**JDBC**

JDBC stands for Java Database Connectivity. JDBC is a Java API to connect and execute the query with the database. It is a part of JavaSE (Java Standard Edition). JDBC API uses JDBC drivers to connect with the database. There are four types of JDBC drivers:

* JDBC-ODBC Bridge Driver,
* Native Driver,
* Network Protocol Driver, and
* Thin Driver

**1) Register the driver class**

|  |
| --- |
| The **forName()** method of Class is used to register the driver class. This method is used to dynamically load the driver class.   * Class.forName("oracle.jdbc.driver.OracleDriver"); |

**2) Create the connection object**

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

* Connection con;

con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","password");

### 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.   * 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. |  * ResultSet rs=stmt.executeQuery("select \* from emp");   **while**(rs.next()){  System.out.println(rs.getInt(1)+" "+rs.getString(2));  } |

### 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. |

* con.close();

# **1)JDBC-ODBC bridge driver**

# The JDBC-ODBC bridge driver uses ODBC driver to connect to the database. The JDBC-ODBC bridge driver converts JDBC method calls into the ODBC function calls. This is now discouraged because of thin driver. Oracle does not support the JDBC-ODBC Bridge from Java 8. Oracle recommends that you use JDBC drivers provided by the vendor of your database instead of the JDBC-ODBC Bridge.

**2)Native-API driver**

The Native API driver uses the client-side libraries of the database. The driver converts JDBC method calls into native calls of the database API. It is not written entirely in java.

**3)Network Protocol driver**

The Network Protocol driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol. It is fully written in java.

**4)Thin driver**

|  |
| --- |
| The thin driver converts JDBC calls directly into the vendor-specific database protocol. That is why it is known  as thin driver. It is fully written in Java language. |

# **DriverManager class**

The DriverManager class acts as an interface between user and drivers. It keeps track of the drivers that are available and handles establishing a connection between a database and the appropriate driver. The DriverManager class maintains a list of Driver classes

|  |  |
| --- | --- |
| **Method** | **Description** |
| public static void registerDriver(Driver driver): | is used to register the given driver with DriverManager. |
| public static void deregisterDriver(Driver driver): | is used to deregister the given driver (drop the driver from the list) with DriverManager. |
| public static Connection getConnection(String url): | is used to establish the connection with the specified url. |
| public static Connection getConnection(String url,String userName,String password): | is used to establish the connection with the specified url, username and password. |

**EG:- // Display the data from the database**

**package** com.lti.database;

**import** java.sql.Connection;

**import** java.sql.DriverManager;

**import** java.sql.ResultSet;

**import** java.sql.SQLException;

**import** java.sql.Statement;

**public** **class** DemoClass {

**public** **static** **void** main(String[] args) {

String url = "jdbc:oracle:thin:@localhost:1521:orcl";

String username = "hr";

String password = "annerajan";

String query = "Select \* from employees";

**try** {

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

Connection con;

con = DriverManager.*getConnection*(url, username, password);

Statement st = con.createStatement();

ResultSet rs = st.executeQuery(query);

**while** (rs.next()) {

System.***out***.println(rs.getInt(1) + " " + rs.getString(2));

}

} **catch** (ClassNotFoundException e) {

e.printStackTrace();

} **catch** (SQLException e) {

e.printStackTrace();

}

}

}

**Eg:- //Create table in the database**

**package** com.lti.database;

**import** java.sql.Connection;

**import** java.sql.DriverManager;

**import** java.sql.SQLException;

**import** java.sql.Statement;

**public** **class** CreateDemo {

**public** **static** **void** main(String[] args) {

String url = "jdbc:oracle:thin:@localhost:1521:orcl";

String username = "hr";

String password = "annerajan";

**try** {

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

Connection con;

con = DriverManager.*getConnection*(url, username, password);

Statement st = con.createStatement();

st.executeQuery("Create table student(SID INTEGER,Studentname varchar(20))");

} **catch** (Exception e) {

System.***out***.println("Some error occurred");

}

}

}

**Eg:- //Insert into the Database**

**package** com.lti.database;

**import** java.sql.Connection;

**import** java.sql.DriverManager;

**import** java.sql.Statement;

**public** **class** InsertDemo {

**public** **static** **void** main(String[] args) {

String url = "jdbc:oracle:thin:@localhost:1521:orcl";

String username = "hr";

String password = "annerajan";

String query = ("insert into student values(101,'Anne')");

**try** {

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

Connection con;

con = DriverManager.*getConnection*(url, username, password);

Statement st = con.createStatement();

st.executeQuery(query);

} **catch** (Exception e) {

System.***out***.println("Error has occurred");

}

}

}

**Eg //Update data in the database**

**package** com.lti.database;

**import** java.sql.Connection;

**import** java.sql.DriverManager;

**import** java.sql.Statement;

**public** **class** UpdateDemo {

**public** **static** **void** main(String[] args) {

String url = "jdbc:oracle:thin:@localhost:1521:orcl";

String username = "hr";

String password = "annerajan";

String query =("Update student set studentname='Anne Rajan' where studentname='Anne'");

**try** {

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

Connection con;

con = DriverManager.*getConnection*(url, username, password);

Statement st = con.createStatement();

st.execute(query);

} **catch** (Exception e) {

System.***out***.println("Error has occurred");

}}}

**Eg //Delete data from database**

**package** com.lti.database;

**import** java.sql.Connection;

**import** java.sql.DriverManager;

**import** java.sql.Statement;

**public** **class** DeleteDemo {

**p**ublic **static** **void** main(String[] args) {

String url = "jdbc:oracle:thin:@localhost:1521:orcl";

String username = "hr";

String password = "annerajan";

String query =("Delete from student where studentname='Mary'");

**try** {

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

Connection con;

con = DriverManager.*getConnection*(url, username, password);

Statement st = con.createStatement();

st.execute(query);

} **catch** (Exception e) {

System.***out***.println("Error has occurred");

}

}}

**Types of statements and types of result set in jdbc and explain with example**

There are three types of statements in JDBC namely, Statement, Prepared Statement, Callable statement. The Statement interface represents the static SQL statement. It helps you to create a general purpose SQL statements using Java.

**Creating a statement**

You can create an object of this interface using the **createStatement()** method of the **Connection** interface.

Create a statement by invoking the **createStatement()** method as shown below.

Statement stmt = null;

try {

stmt = conn.createStatement( );

. . .

}

catch (SQLException e) {

. . .

}

finally {

. . .

}

**Executing the Statement object**

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

* **execute():** This method is used to execute SQL DDL statements, it returns a boolean value specifying whether the ResultSet object can be retrieved.
* **executeUpdate():** This method is used to execute statements such as insert, update, delete. It returns an integer value representing the number of rows affected.
* **executeQuery():** This method is used to execute statements that returns tabular data (example SELECT statement). It returns an object of the class ResultSet.

**Prepared Statement**

The **PreparedStatement** interface extends the Statement interface. It represents a precompiled SQL statement which can be executed multiple times. This accepts parameterized SQL quires and you can pass 0 or more parameters to this query.

Initially, this statement uses place holders “**?**” instead of parameters, later on, you can pass arguments to these dynamically using the **setXXX()** methods of the **PreparedStatement** interface.

You can create an object of the **PreparedStatement** (interface) using the **prepareStatement()** method of the Connection interface. This method accepts a query (parameterized) and returns a PreparedStatement object.

String query="INSERT INTO Employee(Name, Salary, Location)VALUES(?, ?, ?)";

Statement pstmt = con.prepareStatement(query);

Executing the Prepared Statement

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

* **execute():** This method executes normal static SQL statements in the current prepared statement object and returns a boolean value.
* **executeUpdate():** This method executes SQL DML statements such as insert update or delete in the current Prepared statement. It returns an integer value representing the number of rows affected.
* **executeQuery():** This method executes the current prepared statement and returns a **ResultSet** object.

**CallableStatement**

The **CallableStatement** interface provides methods to execute stored procedures. Since the JDBC API provides a stored procedure SQL escape syntax, you can call stored procedures of all RDBMS in a single standard way.

Creating a CallableStatement

You can create an object of the **CallableStatement** (interface) using the **prepareCall()** method of the **Connection** interface.

This method accepts a string variable representing a query to call the stored procedure and returns a **CallableStatement** object.

CallableStatement cstmt = con.prepareCall("{call myProcedure(?, ?, ?)}");

Once you have created the CallableStatement object you can execute it using one of the **execute()** method.

**TYPE\_FORWARD\_ONLY:**  
The result set cannot be scrolled; its cursor moves forward only, from before the first row to after the last row. The rows contained in the result set depend on how the underlying database generates the results. That is, it contains the rows that satisfy the query at either the time the query is executed or as the rows are retrieved.  
  
**TYPE\_SCROLL\_INSENSITIVE:**  
The result can be scrolled; its cursor can move both forward and backward relative to the current position, and it can move to an absolute position. The result set is insensitive to changes made to the underlying data source while it is open. It contains the rows that satisfy the query at either the time the query is executed or as the rows are retrieved.

**TYPE\_SCROLL\_SENSITIVE:**  
The result can be scrolled; its cursor can move both forward and backward relative to the current position, and it can move to an absolute position. The result set reflects changes made to the underlying data source while the result set remains open.  
  
The default ResultSet type is TYPE\_FORWARD\_ONLY.

|  |  |
| --- | --- |
| **STATEMENT** | **PREPAREDSTATEMENT** |
| It is used when SQL query is to be executed only once. | It is used when SQL query is to be executed multiple times. |
| You can not pass parameters at runtime. | You can pass parameters at runtime. |
| Used for CREATE, ALTER, DROP statements. | Used for the queries which are to be executed multiple times. |
| Performance is very low. | Performance is better than Statement. |
| It is base interface. | It extends statement interface. |
| Used to execute normal SQL queries. | Used to execute dynamic SQL queries. |
| It is used for DDL statements. | It is used for any SQL Query. |

**What is a Stored Procedure?**

A stored procedure is a prepared SQL code that you can save, so the code can be reused over and over again.

So if you have an SQL query that you write over and over again, save it as a stored procedure, and then just call it to execute it.

You can also pass parameters to a stored procedure, so that the stored procedure can act based on the parameter value(s) that is passed.

|  |  |
| --- | --- |
| **Stored Procedure** | **Function** |
| is used to perform business logic. | is used to perform calculation. |
| must not have the return type. | must have the return type. |
| may return 0 or more values. | may return only one values. |
| We can call functions from the procedure. | Procedure cannot be called from function. |
| Procedure supports input and output parameters. | Function supports only input parameter. |
| Exception handling using try/catch block can be used in stored procedures. | Exception handling using try/catch can't be used in user defined  functions. |