|  |  |  |
| --- | --- | --- |
| **Statement** | **PreparedStatement** | **CallableStatement** |
| It is suitable to use  Statement only when we know that we will not need to execute the SQL query multiple times. | This interface is useful to use the same **SQL command multiple times,**like inserting number of records. This gives a **better performance**. | This interface is used for executing the SQL stored procedures.   It adds a level of abstraction, so the execution of stored procedures does not have to be DBMS-specific. |
| the Statement doesn’t offer support for the parameterized SQL queries. Parameterized SQL queries is an important protection from SQL injection attacks. | **PreparedStatement pst = con.prepareStatement(“insert into Employee (empid, empname, empsal) values (?, ?, ?)”);**  The **questions marks** in the above statement are called as **parameters**. The values are represented as question marks as their values are not known at compile time but known as runtime only. | the output parameters need to be explicitly defined through the corresponding registerOutParameter() methods; whereas the input parameters are provided in the same manner as with the PreparedStatement. |
| Statement would be suitable for the execution of the DDL (Data Definition Language) statements, such as CREATE, ALTER, DROP. | No SQL injection attacks problem |  |

**JDBC**

java.sql.Statement; Prepared Statement extends Statement; CallableStatement extends PreparedStatement; -- all are interface.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Statement** | **Create/prepare** | **execute** | **Return after execution** | **Operations** |
| Statement | con.createStatement()  con.createStatement(int, int)  con.createStatement(int, int, int)  String sql = “insert into table values(‘fds’,123,443)”; | st.excute(sql) | boolean | Insert, update, Delete |
| st.executeUpdate(sql) | int | Insert, update, Delete |
| st.executeQuery(sql) | ResultSet | Select |
| PreparedStatement | con.prepareStatement(String)  con.prepareStatement(String,int,int)  con.prepareStatement(String,int,int,int)  string = “insert into table values(?,?,?)”;  ps.setString(1,”XXXX”);  ps.setLong(2,23243443);  ps.setInt(3, 4343); | ps.execute() | boolean | Insert, update, Delete |
| ps.executeUpdate() | int | Insert, update, Delete |
| ps.executeQuery() | ResultSet  rs.getInt(1);//rs.getInt(“column\_name”);  rs.getString(2);//rs.getString(“column\_name”); | select |
| CallableStatement | con.prepareCall(String)  con.prepareCall(String, int, int)  con.prepareCall(String, int,int,int)  string =  “{call doMaths(?,?)}”;  cs.setInt(1,23443);//IN parameter  cs.registerOutParameter(2, Types.VARCHAR);//OUT Parameter  cs.setString(2,”xyz”);//IN Parameter  or,  cs.setInt(“param\_name”,123);  cs.registerOutParameter(“param\_name”, Types.VARCHAR);  cs.setString(“param\_name”, “dfsd”); | cs.execute() | boolean | To call stored procedure.  cs.getString(2);  or,  cs.getString(“param\_name”);  if we combine both number and param\_name for place holder value we’ll get runtime exception:  java.sql.SQLException: operation not allowed: Ordinal binding and Named binding cannot be combined! |
| cs.executeUpdate() | int |
| cs.executeQuery() | ResultSet |

Program Architecture:

UserDao.java

public int addUser(UserTo userTo ); public UserTo getUserByUserId(int userid); public List<UserTo> getAllUser(); public int updateUserPass(UserTo userTo); public int deleteUserByUserId(int userId);

UserDao.java

public int addUser(UserTo userTo ){} public UserTo getUserByUserId(int userid){} public List<UserTo> getAllUser(){} public int updateUserPass(UserTo userTo){} public int deleteUserByUserId(int userId){}

DataBase

JdbcUtils.java

public static Connection getConnection(){} public static void cleanup(){}

USR – user\_id, username, password

USR\_PROFILE – user\_id, first\_name, last\_name, mobile, email,

ADDRESS – address\_id, address\_line1, address\_line2, landmark, address\_type, city, state, country, pin, user\_id

* USR and USR\_PROFILE one to one mapping, bidirectional
* USER\_PROFILE and ADDRESS – one to many mapping, USER\_PROFILE to ADDRESS uni/bi directional

BOOKS – book\_id, book\_name, author, price

ORDER – order\_id, price, date, order\_date, status, delivery\_date, total\_books

ORDER\_BOOKS – order\_id, book\_id

* Many to many relationship between order and book
* Unidirectional from order to book

ADMIN – admin\_id, admin\_name, admin\_password

AUDIT – audit\_id, table, operation, created\_by, created\_date

**JSP, Servlet, JDBC**

user.jsp

UserServlet

BusinessDelegateFactory (class)

BusinessDelegate (interface)

BusinessDelegateImpl

DaoFactory (class)

UserDao (interface)

UserDaoImpl

JdbcUtils

UserTo

**JSP, Servlet, Hibernate**

user.jsp

UserServlet

BusinessDelegateFactory (class)

BusinessDelegate (interface)

BusinessDelegateImpl

DaoFactory (class)

UserDao (interface)

UserDaoImpl

HibernateUtils

User

hibernate-cfg.xml

**JSF, hibernate**

user.jsp

UserBean

BusinessDelegateFactory (class)

BusinessDelegate (interface)

BusinessDelegateImpl

DaoFactory (class)

UserDao (interface)

UserDaoImpl

HibernateUtils

User

hibernate-cfg.xml

managed-bean.xml

web.xml

**Spring, JDBC**

UserClient

StartBootstrap

BusinessDelegate (interface)

BusinessDelegateImpl (class)

UserDao (interface)

UserDaoImpl (class)

UserTo

applicationContext.xml

db-config.xml

**Spring, Hibernate**

UserClient

StartBootstrap

BusinessDelegate (interface)

BusinessDelegateImpl (class)

UserDao (interface)

UserDaoImpl (class)

User

applicationContext.xml

db-config.xml

hibernat-cfg.xml

**html, WebService, hibernate**

user.html

UserResource

BusinessDelegateFactory (class)

BusinessDelegate (interface)

BusinessDelegateImpl

DaoFactory (class)

UserDao (interface)

UserDaoImpl

HibernateUtils

User

hibernate-cfg.xml

**JSF, Spring, Hibernate**

**SpringMVC, Spring, Hibernate**