JDBC ASSIGNMENT

1. **What is JDBC (Java Database Connectivity)?**

   JDBC (Java Database Connectivity) is a Java API that enables Java applications to interact with databases in a standardized way. It provides a set of classes and interfaces for:

- Connecting to a database
- Sending SQL queries or updates
- Retrieving and processing the results

Key Features of JDBC:

- Platform-independent access to databases
- Supports SQL-based communication
- Works with relational databases like MySQL, Oracle, PostgreSQL, etc.
- Allows for dynamic SQL execution (at runtime)

2. **Importance of JDBC in Java Programming**

- **Database Connectivity** – Enables Java applications to connect with databases like MySQL, Oracle, etc.
- **Platform Independence** – Works across different databases using standard JDBC drivers.
- **SQL Execution** – Allows executing SQL queries directly from Java code.
- **Data Handling** – Facilitates reading, updating, and managing data efficiently.
- **Integration** – Essential for building database-driven applications like web, desktop, and enterprise apps.

3. **JDBC Architecture: Driver Manager, Driver, Connection, Statement, and ResultSet**

   JDBC Architecture :

1. **DriverManager** – Loads and manages JDBC drivers; establishes database connections.
2. **Driver** – Interface that handles communication with the database.
3. **Connection** – Represents a link between Java application and database.
4. **Statement** – Used to send SQL queries to the database.
5. **ResultSet** – Stores the result of SELECT queries and allows data retrieval.

Flow:

Load Driver → Get Connection → Create Statement → Execute Query → Process ResultSet → Close Connection.

4. **Overview of JDBC Driver Types:**

Type 1: JDBC-ODBC Bridge Driver

a. Uses ODBC driver to connect to the database.
b. **Platform-dependent**, requires ODBC setup.
c. **Slow and outdated**, no longer recommended.

Type 2: Native-API Driver

d. Converts JDBC calls into native database API calls.
e. Requires **native libraries** on client machine.
f. Faster than Type 1 but **platform-dependent**.

Type 3: Network Protocol Driver

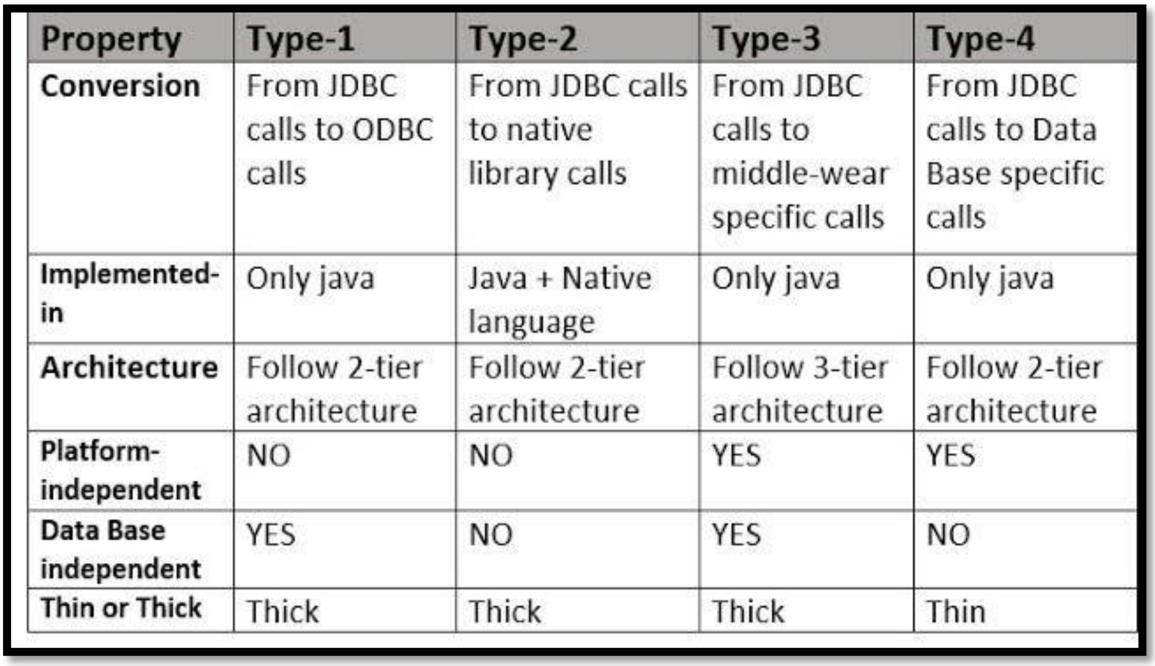
g. Uses a **middleware server** to communicate with the database.
h. Platform-independent and good for internet applications.
i. **Complex setup** due to middleware.

Type 4: Thin Driver

j. Pure Java driver; directly connects to the database.
k. **Platform-independent** and **most widely used**.
l. No native libraries or middleware required.

Type 4 is the most efficient and recommended driver in modern Java applications.

5. **Comparison and Usage of Each Driver Type**



Property	Type-1	Type-2	Type-3	Type-4
Conversion	From JDBC calls to ODBC calls	From JDBC calls to native library calls	From JDBC calls to middle-wear specific calls	From JDBC calls to Data Base specific calls
Implemented-in	Only java	Java + Native language	Only java	Only java
Architecture	Follow 2-tier architecture	Follow 2-tier architecture	Follow 3-tier architecture	Follow 2-tier architecture
Platform-independent	NO	NO	YES	YES
Data Base independent	YES	NO	YES	NO
Thin or Thick	Thick	Thick	Thick	Thin

6.Step-by-Step Process to Establish a JDBC Connection:

1. Import the JDBC packages:
   ```
   import java.sql.*;
   ```
2. Register the JDBC driver:
   ```
   Class.forName("com.mysql.cj.jdbc.Driver");  // Example for MySQL
   ```
3. Open a connection to the database:
   ```
   Connection conn = DriverManager.getConnection(
   "jdbc:mysql://localhost:3306/mydatabase", "username", "password");
   ```
4. Create a statement:
   ```
   Statement stmt = conn.createStatement();
   ```
5. Execute SQL queries:
   ```
   ResultSet rs = stmt.executeQuery("SELECT * FROM employees");
   ```
6. Process the ResultSet:
   ```
   while(rs.next()) {
   System.out.println("Name: " + rs.getString("name"));
   }
   ```
7. Close the connection:
   ```
   rs.close();
   stmt.close();
   conn.close();
   ```

7. Overview of JDBC Statements:

1. Statement:

- Executes simple SQL queries without parameters.
- Suitable for static SQL statements like SELECT * FROM table.
- Created using Connection.createStatement().

Example:

```
Statement stmt = conn.createStatement();

ResultSet rs = stmt.executeQuery("SELECT * FROM employees");
```

2. PreparedStatement:

- Represents precompiled SQL statements that can include parameters (placeholders ?).
- More efficient for repeated execution of similar queries with different values.
- Helps prevent SQL injection attacks by safely setting parameters.
- Created using Connection.prepareStatement(sql).

Example:

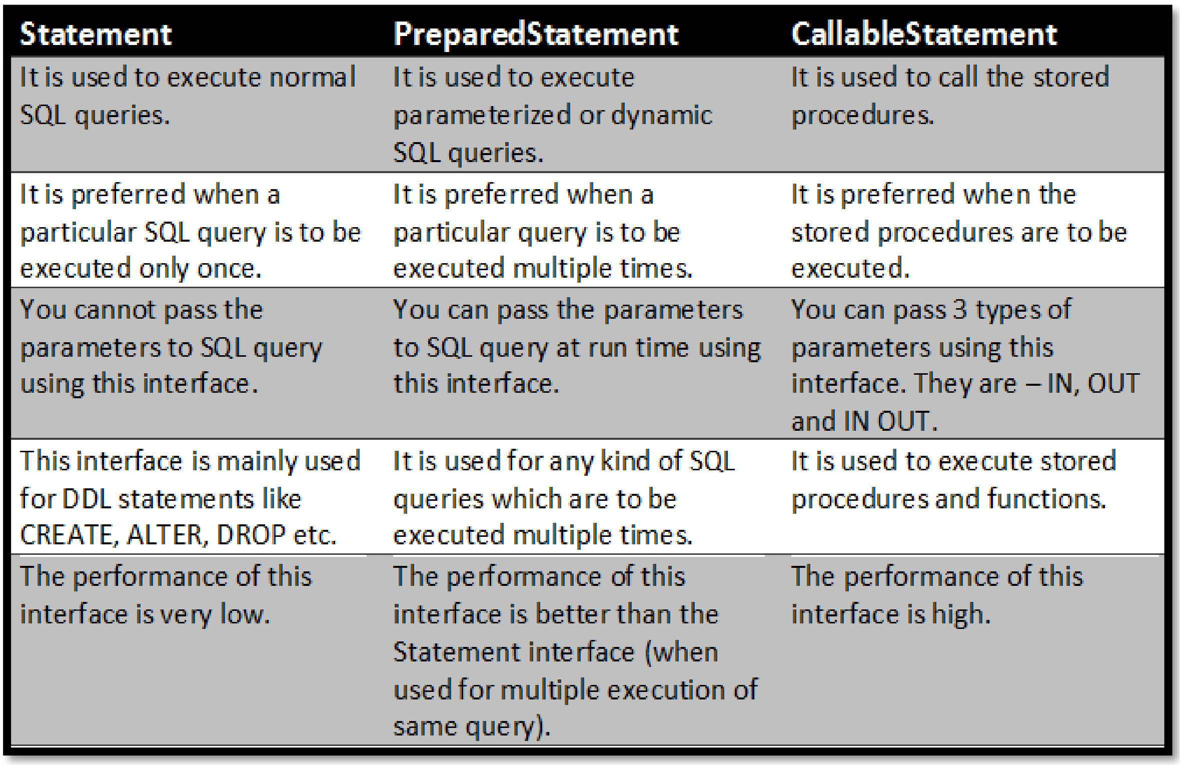
```
PreparedStatement pstmt = conn.prepareStatement("SELECT * FROM employees WHERE department = ?");
pstmt.setString(1, "Sales");
ResultSet rs = pstmt.executeQuery();
```

3. **CallableStatement:**
   - Used to call stored procedures in the database.
   - Supports both input and output parameters.
   - Created using Connection.prepareCall(sql).

Example:

```
CallableStatement cstmt = conn.prepareCall("{call getEmployeeDetails(?)}");
cstmt.s etInt(1, 101);
ResultSet rs = cstmt.executeQuery();
```

8. **Differences between Statement, PreparedStatement, and Callable Statement**



Statement	PreparedStatement	CallableStatement
It is used to execute normal SQL queries.	It is used to execute parameterized or dynamic SQL queries.	It is used to call the stored procedures.
It is preferred when a particular SQL query is to be executed only once.	It is preferred when a particular query is to be executed multiple times.	It is preferred when the stored procedures are to be executed.
You cannot pass the parameters to SQL query using this interface.	You can pass the parameters to SQL query at run time using this interface.	You can pass 3 types of parameters using this interface. They are – IN, OUT and IN OUT.
This interface is mainly used for DDL statements like CREATE, ALTER, DROP etc.	It is used for any kind of SQL queries which are to be executed multiple times.	It is used to execute stored procedures and functions.
The performance of this interface is very low.	The performance of this interface is better than the Statement interface (when used for multiple execution of same query).	The performance of this interface is high.

9. Insert: Adding a new record to the database. Update: Modifying existing records. Select: Retrieving recordsfrom the database. Delete: Removing records from the database.

- **Insert**: Adds a new record to the database.

  Example: INSERT INTO employees (id, name) VALUES (?, ?)
- **Update**:Modifies existing records in the database.

  Example: UPDATE employees SET name = ? WHERE id = ?
- **Select**:Retrieves records from the database.

  Example: SELECT * FROM employees WHERE id = ?
- **Delete**:Removes records from the database.

  Example: DELETE FROM employees WHERE id = ?

10. What is ResultSet in JDBC?

ResultSet is an object in JDBC that **holds the data retrieved from the database** after executing a SELECT query. It acts like a table of data representing the result of the query and allows you to **read row by row** and access column values.

Key Points:

- Obtained by executing a query using Statement or PreparedStatement.
- Supports methods like next() to move through rows.
- Provides getter methods like getString(), getInt(), etc., to fetch column values.
- Can be **scrollable** or **updatable** depending on how it's created.

Example:

```
ResultSet rs = stmt.executeQuery("SELECT * FROM employees");
while (rs.next()) {
String name = rs.getString("name");
int id = rs.getInt("id");
System.out.println(id + ": " + name);
}
```

11. Navigating through ResultSet (first, last, next, previous)

- next() — Moves to the next row.
- previous() — Moves to the previous row (scrollable ResultSet).
- first() — Moves to the first row.
- last() — Moves to the last row.

  Note: For previous(), first (), and last (), create a scrollable ResultSet:

```
Statement stmt = conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_READ_ONLY);
ResultSet rs = stmt.executeQuery("SELECT * FROM table");
```

12. Working with ResultSet to retrieve data from SQL queries

Steps to Retrieve Data:

- **Execute Query**
  Use executeQuery() method to run the SQL SELECT statement.

```
ResultSet rs = stmt.executeQuery("SELECT * FROM employees");
```

- **Navigate Through ResultSet**
  Use rs.next() to move the cursor to the next row (starts before the first row).

```
while (rs.next()) {
// Access data here
}
```

1. **RetrieveColumn Data**
   Use getter methods like getString(), getInt(), etc., to fetch column values by column name or index.

```
String name = rs.getString("name");

int id = rs.getInt(1);  // 1-based column index
```

2. **Close Resources**
   Close ResultSet, Statement, and Connection to free resources.

```
rs.close();

stmt.close();

conn.close();
```

Example:

```
Statement stmt = conn.createStatement();

ResultSet rs = stmt.executeQuery("SELECT id, name FROM employees");

while (rs.next())

{

int id = rs.getInt("id");

String name = rs.getString("name");
System.out.println(id + ": " + name);

}

rs.close();

stmt.close();
```

13. What is DatabaseMetaData?

DatabaseMetaData is an interface in JDBC that provides **information about the database** and its capabilities. It allows a Java program to retrieve metadata such as:

- Database product name and version
- Supported SQL keywords and functions
- List of tables, columns, and their types
- Supported features like transactions, stored procedures, etc.

➢ You get a DatabaseMetaData object from a Connection:

- DatabaseMetaData meta = conn.getMetaData();
- System.out.println("Database Product Name: " + meta.getDatabaseProductName());
- System.out.println("Database Version: " + meta.getDatabaseProductVersion());

14. Importance of Database Metadata in JDBC

- **Database Information:** Helps retrieve details like database name, version, and supported SQL features.
- **Dynamic Queries:** Enables writing database-independent code by adapting to different databases.
- **Schema Discovery:** Allows programs to find tables, columns, primary keys, and indexes at runtime.
- **Tool Support:** Useful for building database tools, report generators, and IDEs.
- **Feature Detection:** Helps check if certain features (e.g., transactions, stored procedures) are supported.

15. **Methods provided by DatabaseMetaData (getDatabaseProductName, getTables, etc.)**

- getDatabaseProductName (): Returns the name of the database (e.g., "MySQL", "Oracle").
- getDatabaseProductVersion():Returns the version of the database.
- getDriverName():Returns the name of the JDBC driver.
- getTables(String catalog, String schemaPattern, String tableNamePattern, String[] types):Retrieves a list of tables and views in the database.
- getColumns(String catalog, String schemaPattern, String tableNamePattern, String columnNamePattern):Retrieves information about columns in a table.
- getPrimaryKeys(String catalog, String schema, String table):Returns primary key columns for a table.
- supportsTransactions():Checks if the database supports transactions.
- getSchemas():Returns available database schemas.

16. **What is ResultSetMetaData?**

ResultSetMetaData is an interface in JDBC that provides **information about the columns of a ResultSet** returned by a SQL query. It helps you get details like:

- Number of columns
- Column names and types
- Column display size
- Whether columns allow NULL values

Usage:

- ➢ You obtain it from a ResultSet object:
  - ▪ ResultSetMetaData rsMeta = rs.getMetaData();
  - ▪ int columnCount = rsMeta.getColumnCount();
  - ▪ String columnName = rsMeta.getColumnName(1);

17. **Importance of ResultSet Metadata in analyzing the structure of query results**

- **Dynamic Data Handling:** Allows programs to work with any query result without knowing column details beforehand.
- **Column Information:** Provides column count, names, types, and sizes to process data correctly.
- **Flexible UI Generation:** Enables building generic data viewers or reports based on result structure.
- **Error Prevention:** Helps validate data types before processing to avoid runtime errors.

- **Metadata-Driven Logic:** Supports writing adaptable code that adjusts to different queries or databases.

18. Methods in ResultSetMetaData (getColumnCount, getColumnName, getColumnType)

- **getColumnCount()**:Returns the number of columns in the ResultSet.
- **getColumnName(int column)**:Returns the name of the specified column (1-based index).
- **getColumnType(int column)**:Returns the SQL type of the specified column as an int (from java.sql.Types).
- **getColumnTypeName(int column)**:Returns the database-specific type name of the column.
- **getColumnDisplaySize(int column)**:Returns the maximum width of the column for display.
- **isNullable(int column)**:Indicates if the column allows NULL values.

Example:

```
ResultSetMetaData rsMeta = rs.getMetaData();
int count = rsMeta.getColumnCount();

for (int i = 1; i <= count; i++) {
System.out.println("Column " + i + ": " + rsMeta.getColumnName(i) +
", Type: " + rsMeta.getColumnTypeName(i));
}
```

19. Introduction to Java Swing for GUI development

Java Swing is a **lightweight, platform-independent GUI toolkit** included in the Java Standard Edition. It provides a rich set of components to build graphical user interfaces such as windows, buttons, text fields, tables, and more.

Key Features:

- **Platform-independent:** Works consistently across different operating systems.
- **Rich Components:** Includes buttons, labels, lists, tables, trees, and more.
- **Customizable:** Supports look-and-feel themes and custom component rendering.
- **Event-driven:** Uses listeners to handle user interactions like clicks and typing.
- **Built on AWT:** Extends the older Abstract Window Toolkit (AWT) but is more flexible and powerful.

Typical Swing Components:

- JFrame — Main window
- JPanel — Container for grouping components
- JButton — Clickable button
- JLabel — Text label
- JTextField — Single-line text input
- JTable — Data table

Example:

```
import javax.swing.*;
public class HelloSwing {
public static void main(String[] args) {
JFrame frame = new JFrame("My Swing App");
JButton button = new JButton("Click Me");
frame.add(button);
frame.setSize(300, 200);
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setVisible(true);
}
}
```

20. How to integrate Swing components with JDBC for CRUD operations

1. **Create Swing UI Components**: Design forms with components like JTextField (input), JButton (actions), and JTable (display data).
2. **Establish JDBC Connection**: Use DriverManager.getConnection() to connect to the database.
3. **Handle User Actions**: Add event listeners to buttons (e.g., Insert, Update, Delete) to trigger JDBC operations.

- **Perform CRUD Operations:**
  - **Create:** Use PreparedStatement to insert data from text fields.
  - **Read:** Execute SELECT queries and display results in JTable.
  - **Update:** Use PreparedStatement with UPDATE SQL, based on user input.
  - **Delete:** Use DELETE queries triggered by button clicks.
  - **Refresh: After** CRUD operations, reload data into JTable to show current database state.

21. What is a CallableStatement?

CallableStatement is a JDBC interface used to **execute stored procedures** in a database. Stored procedures are precompiled SQL code stored in the database that can accept input parameters and return results or output parameters.

Key Points:

- Used to call database stored procedures.
- Supports input, output, and input-output parameters.
- Created using Connection.prepareCall() method.
- Can execute complex database operations efficiently.

Example:

```
CallableStatement cstmt = conn.prepareCall("{call getEmployeeDetails(?)}");
cstmt.setInt(1, 101);
ResultSet rs = cstmt.executeQuery();
```

22. How to call stored procedures using CallableStatement in JDBC

1. **Create a CallableStatement**
   Use Connection.prepareCall() with the stored procedure call syntax:

   ```
   CallableStatement cstmt = conn.prepareCall("{call procedure_name(?, ?)}");
   ```

   The question marks (?) represent parameters.

2. **Set Input Parameters**
   Use methods like setInt(), setString() to pass input values:

   ```
   cstmt.setInt(1, 123);
   ```

   ```
   cstmt.setString(2, "John");
   ```

3. **Register Output Parameters (if any)**
   For output parameters, register their SQL types:

   ```
   cstmt.registerOutParameter(3, java.sql.Types.VARCHAR);
   ```

4. **Execute the CallableStatement**
   Use execute (), execute Query(), or executeUpdate() depending on procedure type:

   ```
   cstmt.execute();
   ```

5. **Retrieve Output Parameters (if any)**
   After execution, get output values:

   ```
   String result = cstmt.getString(3);
   ```

6. **Close Resources**
   Close the CallableStatement and connection.

23. Working with IN and OUT parameters in stored procedures

Stored procedures can use **IN**, **OUT**, or **INOUT** parameters to pass data **to** and **from** the procedure.

Definitions:

- **IN**: Passes a value *into* the procedure.
- **OUT**: Sends a value *out* of the procedure (set inside it).
- **INOUT**: Passes a value in and allows it to be modified and returned.

Example in MySQL:

🔷 **Creating the stored procedure:**

```
DELIMITER //

CREATE PROCEDURE GetEmployeeNameById(
IN emp_id INT,
OUT emp_name VARCHAR (100)
)
BEGIN
SELECT name INTO emp_name
FROM employees
WHERE id = emp_id;
END //

DELIMITER ;
```

🔷 **Calling the procedure in Java (JDBC):**

```
import java.sql.*;
public class StoredProcExample
{
public static void main(String[] args) throws Exception
{
Connection conn = DriverManager.getConnection(
"jdbc:mysql://localhost:3306/mydb", "user", "pass");

CallableStatement stmt = conn.prepareCall("{ call GetEmployeeNameById(?, ?) }");

// Set IN parameter
stmt.setInt(1, 101);

// Register OUT parameter
stmt.registerOutParameter(2, Types.VARCHAR);

// Execute
stmt.execute();

// Get OUT parameter value
String empName = stmt.getString(2);
System.out.println("Employee Name: " + empName);

stmt.close();
conn.close();
}
}
```