

## Module 5

### Java Database Connectivity

Java Database Connectivity (JDBC) is an application programming interface (API) which allows the programmer to connect and interact with databases. It provides methods to query and update data in the database through update statements like SQL's CREATE, UPDATE, DELETE and INSERT and query statements such as SELECT. Additionally, JDBC can run stored procedures.

#### Why Should We Use JDBC

Before JDBC, ODBC API was the database API to connect and execute the query with the database. But, ODBC API uses ODBC driver which is written in C language (i.e. platform dependent and unsecured). That is why Java has defined its own API (JDBC API) that uses JDBC drivers (written in Java language).

We can use JDBC API to handle database using Java program and can perform the following activities:

1. Connect to the database
2. Execute queries and update statements to the database
3. Retrieve the result received from the database.

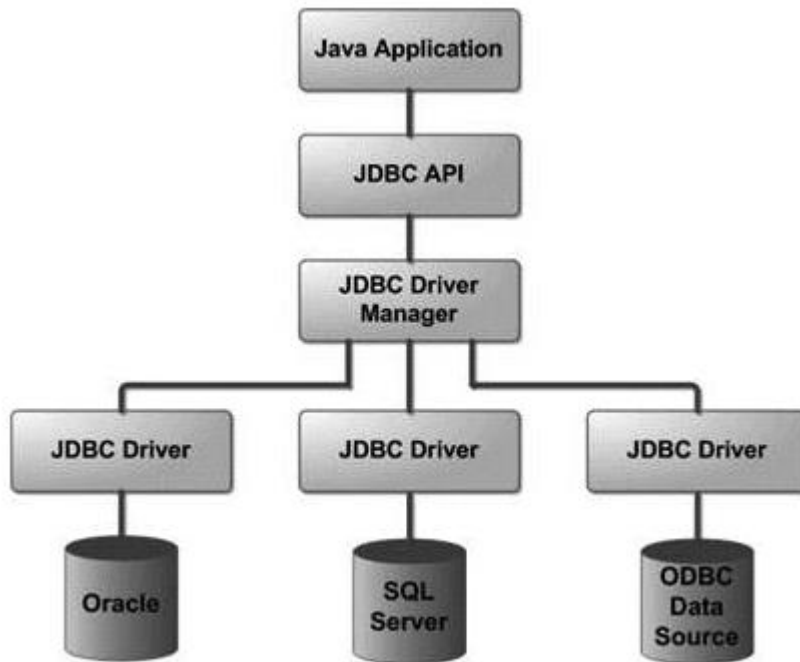
#### JDBC Architecture

The JDBC API supports both two-tier and three-tier processing models for database access but in general, JDBC Architecture consists of two layers –

- **JDBC API:** This provides the application-to-JDBC Manager connection.
- **JDBC Driver API:** This supports the JDBC Manager-to-Driver Connection.

The JDBC API uses a driver manager and database-specific drivers to provide transparent connectivity to heterogeneous databases.

The JDBC driver manager ensures that the correct driver is used to access each data source. Following is the architectural diagram, which shows the location of the driver manager with respect to the JDBC drivers and the Java application –



### Common JDBC Components

The JDBC API provides the following interfaces and classes –

- **DriverManager:** This class manages a list of database drivers. Matches connection requests from the java application with the proper database driver using communication sub protocol. The first driver that recognizes a certain subprotocol under JDBC will be used to establish a database Connection.
- **Driver:** This interface handles the communications with the database server. You will interact directly with Driver objects very rarely. Instead, you use DriverManager objects, which manages objects of this type.
- **Connection:** This interface with all methods for contacting a database. The connection object represents communication context, i.e., all communication with database is through connection object only.
- **Statement:** We use objects created from this interface to submit the SQL statements to the database. Some derived interfaces accept parameters in addition to executing stored procedures.
- **ResultSet:** These objects hold data retrieved from a database after you execute an SQL query using Statement objects. It acts as an iterator to allow you to move through its data.

- **SQLException:** This class handles any errors that occur in a database application.

### JDBC Packages

The JDBC API is contained in two packages. The first package is called `java.sql` and contains core JDBC interfaces of the JDBC API. These include the JDBC interfaces that provide the basics for connecting to the DBMS and interacting with data stored in the DBMS. `java.sql` is part of the J2SE. The other package that contains the JDBC API is `javax.sql`, which extends `java.sql`.

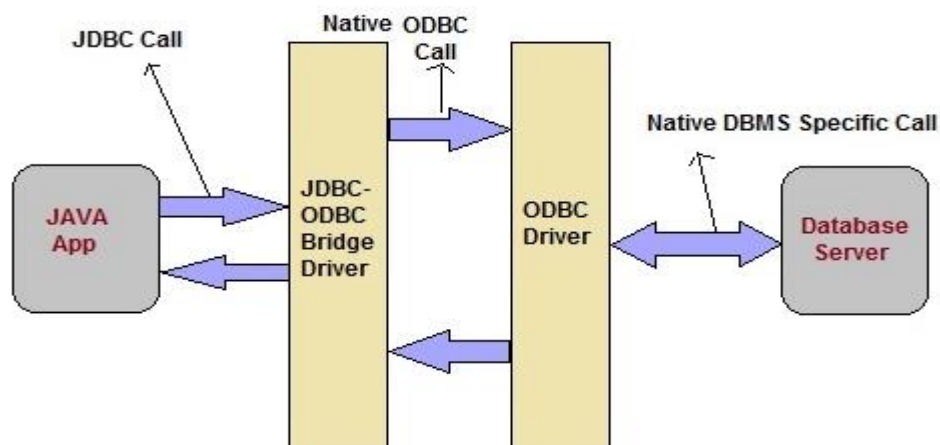
### JDBC Drivers

JDBC Driver is required to process SQL requests and generate result. The following are the different types of driver available in JDBC.

- **Type-1 Driver** or **JDBC-ODBC bridge**
- **Type-2 Driver** or **Native API Partly Java Driver**
- **Type-3 Driver** or **Network Protocol Driver**
- **Type-4 Driver** or **Thin Driver**

#### Type-1 Driver or JDBC-ODBC bridge

**Type-1 Driver** act as a bridge between JDBC and other database connectivity mechanism(ODBC). This driver converts JDBC calls into ODBC calls and redirects the request to the ODBC driver.



#### Advantages

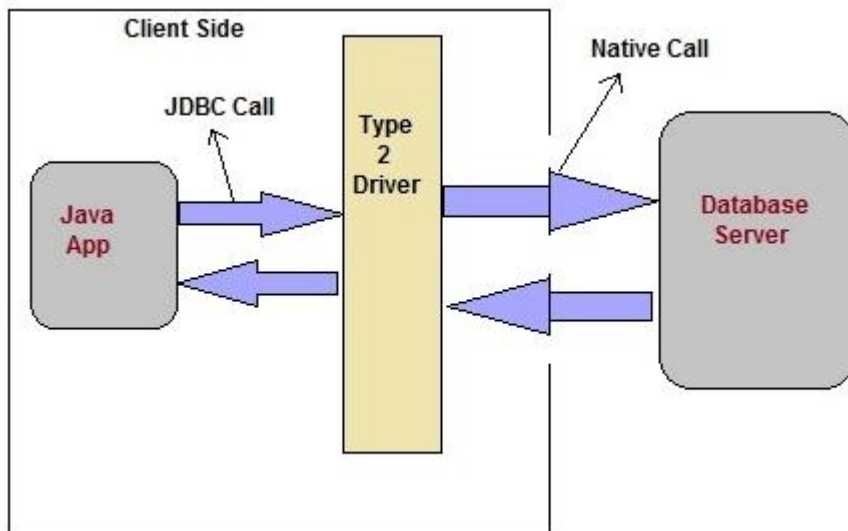
- Easy to use
- Allow easy connectivity to all database supported by the ODBC Driver.

### Disadvantages

- Since the Bridge driver is not written fully in Java, Type 1 drivers are not portable
- A performance issue is seen as a JDBC call goes through the bridge to the ODBC driver, then to the database, and this applies even in the reverse process. They are the slowest of all driver types.
- The client system requires the ODBC Installation to use the driver.
- Not good for the Web.

### Type-2 Driver or Native API Partly Java Driver

This type of driver make use of Java Native Interface(JNI) call on database specific native client API. These native client API are usually written in C and C++.



### Advantages

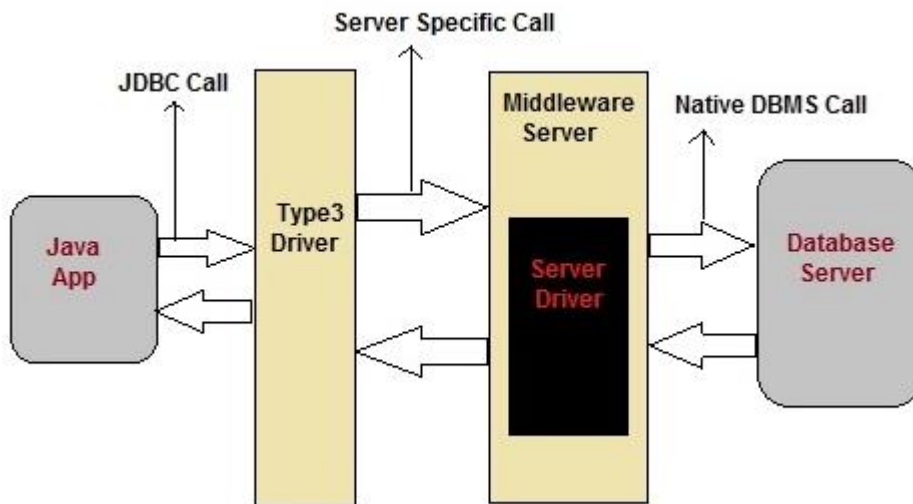
- faster as compared to **Type-1 Driver**
- Contains additional features.

### Disadvantages

- Native API must be installed in the Client System and hence type 2 drivers cannot be used for the Internet.
- Like Type 1 drivers, it's not written in Java Language which forms a portability issue.
- If we change the Database we have to change the native api as it is specific to a database

### Type-3 Driver or Network Protocol Driver

This driver translates the JDBC calls into a database server independent and Middleware server-specific calls. Middleware server further translates JDBC calls into database specific calls.



#### Advantages

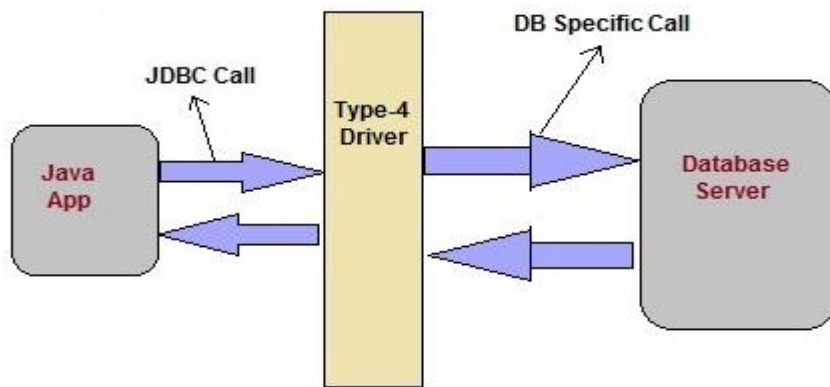
- This driver is server-based, so there is no need for any vendor database library to be present on client machines.
- This driver is fully written in Java and hence Portable. It is suitable for the web.
- This driver is very flexible allows access to multiple databases using one driver.
- They are the most efficient amongst all driver types

#### Disadvantage

- It requires another server application to install and maintain.

### Type-4 Driver or Thin Driver

This is Driver called Pure Java Driver because. This driver interact directly with database. It does not require any native database library, that is why it is also known as Thin Driver.



### Advantage

- The major benefit of using a type 4 jdbc drivers are that they are completely written in Java to achieve platform independence
- It is most suitable for the web.
- You don't need to install special software on the client or server.
- Further, these drivers can be downloaded dynamically

### Disadvantage

- Slow due to increase number of network call.

### Comparison between JDBC Drivers

Type	Type 1	Type 2	Type 3	Type 4
<b>Name</b>	JDBC-ODBC Bridge	Native Code Driver/ JNI	Java Protocol/ Middleware	Database Protocol
<b>Vendor Specific</b>	No	Yes	No	Yes
<b>Portable</b>	No	No	Yes	Yes
<b>Pure Java Driver</b>	No	No	Yes	Yes
<b>Working</b>	JDBC-> ODBC call ODBC -> native call	JDBC call -> native specific call	JDBC call -> middleware specific. Middleware -> native call	JDBC call ->DB specific call
<b>Multiple DB</b>	Yes [only ODBC supported DB]	NO	Yes [DB Driver should be in middleware]	No
<b>Example</b>	MS Access	Oracle OCI driver	IDA Server	MySQL
<b>Execution Speed</b>	Slowest among all	Faster Compared to Type1	Slower Compared to Type2	Fastest among all
<b>Driver</b>	Thick Driver	Thick Driver	Thin Driver	Thin Driver

### Java Database Connectivity with 5 Steps / Overview of the JDBC Process

There are 5 steps to connect any java application with the database using JDBC. These steps are as follows:



#### 1) Register the driver class

The **forName()** method of Class class is used to register the driver class. This method is used to dynamically load the driver class

##### Syntax of forName() method

**public static void** forName(String className)**throws** ClassNotFoundException

##### Example to register the MySQLDriver class

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

#### 2) Create the connection object

The **getConnection()** method of DriverManager class is used to establish connection with the database.

##### Syntax of getConnection() method

**public static** Connection getConnection(String url,String name,String password)

**throws** SQLException

### Example to establish connection with the MySQL database

```
Connection con = DriverManager.getConnection("jdbc:mysql://localhost:3308/java","root","");
```

### 3) Create the Statement object

The createStatement() method of Connection interface is used to create statement. The object of statement is responsible to execute queries with the database.

#### Syntax of createStatement() method

```
public Statement createStatement()throws SQLException
```

### Example to create the statement object

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

### 4) Execute the query

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 of executeQuery() method

```
public ResultSet executeQuery(String sql)throws SQLException
```

### Example to execute query

```
ResultSet rs=stmt.executeQuery("select * from author");

while(rs.next()){
    System.out.println(rs.getString(1)+" "+rs.getString(2) + " "+rs.getInt(3));
}
```

### 5) Close the connection object

By closing connection object statement and ResultSet will be closed automatically. The close() method of Connection interface is used to close the connection.

#### Syntax of close() method

```
public void close()throws SQLException
```



### Example to close connection

```
con.close();
```

### Code snippets for each type of JDBC connection

#### 1. MySQL

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

Connection conn=
DriverManager.getConnection("jdbc:mysql://localhost:PortNo/database
Name","uid", "pwd");
```

#### 2. Oracle

```
Class.forName("oracle.jdbc.driver.OracleDriver");

Connection conn=
DriverManager.getConnection("jdbc:oracle:thin:@hostname:port
Number:databaseName","root", "pwd");
```

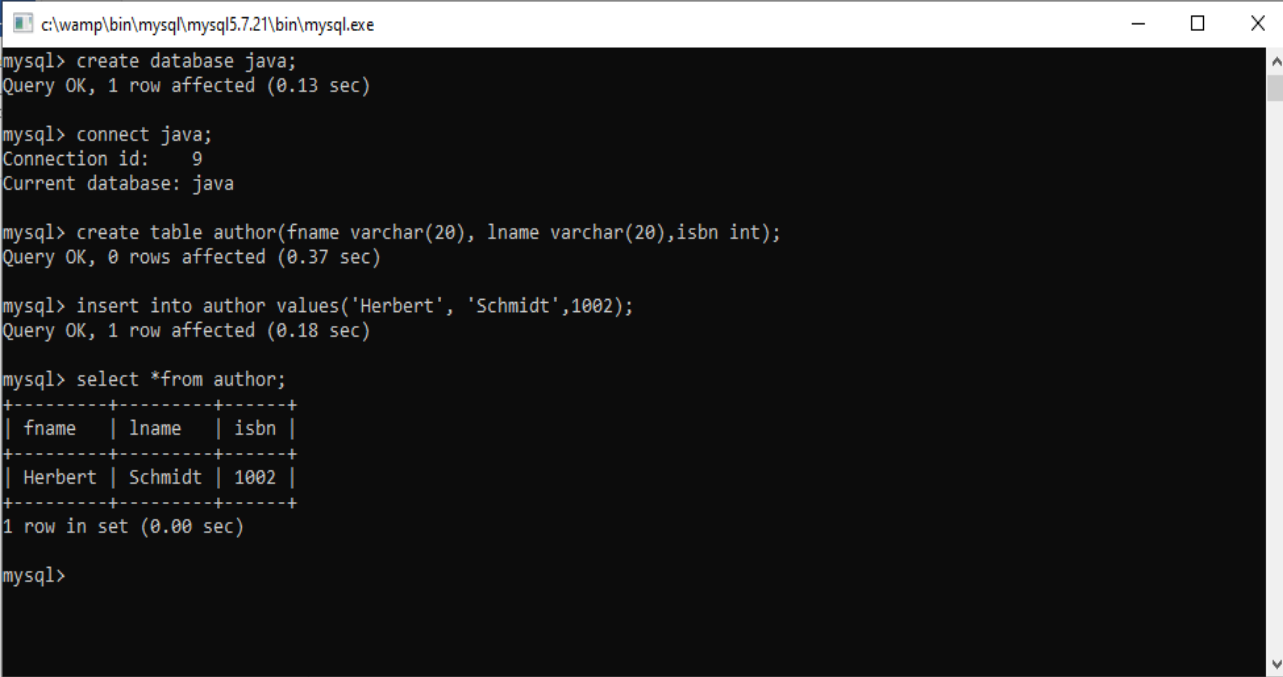
#### 3. DB2

```
Class.forName("com.ibm.db2.jdbc.net.DB2Driver");

Connection conn=
DriverManager.getConnection("jdbc:db2:hostname:port Number
/databaseName")
```

### Working with MYSql

To create a table author under the database named java. The author table has 3 attributes namely first name, last name of the author and isbn of the book and table is populated with the data.



```
c:\wamp\bin\mysql\mysql5.7.21\bin\mysql.exe
mysql> create database java;
Query OK, 1 row affected (0.13 sec)

mysql> connect java;
Connection id: 9
Current database: java

mysql> create table author(fname varchar(20), lname varchar(20),isbn int);
Query OK, 0 rows affected (0.37 sec)

mysql> insert into author values('Herbert', 'Schmidt',1002);
Query OK, 1 row affected (0.18 sec)

mysql> select *from author;
+-----+-----+-----+
| fname | lname | isbn |
+-----+-----+-----+
| Herbert | Schmidt | 1002 |
+-----+-----+-----+
1 row in set (0.00 sec)

mysql>
```

### Different types of statements in JDBC

- **Statement:** Use this for general-purpose access to your database. Useful when you are using static SQL statements at runtime. The Statement interface cannot accept parameters.
- **PreparedStatement:** Use this when you plan to use the SQL statements many times. The PreparedStatement interface accepts input parameters at runtime.
- **CallableStatement:** Use this when you want to access the database stored procedures. The CallableStatement interface can also accept runtime input parameters.

### Executing the Statement object *Statement*

The Statement interface represents the static SQL statement. It helps you to create a general purpose SQL statements using Java.

### Java.sql.Statement

- Used for general-purpose access to your database.
- Useful for **static** SQL statements, e.g. SELECT specific row from table etc.
- The Statement interface defines a standard abstraction to execute the SQL statements requested by a user and return the results by using the ResultSet object.
- The Statement interface is created after the connection to the specified database is made.
- The object is created using the createStatement() method of the Connection interface, as shown in following code snippet:

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

Once you have created the statement object you can execute it using one of the execute methods namely, execute(), executeUpdate() and, executeQuery().

- **boolean execute (String SQL):** Returns a boolean value of true if a ResultSet object can be retrieved; otherwise, it returns false. Use this method to execute SQL DDL statements or when you need to use truly dynamic SQL.
- **int executeUpdate (String SQL):** Returns the number of rows affected by the execution of the SQL statement. Use this method to execute SQL statements for which you expect to get a number of rows affected - for example, an INSERT, UPDATE, or DELETE statement.
- **ResultSet executeQuery (String SQL):** Returns a ResultSet object. Use this method when you expect to get a result set, as you would with a SELECT statement.

### // A JDBC Application to create a student table

```
import java.sql.*;

public class JDBC1
{
    public static void main(String[] args) throws Exception
    {
        Class.forName("com.mysql.jdbc.Driver");
        Connection con =
        DriverManager.getConnection("jdbc:mysql://localhost:3308/java","root","");
        Statement stmt = con.createStatement();

        String query ="create table STUDENT(ST_NAME varchar(20),ST_USN int, SEM int)";
        boolean b=stmt.execute(query);

        System.out.println("TABLE Created "+b);
        con.close();

    }
}
```

### OUTPUT

TABLE Created false

### // A JDBC Application to insert a record into the table

```
import java.sql.*;

public class JDBC1
{
    public static void main(String[] args) throws Exception
    {
        Class.forName("com.mysql.jdbc.Driver");
        Connection con =
        DriverManager.getConnection("jdbc:mysql://localhost:3308/java","root","");
        Statement stmt = con.createStatement();
        String query ="insert into author values('Jim','Keogh',1005)";
        int a=stmt.executeUpdate(query);
        System.out.println("No of Record Inserted "+a);
        con.close();
    }
}
```

### OUTPUT

No of Record Inserted 1

```
mysql> select *from author;
+-----+-----+-----+
| fname | lname | isbn |
+-----+-----+-----+
| Herbert | Schmidt | 1002 |
| Jim    | Keogh  | 1005 |
+-----+-----+-----+
2 rows in set (0.00 sec)
```

### // A JDBC Application to update records of the table

```
import java.sql.*;

public class JDBC1
{
    public static void main(String[] args) throws Exception
    {
        Class.forName("com.mysql.jdbc.Driver");
        Connection con =
        DriverManager.getConnection("jdbc:mysql://localhost:3308/java","root","");
        Statement stmt = con.createStatement();
        String query ="update author set isbn=2000 where fname='Jim'";
        int a=stmt.executeUpdate(query);
        System.out.println("No of Record Inserted   "+a);
        con.close();
    }
}
```

### // A JDBC Application to delete records from the table

```
import java.sql.*;

public class JDBC1
{
    public static void main(String[] args) throws Exception
    {
        Class.forName("com.mysql.jdbc.Driver");
        Connection con =
        DriverManager.getConnection("jdbc:mysql://localhost:3308/java","root","");
        Statement stmt = con.createStatement();
        String query ="delete from author where fname='Jim'";
        int a=stmt.executeUpdate(query);
        System.out.println("No of Record Inserted   "+a);
        con.close();
    }
}
```

### // A JDBC Application to display the contents of the table

```
import java.sql.*;

public class JDBC1
{
    public static void main(String[] args) throws Exception
    {
        Class.forName("com.mysql.jdbc.Driver");
        Connection con =
        DriverManager.getConnection("jdbc:mysql://localhost:3308/java","root","");
```

```
Statement stmt = con.createStatement();
String query = "SELECT fname,lname,isbn from author";
ResultSet rs = stmt.executeQuery(query);
System.out.println("Fname  Lname  ISBN");
while (rs.next())
{
    String fname = rs.getString("fname");
    String lname = rs.getString("lname");
    int isbn = rs.getInt("isbn");
    System.out.println(fname + "  " + lname+"  "+isbn);
    System.out.println("-----");
}
con.close();
System.out.println();
System.out.println();
}
```

### OUTPUT

```
Fname  Lname  ISBN
Herbert  Schmidt  1002
-----
```

### Java PreparedStatement Interface

- The PreparedStatement interface is subclass of the Statement interface, can be used to represent a precompiled query, which can be executed multiple times.
- Prepared Statement is used when you plan to execute same SQL statements many times.
- PreparedStatement interface accepts input parameters at runtime.
- A SQL statement is precompiled and stored in a PreparedStatement object.
- This object can then be used to efficiently execute this statement multiple times.

Prepared statements offer better performance, as they are pre-compiled. [Advantages of Prepared Statement in Java JDBC](#). benefit of using Prepared Statement is it prevents from SQL Injection. PreparedStatement is fast and gives better performance.

Prepared statements are more secure because they use bind variables, which can prevent SQL injection attack.

The most common type of SQL injection attack is SQL manipulation. The attacker attempts to modify the SQL statement by adding elements to the WHERE clause or extending the SQL with the set operators like UNION, INTERSECT etc.

**SQL Injection** is code injection technique where SQL is injected by user (as part of user input) into the back end query. Injected SQL data alters the purpose of original query and upon execution can gives harmful result.

**A SQL injection attack is very dangerous and attacker can,**

1. Read sensitive data from the database.
2. Update database data (Insert/Update/Delete).
3. Slowdown the complete Database system etc

Let us look at the following SQL

```
SELECT * FROM employee where ename='john' AND password='xyfdsw';
```

The attacker can manipulate the SQL as follows

```
SELECT * FROM employee where ename='john' AND password='xyfdsw' or 1 = 1;
```

The above “WHERE” clause is always true because of the operator precedence. The PreparedStatement can prevent this by using bind variables:

```
String strSQL = "SELECT * FROM employee where ename=? AND password=?";
PreparedStatement pstmt = myConnection.prepareStatement(strSQL);
pstmt.setString(1,"john");
pstmt.setString(2, "xyfdsw");
pstmt.execute();
```

### **// A JDBC Application to insert record into the table using PreparedStatement**

```
import java.sql.*;

public class JDBC1
{
    public static void main(String[] args) throws Exception
    {
        Class.forName("com.mysql.jdbc.Driver");
        Connection con =
        DriverManager.getConnection("jdbc:mysql://localhost:3308/java","root","");

        //Creating a Prepared Statement
        String query="INSERT INTO student(ST_NAME, ST_USN, SEM)VALUES(?, ?, ?)";
        PreparedStatement pstmt = con.prepareStatement(query);
        pstmt.setString(1, "Amit");
        pstmt.setInt(2, 300);
        pstmt.setInt(3, 4);
        pstmt.execute();
        System.out.println("Record Inserted");
    }
}
```

```
        con.close();  
    }  
}
```

### **Advantages of PreparedStatement Interface**

- The performance of the application will be faster, if you use PreparedStatement interface because query is compiled only once.
- This is because creating a PreparedStatement object by explicitly giving the SQL statement causes the statement to be precompiled within the database immediately.
- Thus, when the PreparedStatement is later executed, the DBMS does not have to recompile the SQL statement.
- Late binding and compilation is done by DBMS.

### **Disdvantage of PreparedStatement Interface**

The main disadvantage of PreparedStatement is that it can represent only one SQL statement at a time.



## Java CallableStatement Interface

- CallableStatement interface is used to call the stored procedures.
- Therefore, the stored procedure can be called by using an object of the CallableStatement interface.
- The object is created using the prepareCall() method of Connection interface.

```
CallableStatement cs=conn.prepareCall("{call Proc_Name(?,?)}");  
cs.setInt(1,2222);  
cs.registerOutParameter(2,Types.VARCHAR);  
cs.execute();
```

- Three types of parameters exist: IN, OUT, and INOUT.
- PreparedStatement object only uses the IN parameter. The CallableStatement object can use all the three.

Parameter	Description
IN	A parameter whose value is unknown when the SQL statement is created. You bind values to IN parameters with the setXXX() methods.
OUT	A parameter whose value is supplied by the SQL statement it returns. You retrieve values from the OUT parameters with the getXXX() methods.
INOUT	A parameter that provides both input and output values. You bind variables with the setXXX() methods and retrieve values with the getXXX() methods.

### Example of CallableStatement

**Write a Callable Statement program to retrieve branch of the student using {getBranch() procedure} from given enrollment number. Also write code for Stored Procedure**

**Stored Procedure: getbranch()**

```
1. DELIMITER @@
2. DROP PROCEDURE getbranch @@
3. CREATE PROCEDURE databaseName.getbranch
4. (IN enr_no INT, OUT my_branch VARCHAR(10))
5. BEGIN
6. SELECT branch INTO my_branch
7. FROM dietStudent
8. WHERE enr_no=enrno;
9. END @@
10. DELIMITER ;
```

### Callable Statement program

```
1. import java.sql.*;
2. public class CallableDemo {
3. public static void main(String[] args) {
4. try {
5.     Class.forName("com.mysql.jdbc.Driver");
6.     Connection conn= DriverManager.getConnection
7.         ("jdbc:mysql://localhost:3306/Diet", "root", "pwd");
8.
9.     CallableStatement cs=conn.prepareCall("{call getbranch(?,?)}");
10.         cs.setInt(1,2222);
11.         cs.registerOutParameter(2,Types.VARCHAR);
12.         cs.execute();
13.         System.out.println("branch="+cs.getString(2));
14.         cs.close();
15.         conn.close();
16.     }catch(Exceptione){System.out.println(e.toString());}
17. } //PSVM
18. } //class
```

## Differentiate Statement, Prepared Statement and Callable Statement.

Statement	Prepared Statement	Callable Statement
Super interface for Prepared and Callable Statement	extends Statement (sub-interface)	extends PreparedStatement (sub-interface)
Used for executing simple SQL statements like CRUD (create, retrieve, update and delete)	Used for executing dynamic and pre-compiled SQL statements	Used for executing stored procedures
The Statement interface cannot accept parameters.	The PreparedStatement interface accepts input parameters at runtime.	The CallableStatement interface can also accept runtime input parameters.
stmt = conn.createStatement();	PreparedStatement ps=con.prepareStatement ("insert into studentDiet values(?,?,?)");	CallableStatement cs=conn.prepareCall("{call getbranch(?,?)}");
java.sql.Statement is slower as compared to Prepared Statement in java JDBC.	PreparedStatement is faster because it is used for executing precompiled SQL statement in java JDBC.	None