UNIT-VIII

Introduction of Java Database Connectivity

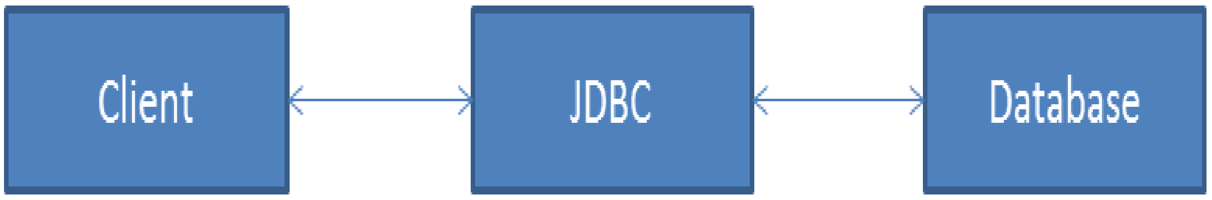
JDBC - Java Database Connectivity.

JDBC provides API or Protocol to interact with different databases.

With the help of JDBC driver we can connect with different types of databases.

Driver is must needed for connection establishment with any database.

A driver works as an interface between the client and a database server.



JDBC have so many classes and interfaces that allow a java application to send request made by user to any specific DBMS(Data Base Management System).

JDBC supports a wide level of portability.

JDBC provides interfaces that are compatible with java application.

components and specification of JDBC:

Components of JDBC:

JDBC has four main components as under and with the help of these components java application can connect with database.

The JDBC API - it provides various methods and interfaces for easy communication with database.

The JDBC DriverManager - it loads database specific drivers in an application to establish connection with database.

The JDBC test suite - it will be used to test an operation being performed by JDBC drivers.

The JDBC-ODBC bridge - it connects database drivers to the database.

JDBC Specification:

Different version of JDBC has different specification as under.

JDBC 1.0 - it provides basic functionality of JDBC

JDBC 2.0 - it provides JDBC API(JDBC 2.0 Core API and JDBC 2.0 Optional Package API).

JDBC 3.0 - it provides classes and interfaces in two packages(java.sql and javax.sql).

JDBC 4.0 - it provides so many extra features like

Auto loading of the driver interface.

Connection management

ROWID data type support.

Enhanced support for large object like BLOB(Binary Large Object) and CLOB(Character Large Object).

What Does JDBC Do?

Simply put, JDBC makes it possible to do three things:

1. establish a connection with a database

2. send SQL statements
3. process the results.

JavaSoft provides three JDBC product components as part of the Java Developer's Kit (JDK):

1. . the JDBC driver manager,
2. . the JDBC driver test suite, and
3. . the JDBC-ODBC bridge.

The JDBC driver manager is the backbone of the JDB architecture. It actually is quite small and simple; its primary function is to connect Java applications to the correct JDBC driver and then get out of the way

JDBC Architecture:

As we all know now that driver is required to communicate with database.

JDBC API provides classes and interfaces to handle request made by user and response made by database.

Some of the important JDBC API are as under.

DriverManager

Driver

Connection

Statement

PreparedStatement

CallableStatement

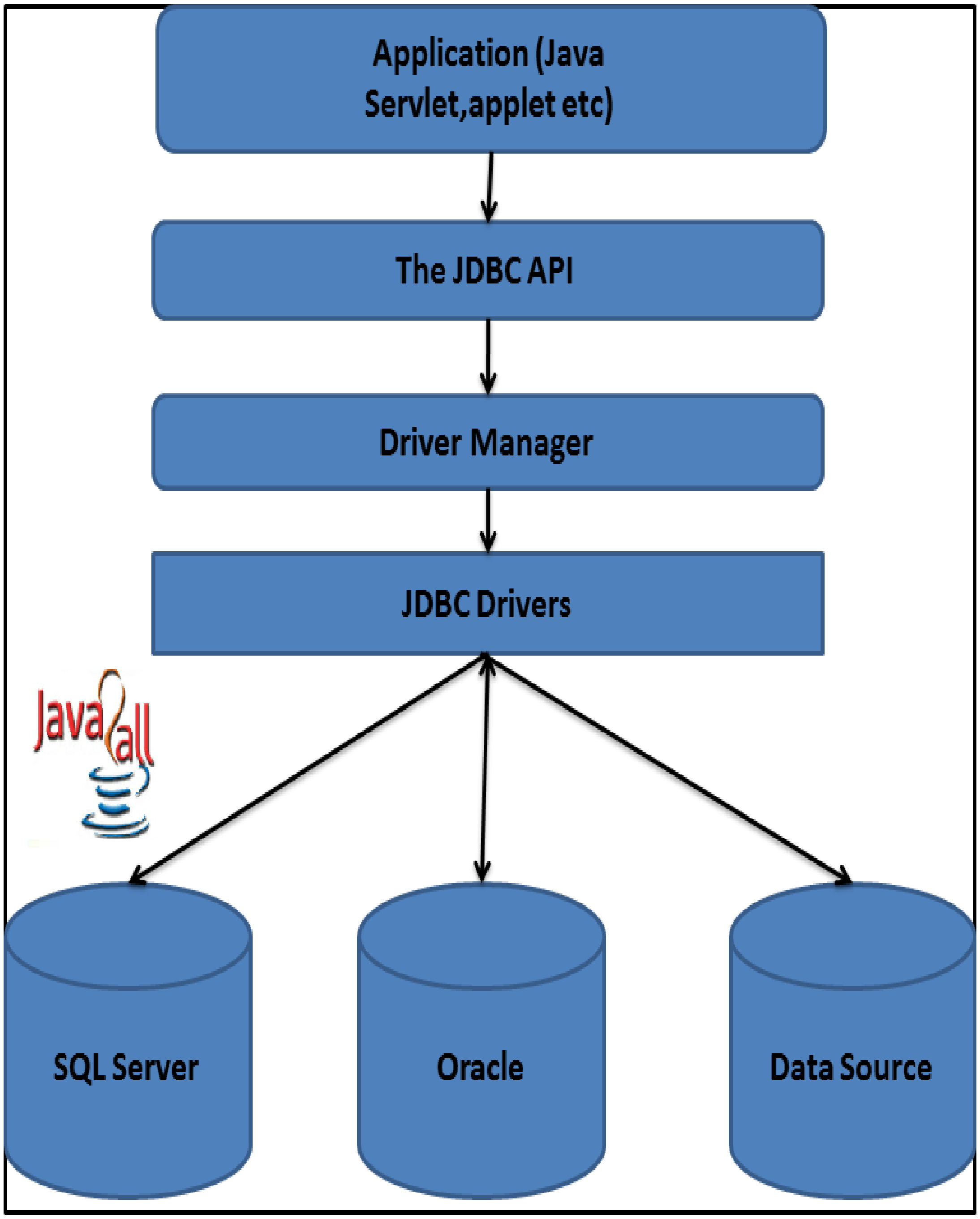
ResultSet

DatabaseMetaData

ResultSetMetaData

Here The DriverManager plays an important role in JDBC architecture.

It uses some database specific drivers to communicate our J2EE application to database.



As per the diagram first of all we have to program our application with JDBC API.

With the help of DriverManager class than we connect to a specific database with the help of spcific database driver.

Java drivers require some library to communicate with the database.

We have four different types of java drivers.

We will learn all that four drivers with architecture in next chapter.

Some drivers are pure java drivers and some are partial.

So with this kind of JDBC architecture we can communicate with specific database.

<u>JDBC Drivers:</u>

JDBC Driver Types:

There are four categories of drivers by which developer can apply a connection between Client (The JAVA application or an applet) to a DBMS.

(1) Type 1 Driver : JDBC-ODBC Bridge.

(2) Type 2 Driver : Native-API Driver (Partly Java driver).

(3) Type 3 Driver : Network-Protocol Driver (Pure Java driver for database Middleware).

(4) Type 4 Driver : Native-Protocol Driver (Pure Java driver directly connected to database).

(1) Type 1 Driver: JDBC-ODBC Bridge :-

The JDBC type 1 driver which is also known as a JDBC-ODBC Bridge is a convert JDBC methods into ODBC function calls.

SunprovidesJDBC-ODBCBridge driver by "sun.jdbc.odbc.JdbcOdbcDriver".

The driver is a platform dependent because it uses ODBC which is depends on native libraries of the operating system and also the driver needs other installation for example, ODBC must be installed on the computer and the database must support ODBC driver.

Type 1 is the simplest compare to all other driver but it's a platform specific i.e. only on Microsoft platform.

For type-1 we create the DSN name.

Steps for DSN:

1. Start

2. Control panel
3. Administrator tools
4. Data source odbc
5. System dsn
6. Add
7. Microsoft oracle for odbc
8. Finish
9. ora

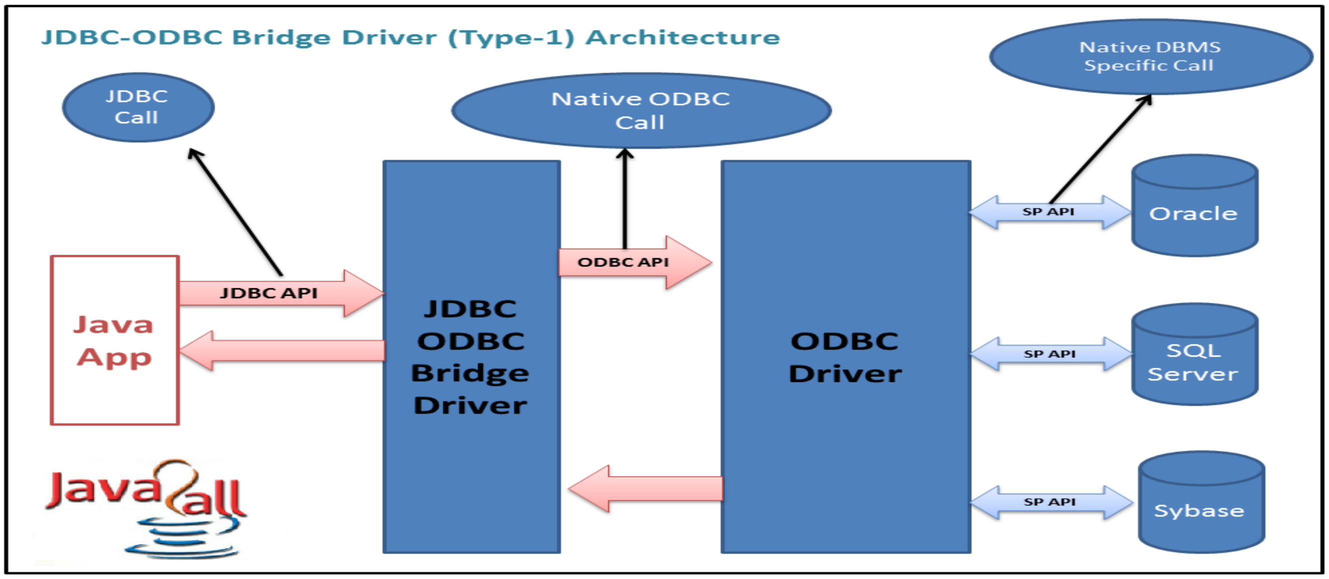
The JDBC-ODBC Bridge is use only when there is no PURE-JAVA driver available for a particular database.

The driver is a platform dependent because it uses ODBC which is depends on native libraries of the operating system and also the driver needs other installation for example, ODBC must be installed on the computer and the database must support ODBC driver.

Java DB is Oracle's supported distribution of the open source Apache Derby database. Its ease of use, standards compliance, full feature set, and small footprint make it the ideal database for Java developers. Java DB is written in the Java programming language, providing "write once, run anywhere" portability. It can be embedded in Java applications, requiring zero administration by the developer or user. It can also be used in client server mode. Java DB is fully transactional and provides a standard SQL interface.

The JDBC driver manager is the backbone of the JDB architecture. It actually is quite small and simple; its primary function is to connect Java applications to the correct JDBC driver and then get out of the way

Architecture Diagram:



Process:

Java Application → JDBC APIs → JDBC Driver Manager → Type 1 Driver → ODBC Driver → Database library APIs → Database

Advantage:

(1) Connect to almost any database on any system, for which ODBC driver is installed.

(2) It's an easy for installation as well as easy(simplest) to use as compare the all other driver.

Disadvantage:

(1) The ODBC Driver needs to be installed on the client machine.

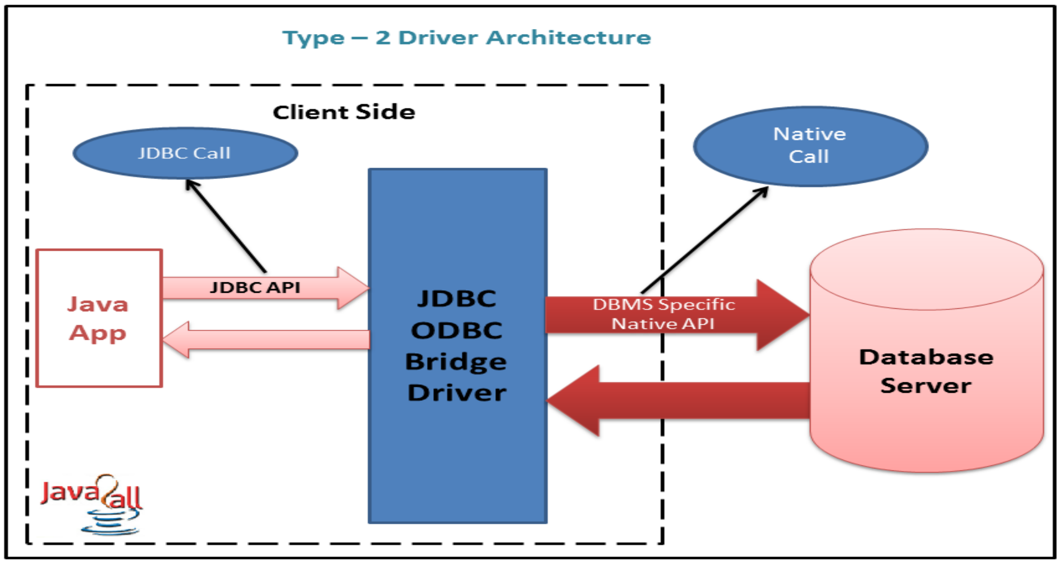
(2) It's a not a purely platform independent because its use ODBC which is depends on native libraries of the operating system on client machine.

(3) Not suitable for applets because the ODBC driver needs to be installed on the client machine.

(2) Type 2 Driver: Native-API Driver (Partly Java driver) :-

The JDBC type 2 driver is uses the libraries of the database which is available at client side and this driver converts the JDBC method calls into native calls of the database so this driver is also known as a Native-API driver.

Architecture Diagram :



Process:

Java Application → JDBC APIs → JDBC Driver Manager → Type 2 Driver → Vendor Client Database library APIs → Database

Advantage:

(1) There is no implantation of JDBC-ODBC Bridge so it's faster than a type 1 driver; hence the performance is better as compare the type 1 driver (JDBC-ODBC Bridge).

Disadvantage:

(1) On the client machine require the extra installation because this driver uses the vendor client libraries.

(2) The Client side software needed so cannot use such type of driver in the web-based application.

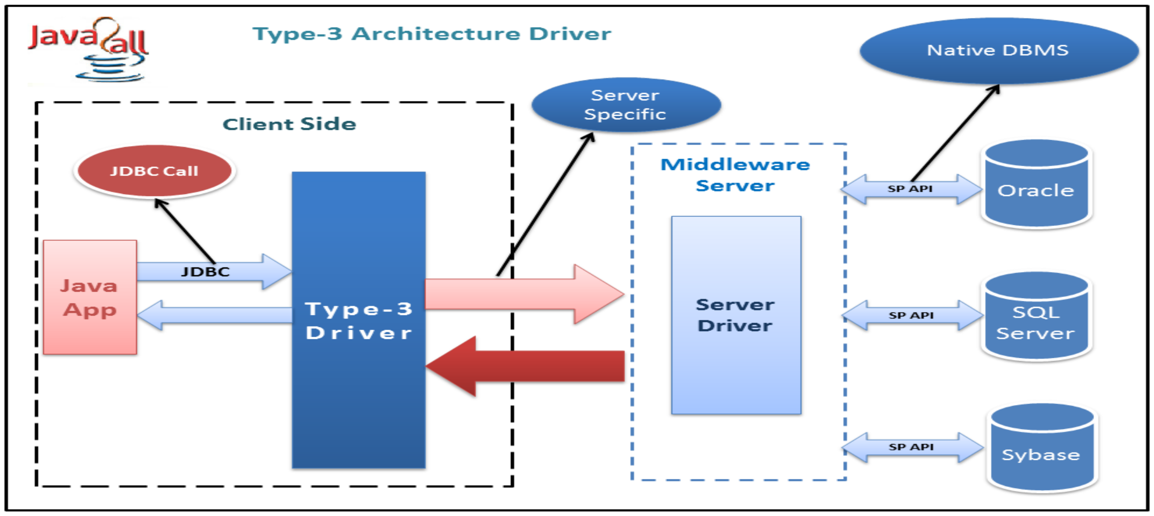
(3) Not all databases have the client side library.

(4) This driver supports all JAVA applications except applets.

(3) Type 3 Driver: Network-Protocol Driver (Pure Java driver for database Middleware) :-

The JDBC type 3 driver uses the middle tier(application server) between the calling program and the database and this middle tier converts JDBC method calls into the vendor specific database protocol and the same driver can be used for multiple databases also so it's also known as a Network-Protocol driver as well as a JAVA driver for database middleware.

Architecture Diagram:



Process:

Java Application → JDBC APIs → JDBC Driver Manager → Type 3 Driver → Middleware (Server)→ any Database

Advantage:

(1) There is no need for the vendor database library on the client machine because the middleware is database independent and it communicates with client.

(2) Type 3 driver can be used in any web application as well as on internet also because there is no any software require at client side.

(3) A single driver can handle any database at client side so there is no need a separate driver for each database.

(4) The middleware server can also provide the typical services such as connections, auditing, load balancing, logging etc.

Disadvantage:

(1) An Extra layer added, may be time consuming.

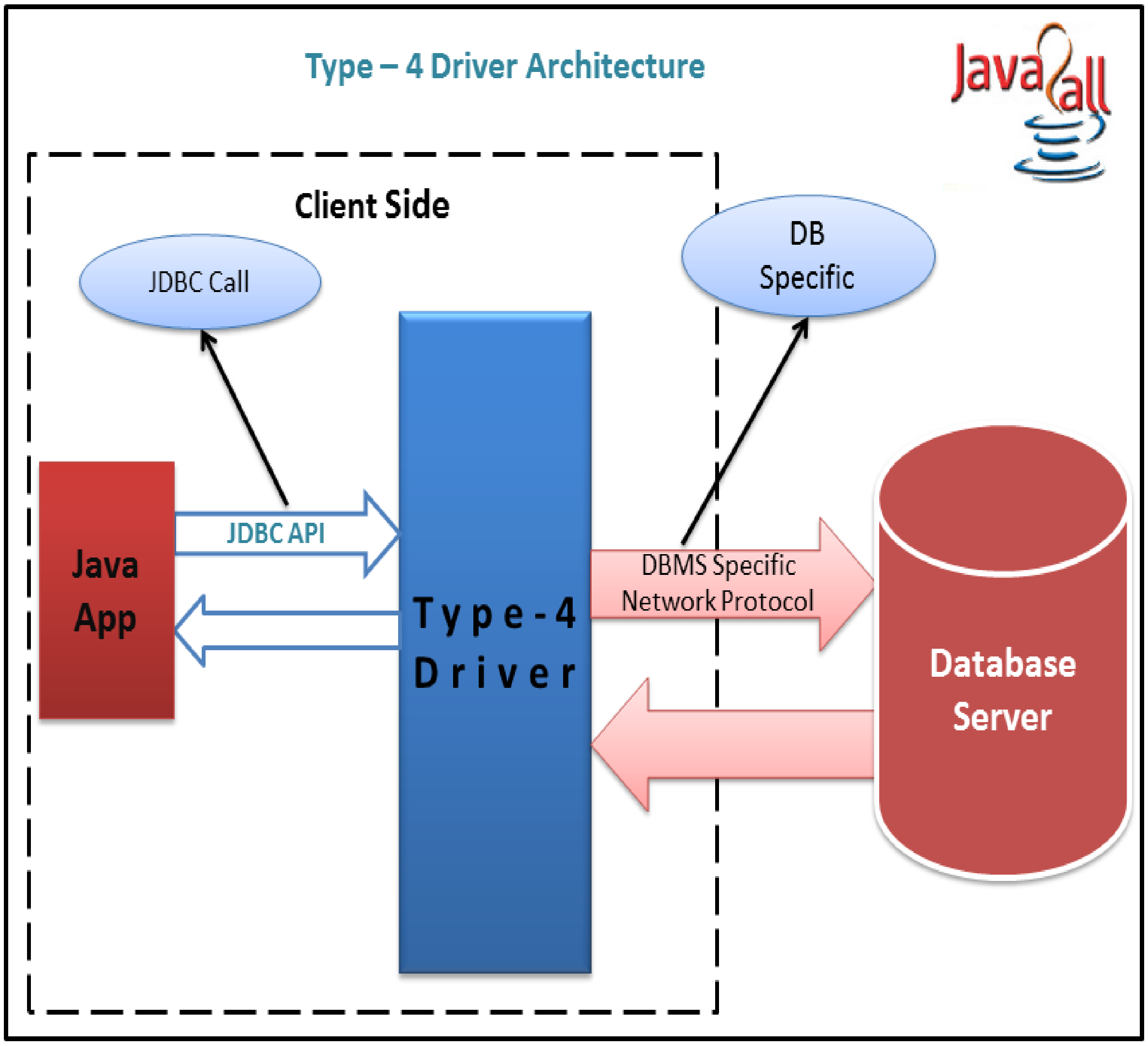
(2) At the middleware develop the database specific coding, may be increase complexity.

(4) Type 4 Driver: Native-Protocol Driver (Pure Java driver directly connected to database) :-

The JDBC type 4 driver converts JDBC method calls directly into the vendor specific database protocol and in between do not need to be converted any other formatted system so this is the fastest way to communicate quires to DBMS and it is completely written in JAVA because of that this is also known as the “direct to database Pure JAVA driver”.

If we are using type-4 driver in oracle then we need to add jar file to the class path because it was given by third party.

Architecture Diagram:



Process:

Java Application → JDBC APIs → JDBC Driver Manager → Type 4 Driver (Pure JAVA Driver) → Database Server

Advantage:

(1) It’s a 100% pure JAVA Driver so it’s a platform independence.

(2) No translation or middleware layers are used so consider as a faster than other drivers.

(3) The all process of the application-to-database connection can manage by JVM so the debugging is also managed easily.

Disadvantage:

(1)There is a separate driver needed for each database at the client side.

(2) Drivers are Database dependent, as different database vendors use different network protocols.

JDBC APIs:

If any java application or an applet wants to connect with a database then there are various classes and interfaces available in java.sql package.

Depending on the requirements these classes and interfaces can be used.

Some of them are list out the below which are used to perform the various tasks with database as well as for connection.

Class or Interface	**Description**
Java.sql.Connection	Create a connection with specific database
Java.sql.DriverManager	The task of DriverManager is to manage the database driver
Java.sql.Statement	It executes SQL statements for particular connection and retrieve the results
Java.sql.PreparedStatement	It allows the programmer to create prepared SQL statements
Java.sql.CallableStatement	It executes stored procedures
Java.sql.ResultSet	This interface provides methods to get result row by row generated by SELECT statements

The Connection interface:

The Connection interface used to connect java application with particular database.

After crating the connection with database we can execute SQL statements for that particular connection using object of Connection and retrieve the results.

The interface has few methods that makes changes to the database temporary or permanently.

The some methods are as given below.

Method	Description
void close()	This method frees an object of type Connection from database and other JDBC resources.
void commit()	This method makes all the changes made since the last commit or rollback permanent. It throws SQLExeception.
Statement createStatement()	This method creates an object of type Statement for sending SQL statements to the database. It throws SQLExeception.
boolean isClosed()	Return true if the connection is close else return false.
CallableStatement prepareCall(String s)	This method creates an object of type CallableStatement for calling the stored procedures from database. It throws SQLExeception.
PreparedStatement prepareStatement(String s)	This method creates an object of type PrepareStatement for sending dynamic (with or without IN parameter) SQL statements to the database. It throws SQLExeception.
void rollback()	This method undoes all changes made to the database.

Statement Interface:

The Statement interface is used for to execute a static query.

It’s a very simple and easy so it also calls a “**Simple Statement**”.

The statement interface has several methods for execute the SQL statements and also get the appropriate result as per the query sent to the database.

Some of the most common methods are as given below

Method	Description
void close()	This method frees an object of type Statement from database and other JDBC resources.
boolean execute(String s)	This method executes the SQL statement specified by s. The getResultSet() method is used to retrieve the result.
ResultSet getResultet()	This method retrieves the ResultSet that is generated by the execute() method.
ResultSet executeQuery(String s)	This method is used to execute the SQL statement specified by s and returns the object of type ResultSet.
int getMaxRows()	This method returns the maximum number of rows those are generated by the executeQuery() method.
Int executeUpdate(String s)	This method executes the SQL statement specified by s. The SQL statement may be a SQL insert, update and delete statement.

The Prepared Statement Interface:

The Prepared Statement interface is used to execute a dynamic query (parameterized SQL statement) with IN parameter.

IN Parameter:-

In some situation where we need to pass different values to an query then such values can be specified as a “?” in the query and the actual values can be passed using the setXXX() method at the time of execution.

Syntax :

```
setXXX(integer data ,XXX value);
```

Where XXX means a data type as per the value we want to pass in the query.

For example,

```
String query = "Select * from Data where ID = ? and Name = ? ";

PreparedStatement ps = con.prepareStatement(query);

    ps.setInt(1, 1);

    ps.setString(2, "Ashutosh Abhangi");
```

The Prepared statement interface has several methods to execute the parameterized SQL statements and retrieve appropriate result as per the query sent to the database.

Some of the most common methods are as given below

Method	Description
void close()	This method frees an object of type Prepared Statement from database and other JDBC resources.
boolean execute()	This method executes the dynamic query in the object of type **Prepared Statement**.The getResult() method is used to retrieve the result.
ResultSet executeQuery()	This method is used to execute the dynamic query in the object of type **Prepared Statement** and returns the object of type ResultSet.
Int executeUpdate()	This method executes the SQL statement in the object of type **Prepared Statement**. The SQL statement may be a SQL insert, update and delete statement.
ResultSetMetaData getMetaData()	The ResultSetMetaData means a deta about the data of ResultSet.This method retrieves an

	object of type ResultSetMetaData that contains information about the columns of the ResultSet object that will be return when a query is execute.
int getMaxRows()	This method returns the maximum number of rows those are generated by the executeQuery() method.

The JDBC API

The JDBC API is based mainly on a set of interfaces, not classes. It's up to the manufacturer of the driver to implement the interfaces with their own set of classes.

A class diagram that shows the basic JDBC classes and interfaces; these make up the core API. Notice that the only concrete class is DriverManager. The rest of the core API is a set of interfaces.

The interfaces of the core JDBC API

DriverManager is used to load a JDBC Driver. A Driver is a software vendor's implementation of the JDBC API. After a driver is loaded, DriverManager is used to get a Connection.

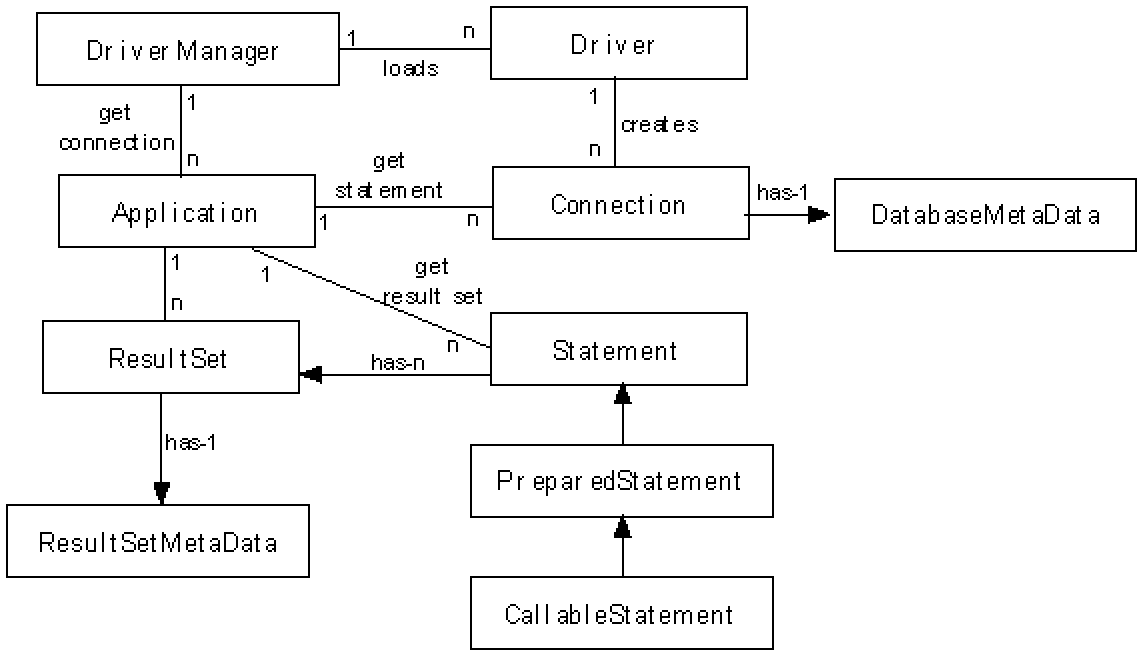
In turn, a Connection is used to create a Statement, or to create and prepare a PreparedStatement or CallableStatement.

Statement and PreparedStatement objects are used to execute SQL statements. CallableStatement objects are used to execute stored procedures.
A Connection can also be used to get a DatabaseMetaData object describing a database's functionality.

The results of executing a SQL statement using a Statement or PreparedStatement are returned as a ResultSet.

A ResultSet can be used to get the actual returned data or a ResultSetMetaData object that can be queried to identify the types of data returned in the ResultSet.

A Struct is a weakly typed object that represents a database object as a record.
A Ref is a reference to an object in a database. It can be used to get to a database object. An Array is a weakly typed object that represents a database collection object as an array. The SQLData interface is implemented by custom classes you write to represent database objects as Java objects in your application. SQLInput and SQLOutput are used by the Driver in the creation of your custom classes during retrieval and storage.

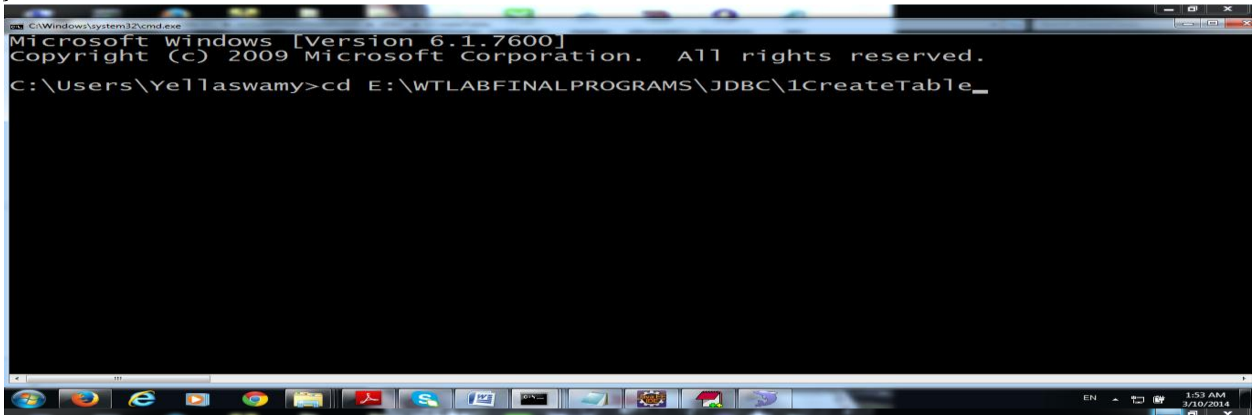

DriverManager
Driver
1
n
loads
1
n
creates
get
connection
1
n
Application
get
statement
1
n
Connection
has-1
DatabaseMetaData
1
n
1
get
result set
n
ResultSet
has-n
Statement
has-1
PreparedStatement
ResultSetMetaData
CallableStatement

UNIT-8
DATABASE ACCESS USING JDBC

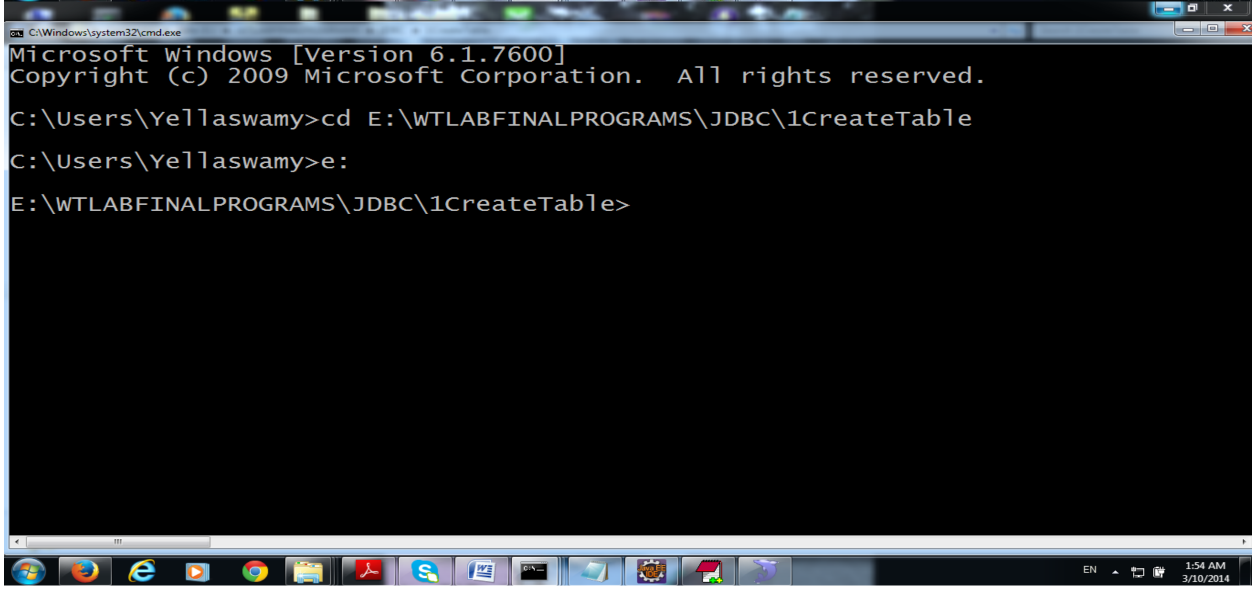
1) Create a table using JDBC

```
//create a Table using Oracle Type4 Driver/MySQL Driver

//Working with Statement Interface
package com.yellaswamy.jdbcexamples;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;
public class CreateTable
{
public static void main(String args[])throws SQLException,ClassNotFoundException
{
        //Get Connection
        Connection con=prepareConnection();

        //obtain a statement
        Statement st=con.createStatement();
        //for oracle
        String query="create table mytable1(col1 varchar2(20),col2 number,col3 number(10,2))";
        //for MySQL
        //String query="create table mytable1(col1 varchar(20),col2 numeric,col3
numeric(10,2))";

        //Execute the query
        st.executeUpdate(query);
        System.out.println("Table created Successfully");


}
public static Connection prepareConnection() throws SQLException,ClassNotFoundException
{
        // TODO Auto-generated method stub

        //Oracle Database
        String driverClassName="oracle.jdbc.driver.OracleDriver";
        String url="jdbc:oracle:thin:@localhost:1521:xe";
        String userName="system";
        String password="manager";

        //for MySQL Database
        /*String driverClassName="com.mysql.jdbc.Driver";
        String url="jdbc:mysql://localhost:3306/cmrcetdb";
```

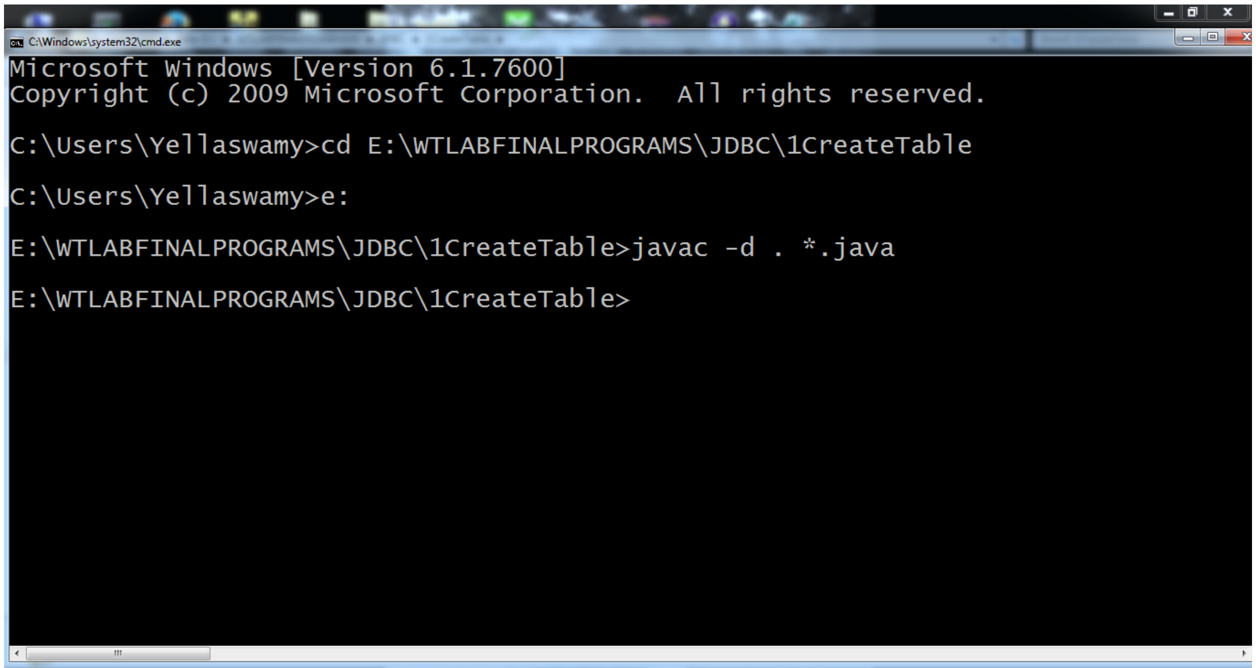
```
		String userName="root";
		String password="root";*/

		//load driver class
		Class.forName(driverClassName);
		//obtain a connection
		Connection conn=DriverManager.getConnection(url,userName,password);

		return conn;
	}
}
```



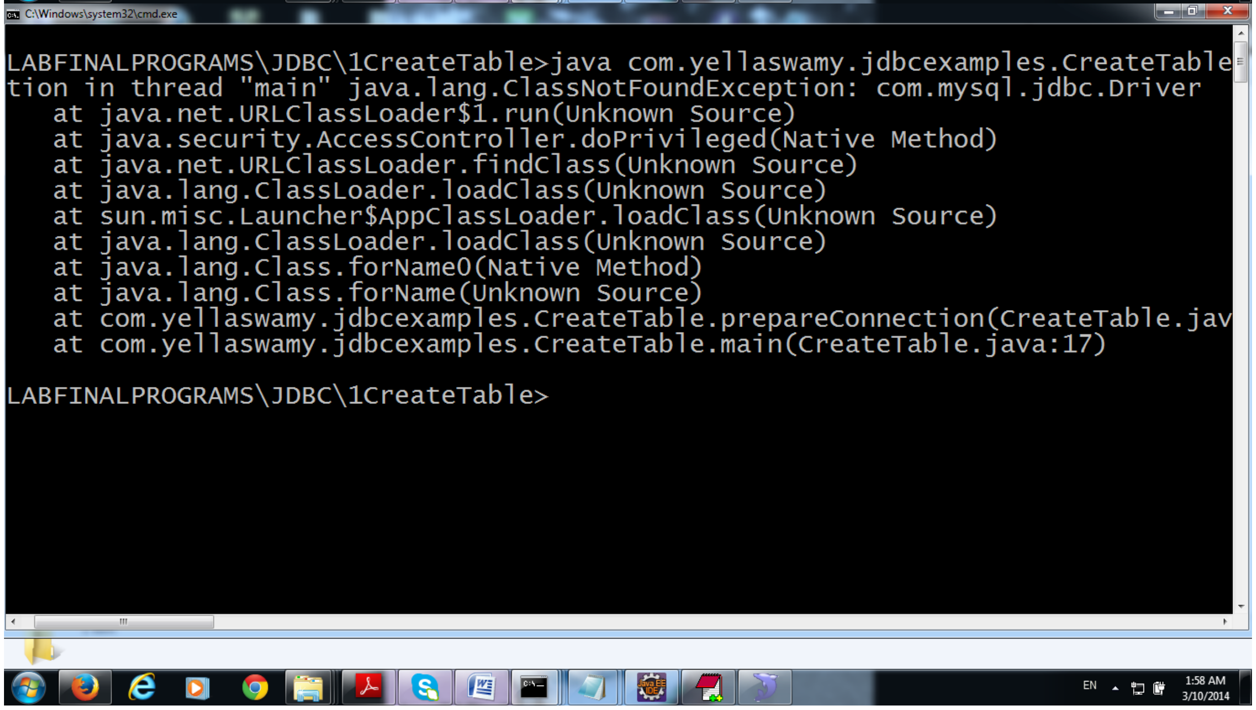
```
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.
C:\Users\Yellaswamy>cd E:\WTLABFINALPROGRAMS\JDBC\1CreateTable_
```



```
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\Yellaswamy>cd E:\WTLABFINALPROGRAMS\JDBC\1CreateTable

C:\Users\Yellaswamy>e:

E:\WTLABFINALPROGRAMS\JDBC\1CreateTable>
```

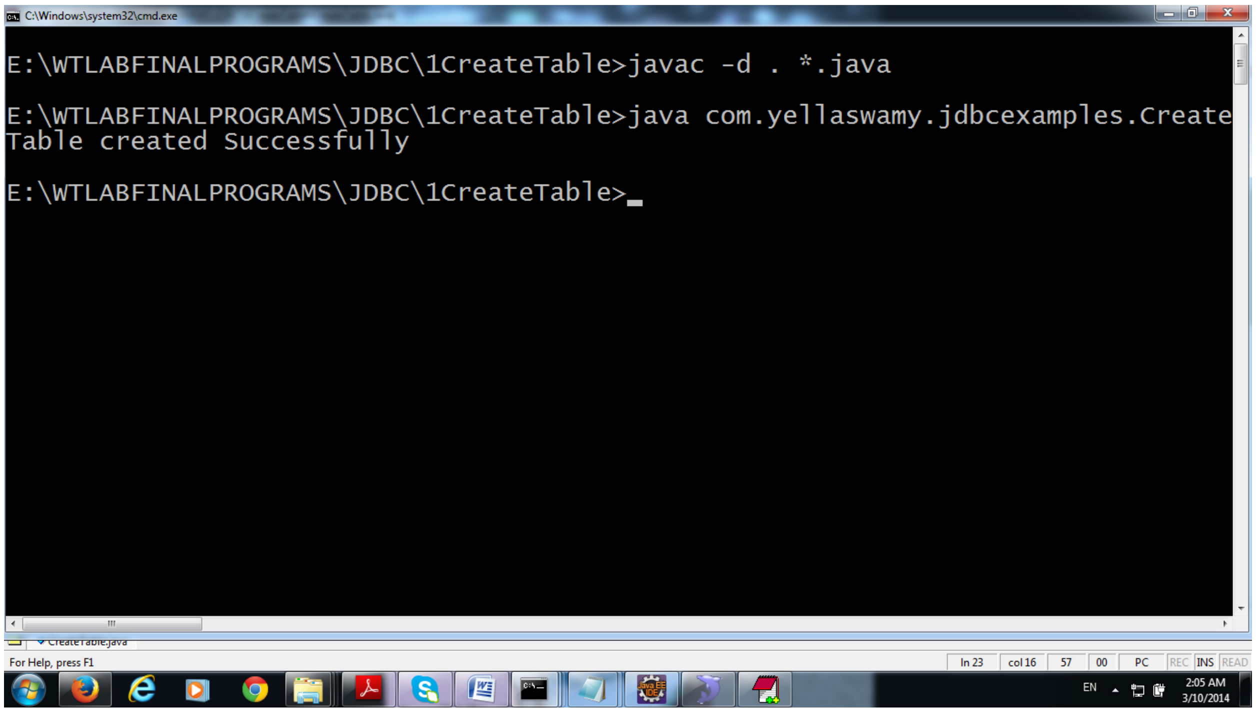


```
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\Yellaswamy>cd E:\WTLABFINALPROGRAMS\JDBC\1CreateTable

C:\Users\Yellaswamy>e:

E:\WTLABFINALPROGRAMS\JDBC\1CreateTable>javac -d . *.java

E:\WTLABFINALPROGRAMS\JDBC\1CreateTable>
```



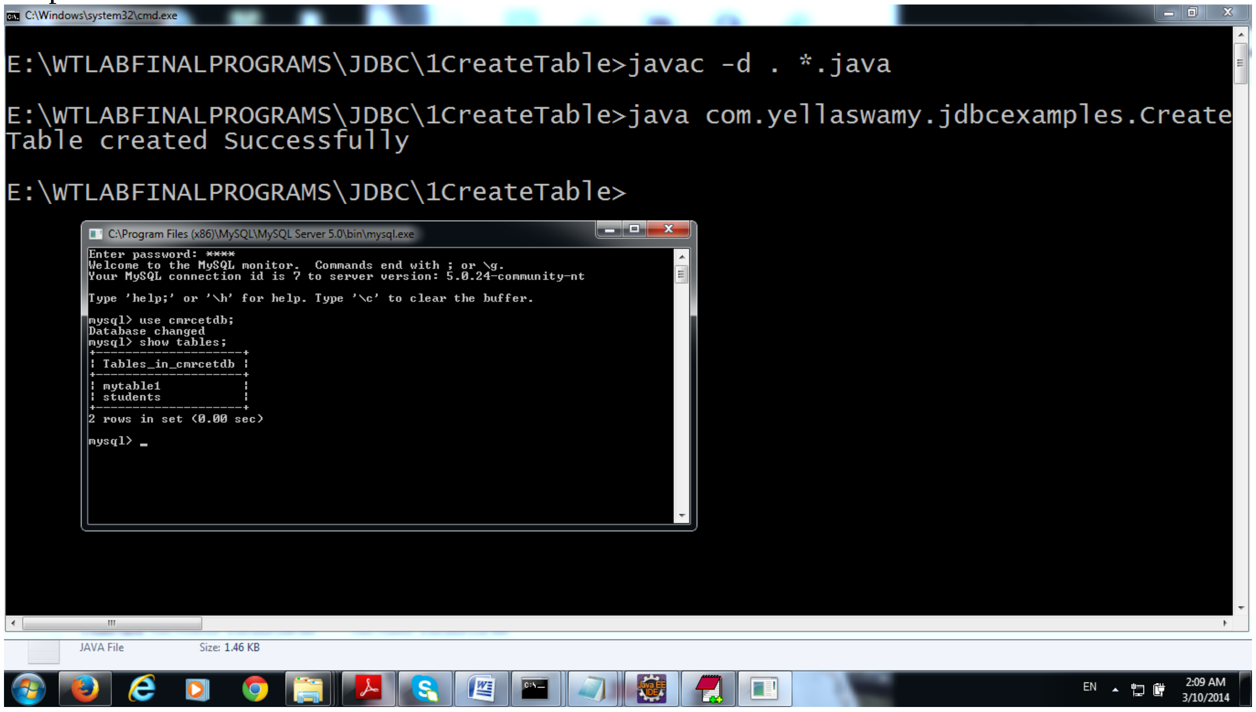
```
LABFINALPROGRAMS\JDBC\1CreateTable>java com.yellaswamy.jdbcexamples.CreateTable
tion in thread "main" java.lang.ClassNotFoundException: com.mysql.jdbc.Driver
      at java.net.URLClassLoader$1.run(Unknown Source)
      at java.security.AccessController.doPrivileged(Native Method)
      at java.net.URLClassLoader.findClass(Unknown Source)
      at java.lang.ClassLoader.loadClass(Unknown Source)
      at sun.misc.Launcher$AppClassLoader.loadClass(Unknown Source)
      at java.lang.ClassLoader.loadClass(Unknown Source)
      at java.lang.Class.forName0(Native Method)
      at java.lang.Class.forName(Unknown Source)
      at com.yellaswamy.jdbcexamples.CreateTable.prepareConnection(CreateTable.jav
      at com.yellaswamy.jdbcexamples.CreateTable.main(CreateTable.java:17)

LABFINALPROGRAMS\JDBC\1CreateTable>
```

E:\WTLABFINALPROGRAMS\JDBC\1CreateTable>set classpath=E:\WTLABFINALPROGRAMS\JDBC\lib\mysql-connector-java-3.0.11-stable-bin.jar;.;



```
E:\WTLABFINALPROGRAMS\JDBC\1CreateTable>javac -d . *.java

E:\WTLABFINALPROGRAMS\JDBC\1CreateTable>java com.yellaswamy.jdbcexamples.Create
Table created Successfully

E:\WTLABFINALPROGRAMS\JDBC\1CreateTable>_
```

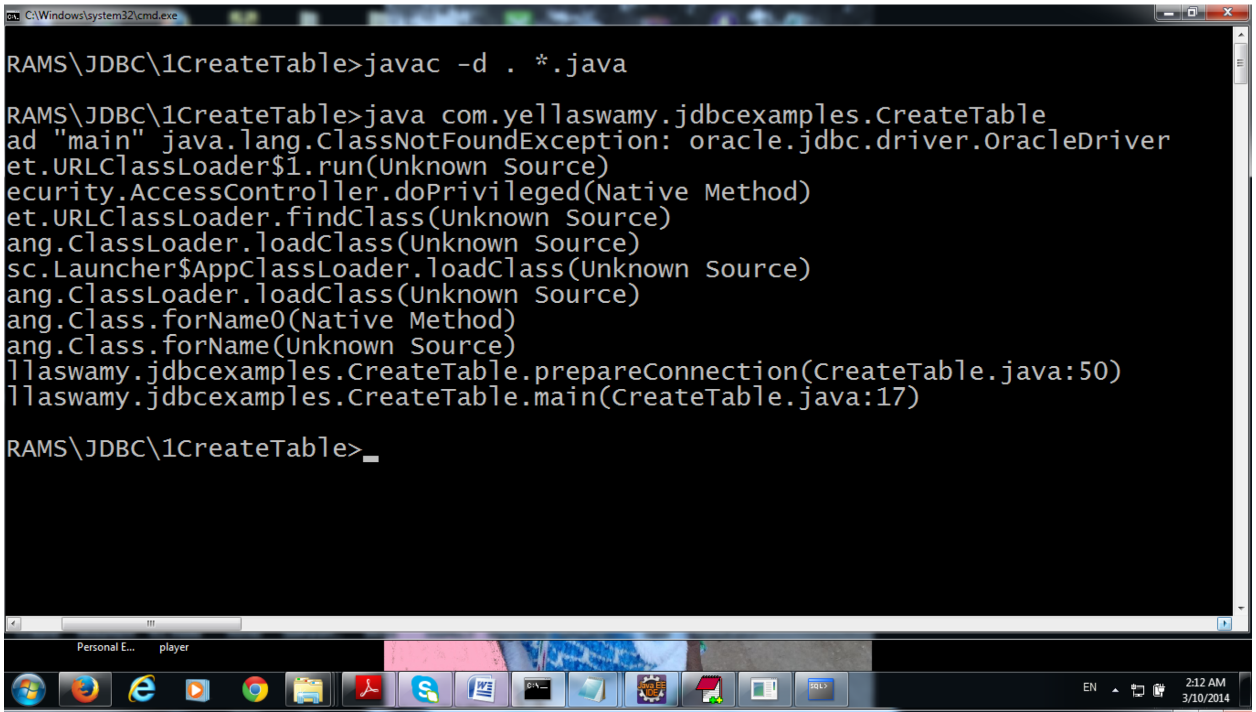
Output:



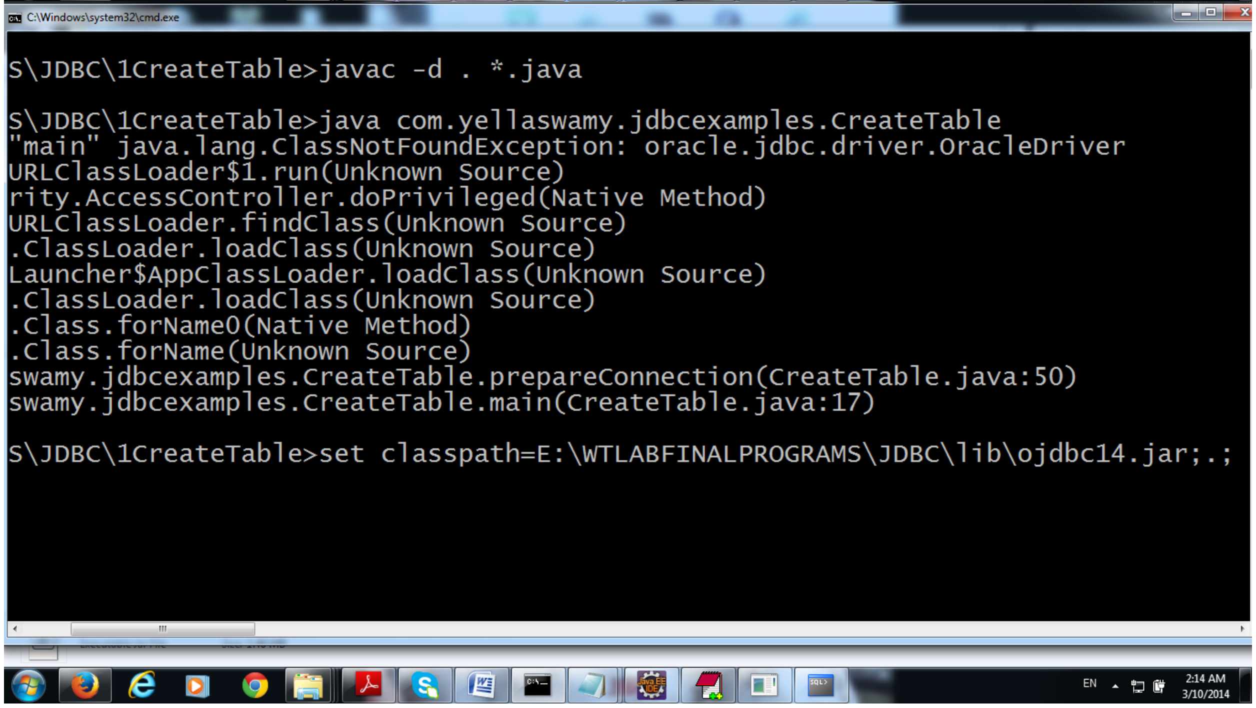
```
E:\WTLABFINALPROGRAMS\JDBC\1CreateTable>javac -d . *.java

E:\WTLABFINALPROGRAMS\JDBC\1CreateTable>java com.yellaswamy.jdbcexamples.Create
Table created Successfully

E:\WTLABFINALPROGRAMS\JDBC\1CreateTable>
```

```
Enter password: ****
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 7 to server version: 5.0.24-community-nt

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> use cmrcetdb;
Database changed
mysql> show tables;
+--------------------+
| Tables_in_cmrcetdb |
+--------------------+
| mytable1           |
| students           |
+--------------------+
2 rows in set (0.00 sec)

mysql> _
```

//For Connecting to Oracle Database

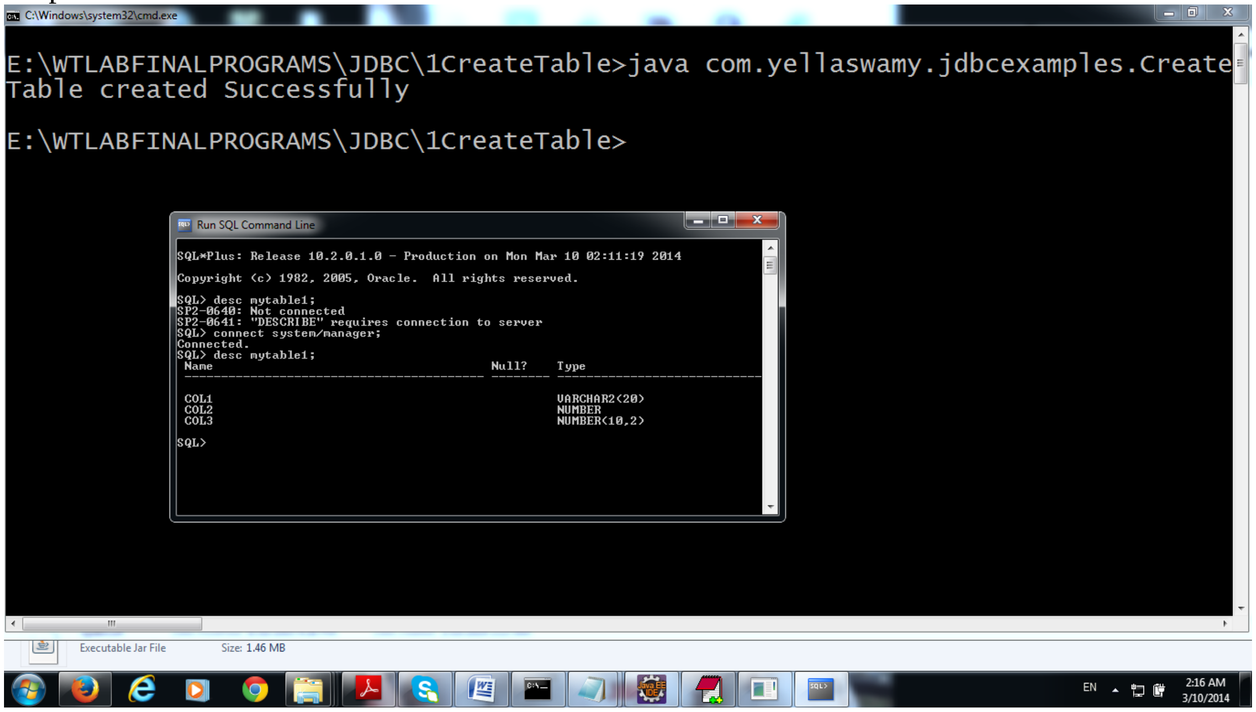


```
RAMS\JDBC\1CreateTable>javac -d . *.java

RAMS\JDBC\1CreateTable>java com.yellaswamy.jdbcexamples.CreateTable
ad "main" java.lang.ClassNotFoundException: oracle.jdbc.driver.OracleDriver
et.URLClassLoader$1.run(Unknown Source)
ecurity.AccessController.doPrivileged(Native Method)
et.URLClassLoader.findClass(Unknown Source)
ang.ClassLoader.loadClass(Unknown Source)
sc.Launcher$AppClassLoader.loadClass(Unknown Source)
ang.ClassLoader.loadClass(Unknown Source)
ang.Class.forName0(Native Method)
ang.Class.forName(Unknown Source)
llaswamy.jdbcexamples.CreateTable.prepareConnection(CreateTable.java:50)
llaswamy.jdbcexamples.CreateTable.main(CreateTable.java:17)

RAMS\JDBC\1CreateTable>_
```



```
S\JDBC\1CreateTable>javac -d . *.java

S\JDBC\1CreateTable>java com.yellaswamy.jdbcexamples.CreateTable
"main" java.lang.ClassNotFoundException: oracle.jdbc.driver.OracleDriver
URLClassLoader$1.run(Unknown Source)
rity.AccessController.doPrivileged(Native Method)
URLClassLoader.findClass(Unknown Source)
.ClassLoader.loadClass(Unknown Source)
Launcher$AppClassLoader.loadClass(Unknown Source)
.ClassLoader.loadClass(Unknown Source)
.Class.forName0(Native Method)
.Class.forName(Unknown Source)
swamy.jdbcexamples.CreateTable.prepareConnection(CreateTable.java:50)
swamy.jdbcexamples.CreateTable.main(CreateTable.java:17)

S\JDBC\1CreateTable>set classpath=E:\WTLABFINALPROGRAMS\JDBC\lib\ojdbc14.jar;.;
```

Output:



```
E:\WTLABFINALPROGRAMS\JDBC\1CreateTable>java com.yellaswamy.jdbcexamples.Create
Table created Successfully

E:\WTLABFINALPROGRAMS\JDBC\1CreateTable>
```

```
SQL*Plus: Release 10.2.0.1.0 - Production on Mon Mar 10 02:11:19 2014

Copyright (c) 1982, 2005, Oracle.  All rights reserved.

SQL> desc mytable1;
SP2-0640: Not connected
SP2-0641: "DESCRIBE" requires connection to server
SQL> connect system/manager;
Connected.
SQL> desc mytable1;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------

 COL1                                               VARCHAR2(20)
 COL2                                               NUMBER
 COL3                                               NUMBER(10,2)

SQL>
```

//2. InsertEx.java

```
//Working with statement
package com.yellaswamy.jdbcexamples;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

public class InsertEx
{

	public static void main(String[] args) throws SQLException, ClassNotFoundException
	{
					//Get Connection
		Connection con=prepareConnection();

		//obtain a statement
		Statement st=con.createStatement();

		//String query="create table mytable(col1 varchar2(20),col2 number,col3
number(10,2))";
		String query="insert into  mytable1 values('yellaswamy',1215,123.13)";
```

```
                //Execute the query
                int count=st.executeUpdate(query);
                System.out.println("Number of rows affected by this query="+count);


        }

public static Connection prepareConnection() throws SQLException,ClassNotFoundException
{
        // TODO Auto-generated method stub
        String driverClassName="oracle.jdbc.driver.OracleDriver";
        String url="jdbc:oracle:thin:@localhost:1521:xe";
        String userName="system";
        String password="manager";

        //load driver class
        Class.forName(driverClassName);
        //obtain a connection
        Connection conn=DriverManager.getConnection(url,userName,password);

        return conn;
}
}
```

//3. PreparedStatementEx1.java

```
package com.yellaswamy.jdbcexamples;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
public class PreparedStatementEx1
{
public static void main(String args[])throws Exception
{
String driverClassName="oracle.jdbc.driver.OracleDriver";
String url="jdbc:oracle:thin:@localhost:1521:xe";
String userName="system";
String password="manager";
        //load driver class
        Class.forName(driverClassName);
        //obtain a connection
        Connection conn=DriverManager.getConnection(url,userName,password);
        String query="insert into mytable values(?,?,?)";
        //step1:Get Prepared Statement
```

```
            PreparedStatement ps=conn.prepareStatement(query);
            //step2:set parameters
            ps.setString(1,"shashank1");
            ps.setInt(2,1216);
            ps.setDouble(3,123.56);
            //Step3:Execute the Query
            int i=ps.executeUpdate();
            System.out.println("Record inserted count="+i);
            //To excecute the query once again
            ps.setString(1,"Kamalamma1");
            ps.setInt(2,1217);
            ps.setDouble(3,125.65);
             i=ps.executeUpdate();
            System.out.println("Query Excecuted for the second time count="+i);
            conn.close();
}
}
```

//4.Resultset Example

```
package com.yellaswamy.jdbcexample;
import java.sql.*;
public class GetAllRows
{
public static void main(String args[])throws
            SQLException, ClassNotFoundException
{                       //Get Connection
Connection con=prepareConnection ();
// Obtain a Statement
Statement st=con.createStatement ();
String query = "select * from mytable";
//Execute the Query
ResultSet rs=st.executeQuery (query);
System.out.println ("COL1\t\tCOL2\tCOL3");
     while (rs.next ())
            {
System.out.print (rs.getString ("COL1") + "\t");
System.out.print (rs.getInt ("COL2") + "\t");
System.out.println (rs.getDouble ("COL3")+"\t");
                        }//while
                        con.close ();
            }//main
public static Connection prepareConnection()
                        throws SQLException,
                        ClassNotFoundException
```

```
                    {
String driverClassName="oracle.jdbc.driver.OracleDriver";
String url="jdbc:oracle:thin:@localhost:1521:xe";
                    String username="system";
                    String password="manager";
                    //Load driver class
                    Class.forName (driverClassName);
                    // Obtain a connection
                    return DriverManager. getConnection (url, username, password);
          }//prepareConnection
}//class
```

CallableStatement Interface

To call the **stored procedures and functions**, CallableStatement interface is used.

We can have business logic on the database by the use of stored procedures and functions that will make the performance better because these are precompiled.

Suppose you need the get the age of the employee based on the date of birth, you may create a function that receives date as the input and returns age of the employee as the output.

1. The java.sql.CallableStatement is a part of JDBC API that describes a standard abstraction for the CallableStatement object,implemented by third-party vendor as a part of JDBC driver.
2. The CallableStatement object provides support for both input and output parameters.

What is the difference between stored procedures and functions.

The differences between stored procedures and functions are given below:

Stored Procedure	**Function**
is used to perform business logic.	is used to perform calculation.
must not have the return type.	must have the return type.
may return 0 or more values.	may return only one values.
We can call functions from the procedure.	Procedure cannot be called from function.
Procedure supports input and output parameters.	Function supports only input parameter.
Exception handling using try/catch block can be used in stored procedures.	Exception handling using try/catch can't be used in user defined functions.

Syntax of creating a stored procedure:

```
Create or [Replace] Procedure procedure_name
[(parameter[,parameter])]
IS
[Declarations]
BEGIN
        Executables
        [EXCEPTION exceptions]
END [Procedure_name]
```

Steps to use CallableStatement in an Application are as follows:

1. Create the CallableStatement object
2. Setting the values of parameters
3. Registering the OUT parameter type
4. Excecuting the stored procedure or function
5. Retrieving the parameter values

Let's discuss these steps in details

Create the CallableStatement object:

The first step to use CallableStatement is to create the CallableStatement object.The CallableStatement object can be created by invoking the prepareCall(String) method of the Connection object.

The syntax to call the prepareCall method in an application is as follows:

```
{call procedure_name(?,?,…)}        //Calling the procedure method with parameters
{call procedure_name}        //with no parameter
```

Setting the values of parameters

After creating the CallableStatement object,you need to set the values for the IN and IN OUT type parameters of stored procedure.The values of these parameters can be set by calling the setXXX() methods of the CallableStatement object.These methods are used to pass the values to the IN OUT parameters.The values for the parameter can be set by using the following syntax:

```
        setXXX(int index,XXX value)
```

Registering the OUT Parameter Type

The OUT or IN OUT parameters,which are used in a procedure and represented by CallableStatement,must be registered to collect the values of the parameters after the stored procedure is executed.

The parameters can be registered by using the following syntax:

```
        registerOutParameter(int index,int type)
```

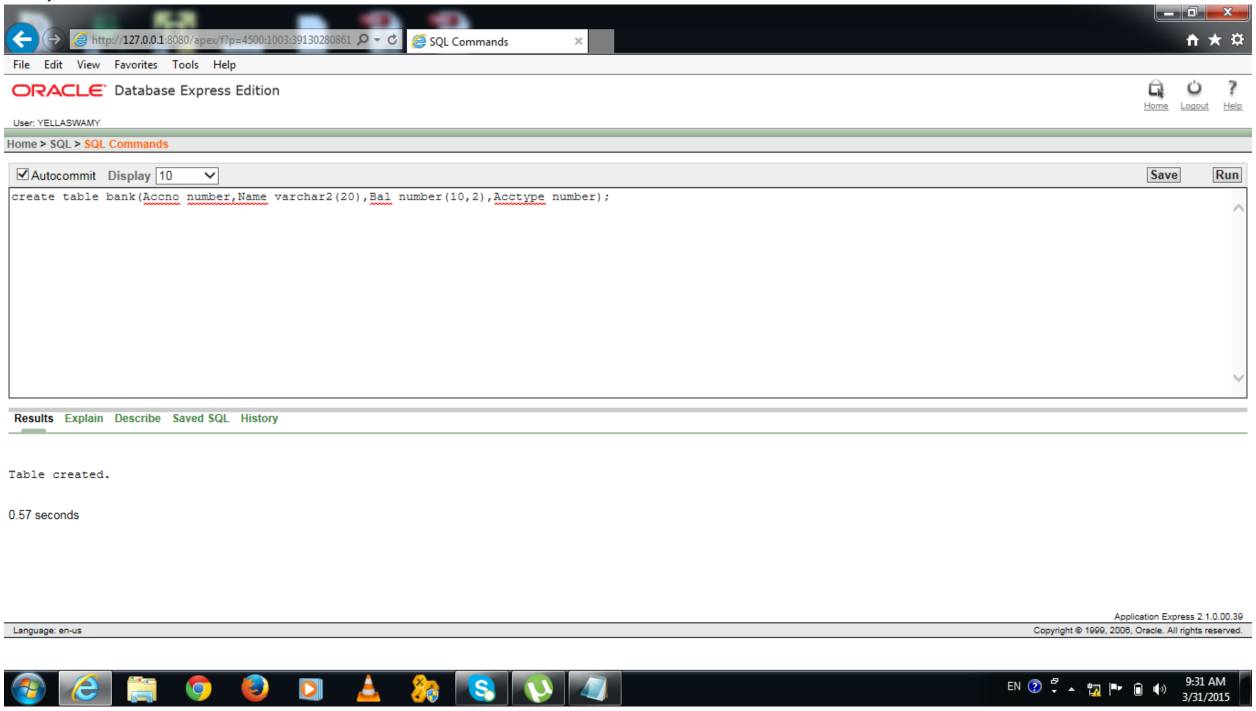
Excecuting the Stored Procedure or Function

After registering the OUT parameter type you need to execute the procedure by using execute() method of CallableStatement object.The execute() method of CallableStatement does not take any argument.

Retrieving the Parameter Values:
After executing the stored procedure,you can retrieve its OUT or IN OUT type parameter values.

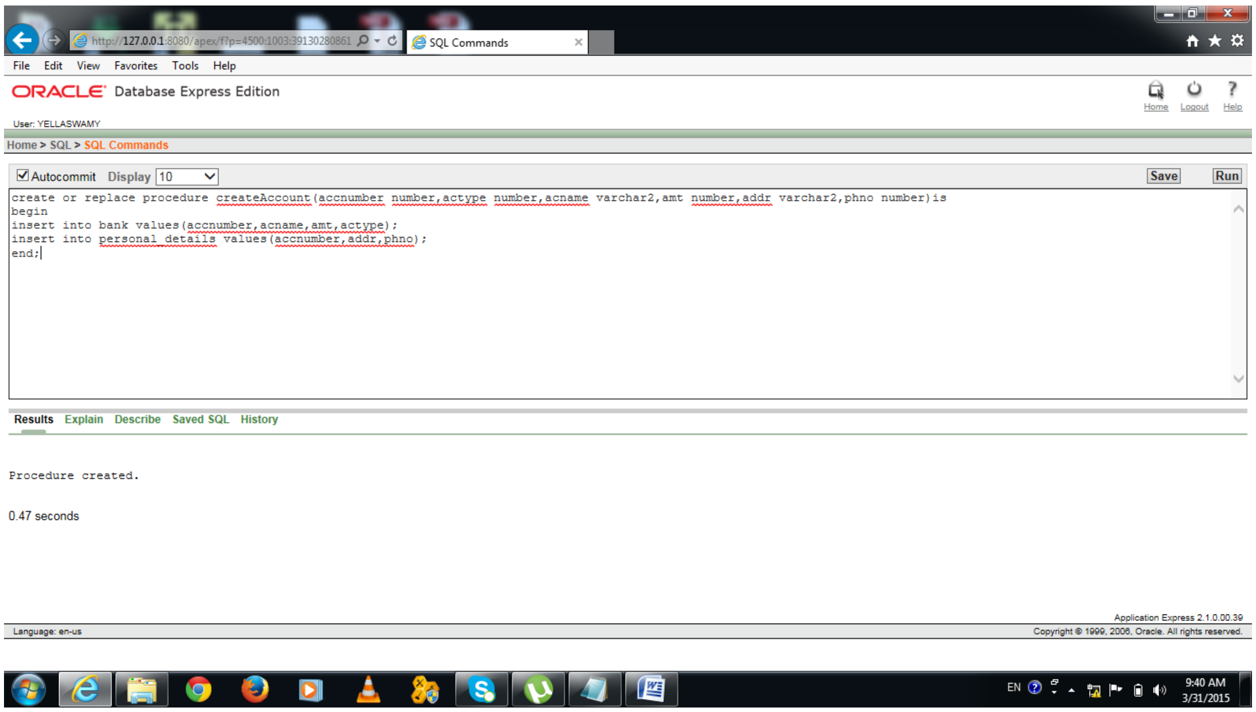
EXECUTING A STORED PROCEDURE WITH IN PARAMETERS

Create the Following Tables:
create table bank(Accno number,Name varchar2(20),Bal number(10,2),Acctype number);
Create table personal_details(Accno number,address varchar2(20),phno number);
Create the following Procedure:
create or replace procedure createAccount(accnumber number,actype number,acname varchar2,amt number,addr varchar2,phno number)is
begin
insert into bank values(accnumber,acname,amt,actype);
insert into personal_details values(accnumber,addr,phno);
end;



```
create table bank(Accno number,Name varchar2(20),Bal number(10,2),Acctype number);
```

```
Table created.

0.57 seconds
```


http://127.0.0.1:8080/apex/f?p=4500:1003:39130280861
SQL Commands
File Edit View Favorites Tools Help
ORACLE Database Express Edition
User: YELLASWAMY
Home > SQL > SQL Commands
Autocommit Display 10
Save
Run
Results Explain Describe Saved SQL History

```
create or replace procedure createAccount(accnumber number,actype number,acname varchar2,amt number,addr varchar2,phno number)is
begin
insert into bank values(accnumber,acname,amt,actype);
insert into personal_details values(accnumber,addr,phno);
end;
```

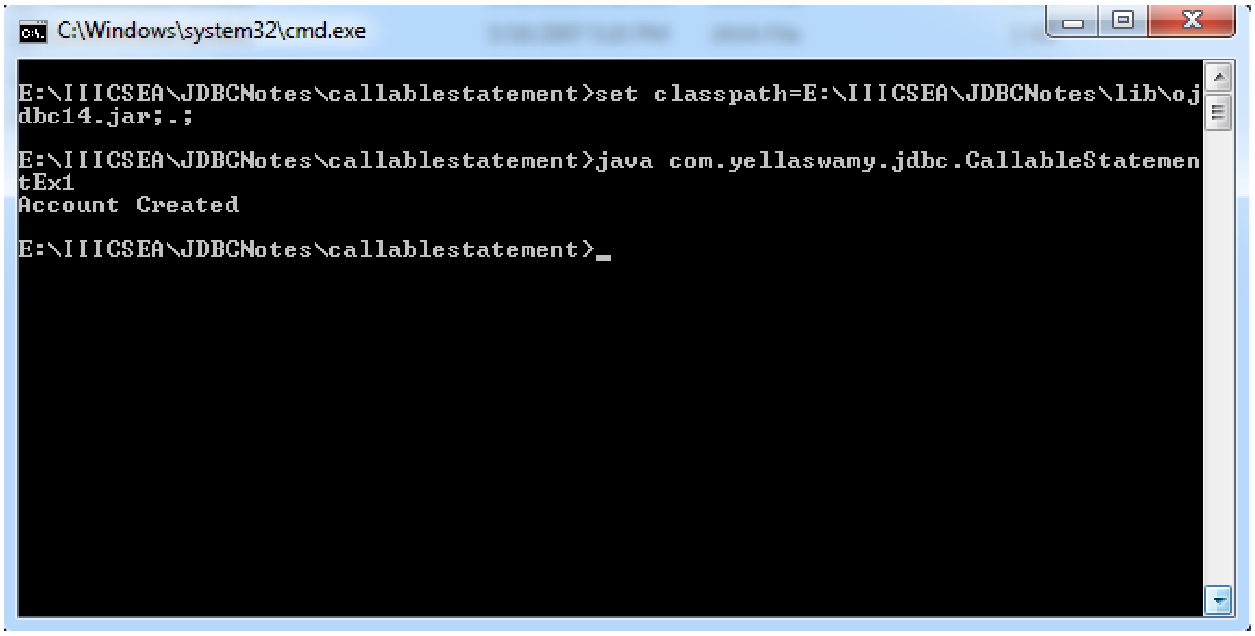
```
Procedure created.
```

0.47 seconds

Application Express 2.1.0.00.39
Language: en-us
Copyright © 1999, 2006, Oracle. All rights reserved.
9:40 AM
3/31/2015

// CallableStatementEx1.java

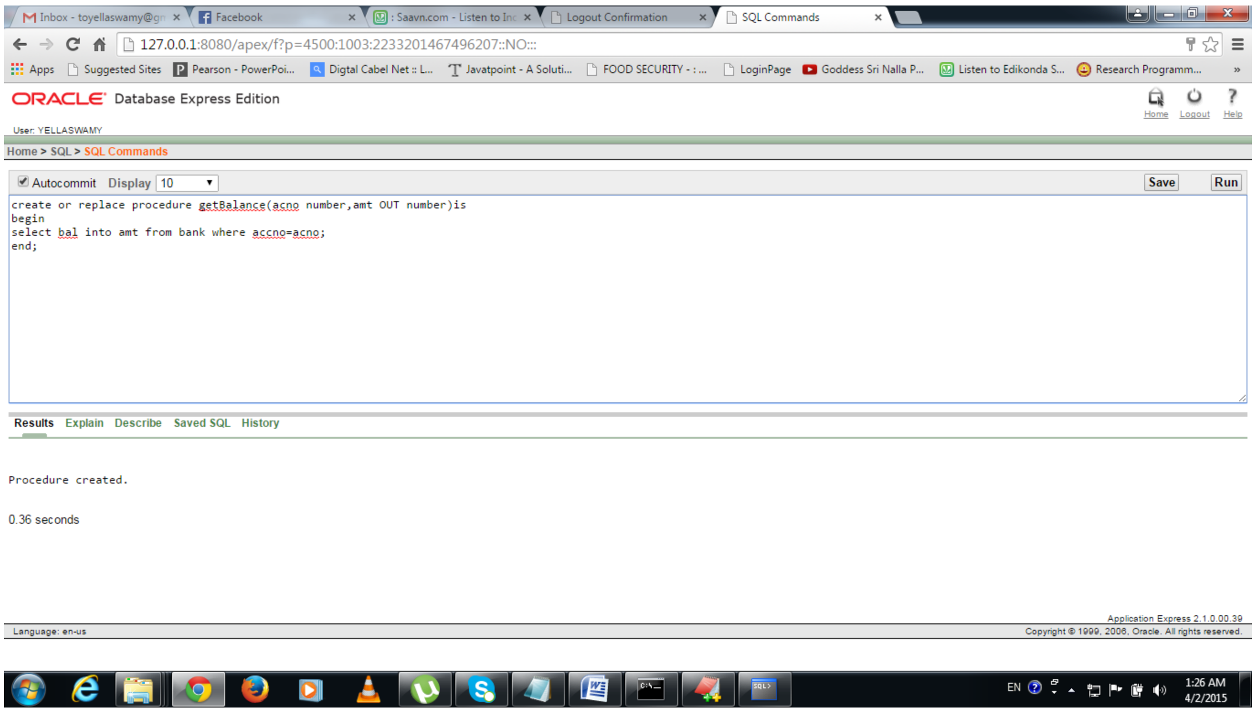
```
package com.yellaswamy.jdbc;

import java.sql.*;
/**
 * @author yellaswamy
 */
public class CallableStatementEx1 {

 public static void main(String s[]) throws Exception {

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

        //Step1: Get CallableStatement
        CallableStatement cs= con.prepareCall("{call createAccount (?,?,?,?,?,?)}");

        //Step2: set IN parameters
        cs.setInt(1, 101);
        cs.setInt(2, 10);
        cs.setString(3, "Yellaswamy");
        cs.setDouble(4, 10000);
        cs.setString(5, "Hyderabad");
        cs.setInt(6, 123456789);

        //Step3 : register OUT parameters
        //In this procedure example we dont have OUT parameters

        //Step4
        cs.execute();

        System.out.println("Account Created");

        con.close();
 }//main
}//class
```



```
E:\IIICSEA\JDBCNotes\callablestatement>set classpath=E:\IIICSEA\JDBCNotes\lib\oj
dbc14.jar;.;

E:\IIICSEA\JDBCNotes\callablestatement>java com.yellaswamy.jdbc.CallableStatemen
tEx1
Account Created

E:\IIICSEA\JDBCNotes\callablestatement>_
```

Executing a Stored Procedure with OUT Parameters:
We create an application that calls a stored procedure name getBalance() by using CallableStatement.
First create a procedure named getBalance() as shown in the following code snippet:

```
create or replace procedure getBalance(acno number,amt OUT number)is
begin
select bal into amt from bank where accno=acno;
end;
```



Autocommit Display 10

```
create or replace procedure getBalance(acno number,amt OUT number)is
begin
select bal into amt from bank where accno=acno;
end;
```

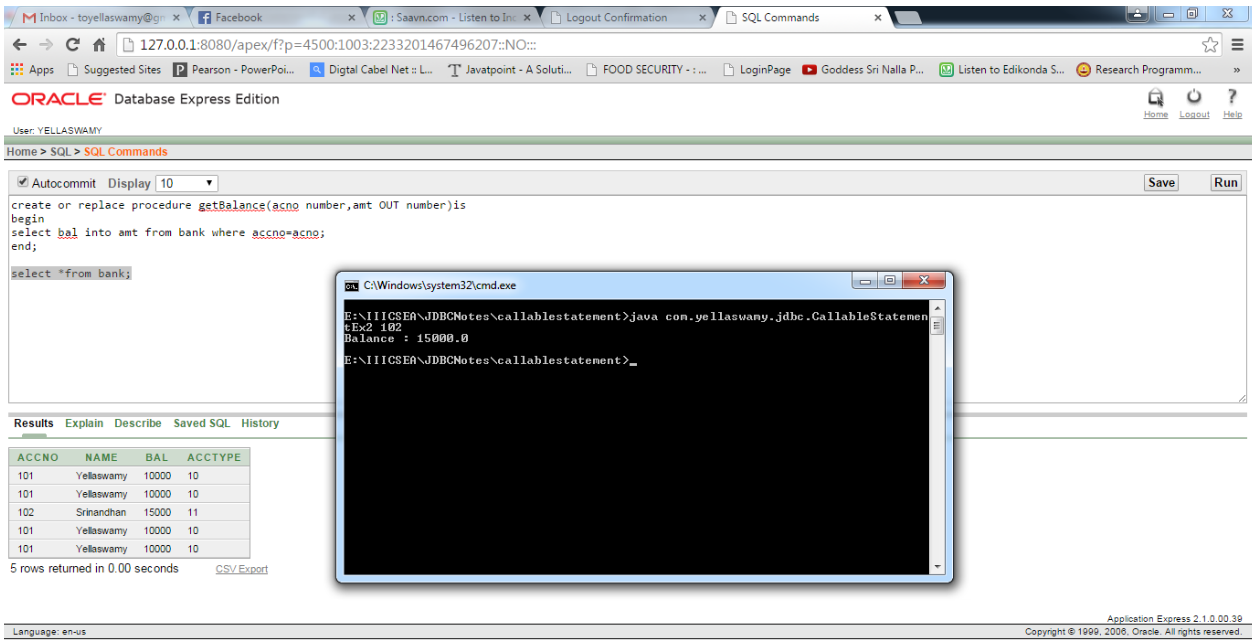
```
Procedure created.
```

0.36 seconds

// CallableStatementEx2.java

```
package com.yellaswamy.jdbc;

import java.sql.*;
/**
 * @author yellaswamy
 */
public class CallableStatementEx2 {
public static void main(String s[]) throws Exception {

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

                    CallableStatement cs= con.prepareCall("{call getBalance(?,?)}");

                    cs.setInt(1, Integer.parseInt(s[0]));
                    cs.registerOutParameter(2, Types.DOUBLE);

                    cs.execute();

                    System.out.println("Balance : "+ cs.getDouble(2));

                    con.close();
          }//main
}//class
```

OUTPUT:

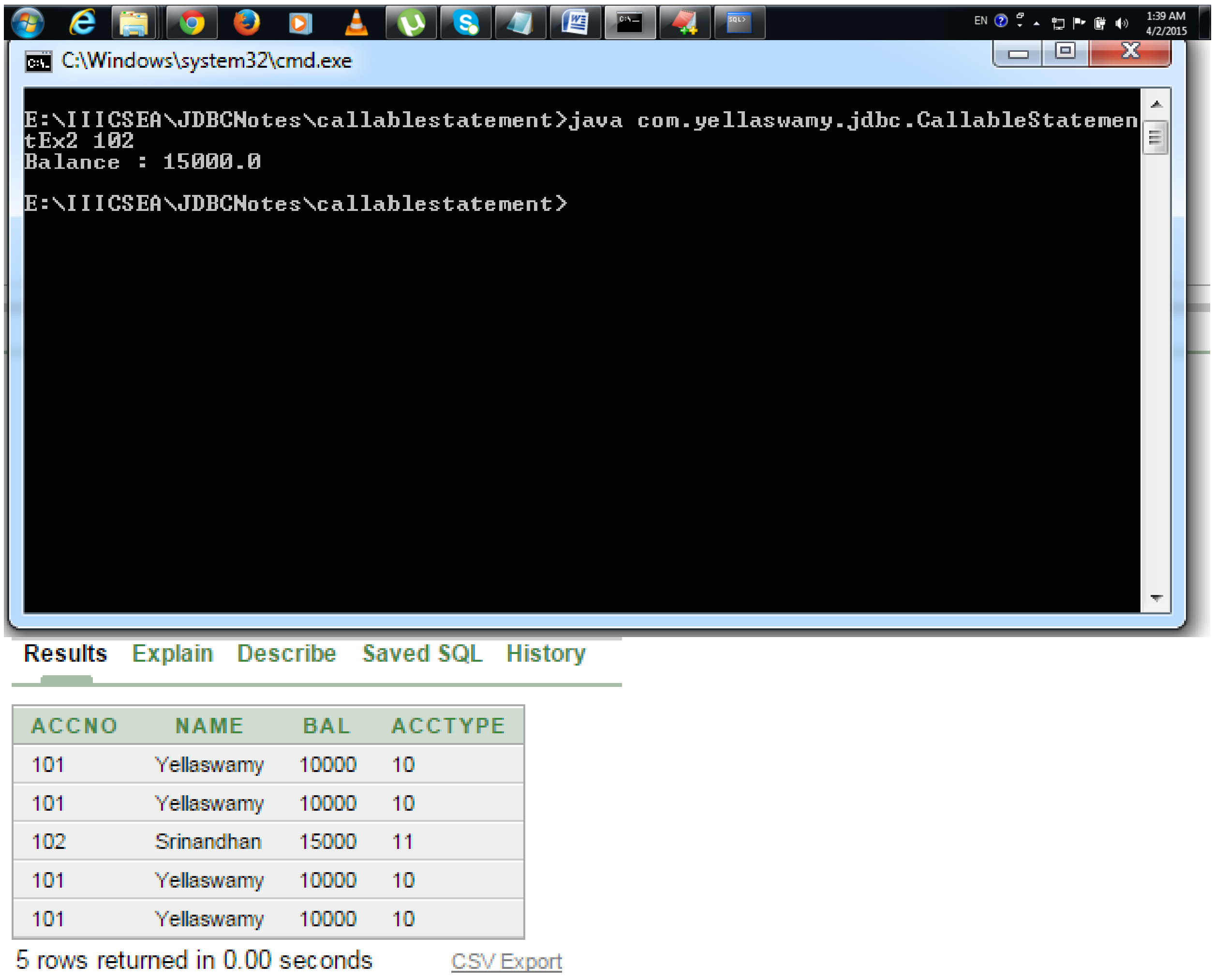


```
create or replace procedure getBalance(acno number,amt OUT number)is
begin
select bal into amt from bank where accno=acno;
end;

select *from bank;
```

```
E:\IIICSEA\JDBCNotes\callablestatement>java com.yellaswamy.jdbc.CallableStatemen
tEx2 102
Balance : 15000.0

E:\IIICSEA\JDBCNotes\callablestatement>_
```

ACCNO	NAME	BAL	ACCTYPE
101	Yellaswamy	10000	10
101	Yellaswamy	10000	10
102	Srinandhan	15000	11
101	Yellaswamy	10000	10
101	Yellaswamy	10000	10

5 rows returned in 0.00 seconds CSV Export



```
E:\IIICSEA\JDBCNotes\callablestatement>java com.yellaswamy.jdbc.CallableStatemen
tEx2 102
Balance : 15000.0

E:\IIICSEA\JDBCNotes\callablestatement>
```

ACCNO	NAME	BAL	ACCTYPE
101	Yellaswamy	10000	10
101	Yellaswamy	10000	10
102	Srinandhan	15000	11
101	Yellaswamy	10000	10
101	Yellaswamy	10000	10

5 rows returned in 0.00 seconds CSV Export

Package javax.sql Description

Provides the API for server side data source access and processing from the Java™ programming language. This package supplements the `java.sql` package and, as of the version 1.4 release, is included in the Java Platform, Standard Edition (Java SE™). It remains an essential part of the Java Platform, Enterprise Edition (Java EE™).

The `javax.sql` package provides for the following:

1. The `DataSource` interface as an alternative to the `DriverManager` for establishing a connection with a data source
2. Connection pooling and Statement pooling
3. Distributed transactions
4. Rowsets

Applications use the `DataSource` and `RowSet` APIs directly, but the connection pooling and distributed transaction APIs are used internally by the middle-tier infrastructure.

Using a `DataSource` Object to Make a Connection

The `javax.sql` package provides the preferred way to make a connection with a data source. The `DriverManager` class, the original mechanism, is still valid, and code using it will continue to run. However, the newer `DataSource` mechanism is preferred because it offers many advantages over the `DriverManager` mechanism.

These are the main advantages of using a `DataSource` object to make a connection:

- Changes can be made to a data source's properties, which means that it is not necessary to make changes in application code when something about the data source or driver changes.
- Connection and Statement pooling and distributed transactions are available through a `DataSource` object that is implemented to work with the middle-tier infrastructure. Connections made through the `DriverManager` do not have connection and statement pooling or distributed transaction capabilities.

Driver vendors provide `DataSource` implementations. A particular `DataSource` object represents a particular physical data source, and each connection the `DataSource` object creates is a connection to that physical data source.

A logical name for the data source is registered with a naming service that uses the Java Naming and Directory Interface™ (JNDI) API, usually by a system administrator or someone performing the duties of a system administrator. An application can retrieve the `DataSource` object it wants by doing a lookup on the logical name that has been registered for it. The application can then use the `DataSource` object to create a connection to the physical data source it represents.

A `DataSource` object can be implemented to work with the middle tier infrastructure so that the connections it produces will be pooled for reuse. An application that uses such a `DataSource` implementation will automatically get a connection that participates in connection pooling. A `DataSource` object can also be implemented to work with the middle tier infrastructure so that the connections it produces can be used for distributed transactions without any special coding.

Connection Pooling and Statement Pooling

Connections made via a `DataSource` object that is implemented to work with a middle tier connection pool manager will participate in connection pooling. This can improve performance dramatically because creating new connections is very expensive. Connection pooling allows a connection to be used and reused, thus cutting down substantially on the number of new connections that need to be created.

Connection pooling is totally transparent. It is done automatically in the middle tier of a Java EE configuration, so from an application's viewpoint, no change in code is required. An application simply uses the `DataSource.getConnection` method to get the pooled connection and uses it the same way it uses any `Connection` object.

The classes and interfaces used for connection pooling are:

- `ConnectionPoolDataSource`
- `PooledConnection`
- `ConnectionEvent`
- `ConnectionEventListener`
- `StatementEvent`
- `StatementEventListener`

The connection pool manager, a facility in the middle tier of a three-tier architecture, uses these classes and interfaces behind the scenes. When a `ConnectionPoolDataSource` object is called on to create a `PooledConnection` object, the connection pool manager will register as a `ConnectionEventListener` object with the new `PooledConnection` object. When the connection is closed or there is an error, the connection pool manager (being a listener) gets a notification that includes a `ConnectionEvent` object.

If the connection pool manager supports `Statement` pooling, for `PreparedStatements`, which can be determined by invoking the method `DatabaseMetaData.supportsStatementPooling`, the connection pool manager will register as a `StatementEventListener` object with the new `PooledConnection` object. When the `PreparedStatement` is closed or there is an error, the connection pool manager (being a listener) gets a notification that includes a `StatementEvent` object.

Distributed Transactions

As with pooled connections, connections made via a `DataSource` object that is implemented to work with the middle tier infrastructure may participate in distributed transactions. This gives an application the ability to involve data sources on multiple servers in a single transaction.

The classes and interfaces used for distributed transactions are:

- `XADataSource`
- `XAConnection`

These interfaces are used by the transaction manager; an application does not use them directly.

The `XAConnection` interface is derived from the `PooledConnection` interface, so what applies to a pooled connection also applies to a connection that is part of a distributed transaction. A transaction manager in the middle tier handles everything transparently. The only change in application code is that an application cannot do anything that would interfere with the transaction manager's handling of the transaction. Specifically, an application cannot call the methods `Connection.commit` or `Connection.rollback`, and it cannot set the connection to be in auto-commit mode (that is, it cannot call `Connection.setAutoCommit(true)`).

An application does not need to do anything special to participate in a distributed transaction. It simply creates connections to the data sources it wants to use via the `DataSource.getConnection` method, just as it normally does. The transaction manager manages the transaction behind the scenes. The `XADataSource` interface creates `XAConnection` objects, and each `XAConnection` object creates an `XAResource` object that the transaction manager uses to manage the connection.

Rowsets

The `RowSet` interface works with various other classes and interfaces behind the scenes. These can be grouped into three categories.

1. Event Notification
   - `RowSetListener`
     A `RowSet` object is a JavaBeans™ component because it has properties and participates in the JavaBeans event notification mechanism. The `RowSetListener` interface is implemented by a component that wants to be notified about events that occur to a particular `RowSet` object. Such a component registers itself as a listener with a rowset via the `RowSet.addRowSetListener` method.

     When the `RowSet` object changes one of its rows, changes all of it rows, or moves its cursor, it also notifies each listener that is registered with it. The listener reacts by carrying out its implementation of the notification method called on it.

- `RowSetEvent`
  As part of its internal notification process, a `RowSet` object creates an instance of `RowSetEvent` and passes it to the listener. The listener can use this `RowSetEvent` object to find out which rowset had the event.

2. Metadata
   - `RowSetMetaData`
     This interface, derived from the `ResultSetMetaData` interface, provides information about the columns in a `RowSet` object. An application can use `RowSetMetaData` methods to find out how many columns the rowset contains and what kind of data each column can contain.

     The `RowSetMetaData` interface provides methods for setting the information about columns, but an application would not normally use these methods. When an application calls the `RowSet` method `execute`, the `RowSet` object will contain a new set of rows, and its `RowSetMetaData` object will have been internally updated to contain information about the new columns.

3. The Reader/Writer Facility
   A `RowSet` object that implements the `RowSetInternal` interface can call on the `RowSetReader` object associated with it to populate itself with data. It can also call on the `RowSetWriter` object associated with it to write any changes to its rows back to the data source from which it originally got the rows. A rowset that remains connected to its data source does not need to use a reader and writer because it can simply operate on the data source directly.
   - `RowSetInternal`
     By implementing the `RowSetInternal` interface, a `RowSet` object gets access to its internal state and is able to call on its reader and writer. A rowset keeps track of the values in its current rows and of the values that immediately preceded the current ones, referred to as the *original* values. A rowset also keeps track of (1) the parameters that have been set for its command and (2) the connection that was passed to it, if any. A rowset uses the `RowSetInternal` methods behind the scenes to get access to this information. An application does not normally invoke these methods directly.
   - `RowSetReader`
     A disconnected `RowSet` object that has implemented the `RowSetInternal` interface can call on its reader (the `RowSetReader` object associated with it) to populate it with data. When an application calls the `RowSet.execute` method, that method calls on the rowset's reader to do much of the work. Implementations can vary widely, but generally a reader makes a connection to the data source, reads data from the data source and populates the rowset with it, and closes the connection. A reader may also update the `RowSetMetaData` object for its rowset. The rowset's internal state is also updated, either by the reader or directly by the method `RowSet.execute`.
   - `RowSetWriter`
     A disconnected `RowSet` object that has implemented the `RowSetInternal` interface can call on its writer (the `RowSetWriter` object associated with it) to

write changes back to the underlying data source. Implementations may vary widely, but generally, a writer will do the following:

- Make a connection to the data source
- Check to see whether there is a conflict, that is, whether a value that has been changed in the rowset has also been changed in the data source
- Write the new values to the data source if there is no conflict
- Close the connection

The `RowSet` interface may be implemented in any number of ways, and anyone may write an implementation. Developers are encouraged to use their imaginations in coming up with new ways to use rowsets.