JDBC

https://www.javatpoint.com/jdbc-driver

```
import
import java.sql.*;
```

JDBC (Java Database Connectivity) is an API in Java that allows Java applications to interact with relational databases. It provides a standard set of interfaces for connecting to databases, executing SQL queries, and processing the results.

Template of statement

```
import java.sql.*;
public class Statementdemo{
 public static vide main(String [] args ){
 try{
    String url="jdbc:mysql://localhost:3306/students";
    Class.forName ="com.mysql.cj.jdbc.Driver";
    Connection con = DriverManager.getConnection(url,"root","root");
    Statement statement = con.createStatement();
    String query=" Write the required query like create ,insert ...";
    statement.executeUpdate(query);
    // int result =statement.executeUpdate(query);//Returns number of row effecte
    //System.out.println(result);
     }catch(Exception e){
        System.out.println("Error");
        }
 }
Query:

CREATE TABLE tablename (Column name column type);
"CREATE TABLE student (ID INT PRIMARY KEY, Name VARCHAR(50))";
INSERT  INTO tablename (Column name1,Column name2,..)VALUES(v1,v2...);
```

```
"INSERT INTO student (id, name, email, semester) VALUES (6, 'John', 'john@gma
INSERT  INTO tablename VALUES(v1,v2...);
"INSERT INTO student  VALUES (6, 'John', 'john@gmail.com', 3);";
UPDATE tablename SET column name= updatedvalue WHERE column name = va
"UPDATE student SET semester = 8 WHERE id = 3;";
"UPDATE student SET name = 'NewName' WHERE id = 1;";
 DELETE  FROM tablename WHERE column name = value'
 "DELETE FROM student WHERE id = 5;";
 SELECT * FROM tablename;
```

Types of JDBC Drivers:

1. **Type 1: JDBC-ODBC Bridge Driver**
   - **Description**: This driver uses an ODBC driver to connect to the database. The JDBC API calls are translated to ODBC calls.
   - **Advantage**: Easy to use, no need for a specific JDBC driver for the database.
   - **Disadvantage**: Performance is slow because it involves multiple translations. Requires ODBC installation.
2. **Type 2: Native-API Driver (Partly Java)**
   - **Description**: This driver uses the native database API (specific to the database) to connect. The JDBC calls are translated into native API calls.
   - **Advantage**: Faster than Type 1 since it uses native API directly.
   - **Disadvantage**: Requires platform-specific native libraries, making it less portable.
3. **Type 3: Network Protocol Driver (Fully Java)**
   - **Description**: This driver uses a middleware server to translate JDBC calls into the database-specific protocol. It allows connections over the network.
   - **Advantage**: Database-independent and can connect to multiple types of databases.

- **Disadvantage**: Requires a separate middleware component, adding complexity.

4. **Type 4: Thin Driver (Pure Java)**
    - **Description**: This driver is written completely in Java and communicates directly with the database using the database's own protocol.
    - **Advantage**: Platform-independent and offers high performance. No middleware or native libraries required.
    - **Disadvantage**: Database-specific driver needed for each database.

Summary:

- **Type 1**: JDBC-ODBC Bridge (Slow, requires ODBC)
- **Type 2**: Native API Driver (Faster, platform-specific)
- **Type 3**: Network Protocol Driver (Flexible, uses middleware)
- **Type 4**: Thin Driver (Fast, fully Java, direct communication)

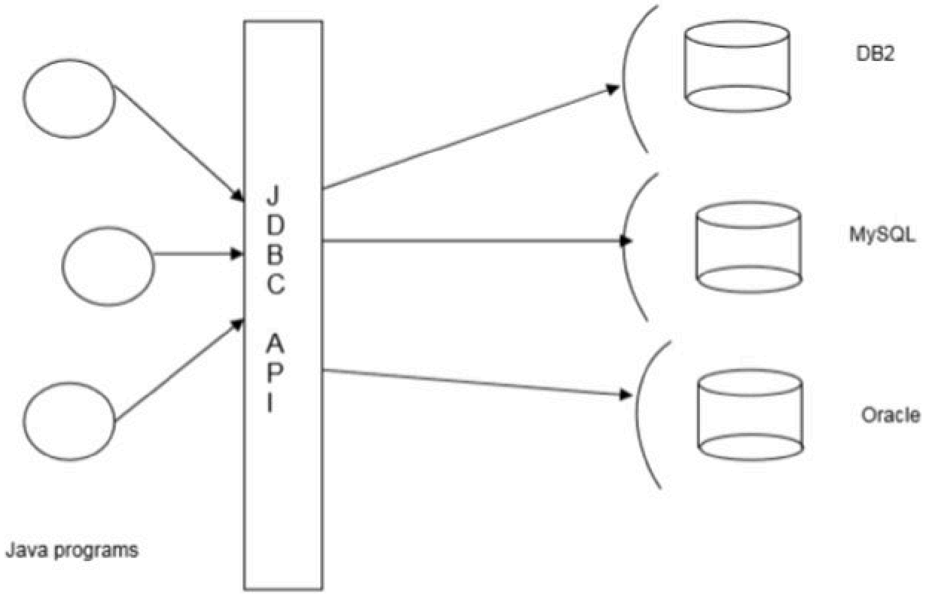
The most common and recommended driver today is **Type 4** for its efficiency, portability, and ease of use.

```
package JDBC;

import java.sql.*;
public class InsertPrepStmt {
    public static void main(String[] args) {
        try {
            String url = "jdbc:mysql://localhost:3306/student"; // Database URL
            Class.forName("com.mysql.cj.jdbc.Driver"); // Load MySQL JDBC Driver
            Connection con = DriverManager.getConnection(url, "root", "root"); // Est
            String query = "INSERT INTO student (id, name, email, semester) VALUES
            PreparedStatement preparedStatement = con.prepareStatement(query);
            preparedStatement.setInt(1, 5); // ID
            preparedStatement.setString(2, "Sita"); // Name
            preparedStatement.setString(3, "sita@gmail.com"); // Email
            preparedStatement.setInt(4, 1); // Semester
            preparedStatement.executeUpdate();

            preparedStatement.setInt(1, 6); // ID
            preparedStatement.setString(2, "Shyam"); // Name
            preparedStatement.setString(3, "shyam@gmail.com"); // Email
            preparedStatement.setInt(4, 1); // Semester
            preparedStatement.executeUpdate();

            preparedStatement.setInt(1, 7); // ID
            preparedStatement.setString(2, "Alka"); // Name
            preparedStatement.setString(3, "alka@gmail.com"); // Email
            preparedStatement.setInt(4, 5); // Semester
            preparedStatement.executeUpdate();
            System.out.println(" record inserted successfully."); // Print success mess
        } catch (Exception e) {
            e.printStackTrace(); // Print the error stack trace
        }
    }
}
```



The figure illustrates how JDBC (Java Database Connectivity) API works as an interface between Java programs and different types of databases. Here's what the diagram shows:

- On the left side are Java programs (represented by circles) that need to interact with databases
- In the middle is the JDBC API, which acts as a bridge or intermediary layer
- On the right side are different types of databases that can be connected to:
    - DB2
    - MySQL
    - Oracle

The arrows indicate that Java programs can communicate with any of these different database systems through the JDBC API, demonstrating JDBC's role as a standardized way to connect Java applications with various database management systems.

Difference Between `Statement` and `PreparedStatement` in Java JDBC:

Feature	Statement	PreparedStatement
Definition	Used to execute static SQL queries.	Used to execute precompiled SQL queries.
SQL Query	Hardcoded in the Java code, written directly.	Query is precompiled and can take parameters.
Efficiency	Slower for repeated execution; parsed every time.	Faster for repeated execution; compiled once.
Security	Prone to **SQL injection** attacks.	Safer against SQL injection due to parameterized queries.
Dynamic Queries	Cannot easily reuse the same query with different values.	Can reuse queries with different values using parameters (placeholders `?`).
Syntax	`Statement stmt = con.createStatement();`	`PreparedStatement pstmt = con.prepareStatement(query);`
Use Case	Best for simple, one-time SQL queries.	Best for queries with dynamic parameters or repeated execution.

Summary:

- **Statement**: Simple but less secure and less efficient for multiple executions.
- **PreparedStatement**: Precompiled, faster for repeated use, and safer against SQL injection.

Simple Difference Between `Statement` and `PreparedStatement`:

Feature	Statement	PreparedStatement
SQL Query	Directly written in code.	Uses placeholders (`?`) for parameters.
Speed	Slower if run many times.	Faster for repeated use.
Security	Can be attacked with **SQL injection**.	Safe from SQL injection.
Reusability	Cannot reuse with different values easily.	Can reuse by changing parameter values.

Use Case	Good for simple, one-time queries.	Better for queries that run multiple times.

Summary:

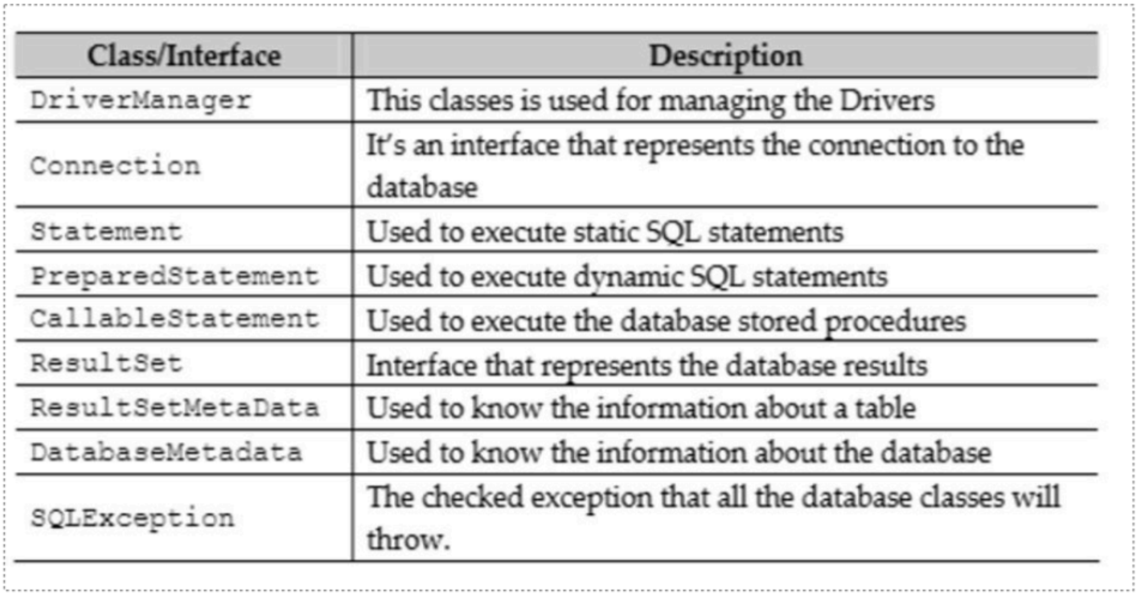
- **Statement**: Simple but less secure and slower for repeated queries.
- **PreparedStatement**: Safer, faster, and better for reusing queries with different values.

Uses of `Statement`:

- Executes simple SQL queries (e.g., `SELECT`, `INSERT`, `UPDATE`, `DELETE`).
- Used when queries do not require parameters.
- Ideal for running one-time, static SQL queries.

Uses of `PreparedStatement`:

- Executes parameterized SQL queries (with `?` placeholders).
- Suitable for dynamic queries with input parameters (e.g., `WHERE id = ?`).
- Prevents SQL injection by separating queries from data.
- Ideal for repeated executions of the same query with different parameters.



Class/Interface	Description
`DriverManager`	This classes is used for managing the Drivers
`Connection`	It's an interface that represents the connection to the database
`Statement`	Used to execute static SQL statements
`PreparedStatement`	Used to execute dynamic SQL statements
`CallableStatement`	Used to execute the database stored procedures
`ResultSet`	Interface that represents the database results
`ResultSetMetaData`	Used to know the information about a table
`DatabaseMetadata`	Used to know the information about the database
`SQLException`	The checked exception that all the database classes will throw.

- Register the driver class:
  - First step is to load or register the JDBC driver for the database. Class class provides forName method to dynamically load the driver class.
  - Syntax: Class.forName("driver ClassName");
- Making a Connection:
  - DriverManager class provides the facility to create a connection between a database and the appropriate driver.
  - To open a database connection we can call getConnection method of DriverManager class.
  - Syntax: Connection connection =DriverManager.getConnection (url,username, password) ;

- Creating Statement:
  - The statement object is used to execute the query against the database. statement object can be any one of the Statement, CallableStatement, and PreparedStatement types.
  - To create a statement object we have to call createStatement method of Connection interface.
  - Syntax: Statement stmt=connection.createStatement();
- Executing Statement:
  - The executeQuery method of Statement interface is used to execute queries to the database.
  - This method returns the object of ResultSet that can be used to get all the records of a table.
  - Syntax: ResultSet resultSet = stmt.executeQuery(selectQuery);
- Closing Connection:
  - After done with the database connection we have to close it. Use close method of Connection interface to close database connection.
  - The statement and ResultSet objects will be closed automatically when we close the connection object.
  - Syntax: connection.close());

executeQuery vs. executeUpdate

- The executeQuery method is used to execute a SELECT statement and returns a ResultSet with the number of rows selected.

- The executeUpdate() si used to execute SQL statements such as INSERT, UPDATE or DELETE and it returns the number of rows affected.

Shortened Comparison of JDBC vs ODBC:

Aspect	**JDBC**	**ODBC**
Language	Java-specific	Language-independent
Platform	Platform-independent	Platform-dependent
Security	More secure (supports `PreparedStatement`)	Less secure (manual query construction)
Usage	Easier for Java applications	Requires driver setup and configuration
Performance	Faster for Java apps	May have performance overhead
Driver	Requires JDBC drivers	Requires ODBC drivers and DSN setup
Setup	Simple setup for Java	More complex, needs additional configuration

ODBC (Open Database Connectivity) is a standard API for accessing database management systems (DBMS), allowing applications to communicate with any DBMS using ODBC drivers. It is platform-independent, but requires specific drivers and DSN configuration for each database.

```
import java.sql.*;

public class InsertRecordsPreparedStatement {
    public static void main(String[] args) {
        try {
            // Load JDBC driver
            Class.forName("com.mysql.cj.jdbc.Driver");
            // Establish connection
            Connection con = DriverManager.getConnection("jdbc:mysql://localhost:

            String query = "INSERT INTO student (ID, Name, Email, Semester) VALUE
            PreparedStatement pstmt = con.prepareStatement(query);

            // Insert 10 records using PreparedStatement
            for (int i = 1; i <= 10; i++) {
                pstmt.setInt(1, i); // ID
                pstmt.setString(2, "Student" + i); // Name
                pstmt.setString(3, "student" + i + "@example.com"); // Email
                pstmt.setInt(4, i % 8 + 1); // Semester
                pstmt.executeUpdate();
            }

            System.out.println("Records inserted successfully using PreparedStateme
            pstmt.close();S
            con.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```