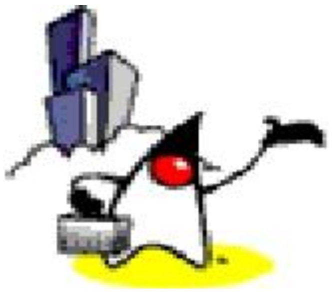
JDBC

Contents

What is JDBC?



What is JDBC?

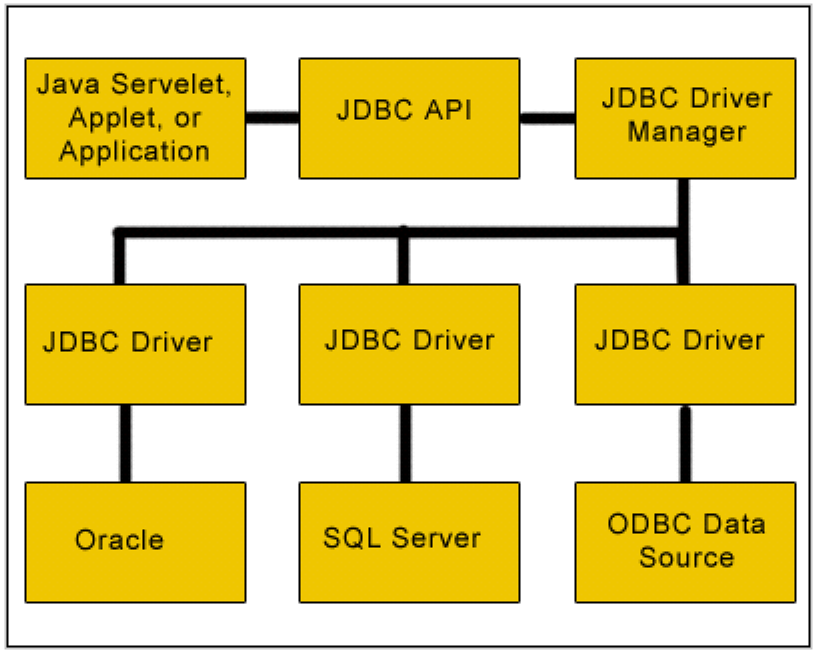
- Standard Java API for accessing relational database
  - Hides database specific details from application
- Part of Java SE (J2SE)

JDBC API

- Defines a set of Java Interfaces, which are implemented by vendor-specific JDBC Drivers
  - Applications use this set of Java interfaces for performing database operations - portability
- Majority of JDBC API is located in java.sql package
  - DriverManager, Connection, ResultSet, DatabaseMetaData, ResultSetMetaData, PreparedStatement, CallableStatement and Types
- Other advanced functionality exists in the javax.sql package
  - DataSource

JDBC API

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



JDBC Driver Manager

- DriverManager is the backbone of the JDBC architecture. It is quite small and simple.
- Main purpose is managing the different types of JDBC database driver.
  - On running an application, load all the drivers found in the system property jdbc. drivers.
  - When opening a connection to a database, choose the most appropriate driver from the previously loaded drivers.

JDBC Driver

- Database specific implemention of JDBC interfaces
  - Every database server has corresponding JDBC driver(s)
  - A JDBC driver provides JDBC applications with database independence.
    - If the back-end database changes, only the JDBC driver need be replaced with few code modifications required.
- the list of available drivers:
  - http://www.java2s.com/Tutorial/Java/0340__Database/AListofJDBCDriversconnectionstringdrivername.htm

Database URL

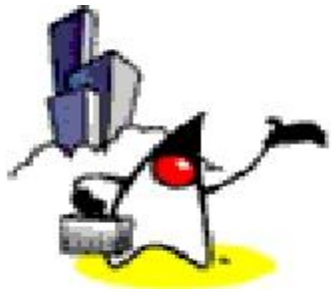
- Used to make a connection to the database
  - Can contain server, port, protocol etc…
- jdbc:subprotocol_name:driver_dependant_databasename
  - Oracle thin driver

    jdbc:oracle:thin:@machinename:1521:dbname
  - Derby

    jdbc:derby://localhost:1527/sample
  - Pointbase

    jdbc:pointbase:server://localhost/sample

Step By Step Usage of JDBC



Steps of Using JDBC

1. Load DB-specific JDBC driver
2. Get a Connection object
3. Get a Statement object
4. Execute queries and/or updates
5. Read results
6. Read Meta-data (optional step)
7. Close Statement and Connection objects

1. Load DB-Specific Database Driver

- To manually load the database driver and register it with the DriverManager, load its class file

    Class.forName(<database-driver>)

```
try {
        // This loads an instance of the Pointbase DB Driver.
        // The driver has to be in the classpath.
        Class.forName("org.apache.derby.jdbc.ClientDriver");

        }catch (ClassNotFoundException cnfe){
                System.out.println("" + cnfe);
      }
```

2. Get a Connection Object

- DriverManager class is responsible for selecting the database and creating the database connection
- Create the database connection as follows:

```
try {
      Connection connection =
    DriverManager.getConnection("jdbc:derby://localhost:1527/sample", “app"," app ");
} catch(SQLException sqle) {
      System.out.println("" + sqle);
}
```

DriverManager & Connection

- java.sql.DriverManager
  - getConnection(String url, String user, String password) throws SQLException
- java.sql.Connection
  - Statement createStatement() throws SQLException
  - void close() throws SQLException
  - void setAutoCommit(boolean b) throws SQLException
  - void commit() throws SQLException
  - void rollback() throws SQLException

3. Get a Statement Object

- Create a Statement Object from Connection object
  - java.sql.Statement
    - ResultSet executeQuery(string sql)
    - int executeUpdate(String sql)
  - Example:
    - Statement statement = connection.createStatement();
- The same Statement object can be used for many, unrelated queries

4. Executing Query or Update

- From the Statement object, the 2 most used commands are
  - (a) QUERY (SELECT)
    - ResultSet rs = statement.executeQuery("select * from customer_tbl");
  - (b) ACTION COMMAND (UPDATE/DELETE)
    - int iReturnValue = statement.executeUpdate("update manufacture_tbl set name = ‘IBM' where mfr_num = 19985678");

5. Reading Results

- Loop through ResultSet retrieving information
  - java.sql.ResultSet
    - boolean next()
    - xxx getXxx(int columnNumber)
    - xxx getXxx(String columnName)
    - void close()
- The iterator is initialized to a position before the first row
  - You must call next() once to move it to the first row

5. Reading Results (Continued)

- Once you have the ResultSet, you can easily retrieve the data by looping through it

```
while (rs.next()){
   // Wrong this will generate an error
   String value0 = rs.getString(0);

   // Correct!
   String value1 = rs.getString(1);
   int    value2 = rs.getInt(2);
   int    value3 = rs.getInt(“ADDR_LN1");
}
```

5. Reading Results (Continued)

- When retrieving data from the ResultSet, use the appropriate getXXX() method
  - getString()
  - getInt()
  - getDouble()
  - getObject()
- There is an appropriate getXXX method of each java.sql.Types datatype

6. Read ResultSet MetaData and DatabaseMetaData (Optional)

- Once you have the ResultSet or Connection objects, you can obtain the Meta Data about the database or the query
- This gives valuable information about the data that you are retrieving or the database that you are using
  - ResultSetMetaData rsMeta = rs.getMetaData();
  - DatabaseMetaData dbmetadata = connection.getMetaData();
    - There are approximately 150 methods in the DatabaseMetaData class.

ResultSetMetaData Example

```
ResultSetMetaData meta = rs.getMetaData();
//Return the column count
int iColumnCount = meta.getColumnCount();

for (int i =1 ; i <= iColumnCount ; i++){
  System.out.println(“Column Name: " + meta.getColumnName(i));
  System.out.println(“Column Type" + meta.getColumnType(i));
  System.out.println("Display Size: " +
     meta.getColumnDisplaySize(i) );
}
```

Examples

- **Connecting to a MySQL Database**
  - **MySQLConnect.java**
- **Creating a Database**
  - **CreateDatabase.java**
- **Creating a Database Table**
  - **CreateTable.java**
- **Deleting a Table from Database:**
  - **DeleteTable.java**
- **Retrieving Tables from a Database**
  - **A**llTableName.java

Examples

- **Inserting values in MySQL database table**
  - InsertValues.java
- **Retrieving All Rows from a Database Table**
  - **GetAllRows.java**
- **Getting Column Names from a database table in Java**
  - **ColumnName.java**
- **Arrange a Column of Database Table**
  - **ColumnDescOrder.java**

Prepared Statements

PreparedStatement

- Sometimes it is more convenient to use a PreparedStatement object for sending SQL statements to the database.
- The contained SQL is sent to the database and compiled or prepared beforehand
- unlike a Statement object, it is given an SQL statement when it is created.
- Prepared statements can take parameters, you can use the same statement and supply it with different values each time you execute it.

PreparedStatement Steps

1. You register the drive and create the db connection in the usual manner
2. Once you have a db connection, create the prepared statement object

```
PreparedStatement updateSales =
        con.prepareStatement("UPDATE OFFER_TBL SET
        QUANTITY = ? WHERE ORDER_NUM = ? ");
// "?" are referred to as Parameter Markers
// Parameter Markers are referred to by number,
//  starting from 1, in left to right order.
// PreparedStatement's setXXX() methods are used to
        set
//  the IN parameters, which remain set until changed.
```

PreparedStatement Steps cont.

3. Bind in your variables. The binding in of variables is positional based

```
updateSales.setInt(1, 75);
updateSales.setInt(2, 10398001);
```

4. Once all the vairables have been bound, then you execute the prepared statement

```
int iUpdatedRecords = updateSales.executeUpdate();
```

Comparision

- Code Fragment 1:

```
String updateString = "UPDATE COFFEES SET SALES = 75 " +
    "WHERE COF_NAME LIKE 'Colombian'";
stmt.executeUpdate(updateString);
```

- Code Fragment 2:

```
PreparedStatement updateSales = con.prepareStatement(
    "UPDATE COFFEES SET SALES = ? WHERE COF_NAME LIKE ? ");
updateSales.setInt(1, 75);
updateSales.setString(2, "Colombian");
updateSales.executeUpdate():
```

Using a Loop to Set Values

```
PreparedStatement updateSales;
String updateString = "update COFFEES " + "set SALES = ? where COF_NAME
    like ?";
updateSales = con.prepareStatement(updateString);
int [] salesForWeek = {175, 150, 60, 155, 90};
String [] coffees = {"Colombian", "French_Roast", "Espresso",
    "Colombian_Decaf", "French_Roast_Decaf"};
int len = coffees.length;
for(int i = 0; i < len; i++) {
    updateSales.setInt(1, salesForWeek[i]);
     updateSales.setString(2, coffees[i]);
     updateSales.executeUpdate();
}
```

PreparedStatement cont.

- If the prepared statement object is a select statement, then you execute it, and loop through the result set object the same as in the Basic JDBC example:

```
PreparedStatement itemsSold =
   con.prepareStatement("select o.order_num,
   o.customer_num, c.name, o.quantity from order_tbl o,
   customer_tbl c where o.customer_num =
   c.customer_num and o.customer_num = ?;");
itemsSold.setInt(1,10398001);
ResultSet rsItemsSold = itemsSold.executeQuery();
while (rsItemsSold.next()){
   System.out.println( rsItemsSold.getString(“NAME") + "
   sold "+ rsItemsSold.getString(“QUANTITY") + " unit(s)");
}
```

Transaction

Transaction

- A transaction is a set of one or more statements that are executed together as a unit, so either all of the statements are executed, or none of the statements is executed.
- When a connection is created, it is in auto-commit mode.
  - each individual SQL statement is treated as a transaction and is automatically committed right after it is executed.

Transaction

- The way to group two or more statements into a transaction is to disable auto-commit mode.

```
con.setAutoCommit(false);
```

- Once auto-commit mode is disabled, no SQL statements are committed until you call the method commit explicitly.
- The entire transaction can be rolled back.

JDBC Transaction Methods

- setAutoCommit()
  - If set true, every executed statement is committed immediately
- commit()
  - Relevant only if setAutoCommit(false)
  - Commit operations performed since the opening of a Connection or last commit() or rollback() calls
- rollback()
  - Relevant only if setAutoCommit(false)
  - Cancels all operations performed

Transactions Example

```
Connection connection = null;
    try {
       connection = 
    DriverManager.getConnection("jdbc:oracle:thin:@machinename
    :1521:dbname","username","password");
     connection.setAutoCommit(false);

    PreparedStatement updateQty = 
    connection.prepareStatement("UPDATE STORE_SALES SET 
    QTY = ? WHERE ITEM_CODE = ? ");
```

Transaction Example cont.

```
int [][] arrValueToUpdate =
{ {123, 500} ,
  {124, 250},
  {125, 10},
  {126, 350} };

  int iRecordsUpdate = 0;
  for ( int items=0 ; items < arrValueToUpdate.length ;
items++) {
        int itemCode = arrValueToUpdate[items][0];
        int qty = arrValueToUpdate[items][1];
```

Transaction Example cont.

```
            updateQty.setInt(1,qty);
            updateQty.setInt(2,itemCode);
            iRecordsUpdate += updateQty.executeUpdate();
        }
        connection.commit();
        System.out.println(iRecordsUpdate + " record(s) have been
updated");
    } catch(SQLException sqle) {
        System.out.println("" + sqle);
```

Transaction Example cont.

```
 try {
          connection.rollback();
} catch(SQLException sqleRollback) {
          System.out.println("" + sqleRollback);
           }
}
finally {
         try {
                connection.close();
           }
           catch(SQLException sqleClose) {
               System.out.println("" + sqleClose);
           }
       }
```

Rolling Back to a Savepoint

- The JDBC 3.0 API adds the method Connection.setSavepoint, which sets a savepoint within the current transaction.

```
Statement stmt = conn.createStatement();
int rows = stmt.executeUpdate("INSERT INTO TAB1 (COL1) VALUES " +
    "(?FIRST?)");
// set savepoint
Savepoint svpt1 = conn.setSavepoint("SAVEPOINT_1");
 rows = stmt.executeUpdate("INSERT INTO TAB1 (COL1) " + "VALUES
    (?SECOND?)");
...
conn.rollback(svpt1);
...
conn.commit();
```

Releasing a Savepoint

- Any savepoints created in a transaction are automatically released and become invalid when
  - the transaction is committed
  - the entire transaction is rolled back.
    - Rolling a transaction back to a savepoint automatically releases and makes invalid any other savepoints that were created after the savepoint in question.
- Releasing a Savepoint
  - void **releaseSavepoint**(Savepoint savepoint)
  - The method Connection.releaseSavepoint removes a Savepoint from the current transaction.
- Once a savepoint has been released, attempting to reference it in a rollback operation causes an SQLException to be thrown.

Join

Join tables in the specific database

- Sometimes you need to use two or more tables to get the data you want.
- A join is a database operation that relates two or more tables by means of values that they share in common.
- Joining is the type of query for retrieving data from two or more tables in specific database.
- Types to join the tables: Natural join, Natural left join, Natural right join and so on.

Join

- **Join:** A join provides the facility to connect two tables are merged to each other according to field that is common and creates a new virtual table.
- **Natural Join:** It is a type of join that retrieves data within specified tables to specific field is matched.
- **Natural Left Join:** In this operation both tables are merged to each other according to common fields but the priority is given to the first table in database.
- **Natural Right Join:** This operation join tables on the basis of matching fields but priority will be given to the right table in database.

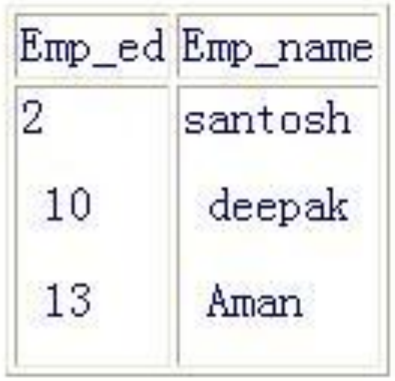
Natural Join

- **Description of program:**
  - the **NATURAL JOIN** operation is performed within two tables: employee and Emp_sal. The employee table holds the Emp_ed and Emp_name fields and Emp_sal table contains the Emp_name and Emp_sal. We are making use of the emp_name to join the tables.
- **Description of code:**

  SELECT *FROM employee NATURAL JOIN Emp_sal
- **NatJoinTable.java**

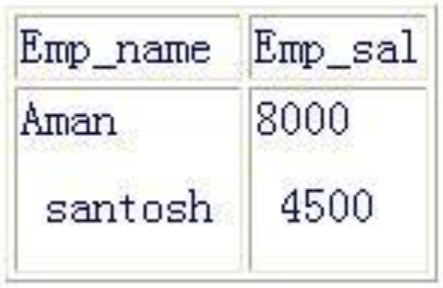
Result

Table:- employee:



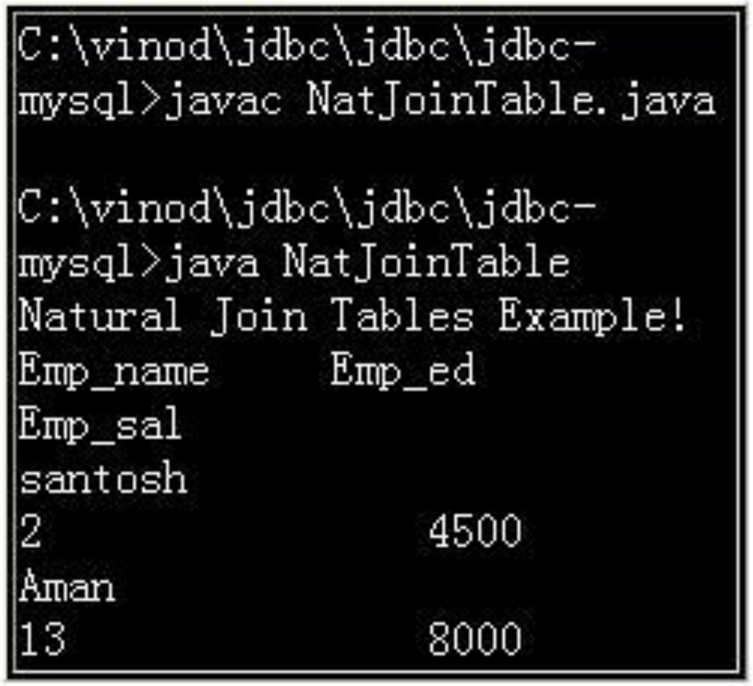
Emp_ed	Emp_name
2	santosh
10	deepak
13	Aman

Table:- Emp_sal:



Emp_name	Emp_sal
Aman	8000
santosh	4500

Output of program:



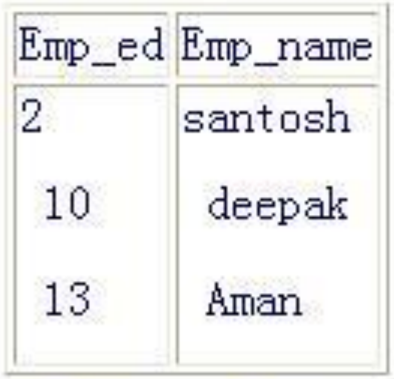
```
C:\vinod\jdbc\jdbc\jdbc-
mysql>javac NatJoinTable.java

C:\vinod\jdbc\jdbc\jdbc-
mysql>java NatJoinTable
Natural Join Tables Example!
Emp_name     Emp_ed
Emp_sal
santosh
2                4500
Aman
13               8000
```

Natural Left Join

SELECT *FROM employee NATURAL LEFT JOIN Emp_sal

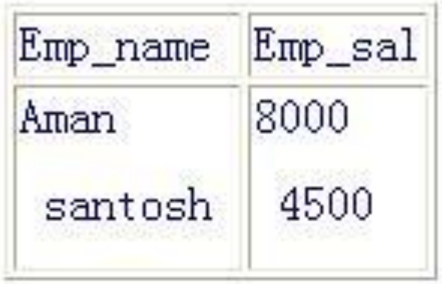
- NatLeftJoinTable.java

Result



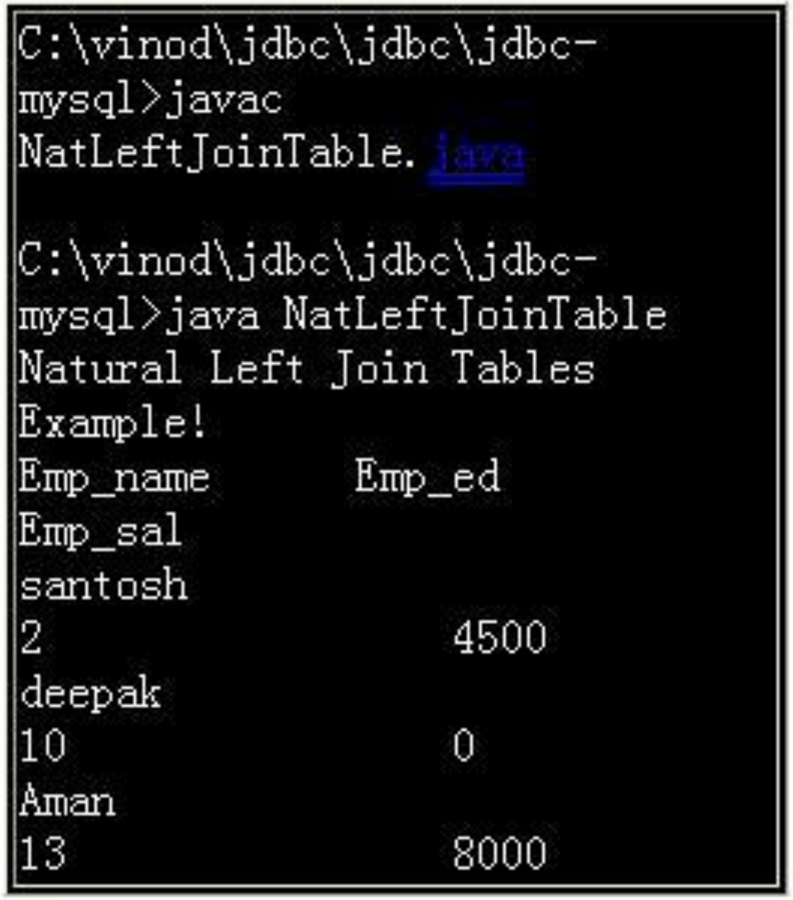
Emp_ed	Emp_name
2	santosh
10	deepak
13	Aman

Table:- Emp_sal:



Emp_name	Emp_sal
Aman	8000
santosh	4500

Output of program:



```
C:\vinod\jdbc\jdbc\jdbc-
mysql>javac
NatLeftJoinTable.java

C:\vinod\jdbc\jdbc\jdbc-
mysql>java NatLeftJoinTable
Natural Left Join Tables
Example!
Emp_name      Emp_ed
Emp_sal
santosh
2                      4500
deepak
10                     0
Aman
13                     8000
```

Natural Right Join

SELECT *FROM employee NATURAL RIGHT JOIN Emp_sal

- NatRightJoinTable.java

Result

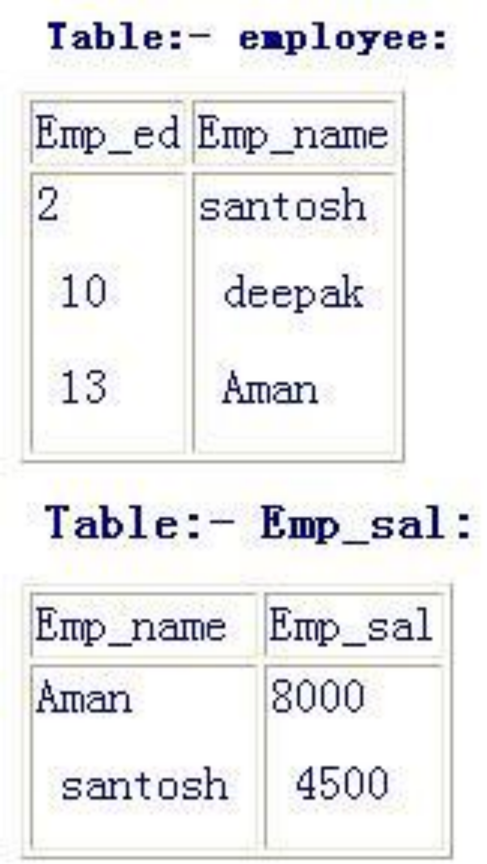
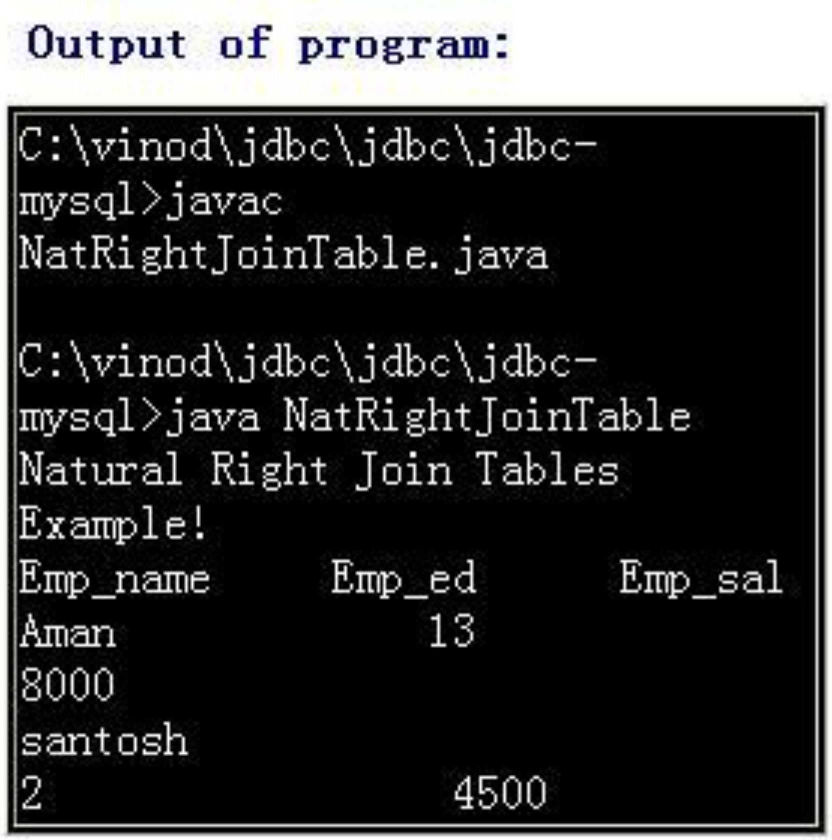


Table:- employee:

Emp_ed	Emp_name
2	santosh
10	deepak
13	Aman

Table:- Emp_sal:

Emp_name	Emp_sal
Aman	8000
santosh	4500



Output of program:

```
C:\vinod\jdbc\jdbc\jdbc-
mysql>javac
NatRightJoinTable.java

C:\vinod\jdbc\jdbc\jdbc-
mysql>java NatRightJoinTable
Natural Right Join Tables
Example!
Emp_name     Emp_ed      Emp_sal
Aman            13
8000
santosh
2                 4500
```

The End!