color);
    statement.setInt(2, id);
    statement.executeUpdate();
}
```



## Вызов хранимых процедур

Хранимые процедуры в SQL поддерживают параметры типов IN/OUT/INOUT. `java.sql.PreparedStatement` не позволяет задавать параметры режима OUT. Для этих целей используется `java.sql.CallableStatement`.

```
String sql = "call get_default_color(?)";  
try (CallableStatement statement = connection.prepareCall(sql)) {  
    statement.registerOutParameter(1, Types.VARCHAR);  
    statement.execute();  
    String defaultColor = statement.getString(1);  
}
```



---

# JDBC API: Курсоры



## Навигация по выборке

```
boolean absolute(int row) throws SQLException;  
boolean relative( int rows ) throws SQLException;  
boolean first() throws SQLException;  
boolean last() throws SQLException;  
boolean next() throws SQLException;  
boolean previous() throws SQLException;
```



## Тип курсора

`ResultSet.TYPE_FORWARD_ONLY`

Навигация по выборке допустима только вперед.

`ResultSet.TYPE_SCROLL_INSENSITIVE`

Навигация по выборке не учитывает изменения, внесенные после выполнения запроса, связанного с выборкой.

`ResultSet.TYPE_SCROLL_SENSITIVE`


Навигация по выборке учитывает изменения, внесенные после выполнения запроса, связанного с выборкой.



## Работа с данными выборки

```
// не изменяют бд
int getInt(int columnIndex) throws SQLException;
int getInt(String columnLabel) throws SQLException;
void updateInt(int columnIndex, int x) throws SQLException;
void updateInt(String columnLabel, int x) throws SQLException;
void moveToInsertRow() throws SQLException;
```

```
// изменяют бд
void updateRow() throws SQLException;
void insertRow() throws SQLException;
void deleteRow() throws SQLException;
```



## Режим доступа

`ResultSet.CONCUR_READ_ONLY`

Допускается только чтение.

`ResultSet.CONCUR_UPDATABLE`

Допускается чтение и запись.



## Тип курсора и режим доступа задаются при создании запроса

```
Statement statement = connection.createStatement(  
    ResultSet.TYPE_SCROLL_INSENSITIVE,  
    ResultSet.CONCUR_READ_ONLY);  
// ...
```

**Кажется, что с ResultSet  
МНОГО ВОЗНИ...**

---

—

ORM





## Представление данных в Java

В Java мы работаем с классами, пользоваться в логике наборами `ResultSet` неудобно. Отношения хорошо представимы в виде классов.



## Figure POJO — Plain Old Java Object

```
public class Figure {  
  
    int id;  
  
    String shape;  
  
    String color;  
  
    // getters and setters  
  
}
```



## DAO — Data Access Object

```
public class FigureDao {  
    public void create(Figure figure) {  
        // insert a new row into a table  
    }  
    public Figure read(int id) {  
        // convert table row to a Figure class  
    }  
    public void update(Figure figure) {  
        // update table row from a given class  
    }  
    public void delete(Figure figure) {  
        // delete row from a table  
    }  
}
```



Если мы реализуем `FigureDao`, то в дальнейшем в программе нам больше не придется заботиться о работе с базой данных.



## POJO для отношения Sign

```
public class Sign {  
  
    int id;  
  
    Figure figure; // foreign key  
  
    String label;  
  
    // getters and setters  
  
}
```



## DAO для отношения Sign

При реализации SignDao мы получим код, очень похожий на FaigureDao.

Программисты очень не любят писать несколько раз похожий код.

В случае с большими базами данных отношений, а соответственно, и DAO-классов может быть очень много.

**На помощь приходят ORM-  
библиотеки**

---



# ORM

## ORM — Object-Relationship Mapping

Библиотека (или фреймворк), задачей которой является автоматизация (через статическую генерацию или механизмы интроспекции) преобразований Java-классов в кортежи отношений базы данных.

По сути ORM нужен для того, чтобы получить DAO-классы всех отношений без необходимости реализовывать эти классы вручную.







## Пример использования requery

```
public interface Figure {  
    @Key @Generated int getId();  
    String getShape();  
    String getColor();  
}
```

```
Result<Figure> figures = data  
    .select(Figure.class)  
    .where(Figure.COLOR.equal("зеленый"))  
    .orderBy(Figure.SHAPE)  
    .get();
```

# ORM не всегда хорошо

---

---

NoSQL\*