1. Назначение и возможности JDBC?

 JDBC— платформенно-независимый промышленный стандарт взаимодействия Java-приложений с различными [СУБД](https://ru.bmstu.wiki/%D0%A1%D0%A3%D0%91%D0%94_(%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0_%D0%A3%D0%BF%D1%80%D0%B0%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F_%D0%91%D0%B0%D0%B7%D0%B0%D0%BC%D0%B8_%D0%94%D0%B0%D0%BD%D0%BD%D1%8B%D1%85)), реализованный в виде пакета java.sql, входящего в состав Java SE . Технологии драйверов JDBC лежат в одной из четырех категорий. Приложения и апплеты могут обращаться к БД через JDBC API, используя новые драйвера на основе технологии JDBC.  Это стандарт взаимодействия с БД. Его интерфейс поставляется в стандартной JDK в виде пакета java.sql. Какой бы библиотекой для работы с базой данных вы ни пользовались, в ее основе почти всегда лежит низкоуровневый JDBC.

2. Что такое JDBC драйвер? Какие бывают типы?

**JDBC-драйвер** – реализация JDBC для определенной базы данных. В приложении может быть зарегистрировано несколько разных драйверов. При соединении к базе, нужный выбирается исходя из URL соединения. Например, для jdbc:mysql://localhost:3306/db\_name будет использован MySQL JDBC драйвер (при наличии). Драйверы делятся на 4 тип

**Тип 1 – JDBC-ODBC мост.** Делегирует работу с базой установленному в системе ODBC-драйверу. Платформозависимый. Не поддерживается с Java 8.  
  
**Тип 2 – Native API.** Делегирует работу с базой библиотеке, установленной в системе. Платформозависимый. Библиотеки в отличие от ODBC специфичны для конкретной базы, поэтому такой драйвер обычно работает быстрее.  
  
**Тип 3 – драйвер сетевого протокола.** Работает с промежуточным слоем (JavaEE сервером приложений), который транслирует запросы в сетевой протокол, с которым работает конкретная БД. Такой драйвер полностью реализован на Java.  
  
**Тип 4 – драйвер протокола БД/тонкий драйвер.** Полная реализация протокола взаимодействия с базой данных. Отличается от типа 3 тем, что здесь логика протокола находится не на внешнем промежуточном слое, а прямо в самом драйвере.

3. Какие шаги (перечислите) нужно выполнить для извлечения данных?

1. Подключение драйвера базы данных MySQL
2. Установка соединения с БД
3. Создание объекта для передачи запросов
4. Выполнение запроса
5. Обработка результатов выполнения запроса
6. Закрытие соединения, statement

4. Как и где задать параметры соединения?

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");

5. Перечислите основные JDBC API компоненты.

Основные компоненты JDBC Api включают:

1. DriverManager:
   * Это класс, использующийся для управления списком Driver (database drivers).
2. *Driver*:
   * Это интерфейс, использующийся для соединения коммуникации с базой данных, управления коммуникации с базой данных. Когда загружается Driver, программисту не нужно конкретно вызывать его.
3. *Connection*:
   * Интерфейс со всеми методами связи с базой данных. Он описывает коммуникационный контекст. Вся связь с базой данных осуществляется только через объект соединения (connection).
4. *Statement*:
   * Это интерфейс, включающий команду SQL отправленный в базу данных для анализа, обобщения, планирования и выполнения.

*ResultSet :*

* + *ResultSet* представляет набор записей, извлеченных из-за выполнения запроса.

6. Какая разница между еxecute, executeQuery, executeUpdate?

Класс **Statement** содержит три различных метода выполнения SQL-выражений : executeQuery, executeUpdate и execute, которые вызываются в зависимости от текста SQL-запроса.

**executeQuery**

Метод **executeQuery** используется в запросах, результатом которых является один единственный набор значений, таких как запросов типа SELECT.

**executeUpdate**

Метод **executeUpdate** следует использовать, как для выполнения операторов управления данными типа INSERT, UPDATE или DELETE (DML - Data Manipulation Language), так и для операторов определения структуры базы данных CREATE TABLE, DROP TABLE (DDL - Data Definition Language).

Результатом выполнения операторов INSERT, UPDATE, или DELETE является изменения одной или более строк таблицы.

Результатом выполнения метода **executeUpdate** является целочисленное значение, определяющее, сколько строк было модифицировано. Для выражений **DML**, которые не оперируют со строками, возвращаемое методом executeUpdate значение всегда равно нулю.

**execute**

Метод **execute** используется, когда операторы SQL возвращают более одного набора данных, более одного счетчика обновлений или и то, и другое. Такая возможность редко используется программистами.

Объекты Statement сами по себе не "помнят" SQL-выражение. Оно передается в качестве аргумента методов Statement.executeXXX.

Необходимо отметить, что **PreparedStatement**, наследующий все методы Statement, имеет свои реализации методов executeQuery, executeUpdate и execute. Объекты PreparedStatement не принимают SQL-выражения в виде аргументов этих методов, так как они уже содержат прекомпилированные SQL-выражения.

**CallableStatement** наследуют методы от PreparedStatement без параметров. Использование аргументов в методах executeXXX объектов PreparedStatement и CallableStatement приведет к генерации ошибки SQLException.

Объекты **Statement** закрываются автоматически сборщиком мусора виртуальной машины Java. Тем не менее рекомендуется закрывать их явно после того, как работа с ними завершена. Закрытие объектов Statement после их использования освобождает ресурсы СУБД и позволяет избежать проблем с памятью.

7. Расскажите об интерфейсе ResultSet.

Класс ResultSet представляет результирующий набор данных и обеспечивает приложению построчный доступ к результатам запросов. При обработке запроса **ResultSet** поддерживает указатель на текущую обрабатываемую строку.

Доступ к данным **ResultSet** обеспечивает посредством набора get-методов, которые организуют доступ к колонкам текущей строки. Метод ResultSet.next используется для перемещения к следующей строке ResultSet, делая ее текущей.

**ResultSet** содержит так называемый курсор, который позиционируется на текущей строке данных. При вызове метода **next**, курсор перемещается на следующую строку.

При открытии набора данных **ResultSet** курсор расположен перед первой строкой, и первый вызов next передвигает его на первую строку.

**ResultSet** хранит курсор до самого закрытия или пока не закроется родительский объект Statement.

Курсор для результирующей таблицы имеет имя. Если БД поддерживает позиционированные обновления или позиционированные удаления, то командам обновления или удаления можно передать в качестве параметра имя курсора, которое можно получить с помощью вызова метода getCursorName()

8. Как и для чего используют PreparedStatement?

Кроме класса Statement в java.sql мы можем использовать для выполнения запросов еще один класс - **PreparedStatement**. ►PreparedStatement - используется для часто повторяющихся запросов SQL ♣ Ускоряет обмен информацией с базой данных при многократном выполнении однотипных запросов ♣ невозможен sql injection attacks

9. Как и для чего используют транзакции?

Транзакции единица работы, обладающая свойствами ACID (Атомарность, Согласованность, Изолированность, Долговечность) интерфейс Connection commit() - подтверждает выполнение SQL-запросов rollback() - отменяет действие всех запросов SQL, начиная от последнего вызова commit()

10. Перечислите и поясните типы чтения данных в транзакциях. Что такое уровни изоляции?

В стандарте SQL описывается четыре уровня изоляции транзакций — Read uncommited (Чтение незафиксированных данных), Read committed (Чтение зафиксированных данных), Repeatable read (Повторяемое чтение) и Serializable (Сериализуемость).

По умолчанию в PostgreSQL уровень изоляции Read Committed. Такой уровень изоляции всегда позволяет видеть изменения внесённые успешно завершёнными транзакциями в оставшихся параллельно открытых транзакциях. В транзакции, работающей на этом уровне, запрос SELECT (без предложения FOR UPDATE/SHARE) видит только те данные, которые были зафиксированы до начала запроса; он никогда не увидит незафиксированных данных или изменений, внесённых в процессе выполнения запроса параллельными транзакциями. По сути запрос SELECT видит снимок базы данных в момент начала выполнения запроса. Однако SELECT видит результаты изменений, внесённых ранее в этой же транзакции, даже если они ещё не зафиксированы. Также заметьте, что два последовательных оператора SELECT могут видеть разные данные даже в рамках одной транзакции, если какие-то другие транзакции зафиксируют изменения после выполнения первого SELECT.

Изоляция уровня Serializable обеспечивает беспрепятственный доступ к базе данных транзакциям с SELECT запросами. Но для транзакций с запросами UPDATE и DELETE, уровень изоляции Serializable не допускает модификации одной и той же строки в рамках разных транзакций. При изоляции такого уровня все транзакции обрабатываются так, как будто они все запущены последовательно (одна за другой). Если две одновременные транзакции попытаются обновить одну и туже строку, то это будет не возможно. В таком случае PostgreSQL принудит транзакцию, вторую, да и все последующие, что пытались изменить строку к отмене (откату — ROLLBACK).

***Уровень изоляции*** задает степень защищенности выбираемых транзакцией данных от возможности изменения другими транзакциями. Прежде чем приступить к подробному рассмотрению существующих уровней изоляции, рассмотрим несколько сценариев, которые могут возникнуть, если не использовать блокировку, и, следовательно, отсутствует изоляция транзакций.

11. Что такое DatabaseMetaData интерфейс?

**Интерфейс** java.SQL.**DatabaseMetaData** предоставляет информацию об уровнях изолированности транзакций, которые поддерживаются данной СУБД. Уровень изоляции транзакции используемый СУБД можно задать с помощью метода setTransactionIsolation() объекта java.SQL.Connection.

12. Зачем используют шаблон DAO

**Шаблон** **DAO** **используется** для связи программы, написанной на Java с реляционными базами данных через интерфейс JDBC. JDBC API позволяет в приложениях **использовать** SQL-команды, являющиеся стандартным средством доступа к таблицам. Также **DAO** — это промежуточный слой, который скрывает от клиента реализацию взаимодействия с разными хранилищами данных, способы и механизмы хранения, предлагая при этом единые требования по функционалу в виде интерфейса.