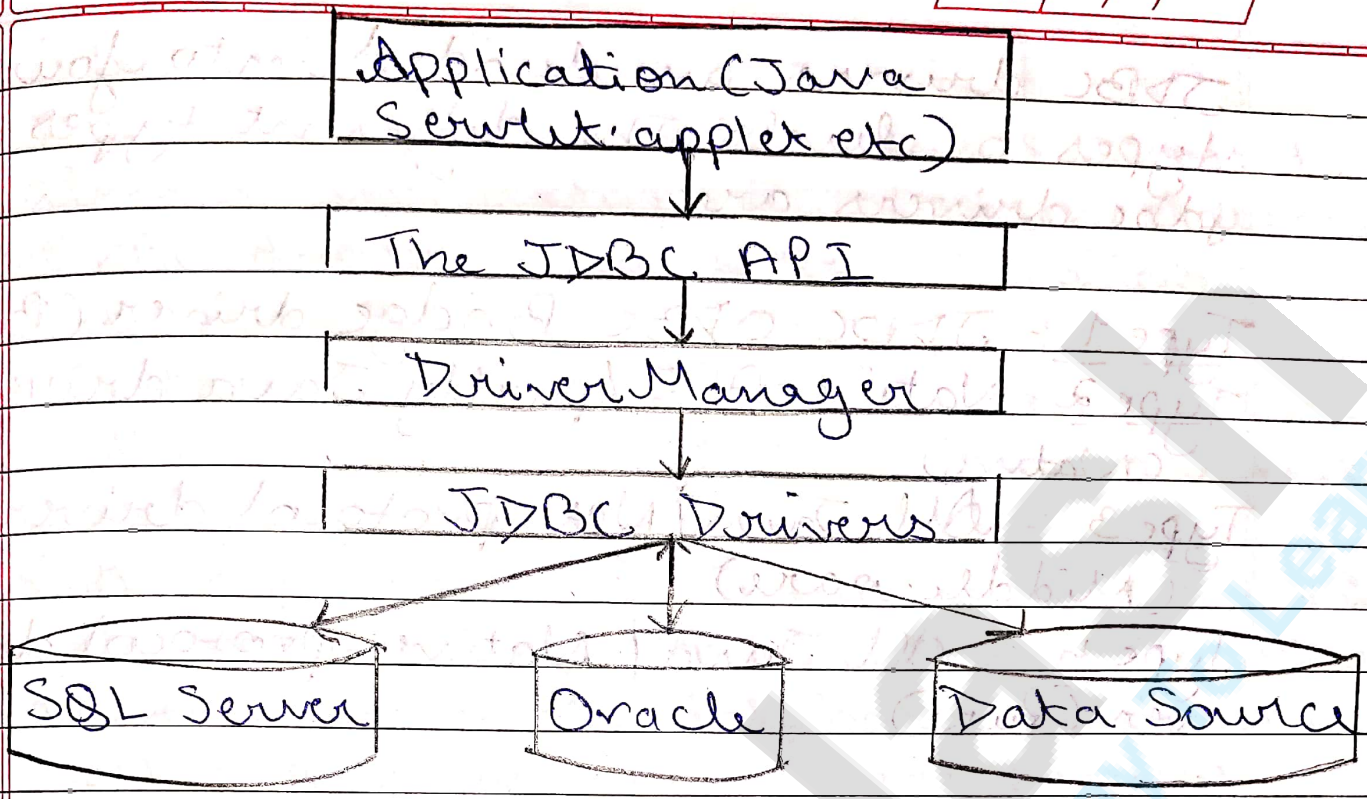
Unit 9

JDBC Architecture

As we know that driver is required to communicate with database. JDBC API provides classes and interfaces to handle request made by user and response made by database. Some of the important JDBC API are as under:

- DriverManager
- Connection
- PreparedStatement
- ResultSet
- ResultMetaData
- Driver
- Statement
- CallableStatement
- DatabaseMetaData

Here the DriverManager plays an important role in JDBC architecture. It uses some database specific drivers to communicate our J2EE application to database.



As per the diagram first of all we have to program our application with JDBC API.
With the help of DriverManager class, then we connect to a specific database with the help of specific database driver.
Java drivers require some library to communicate with the database.
We have four different types of java drivers.
Some drivers are pure java drivers and some are partial.
So with this kind of JDBC architecture we can communicate with specific database.

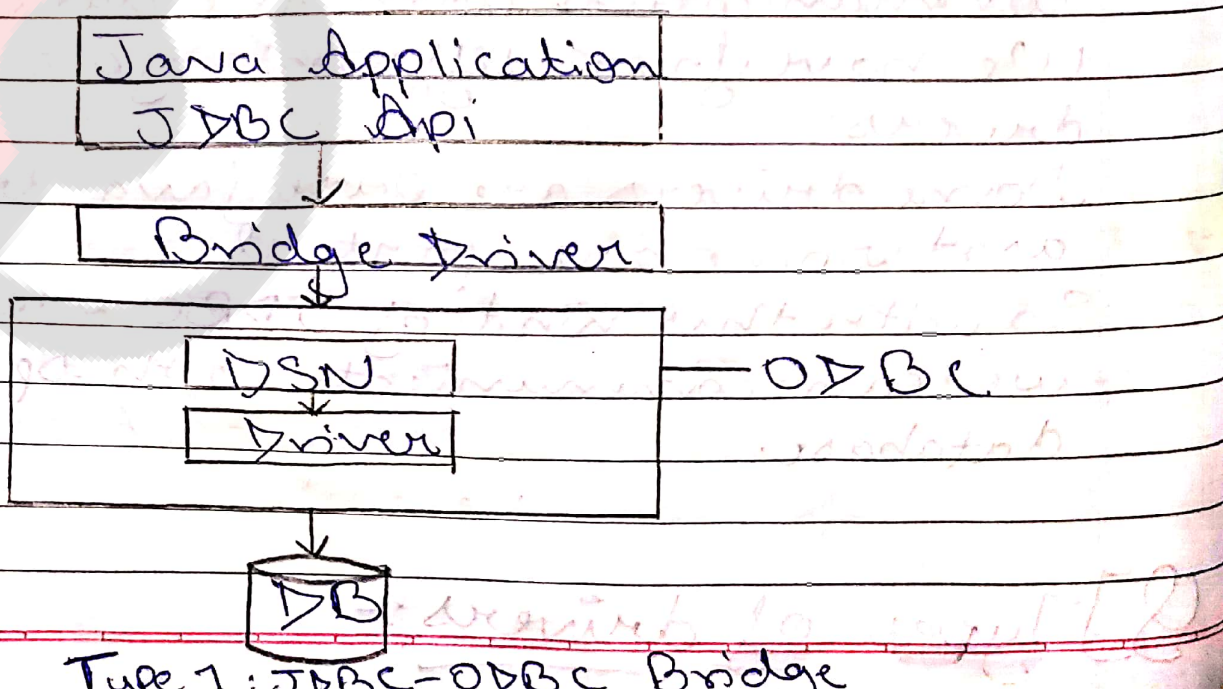
Q) Types of drivers:-

JDBC drivers are divided into four types or levels. The different types of jdbc drivers are:

Type 1 :- JDBC-ODBC Bridge driver (Bridge)
Type 2 :- Native-API/partly Java driver (Native)
Type 3 :- All Java/Net-protocol driver (Middleware)
Type 4 :- All Java/Native-protocol driver (Pure)

Type 1 JDBC Driver

JDBC-ODBC Bridge driver - The type 1 driver translates all JDBC calls into ODBC calls and sends them to the ODBC driver. ODBC is a generic API. The JDBC-ODBC Bridge driver is recommended only for experimental use or when no other alternative is available.



Type 1 : JDBC-ODBC Bridge

PAGE No.
DATE / /

Advantage

The JDBC ODBC Bridge allows access to almost any database, since the database's ODBC drivers are already available.

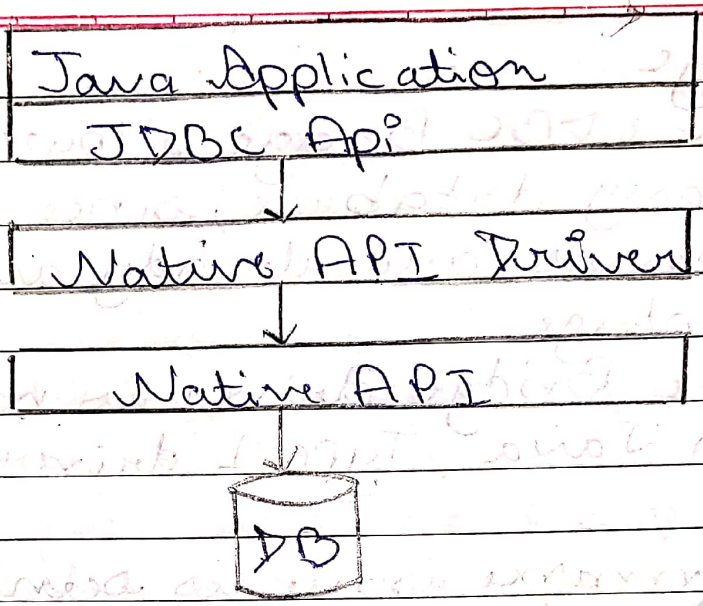
Disadvantage

1) Since the Bridge driver is not written fully in Java, Type 1 drivers are not possible
2) 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.
3) The client system requires the ODBC installation to use the driver.
4) Not good for the web.

Type 2 JDBC Driver

Native-API / Party Java driver

The distinctive characteristic of type 2 jdbc drivers are that Type 2 drivers convert JDBC calls into database-specific calls i.e this driver is specific to a particular database. Some distinctive characteristic of type 2 jdbc drivers are show below. Example: Oracle will have oracle native api.



Type 2: Native api/Partly Java Driver

Advantage

The distinctive characteristic of type 2 jdbc drivers are that they are typically offering better performance than the JDBC-ODBC Bridge as the layers of communication (tiers) are less than that of Type 1 and also it uses Native api which is Database specific.

Disadvantage

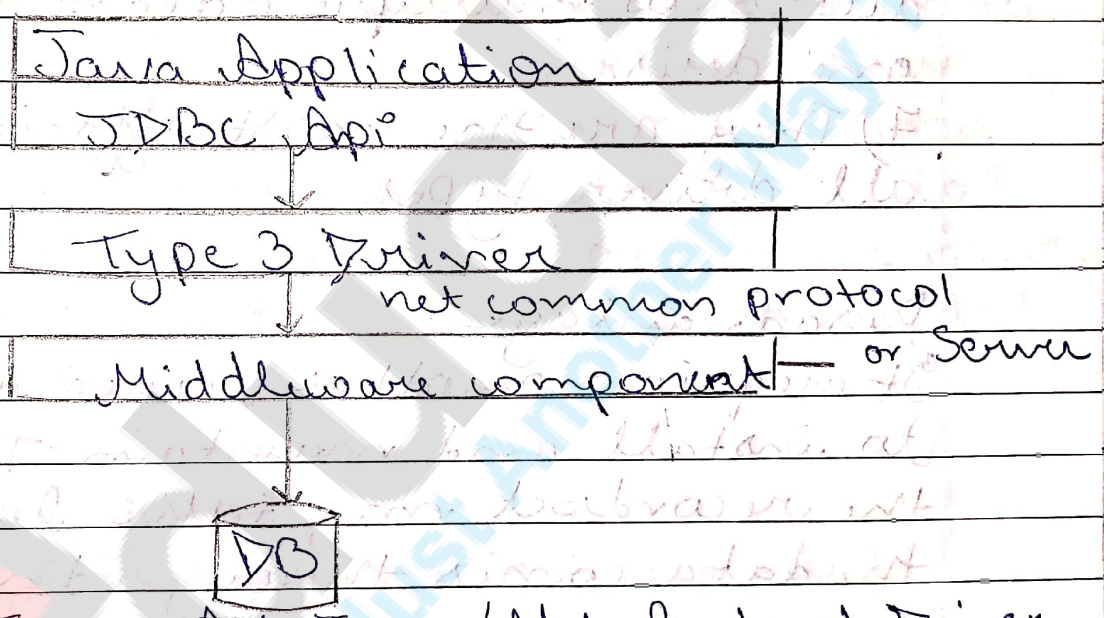
1) Native API must be installed in the Client System and hence type 2 drivers cannot be used for the internet.
2) Like Type 1 drivers, it's not written in Java language which forms a portability issue.
3) If we change the Database we have to change the native api as it is specific to a database.
4) Mostly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

5) Usually not thread safe.

Type 3 JDBC Driver

All Java/Net-protocol driver

Type 3 database requests are passed through the network to the middle-tier server. The middle-tier then translates the request to the database. If the middle-tier server can in turn use Type 1, Type 2 or Type 4 drivers.



Type 3: All Java / Net-Protocol Driver

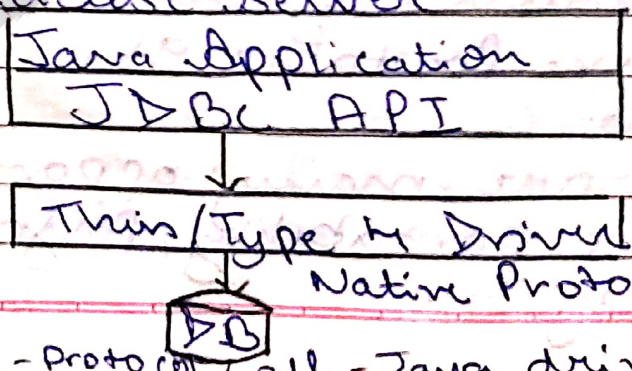
Advantage

1) This driver is server-based, so there is no need for any vendor database library to be present on client machines.
2) This driver is fully written in Java and hence Portable. It is suitable for the web.
3) There are many opportunities to optimize portability, performance and scalability.

4) The net protocol can be designed to make the client JDBC driver very small and fast to load.
5) The type 3 driver typically provides support for features such as caching (connections, query results, and so on), load balancing and advanced system administration such as logging and auditing.
6) This driver is very flexible allows access to multiple databases using one driver.
7) They are the most efficient amongst all driver types.

Disadvantage
It requires another server application to install and maintain. Traversing the recordset may take longer, since the data comes through the backend server.

Type 4 JDBC Driver
Native-protocol/all-Java driver
The Type 4 uses java networking libraries to communicate directly with the database server.



Type 4: Native-Protocol/all-Java driver

PAGE No.
DATE / /

Advantage

1) The major benefit of using a type 4 ~~jdbc~~ jdbc drivers are that they are completely written in Java to achieve platform independence and eliminate deployment administration issues. It is most suitable for the web.

2) Number of translation layers is very less i.e type 4 JDBC drivers dont have to translate database requests to ODBC or a native connectivity interface or to pass the request on to another server, performance is typically quite good.

3) You dont need to install special software on the client or server. Further, these drivers can be downloaded dynamically.

Disadvantage

With type 4 drivers, the user needs a different driver for each database.

Q) Java.sql package

This package includes classes and interface to perform almost all JDBC operation such as creating and executing SQL Queries.

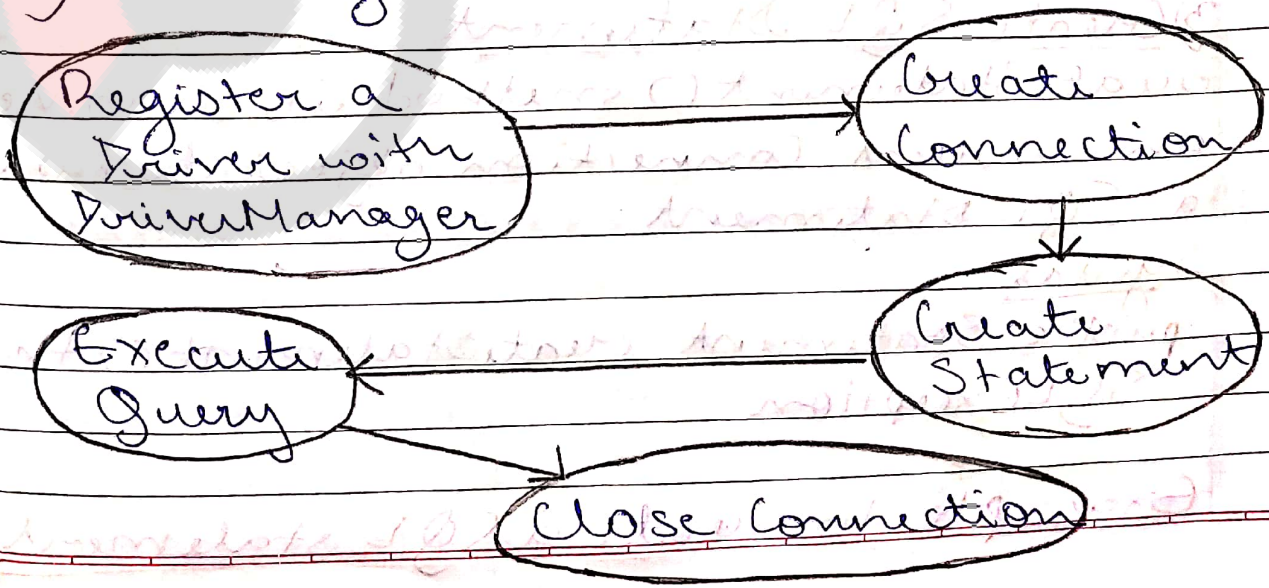
Important classes and interface of java.sql package

Classes / interface	Description
java.sql.BLOB	Provide support for BLOB (Binary Large Object) SQL type.
java.sql.Connection	creates a connection with specific database.
java.sql.CallableStatement	Execute stored procedures
java.sql.CLOB	Provide support for CLOB (Character Large Object) SQL type.
java.sql.Date	Provide support for Date SQL type.
java.sql.Driver	create an instance of a driver with DriverManager
java.sql.DriverManager	This class manages database drivers
java.sql.PreparedStatement	Used to create and execute parameterized query.
java.sql.ResultSet	It is an interface that provide methods to access the result

	row-by-row.
java.sql.Savepoint	Specify savepoint in transaction.
java.sql.SQLException	Encapsulate all JDBC related exception.
java.sql.Statement	This interface is used to execute SQL statement

Q] Establishing connectivity and working with connection interface / Steps to connect a Java Application to Database
The following 5 steps are the basic steps involved in connecting a Java application with Database using JDBC.

1) Register the Driver
2) Create a Connection
3) Create SQL Statement
4) Execute SQL Statement
5) Closing the connection



PAGE No.
DATE / /

1) Register the Driver
Class.forName() is used to load the driver class explicitly
Example to register with JDBC-ODBC Driver
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

2) Create a Connection
getConnection() method of DriverManager class is used to create a connection
Syntax
getConnection(String url)
getConnection(String url, String username, String password)
getConnection(String url, Properties info)

Example to establish connection with Oracle Driver
Connection con = DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:XE", "username", "password");

3) Create SQL Statement
createStatement() method is invoked on current Connection object to create a SQL statement.
Syntax
public Statement createStatement() throws SQLException

Example to create a SQL statement

Statement s = con.createStatement();

4) Execute SQL Statement

executeQuery() method of Statement interface is used to execute SQL statements.

Syntax

public ResultSet executeQuery(String query) throws SQLException

Example to execute a SQL statement

ResultSet rs = s.executeQuery("Select * from user");
while(rs.next())
{
System.out.println(rs.getString(1)+" "+rs.getString(2));
}

5) Closing the connection

After executing SQL statement you need to close the connection and release the session. The close() method of Connection interface is used to close the connection.

Syntax

public void close() throws SQLException

Example of closing a connection

con.close();

PAGE No.
DATE / /

7] Working with statement interface / Working with PreparedStatement interface

Statement interface in JDBC

Statement interface resides in java.sql package and it is used to execute a static SQL statement and returning the result of the executed query.
Statement interface has two sub-interfaces CallableStatement and PreparedStatement.

Statement Interface Methods

The Statement interface provides the following important methods:

Methods	Description
public boolean execute (String sql)	It executes the given SQL query, which may return multiple results.
public int executeBatch()	It submits the batch of commands to the database and returns an array of update counts.
public ResultSet executeQuery()	Executes the given SQL queries which returns the single ResultSet object.
public int executeUpdate (String sql)	It performs the execution of DML (insert, update or

	delete() statements.
public Connection getConnection()	It retrieves the connection object that produced the statement object.

Example: Performing select operation with Statement Intefface

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import.java.sql.Statement;
public class SelectTest{
   public static void main(String args[])throws
Exception{
  Class.forName("oracle.jdbc.OracleDriver");
  Connection con=DriverManager.getConnection
("jdbc:oracle:thin@localhost:1521:XE","scott",
"tiger");
Statement st=con.createStatement();
ResultSet rs=st.executeQuery("select*from
student");
   while(rs.next()!=false)
   {
  System.out.println(rs.getInt(1)+" "+rs.getS-
tring(2)+" "+rs.getString(3)+" "+rs.getString
(4));
   }
  rs.close();
  st.close();
  con.close();
  }
}
```

PAGE No.
DATE / /

PreparedStatement Interface

The PreparedStatement interface extends the Statement interface. It represents precompiled SQL statements and stores it in a PreparedStatement object.
It increases the performance of the application because the query is compiled only once.
The PreparedStatement is easy to reuse with new parameters.

Creating PreparedStatement Object

```
String sql = "Select * from Student where rollNO = ?";
PreparedStatement ps = con.prepareStatement(sql);
```

Note: All the parameter are represented by "?" symbol and each parameter is referred to by its origin position.

Example:- Insert operation with PreparedStatement Interface

```
import java.sql.*;
class PreparedStatDemo{
  public static void main(String args[])
  {
    try
    {
  Class.forName("oracle.jdbc.driver.OracleDriver");
  Connection con = DriverManager.getConnec-
```

```
tion ("jdbc:oracle:thin:@localhost:1521:XE"
,"username","password");
Prepared Statement ps = con.prepareStatement
("insert into Student values (?,?,?)");
   ps.setInt(1,101);
   ps.setString(2,"Surendra");
   ps.setString(3,"MCA");
   ps.executeUpdate();
   con.close();
  }
  catch (Exception e)
  {
     System.out.println(e);
  }
 }
}
```

CallableStatement Interface

The CallableStatement interface is used to execute the SQL stored procedure in a database. The JDBC API provides stored procedures to be called in a standard way for all RDBMS.

A stored procedure works like a function or method in a class. The stored procedure makes the performance better because these are precompiled queries.

Creating CallableStatement Interface

The instance of a CallableStatement is created by calling prepareCall() method on a Connection object.

For example:

```
CallableStatement callableStatement = con.
prepareCall("{call procedures(?,?)}");
```

Example: CallableStatement Interface using Stored procedure

Creating stored procedure

```
create or replace procedure "insertStudents"
(rollno IN NUMBER,
name IN VARCHAR2,
course IN VARCHAR2)
is
begin
insert into Student values(rollno, name,
course);
end;
/
```

```
// ProcedureDemo.java
import java.sql.*;
class ProcedureDemo {
  public static void main(String args[])
    {
      try
      {
        Class.forName("oracle.jdbc.driver.OracleDriver");
Connection con = DriverManager.getConnection
("jdbc:oracle:thin:@localhost:1521:xe","scott",
  "tiger");
CallableStatement = con.prepareCall("{call
```

```
insertStudents (?,?)");
  stmt.setInt(1,101);
  stmt.setString(2,Vinod);
  stmt.setString(3,BE);
  stmt.execute();
System.out.println("Record inserted successfully");
  con.close();
  stmt.close();
}
catch (Exception e)
  {
   e.printStackTrace();
  }
 }
}
```

Note :- The ProcedureDemo.java file inserts the record in Students table in Oracle database by use of stored procedure.

Working with ResultSet interface/ Working with ResultSetMetaData interface

ResultSet Interface

The result of the query after execution of database statement is returned as table of data according to rows and columns. This data is accessed using the ResultSet interface.

A default ResultSet object is not upd-

atable and the cursor moves only in forward direction.

Creating ResultSet Interface

To execute a Statement or PreparedStatement, we create ResultSet object:

Example

```
Statement stmt = connection.createStatement();
ResultSet result = stmt.executeQuery
  ("select * from students");
or
String sql = "select * from students";
PreparedStatement stmt = con.prepareStatement(sql);
ResultSet result = stmt.executeQuery();
```

ResultSet Interface Methods

Methods	Description
public boolean absolute (int row)	Moves the cursor to the specified row in the ResultSet object.
public void beforeFirst()	It moves the cursor just before the first row i.e front of the ResultSet
public void afterLast()	Moves the cursor to the end of the ResultSet object, just after the

	last row.
public boolean first()	Moves the cursor to first value of ResultSet object.
public boolean last()	Moves the cursor to the last row of the ResultSet object.
public boolean previous()	Just moves the cursor to the previous row in the ResultSet object.
public boolean next()	It moves the cursor forward one row from its current position.
public int getInt(int columnIndex)	It retrieves the value of the column in current row as int is given ResultSet object.
public String getString(int columnIndex)	It retrieves the value of the column in current row as int in given ResultSet object.

public void relative (int rows)	It moves the cursor to a relative number of rows.

Types of ResultSet Interface

1) ResultSet.TYPE_FORWARD_ONLY
The ResultSet can only be navigating forward.

2) ResultSet.TYPE_SCROLL_INSENSITIVE
The ResultSet can be navigated both in forward and backward direction. It can also jump from current position to another position. The ResultSet is not sensitive to change made by others.

3) ResultSet.TYPE_SCROLL_SENSITIVE
The ResultSet can be navigated in both forward and backward direction. It can also jump from current position to another position. The ResultSet is sensitive to change made by others to the database.

Example: Program to illustrate ResultSet interface with Scrollable

```
import java.sql.*;
class ResultSetTest{
public static void main (String args[])
{
    Connection con=null;
    Statement stmt=null;
    ResultSet rs=null;
```

```
try {
Class.forName("oracle.jdbc.driver.OracleDriver");
con=DriverManager.getConnection("jdbc:oracle:
thin:@localhost:1521:xe","scott","tiger");
stmt=con.createStatement(ResultSet.TYPE_SCROLL_
SENSITIVE,ResultSet.CONCUR_UPDATABLE);
rs=stmt.executeQuery("Select * from Student");
rs.absolute(5);
System.out.println(rs.getInt(1)+" "+rs.getStri
ng(2)+" "+rs.getString(3));
  rs.close();
  stmt.close();
  con.close();
 }
 catch(Exception e)
 {
  e.printStackTrace();
 }
 }
}
```

ResultSetMetaData Interface

ResultSetMetaData is an interface which provides information about a result set that is returned by an executeQuery() method. The ResultSetMetaData extends the wrapper interface.

The interface provides the metadata about the database. It includes the information about the names of the columns, number of columns etc.

PAGE No.
DATE / /

ResultSetMetaData Interface Methods

Methods	Description
public String getColumnClassName(int column)	It returns the name of the Java class whose instances are created.
public int getColumnCount()	It returns the number of columns in the ResultSet object.
public String getColumnName(int column)	Returns the column name from the ResultSet object.
public int getColumnType(int column)	It retrieves the column's which is designated in SQL
public String getSchemaName(int column)	Return the table's schema which designed with column.
public String getTableName(int column)	Returns the designed SQL table's name.

Example: Illustrating the ResultSetMetaData Interface

```
//RSMTest.java

import java.sql.*;
class RSMTest{
```

```
public static void main (String args[])
{
  try
  {
  Class.forName ("oracle.jdbc.driver.OracleDriver");
Connection con = DriverManager.getConnection (
"jdbc:oracle:thin:@localhost:1521:xe", "scott", "tiger");
PreparedStatement ps = con.prepareStatement ("
select * from Students");
ResultSet rs = ps.executeQuery();
ResultSetMetaData rsmd = rs.getMetaData();
for (int i=1; i<=cols; i++)
{
String colName = rsmd.getColumnName(i);
String colType = rsmd.getColumnTypeName(i);
System.out.println (colName + " of type" +
colType);
  }
  rs.close();
  ps.close();
  con.close();
  }
  catch (Exception e)
  {
  e.printStackTrace();
  }
  }
}
```