1. **Что такое JDBC?**
2. **Как выполнить SQL-запрос в Java с использованием JDBC?**
3. **Какие преимущества использования JDBC перед другими методами доступа к базе данных?**

✅ 1. Что такое JDBC

JDBC (Java Database Connectivity) — это стандартный **API для работы с базами данных в Java**.

Основные особенности:

- Позволяет выполнять SQL-запросы из Java-приложения
- Поддерживает все основные типы SQL-операций (SELECT, INSERT, UPDATE, DELETE)
- Независим от конкретной СУБД (через JDBC-драйвер)
- Обеспечивает работу с транзакциями, подготовленными запросами (PreparedStatement) и результатами (ResultSet)

JDBC — это уровень **низкоуровневого доступа к БД**, на основе которого строятся более высокоуровневые фреймворки, например Spring Data или JPA.

✅ 2. Выполнение SQL-запроса через JDBC

Шаги:

1. **Загрузка драйвера СУБД**

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

2. **Подключение к базе**

```
Connection connection = DriverManager.getConnection(
    "jdbc:mysql://localhost:3306/mydb", "username", "password");
```

3. **Создание Statement / PreparedStatement**

```
// Простой Statement
Statement stmt = connection.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM employees");

// PreparedStatement для параметров
PreparedStatement ps = connection.prepareStatement(
    "SELECT * FROM employees WHERE id = ?");
ps.setInt(1, 100);
```

```
ResultSet rs2 = ps.executeQuery();
```

4. **Обработка результата**

```
while (rs.next()) {
    System.out.println(rs.getString("name") + " - " + rs.getInt("salary"));
}
```

5. **Закрытие ресурсов**

```
rs.close();
stmt.close();
connection.close();
```

✅ 3. Преимущества использования JDBC

Преимущество	Описание
Стандартность	Работает с любой СУБД при наличии драйвера
Прямой доступ к SQL	Можно писать любые SQL-запросы, без ограничений ORM
Поддержка транзакций	Возможность контролировать commit/rollback вручную
PreparedStatement	Защита от SQL-инъекций и повышение производительности
Гибкость	Можно интегрировать с любыми фреймворками или писать свои DAO

Минус: низкоуровневый код требует ручного управления ресурсами и транзакциями.
Для упрощения обычно используют Spring JDBC или JPA/Hibernate.

1. **Какие основные интерфейсы используются в JDBC?**
2. **Как управлять транзакциями в JDBC?**
3. **Как использовать PreparedStatement для защиты от SQL-инъекций?**

✅ 1. Основные интерфейсы JDBC

Интерфейс	Описание
Driver	Интерфейс драйвера СУБД, реализует соединение с базой
Connection	Представляет соединение с базой данных. Используется для выполнения SQL-запросов и управления транзакциями
Statement	Используется для выполнения статических SQL-запросов (SELECT, INSERT, UPDATE, DELETE)
PreparedStatement	Подготовленный SQL-запрос с параметрами. Более безопасный и эффективный, чем Statement
CallableStatement	Выполнение хранимых процедур
ResultSet	Результат выполнения SELECT-запроса, позволяет перебрать строки и получить значения колонок
ResultSetMetaData	Метаданные результата (названия и типы колонок)
DatabaseMetaData	Метаданные базы данных (информация о таблицах, колонках, поддерживаемых типах)

✅ 2. Управление транзакциями в JDBC

По умолчанию JDBC использует **автоматический коммит** после каждого запроса.

Отключение автокоммита и ручное управление:

```
Connection connection = DriverManager.getConnection(...);
connection.setAutoCommit(false); // отключаем автокоммит

try {
    Statement stmt = connection.createStatement();
    stmt.executeUpdate("INSERT INTO employees(name, salary) VALUES('Anton',
5000)");
    stmt.executeUpdate("UPDATE accounts SET balance = balance - 500 WHERE id
= 1");

    connection.commit(); // подтверждение изменений
} catch (SQLException e) {
    connection.rollback(); // откат при ошибке
} finally {
    connection.close();
}
```

Важно: всегда закрывать соединение и rollback в случае исключений, чтобы не оставить базу в неконсистентном состоянии.

✅ 3. Использование PreparedStatement для защиты от SQL-инъекций

Проблема: динамическая вставка данных в SQL через Statement опасна:

```
Statement stmt = connection.createStatement();
String sql = "SELECT * FROM users WHERE username = '" + userInput + "'";
ResultSet rs = stmt.executeQuery(sql); // уязвимо к SQL-инъекциям
```

Решение: PreparedStatement

```
String sql = "SELECT * FROM users WHERE username = ?";
PreparedStatement ps = connection.prepareStatement(sql);
ps.setString(1, userInput); // безопасная подстановка значения
ResultSet rs = ps.executeQuery();
```

Преимущества:

1. **Безопасность** — параметры автоматически экранируются, SQL-инъекция невозможна
2. **Повышенная производительность** — подготовленные запросы могут кешироваться драйвером
3. **Читабельность** — код становится более чистым при множественных параметрах

📌 Итоговое резюме

Тема	Суть
Основные интерфейсы	Driver, Connection, Statement, PreparedStatement, CallableStatement, ResultSet, DatabaseMetaData
Транзакции	`setAutoCommit(false)`, `commit()`, `rollback()`
Защита от SQL-инъекций	Использовать PreparedStatement и параметризованные запросы

---

1. **Как реализовать пул соединений с базой данных в JDBC?**
2. **Какие шаблоны проектирования часто используются при работе с JDBC?**
3. **Какие стратегии могут быть использованы для оптимизации производительности JDBC-приложений?**

✅ 1. Пул соединений (Connection Pool) в JDBC

Проблема: создание нового соединения к базе данных дорого по ресурсам и времени.

Решение: использовать **пул соединений** — заранее создать несколько соединений и переиспользовать их.

Популярные реализации:

- **HikariCP** — быстрый и легковесный
- **Apache DBCP**
- **C3P0**

Пример с HikariCP:

```
import com.zaxxer.hikari.HikariConfig;
import com.zaxxer.hikari.HikariDataSource;

HikariConfig config = new HikariConfig();
config.setJdbcUrl("jdbc:mysql://localhost:3306/mydb");
config.setUsername("root");
config.setPassword("password");
config.setMaximumPoolSize(10);

HikariDataSource dataSource = new HikariDataSource(config);

try (Connection conn = dataSource.getConnection()) {
    // работа с базой
}
```

Плюсы пула соединений:

- Снижение затрат на создание соединений
- Повышение производительности многопоточных приложений
- Контроль количества одновременных соединений

✅ 2. Шаблоны проектирования при работе с JDBC

Шаблон	Описание и применение
DAO (Data Access Object)	Абстракция доступа к данным. Изолирует SQL-запросы от бизнес-логики
Template Method (Spring JDBC Template)	Упрощает работу с JDBC: управление соединениями, закрытие ресурсов, обработка исключений
Factory (Connection Factory)	Создание и конфигурация соединений с базой данных через единый объект
Singleton	Используется для пула соединений или DataSource, чтобы была единая точка доступа к БД

Пример DAO:

```
public class EmployeeDAO {
    private DataSource dataSource;

    public EmployeeDAO(DataSource dataSource) {
        this.dataSource = dataSource;
    }

    public Employee getEmployeeById(int id) throws SQLException {
        String sql = "SELECT * FROM employees WHERE id = ?";
        try (Connection conn = dataSource.getConnection();
             PreparedStatement ps = conn.prepareStatement(sql)) {
            ps.setInt(1, id);
            ResultSet rs = ps.executeQuery();
            if (rs.next()) {
                return new Employee(rs.getInt("id"), rs.getString("name"));
            }
        }
        return null;
    }
}
```

✅ 3. Стратегии оптимизации производительности JDBC-приложений

1. **Пул соединений**
   - Повторное использование соединений вместо постоянного создания/закрытия
2. **PreparedStatement**
   - Кеширование плана выполнения запроса в базе
   - Защита от SQL-инъекций
3. **Batch Operations**
   - Выполнение нескольких вставок/обновлений за один вызов

```
PreparedStatement ps = conn.prepareStatement("INSERT INTO employees(name)
VALUES (?)");
for (String name : names) {
    ps.setString(1, name);
    ps.addBatch();
}
ps.executeBatch();
```

4. **Fetch Size и ResultSet**
   - Установка `stmt.setFetchSize(1000)` для больших выборок, чтобы избежать загрузки всей таблицы в память
5. **Транзакции**
   - Использование `setAutoCommit(false)` для группировки нескольких операций в одну транзакцию
6. **Индексы и оптимизация SQL**
   - Эффективные запросы и индексы ускоряют выборку данных
7. **Кэширование на стороне приложения**
   - Снижение количества обращений к базе данных для часто используемых данных

📌 Итоговое резюме

Тема	Суть
Пул соединений	Повышает производительность, контролирует количество соединений
Шаблоны проектирования	DAO, Template, Factory, Singleton для упрощения и структурирования кода
Оптимизация	PreparedStatement, batch-операции, fetch size, транзакции, индексы, кэширование

---

1. **Что такое CRUD?**
2. **Какова основная цель использования пула соединений в JDBC?**
3. **Какие преимущества предоставляет использование пула соединений при работе с базой данных?**

✅ 1. Что такое CRUD

CRUD — это аббревиатура для основных операций с данными в базе:

Буква	Расшифровка	SQL-пример
C	Create — создание	`INSERT INTO employees(name, salary) VALUES('Anton', 5000)`
R	Read — чтение	`SELECT * FROM employees WHERE id = 100`
U	Update — обновление	`UPDATE employees SET salary = 6000 WHERE id = 100`
D	Delete — удаление	`DELETE FROM employees WHERE id = 100`

Цель CRUD: стандартный набор операций для управления данными, используемый в приложениях и при проектировании БД.

✅ 2. Основная цель использования пула соединений в JDBC

Создание соединения с базой данных — дорогая по ресурсам операция.

Цель пула соединений:

- Создать заранее несколько соединений
- Повторно использовать их в приложении
- Снизить накладные расходы на открытие и закрытие соединений

✅ 3. Преимущества использования пула соединений

Преимущество	Описание
Повышение производительности	Не нужно создавать новое соединение для каждого запроса
Управление ресурсами	Ограничение максимального числа одновременно открытых соединений
Стабильность многопоточных приложений	Многим потокам можно быстро выдавать готовые соединения
Снижение нагрузки на базу	Меньше открытых и закрытых соединений → меньше затрат на сервер БД
Поддержка мониторинга и таймаутов	Современные пулы (HikariCP, DBCP) позволяют контролировать состояние соединений и предотвращать «зависание»

В целом, пул соединений — обязательная практика для высокопроизводительных Java-приложений, работающих с базой данных.

---

1. **Как можно реализовать пул соединений в Java приложении?**
2. **Какие есть стандартные библиотеки или фреймворки для работы с пулами соединений в Java?**
3. **Какие шаги необходимо выполнить для интеграции пула соединений с CRUD-приложением на Java?**

✅ 1. Как реализовать пул соединений в Java

В Java есть два подхода: **самостоятельная реализация** и **использование готовых библиотек**.

1.1 Самостоятельная реализация (очень простой вариант)

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.util.ArrayList;
import java.util.List;

public class SimpleConnectionPool {
    private List<Connection> pool = new ArrayList<>();
    private String url, user, password;
    private int poolSize;

    public SimpleConnectionPool(String url, String user, String password, int
poolSize) throws Exception {
        this.url = url; this.user = user; this.password = password;
this.poolSize = poolSize;
        for (int i = 0; i < poolSize; i++) {
            pool.add(DriverManager.getConnection(url, user, password));
        }
    }

    public synchronized Connection getConnection() {
        if (!pool.isEmpty()) {
            return pool.remove(0);
        }
        return null; // или ждать освобождения
    }

    public synchronized void releaseConnection(Connection conn) {
        pool.add(conn);
    }
}
```

Минусы: нет мониторинга, нет автоматического восстановления упавших соединений, сложно масштабировать.

1.2 Использование готовых библиотек (рекомендуется)

Библиотека	Особенности
HikariCP	Высокая производительность, легковесная, стандарт де-факто
Apache DBCP	Более старая, поддержка старых проектов
C3P0	Поддержка старых приложений, менее производительная
Tomcat JDBC Pool	Интеграция с Tomcat, удобен для Spring Boot

Пример с HikariCP:

```
import com.zaxxer.hikari.HikariConfig;
import com.zaxxer.hikari.HikariDataSource;

HikariConfig config = new HikariConfig();
config.setJdbcUrl("jdbc:mysql://localhost:3306/mydb");
config.setUsername("root");
config.setPassword("password");
config.setMaximumPoolSize(10);

HikariDataSource dataSource = new HikariDataSource(config);

try (Connection conn = dataSource.getConnection()) {
    // выполнение CRUD-операций
}
```

✅ 2. Шаги интеграции пула соединений с CRUD-приложением

1. **Выбор библиотеки пула**
   - HikariCP, DBCP, C3P0 и т.д.
2. **Настройка DataSource / пула**
   - URL базы данных, имя пользователя, пароль
   - Максимальное число соединений
   - Таймауты и политика проверки соединений
3. **Изменение DAO/CRUD-кода**
   - Вместо `DriverManager.getConnection` использовать `DataSource.getConnection()`
4. **Использование PreparedStatement**
   - Для безопасности и производительности при многократных запросах
5. **Закрытие соединений корректно**
   - В try-with-resources или через метод `releaseConnection` (если свой пул)
6. **Мониторинг пула (опционально)**
   - HikariCP предоставляет метрики через JMX
   - Позволяет отслеживать активные, свободные соединения

✅ 3. Преимущества интеграции пула с CRUD-приложением

Преимущество	Описание
Быстрое получение соединений	Не создаём новое соединение каждый раз
Контроль ресурсов	Ограничение числа одновременных соединений
Повышенная стабильность	Пул проверяет живость соединений и восстанавливает упавшие
Простая интеграция с DAO	Все CRUD-операции получают соединение из пула через DataSource
Поддержка транзакций	Транзакции работают как обычно через Connection

---

1. **Какие параметры конфигурации пула соединений важны для оптимизации производительности приложения и почему?**
2. **Какие проблемы могут возникнуть при использовании пула соединений и как их решить?**
3. **Как реализовать механизм проверки живучести соединений в пуле и зачем это нужно?**

✅ 1. Важные параметры конфигурации пула соединений

Параметр	Описание	Влияние на производительность
maximumPoolSize	Максимальное количество соединений в пуле	Слишком мало → очередь запросов → задержки; слишком много → нагрузка на БД
minimumIdle	Минимальное количество всегда готовых соединений	Быстрое обслуживание первых запросов без создания новых соединений
connectionTimeout	Время ожидания получения соединения из пула	Слишком короткое → ошибки при высокой нагрузке; слишком длинное → долгие задержки
idleTimeout	Время простоя соединения до закрытия	Контролирует освобождение неиспользуемых соединений
maxLifetime	Максимальное время жизни соединения	Позволяет избежать проблем с устаревшими или закрытыми соединениями на стороне БД
validationQuery / healthCheck	SQL-запрос для проверки живучести соединения	Обеспечивает, что соединение живое и корректное, предотвращает ошибки при использовании «мертвых» соединений

Важно: балансировка этих параметров зависит от нагрузки приложения и характеристик базы данных.

✅ 2. Возможные проблемы при использовании пула и способы решения

Проблема	Причина	Решение
Истощение пула (pool exhaustion)	Все соединения заняты, новые запросы ждут	Увеличить `maximumPoolSize`, оптимизировать использование соединений, уменьшить время жизни транзакций
Утечки соединений (connection leak)	Соединения не закрываются	Использовать try-with-resources или `finally` для закрытия, включить мониторинг утечек (HikariCP leakDetection)
Использование «мертвых» соединений	Соединение закрыто на стороне БД	Включить проверку живучести (`connectionTestQuery`, `testOnBorrow`)
Высокая конкуренция потоков	Много потоков ждут соединений	Подобрать правильный `maximumPoolSize`, использовать асинхронные операции

✅ 3. Проверка живучести соединений (Connection Validation)

Зачем нужно:

- Избежать использования закрытых/мертвых соединений
- Предотвратить ошибки выполнения SQL
- Поддерживать стабильность приложения

Как реализуется:

1. Validation query (HikariCP, DBCP)

```
config.setConnectionTestQuery("SELECT 1");
config.setValidationTimeout(1000); // проверка за 1 секунду
```

- Каждое соединение проверяется перед выдачей из пула

2. Проверка по API драйвера

```
if (!connection.isValid(2)) {
    connection.close();
    // создать новое соединение
}
```

3. Проверка при простое (idle)

- Пул может периодически проверять простые соединения через `idleTimeout` + `testWhileIdle`

- Удалять неработающие соединения