Q1). What is JDBC (Java Database Connectivity)?

Ans: JDBC ek **Java API** (Application Programming Interface) hai jo Java applications ko **relational database** se connect karne aur usme data access karne ki suvidha deta hai. Iska use karke hum **Java code ke zariye database me SQL queries** (jaise ki SELECT, INSERT, UPDATE, DELETE) chala sakte hain.
JDBC ka istemal tab hota hai jab hume kisi database se data lena ho ya usme koi data store/update/delete karna ho. Ye Java aur database ke beech **bridge** ka kaam karta hai.

JDBC ke Components:

1. **DriverManager** – Database drivers ko manage karta hai.
2. **Connection** – Database se connection establish karta hai.
3. **Statement** – SQL queries ko run karta hai.
4. **ResultSet** – Query ka result store karta hai.
5. **SQLException** – Errors ko handle karta hai.

.

Q2). Importance of JDBC in Java Programming

Ans: Importance of JDBC in Java Programming

1. Database Connectivity: JDBC enables Java applications to connect to various relational databases, allowing data storage, retrieval, and manipulation.
2. Platform Independence: JDBC provides a standardized API, making Java applications database-independent and portable across different platforms.
3. Standardized API: JDBC offers a uniform interface for interacting with databases, simplifying development and reducing database-specific complexities.
4. SQL Support: JDBC allows execution of SQL queries, enabling developers to leverage database features and perform complex data operations.
5. Transaction Management: JDBC supports transaction management, ensuring data consistency and integrity in database operations.

Q3). JDBC Architecture: Driver Manager, Driver, Connection, Statement, and ResultSet

Ans: JDBC Architecture Components

1. Driver Manager: Manages JDBC drivers, allowing applications to connect to databases.
2. Driver: Database-specific implementation of the JDBC API, translating Java calls into database-specific protocols.
3. Connection: Represents a physical connection to the database, enabling SQL execution and transaction management.

4. Statement: Used to execute SQL queries, including SELECT, INSERT, UPDATE, and DELETE statements.
5. ResultSet: Holds the results of a query, providing access to data retrieved from the database.

Q4). Overview of JDBC Driver Types:

- **Type 4: Thin Drive**
- **Type 3: Network Protocol Driver**
- **Type 2: Native-API Driver**
- **Type 1: JDBC-ODBC Bridge Driver**

Ans: JDBC Driver Types

Overview of Each Type

1. Type 1: JDBC-ODBC Bridge Driver:
    - Acts as a bridge between JDBC and ODBC drivers
    - Uses ODBC driver to connect to the database
    - Not recommended for production use due to performance and security concerns
2. Type 2: Native-API Driver:
    - Uses native database APIs to connect to the database
    - Requires database-specific client libraries on the client-side
    - Platform-dependent and may have performance benefits
3. Type 3: Network Protocol Driver:
    - Uses a middleware server to connect to the database
    - Translates JDBC calls into database-specific network protocol
    - Provides flexibility and scalability
4. Type 4: Thin Driver:
    - Directly communicates with the database using database-specific network protocol
    - No middleware or native API required
    - Platform-independent, fast, and efficient

Q5). Comparison and Usage of Each Driver Type

Ans: Comparison of JDBC Driver Types

Driver Type	Description	Advantages	Disadvantages
Type 1: JDBC-ODBC Bridge	Bridge between JDBC and ODBC	Easy to use, supports many databases	Performance issues, security concerns, not recommended for production

|

| Type 2: Native-API Driver | Uses native database APIs | Good performance, database-specific features | Platform-dependent, requires client libraries |

| Type 3: Network Protocol Driver | Middleware-based, translates JDBC calls | Flexible, scalable, supports multiple databases | Additional middleware layer, potential performance overhead |

| Type 4: Thin Driver | Direct database connection, no middleware | Fast, efficient, platform-independent | Database-specific, may require specific configuration |

Q6). Step-by-Step Process to Establish a JDBC Connection:

1. **Import the JDBC packages**
2. **Register the JDBC driver**
3. **Open a connection to the database**
4. **Create a statement**
5. **Execute SQL queries**
6. **Process the result set**
7. **Close the connection**

Ans: Establishing a JDBC Connection

Step-by-Step Process

1. Import JDBC Packages:
    - Import necessary JDBC classes, such as `java.sql.Connection`, `java.sql.DriverManager`, `java.sql.Statement`, and `java.sql.ResultSet`.
2. Register JDBC Driver:
    - Register the JDBC driver using `Class.forName()` or `DriverManager.registerDriver()`.
3. Open Connection:
    - Use `DriverManager.getConnection()` to establish a connection to the database, providing the database URL, username, and password.
4. Create Statement:
    - Create a `Statement` object using the `Connection` object's `createStatement()` method.
5. Execute SQL Queries:
    - Use the `Statement` object to execute SQL queries, such as `SELECT`, `INSERT`, `UPDATE`, or `DELETE`.

6. Process Result Set:
    - If executing a `SELECT` query, process the `ResultSet` object to retrieve the query results.
7. Close Connection:
    - Close the `Connection` object to release resources and free up database connections.

Q7). Overview of JDBC Statements:

- **Statement: Executes simple SQL queries without parameters**
- **CallableStatement: Used to call stored procedures**
- **PreparedStatement: Precompiled SQL statements for queries withparameters.**

Ans: JDBC Statements Overview

Types of JDBC Statements

1. Statement:
    - Executes simple SQL queries without parameters.
    - Suitable for static SQL queries.
    - May be vulnerable to SQL injection attacks.
2. PreparedStatement:
    - Precompiled SQL statements for queries with parameters.
    - Improves performance and security by preventing SQL injection attacks.
    - Parameters can be set using `setInt()`, `setString()`, etc.
3. CallableStatement:
    - Used to call stored procedures in the database.
    - Supports input and output parameters.
    - Enables interaction with database-specific stored procedures.

Comparison

- Performance: PreparedStatement is generally faster due to precompilation.
- Security: PreparedStatement and CallableStatement are more secure than Statement.
- Flexibility: CallableStatement provides flexibility in interacting with stored procedures.

Q8). Differences between Statement, PreparedStatement, and CallableStatement
Ans: Differences Between Statement, PreparedStatement, and CallableStatement

Statement
1. Purpose: Executes simple SQL queries without parameters.
2. Security: Vulnerable to SQL injection attacks.
3. Performance: May have performance issues due to compilation overhead.

PreparedStatement
1. Purpose: Executes precompiled SQL queries with parameters.
2. Security: Prevents SQL injection attacks by parameterizing queries.
3. Performance: Improves performance due to precompilation.

CallableStatement
1. Purpose: Executes stored procedures in the database.
2. Security: Supports secure interaction with stored procedures.
3. Flexibility: Enables interaction with database-specific stored procedures.

Key Differences
- Parameter Handling: PreparedStatement and CallableStatement support parameterized queries, while Statement does not.
- Security: PreparedStatement and CallableStatement are more secure than Statement due to parameterization.
- Performance: PreparedStatement and CallableStatement generally offer better performance due to precompilation.

Q9). Insert: Adding a new record to the database
Ans: Inserting Data into a Database

Using JDBC
To insert data into a database using JDBC, you can follow these steps:

1. Create a `PreparedStatement` object with an INSERT query.
2. Set the parameter values using `setInt()`, `setString()`, etc.
3. Execute the query using `executeUpdate()`.

Q10). Update: Modifying existing records.
Ans: Updating Data in a Database

Using JDBC
To update existing records in a database using JDBC, follow these steps:

1. Create a `PreparedStatement` object with an UPDATE query.
2. Set the parameter values using `setInt()`, `setString()`, etc.
3. Execute the query using `executeUpdate()`.

Q11). Select: Retrieving records from the database
Ans: Retrieving Records from a Database

Using JDBC
To retrieve records from a database using JDBC, follow these steps:

1. Create a `PreparedStatement` object with a SELECT query.
2. Set parameter values using `setInt()`, `setString()`, etc. (if needed).
3. Execute the query using `executeQuery()`.
4. Process the `ResultSet` object to retrieve data.

Q12). Delete: Removing records from the database
Ans: Deleting Records from a Database

Using JDBC
To delete records from a database using JDBC, follow these steps:

1. Create a `PreparedStatement` object with a DELETE query.
2. Set parameter values using `setInt()`, `setString()`, etc. (if needed).
3. Execute the query using `executeUpdate()`.

Q13). What is ResultSet in JDBC?
Ans: ResultSet in JDBC

A `ResultSet` object represents a set of data retrieved from a database after executing a SQL query. It provides a way to access and manipulate the data returned by the query.

Key Features
1. Cursor Navigation: `ResultSet` provides methods to navigate through the result set, such as `next()`, `previous()`, `first()`, and `last()`.

2. Data Access: You can access data using column names or indices, with methods like `getString()`, `getInt()`, `getDate()`, etc.

3. Data Types: `ResultSet` supports various data types, including strings, integers, dates, and more.

Types of ResultSets
1. Forward-only: The default type, allowing only forward navigation.
2. Scrollable: Allows navigation in both forward and backward directions.
3. Updatable: Enables updating data in the result set.

Q14). Navigating through ResultSet (first, last, next, previous)
Ans: Navigating through ResultSet

Methods for Navigation
1. `next()`: Moves the cursor to the next row in the result set.

2. `previous()`: Moves the cursor to the previous row in the result set (only applicable for scrollable result sets).

3. `first()`: Moves the cursor to the first row in the result set (only applicable for scrollable result sets).

4. `last()`: Moves the cursor to the last row in the result set (only applicable for scrollable result sets).

Creating Scrollable ResultSets
To use `previous()`, `first()`, or `last()`, create a scrollable result set:

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

Example Usage

```
if (rs.next()) {
   // Process the first row
   System.out.println(rs.getString("name"));

   // Move to the last row
   if (rs.last()) {
      System.out.println(rs.getString("name"));
   }

   // Move to the previous row
   if (rs.previous()) {
      System.out.println(rs.getString("name"));
   }
}
```

Q15). Working with ResultSet to retrieve data from SQL queries

Ans: Retrieving Data from ResultSet

Accessing Data

1. By Column Index: Use `getString(1)`, `getInt(2)`, etc.
2. By Column Name: Use `getString("column_name")`, `getInt("id")`, etc.

Example

```
ResultSet rs = stmt.executeQuery("SELECT id, name, email FROM customers");

while (rs.next()) {
   int id = rs.getInt("id");
   String name = rs.getString("name");
   String email = rs.getString("email");

   System.out.println("ID: " + id + ", Name: " + name + ", Email: " + email);
}
```

Retrieving Different Data Types

1. `getString()`: Retrieves string data.

2. `getInt()`: Retrieves integer data.
3. `getDate()`: Retrieves date data.
4. `getBoolean()`: Retrieves boolean data.

Q16). What is DatabaseMetaData?
Ans: DatabaseMetaData

`DatabaseMetaData` is an interface in JDBC that provides information about the database, such as:

1. Database Product Information: Database name, version, and driver information.

2. Database Features: Support for SQL features, such as transactions, stored procedures, and batch updates.

3. Database Objects: Information about tables, views, procedures, and other database objects.

Retrieving DatabaseMetaData

```
DatabaseMetaData metaData = conn.getMetaData();

// Get database product information
String dbName = metaData.getDatabaseProductName();
String dbVersion = metaData.getDatabaseProductVersion();

// Get database features
boolean supportsTransactions = metaData.supportsTransactions();
```

Q17). Importance of Database Metadata in JDBC
Ans: Importance of Database Metadata in JDBC

Key Benefits
1. Database Independence: Write code that adapts to different database systems.
2. Dynamic Query Generation: Generate queries based on database structure.
3. Feature Detection: Determine database support for specific features.
4. Database Discovery: Explore database structure and objects.

Use Cases
1. Database Tooling: Develop tools that work with multiple databases.
2. ORM Frameworks: Implement Object-Relational Mapping (ORM) frameworks.
3. Query Builders: Build dynamic query generators.

Advantages
1. Flexibility: Adapt to changing database structures or features.
2. Portability: Write code that works across different database systems.
3. Automated Database Exploration: Automate database discovery and analysis.

Q18). Methods provided by DatabaseMetaData (getDatabaseProductName, getTables, etc.)
Ans: Methods Provided by DatabaseMetaData

Database Product Information*
1. `getDatabaseProductName()`: Returns the database product name.
2. `getDatabaseProductVersion()`: Returns the database product version.

Database Features
1. `supportsTransactions()`: Returns whether the database supports transactions.
2. `supportsStoredProcedures()`: Returns whether the database supports stored procedures.

Database Objects
1. `getTables()`: Returns a ResultSet containing information about the tables in the database.
2. `getColumns()`: Returns a ResultSet containing information about the columns in a table.
3. `getProcedures()`: Returns a ResultSet containing information about the stored procedures in the database.

Other Methods
1. `getDriverName()`: Returns the JDBC driver name.
2. `getDriverVersion()`: Returns the JDBC driver version.
3. `getSQLKeywords()`: Returns a comma-separated list of SQL keywords.

Q19). What is ResultSetMetaData?
Ans: ResultSetMetaData

`ResultSetMetaData` is an interface in JDBC that provides information about the structure and properties of a `ResultSet` object, such as:

1. Column Information: Number of columns, column names, data types, and properties.
2. Column Properties: Column size, precision, scale, and nullability.

Methods
1. `getColumnCount()`: Returns the number of columns in the ResultSet.
2. `getColumnName()`: Returns the name of a column.
3. `getColumnType()`: Returns the SQL type of a column.
4. `getColumnTypeName()`: Returns the database-specific type name of a column.

Use Cases
1. Dynamic Data Processing: Process data without knowing the column structure beforehand.
2. Data Validation: Validate data based on column properties.
3. Data Transformation: Transform data based on column types and properties.

Q20). Importance of ResultSet Metadata in analyzing the structure of query results

Ans: Importance of ResultSet Metadata

Key Benefits
1. Dynamic Data Analysis: Analyze query results without prior knowledge of column structure.
2. Column Information: Retrieve column names, data types, and properties.
3. Data Validation: Validate data based on column properties.
4. Data Transformation: Transform data based on column types and properties.

Use Cases
1. Data Exploration: Explore query results and understand column structure.
2. Dynamic Reporting: Generate reports based on query results.
3. Data Integration: Integrate data from different sources.

Advantages
1. Flexibility: Handle dynamic or unknown data structures.
2. Data Integrity: Ensure data consistency and validity.
3. Improved Error Handling: Handle errors and exceptions more effectively.

Q21). Methods in ResultSetMetaData (getColumnCount, getColumnName, getColumnType)
Ans: Methods in ResultSetMetaData

Column Information
1. `getColumnCount()`: Returns the number of columns in the ResultSet.
2. `getColumnName(int column)`: Returns the name of a column.
3. `getColumnLabel(int column)`: Returns the label of a column.

Column Data Types
1. `getColumnType(int column)`: Returns the SQL type of a column.
2. `getColumnTypeName(int column)`: Returns the database-specific type name of a column.

Column Properties
1. `isNullable(int column)`: Returns whether a column allows null values.
2. `getPrecision(int column)`: Returns the precision of a column.
3. `getScale(int column)`: Returns the scale of a column.

Example Usage

```
ResultSet rs = stmt.executeQuery("SELECT * FROM customers");
ResultSetMetaData metaData = rs.getMetaData();

int columnCount = metaData.getColumnCount();
for (int i = 1; i <= columnCount; i++) {
    String columnName = metaData.getColumnName(i);
    String columnType = metaData.getColumnTypeName(i);
    System.out.println("Column Name: " + columnName + ", Type: " + columnType);
}
```

Q22). Write SQL queries for:
- **Inserting a record into a table**
- **Updating specific fields of a record.**
- **Selecting records based on certain conditions.**
- **Deleting specific records.**

Ans: SQL Queries

Inserting a Record

```
INSERT INTO customers (id, name, email)
VALUES (1, ‘Miku Kumar’, 'mikunawada1208@gmail.com');
```

Updating Specific Fields

```
UPDATE customers
SET name = 'Miku, email = ‘mikunawada1208@gmail.com'
WHERE id = 1;
```

Selecting Records

- Select all records

```
SELECT * FROM customers;
```

- Select records based on a condition

```
SELECT * FROM customers
WHERE country = 'INDIA';
```

- Select specific columns

```
SELECT name, email FROM customers;
```

Deleting Specific Records

```
DELETE FROM customers
WHERE id = 1;
```

Q23). Implement these queries in Java using JDBC.

Ans: Java JDBC Implementation

Inserting a Record

```
String query = "INSERT INTO customers (id, name, email) VALUES (?, ?, ?)";
PreparedStatement pstmt = conn.prepareStatement(query);
pstmt.setInt(1, 1);
pstmt.setString(2, "Miku Kumar");
pstmt.setString(3, "mikunawada1208@gmail.com");
pstmt.executeUpdate();
```
```
```

Updating Specific Fields

```
String query = "UPDATE customers SET name = ?, email = ? WHERE id = ?";
PreparedStatement pstmt = conn.prepareStatement(query);
pstmt.setString(1, "Miku Kumar");
pstmt.setString(2, "mikunawada1208@gmail.com");
pstmt.setInt(3, 1);
pstmt.executeUpdate();
```

Selecting Records

```
String query = "SELECT * FROM customers WHERE country = ?";
PreparedStatement pstmt = conn.prepareStatement(query);
pstmt.setString(1, "INDIA");
ResultSet rs = pstmt.executeQuery();

while (rs.next()) {
   int id = rs.getInt("id");
   String name = rs.getString("name");
   String email = rs.getString("email");
   System.out.println("ID: " + id + ", Name: " + name + ", Email: " + email);
}
```

Deleting Specific Records

```
String query = "DELETE FROM customers WHERE id = ?";
PreparedStatement pstmt = conn.prepareStatement(query);
pstmt.setInt(1, 1);
pstmt.executeUpdate();
```

Q24). Introduction to Java Swing for GUI development

Ans: Introduction to Java Swing

Java Swing is a GUI toolkit for Java that provides a wide range of components and tools for building desktop applications. It offers a robust and flexible framework for creating user interfaces.

Key Features

1. Components: Buttons, labels, text fields, tables, trees, and more.
2. Layout Managers: Arrange components in various layouts.
3. Event Handling: Respond to user interactions.

4. Customization: Customize component appearance and behavior.

Benefits
1. Cross-Platform: Run on multiple platforms with minimal modifications.
2. Extensive Libraries: Leverage a vast collection of pre-built components.
3. Customizable: Tailor components to meet specific needs.

Common Components
1. `JFrame`: Top-level container for GUI applications.
2. `JPanel`: Container for grouping components.
3. `JButton`: Button component.
4. `JLabel`: Text or icon component.
5. `JTextField`: Text input component.

Getting Started
1. Import Swing packages.
2. Create a `JFrame` or other top-level container.
3. Add components and layout managers.
4. Handle events and customize components.

Example

```
import javax.swing.*;
import java.awt.*;

public class HelloWorld {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Hello, World!");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JLabel label = new JLabel("Hello, World!");
        frame.getContentPane().add(label);
        frame.pack();
        frame.setVisible(true);
    }
}
```

Q25). How to integrate Swing components with JDBC for CRUD operations
Ans: Integrating Swing with JDBC

Steps
1. Create Swing GUI: Design a GUI with Swing components (e.g., text fields, buttons,

tables).
2. Establish JDBC Connection: Connect to the database using JDBC.
3. Perform CRUD Operations: Use JDBC to perform CRUD operations based on user interactions.

Example

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.sql.*;

public class CRUDExample {
    private JFrame frame;
    private JTextField idField;
    private JTextField nameField;
    private JButton insertButton;
    private JButton updateButton;
    private JButton deleteButton;
    private JButton selectButton;

    public CRUDExample() {
        // Create GUI components
        frame = new JFrame("CRUD Example");
        idField = new JTextField(10);
        nameField = new JTextField(10);
        insertButton = new JButton("Insert");
        updateButton = new JButton("Update");
        deleteButton = new JButton("Delete");
        selectButton = new JButton("Select");

        // Add action listeners
        insertButton.addActionListener(new InsertActionListener());
        updateButton.addActionListener(new UpdateActionListener());
        deleteButton.addActionListener(new DeleteActionListener());
        selectButton.addActionListener(new SelectActionListener());

        // Layout components
        frame.setLayout(new FlowLayout());
        frame.add(idField);
        frame.add(nameField);
        frame.add(insertButton);
        frame.add(updateButton);
        frame.add(deleteButton);
        frame.add(selectButton);

        // Set up frame
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack();
        frame.setVisible(true);
    }

    // Action listeners for CRUD operations
    private class InsertActionListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            // Insert data into database
            try (Connection conn =
DriverManager.getConnection("jdbc:mysql://localhost:3306/mydb", "username",
"password")) {
                PreparedStatement pstmt = conn.prepareStatement("INSERT INTO customers
(id, name) VALUES (?, ?)");
                pstmt.setInt(1, Integer.parseInt(idField.getText()));
                pstmt.setString(2, nameField.getText());
                pstmt.executeUpdate();
            } catch (SQLException ex) {
                JOptionPane.showMessageDialog(frame, "Error inserting data: " +
ex.getMessage());
            }
        }
    }

    private class UpdateActionListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            // Update data in database
            try (Connection conn =
DriverManager.getConnection("jdbc:mysql://localhost:3306/mydb", "username",
"password")) {
                PreparedStatement pstmt = conn.prepareStatement("UPDATE customers SET
name = ? WHERE id = ?");
                pstmt.setString(1, nameField.getText());
                pstmt.setInt(2, Integer.parseInt(idField.getText()));
                pstmt.executeUpdate();
            } catch (SQLException ex) {
                JOptionPane.showMessageDialog(frame, "Error updating data: " +
ex.getMessage());
```

```
        }
      }
    }

    private class DeleteActionListener implements ActionListener {
      public void actionPerformed(ActionEvent e) {
        // Delete data from database
        try (Connection conn =
DriverManager.getConnection("jdbc:mysql://localhost:3306/mydb", "username",
"password")) {
          PreparedStatement pstmt = conn.prepareStatement("DELETE FROM
customers WHERE id = ?");
          pstmt.setInt(1, Integer.parseInt(idField.getText()));
          pstmt.executeUpdate();
        } catch (SQLException ex) {
          JOptionPane.showMessageDialog(frame, "Error deleting data: " +
ex.getMessage());
        }
      }
    }

    private class SelectActionListener implements ActionListener {
      public void actionPerformed(ActionEvent e) {
        // Select data from database
        try (Connection conn =
DriverManager.getConnection("jdbc:mysql://localhost:3306/mydb", "username",
"password")) {
          PreparedStatement pstmt = conn.prepareStatement("SELECT * FROM
customers WHERE id = ?");
          pstmt.setInt(1, Integer.parseInt(idField.getText()));
          ResultSet rs = pstmt.executeQuery();
          if (rs.next()) {
            nameField.setText(rs.getString("name"));
          } else {
            JOptionPane.showMessageDialog(frame, "No data found");
          }
        } catch (SQLException ex) {
          JOptionPane.showMessageDialog(frame, "Error selecting data: " +
ex.getMessage());
        }
      }
    }
    public static void main(String[] args) {
        new CRUDExample();
    }
}
```

Q26). What is a CallableStatement?

Ans: CallableStatement

A `CallableStatement` is a type of SQL statement in JDBC that allows you to execute stored procedures in a database. It provides a way to call stored procedures with input parameters and retrieve output parameters.

Benefits

1. Encapsulation: Stored procedures encapsulate complex logic and operations.
2. Reusability: Stored procedures can be reused across multiple applications.
3. Performance: Stored procedures can improve performance by reducing network traffic.

Creating a CallableStatement

```
String procedureCall = "{call procedure_name(?, ?)}";
CallableStatement  cstmt = conn.prepareCall(procedureCall);
cstmt.setString(1, "input_value");
cstmt.registerOutParameter(2, Types.VARCHAR);
cstmt.execute();
String output = cstmt.getString(2);
```

Use Cases

1. Complex Business Logic: Execute complex business logic encapsulated in stored procedures.
2. Data Validation: Use stored procedures to validate data before inserting or updating.
3. Data Transformation: Use stored procedures to transform data.

Q27). How to call stored procedures using CallableStatement in JDBC

Ans: Calling Stored Procedures using CallableStatement

Steps

1. Create a CallableStatement: Use the `prepareCall()` method to create a `CallableStatement` object.
2. Set Input Parameters: Use setter methods (e.g., `setString()`, `setInt()`) to set input

parameters.
3. Register Output Parameters: Use the `registerOutParameter()` method to register output parameters.
4. Execute the Procedure: Use the `execute()` method to execute the stored procedure.
5. Retrieve Output Parameters: Use getter methods (e.g., `getString()`, `getInt()`) to retrieve output parameters.

Example

```
// Create a CallableStatement
String procedureCall = "{call get_customer_name(?, ?)}";
CallableStatement  cstmt = conn.prepareCall(procedureCall);

// Set input parameter
cstmt.setInt(1, 1);

// Register output parameter
cstmt.registerOutParameter(2,  Types.VARCHAR);

// Execute the procedure
cstmt.execute();

// Retrieve output parameter
String customerName = cstmt.getString(2);
System.out.println("Customer Name: " + customerName);
```

Q28). Working with IN and OUT parameters in stored procedures

Ans: Working with IN and OUT Parameters

IN Parameters
1. Purpose: Pass values to a stored procedure.
2. Usage: Use setter methods (e.g., `setString()`, `setInt()`) to set IN parameter values.

OUT Parameters
1. Purpose: Return values from a stored procedure.
2. Usage: Use the `registerOutParameter()` method to register OUT parameters and getter methods (e.g., `getString()`, `getInt()`) to retrieve values.

Example Stored Procedure

```
CREATE PROCEDURE get_customer_info(
    IN customer_id INT,
    OUT customer_name VARCHAR(255),
    OUT customer_email VARCHAR(255)
)
BEGIN
    SELECT name, email INTO customer_name, customer_email
    FROM customers
    WHERE id = customer_id;
END;
```

Example Java Code

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

// Set IN parameter
cstmt.setInt(1, 1);

// Register OUT parameters
cstmt.registerOutParameter(2,  Types.VARCHAR);
cstmt.registerOutParameter(3,  Types.VARCHAR);

// Execute the procedure
cstmt.execute();

// Retrieve OUT parameters
String customerName = cstmt.getString(2);
String customerEmail = cstmt.getString(3);

System.out.println("Customer Name: " + customerName);
System.out.println("Customer Email: " + customerEmail);
```