### What is JDBC?

JDBC (Java Database Connectivity) is a Java API that enables Java applications to interact with relational databases like MySQL, PostgreSQL, Oracle, etc. It provides methods for querying and updating the database, managing connections, and handling errors.

### Key Components of JDBC

1. **DriverManager**: It manages a list of database drivers.
2. **Connection**: Establishes a connection to the database.
3. **Statement**: Used to execute queries against the database.
4. **ResultSet**: Holds the results of a query.
5. **PreparedStatement**: A better version of Statement with precompiled SQL queries to prevent SQL injection.
6. **SQLException**: Handles database-related errors.

### Step 1: Setup JDBC

Before you can start coding, you need to:

1. Install the database (e.g., MySQL, PostgreSQL).
2. Download the JDBC driver for your database.
3. Add the JDBC driver JAR to your project’s classpath.

Let’s assume we are using MySQL here. Download the MySQL JDBC driver (Connector/J) from [MySQL official site](https://dev.mysql.com/downloads/connector/j/).

### Step 2: Code Structure

A typical JDBC program has the following structure:

1. Load the database driver
2. Establish a connection
3. Create a statement or prepared statement
4. Execute a query
5. Process the result
6. Close the connection

### Example 1: Basic JDBC Example

Let's start with a simple example where we connect to a MySQL database, execute a query, and fetch the results.

#### Code:

java

Copy

import java.sql.\*;

public class JdbcExample {

public static void main(String[] args) {

// Step 1: Load the JDBC driver (optional for newer versions of Java)

try {

// For MySQL

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

// Step 2: Establish the connection

Connection conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/testdb", "root", "password");

// Step 3: Create a statement object

Statement stmt = conn.createStatement();

// Step 4: Execute a query

String query = "SELECT \* FROM employees";

ResultSet rs = stmt.executeQuery(query);

// Step 5: Process the result

while (rs.next()) {

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

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

String position = rs.getString("position");

System.out.println("ID: " + id + ", Name: " + name + ", Position: " + position);

}

// Step 6: Close resources

rs.close();

stmt.close();

conn.close();

} catch (ClassNotFoundException | SQLException e) {

e.printStackTrace();

}

}

}

#### Explanation:

1. **Load JDBC Driver**: Class.forName("com.mysql.cj.jdbc.Driver"); loads the MySQL JDBC driver class. In newer versions of Java, this might not be necessary as the driver is auto-loaded.
2. **Establish Connection**: DriverManager.getConnection() establishes a connection to the MySQL database using a connection string (jdbc:mysql://localhost:3306/testdb), username, and password.
3. **Create Statement**: conn.createStatement() creates a Statement object to execute SQL queries.
4. **Execute Query**: stmt.executeQuery(query) executes the SQL query. It returns a ResultSet object that holds the data.
5. **Process Result**: rs.next() iterates through the result set and retrieves column values using methods like getInt() and getString().
6. **Close Resources**: It's important to close ResultSet, Statement, and Connection to avoid memory leaks.

### Example 2: Using PreparedStatement (for safer queries)

A **PreparedStatement** is used when you want to execute the same SQL query multiple times with different parameters, or when you want to prevent SQL injection attacks.

#### Code:

java

Copy

import java.sql.\*;

public class JdbcPreparedStatementExample {

public static void main(String[] args) {

try {

// Load the JDBC driver

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

// Establish the connection

Connection conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/testdb", "root", "password");

// Prepare an SQL statement with placeholders

String query = "INSERT INTO employees (name, position) VALUES (?, ?)";

PreparedStatement pstmt = conn.prepareStatement(query);

// Set values for placeholders

pstmt.setString(1, "John Doe");

pstmt.setString(2, "Manager");

// Execute the query

int rowsAffected = pstmt.executeUpdate();

System.out.println("Rows inserted: " + rowsAffected);

// Close resources

pstmt.close();

conn.close();

} catch (ClassNotFoundException | SQLException e) {

e.printStackTrace();

}

}

}

#### Explanation:

1. **PreparedStatement**: conn.prepareStatement(query) creates a PreparedStatement, which is used for executing SQL queries with parameters (the ? placeholders).
2. **Set Parameters**: pstmt.setString(1, "John Doe"); sets the value for the first parameter (?) in the SQL query.
3. **Execute Update**: pstmt.executeUpdate() is used for SQL commands like INSERT, UPDATE, and DELETE. It returns the number of affected rows.

### Example 3: Update or Delete Data Using JDBC

Let's use JDBC to update data in the database.

#### Code:

java

Copy

import java.sql.\*;

public class JdbcUpdateDeleteExample {

public static void main(String[] args) {

try {

// Load the JDBC driver

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

// Establish the connection

Connection conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/testdb", "root", "password");

// Update the employee's position

String updateQuery = "UPDATE employees SET position = ? WHERE id = ?";

PreparedStatement pstmt = conn.prepareStatement(updateQuery);

pstmt.setString(1, "Senior Manager");

pstmt.setInt(2, 1); // Update position for employee with ID 1

int rowsAffected = pstmt.executeUpdate();

System.out.println("Rows updated: " + rowsAffected);

// Delete an employee

String deleteQuery = "DELETE FROM employees WHERE id = ?";

PreparedStatement pstmtDelete = conn.prepareStatement(deleteQuery);

pstmtDelete.setInt(1, 2); // Delete employee with ID 2

int rowsDeleted = pstmtDelete.executeUpdate();

System.out.println("Rows deleted: " + rowsDeleted);

// Close resources

pstmt.close();

pstmtDelete.close();

conn.close();

} catch (ClassNotFoundException | SQLException e) {

e.printStackTrace();

}

}

}

#### Explanation:

1. **Update**: pstmt.setString(1, "Senior Manager"); and pstmt.setInt(2, 1); update the position of an employee with id = 1.
2. **Delete**: Similar to the update operation, DELETE is executed using the executeUpdate() method.

### Example 4: Handling Exceptions

JDBC operations can throw SQLException, so we need to handle it properly to troubleshoot database issues.

java

Copy

import java.sql.\*;

public class JdbcExceptionHandlingExample {

public static void main(String[] args) {

try {

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

Connection conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/testdb", "root", "wrongpassword");

} catch (ClassNotFoundException e) {

System.out.println("JDBC Driver not found.");

e.printStackTrace();

} catch (SQLException e) {

System.out.println("Database connection error: " + e.getMessage());

e.printStackTrace();

}

}

}

#### Explanation:

1. **SQLException**: This exception is thrown when there's an issue with database operations (e.g., wrong credentials, connection issues).
2. **ClassNotFoundException**: This exception is thrown if the JDBC driver is not found.

### Conclusion

This tutorial has covered the basics of JDBC, including:

1. Connecting to a database
2. Executing queries using Statement and PreparedStatement
3. Updating and deleting records
4. Handling exceptions