CONTENT

- JDBC
- JDBC DRIVERS AND TYPES OF DRIVERS
- JDBC CONNECTIVITY
- DATABASE OPERATION: INSERT,DELETE,SELECT
- PREPARED STATEMENTS
- RESULTSET
- METADATA
- TRANSECTING QUERY
- BATCH UPDATION

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.

What is API (Application Programming Interface)?

API is a document that contains description of all features of a product or software. It represents classes and interfaces that software programs can follow to communicate with each other.

JDBC Components JDBC includes four components

1. The JDBC API

The JDBC API provides programmatic access to relational data from the Java programming language. Using the JDBC API, applications can execute SQL statements, retrieve results, and propagate changes back to an underlying data source. The JDBC API can also interact with multiple data sources in a distributed, heterogeneous environment.

2. JDBC Driver Manager

The JDBC DriverManager class defines objects which can connect Java applications to a JDBC driver. DriverManager has traditionally been the backbone of the JDBC architecture. It is quite small and simple.

3. JDBC Test Suite

The JDBC driver test suite helps you to determine that JDBC drivers will run your program. These tests are not comprehensive or exhaustive, but they do exercise many of the important features in the JDBC API.

4. JDBC-ODBC Bridge

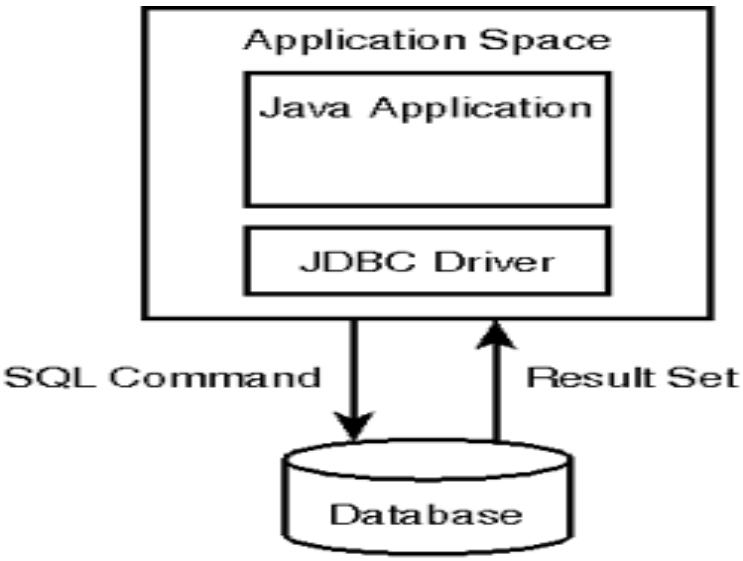
The Java Software Bridge provides JDBC access via ODBC drivers. Note that you need to load ODBC binary code onto each client machine that uses this driver. As a result, the ODBC driver is most appropriate on a corporate network where client installations are not a major problem, or for application server code written in Java in a three-tier architecture.

JDBC Architecture

The JDBC API supports both two-tier and three-tier processing models for database access

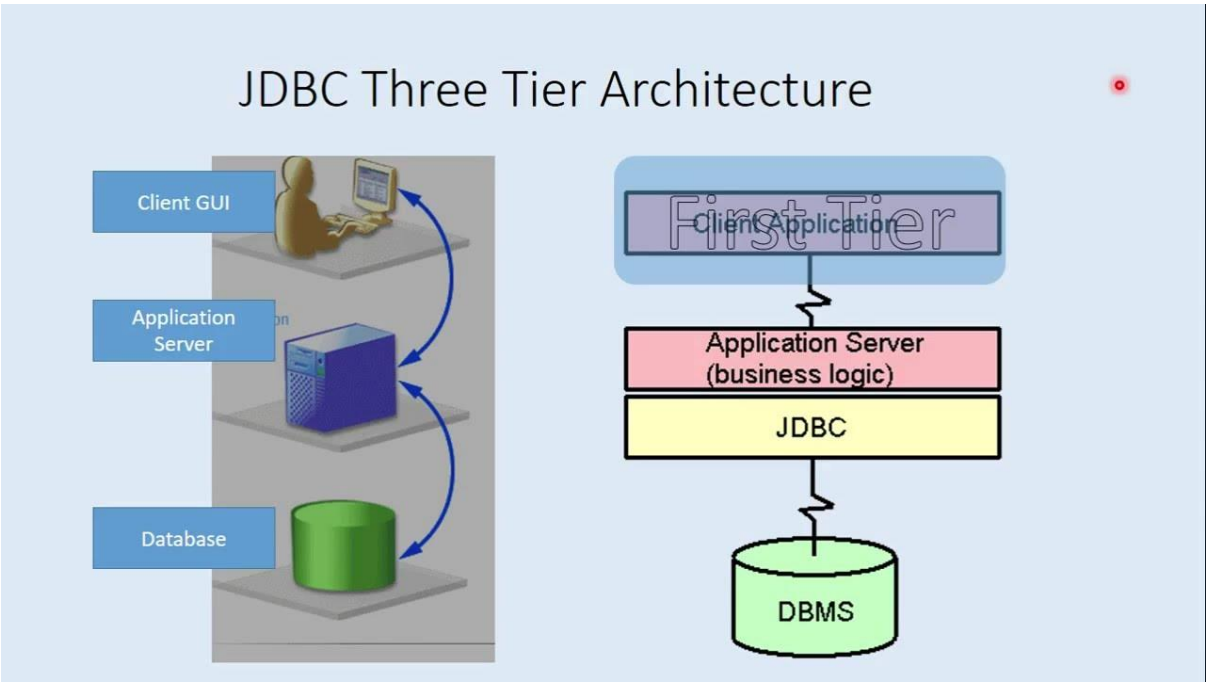
. • Two tier architecture

In the two-tier model, a Java applet or application talks directly to the data source. This requires a JDBC driver that can communicate with the particular data source being accessed. A user's commands are delivered to the database or other data source, and the results of those statements are sent back to the user. The data source may be located on another machine to which the user is connected via a network. This is referred to as a client/server configuration, with the user's machine as the client, and the machine housing the data source as the server. The network can be an intranet, which, for example, connects employees within a corporation, or it can be the Internet.



Three tier architecture

In the three-tier model, commands are sent to a "middle tier" of services, which then sends the commands to the data source. The data source processes the commands and sends the results back to the middle tier, which then sends them to the user. MIS directors find the three-tier model very attractive because the middle tier makes it possible to maintain control over access and the kinds of updates that can be made to corporate data. Another advantage is that it simplifies the deployment of applications. Finally, in many cases, the three-tier architecture can provide performance advantages. With enterprises increasingly using the Java programming language for writing server code, the JDBC API is being used more and more in the middle tier of a three-tier architecture. Some of the features that make JDBC a server technology are its support for connection pooling, distributed transactions, and disconnected row sets. The JDBC API is also what allows access to a data source from a Java middle tier.



JDBC Driver

A JDBC driver is a software component enabling a java application to interact with a database. JDBC requires drivers for each database. The JDBC driver gives out the connection to the database and implements the protocol for transferring the query and result between client and database. JDBC drivers are client-side adaptors that convert request from java program to a protocol that that the DBMS can understand.

There are four types of driver

1. JDBC-ODBC Bridge Driver (Type 1)
2. Native API Driver (Type 2/Partially Java Driver)
3. Network Protocol Driver (Type 3/Fully Java Driver)
4. Thin Driver (Type 4/Fully Java Driver)

1. JDBC-ODBC Bridge Driver (Type 1)

- Uses ODBC bridge driver to connect to database.
- Converts JDBC methods call into function call.

- ODBC drivers need to be installed on client machine.
- This driver is platform dependent.

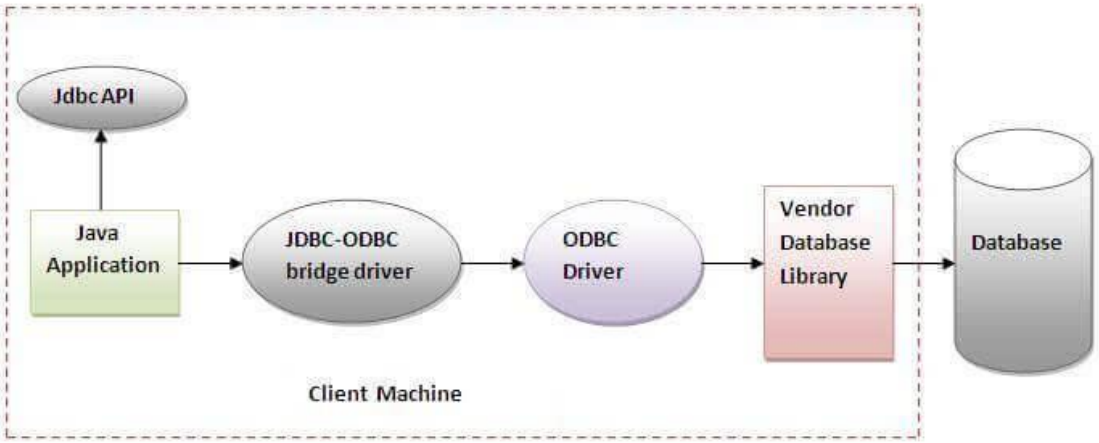


Figure- JDBC-ODBC Bridge Driver

Advantages:

- easy to use.
- can be easily connected to any database.

Disadvantages:

- Performance degraded because JDBC method call is converted into the ODBC function calls.
- The ODBC driver needs to be installed on the client machine.

2) Native-API driver

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

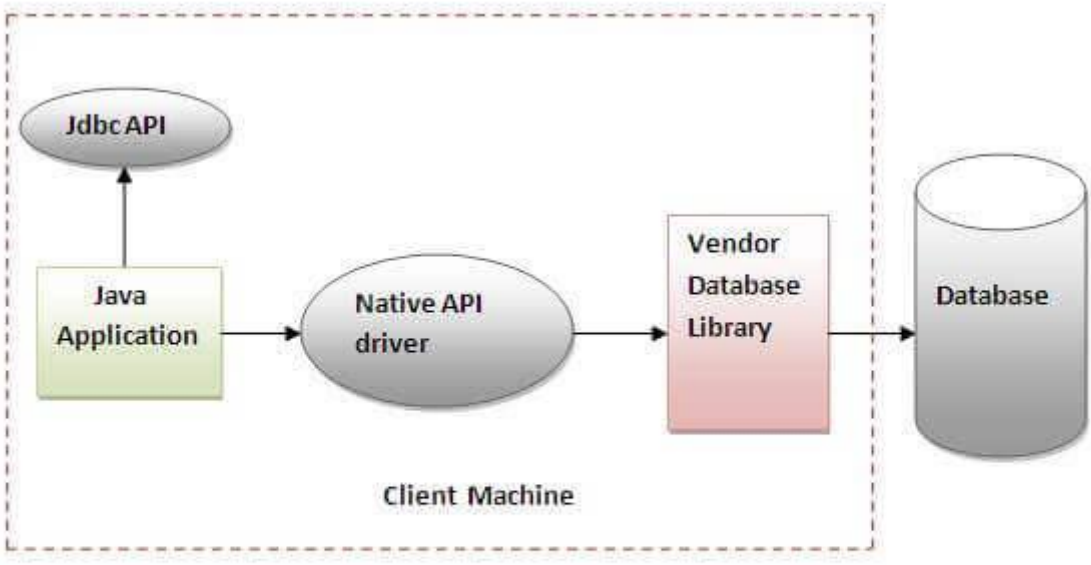


Figure- Native API Driver

Advantage:

- performance upgraded than JDBC-ODBC bridge driver.

Disadvantage:

- The Native driver needs to be installed on the each client machine.
- The Vendor client library needs to be installed on client machine.

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.

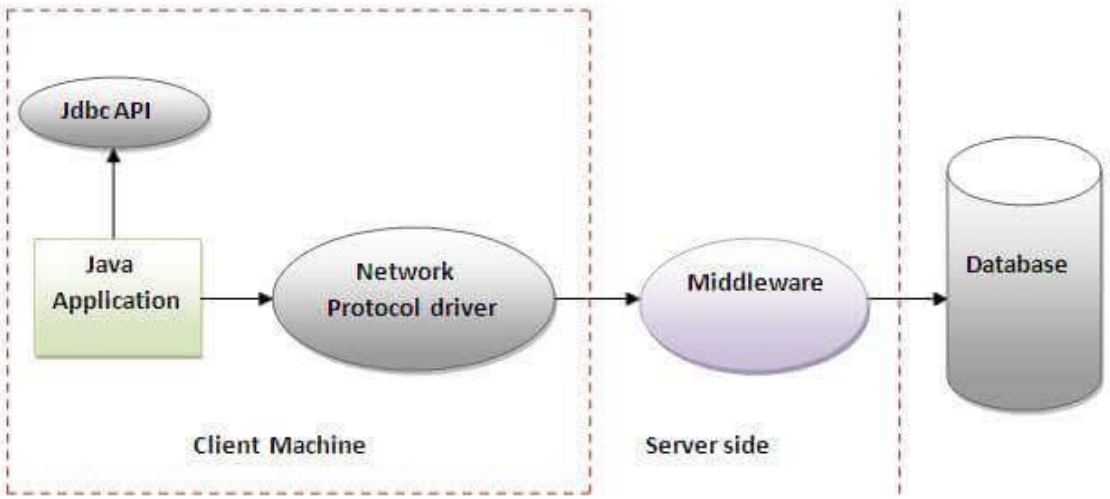


Figure- Network Protocol Driver

Advantage:

- No client side library is required because of application server that can perform many tasks like auditing, load balancing, logging etc.

Disadvantages:

- Network support is required on client machine.
- Requires database-specific coding to be done in the middle tier.
- Maintenance of Network Protocol driver becomes costly because it requires database-specific coding to be done in the middle tier.

---

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.

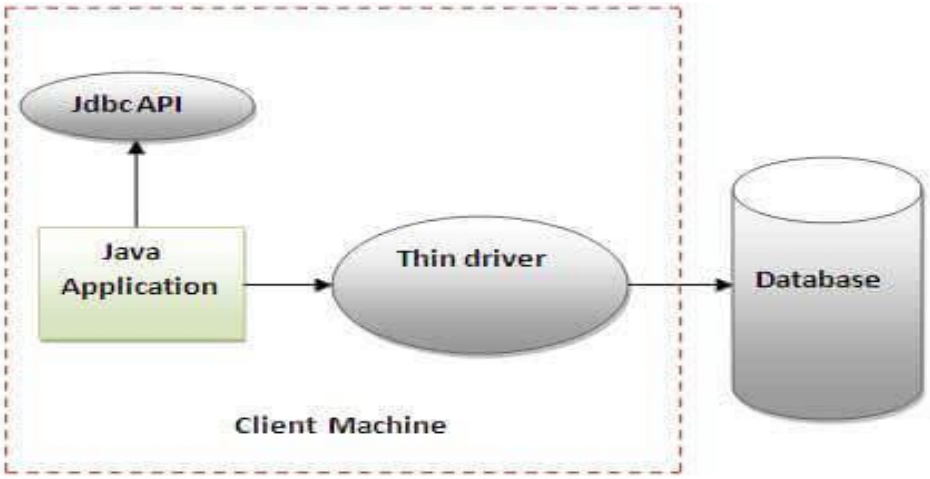


Figure- Thin Driver

Advantage:

- Better performance than all other drivers.
- No software is required at client side or server side.

Disadvantage:

- Drivers depend on the Database.

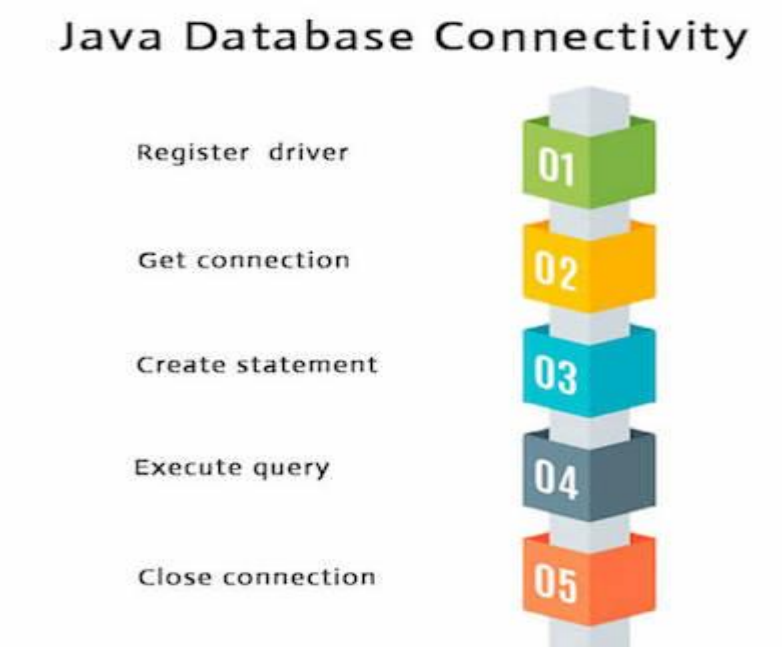
Required Steps

The following steps are required to create a new Database using JDBC application –

- **Import the packages** – Requires that you include the packages containing the JDBC classes needed for database programming. Most often, using *import java.sql.** will suffice.
- **Open a connection** –Requires using the *DriverManager.getConnection()* method to create a Connection object, which represents a physical connection with the database server.

  To create a new database, you need not give any database name while preparing database URL as mentioned in the below example.

- **Execute a query** – Requires using an object of type Statement for building and submitting an SQL statement to the database.
- **Clean up the environment**. try with resources automatically closes the resources.



1) Register the driver class

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

Syntax of forName() method

```
public static void forName(String className)throws ClassNotFoundExceptio
n
```

EX.

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

Create the

connection object

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

Syntax of getConnection() method

```
1) public static Connection getConnection(String url)throws SQLException
2) public static Connection getConnection(String url,String name,String passwo
rd)
throws SQLException
```

EX.

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

Create the Statement object

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

```
public Statement createStatement()throws SQLException
    Statement stmt=con.createStatement();
```

Execute the query

The executeQuery() method of Statement interface is used to execute queries to
database. This method returns the object of ResultSet that can be used to get all
records of a table.

Syntax of executeQuery() method

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

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

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

Close the connection object

By closing connection object statement and ResultSet will be closed automatically. T
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();
```

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

public class JDBCExample {
  static final String DB_URL = "jdbc:mysql:  //localhost/";
  static final String USER = "guest";
  static final String PASS = "guest123";

  public static void main(String[] args) {
    // Open a connection
    try(Connection conn = DriverManager.getConnection(DB_URL, USER,
PASS);
      Statement stmt = conn.createStatement();
    ) {
      String sql = "CREATE DATABASE STUDENTS";
      stmt.executeUpdate(sql);
      System.out.println("Database created successfully...");
    } catch (SQLException e) {
      e.printStackTrace();
    }
  }
}
```

Code to create a Table

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

```
    // Open a connection
    try(Connection conn = DriverManager.getConnection(DB_URL, USER,
PASS);
       Statement stmt = conn.createStatement();
    ) {
       String sql = "CREATE DATABASE STUDENTS";
       stmt.executeUpdate(sql);
       System.out.println("Database created successfully...");
    } catch (SQLException e) {
       e.printStackTrace();
    }
```

Code to drop database

```
try(Connection conn = DriverManager.getConnection(DB_URL, USER, PASS);
       Statement stmt = conn.createStatement();
    ) {
       String sql = "DROP DATABASE STUDENTS";
       stmt.executeUpdate(sql);
       System.out.println("Database dropped successfully...");
    } catch (SQLException e) {
       e.printStackTrace();
    }
```

Create tables

```
String sql = "CREATE TABLE REGISTRATION " +
                "(id INTEGER not NULL, " +
                " first VARCHAR(255), " +
                " last VARCHAR(255), " +
                " age INTEGER, " +
                " PRIMARY KEY ( id ))";

       stmt.executeUpdate(sql);
```

create table

```
try(Connection conn = DriverManager.getConnection(DB_URL, USER, PASS);
       Statement stmt = conn.createStatement();
    ) {
```

```
        String sql = "CREATE TABLE REGISTRATION " +
                "(id INTEGER not NULL, " +
                " first VARCHAR(255), " +
                " last VARCHAR(255), " +
                " age INTEGER, " +
                " PRIMARY KEY ( id ))";

        stmt.executeUpdate(sql);
        System.out.println("Created table in given database...");
    } catch (SQLException e) {
        e.printStackTrace();
    }
```

Drop tables

```
try(Connection conn = DriverManager.getConnection(DB_URL, USER, PASS);
        Statement stmt = conn.createStatement();
    ) {
        String sql = "DROP TABLE REGISTRATION";
        stmt.executeUpdate(sql);
        System.out.println("Table deleted in given database...");
    } catch (SQLException e) {
        e.printStackTrace();
    }
```

Insert records

```
try(Connection conn = DriverManager.getConnection(DB_URL, USER, PASS);
        Statement stmt = conn.createStatement();
    ) {
        // Execute a query
        System.out.println("Inserting records into the table...");
        String sql = "INSERT INTO Registration VALUES (100, 'Zara', 'Ali',
18)";
        stmt.executeUpdate(sql);
        sql = "INSERT INTO Registration VALUES (101, 'Mahnaz', 'Fatma',
25)";
        stmt.executeUpdate(sql);
        sql = "INSERT INTO Registration VALUES (102, 'Zaid', 'Khan', 30)";
```

```
        stmt.executeUpdate(sql);
        sql = "INSERT INTO Registration VALUES(103, 'Sumit', 'Mittal',
28)";
        stmt.executeUpdate(sql);
        System.out.println("Inserted records into the table...");
    } catch (SQLException e) {
        e.printStackTrace();
    }
```

Prepared Statement interface

The Prepared Statement interface is a sub interface of Statement. It is used to execute parameterized query.

example of parameterized query:

String sql="insert into emp values(?,?,?)";

As you can see, we are passing parameter (?) for the values. Its value will be set by calling the setter methods of Prepared Statement.

Why use Prepared Statement?

Improves performance: The performance of the application will be faster if you use Prepared Statement interface because query is compiled only once.

Method	Description
public void setInt(int paramIndex, int value)	sets the integer value to the given parameter index.
public void setString(int paramIndex, String value)	sets the String value to the given parameter index.
public void setFloat(int paramIndex, float value)	sets the float value to the given parameter index.

public void setDouble(int paramIndex, double value)	sets the double value to the given parameter index.
public int executeUpdate()	executes the query. It is used for create, drop, insert, update, delete etc.
public ResultSet executeQuery()	executes the select query. It returns an instance of ResultSet.

Example of PreparedStatement interface that inserts the record

First of all create table as given below:

```
create table emp(id number(10),name varchar2(50));
```

```
import java.sql.*;
class InsertPrepared{
public static void main(String args[]){
try{
Class.forName("oracle.jdbc.driver.OracleDriver");

Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:152
1:xe","system","oracle");

PreparedStatement stmt=con.prepareStatement("insert into Emp values(?,?)");

stmt.setInt(1,101);          //1 specifies the first parameter in the query
stmt.setString(2,"Ratan");

int i=stmt.executeUpdate();
System.out.println(i+" records inserted");

con.close();

}catch(Exception e){ System.out.println(e);}

}
}
```

The PreparedStatement Objects

The *PreparedStatement* interface extends the Statement interface, which gives you added functionality with a couple of advantages over a generic Statement object.

This statement gives you the flexibility of supplying arguments dynamically.

Creating PreparedStatement Object

```
PreparedStatement pstmt = null;
try {
   String SQL = "Update Employees SET age = ? WHERE id = ?";
   pstmt = conn.prepareStatement(SQL);
   . . .
}
catch (SQLException e) {
   . . .
}
finally {
   . . .
}
```

All parameters in JDBC are represented by the **?** symbol, which is known as the parameter marker. You must supply values for every parameter before executing the SQL statement.

The **setXXX()** methods bind values to the parameters, where **XXX** represents the Java data type of the value you wish to bind to the input parameter. If you forget to supply the values, you will receive an SQLException.

Each parameter marker is referred by its ordinal position. The first marker represents position 1, the next position 2, and so forth. This method differs from that of Java array indices, which starts at 0.

All of the **Statement object's** methods for interacting with the database

(a) execute(),

(b) executeQuery(), and

(c) executeUpdate() also work with the PreparedStatement object. However, the methods are modified to use SQL statements that can input the parameters.

Closing PreparedStatement Object

Just as you close a Statement object, for the same reason you should also close the PreparedStatement object.

A simple call to the close() method will do the job. If you close the Connection object first, it will close the PreparedStatement object as well. However, you should always explicitly close the PreparedStatement object to ensure proper cleanup.

```
PreparedStatement pstmt = null;
try {
   String SQL = "Update Employees SET age = ? WHERE id = ?";
   pstmt = conn.prepareStatement(SQL);
   . . .
}
catch (SQLException e) {
   . . .
}
finally {
   pstmt.close();
}
```

ResultSet interface

The object of ResultSet maintains a cursor pointing to a row of a table. Initially, cursor points to before the first row.

By default, ResultSet object can be moved forward only and it is not updatable.

But we can make this object to move forward and backward direction by passing either TYPE_SCROLL_INSENSITIVE or TYPE_SCROLL_SENSITIVE in createStatement(int,int) method as well as we can make this object as updatable by:

1. Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,

2. ResultSet.CONCUR_UPDATABLE);

Commonly used methods of ResultSet interface

1) public boolean next():	is used to move the cursor to the one row next from the current position.
2) public boolean previous():	is used to move the cursor to the one row previous from the current position.
3) public boolean first():	is used to move the cursor to the first row in result set object.
4) public boolean last():	is used to move the cursor to the last row in result set object.
5) public boolean absolute(int row):	is used to move the cursor to the specified row number in the ResultSet object.
6) public boolean relative(int row):	is used to move the cursor to the relative row number in the ResultSet object, it may be positive or negative.
7) public int getInt(int columnIndex):	is used to return the data of specified column index of the current row as int.
8) public int getInt(String columnName):	is used to return the data of specified column name of the current row as int.
9) public String getString(int columnIndex):	is used to return the data of specified column index of the current row as String.
10) public String getString(String columnName):	is used to return the data of specified column name of the current row as String.

Example of Scrollable ResultSet

Let's see the simple example of ResultSet interface to retrieve the data of 3rd row.

```
import java.sql.*;
class FetchRecord{
public static void main(String args[])throws Exception{

Class.forName("oracle.jdbc.driver.OracleDriver");
Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1
521:xe","system","oracle");
Statement stmt=con.createStatement(ResultSet.TYPE_SCROLL_SENSITI
VE,ResultSet.CONCUR_UPDATABLE);
ResultSet rs=stmt.executeQuery("select * from emp765");

//getting the record of 3rd row
rs.absolute(3);
System.out.println(rs.getString(1)+" "+rs.getString(2)+" "+rs.getString(3));

con.close();
}}
```

Java ResultSetMetaData Interface

The metadata means data about data i.e. we can get further information from the data.

If you have to get metadata of a table like total number of column, column name, column type etc. , ResultSetMetaData interface is useful because it provides methods to get metadata from the ResultSet object.

Commonly used methods of ResultSetMetaData interface

Method	Description
public int getColumnCount()throws SQLException	it returns the total number of columns in the ResultSet object.

public String getColumnName(int index)throws SQLException	it returns the column name of the specified column index.
public String getColumnTypeName(int index)throws SQLException	it returns the column type name for the specified index.
public String getTableName(int index)throws SQLException	it returns the table name for the specified column index.

How to get the object of ResultSetMetaData:

The getMetaData() method of ResultSet interface returns the object ResultSetMetaData. Syntax:

```
1. public ResultSetMetaData getMetaData()throws SQLException
```

Example of ResultSetMetaData interface :

```
1. import java.sql.*;
2. class Rsmd{
3. public static void main(String args[]){
4. try{
5. Class.forName("oracle.jdbc.driver.OracleDriver");
6. Connection con=DriverManager.getConnection(
7. "jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
8.
9. PreparedStatement ps=con.prepareStatement("select * from emp");
10.     ResultSet rs=ps.executeQuery();
11.ResultSetMetaData rsmd=rs.getMetaData();
12.
13.     System.out.println("Total columns: "+rsmd.getColumnCount());
14.     System.out.println("Column Name of 1st column: "+rsmd.getColumnNam
   e(1));
15.     System.out.println("Column Type Name of 1st column: "+rsmd.getColum
   nTypeName(1));
16.
```

```
17.         con.close();
18.         }catch(Exception e){ System.out.println(e);}
19.         }
20.         }
```

```
Output:Total columns: 2
       Column Name of 1st column: ID
       Column Type Name of 1st column: NUMBER
```

Callable

There are two ways of creating threads – one by extending the Thread class and other by creating a thread with a Runnable. However, one feature lacking in Runnable is that we cannot make a thread return result when it terminates, i.e. when run() completes. For supporting this feature, the Callable interface is present in Java.

Callable vs Runnable

- For implementing Runnable, the run() method needs to be implemented which does not return anything, while for a Callable, the call() method needs to be implemented which returns a result on completion. Note that a thread can't be created with a Callable, it can only be created with a Runnable.
- Another difference is that the call() method can throw an exception whereas run() cannot.

Method signature that has to overridden for implementing Callable.

```
public Object call() throws Exception;
```

```
// Java program to illustrate Callable
// to return a random number
import java.util.Random;
import java.util.concurrent.Callable;
import java.util.concurrent.FutureTask;

class CallableExample implements Callable
{

    public Object call() throws Exception
```

```
    {
        // Create random number generator
        Random generator = new Random();

        Integer randomNumber = generator.nextInt(5);

        // To simulate a heavy computation,
        // we delay the thread for some random time
        Thread.sleep(randomNumber * 1000);

        return randomNumber;
    }
}
```

Future

When the call() method completes, answer must be stored in an object known to the main thread, so that the main thread can know about the result that the thread returned. How will the program store and obtain this result later? For this, a Future object can be used. Think of a Future as an object that holds the result – it may not hold it right now, but it will do so in the future (once the Callable returns). Thus, a Future is basically one way the main thread can keep track of the progress and result from other threads. To implement this interface, 5 methods have to be overridden, but as the example below uses a concrete implementation from the library, only the important methods are listed here.

Observe that Callable and Future do two different things – Callable is similar to Runnable, in that it encapsulates a task that is meant to run on another thread, whereas a Future is used to store a result obtained from a different thread. In fact, the Future can be made to work with Runnable as well, which is something that will become clear when Executors come into the picture.

public boolean cancel(boolean mayInterrupt): Used to stop the task. It stops the task if it has not started. If it has started, it interrupts the task only if mayInterrupt is true.
public Object get() throws InterruptedException, ExecutionException: Used to get the result of the task. If the task is complete, it returns the result immediately, otherwise it waits till the task is complete and then returns the result.
public boolean isDone(): Returns true if the task is complete and false otherwise
To create the thread, a Runnable is required. To obtain the result, a Future is required.

The Java library has the concrete type FutureTask, which implements Runnable and Future, combining both functionality conveniently.
A FutureTask can be created by providing its constructor with a Callable. Then the FutureTask object is provided to the constructor of Thread to create the Thread object. Thus, indirectly, the thread is created with a Callable. For further emphasis, note that there is no way to create the thread directly with a Callable.

Here is the code for the full example using Callable and FutureTask.

```
// Java program to illustrate Callable and FutureTask
// for random number generation
import java.util.Random;
import java.util.concurrent.Callable;
import java.util.concurrent.FutureTask;

class CallableExample implements Callable
{

  public Object call() throws Exception
  {
    Random generator = new Random();
    Integer randomNumber = generator.nextInt(5);

    Thread.sleep(randomNumber * 1000);

    return randomNumber;
  }

}
```

```
public class CallableFutureTest
{
  public static void main(String[] args) throws Exception
  {

    // FutureTask is a concrete class that
    // implements both Runnable and Future
    FutureTask[] randomNumberTasks = new FutureTask[5];

    for (int i = 0; i < 5; i++)
    {
      Callable callable = new CallableExample();

      // Create the FutureTask with Callable
      randomNumberTasks[i] = new FutureTask(callable);

      // As it implements Runnable, create Thread
      // with FutureTask
      Thread t = new Thread(randomNumberTasks[i]);
      t.start();
    }

    for (int i = 0; i < 5; i++)
    {
      // As it implements Future, we can call get()
      System.out.println(randomNumberTasks[i].get());

      // This method blocks till the result is obtained
      // The get method can throw checked exceptions
      // like when it is interrupted. This is the reason
      // for adding the throws clause to main
    }
```

```
  }
}
```

Output:

```
4
2
3
3
0
```

The Callable Statement Objects

Just as a Connection object creates the Statement and PreparedStatement objects, it also creates the Callable Statement object, which would be used to execute a call to a database stored procedure.

```
DROP PROCEDURE IF EXISTS `EMP`.`getEmpName` $$
CREATE PROCEDURE `EMP`.`getEmpName`
  (IN EMP_ID INT, OUT EMP_FIRST VARCHAR(255))
BEGIN
  SELECT first INTO EMP_FIRST
  FROM Employees
  WHERE ID = EMP_ID;
END $$

DELIMITER ;
```

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

Here are the definitions of each –

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 theOUT 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.

The following code snippet shows how to employ the **Connection.prepareCall()** method to instantiate a **CallableStatement** object based on the preceding stored procedure –

```
CallableStatement cstmt = null;
try {
   String SQL = "{call getEmpName (?, ?)}";
   cstmt = conn.prepareCall (SQL);
   . . .
}
catch (SQLException e) {
   . . .
}
finally {
   . . .
}
```

The String variable SQL, represents the stored procedure, with parameter placeholders.

Using the CallableStatement objects is much like using the PreparedStatement objects. You must bind values to all the parameters before executing the statement, or you will receive an SQLException.

If you have IN parameters, just follow the same rules and techniques that apply to a PreparedStatement object; use the setXXX() method that corresponds to the Java data type you are binding.

When you use OUT and INOUT parameters you must employ an additional CallableStatement method, registerOutParameter(). The registerOutParameter() method binds the JDBC data type, to the data type that the stored procedure is expected to return.

Once you call your stored procedure, you retrieve the value from the OUT parameter with the appropriate getXXX() method. This method casts the retrieved value of SQL type to a Java data type.

Closing CallableStatement Object

Just as you close other Statement object, for the same reason you should also close the CallableStatement object.

A simple call to the close() method will do the job. If you close the Connection object first, it will close the CallableStatement object as well. However, you should always explicitly close the CallableStatement object to ensure proper cleanup.

```
CallableStatement cstmt = null;
try {
   String SQL = "{call getEmpName (?, ?)}";
   cstmt = conn.prepareCall (SQL);
   . . .
}
catch (SQLException e) {
   . . .
}
finally {
   cstmt.close();
}
```

Ans. Java.sql.Statement
Explain Statement Interface with appropriate example.
Ans. Java.sql.Statement
Ans.

Batch Processing in JDBC

Instead of executing a single query, we can execute a batch (group) of queries. It makes the performance fast.

The java.sql.Statement and java.sql.PreparedStatement interfaces provide methods for batch processing.

Advantage of Batch Processing

Fast Performance

Methods of Statement interface

The required methods for batch processing are given below:

Method	Description
void addBatch(String query)	It adds query into batch.
int[] executeBatch()	It executes the batch of queries.

Example of batch processing in jdbc

Let's see the simple example of batch processing in jdbc. It follows following steps:

- Load the driver class
- Create Connection
- Create Statement
- Add query in the batch
- Execute Batch
- Close Connection

```
import java.sql.*;
class FetchRecords{
public static void main(String args[])throws Exception{
Class.forName("oracle.jdbc.driver.OracleDriver");
```

```
Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
con.setAutoCommit(false);

Statement stmt=con.createStatement();
stmt.addBatch("insert into user420 values(190,'abhi',40000)");
stmt.addBatch("insert into user420 values(191,'umesh',50000)");

stmt.executeBatch();//executing the batch

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

If you see the table user420, two records has been added.

Example of batch processing using PreparedStatement

```
import java.sql.*;
import java.io.*;
class BP{
public static void main(String args[]){
try{

Class.forName("oracle.jdbc.driver.OracleDriver");
Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","oracle");

PreparedStatement ps=con.prepareStatement("insert into user420 values (?,?,?)");

BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
while(true){

System.out.println("enter id");
String s1=br.readLine();
int id=Integer.parseInt(s1);
```

```
System.out.println("enter name");
String name=br.readLine();

System.out.println("enter salary");
String s3=br.readLine();
int salary=Integer.parseInt(s3);

ps.setInt(1,id);
ps.setString(2,name);
ps.setInt(3,salary);

ps.addBatch();
System.out.println("Want to add more records y/n");
String ans=br.readLine();
if(ans.equals("n")){
break;
}

}
ps.executeBatch();

System.out.println("record successfully saved");

con.close();
}catch(Exception e){System.out.println(e);}

}}
```

It will add the queries into the batch until user press n. Finally it executes the batch. Thus all the added queries will be fired.