CHAPTER-15

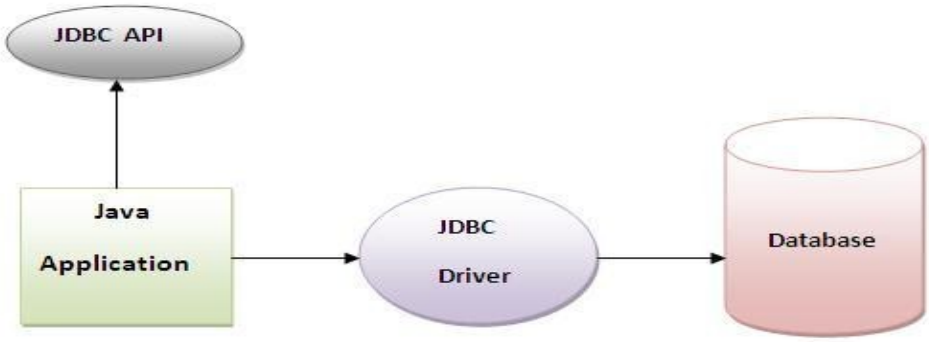
JDBC

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 commonly associated with database usage:

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



Why use JDBC?

Before JDBC, ODBC API was used to connect and execute query to the database. But ODBC API uses ODBC driver that is written in C language which is plateform dependent and unsecured. That is why

Sun Microsystem has defined its own API (JDBC API) that uses JDBC driver written in Java language

Database Connectivity History

- Before APIs like JDBC and ODBC, database connectivity was tedious.
- Each database vendor provided a function library for accessing their database.
- The connectivity library was proprietary.
- If the database vendor changed for the application, the data access portions had to be rewritten.
- If the application was poorly structured, rewriting its data access might involve rewriting the majority of the application.
- The costs incurred generally meant that application developers were stuck with a particular database product for a given application.

JDBC Driver

A JDBC driver is a set of Java classes that implement the JDBC interfaces, targeting a specific database. The JDBC interfaces comes with standard Java, but the implementation of these interfaces is specific to the database you need to connect to. Such an implementation is called a JDBC driver.

JDBC drivers implement the interfaces in the JDBC API for interacting with your database server.

For example, using JDBC drivers enable you to open database connections and to interact with it by sending SQL or database commands then receiving results with Java.

The ***Java.sql*** package that ships with JDK contains various classes with their behaviors defined and their actual implementations are done in third-party drivers. Third party vendors implements the *java.sql.Driver* interface in their database driver.

JDBC driver implementations vary because of the wide variety of operating systems and hardware platforms in which Java operates. Sun has divided the implementation types into four categories, Types 1, 2, 3, and 4, which is explained below:

There are 4 types of JDBC drivers:

1. Type1: JDBC-ODBC bridge driver
2. Type2: Native-API driver (partially java driver)
3. Type3: Network Protocol driver (fully java driver)
4. Type4: Thin or Direct driver (fully java driver)

1. Type 1 : JDBC-ODBC bridge driver (Obsolete in JDK1.8)

In a Type 1 driver, a JDBC bridge is used to access ODBC drivers installed on each client machine. Using ODBC, requires configuring on your system a Data Source Name (DSN) that represents the target database. It converts JDBC method calls into the ODBC function calls.

When Java first came out, this was a useful driver because most databases only supported ODBC access but now this type of driver is recommended only for experimental use or when no other alternative is available.

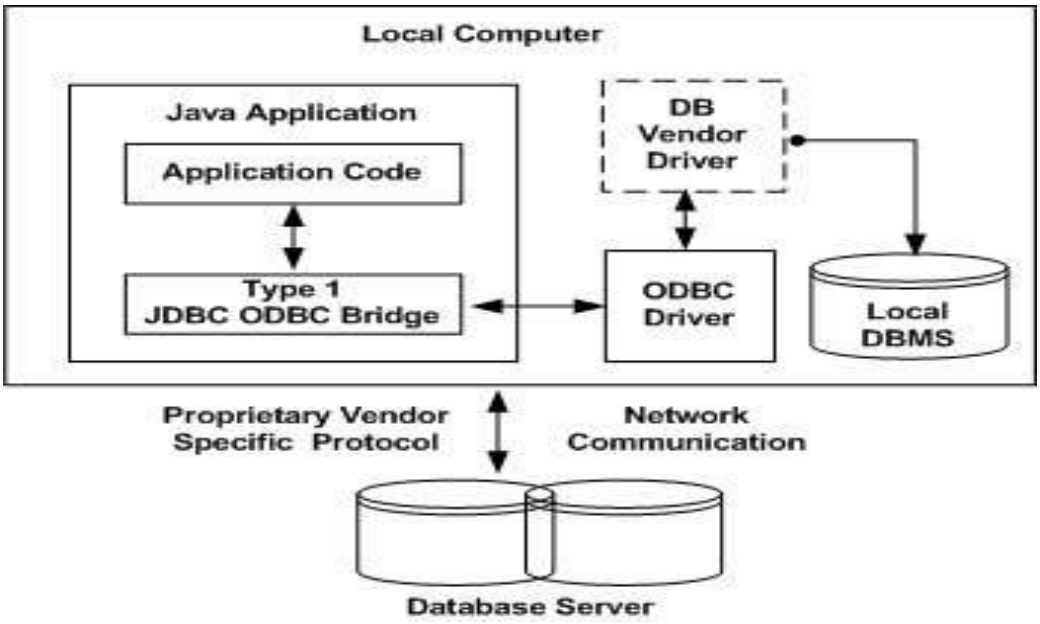


Figure JDBC-ODBC Bridge Driver

Advantage:

- Easy to use
- Allow easy connectivity to all database supported by the ODBC Driver.

Disadvantage

- Performance degraded because JDBC method call is converted into the ODBC function calls.
- The ODBC driver needs to be installed on the client machine.
- Compare to others it is very slow.
- Not Supported by JDK1.8

2) Type 2: JDBC-Native API Driver

In a Type 2 driver, JDBC API calls are converted into native C/C++ API calls which are unique to the database. These drivers typically provided by the database vendors and used in the same manner as the JDBC-ODBC Bridge, the vendor-specific driver must be installed on each client machine.

If we change the Database we have to change the native API as it is specific to a database and they are mostly obsolete now but you may realize some speed increase with a Type 2 driver, because it eliminates ODBC's overhead.

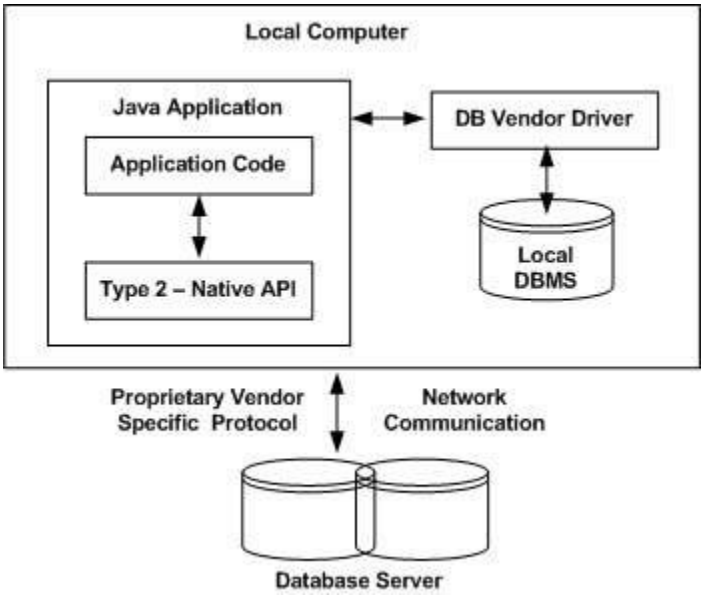


Figure JDBC-Native API Driver

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

Advantage:

- performance upgraded than JDBC-ODBC bridge driver.

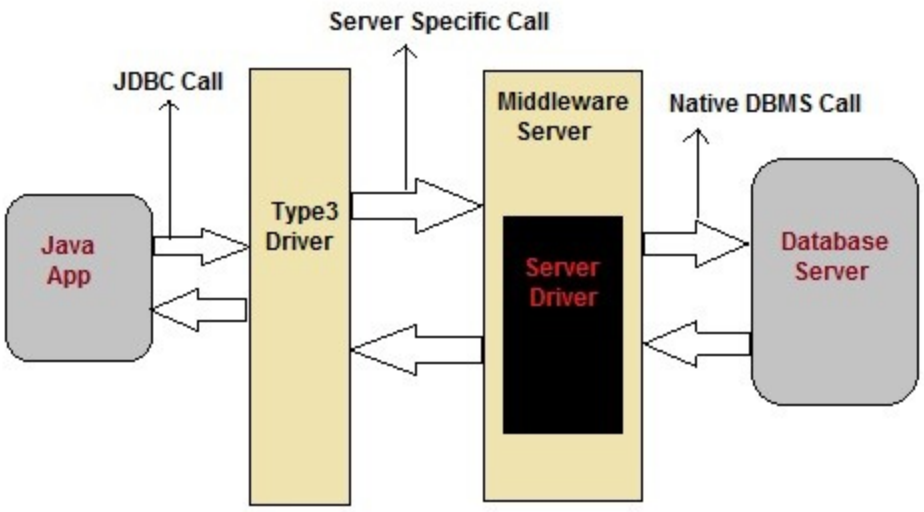
Disadvantage:

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

3) Type 3: JDBC-Net pure Java:

In a Type 3 driver, a three-tier approach is used to accessing databases. The JDBC clients use standard network sockets to communicate with an middleware application server. The socket information is then translated by the middleware application server into the call format required by the DBMS, and forwarded to the database server.

This kind of driver is extremely flexible, since it requires no code installed on the client and a single driver can actually provide access to multiple databases.



You can think of the application server as a JDBC "proxy," meaning that it makes calls for the client application. As a result, you need some knowledge of the application server's configuration in order to effectively use this driver type.

Advantage:

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

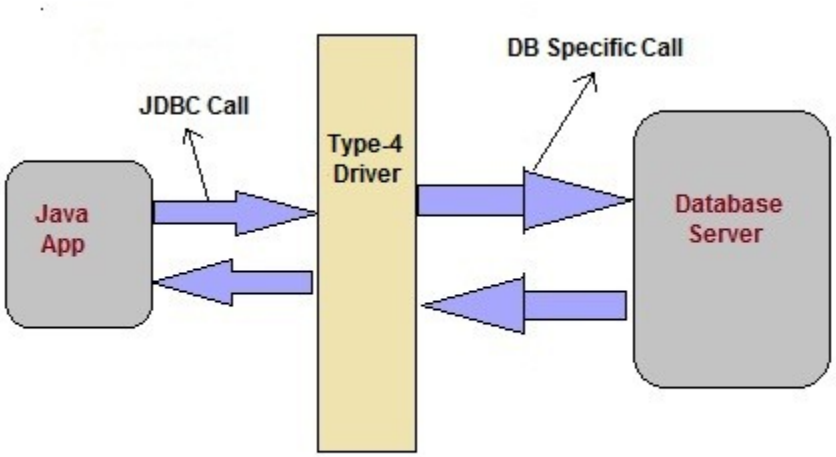
Disadvantages:

- 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) Type 4: 100% pure Java:

In a Type 4 driver, a pure Java-based driver that communicates directly with vendor's database through socket connection. This is the highest performance driver available for the database and is usually provided by the vendor itself.

This kind of driver is extremely flexible, you don't need to install special software on the client or server. Further, these drivers can be downloaded dynamically.



MySQL's Connector/J driver is a Type 4 driver. Because of the proprietary nature of their network protocols, database vendors usually supply type 4 drivers.

Advantage:

- Does not require any native library.
- Does not require any Middleware server.
- Better Performance than other driver.

Disadvantage:

Slow due to increase number of network call.

Which Driver should be used?

- If you are accessing one type of database, such as Oracle, Sybase, or IBM, the preferred driver type is 4.
- If your Java application is accessing multiple types of databases at the same time, type 3 is the preferred driver.
- Type 2 drivers are useful in situations where a type 3 or type 4 driver is not available yet for your database.
- The type 1 driver is not considered a deployment-level driver and is typically used for development and testing purposes only.

DBC 4.0 API

JDBC 4.0 API is mainly divided into two package

1. `java.sql`
2. `javax.sql`

java.sql package

This package include 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 the 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 statements.

javax.sql package

This package is also known as JDBC extension API. It provides classes and interface to access server-side data.

Important classes and interface of javax.sql package

classes/interface	Description
javax.sql.ConnectionEvent	Provide information about occurence of event.
javax.sql.ConnectionEventListener	Used to register event generated by **PooledConnection** object.
javax.sql.DataSource	Represent the **DataSource** interface used in an application.
javax.sql.PooledConnection	provide object to manage connection pools.

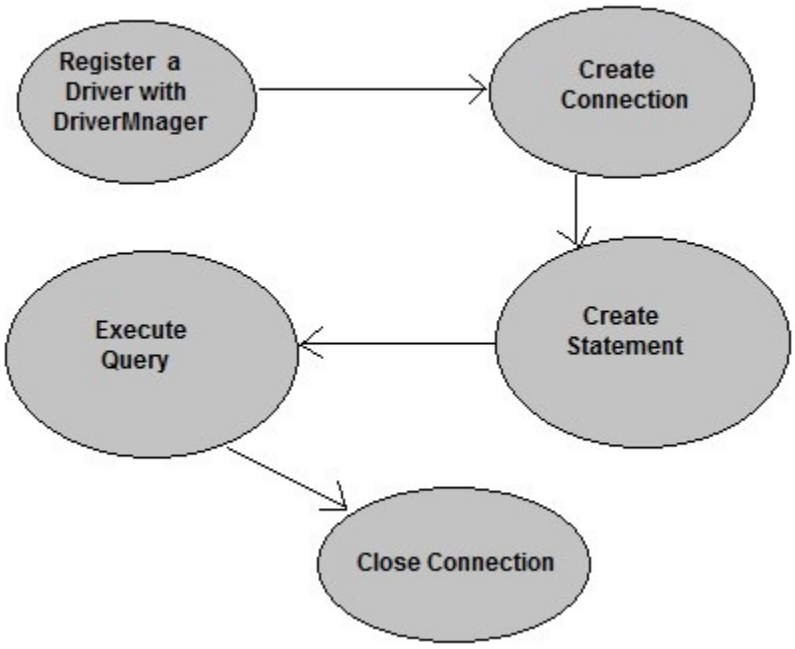
5 Steps to connect to the database in java

There are 5 steps to connect any java application with the database in java using JDBC. They are as follows:

There are following six steps involved in building a 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 communication channel with the database.
- **Open a connection:**

  Requires using the *DriverManager.getConnection()* method to create a Connection object, which represents a physical connection with the database.
- **Execute a query:** Requires using an object of type **Statement** for building and submitting an SQL statement to the database.
- **Extract data from result set:** Requires that you use the appropriate *ResultSet.getXXX()* method to retrieve the data from the result set.
- **Clean up the environment:** Requires explicitly closing all database resources versus relying on the JVM's garbage collection.



1) Register the driver class

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

Syntax of forName() method

public static void forName(String className)throws ClassNotFoun dException

Example to register the Driver class

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

Class.forName("com.mysql.jdbc.Driver");                / / for MySQL
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");  // for MS-Access
```

2) Create the connection object

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

Syntax of getConnection() method

1. public static Connection getConnection(String url)throws SQL Exception
2. public static Connection getConnection(String url,String name, String password)
3. throws SQLException

Example to establish connection with the database

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


Connection con=DriverManager.getConnection(
"jdbc:mysql://localhost:3306/employeedb","root","root");   //for
MySQL

Connection
con=DriverManager.getConnection("Jdbc:Odbc:empdsn");// MS-
Access with  DSN


String database="EmpDB.accdb";//Here database exists in the curre
nt directory
```

```
Connection con=DriverManager.getConnection("jdbc:odbc:Driver=
{Microsoft Access Driver (*.accdb)};     DBQ=" + database ");
// MS-Access Without DSN
```

3) Create the Statement object

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

Syntax of createStatement() method

```
public Statement createStatement()throws SQLException
```

Example to create the statement object

```
Statement stmt=con.createStatement();
```

4) Execute the query

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

Syntax of executeQuery() method

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

Example to execute query

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

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

}

5) Close the connection object

By closing connection object statement and ResultSet will be closed automatically. The close() method of Connection interface is used to close the connection.

Syntax of close() method

public void close()throws SQLException

Example to close connection

con.close();

Example to connect to the Oracle database

For connecting java application with the oracle database, you need to follow 5 steps to perform database connectivity. In this example we are using Oracle10g as the database. So we need to know following information's for the oracle database:

1. **Driver class:** The driver class for the oracle database is **oracle.jdbc.driver.OracleDriver**.
2. **Connection URL:** The connection URL for the oracle10G database is **jdbc:oracle:thin:@localhost:1521:xe** where jdbc is the API, oracle is the database, thin is the driver, **localhost** is the server name on which oracle is running, we may also use IP address, 1521 is the port number and XE is the Oracle service name. You may get all these information from the tnsnames.ora file.
3. **Username:** The default username for the oracle database is **system**.

4. **Password:** Password is given by the user at the time of installing the oracle database.

Let's first create a table in oracle database.

create table emp(id number(10),name varchar2(40),age number(3));

Example of Connecting to Oracle Database using Thin Driver

To connect a Java application with Oracle database using Thin Driver. You need to follow the following steps

1. **Load Driver Class:** The Driver Class for oracle database is **oracle.jdbc.driver.OracleDriver** and **Class.forName("oracle.jdbc.driver.OracleDriver")** method is used to load the driver class for Oracle database.
2. **Create Connection:** For creating a connection you will need a Connection URL. The Connection URL for Oracle is

   You will also require **Username** and **Password** of your Oracle Database Server for creating connection.

3. **Loading jar file:** To connect your java application with Oracle, you will also need to load **ojdbc14.jar** file. This file can be loaded into 2 ways.

1. Copy the jar file into C:\Program Files\Java\jre7\lib\ext folder.

   or,

2. Set it into classpath.

NOTE: Here we are discussing about Oracle 10g as database. For other version of Oracle you will be require to do some small changes in the Connection URL.

Example: Accessing record from Emp table in Java application

```
import java.sql.*;
class OracleCon
{
    public static void main(String args[])
    {
       try
       {
          //step1 load the driver class
          Class.forName("oracle.jdbc.driver.OracleDriver");

          //step2 create  the connection object
          Connection con=DriverManager.getConnection(
          "jdbc:oracle:thin:@localhost:1521:xe","system","oracle")
       ;

          //step3 create the statement object
          Statement stmt=con.createStatement();

          //step4 execute query
```

```
            ResultSet rs=stmt.executeQuery("select * from emp");
            while(rs.next())
            System.out.println(rs.getInt(1)+"  "+rs.getString(2)+"  "+r
        s.getString(3));

        //step5 close the connection object
        con.close();

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

    }
}
```

To connect java application with the Oracle database **ojdbc14.jar** file is required to be loaded.

Two ways to load the jar file:

1. paste the **ojdbc14.jar** file in jre/lib/ext folder
2. set classpath

1) paste the ojdbc14.jar file in JRE/lib/ext folder:

Firstly, search the ojdbc14.jar file then go to JRE/lib/ext folder and paste the jar file here.

2) set classpath:

There are two ways to set the classpath:

- temporary
- permanent

How to set the temporary classpath:

Firstly, search the **ojdbc14.jar** file then open command prompt and write:

1. C:>set classpath=c:\folder\ojdbc14.jar;.;

How to set the permanent classpath:

Go to environment variable then click on new tab. In variable name write **classpath** and in variable value paste the path to **ojdbc14.jar** by appending **ojdbc14.jar;.;** as

C:\oraclexe\app\oracle\product\10.2.0\server\jdbc\lib\ojdbc14.jar;.;

Inserting record into a table using java application

```
import java.sql.*;
class Test
{
public static void main(String []args)
{
try{
//Loading driver...
Class.forName("oracle.jdbc.driver.OracleDrive
r");

//creating connection...
Connection con = DriverManager.getConnection

("jdbc:oracle:thin:@localhost:1521:XE","usern
ame","password");
```

```
PreparedStatement
pst=con.prepareStatement("insert into Student
values(?,?)");

          pst.setInt(1,104);
          pst.setString(2,"Alex");
      pst.executeUpdate();

con.close();  //closing connection
}catch(Exception e){
e.printStacktrace();
}
}
}
```

Connecting to MySQL Database using Thin Driver

To connect a Java application with MySQL database using Thin Driver. You need to follow the following 5 steps

In this example we are using MySql as the database. So we need to know following informations for the mysql database:

1. **Driver class:** The driver class for the mysql database is **com.mysql.jdbc.Driver**.
2. **Connection URL:** The connection URL for the mysql database is **jdbc:mysql://localhost:3306/empdb** where jdbc is the API, mysql is the database, localhost is the server name on which mysql is running, we may also use IP address, 3306 is the port number and **test** is the database name. We may use any database, in such case, you need to replace the **test** with your database name.

3. **Loading jar file:** To connect your java application with MySQL, you will also need to load `mysql-connector.jar` file. This file can be loaded into 2 ways.

    I. Copy the jar file into `C:\Program Files\Java\jre7\lib\ext` folder.
        a. or,
    II. Set it into classpath.

1) paste the mysqlconnector.jar file in JRE/lib/ext folder:

Download the **mysqlconnector.jar** file. Go to C:\Program Files\Java\jdk1.7.0_21\jre\lib\ext folder and paste the jar file here.

2) set classpath:

There are two ways to set the classpath:

- temporary
- permament

How to set the temporary classpath

open comman prompt and write:

1. C:>set classpath=c:\folder\mysql-connector-java-5.0.8-bin.jar;.;

How to set the permanent classpath

Go to environment variable then click on new button. In variable name write **classpath** and in variable value paste the path to the mysqlconnector.jar file by appending mysqlconnector.jar;.; as C:\Program Files\Java\jdk1.7.0_21\jre\lib\ext

Let's first create a table in the **mysql** database, but before creating table, we need to create database first.

```
create database empdb;
use empdb;
create table emp(id int(10),name varchar(40),age int(3));
```

Example to Connect Java Application with mysql database

In this example, empdb is the database name, root is the username and password.

```
import java.sql.*;
class MysqlCon{
public static void main(String args[]){
try{
Class.forName("com.mysql.jdbc.Driver");

Connection con=DriverManager.getConnection(
"jdbc:mysql://localhost:3306/empdb","root","root");
```

```
//here empdb is database name, root is username and password

Statement stmt=con.createStatement();

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

while(rs.next())
System.out.println(rs.getInt(1)+"  "+rs.getString(2)+"  "+rs.getStr
ing(3));

con.close();

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

}
}
```

To connect java application with the mysql database **mysqlconnector-java.jar** file is required to be loaded.

Inserting record into a table using java application

```
import java.sql.*;
class Test
{
public static void main(String []args)
{
try{
//Loading driver
Class.forName("com.mysql.jdbc.Driver");

//creating connection
```

```
Connection con = DriverManager.getConnection
                ("jdbc:mysql:/
/localhost:3306/test","username","password");

PreparedStatement pst=con.prepareStatement("insert
into Student values(?,?)");

          pst.setInt(1,104);
          pst.setString(2,"Alex");
      pst.executeUpdate();

con.close();  //closing connection
}catch(Exception e){
e.printStacktrace();
}
}
}
```

Connectivity with Access without DSN

There are two ways to connect java application with the access database.

1. Without DSN (Data Source Name)
2. With DSN

Java is mostly used with Oracle, mysql, or DB2 database. So you can learn this topic only for knowledge.

Example to Connect Java Application with access without DSN

In this example, we are going to connect the java program with the access database. In such case, we have created the login table in the access database. There is only one column in the table named name. Let's get all the name of the login table.

```
import java.sql.*;
class Test{
public static void main(String ar[]){
 try{
   String database="empdb.accdb";//Here database exists in the curre
nt directory

   String url="jdbc:odbc:Driver={Microsoft Access Driver (*.accdb)
};
                DBQ=" + database + ";

   Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
   Connection c=DriverManager.getConnection(url);
   Statement st=c.createStatement();
   ResultSet rs=st.executeQuery("select * from login");

   while(rs.next()){
    System.out.println(rs.getString(1));
   }

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

}}
```

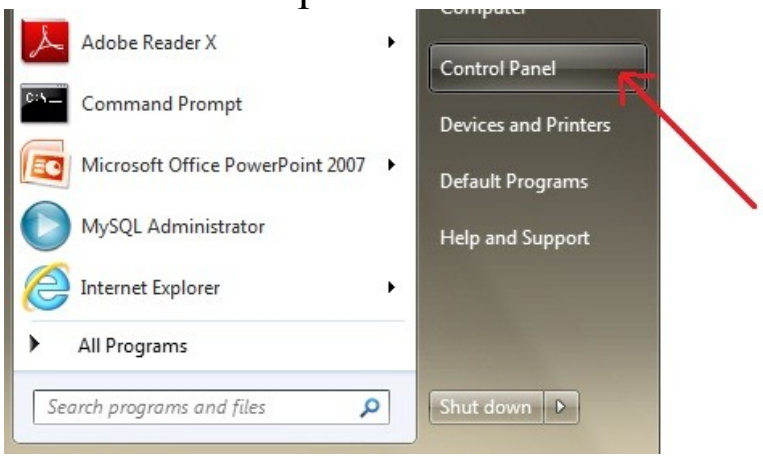
Connecting to Access Database using Type-1 Driver

To connect a Java application with Access database using JDBC-ODBC Bridge(type-1) Driver.
To connect java application with type1 driver, create DSN first, here we are assuming your dsn name is mydsn.

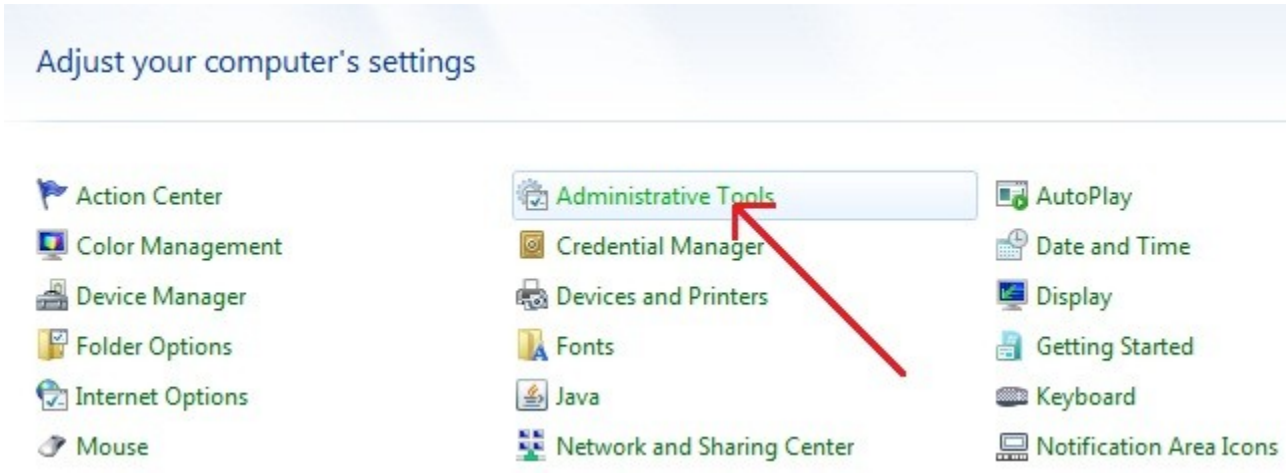
You need to follow the following steps:

Create DSN Name

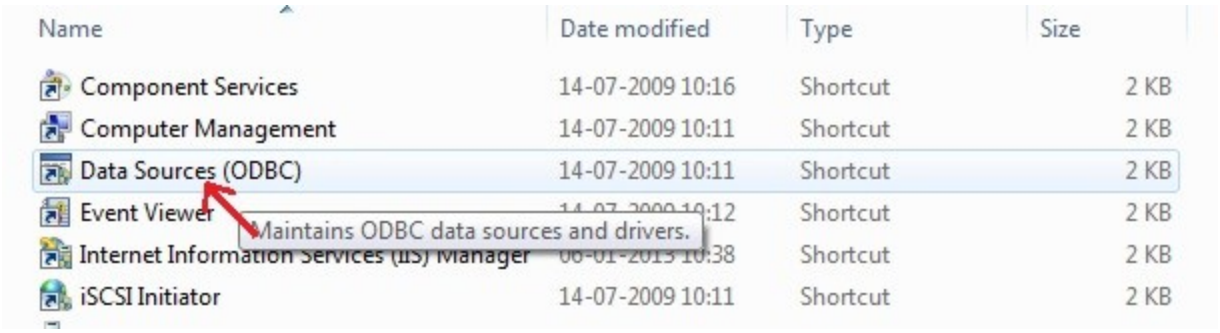
1. Go to control panel



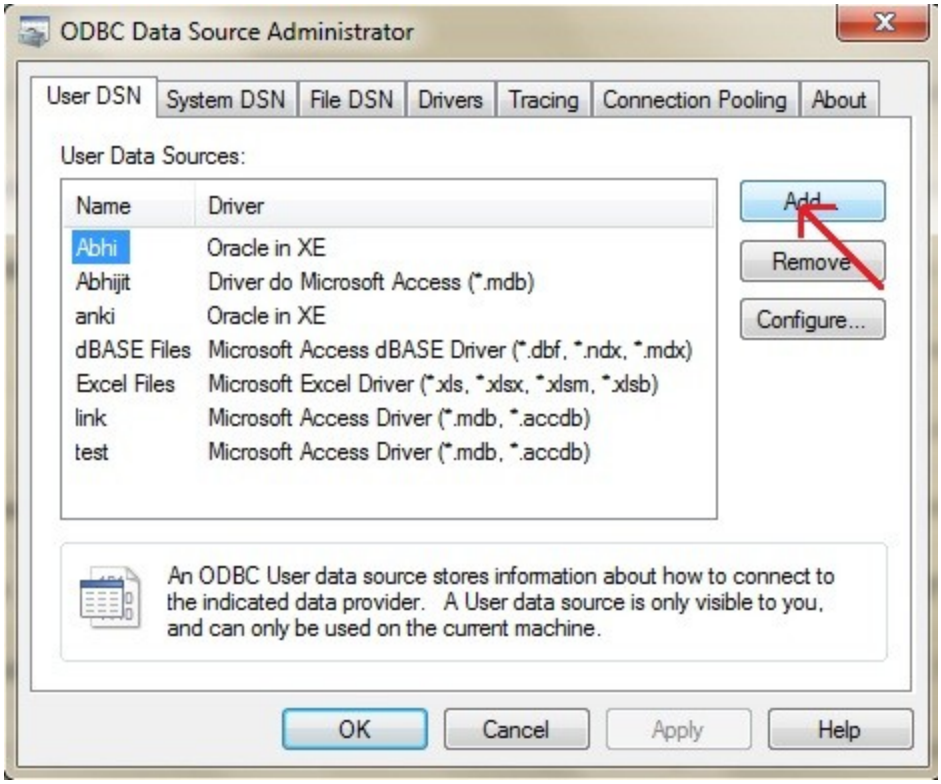
1. **Go to Administrative tools**



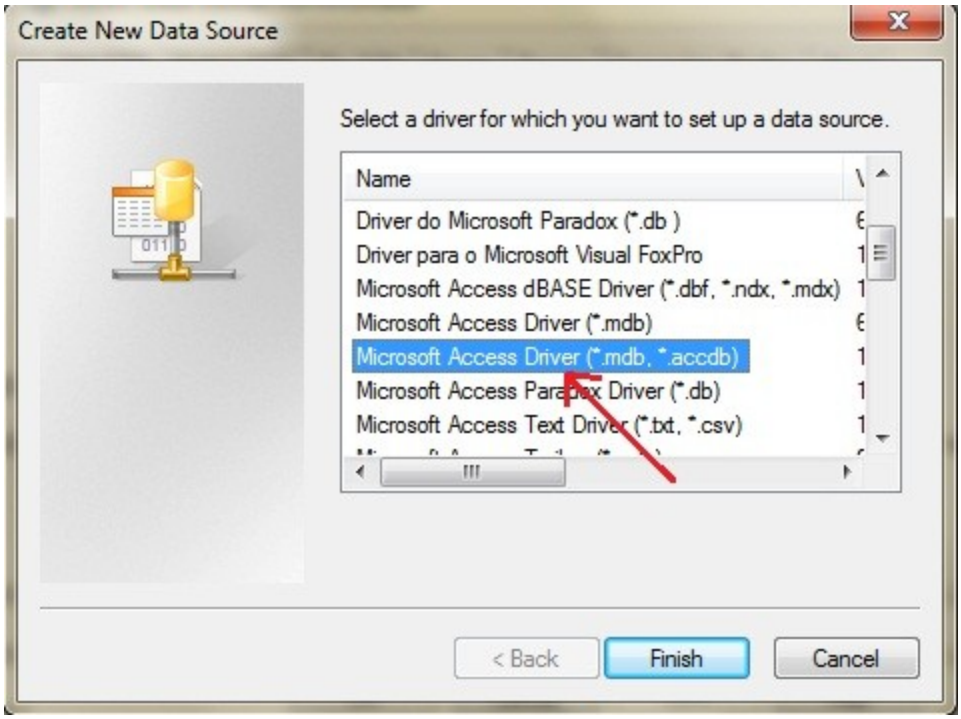
2. **Select Data Source(ODBC)**



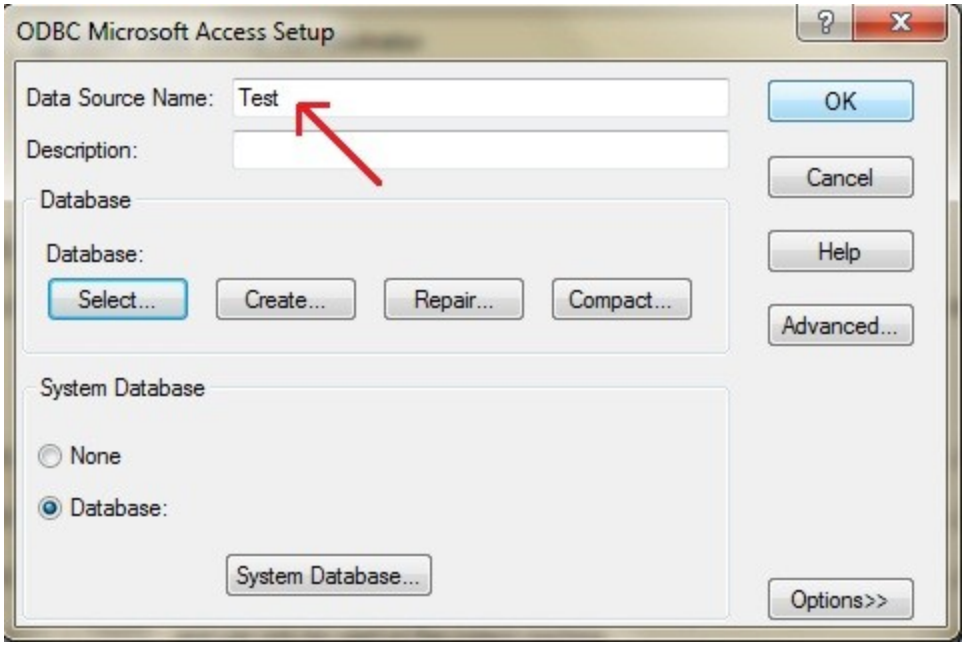
3. **Add new DSN name, select add**



4. **Select Access driver from the list, click on finish**



5. **Give a DSN name, click ok**



NOTE: Here we are showing this example to create DSN in Window 7 os. For other operating system you need to do small changes.

Example

We suppose that you have created a **login** table with username and password column name in access database.

```
import java.sql.*;
class Test{
public static void main(String ar[]){
 try{
   String url="jdbc:odbc:mydsn";
   Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
   Connection c=DriverManager.getConnection(url);
   Statement st=c.createStatement();
   ResultSet rs=st.executeQuery("select * from login");

   while(rs.next()){
    System.out.println(rs.getString(1));
   }

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

}}
```

OUTPUT

```
Exception in thread "main"
java.lang.ClassNotFoundException:
sun.jdbc.odbc.JdbcOdbcDriver

    at
java.net.URLClassLoader.findClass(URLClassLoader.j
ava:381)
    at
java.lang.ClassLoader.loadClass(ClassLoader.java:4
24)
    at
sun.misc.Launcher$AppClassLoader.loadClass(Launche
r.java:331)
    at
java.lang.ClassLoader.loadClass(ClassLoader.java:3
57)
    at java.lang.Class.forName0(Native Method)
    at java.lang.Class.forName(Class.java:264)
    at JDBCExample1.main(JDBCExample1.java:8)
```

java.lang.classnotfoundexception sun.jdbc.odbc.jdbcodbcdriver exception comes in Java 8 because it has removed the JDBC ODBC bridge driver class "sun.jdbc.odbc.jdbcodbcdriver" from JDK and JRE. This class is required to connect any database using Object database connectivity driver e.g. Microsoft Access, but unfortunately you cannot use it from JDK 8 onward. In order to solve this error, just use Jackcess library or a commercial driver like

HXTT. Normally, in pre Java 8 world, java.lang.classnotfoundexception sun.jdbc.odbc.jdbcodbcdriver error comes when you try to connect to Microsoft Access database from Java using JDBC and JDBC ODBC bridge driver is not available in classpath. If you remember, In order to open SQL connection to database, first step is to load and register the driver. In order to load driver, we use Class.forName("sun.jdbc.odbc.JdbcOdbcDriver"); and this method throws *java.lang.classnotfoundexception sun.jdbc.odbc.jdbcodbcdriver*, if ClassLoader is not able to find the requested class (sun.jdbc.odbc.JdbcOdbcDriver) in CLASSPATH. In order to connect to MS Access, we need type 1 JDBC driver, also known as *JDBC ODBC bridge driver* and the class in question, JdbcOdbcDriver is driver implementation to connect to Open database connectivity driver installed in machine.

Just remember that **sun.jdbc.odbc.JdbcOdbcDriver** is a standard class from JDK API and it doesn't come with any external JAR like other vendor database's JDBC drivers e.g. JDBC driver to connect Oracle database comes on ojdbc6.jar and MySQL driver comes in mysql-connector-java-5.1.23-bin.jar. JdbcOdbcDriver class is present in rt.jar, which is always included in Classpath, as this JAR file is part of the JRE.

DriverManager class:

The **DriverManager** class acts as an interface between user and drivers. It keeps track of the drivers that are available and handles establishing a connection between a database and the appropriate driver. The **DriverManager** class maintains a list of Driver classes that have registered themselves by calling the method **DriverManager.registerDriver()**.

Commonly used methods of DriverManager class:

1) public static void registerDriver(Driver driver):	is used to register the given driver with DriverManager.
2) public static void deregisterDriver(Driver driver):	is used to deregister the given driver (drop the driver from the list) with DriverManager.
3) public static Connection getConnection(String url):	is used to establish the connection with the specified url.
4) public static Connection getConnection(String url,String userName,String password):	is used to establish the connection with the specified url, username and password.

Connection interface:

A Connection is the session between java application and database. The Connection interface is a factory of **Statement**, **PreparedStatement**, and **DatabaseMetaData** i.e. object of Connection can be used to get the object of **Statement** and **DatabaseMetaData**. The Connection interface provide many methods for transaction management like **commit()**, **rollback()** etc.

By default, connection commits the changes after executing queries.

Commonly used methods of Connection interface:

1) public Statement createStatement(): creates a statement object that can be used to execute SQL queries.

2) public Statement createStatement(int resultSetType,int resultSetConcurrency): Creates a Statement object that will generate **ResultSet** objects with the given type and concurrency.

3) public void setAutoCommit(boolean status): is used to set the commit status. By default it is true.

4) public void commit(): saves the changes made since the previous commit/rollback permanent.

5) public void rollback(): Drops all changes made since the previous commit/rollback.

6) public void close(): closes the connection and Releases a JDBC resources immediately.

Statement interface

Once a connection is obtained we can interact with the database. The JDBC ***Statement, CallableStatement,*** and ***PreparedStatement*** interfaces define the methods and properties that enable you to send SQL or PL/SQL commands and receive data from your database.

Following table provides a summary of each interface's purpose to understand how do you decide which interface to use?

Interfaces	Recommended Use
Statement	Use for general-purpose access to your database. Useful when you are using static SQL statements at runtime. The Statement interface cannot accept parameters.
PreparedStatement	Use when you plan to use the SQL statements many times. The PreparedStatement interface accepts input parameters at runtime.
CallableStatement	Use when you want to access database stored procedures. The CallableStatement interface can also accept runtime input parameters.

The Statement Objects:

Creating Statement Object:

Before you can use a Statement object to execute a SQL statement, you need to create one using the Connection object's **createStatement( )** method, as in the following example:

```
Statement stmt = null;
try {
   stmt = conn.createStatement( );
   . . .
```

```
}
catch (SQLException e) {
   . . .
}
```

Once you've created a Statement object, you can then use it to execute a SQL statement with one of its three execute methods.

- **boolean execute(String SQL)** : Returns a boolean value of true if a **ResultSet** object can be retrieved; otherwise, it returns false. Use this method to execute SQL DDL statements or when you need to use truly dynamic SQL.
- **int executeUpdate(String SQL)** : Returns the numbers of rows affected by the execution of the SQL statement. Use this method to execute SQL statements for which you expect to get a number of rows affected - for example, an INSERT, UPDATE, or DELETE statement.
- **ResultSet executeQuery(String SQL)** : Returns a **ResultSet** object. Use this method when you expect to get a result set, as you would with a SELECT statement.

Closing Statement Object:

Just as you close a Connection object to save database resources, for the same reason you should also close the Statement object.

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

```
Statement stmt = null;
try {
   stmt = con.createStatement( );
```

```
    . . .
Stmt.close();
}
catch (SQLException e) {
    . . .
}
```

The PreparedStatement Objects:

JDBC PreparedStatement can be used when you plan to use the same SQL statement many times. It is used to handle precompiled query. If we want to execute same query with different values for more than one time then precompiled queries will reduce the no of compilations. **Connection.prepareStatement()** method can provide you **PreparedStatment** object. This object provides **setXXX()** methods to provide query values. Below example shows how to use PreparedStatement.

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) {
```

```
        . . .
    }
```

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 to 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 start 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 take input the parameters.

Example of Statement interface

Let's see the simple example of Statement interface to insert, update and delete the record.

```
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:th
      in:@localhost:1521:xe","s   ystem","oracle");
```

```
Statement stmt=con.createStatement();

//stmt.executeUpdate("insert into pers values(‘E101’,'Irfan',’Cl
erk’,5)");

//int result=stmt.executeUpdate("update pers set ename='Vimal',dno
=1  where dno=1");
int result=stmt.executeUpdate("delete from pers where dno=33");

System.out.println(result+" records affected");
con.close();
}}
```

Methods of PreparedStatement interface

The important methods of PreparedStatement interface are given below:

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));
```

Now insert records in this table by the code given below:

```
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:1521:xe","system","oracle");

PreparedStatement stmt=con.prepareStatement("insert into Emp val
ues(?,?)");
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);}

}
}
```

Example of PreparedStatement interface that updates the record

```
PreparedStatement stmt=con.prepareStatement("update emp set nam
e=? where id=?");
stmt.setString(1,"Empdb");//1 specifies the first parameter in the que
ry i.e. name
stmt.setInt(2,101);

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

Example of PreparedStatement interface that deletes the record

```
PreparedStatement stmt=con.prepareStatement("delete from emp w
here id=?");
stmt.setInt(1,101);

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

Example of PreparedStatement interface that retrieve the records of a table

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

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

Example of PreparedStatement to insert records until user press n

```
import java.sql.*;
import java.io.*;
class RS{
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");

PreparedStatement ps=con.prepareStatement("insert into emp130 va
lues(?,?,?)");

BufferedReader br=new BufferedReader(new InputStreamReader(S
ystem.in));

do{
System.out.println("enter id:");
int id=Integer.parseInt(br.readLine());
System.out.println("enter name:");
String name=br.readLine();
System.out.println("enter salary:");
float salary=Float.parseFloat(br.readLine());

ps.setInt(1,id);
ps.setString(2,name);
ps.setFloat(3,salary);
int i=ps.executeUpdate();
```

```
System.out.println(i+" records affected");

System.out.println("Do you want to continue: y/n");
String s=br.readLine();
if(s.startsWith("n")){
break;
}
}while(true);

con.close();
}}
```

The CallableStatement Objects:

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

CallableStatement Object:

Suppose, you need to execute the following Oracle stored procedure:

```
CREATE OR REPLACE PROCEDURE getEmpName
    (EMP_ID IN NUMBER, EMP_FIRST OUT VARCHAR)
AS
BEGIN
   SELECT first INTO EMP_FIRST
   FROM Employees
   WHERE ID = EMP_ID;
END;
```

NOTE: Above stored procedure has been written for Oracle, but we are working with MySQL database so let us write same stored procedure for MySQL as follows to create it in EMP database:

```
DELIMITER $$

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 ;
```

Three types of parameters exist: IN, OUT, and INOUT. The PreparedStatement object only uses the IN parameter. The CallableStatement object can use all 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) {
   . . .
}
```

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

Using CallableStatement objects is much like using PreparedStatement objects. You must bind values to all 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 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.

ResultSet interface

java.sql.ResultSet also an interface and is used to retrieve SQL select query results. A **ResultSet** object maintains a cursor pointing to its current row of data. Initially the cursor is positioned before the first row. The **next()** method moves the cursor to the next row, and because it returns **false** when there are no more rows in the **ResultSet** object, it can be used in a while loop to iterate through the result set. It provides **getXXX()** methods to get data from each iteration. Here XXX represents datatypes.

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 to 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 generated the ResultSet is created.

JDBC provides 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 indicate 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 RS Type 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 forwards and backwards, 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 forwards and backwards, 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.

Our all the 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 = con.createStatement(
```

```
ResultSet.TYPE_FORWARD_ONLY,

ResultSet.CONCUR_READ_ONLY);
}
catch(Exception ex) {
   ....
}
```

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:

```
try
{
   Statement stmt = con.createStatement(

ResultSet.TYPE_SCROLL_INSENSITIVE,

ResultSet.CONCUR_UPDATABLE);
}
catch(Exception ex) {
   ....
}
```

Navigating a Result Set:

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

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	is used to return the data of specified

getString(String columnName):	column name of the current row as String.

Updating a Result Set:

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.

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:1521:xe","system","oracle");
Statement stmt=con.createStatement(ResultSet.TYPE_SCROLL_
SENSITIVE,ResultSet.CONCUR_UPDATABLE);
ResultSet rs=stmt.executeQuery("select * from pers");

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

con.close();

}}
```

Commit & Rollback

If your JDBC Connection is in *auto-commit* mode, which it is by default, then every SQL statement is committed to the database upon its completion.

That may be fine for simple applications, but there are three reasons why you may want to turn off auto-commit and manage your own transactions:

- To increase performance
- To maintain the integrity of business processes

- To use distributed transactions

Transactions enable you to control if, and when, changes are applied to the database. It treats a single SQL statement or a group of SQL statements as one logical unit, and if any statement fails, the whole transaction fails.

To enable manual- transaction support instead of the *auto-commit* mode that the JDBC driver uses by default, use the Connection object's **setAutoCommit()** method. If you pass a boolean false to setAutoCommit( ), you turn off auto-commit. You can pass a boolean true to turn it back on again.

For example, if you have a Connection object named con, code the following to turn off auto-commit:

```
conn.setAutoCommit(false);
```

Once you are done with your changes and you want to commit the changes then call **commit()** method on connection object as follows:

```
conn.commit( );
```

Otherwise, to roll back updates to the database made using the Connection named conn, use the following code:

```
conn.rollback( );
```

Example:

```
try{
   //Assume a valid connection object conn
   conn.setAutoCommit(false);
   Statement stmt = conn.createStatement();
```

```
  String SQL = "INSERT INTO Employees  " +
               "VALUES (106, 20, 'Rita', 'Tez')";
  stmt.executeUpdate(SQL);
  //Submit a malformed SQL statement that breaks
  String SQL = "INSERTED IN Employees  " +
               "VALUES (107, 22, 'Sita', 'Singh')";
  stmt.executeUpdate(SQL);
  // If there is no error.
  conn.commit();
}catch(SQLException se){
  // If there is any error.
  conn.rollback();
}
```

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 of ResultSetMetaData. Syntax:

```
public ResultSetMetaData getMetaData()throws SQLExceptio
n
```

Example of ResultSetMetaData interface :

```
import java.sql.*;
class Rsmd{
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("select * from em
p");
ResultSet rs=ps.executeQuery();

ResultSetMetaData rsmd=rs.getMetaData();
```

```
System.out.println("Total columns: "+rsmd.getColumnCount()
);
System.out.println("Column Name of 1st column: "+rsmd.getC
olumnName(1));
System.out.println("Column Type Name of 1st column: "+rsm
d.getColumnTypeName(1));

con.close();

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

}
}
```

DatabaseMetaData interface:

DatabaseMetaData interface provides methods to get meta data of a database such as database product name, database product version, driver name, name of total number of tables, name of total number of views etc.

Commonly used methods of DatabaseMetaData interface

- **public String getDriverName()throws SQLException:** it returns the name of the JDBC driver.
- **public String getDriverVersion()throws SQLException:** it returns the version number of the JDBC driver.
- **public String getUserName()throws SQLException:** it returns the username of the database.
- **public String getDatabaseProductName()throws SQLException:** it returns the product name of the database.
- **public String getDatabaseProductVersion()throws**

SQLException: it returns the product version of the database.

- **public ResultSet getTables(String catalog, String schemaPattern, String tableNamePattern, String[] types)throws SQLException:** it returns the description of the tables of the specified catalog. The table type can be TABLE, VIEW, ALIAS, SYSTEM TABLE, SYNONYM etc.

How to get the object of DatabaseMetaData:

The getMetaData() method of Connection interface returns the object of DatabaseMetaData. Syntax:

1. public DatabaseMetaData getMetaData()throws SQLException

Simple Example of DatabaseMetaData interface :

```
import java.sql.*;
class Dbmd{
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");

DatabaseMetaData dbmd=con.getMetaData();

System.out.println("Driver Name: "+dbmd.getDriverName());
System.out.println("Driver Version: "+dbmd.getDriverVersion
());
System.out.println("UserName: "+dbmd.getUserName());
```

```
System.out.println("Database Product Name: "+dbmd.getData
baseProductName());
System.out.println("Database Product Version: "+dbmd.getDat
abaseProductVersion());

con.close();

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

}
}
```

Output:Driver Name: Oracle JDBC Driver
Driver Version: 10.2.0.1.0XE
Database Product Name: Oracle
Database Product Version: Oracle Database 10g Express Edition Release 10.2.0.1.0 -Production

Example of DatabaseMetaData interface that prints total number of tables :

```
import java.sql.*;
class Dbmd2{
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");

DatabaseMetaData dbmd=con.getMetaData();
String table[]={"TABLE"};
ResultSet rs=dbmd.getTables(null,null,null,table);
```

```
while(rs.next()){
System.out.println(rs.getString(3));
}

con.close();

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

}
}
```

Example of DatabaseMetaData interface that prints total number of views :

```
import java.sql.*;
class Dbmd3{
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");

DatabaseMetaData dbmd=con.getMetaData();
String table[]={"VIEW"};
ResultSet rs=dbmd.getTables(null,null,null,table);

while(rs.next()){
System.out.println(rs.getString(3));
}

con.close();
```

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

}
}
```

Write an example for batch update using Statement.

Batch update is nothing but executing set of queries at a time. Batch updates reduces number of database calls. In batch processing, batch should not contain select query. You can add queries by calling addBatch() method and can execute the bunch of queries by calling executeBatch() method. When using batch updates with Statement object, you can use multiple types of queries which can be acceptable in executeUpdate() method.

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

public class MyStatementBatchUpdate {

   public static void main(String a[]){

      Connection con = null;
      Statement st = null;
      try {
         Class.forName("oracle.jdbc.driver.OracleDriver");
         con = DriverManager.
```

```
            getConnection("jdbc:oracle:thin:@<hostname>:<port
num>:<DB name>"
                ,"user","password");
            con.setAutoCommit(false);
            st = con.createStatement();
            st.addBatch("update emp set sal=3000 where empid=200");
            st.addBatch("insert into emp values (100,4000)");
            st.addBatch("update emp set emp name='Ram' where
empid=345");
            int count[] = st.executeBatch();
            for(int i=1;i<=count.length;i++){
               System.out.println("Query "+i+" has effected "+count[i]+"
times");
            }
            con.commit();
         } catch (ClassNotFoundException e) {
            try {
               con.rollback();
            } catch (SQLException e1) {
               e1.printStackTrace();
            }
            e.printStackTrace();
         } catch (SQLException e) {
            try {
               con.rollback();
            } catch (SQLException e1) {
               e1.printStackTrace();
            }
            e.printStackTrace();
         } finally{
            try{
               if(st != null) st.close();
               if(con != null) con.close();
```

```
            } catch(Exception ex){}
        }
    }
}
```

Example to store image in Oracle database

You can store images in the database in java by the help of **PreparedStatement** interface.

The **setBinaryStream()** method of PreparedStatement is used to set Binary information into the parameterIndex.

Signature of setBinaryStream method

The syntax of setBinaryStream() method is given below:

1. public void setBinaryStream(int paramIndex,InputStream stream) throws SQLException
2. public void setBinaryStream(int paramIndex,InputStream stream,long length) throws SQLException.

For storing image into the database, BLOB (Binary Large Object) datatype is used in the table. For example:

```
CREATE TABLE  "IMGTABLE"
  (    "NAME" VARCHAR2(4000),
   "PHOTO" BLOB
  )
/
```

Let's write the jdbc code to store the image in the database. Here we are using d:\\d.jpg for the location of image. You can change it according to the image location.

```
import java.sql.*;
import java.io.*;
public class InsertImage {
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 imgta
ble values(?,?)");
ps.setString(1,"sonoo");

FileInputStream fin=new FileInputStream("d:\\g.jpg");
ps.setBinaryStream(2,fin,fin.available());
int i=ps.executeUpdate();
System.out.println(i+" records affected");

con.close();
}catch (Exception e) {e.printStackTrace();}
}
}
```

If you see the table, record is stored in the database but image will not be shown. To do so, you need to retrieve the image from the database which we are covering in the next page.

Example to retrieve image from Oracle database

By the help of **PreparedStatement** we can retrieve and store the image in the database.

The **getBlob()** method of PreparedStatement is used to get Binary information, it returns the instance of Blob. After calling the **getBytes()** method on the blob object, we can get the array of binary information that can be written into the image file.

Signature of getBlob() method of PreparedStatement

```
public Blob getBlob()throws SQLException
```

Signature of getBytes() method of Blob interface

```
public  byte[] getBytes(long pos, int length)throws SQLException
```

We are assuming that image is stored in the imgtable.

```
CREATE TABLE  "IMGTABLE"
  (    "NAME" VARCHAR2(4000),
   "PHOTO" BLOB
  )
/
```

Now let's write the code to retrieve the image from the database and write it into the directory so that it can be displayed.

In AWT, it can be displayed by the Toolkit class. In servlet, jsp, or html it can be displayed by the img tag.

```
import java.sql.*;
import java.io.*;
public class RetrieveImage {
```

```
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("select * from im
gtable");
ResultSet rs=ps.executeQuery();
if(rs.next()){//now on 1st row

Blob b=rs.getBlob(2);//2 means 2nd column data
byte barr[]=b.getBytes(1,(int)b.length());//1 means first image

FileOutputStream fout=new FileOutputStream("d:\\sonoo.jpg")
;
fout.write(barr);

fout.close();
}//end of if
System.out.println("ok");

con.close();
}catch (Exception e) {e.printStackTrace();  }
}
}
```

Now if you see the d drive, sonoo.jpg image is created.