JDBC - Introduction

What is JDBC?

JDBC stands for **J**ava **D**ata**b**ase **C**onnectivity, which is a standard Java API for database-independent connectivity between the Java programming language and a wide range of databases.

The JDBC library includes APIs for each of the tasks mentioned below that are commonly associated with database usage.

- Making a connection to a database.
- Creating SQL or MySQL statements.
- Executing SQL or MySQL queries in the database.
- Viewing & Modifying the resulting records.

Fundamentally, JDBC is a specification that provides a complete set of interfaces that allows for portable access to an underlying database. Java can be used to write different types of executables, such as −

- Java Applications
- Java Applets
- Java Servlets
- Java ServerPages (JSPs)
- Enterprise JavaBeans (EJBs).

All of these different executables are able to use a JDBC driver to access a database, and take advantage of the stored data.

JDBC provides the same capabilities as ODBC, allowing Java programs to contain database-independent code.

Pre-Requisite

Before moving further, you need to have a good understanding of the following two subjects −

- Core JAVA Programming
- SQL or MySQL Database

JDBC Architecture

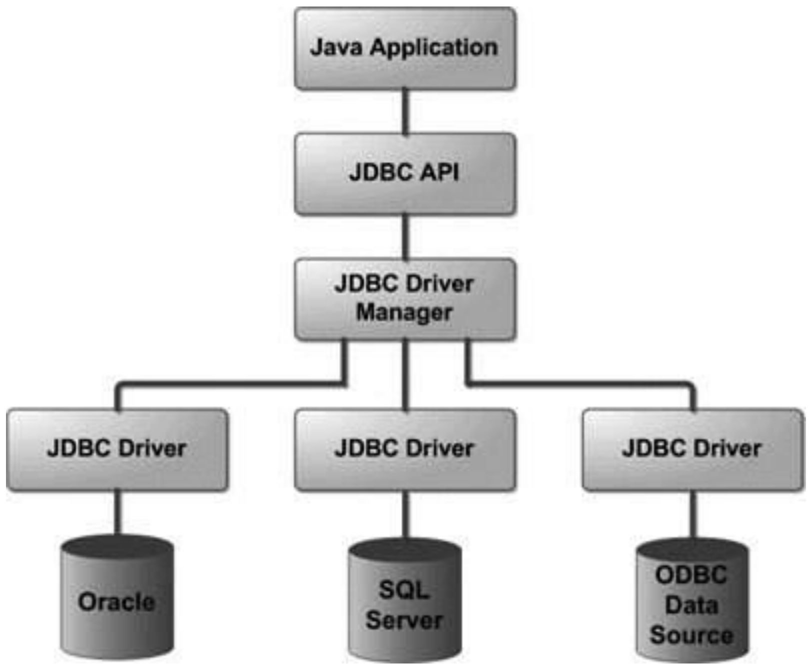
The JDBC API supports both two-tier and three-tier processing models for database access but in general, JDBC Architecture consists of two layers −

- **JDBC API** − This provides the application-to-JDBC Manager connection.
- **JDBC Driver API** − This supports the JDBC Manager-to-Driver Connection.

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

The JDBC driver manager ensures that the correct driver is used to access each data source. The driver manager is capable of supporting multiple concurrent drivers connected to multiple heterogeneous databases.

Following is the architectural diagram, which shows the location of the driver manager with respect to the JDBC drivers and the Java application −



Common JDBC Components

The JDBC API provides the following interfaces and classes −

- **DriverManager** − This class manages a list of database drivers. Matches connection requests from the java application with the proper database driver using communication sub protocol. The first driver that recognizes a certain subprotocol under JDBC will be used to establish a database Connection.
- **Driver** − This interface handles the communications with the database server. You will interact directly with Driver objects very rarely. Instead, you use DriverManager objects, which manages objects of this type. It also abstracts the details associated with working with Driver objects.
- **Connection** − This interface with all methods for contacting a database. The connection object represents communication context, i.e., all communication with database is through connection object only.
- **Statement** − You use objects created from this interface to submit the SQL statements to the database. Some derived interfaces accept parameters in addition to executing stored procedures.
- **ResultSet** − These objects hold data retrieved from a database after you execute an SQL query using Statement objects. It acts as an iterator to allow you to move through its data.
- **SQLException** − This class handles any errors that occur in a database application.

The JDBC 4.0 Packages

The java.sql and javax.sql are the primary packages for JDBC 4.0. This is the latest JDBC version at the time of writing this tutorial. It offers the main classes for interacting with your data sources.

The new features in these packages include changes in the following areas −

- Automatic database driver loading.
- Exception handling improvements.
- Enhanced BLOB/CLOB functionality.
- Connection and statement interface enhancements.
- National character set support.
- SQL ROWID access.
- SQL 2003 XML data type support.
- Annotations.

JDBC - SQL Syntax

Structured **Q**uery **L**anguage (SQL) is a standardized language that allows you to perform operations on a database, such as creating entries, reading content, updating content, and deleting entries.

SQL is supported by almost any database you will likely use, and it allows you to write database code independently of the underlying database.

This chapter gives an overview of SQL, which is a prerequisite to understand JDBC concepts. After going through this chapter, you will be able to Create, **C**reate, **R**ead, **U**pdate, and **D**elete (often referred to as **CRUD** operations) data from a database.

For a detailed understanding on SQL, you can read our MySQL Tutorial.

Create 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.

Sample Code

Copy and paste the following example in JDBCExample.java, compile and run as follows −

```
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();
      }
   }
}
```

Now let us compile the above example as follows −

```
C:\>javac JDBCExample.java
C:\>
```

When you run **JDBCExample**, it produces the following result −

```
C:\>java JDBCExample
Database created successfully...
C:\>
```

The CREATE DATABASE statement is used for creating a new database. The syntax is −

```
SQL> CREATE DATABASE DATABASE_NAME;
```

Example

The following SQL statement creates a Database named EMP −

```
SQL> CREATE DATABASE EMP;
```

Drop Database

The DROP DATABASE statement is used for deleting an existing database. The syntax is −

```
SQL> DROP DATABASE DATABASE_NAME;
```

Note − To create or drop a database you should have administrator privilege on your database server. Be careful, deleting a database would loss all the data stored in the database.

Create Table

The CREATE TABLE statement is used for creating a new table. The syntax is −

```
SQL> CREATE TABLE table_name
(
   column_name column_data_type,
   column_name column_data_type,
   column_name column_data_type
   ...
);
```

Example

The following SQL statement creates a table named Employees with four columns −

```
SQL> CREATE TABLE Employees
(
  id INT NOT NULL,
  age INT NOT NULL,
  first VARCHAR(255),
  last VARCHAR(255),
  PRIMARY KEY ( id )
);
```

Drop Table

The DROP TABLE statement is used for deleting an existing table. The syntax is −

SQL> DROP TABLE table_name;

Example

The following SQL statement deletes a table named Employees −

```
SQL> DROP TABLE Employees;
```

JDBC - Select Database Example

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 the 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 a **selected** database.
  Selection of database is made while you prepare database URL. Following example would make connection with **STUDENTS** database.
- **Clean up the environment** − try with resources automatically closes the resources.

Sample Code

Copy and paste the following example in JDBCExample.java, compile and run as follows −

```
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/TUTORIALSPOINT";
   static final String USER = "guest";
   static final String PASS = "guest123";

   public static void main(String[] args) {
      System.out.println("Connecting to a selected database...");
      // Open a connection
      try(Connection conn = DriverManager.getConnection(DB_URL, USER, PASS);) {
         System.out.println("Connected database successfully...");
      } catch (SQLException e) {
         e.printStackTrace();
      }
   }
}
```

Now let us compile the above example as follows −

```
C:\>javac JDBCExample.java
C:\>
```

When you run **JDBCExample**, it produces the following result −

```
C:\>java JDBCExample
Connecting to a selected database...
Connected database successfully...
C:\>
```

JDBC - Drop Database Example

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 a database server. Deleting a database does not require database name to be in your database URL. Following example would delete **STUDENTS** database.
- **Execute a query** − Requires using an object of type Statement for building and submitting an SQL statement to delete the database.
- **Clean up the environment** − try with resources automatically closes the resources.

Sample Code

Copy and paste the following example in JDBCExample.java, compile and run as follows −

```
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 = "DROP DATABASE STUDENTS";
      stmt.executeUpdate(sql);
      System.out.println("Database dropped successfully...");
    } catch (SQLException e) {
      e.printStackTrace();
    }
  }
}
```

Now let us compile the above example as follows −

```
C:\>javac JDBCExample.java
C:\>
```

INSERT Data

The syntax for INSERT, looks similar to the following, where column1, column2, and so on represents the new data to appear in the respective columns −

```
SQL> INSERT INTO table_name VALUES (column1, column2, ...);
```

Example

The following SQL INSERT statement inserts a new row in the Employees database created earlier −

```
SQL> INSERT INTO Employees VALUES (100, 18, 'Zara', 'Ali');
```

SELECT Data

The SELECT statement is used to retrieve data from a database. The syntax for SELECT is −

```
SQL> SELECT column_name, column_name, ...
    FROM table_name
    WHERE conditions;
```

The WHERE clause can use the comparison operators such as =, !=, <, >, <=,and >=, as well as the BETWEEN and LIKE operators.

Example

The following SQL statement selects the age, first and last columns from the Employees table, where id column is 100 −

```
SQL> SELECT first, last, age
    FROM Employees
    WHERE id = 100;
```

The following SQL statement selects the age, first and last columns from the Employees table where *first* column contains *Zara* −

```
SQL> SELECT first, last, age
    FROM Employees
    WHERE first LIKE '%Zara%';
```

UPDATE Data

The UPDATE statement is used to update data. The syntax for UPDATE is −

```
SQL> UPDATE table_name
    SET column_name = value, column_name = value, ...
    WHERE conditions;
```

The WHERE clause can use the comparison operators such as =, !=, <, >, <=,and >=, as well as the BETWEEN and LIKE operators.

Example

The following SQL UPDATE statement changes the age column of the employee whose id is 100 −

```
SQL> UPDATE Employees SET age=20 WHERE id=100;
```

DELETE Data

The DELETE statement is used to delete data from tables. The syntax for DELETE is −

```
SQL> DELETE FROM table_name WHERE conditions;
```

The WHERE clause can use the comparison operators such as =, !=, <, >, <=,and >=, as well as the BETWEEN and LIKE operators.

Example

The following SQL DELETE statement deletes the record of the employee whose id is 100 −

```
SQL> DELETE FROM Employees WHERE id=100;
```

Instant query() Method in Java with Examples

query() method of an **Instant** class used to query this instant using the specified query as parameter.The TemporalQuery object passed as parameter define the logic to be used to obtain the result from this instant.

Syntax:

```
public <R> R query(TemporalQuery<R> query)
```

Parameters: This method accepts only one parameter **query** which is the query to invoke.

Return value: This method returns the query result, null may be returned.

Exception: This method throws following Exceptions:

- **DateTimeException** – if unable to query .
- **ArithmeticException** – if numeric overflow occurs.

Below programs illustrate the query() method:

Program 1:

```
// Java program to demonstrate
// Instant.query() method

import java.time.*;
import java.time.temporal.*;

public class GFG {
    public static void main(String[] args)
    {

        // create Instant object
        Instant instant
            = Instant.parse("2018-12-31T10:15:30.00Z");

        // apply query method of Instant class
        String value
            = instant.query(TemporalQueries.precision())
                  .toString();
        // print the result
        System.out.println("Precision value for Instant is "
                           + value);
    }
}
```

Output:

```
Precision value for Instant is Nanos
```

Program 2: Showing if query did not found the required object then it returns null.

```
// Java program to demonstrate
// Instant.query() method

import java.time.*;
import java.time.temporal.*;

public class GFG {
    public static void main(String[] args)
    {

        // create Instant object
        Instant instant
            = Instant.parse("2018-12-31T10:15:30.00Z");

        // apply query method of Instant class
        // print the result
        System.out.println("Zone value for Instant is "
                           +
instant.query(TemporalQueries.offset()));
    }
}
```

Output:

```
Zone value for Instant is null
```

JDBC - Update Records Example

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 a database server.
- **Execute a query** − Requires using an object of type Statement for building and submitting an SQL statement to update records in a table. This Query makes use of **IN** and **WHERE** clause to update conditional records.
- **Clean up the environment** − try with resources automatically closes the resources.

Sample Code

Copy and paste the following example in JDBCExample.java, compile and run as follows −

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

public class JDBCExample {
   static final String DB_URL = "jdbc:mysql://localhost/TUTORIALSPOINT";
   static final String USER = "guest";
   static final String PASS = "guest123";
   static final String QUERY = "SELECT id, first, last, age FROM Registration";

   public static void main(String[] args) {
      // Open a connection
      try(Connection conn = DriverManager.getConnection(DB_URL, USER, PASS);
      Statement stmt = conn.createStatement();
    ) {
      String sql = "UPDATE Registration " +
        "SET age = 30 WHERE id in (100, 101)";
      stmt.executeUpdate(sql);
      ResultSet rs = stmt.executeQuery(QUERY);
      while(rs.next()){
        //Display values
        System.out.print("ID: " + rs.getInt("id"));
        System.out.print(", Age: " + rs.getInt("age"));
        System.out.print(", First: " + rs.getString("first"));
        System.out.println(", Last: " + rs.getString("last"));
      }
      rs.close();
    } catch (SQLException e) {
      e.printStackTrace();
    }
  }
}
```

Now let us compile the above example as follows −

```
C:\>javac JDBCExample.java
C:\>
```

When you run **JDBCExample**, it produces the following result −

```
C:\>java JDBCExample
ID: 100, Age: 30, First: Zara, Last: Ali
ID: 101, Age: 30, First: Mahnaz, Last: Fatma
ID: 102, Age: 30, First: Zaid, Last: Khan
ID: 103, Age: 28, First: Sumit, Last: Mittal
C:\>
```

JDBC - Delete Records Example

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.
- **Register the JDBC driver** − Requires that you initialize a driver so you can open a communications channel with the database.
- **Open a connection** − Requires using the *DriverManager.getConnection()* method to create a Connection object, which represents a physical connection with a database server.
- **Execute a query** − Requires using an object of type Statement for building and submitting an SQL statement to delete records from a table. This Query makes use of the **WHERE** clause to delete conditional records.
- **Clean up the environment** − try with resources automatically closes the resources.

Sample Code

Copy and paste the following example in JDBCExample.java, compile and run as follows −

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

public class JDBCExample {
   static final String DB_URL = "jdbc:mysql://localhost/TUTORIALSPOINT";
   static final String USER = "guest";
   static final String PASS = "guest123";
   static final String QUERY = "SELECT id, first, last, age FROM Registration";

   public static void main(String[] args) {
     // Open a connection
     try(Connection conn = DriverManager.getConnection(DB_URL, USER, PASS);
       Statement stmt = conn.createStatement();
     ) {
       String sql = "DELETE FROM Registration " +
         "WHERE id = 101";
       stmt.executeUpdate(sql);
       ResultSet rs = stmt.executeQuery(QUERY);
       while(rs.next()){
         //Display values
         System.out.print("ID: " + rs.getInt("id"));
         System.out.print(", Age: " + rs.getInt("age"));
         System.out.print(", First: " + rs.getString("first"));
         System.out.println(", Last: " + rs.getString("last"));
       }
       rs.close();
     } catch (SQLException e) {
       e.printStackTrace();
     }
   }
}
```

Now let us compile the above example as follows −

```
C:\>javac JDBCExample.java
C:\>
```

When you run **JDBCExample**, it produces the following result −

```
C:\>java JDBCExample
ID: 100, Age: 30, First: Zara, Last: Ali
ID: 102, Age: 30, First: Zaid, Last: Khan
ID: 103, Age: 28, First: Sumit, Last: Mittal
C:\>
```

Types of Statements in JDBC

The statement interface is used to create SQL basic statements in Java it provides methods to execute queries with the database. There are different types of statements that are used in JDBC as follows:

- Create Statement
- Prepared Statement
- Callable Statement

1. **Create a Statement:** From the connection interface, you can create the object for this interface. It is generally used for general–purpose access to databases and is useful while using static SQL statements at runtime.

Syntax:

```
Statement statement = connection.createStatement();
```

Implementation: Once the Statement object is created, there are three ways to execute it.

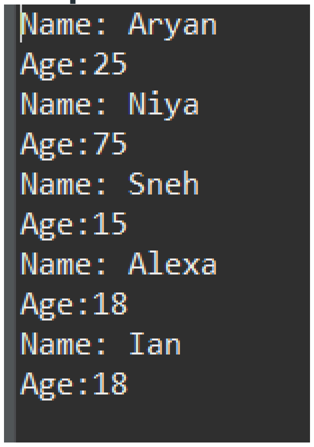
- ***boolean execute(String SQL):*** If the ResultSet object is retrieved, then it returns true else false is returned. Is used to execute SQL DDL statements or for dynamic SQL.
- **int executeUpdate(String SQL):** Returns number of rows that are affected by the execution of the statement, used when you need a number for INSERT, DELETE or UPDATE statements.
- ***ResultSet executeQuery(String SQL):*** Returns a ResultSet object. Used similarly as SELECT is used in SQL.

Example:

- Java

```
// Java Program illustrating Create Statement in JDBC

// Importing Database(SQL) classes
import java.sql.*;

// Class
class GFG {

    // Main driver method
    public static void main(String[] args)
    {

        // Try block to check if any exceptions occur
        try {

            // Step 2: Loading and registering drivers

            // Loading driver using forName() method
            Class.forName("com.mysql.cj.jdbc.Driver");
            // Registering driver using DriverManager
            Connection con = DriverManager.getConnection(
                "jdbc:mysql:///world", "root", "12345");

            // Step 3: Create a statement
            Statement statement = con.createStatement();
            String sql = "select * from people";

            // Step 4: Execute the query
            ResultSet result = statement.executeQuery(sql);

            // Step 5: Process the results

            // Condition check using hasNext() method which
            // holds true till there is single element
            // remaining in List
            while (result.next()) {

                // Print name an age
                System.out.println(
                    "Name: " + result.getString("name"));
                System.out.println(
                    "Age:" + result.getString("age"));
            }
        }

        // Catching database exceptions if any
        catch (SQLException e) {

            // Print the exception
            System.out.println(e);
        }

        // Catching generic ClassNotFoundException if any
        catch (ClassNotFoundException e) {

            // Print and display the line number
            // where exception occurred
            e.printStackTrace();
        }
    }
}
```

Output: Name and age are as shown for random inputs



```
Name: Aryan
Age:25
Name: Niya
Age:75
Name: Sneh
Age:15
Name: Alexa
Age:18
Name: Ian
Age:18
```

2. Prepared Statement represents a recompiled SQL statement, that can be executed many times. This accepts parameterized SQL queries. In this, “?” is used instead of the parameter, one can pass the parameter dynamically by using the methods of PREPARED STATEMENT at run time.

Illustration:

Considering in the people database if there is a need to INSERT some values, SQL statements such as these are used:

```
INSERT INTO people VALUES ("Ayan",25);

INSERT INTO people VALUES("Kriya",32);
```

To do the same in Java, one may use Prepared Statements and set the values in the ? holders, setXXX() of a prepared statement is used as shown:

```
String query = "INSERT INTO people(name, age)VALUES(?, ?)";

Statement pstmt = con.prepareStatement(query);

pstmt.setString(1,"Ayan");

ptstmt.setInt(2,25);

// where pstmt is an object name
```

Implementation: Once the PreparedStatement object is created, there are three ways to execute it:

- ***execute():*** This returns a boolean value and executes a static SQL statement that is present in the prepared statement object.
- ***executeQuery():*** Returns a ResultSet from the current prepared statement.
- ***executeUpdate():*** Returns the number of rows affected by the DML statements such as INSERT, DELETE, and more that is present in the current Prepared Statement.

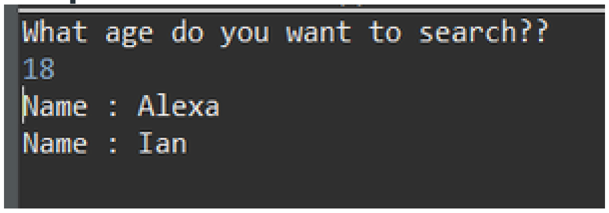
Example:

- Java

```
// Java Program illustrating Prepared Statement in JDBC

// Step 1: Importing DB(SQL here) classes
import java.sql.*;
// Importing Scanner class to
// take input from the user
import java.util.Scanner;

// Main class
class GFG {

    // Main driver method
    public static void main(String[] args)
    {
        // try block to check for exceptions
        try {

            // Step 2: Establish a connection

            // Step 3: Load and register drivers

            // Loading drivers using forName() method
            Class.forName("com.mysql.cj.jdbc.Driver");

            // Scanner class to take input from user
            Scanner sc = new Scanner(System.in);

            // Display message for ease for user
            System.out.println(
                "What age do you want to search?? ");
            // Reading age an primitive datatype from user
            // using nextInt() method
            int age = sc.nextInt();

            // Registering drivers using DriverManager
            Connection con = DriverManager.getConnection(
                "jdbc:mysql:///world", "root", "12345");

            // Step 4: Create a statement
            PreparedStatement ps = con.prepareStatement(
                "select name from world.people where age = ?");

            // Step 5: Execute the query
            ps.setInt(1, age);
            ResultSet result = ps.executeQuery();

            // Step 6: Process the results

            // Condition check using next() method
            // to check for element
            while (result.next()) {

                // Print and display elements(Names)
                System.out.println("Name : "
                                   + result.getString(1));
            }

            // Step 7: Closing the connections
            // (Optional but it is recommended to do so)
        }

        // Catch block to handle database exceptions
        catch (SQLException e) {

            // Display the DB exception if any
            System.out.println(e);
        }

        // Catch block to handle class exceptions
        catch (ClassNotFoundException e) {

            // Print the line number where exception occurred
            // using printStackTrace() method if any
```

```
            e.printStackTrace();
        }
    }
}
```

Output:



```
What age do you want to search??
18
Name : Alexa
Name : Ian
```

3. Callable Statement are stored procedures which are a group of statements that we compile in the database for some task, they are beneficial when we are dealing with multiple tables with complex scenario & rather than sending multiple queries to the database, we can send the required data to the stored procedure & lower the logic executed in the database server itself. The Callable Statement interface provided by JDBC API helps in executing stored procedures.
Syntax: To prepare a CallableStatement

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

Implementation: Once the callable statement object is created

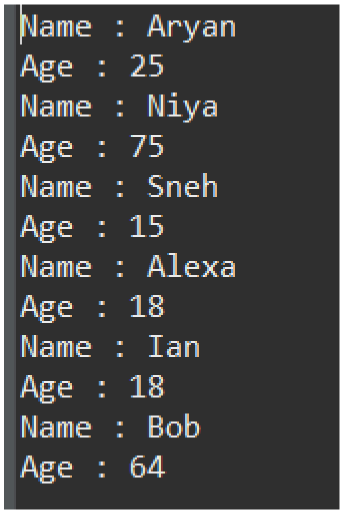
- ***execute()*** is used to perform the execution of the statement.

Example:

- Java

```
// Java Program illustrating Callable Statement in JDBC
// Step 1: Importing DB(SQL) classes
import java.sql.*;

// Main class
class GFG {

    // Main driver method
    public static void main(String[] args)
    {
        // Try block to check if any exceptions occurs
        try {

            // Step 2: Establish a connection

            // Step 3: Loading and registering drivers

            // Loading driver using forName() method
            Class.forName("com.mysql.cj.jdbc.Driver");

            // Registering driver using DriverManager
            Connection con = DriverManager.getConnection(
                "jdbc:mysql:///world", "root", "12345");

            // Step 4: Create a statement
            Statement s = con.createStatement();

            // Step 5: Execute the query
            // select * from people

            CallableStatement cs
                = con.prepareCall("{call peopleinfo(?,?)}");
            cs.setString(1, "Bob");
            cs.setInt(2, 64);
            cs.execute();
            ResultSet result
                = s.executeQuery("select * from people");

            // Step 6: Process the results

            // Condition check using next() method
            // to check for element
            while (result.next()) {

                // Print and display elements (Name and Age)
                System.out.println("Name : "
                                   + result.getString(1));
                System.out.println("Age : "
                                   + result.getInt(2));
            }
        }

        // Catch statement for DB exceptions
        catch (SQLException e) {
            // Print the exception
            System.out.println(e);
        }

        // Catch block for generic class exceptions
        catch (ClassNotFoundException e) {

            // Print the line number where exception occurred
            e.printStackTrace();
        }
    }
}
```

Output:



```
Name : Aryan
Age : 25
Name : Niya
Age : 75
Name : Sneh
Age : 15
Name : Alexa
Age : 18
Name : Ian
Age : 18
Name : Bob
Age : 64
```

JDBC - Result Sets

The SQL statements that read data from a database query, return the data in a result set. The SELECT statement is the standard way to select rows from a database and view them in a result set. The *java.sql.ResultSet* interface represents the result set of a database query.

A ResultSet object maintains a cursor that points to the current row in the result set. The term "result set" refers to the row and column data contained in a ResultSet object.

The methods of the ResultSet interface can be broken down into three categories −

- **Navigational methods** − Used to move the cursor around.
- **Get methods** − Used to view the data in the columns of the current row being pointed by the cursor.
- **Update methods** − Used to update the data in the columns of the current row. The updates can then be updated in the underlying database as well.

The cursor is movable based on the properties of the ResultSet. These properties are designated when the corresponding Statement that generates the ResultSet is created.

JDBC provides the following connection methods to create statements with desired ResultSet −

- **createStatement(int RSType, int RSConcurrency);**
- **prepareStatement(String SQL, int RSType, int RSConcurrency);**
- **prepareCall(String sql, int RSType, int RSConcurrency);**

The first argument indicates the type of a ResultSet object and the second argument is one of two ResultSet constants for specifying whether a result set is read-only or updatable.

Type of ResultSet

The possible RSType are given below. If you do not specify any ResultSet type, you will automatically get one that is TYPE_FORWARD_ONLY.

Type	Description
ResultSet.TYPE_FORWARD_ONLY	The cursor can only move forward in the result set.
ResultSet.TYPE_SCROLL_INSENSITIVE	The cursor can scroll forward and backward, and the result set is not sensitive to changes made by others to the database that occur after the result set was created.
ResultSet.TYPE_SCROLL_SENSITIVE.	The cursor can scroll forward and backward, and the result set is sensitive to changes made by others to the database that occur after the result set was created.

Concurrency of ResultSet

The possible RSConcurrency are given below. If you do not specify any Concurrency type, you will automatically get one that is CONCUR_READ_ONLY.

Concurrency	Description
ResultSet.CONCUR_READ_ONLY	Creates a read-only result set. This is the default
ResultSet.CONCUR_UPDATABLE	Creates an updateable result set.

All our examples written so far can be written as follows, which initializes a Statement object to create a forward-only, read only ResultSet object −

```
try {
   Statement stmt = conn.createStatement(ResultSet.TYPE_FORWARD_ONLY,
ResultSet.CONCUR_READ_ONLY);
}
catch(Exception ex) {
   ....
}
finally {
  ....
}
```

Navigating a Result Set

There are several methods in the ResultSet interface that involve moving the cursor, including −

S.N.	Methods & Description
1	**public void beforeFirst() throws SQLException** Moves the cursor just before the first row.
2	**public void afterLast() throws SQLException** Moves the cursor just after the last row.
3	**public boolean first() throws SQLException** Moves the cursor to the first row.
4	**public void last() throws SQLException** Moves the cursor to the last row.
5	**public boolean absolute(int row) throws SQLException** Moves the cursor to the specified row.
6	**public boolean relative(int row) throws SQLException** Moves the cursor the given number of rows forward or backward, from where it is currently pointing.
7	**public boolean previous() throws SQLException** Moves the cursor to the previous row. This method returns false if the previous row is off the result set.
8	**public boolean next() throws SQLException**

	Moves the cursor to the next row. This method returns false if there are no more rows in the result set.
9	**public int getRow() throws SQLException** Returns the row number that the cursor is pointing to.
10	**public void moveToInsertRow() throws SQLException** Moves the cursor to a special row in the result set that can be used to insert a new row into the database. The current cursor location is remembered.
11	**public void moveToCurrentRow() throws SQLException** Moves the cursor back to the current row if the cursor is currently at the insert row; otherwise, this method does nothing

For a better understanding, let us study Navigate - Example Code.

Viewing a Result Set

The ResultSet interface contains dozens of methods for getting the data of the current row.

There is a get method for each of the possible data types, and each get method has two versions −

- One that takes in a column name.
- One that takes in a column index.

For example, if the column you are interested in viewing contains an int, you need to use one of the getInt() methods of ResultSet −

S.N.	Methods & Description
1	**public int getInt(String columnName) throws SQLException** Returns the int in the current row in the column named columnName.
2	**public int getInt(int columnIndex) throws SQLException**

	Returns the int in the current row in the specified column index. The column index starts at 1, meaning the first column of a row is 1, the second column of a row is 2, and so on.

Similarly, there are get methods in the ResultSet interface for each of the eight Java primitive types, as well as common types such as java.lang.String, java.lang.Object, and java.net.URL.

There are also methods for getting SQL data types java.sql.Date, java.sql.Time, java.sql.TimeStamp, java.sql.Clob, and java.sql.Blob. Check the documentation for more information about using these SQL data types.

For a better understanding, let us study Viewing - Example Code.

Updating a Result Set

The ResultSet interface contains a collection of update methods for updating the data of a result set.

As with the get methods, there are two update methods for each data type −

- One that takes in a column name.
- One that takes in a column index.

For example, to update a String column of the current row of a result set, you would use one of the following updateString() methods −

S.N.	Methods & Description
1	**public void updateString(int columnIndex, String s) throws SQLException** Changes the String in the specified column to the value of s.
2	**public void updateString(String columnName, String s) throws SQLException** Similar to the previous method, except that the column is specified by its name instead of its index.

There are update methods for the eight primitive data types, as well as String, Object, URL, and the SQL data types in the java.sql package.

Updating a row in the result set changes the columns of the current row in the ResultSet object, but not in the underlying database. To update your changes to the row in the database, you need to invoke one of the following methods.

S.N.	Methods & Description
1	**public void updateRow()** Updates the current row by updating the corresponding row in the database.
2	**public void deleteRow()** Deletes the current row from the database
3	**public void refreshRow()** Refreshes the data in the result set to reflect any recent changes in the database.
4	**public void cancelRowUpdates()** Cancels any updates made on the current row.
5	**public void insertRow()** Inserts a row into the database. This method can only be invoked when the cursor is pointing to the insert row.

Display Records From Database Using JTable

Introduction

This article explains how to display fetched information using JTable in Java. The NetBeans IDE is used for creating this app.

What is JTable

It is just like a simple table. It displays records in a tabular format. It is a component of Swing and is used for GUI programming based on a matrix format.

Display Emp Records from Database in JTable

For creating this app we need the following files:

1. Java file (DisplayEmpData.java)
2. SQL table (emp.sql)

1. DisplayEmpData.java

This Java file consists of all the logic. First of all we initialize JFrame components using a constructor then create a database connection and finally set the database value to the textfield. If the given name is not found in the database then it displays an error message and displays it by running the constructor.

2. emp.sql

For fetching records we need a database table; for that we create an "emp" table in our database.

Syntax
emp.sql

```
create table emp
(
    uname varchar2(20), umail varchar2(30),
    upass varchar2(20), ucountry varchar2(20)
);
```

SQL
Copy

Insert some rows in it

```
insert into emp values ('sandeep', 'sandy05.1991@gmail.com',
'welcome', 'India');
insert into emp values ('rahul', 'rahul@gmail.com' , '123',
'India');
```

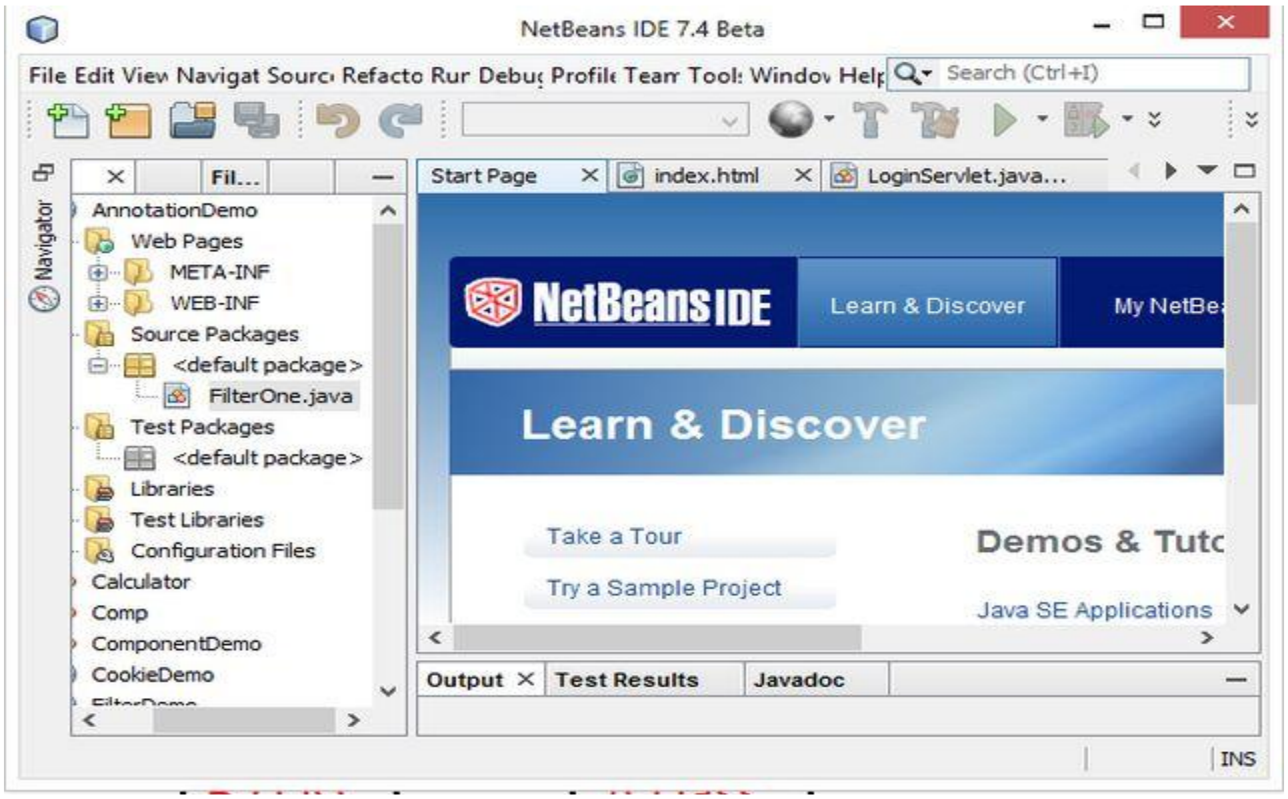
SQL
Copy

Now let's start creating this app. Use the following procedure to do that in the NetBeans IDE.

For creating this app in the NetBeans IDE, we need to use the following procedure.

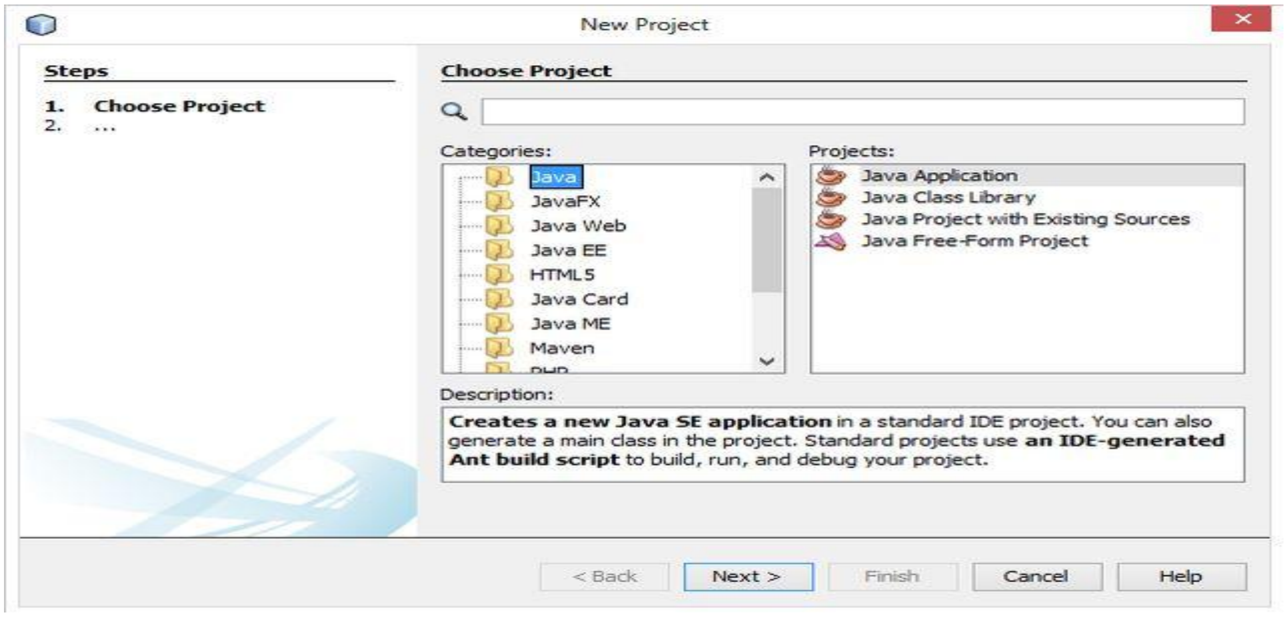
Step 1

Open the NetBeans IDE.



Step 2

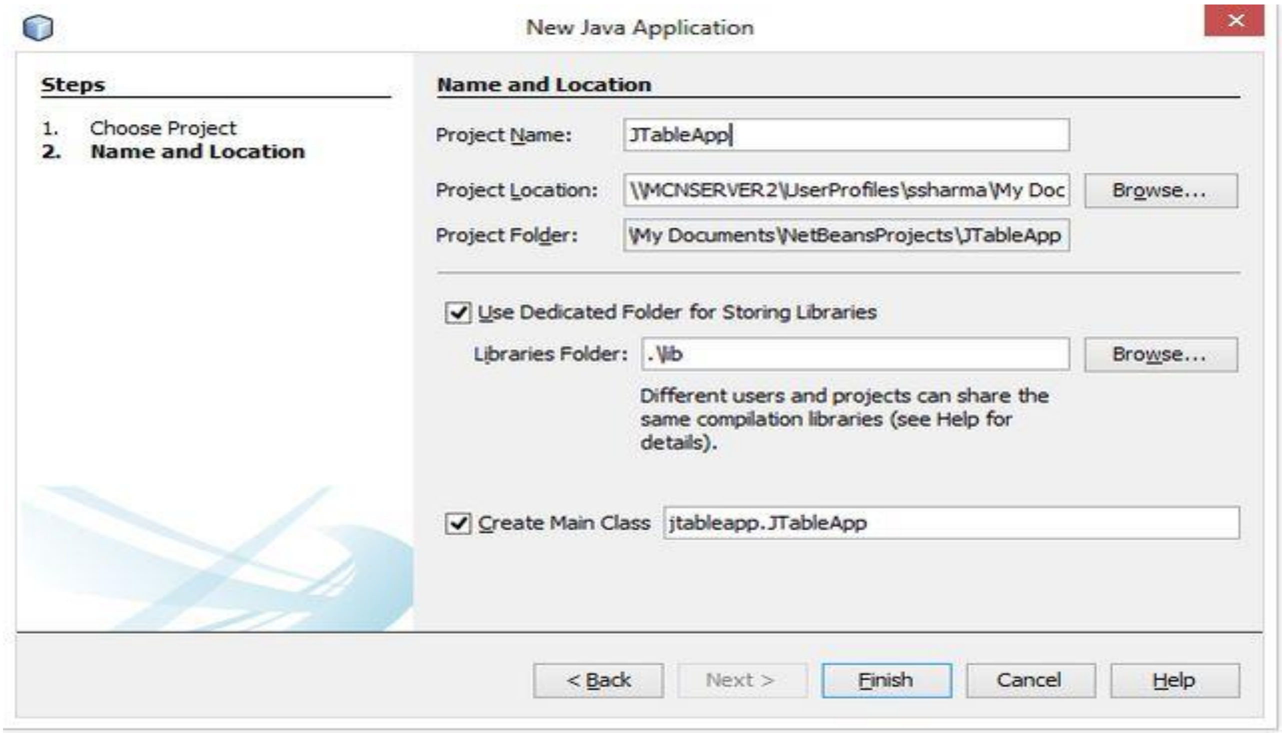
Choose "Java" -> "Java Application" as in the following.



Creates a new Java SE application in a standard IDE project. You can also generate a main class in the project. Standard projects use **an IDE-generated Ant build script** to build, run, and debug your project.

Step 3

Now type your project name as "JTableApp" as in the following.



Project Name: JTableApp

Project Location: \\MCNSERVER2\UserProfiles\ssharma\My Doc

Project Folder: \My Documents\NetBeansProjects\JTableApp

✓ Use Dedicated Folder for Storing Libraries

Libraries Folder: .\lib

Different users and projects can share the same compilation libraries (see Help for details).

✓ Create Main Class jtableapp.JTableApp

Step 4

Now create a new Java Class with the name "DisplayEmpData" and provide the following code for it

DisplayEmpData.java

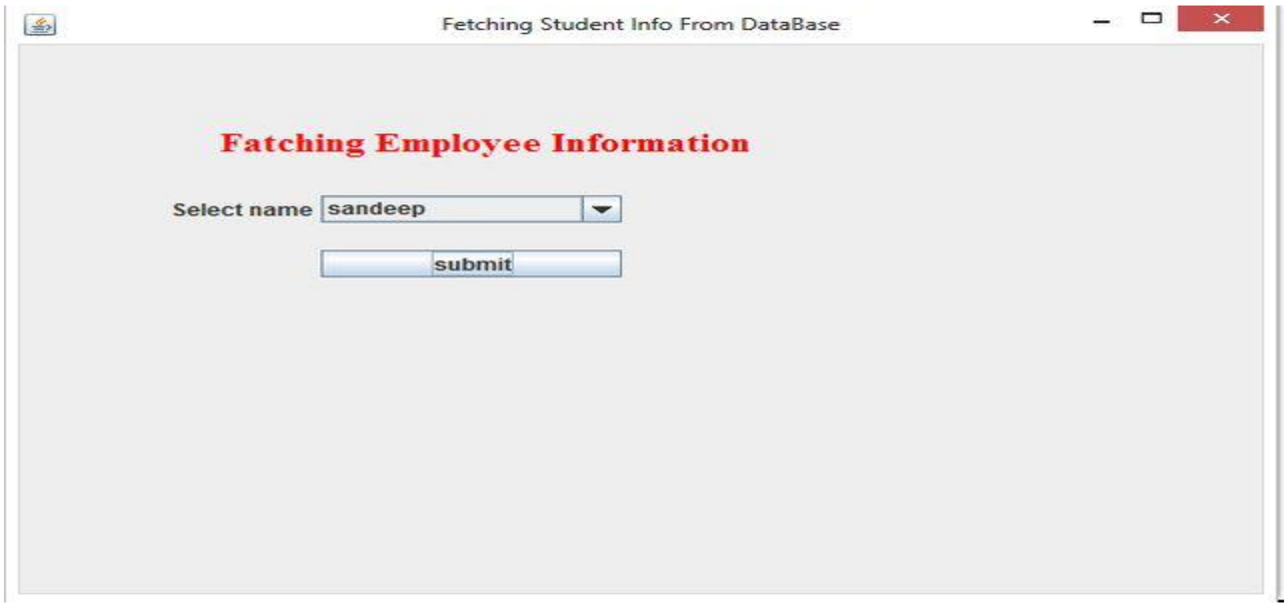
```
import java.awt.*;
import java.awt.event.*;
import java.sql.*;
import java.util.Vector;
import javax.swing.*;
import javax.swing.table.DefaultTableModel;
public class DisplayEmpData extends JFrame implements
ActionListener {
    JFrame frame1;
    JLabel l0, l1, l2;
    JComboBox c1;
    JButton b1;
    Connection con;
    ResultSet rs, rs1;
    Statement st, st1;
    PreparedStatement pst;
    String ids;
    static JTable table;
    String[] columnNames = {"User name", "Email", "Password",
"Country"};
    String from;
    DisplayEmpData() {
        l0 = new JLabel("Fatching Employee Information");
        l0.setForeground(Color.red);
        l0.setFont(new Font("Serif", Font.BOLD, 20));
        l1 = new JLabel("Select name");
        b1 = new JButton("submit");
        l0.setBounds(100, 50, 350, 40);
        l1.setBounds(75, 110, 75, 20);
        b1.setBounds(150, 150, 150, 20);
        b1.addActionListener(this);
        setTitle("Fetching Student Info From DataBase");
        setLayout(null);
        setVisible(true);
        setSize(500, 500);

setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
        add(l0);
        add(l1);;
        add(b1);
        try {
            Class.forName("oracle.jdbc.driver.OracleDriver");
            con =
DriverManager.getConnection("jdbc:oracle:thin:@mcndesktop07:152
1:xe", "sandeep", "welcome");
            st = con.createStatement();
            rs = st.executeQuery("select uname from emp");
            Vector v = new Vector();
            while (rs.next()) {
                ids = rs.getString(1);
                v.add(ids);
            }
            c1 = new JComboBox(v);
            c1.setBounds(150, 110, 150, 20);
            add(c1);
            st.close();
            rs.close();
        } catch (Exception e) {
        }
    }
    public void actionPerformed(ActionEvent ae) {
        if (ae.getSource() == b1) {
            showTableData();
        }
    }
    public void showTableData() {
        frame1 = new JFrame("Database Search Result");
        frame1.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame1.setLayout(new BorderLayout());
        //TableModel tm = new TableModel();
        DefaultTableModel model = new DefaultTableModel();
        model.setColumnIdentifiers(columnNames);
        //DefaultTableModel model = new
DefaultTableModel(tm.getData1(), tm.getColumnNames());
        //table = new JTable(model);
        table = new JTable();
        table.setModel(model);

table.setAutoResizeMode(JTable.AUTO_RESIZE_ALL_COLUMNS);
        table.setFillsViewportHeight(true);
        JScrollPane scroll = new JScrollPane(table);
        scroll.setHorizontalScrollBarPolicy(
                JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED);
        scroll.setVerticalScrollBarPolicy(
                JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED);
        from = (String) c1.getSelectedItem();
        //String textvalue = textbox.getText();
        String uname = "";
        String email = "";
        String pass = "";
        String cou = "";
        try {
            pst = con.prepareStatement("select * from emp where
UNAME='" + from + "'");
            ResultSet rs = pst.executeQuery();
            int i = 0;
            if (rs.next()) {
                uname = rs.getString("uname");
                email = rs.getString("umail");
                pass = rs.getString("upass");
                cou = rs.getString("ucountry");
                model.addRow(new Object[]{uname, email, pass,
cou});
                i++;
            }
            if (i < 1) {
                JOptionPane.showMessageDialog(null, "No Record
Found", "Error", JOptionPane.ERROR_MESSAGE);
            }
            if (i == 1) {
                System.out.println(i + " Record Found");
            } else {
                System.out.println(i + " Records Found");
            }
        } catch (Exception ex) {
            JOptionPane.showMessageDialog(null,
ex.getMessage(), "Error", JOptionPane.ERROR_MESSAGE);
        }
        frame1.add(scroll);
        frame1.setVisible(true);
        frame1.setSize(400, 300);
    }
    public static void main(String args[]) {
        new DisplayEmpData();
    }
}
```

Java
Copy

Step 5

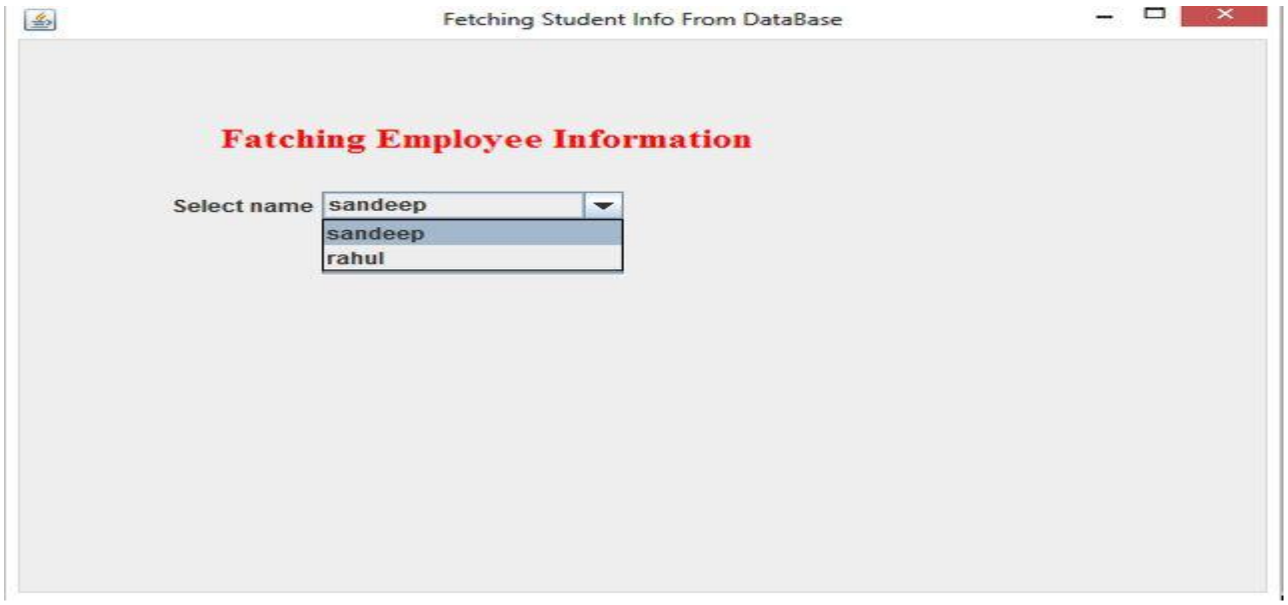
Now our project is ready to run.

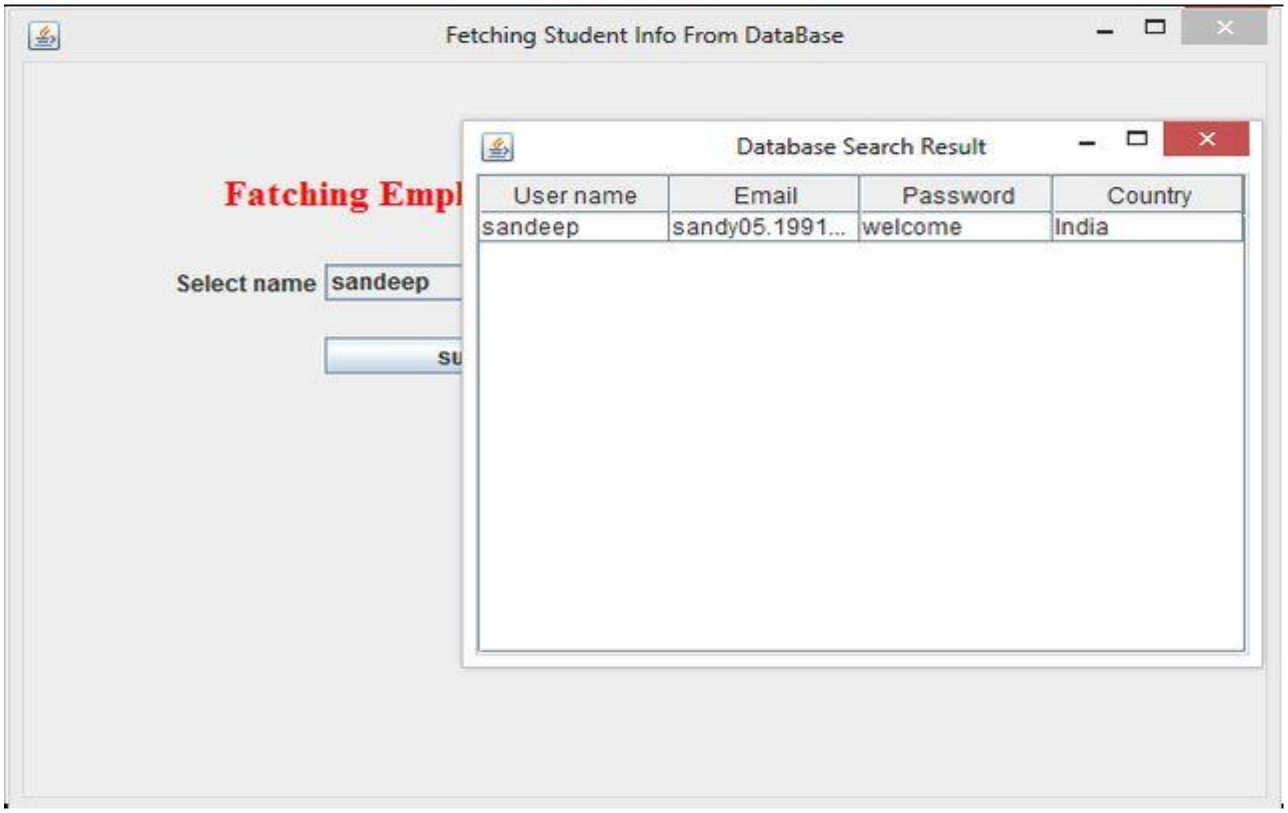
Right-click on the project menu then choose "Run". The following output will be generated.


Fetching Student Info From DataBase
Fatching Employee Information
Select name
sandeep
submit

Step 6

Now choose any name from the ComboBox and click on the "Submit" button as in the following.

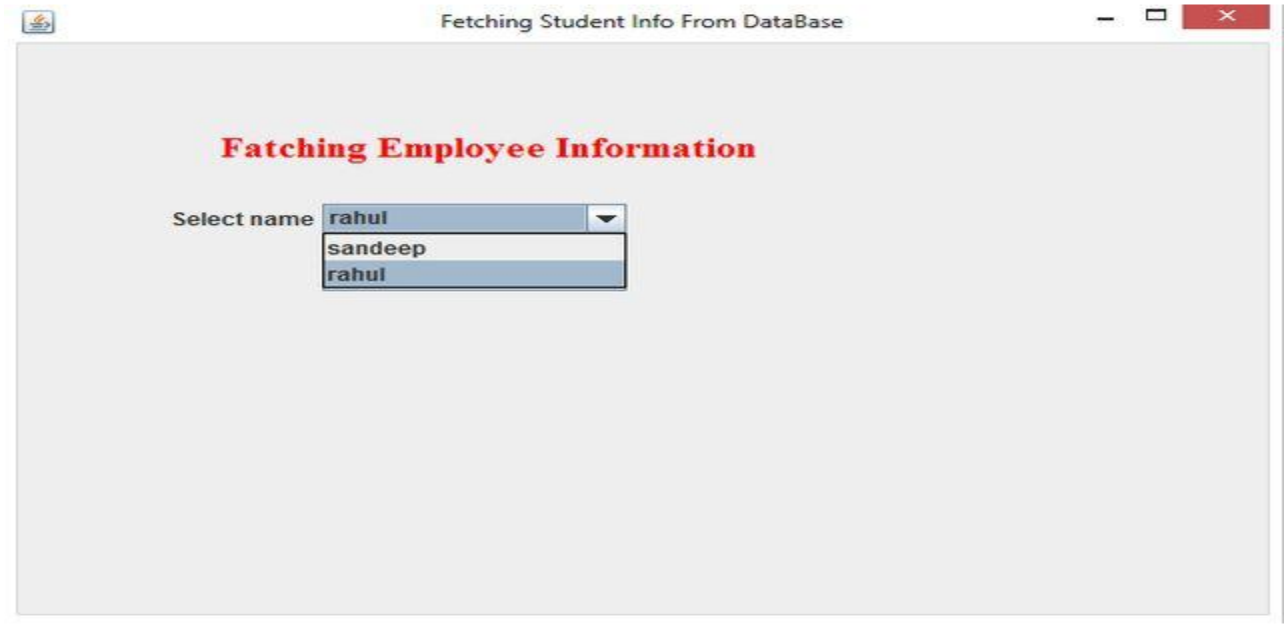

Fetching Student Info From DataBase
Fatching Employee Information
Select name
sandeep
sandeep
rahul

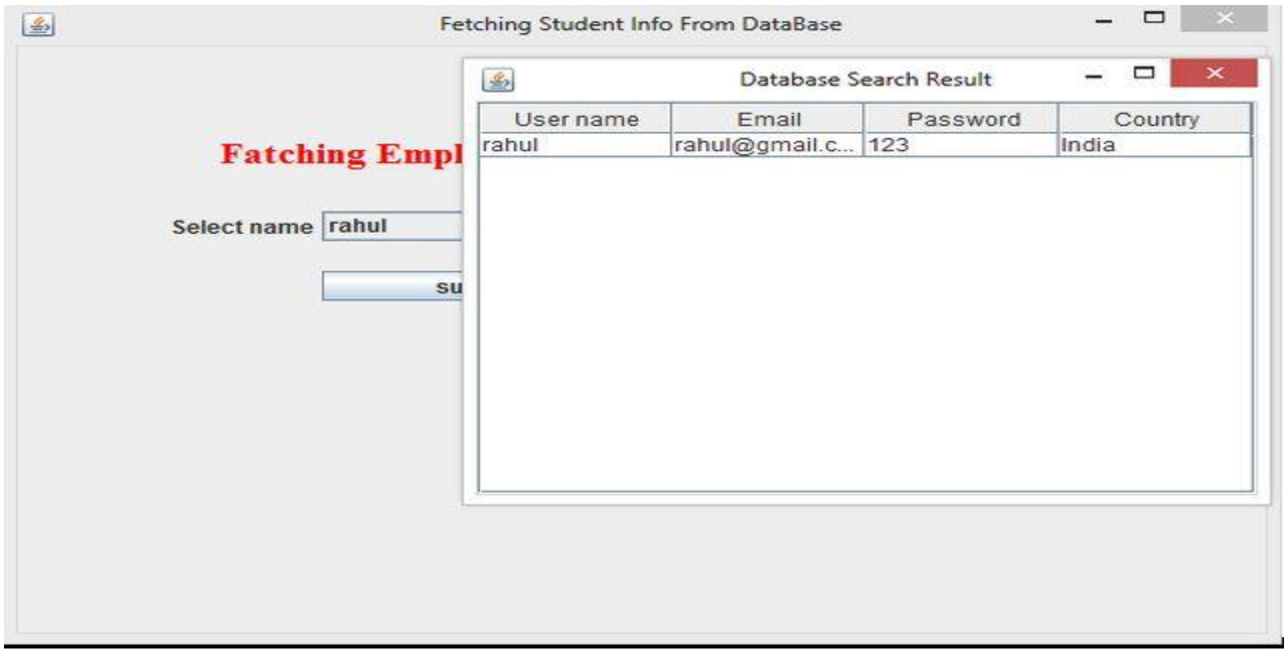


User name	Email	Password	Country
sandeep	sandy05.1991...	welcome	India

Step 7

Now choose another name as in the following.


Fetching Student Info From DataBase
Fatching Employee Information
Select name
rahul
sandeep
rahul



User name	Email	Password	Country
rahul	rahul@gmail.c...	123	India