MODULE 5

JDBC

What is JDBC?

- **Java Database Connectivity (JDBC)**: JDBC is a Java API that enables Java applications to interact with relational databases. It provides a standardized way for Java programs to execute SQL queries, retrieve results, and manage databases independently of the underlying database system.

Components of JDBC Drivers

- **JDBC Drivers**: JDBC drivers are essential components that facilitate the communication between Java applications (J2EE environments) and DBMSs. Here's what JDBC drivers typically do:
    - **Open Connection**: Establish a connection between the Java application (or J2EE environment) and the DBMS.
    - **Translate SQL Statements**: Translate SQL statements from the Java application into the specific syntax understood by the DBMS.
    - **Process Messages**: Handle communication of SQL statements and data between Java and the DBMS.
    - **Error Handling**: Capture and return error messages conforming to JDBC specifications to the Java application.
    - **Transaction Management**: Provide transaction management capabilities adhering to JDBC standards, ensuring data integrity and consistency.
    - **Close Connection**: Close the connection between the Java application and the DBMS when no longer needed, to free up resources.

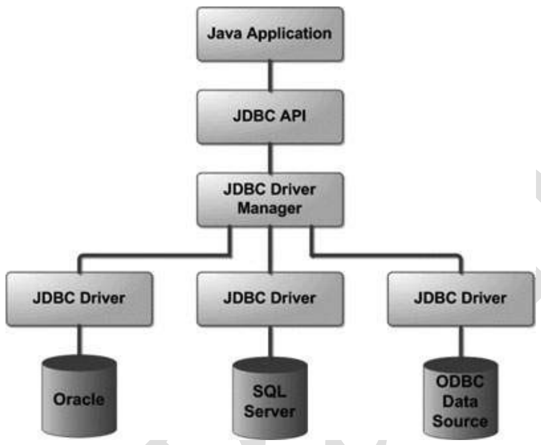
JDBC API Usage

- **Using JDBC**: To use JDBC in a Java application:
    - Load appropriate JDBC driver class (Class.forName("driver_class_name")).
    - Establish connection (DriverManager.getConnection(url, username, password)).
    - Create and execute SQL statements (Statement, PreparedStatement, CallableStatement).
    - Process result sets (ResultSet) returned by queries.
    - Handle transactions (Connection.setAutoCommit(false), Connection.commit(), Connection.rollback()).
    - Close resources (ResultSet.close(), Statement.close(), Connection.close()).

Advantages of JDBC

- **Database Independence**: JDBC allows Java applications to work with different DBMSs without modifying the application code significantly.
- **Ease of Development**: Provides a straightforward API for database operations, making it easier to develop database-driven applications in Java.
- **Integration with Java EE**: Seamlessly integrates with Java EE technologies, allowing for scalable and robust enterprise applications.

JDBC Architecture



The JDBC (Java Database Connectivity) drivers you've mentioned represent different approaches to connecting Java applications to databases

Type 1: JDBC-ODBC Bridge Driver

1. Description:

- Also known as: JDBC-ODBC Bridge.
- Developed by: Microsoft.
- Functionality: Acts as a bridge between JDBC and ODBC (Open Database Connectivity).
- Usage: Allows Java applications to access databases through ODBC drivers.
- DBMS Independence: Not truly DBMS-independent; relies on ODBC drivers installed on the system.

Type 2: Java/Native Code Driver

1. Description:

- Generates: Platform-specific code (native code) that interacts with specific databases.
- Developed by: Database vendors who provide their own Java/native code drivers.
- Portability: Provides good portability for Java applications within the same DBMS vendor's ecosystem.

- Limitation: Not compatible with other DBMS vendors' databases without additional drivers.

Type 3: Network Protocol Driver (Middleware Driver)

1. Description:

- Commonly used: Middleware driver.
- Functionality: Converts JDBC calls into a middleware-specific protocol.
- Middleware: Acts as an intermediate layer between Java application and DBMS.
- Translation: Translates JDBC calls into the native protocol of the DBMS it communicates with.
- Java Protocol: Handles communication using a protocol understood by the middleware and the DBMS.

Type 4: Thin Driver (Direct to Database Protocol Driver)

1. Description:

- Referred to as: Type 4 database protocol driver.
- Functionality: Directly converts JDBC calls into the native protocol of the DBMS.
- Efficiency: Generally the fastest communication method because it bypasses middleware layers.
- Usage: Suitable for scenarios where direct, efficient communication with the DBMS is preferred.

<u>**Steps of the JDBC Process**</u>

1. Loading the JDBC Driver

```
// Loading the JDBC driver
try {
   Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
} catch (ClassNotFoundException e) {
   e.printStackTrace();
}
```

- **Explanation**:
  - The JDBC driver must be loaded into memory using Class.forName() before establishing a connection to the database. This step ensures that the driver class is available for use.

2. Connecting to the DBMS

```
// Connecting to the DBMS
```

```
String url = "jdbc:odbc:DataSourceName";
String userId = "username";
String password = "password";
Connection db = null;

try {
   db = DriverManager.getConnection(url, userId, password);
} catch (SQLException e) {
   e.printStackTrace();
}
```

- **Explanation**:
    - DriverManager.getConnection() establishes a connection to the DBMS using the specified URL, username, and password.
    - The Connection object (db) represents the connection to the database and is used throughout the process to interact with the database.

3. Creating and Executing a Statement

```
// Creating and executing a statement
Statement dataRequest = null;
ResultSet results = null;

try {
   String query = "SELECT * FROM Customers";
   dataRequest = db.createStatement();
   results = dataRequest.executeQuery(query);
} catch (SQLException e) {
   e.printStackTrace();
}
```

- **Explanation**:
    - Connection.createStatement() creates a Statement object (dataRequest) that can be used to execute SQL queries against the database.
    - Statement.executeQuery(query) executes the SQL query (query) and returns a ResultSet (results) containing the data retrieved from the database.

4. Processing Data Returned by the DBMS

```
// Processing data returned by the DBMS
try {
   while (results.next()) {
      String fname = results.getString("Fname");
      // Process data as needed
      System.out.println("First Name: " + fname);
   }
} catch (SQLException e) {
   e.printStackTrace();
} finally {
   // Closing ResultSet and Statement
   try {
      if (results != null) results.close();
      if (dataRequest != null) dataRequest.close();
   } catch (SQLException e) {
      e.printStackTrace();
}
```

- **Explanation**:
  - ResultSet.next() moves the cursor to the next row in the ResultSet.
  - ResultSet.getString("Fname") retrieves the value of the column "Fname" from the current row of the ResultSet.
  - Use a loop (while (results.next())) to iterate through each row of the ResultSet and process the retrieved data.

5. Terminating the Connection with the DBMS

```
// Terminating the connection with the DBMS
try {
   if (db != null) {
      db.close();
   }
} catch (SQLException e) {
    e.printStackTrace();
}
```

- **Explanation**:
    - Connection.close() terminates the connection to the database (db) once the operations are completed.
    - It's important to close the connection and release resources to prevent memory leaks and maintain efficient database usage.

JDBC Connection Methods:

1. getConnection(String url)

```
// Example of getConnection(String url)
String url = "jdbc:mysql://localhost:3306/mydatabase";
try {
    // Load the JDBC driver
    Class.forName("com.mysql.jdbc.Driver");
    // Establish connection using URL
    Connection db = DriverManager.getConnection(url);
    // Use the connection...
} catch (ClassNotFoundException | SQLException e) {
    e.printStackTrace();
}
```

- **Explanation**:
    - **Purpose**: This method is used when the database grants access to anyone without requiring a username or password.
    - **URL Format**: The url parameter specifies the JDBC URL for connecting to the database, which includes the protocol (jdbc), JDBC driver name (mysql), and database name (mydatabase).
    - **Usage**: Suitable for scenarios where database access does not require authentication.

2. getConnection(String url, String user, String password)

```
// Example of getConnection(String url, String user, String password)
String url = "jdbc:mysql://localhost:3306/mydatabase";
String user = "username";
String password = "password";
try {
    // Load the JDBC driver
    Class.forName("com.mysql.jdbc.Driver");
    // Establish connection using URL, username, and password
    Connection db = DriverManager.getConnection(url, user, password);
    // Use the connection...
} catch (ClassNotFoundException | SQLException e) {
    e.printStackTrace();
}
```

- **Explanation**:
    - **Purpose**: Used when the database requires authentication with a username and password.
    - **URL Format**: The url parameter specifies the JDBC URL similar to the previous example.
    - **Username and Password**: The user and password parameters provide the credentials required to authenticate and access the database.
    - **Usage**: Typically used in production environments where database access is restricted to authorized users.

3. getConnection(String url, Properties props)

```
// Example of getConnection(String url, Properties props)
String url = "jdbc:mysql://localhost:3306/mydatabase";
Properties props = new Properties();
props.setProperty("user", "username");
props.setProperty("password", "password");

try {
    // Load the JDBC driver
    Class.forName("com.mysql.jdbc.Driver");
    // Establish connection using URL and properties
    Connection db = DriverManager.getConnection(url, props);
    // Use the connection...
} catch (ClassNotFoundException | SQLException e) {
    e.printStackTrace();
}
```

- **Explanation**:
  - **Purpose**: Used when the database requires additional properties besides username and password for authentication.
  - **Properties Object**: The props parameter is a Properties object that contains key-value pairs of additional properties required by the database.
  - **Loading Properties**: Properties can be loaded from a configuration file (like text.txt in this example) using Properties.load() method.
  - **Usage**: Useful for configuring specific database settings or options required for connection establishment, such as SSL settings, connection pooling parameters, etc.

<ins>**Timeout**</ins>

1. Competition for Database Usage:

- In J2EE environments, multiple components may compete for database resources, potentially causing performance degradation.

2. Delayed Database Availability:

- Connections to databases may experience delays or timeouts due to database unavailability or heavy load.

3. Setting Login Timeout:

- DriverManager.setLoginTimeout(int sec) allows setting a timeout period in seconds for establishing a database connection. After this period, if the connection cannot be established, a SQLException is thrown.

4. Getting Login Timeout:

- DriverManager.getLoginTimeout() retrieves the current login timeout setting in seconds.

<ins>**Connection Pool**</ins>

1. Need for Connection Pooling:

- Applications requiring frequent database interactions face performance issues if they repeatedly open/close connections. Leaving connections open ties up resources and limits database access for other clients.

2. Problems with Open/Close Approach:

- Open connections can limit database access for others, while reconnecting frequently is time-consuming.

3. Introduction of Connection Pooling:

- JDBC 2.1 introduced connection pooling to manage database connections efficiently.

4. Concept of Connection Pool:

- Connection pooling involves pre-creating a set of database connections that are stored in memory. These connections can be reused, eliminating the need for repeated connection establishment.

5. DataSource Interface Usage:

- DataSource interface is used to access the connection pool, typically implemented in application servers.

6. Types of Connections:

- **Logical:** Connections managed by the application server.
- **Physical:** Actual database connections.

7. Connecting to a Connection Pool:

```
// Example code to connect to a connection pool

Context context = new InitialContext();

DataSource pool = (DataSource) context.lookup("java:comp/env/jdbc/pool");

Connection db = pool.getConnection();
```

- **Explanation of Code**:
    - InitialContext() initializes the context for looking up resources.
    - context.lookup("java:comp/env/jdbc/pool") retrieves the DataSource named "pool" defined in the application server's environment.
    - pool.getConnection() retrieves a connection from the connection pool managed by the application server.

<u>**Statement Object in JDBC**</u>

The Statement object in JDBC is used to execute SQL queries against a database. Here's a brief explanation and examples of its usage:

1. **Immediate Execution**:
    - The Statement object executes SQL queries immediately without precompiling them.
2. **Executing Queries with executeQuery()**:
    - The executeQuery() method of Statement is used to execute SELECT queries that retrieve data from the database. It returns a ResultSet object containing the query results.

```
// Example of executing a SELECT query
```

```
String url = "jdbc:odbc:JdbcOdbcDriver";
String userId = "jim";
String password = "Keogh";
Statement statement;
Connection db;
ResultSet resultSet;

try {
   // Load JDBC driver and connect to the database
   Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
   db = DriverManager.getConnection(url, userId, password);

   // Execute SELECT query
   String query = "SELECT * FROM Customers";
   statement = db.createStatement();
   resultSet = statement.executeQuery(query);

   // Process the ResultSet
   while (resultSet.next()) {
      // Retrieve data from the result set
      String customerName = resultSet.getString("CustomerName");
      int age = resultSet.getInt("Age");
      // Process retrieved data
   }
} catch (SQLException | ClassNotFoundException e) {
   e.printStackTrace();
} finally {
   // Close resources
   try {
      if (resultSet != null) resultSet.close();
```

```
        if (statement != null) statement.close();
        if (db != null) db.close();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

3. **Executing DML and DDL Operations with executeUpdate()**:
   - The executeUpdate() method of Statement is used for executing Data Manipulation Language (DML) or Data Definition Language (DDL) operations like INSERT, UPDATE, DELETE, CREATE, etc. It returns an integer indicating the number of rows affected.

```
// Example of executing an UPDATE query
try {
    String updateQuery = "UPDATE Customer SET PAID = 'Y' WHERE BALANCE = '0'";
    statement = db.createStatement();
    int rowsAffected = statement.executeUpdate(updateQuery);
    System.out.println("Rows updated: " + rowsAffected);
} catch (SQLException e) {
    e.printStackTrace();
} finally {
    // Close resources
    try {
        if (statement != null) statement.close();
        if (db != null) db.close();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

<u>Calling a Stored Procedure</u>

To call a stored procedure in JDBC, you use CallableStatement, which is a subclass of PreparedStatement. Here's an example:

```
// Example of calling a stored procedure
```

```
try {
    CallableStatement callableStatement = db.prepareCall("{call my_stored_procedure(?, 
?)}");
    callableStatement.setString(1, "parameter1_value");
    callableStatement.setInt(2, 123);

    // Execute the stored procedure
    callableStatement.execute();

    // Optionally retrieve output parameters or result sets
    // Example: ResultSet resultSet = callableStatement.getResultSet();

} catch (SQLException e) {
    e.printStackTrace();
} finally {
    // Close resources
    try {
        if (callableStatement != null) callableStatement.close();
        if (db != null) db.close();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

- **Explanation**:
    - **CallableStatement**: Used to call stored procedures. Parameters are set using setXxx() methods.
    - **Executing**: Use execute() to execute the stored procedure.
    - **Handling Results**: Retrieve output parameters or result sets if applicable.

These examples demonstrate the usage of Statement for executing queries and CallableStatement for calling stored procedures in JDBC.

PreparedStatement Object in JDBC

The PreparedStatement object in JDBC allows efficient execution of parameterized SQL queries. Here's a brief explanation and an example program:

1. **Compilation and Execution**:
   - SQL queries are compiled by the database management system (DBMS) before execution using PreparedStatement.
2. **Parameterized Queries**:
   - Placeholders (usually question marks ?) are used in the query string to represent parameters whose values will be supplied later.
3. **Setting Parameters**:
   - setXxx() methods (e.g., setString(), setInt(), etc.) are used to set values for placeholders in the prepared statement.
4. **Late Binding**:
   - Values provided to setXxx() methods replace placeholders at runtime, allowing for efficient query execution and preventing SQL injection.

Example Program using PreparedStatement

```
import java.sql.*;


public class PreparedStatementExample {


  public static void main(String[] args) {
    String url = "jdbc:odbc:JdbcOdbcDriver";
    String userId = "jim";
    String password = "Keogh";
    Connection db = null;
    PreparedStatement pstatement = null;
    ResultSet rs = null;


    try {
      // Load JDBC driver and connect to the database
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      db = DriverManager.getConnection(url, userId, password);
```

```
            // Example of executing a SELECT query with PreparedStatement
            String query = "SELECT * FROM Customers WHERE cno = ?";
            pstatement = db.prepareStatement(query);
            pstatement.setString(1, "123"); // Set value for the first placeholder

            // Execute the query
            rs = pstatement.executeQuery();

            // Process the ResultSet
            while (rs.next()) {
                String customerName = rs.getString("CustomerName");
                int age = rs.getInt("Age");
                // Process retrieved data
                System.out.println("Customer Name: " + customerName + ", Age: " + age);
            }

        } catch (SQLException | ClassNotFoundException e) {
            e.printStackTrace();
        } finally {
            // Close resources
            try {
                if (rs != null) rs.close();
                if (pstatement != null) pstatement.close();
                if (db != null) db.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
```

```
}
```

- **Explanation of the Program**:
  - The program demonstrates how to use PreparedStatement to execute a parameterized SELECT query.
  - **Step 1**: Load the JDBC driver and establish a connection to the database.
  - **Step 2**: Define a SQL query with a placeholder (?) for the customer number.
  - **Step 3**: Create a PreparedStatement object using the query string.
  - **Step 4**: Set the value for the placeholder using setString() method.
  - **Step 5**: Execute the query with executeQuery() and retrieve results into a ResultSet.
  - **Step 6**: Process the ResultSet to fetch and display data retrieved from the database.

<u>**CallableStatement Object in JDBC**</u>

The CallableStatement object in JDBC is used to call stored procedures from within a Java application. Here's a brief explanation and an example program:

1. **Calling Stored Procedures**:
   - A stored procedure is a named block of code stored in the database, typically written in languages like Transact-SQL (T-SQL) for SQL Server or PL/SQL for Oracle.
2. **Parameters in CallableStatement**:
   - CallableStatement supports three types of parameters: IN, OUT, and INOUT.
   - IN parameters are used to pass values into the stored procedure using setXxx() methods.
   - OUT parameters are used to retrieve values returned by the stored procedure. They must be registered using registerOutParameter() before fetching using getXxx() methods.
   - INOUT parameters serve dual purpose: passing data into and retrieving data from the stored procedure.
3. **Example Program to Call a Stored Procedure**:

Assume you have a stored procedure getEmpName in Oracle that retrieves the first name of an employee based on their ID:

```
import java.sql.*;


public class CallableStatementExample {
    public static void main(String[] args) {
        String url = "jdbc:oracle:thin:@localhost:1521:xe";
        String user = "your_username";
        String password = "your_password";
        Connection conn = null;
        CallableStatement cstmt = null;
        ResultSet rs = null;

        try {
            // Establish connection
            conn = DriverManager.getConnection(url, user, password);

            // Prepare the callable statement to call the stored procedure
            String sql = "{call getEmpName(?, ?)}";
            cstmt = conn.prepareCall(sql);

            // Set input parameter EMP_ID
            int empId = 101;
            cstmt.setInt(1, empId);

            // Register OUT parameter EMP_FIRST as VARCHAR
            cstmt.registerOutParameter(2, Types.VARCHAR);

            // Execute the stored procedure
            cstmt.execute();

            // Retrieve the output parameter EMP_FIRST
            String empFirstName = cstmt.getString(2);
            System.out.println("Employee with ID " + empId + " has first name: " +
empFirstName);
```

```
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            // Close resources
            try {
                if (rs != null) rs.close();
                if (cstmt != null) cstmt.close();
                if (conn != null) conn.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
}
```

- **Explanation of the Program**:
    - The program demonstrates how to use CallableStatement to call a stored procedure getEmpName in Oracle.
    - **Step 1**: Establish a connection to the Oracle database using DriverManager.getConnection().
    - **Step 2**: Prepare the CallableStatement with the SQL string {call getEmpName(?, ?)}.
    - **Step 3**: Set the IN parameter using setInt() for EMP_ID.
    - **Step 4**: Register the OUT parameter using registerOutParameter() for EMP_FIRST.
    - **Step 5**: Execute the stored procedure using execute().
    - **Step 6**: Retrieve the value of the OUT parameter using getString().

ResultSet

1. **Purpose of ResultSet**:
   - A ResultSet object allows Java programs to retrieve and process data returned by a SQL query executed against a database.
2. **Organizing Data**:
   - Data in a ResultSet is logically organized into a virtual table, containing rows and columns. It also includes metadata describing the structure of the result set.
3. **Virtual Cursor**:
   - ResultSet uses a virtual cursor to navigate through rows of data. Initially, the cursor is positioned before the first row. The next() method moves the cursor to the next row.
4. **Accessing Data**:
   - Use methods like getInt(), getString(), etc., to retrieve data from the current row pointed to by the cursor. The method name (getInt, getString, etc.) should correspond to the data type of the column in the database.
   - Example: int id = rs.getInt("id"); retrieves the integer value from the "id" column.
5. **Iterating through ResultSet**:
   - To process each row in the ResultSet, use a loop (while loop in this case) combined with the next() method to move to the next row until all rows are processed.

Here's a simplified version of the example provided:

```
import java.sql.*;
public class ResultSetExample {
  public static void main(String[] args) {
     String url = "jdbc:mysql://localhost:3306/mydatabase";
     String user = "username";
     String password = "password";

     try {
        Connection conn = DriverManager.getConnection(url, user, password);
        Statement stmt = conn.createStatement();
```

```
            String sql = "SELECT id, first, last, age FROM Employees";
            ResultSet rs = stmt.executeQuery(sql);

            // Iterate through the result set
            while(rs.next()) {
                int id = rs.getInt("id");
                int age = rs.getInt("age");
                String first = rs.getString("first");
                String last = rs.getString("last");

                // Print each row's data
                System.out.print("ID: " + id);
                System.out.print(", Age: " + age);
                System.out.print(", First: " + first);
                System.out.println(", Last: " + last);
            }

            // Close resources
            rs.close();
            stmt.close();
            conn.close();

        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

- **Explanation**:
    - Establishes a connection to the database (getConnection()).
    - Creates a statement (createStatement()).
    - Executes a SQL query (executeQuery(sql)).
    - Iterates through the ResultSet using while(rs.next()).
    - Retrieves values using getInt(), getString().
    - Closes the ResultSet, Statement, and Connection after processing.

<u>**Scrollable ResultSet**</u>

A **Scrollable ResultSet** in JDBC allows for navigating both forward and backward through the result set, and it also supports positioning the cursor at specific rows.

Explanation:

1. **Enhanced Cursor Movement**:
    - Before JDBC 2.1, result sets were only forward-moving. With JDBC 2.1 and later versions, result sets can be scrollable, meaning you can move the cursor to specific rows or move backward through the result set.
2. **Methods for Cursor Positioning**:
    - JDBC provides several methods to position the cursor at specific rows:
        - first(): Positions the cursor at the first row.
        - last(): Positions the cursor at the last row.
        - absolute(int row): Moves the cursor to the specified row number.
        - relative(int rows): Moves the cursor relative to the current position by the specified number of rows (positive or negative).
        - previous(): Moves the cursor to the previous row.
        - getRow(): Returns the current row number.
3. **Scrollable Result Set Types**:
    - When creating a statement, you can specify the type of result set:
        - TYPE_FORWARD_ONLY (default): Moves only forward.
        - TYPE_SCROLL_INSENSITIVE: Allows scrolling in both directions, insensitive to changes made by others.
        - TYPE_SCROLL_SENSITIVE: Allows scrolling in both directions, sensitive to changes made by others.
4. **Example Code**:

- Below is an example demonstrating how to create and use a scrollable result set in JDBC:

```
import java.sql.*;
public class ScrollableResultSetExample {
   public static void main(String[] args) {
      String url = "jdbc:mysql://localhost:3306/mydatabase";
      String user = "username";
      String password = "password";
      try {
         Connection conn = DriverManager.getConnection(url, user, password);
         // Create a scrollable statement with a scroll-insensitive result set
         Statement stmt = conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_READ_ONLY);
         String sql = "SELECT * FROM emp";
         ResultSet rs = stmt.executeQuery(sql);
         // Move to the last row
         rs.last();
         // Print data from the last row
         System.out.println("Last Row:");
         System.out.println("ID: " + rs.getInt("id"));
         System.out.println("Name: " + rs.getString("name"));
         System.out.println("Salary: " + rs.getDouble("salary"));
         // Move to the first row
         rs.first();
         // Print data from the first row
         System.out.println("\nFirst Row:");
         System.out.println("ID: " + rs.getInt("id"));
         System.out.println("Name: " + rs.getString("name"));
         System.out.println("Salary: " + rs.getDouble("salary"));


         // Move to a specific absolute row (2nd row)
```

```
            rs.absolute(2);

            // Print data from the 2nd row
            System.out.println("\nRow 2:");
            System.out.println("ID: " + rs.getInt("id"));
            System.out.println("Name: " + rs.getString("name"));
            System.out.println("Salary: " + rs.getDouble("salary"));

            // Close resources
            rs.close();
            stmt.close();
            conn.close();

        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Explanation of Example:

- **Connection Establishment**: Establishes a connection to the database using getConnection().
- **Statement Creation**: Creates a scrollable statement (TYPE_SCROLL_INSENSITIVE) using createStatement().
- **Query Execution**: Executes a SQL query to retrieve data from the "emp" table.
- **Cursor Movement**: Demonstrates moving to the last row (last()), first row (first()), and an absolute row (absolute(2)).
- **Data Retrieval**: Retrieves and prints data from different rows using getInt(), getString(), and getDouble() methods of ResultSet.
- **Resource Management**: Closes ResultSet, Statement, and Connection in a finally block to ensure proper resource cleanup.

updatable ResultSet

An **updatable ResultSet** in JDBC allows modifications to the rows retrieved from a database table directly through the ResultSet. Here's an explanation with an example:

Explanation:

1. **ResultSet Updatability**:
   - By specifying ResultSet.CONCUR_UPDATABLE when creating a statement, the ResultSet allows for updates, deletions, and insertions directly to the underlying data source. This means changes made through the ResultSet can affect the database once appropriate methods (updateRow(), deleteRow(), insertRow()) are called.
2. **Updating Rows in ResultSet**:
   - After retrieving a ResultSet using executeQuery(), you can update columns in the current row using updateXXX() methods (updateString(), updateInt(), etc.). Once all updates are applied, call updateRow() to commit these changes back to the database.
3. **Example - Updating a Row**:

```
try {
  String query = "SELECT Fname, Lname FROM Customers WHERE Fname = 'Mary'
AND Lname = 'Smith'";
  Statement stmt = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);
  ResultSet rs = stmt.executeQuery(query);


  // Move to the first row (assuming it exists)
  if (rs.next()) {
    rs.updateString("Lname", "Johnson"); // Update the last name
    rs.updateRow(); // Commit the update to the database
    System.out.println("Row updated successfully.");
  } else {
    System.out.println("No matching row found.");
  }


  rs.close();
  stmt.close();
} catch (SQLException e) {
    e.printStackTrace();
}
```

4. **Deleting Rows in ResultSet**:
    - Use the deleteRow() method to delete the current row in the ResultSet. This method removes the row from the database as well, effectively deleting it.
5. **Example - Deleting a Row**:

```
try {
    String query = "SELECT Fname, Lname FROM Customers WHERE Fname = 'Mary'
AND Lname = 'Smith'";
    Statement stmt = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);
    ResultSet rs = stmt.executeQuery(query);

    // Move to the first row (assuming it exists)
    if (rs.next()) {
        rs.deleteRow(); // Delete the current row
        System.out.println("Row deleted successfully.");
    } else {
        System.out.println("No matching row found.");
    }

    rs.close();
    stmt.close();
} catch (SQLException e) {
    e.printStackTrace();
}
```

6. **Inserting Rows into ResultSet**:
    - Use insertRow() after setting values using updateXXX() methods to add a new row to the ResultSet. This inserts a new row into the database as well.
7. **Example - Inserting a Row**:

```
try {
    String query = "SELECT Fname, Lname FROM Customers WHERE Fname = 'Mary'
AND Lname = 'Smith'";
    Statement stmt = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);
    ResultSet rs = stmt.executeQuery(query);


    // Move to the insert row (if applicable)
    rs.moveToInsertRow();
    rs.updateString(1, "John"); // Set first name
    rs.updateString(2, "Smith"); // Set last name
    rs.insertRow(); // Insert the new row
    System.out.println("Row inserted successfully.");


    rs.close();
    stmt.close();
} catch (SQLException e) {
    e.printStackTrace();
}
```

8. **Impact of Changes**:
   - Changes made to the ResultSet affect only the ResultSet itself until committed to the database using appropriate DML statements (UPDATE, INSERT, DELETE).

Transaction:

A transaction in JDBC consists of a series of SQL statements that must all succeed for the transaction to be committed. If any statement fails, the entire transaction should be rolled back to maintain data consistency.

Committing a Transaction:

To commit a transaction and make all changes permanent in the database, the commit() method of the Connection object is called. Until commit() is called, changes made by SQL statements are not finalized in the database.

Example:

```
try {
    // Assuming auto-commit is disabled
    String query = "UPDATE Customer SET Street = '5 Main Street' WHERE FirstName =
'Bob'";
    Statement stmt = conn.createStatement();
    stmt.executeUpdate(query);
    conn.commit(); // Commit the transaction
} catch (SQLException e) {
    conn.rollback(); // Rollback the transaction if there's an error
    e.printStackTrace();
}
```

AutoCommit Feature:

By default, JDBC enables the auto-commit mode where each SQL statement is automatically committed as soon as it is executed. To manage transactions manually, auto-commit should be disabled by calling setAutoCommit(false) on the Connection object.

Example:

```
try {
    conn.setAutoCommit(false); // Disable auto-commit
    // Perform multiple SQL operations
    // ...
    conn.commit(); // Commit all changes at once
} catch (SQLException e) {
    conn.rollback(); // Rollback if there's an error
    e.printStackTrace();
} finally {
    conn.setAutoCommit(true); // Enable auto-commit after transaction
}
```

Rolling Back a Transaction:

If an error occurs during a transaction or if the transaction needs to be canceled, the rollback() method of the Connection object is used to revert all changes made since the transaction began.

Example:

```
try {
    conn.setAutoCommit(false); // Disable auto-commit
    // Perform SQL operations
    // ...
    conn.rollback(); // Rollback changes
} catch (SQLException e) {
    e.printStackTrace();
} finally {
    conn.setAutoCommit(true); // Enable auto-commit after rollback
}
```

Savepoints:

Savepoints allow you to set points within a transaction where you can roll back to a specific state without rolling back the entire transaction. This is useful when you want to selectively undo parts of a transaction.

Example:

```
try {
    conn.setAutoCommit(false); // Disable auto-commit
    // Perform SQL operations
    Savepoint sp1 = conn.setSavepoint("sp1"); // Create a savepoint
    // More SQL operations
    // ...
    conn.rollback(sp1); // Rollback to the savepoint
    // Continue with more SQL operations
    conn.commit(); // Commit changes
} catch (SQLException e) {
    e.printStackTrace();
} finally {
    conn.setAutoCommit(true); // Enable auto-commit after transaction
}
```

Release Savepoints:

Once a savepoint is no longer needed, it can be released using releaseSavepoint() on the Connection object.

Example:

```
try {
    Savepoint sp1 = conn.setSavepoint("sp1"); // Create a savepoint
    // Perform SQL operations
    // ...
    conn.releaseSavepoint(sp1); // Release the savepoint
    conn.commit(); // Commit changes
} catch (SQLException e) {
    e.printStackTrace();
}
```

By managing transactions effectively using these methods, JDBC applications ensure data integrity and consistency, even in the event of errors or interruptions during database operations.

Batch execution in JDBC

Batch Execution Overview:

- Batch execution allows you to group multiple SQL statements into a batch using the addBatch() method of the Statement object. Once added to the batch, all statements are executed together using the executeBatch() method.

addBatch() Method:

- The addBatch() method of the Statement object adds a SQL statement to the batch. It takes a SQL string as its parameter and queues it for execution.

executeBatch() Method:

- The executeBatch() method of the Statement object executes all the SQL statements that have been added to the batch using addBatch(). It returns an array of integers that represents the update counts for each statement in the batch.

Example of Batch Execution:

```
String query1 = "UPDATE Customers SET street = '5th Main' WHERE Fname = 'Bob'";
String query2 = "UPDATE Customers SET street = '10th Main' WHERE Fname = 'Tom'";


try {
    Statement stmt = conn.createStatement();
    stmt.addBatch(query1);
    stmt.addBatch(query2);

    int[] updateCounts = stmt.executeBatch();

    // Process update counts (if needed)
    for (int count : updateCounts) {
        System.out.println("Rows updated: " + count);
    }

    conn.commit(); // Commit the transaction after batch execution
} catch (SQLException e) {
    conn.rollback(); // Rollback if there's an error
    e.printStackTrace();
}
```

Explanation of the Example:

- In the example above, two UPDATE statements (query1 and query2) are added to the batch using addBatch().
- executeBatch() is then called to execute both statements in one go. It returns an array updateCounts containing the number of rows affected by each statement.
- After executing the batch, you can iterate over updateCounts to process the result of each statement's execution.
- Finally, the transaction is committed (conn.commit()) if all statements were executed successfully. If any statement fails, the transaction is rolled back (conn.rollback()).

Benefits of Batch Execution:

- Reduces network round-trips: Batch execution reduces the number of times your application needs to communicate with the database server, improving performance.
- Atomicity: All statements in a batch are executed as a single unit of work, ensuring atomicity. If one statement fails, the entire batch can be rolled back.
- Efficiency: Especially useful when performing multiple updates or inserts, as it minimizes overhead associated with statement execution.

Metadata Interface in JDBC

1. **DatabaseMetaData Interface**:
   - DatabaseMetaData is an interface in JDBC used to retrieve metadata information about a database.
   - Metadata provides information about the database structure, such as its tables, columns, primary keys, etc.
2. **Accessing Metadata**:
   - Metadata in JDBC is accessed through the getMetaData() method of the Connection object.

Database Metadata Methods

3. **Methods to Retrieve Database Metadata**:
   - getDatabaseProductName(): Returns the name of the database product.
   - getUserName(): Returns the username used to establish the connection.
   - getURL(): Returns the URL of the database.
   - getSchemas(): Returns the names of all schemas in the database.
   - getPrimaryKeys(): Returns information about primary keys in a specified table.
   - getTables(): Returns information about tables in the database.

ResultSet Metadata

4. **ResultSetMetaData**:
   - ResultSetMetaData provides metadata about the columns in a ResultSet.
   - Retrieved using ResultSet.getMetaData().
5. **Methods to Retrieve ResultSet Metadata**:
   - getColumnCount(): Returns the number of columns in the ResultSet.
   - For each column:
     - getColumnName(int): Returns the name of the column.
     - getColumnType(int): Returns the SQL type of the column.
     - getColumnTypeName(int): Returns the database-specific type name of the column.
     - isNullable(int): Indicates whether the column allows NULL values.

SQL Data Types and Corresponding Java Types

6. **Mapping SQL Data Types to Java Types**:

- These mappings are crucial for setting and getting values using PreparedStatement and ResultSet.

SQL Data Type	**Java Type**
VARCHAR	java.lang.String
CHAR	java.lang.String
LONGVARCHAR	java.lang.String
BIT	boolean
NUMERIC	java.math.BigDecimal
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
FLOAT	float
DOUBLE	double
VARBINARY	byte[]
BINARY	byte[]
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp
CLOB	java.sql.Clob
BLOB	java.sql.Blob
ARRAY	java.sql.Array
REF	java.sql.Ref

Exception Handling in JDBC

1. **Basic Exception Handling**:
    - Exceptions in JDBC, like in Java, are handled using try, catch, and optionally finally blocks.
    - When an exception occurs, control is transferred to the nearest catch block that matches the type of exception thrown.
2. **Types of JDBC Exceptions**:
    - **SQLException**: This is the most common exception encountered in JDBC.
        - It typically occurs due to SQL syntax errors, database connection issues, or other database-related errors.
        - Methods like getNextException() and getErrorCode() in SQLException help in retrieving detailed error information and vendor-specific error codes respectively.
3. **SQLWarnings**:
    - SQLWarning instances are warnings issued by the database or JDBC driver.
    - These warnings are less severe than exceptions but should be checked to ensure proper handling of database operations.
    - Methods like getWarnings() and getNextWarning() help in retrieving and processing warnings associated with a database connection.
4. **DataTruncation**:
    - DataTruncation exceptions occur when data is truncated during insertion or update due to its size exceeding the database column's capacity.
    - These exceptions provide information about which data was truncated and the context in which it occurred.

Difference Between Statement and PreparedStatement

Statement:

- **Purpose:** Used for executing static SQL queries.
- **Compilation:** Executes SQL queries directly against the database each time they are executed.
- **Execution:** Suitable for executing SQL queries that are not parameterized.
- **Metadata Verification:** Verifies metadata (e.g., column types) in the database for each execution.
- **SQL Injection:** Prone to SQL injection attacks when directly embedding user input into queries.
- **Usage:** Ideal for executing simple, infrequently used SQL queries.

PreparedStatement:

- **Purpose:** Used for executing dynamic SQL queries with parameters.
- **Compilation:** Precompiled once and stored in the database for reuse.
- **Execution:** Executes parameterized SQL queries efficiently multiple times.
- **Metadata Verification:** Verifies metadata in the database only once during preparation.
- **SQL Injection:** Mitigates SQL injection risks by using parameterized queries.
- **Usage:** Ideal for executing SQL queries repeatedly with different parameter values.

Overview of java.sql.* Package Classes and Interfaces

The java.sql. * Package provides essential classes and interfaces for interacting with databases using JDBC. These components facilitate connection management, statement execution, handling of result sets, transaction management, and retrieval of database metadata.

1. **java.sql.Blob:**

   Represents Binary Large Object (BLOB) data type in SQL databases.

2. **java.sql.Connection:**

   Interface used to establish a connection with a specific database.

   Methods**:** setSavepoint(), rollback(), commit(), setAutoCommit().

3. **java.sql.CallableStatement:**

   Subinterface of PreparedStatement used to execute stored procedures.

   Methods: execute(), registerOutParameter().

4. **java.sql.Clob:**

   Represents Character Large Object (CLOB) data type in SQL databases.

5. **java.sql.Date:**

   Represents Date SQL type in Java.

6. **java.sql.Driver:**

   Interface used to create an instance of a database driver with DriverManager.

7. **java.sql.DriverManager:**

   Manages the database drivers.

   Methods: getConnection(), setLoginTimeout(), getLoginTimeout().

8. **java.sql.PreparedStatement:**

   Subinterface of Statement used to execute parameterized SQL queries.

   Methods: executeQuery(), executeUpdate().

9. **java.sql.ResultSet:**

   Interface used to access a result row by row.

   Methods: next(), last(), first().

10. **java.sql.Savepoint:**

    Represents a savepoint in a transaction.

11. **java.sql.SQLException:**

    Encapsulates JDBC-related exceptions.

12. **java.sql.Statement:**

    Interface used to execute SQL statements.

13. **java.sql.DatabaseMetaData:**

    Interface used to retrieve metadata about the database.

Methods: Various methods like getDatabaseProductName(), getUserName(), getURL().