Chapter 8
JDBC

8.1 Introduction

JDBC is a Java standard that provides the interface for connecting from Java to relational databases. The JDBC standard is defined by Sun Microsystems and implemented through the standard `java.sql` interfaces. This allows individual providers to implement and extend the standard with their own JDBC drivers.

JDBC stands for

Java Database Connectivity, 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

JDBC API is a Java API that can access any kind of tabular data, especially data stored in a Relational Database. JDBC works with Java on a variety of platforms, such as Windows, Mac OS, and the various versions of UNIX.

8.2 Design Of JDBC

Just as Java was designed to provide platform independence from hardware/software platforms, so too JDBC has been designed to provide some degree of database independence for developers. JDBC is designed to provide a database-neutral API for accessing relational databases from different vendors. Just as a Java application does not need to be aware of the operating system platform on which it is running, so too JDBC has been designed so that the database application can use the same methods to access data regardless of the underlying database product.

JDBC was developed to work with the most common type of database: **the relational database.** This is not to say that JDBC cannot be used with another type of database. In fact, there are JDBC drivers that allow the API to be used to connect to both high-end, mainframe databases, which are not relational, and to access flat files and spreadsheets as databases (which are definitely not relational). But the reality is that JDBC is most commonly used with relational databases.

The technical definition of a relational database is a database that stores data as a collection of related entities. These entities are composed of attributes that describe the entity, and each entity has a collection of rows. Another way to think about a relational database is that it stores information on real-world objects (the entities). The information about the objects is contained in the attributes for the object.

Since real world objects have some type of relation to each other, we must have a facility for expressing relations between the objects in the database. The relationships between the database objects is described using a query language, the most popular of which is the Structured Query Language (SQL).

JavaSoft's JDBC consists of two layers: the **JDBC API** and the **JDBC Driver Manager API.**

The JDBC API is the top layer and is the programming interface in Java to *structured query language* (SQL) which is the standard for accessing relational databases.

The JDBC API communicates with the JDBC Driver Manager API, sending it various SQL statements. The manager communicates (transparent to the programmer) with the various third party drivers (provided by Database vendors like Oracle) that actually connect to the database and return the information from the query.

JDBC Architecture:

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

JDBC API: This provides the application-to-JDBC Manager connection.

JDBC Driver API: This supports the JDBC Manager-to-Driver Connection.

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

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

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

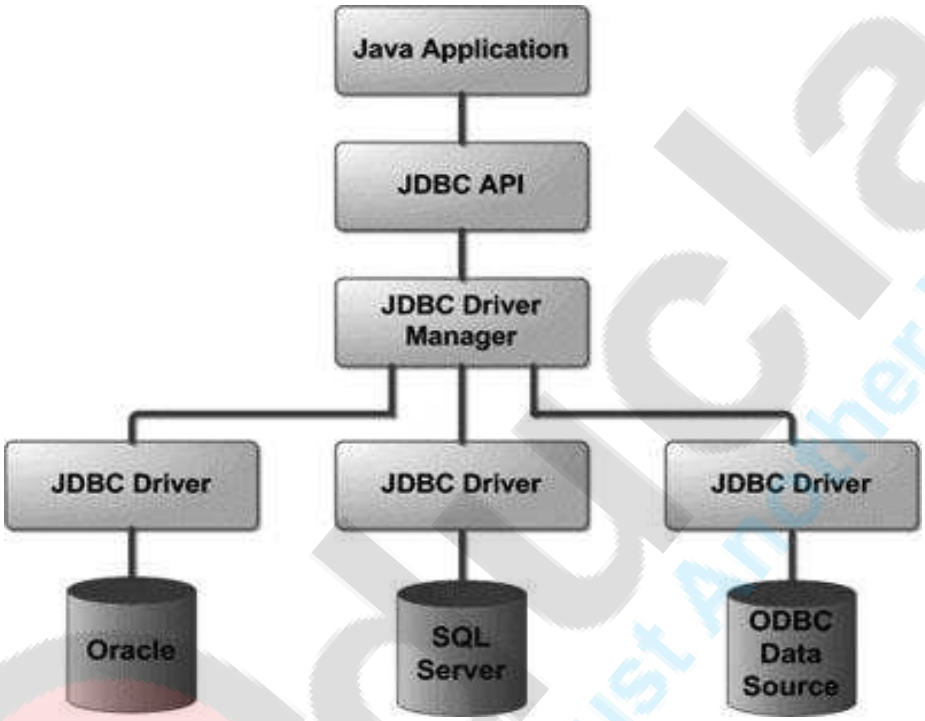

Fig. Architectural diagram

Common JDBC Components:

The JDBC API provides the following interfaces and classes:

- **DriverManager:** This class manages a list of database drivers. Matches connection requests from the java application with the proper database driver using communication subprotocol. The first driver that recognizes a certain subprotocol under JDBC will be used to establish a database Connection.
- **Driver:** This interface handles the communications with the database server. You will interact directly with Driver objects very rarely. Instead, you use DriverManager

objects, which manages objects of this type. It also abstracts the details associated with working with Driver objects

- **Connection :** This interface with all methods for contacting a database. The connection object represents communication context, i.e., all communication with database is through connection object only.
- **Statement :** You use objects created from this interface to submit the SQL statements to the database. Some derived interfaces accept parameters in addition to executing stored procedures.
- **ResultSet:** These objects hold data retrieved from a database after you execute an SQL query using Statement objects. It acts as an iterator to allow you to move through its data.
- **SQLException:** This class handles any errors that occur in a database application.

8.3 JDBC configuration

8.3.1 What is JDBC Driver ?

JDBC drivers implement the defined 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 behaviours defined and their actual implementaions are done in third-party drivers. Third party vendors implements the `java.sql.Driver` interface in their database driver.

8.3.2 JDBC Drivers Types:

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:

Type 1: JDBC-ODBC Bridge Driver:

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.

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.

The JDBC-ODBC bridge that comes with JDK 1.2 is a good example of this kind of driver.

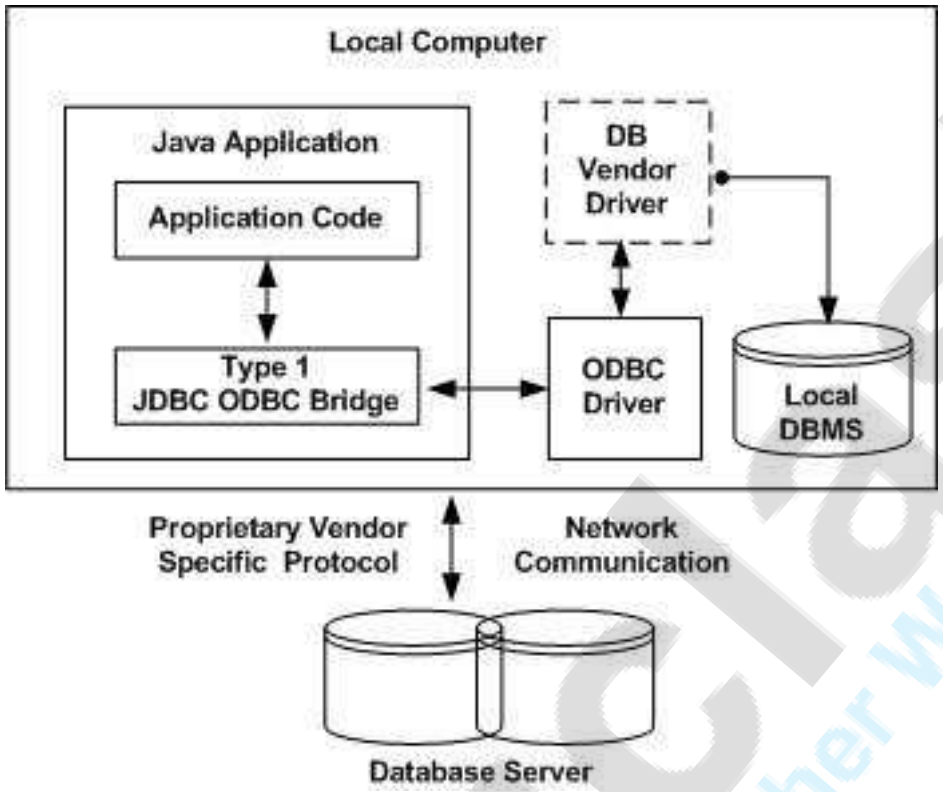

Fig. Type 1: JDBC-ODBC Bridge Driver

Type 2: JDBC-Native API:

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.

The Oracle Call Interface (OCI) driver is an example of a Type 2 driver.

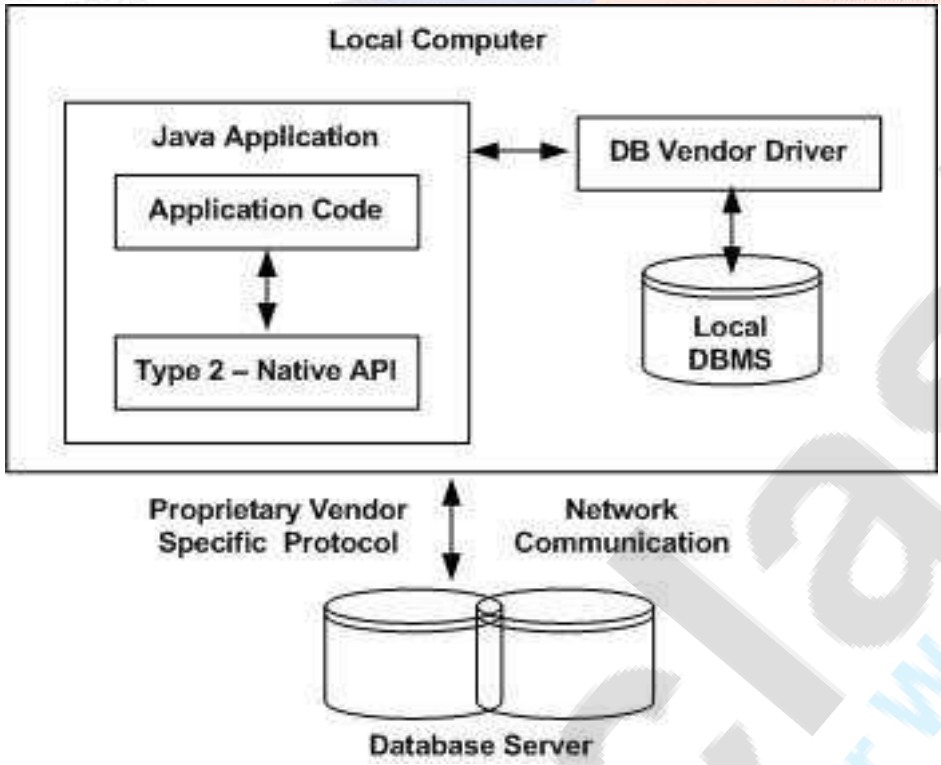

Fig. Type 2: JDBC-Native API

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.

Your application server might use a Type 1, 2, or 4 driver to communicate with the database, understanding the nuances will prove helpful.

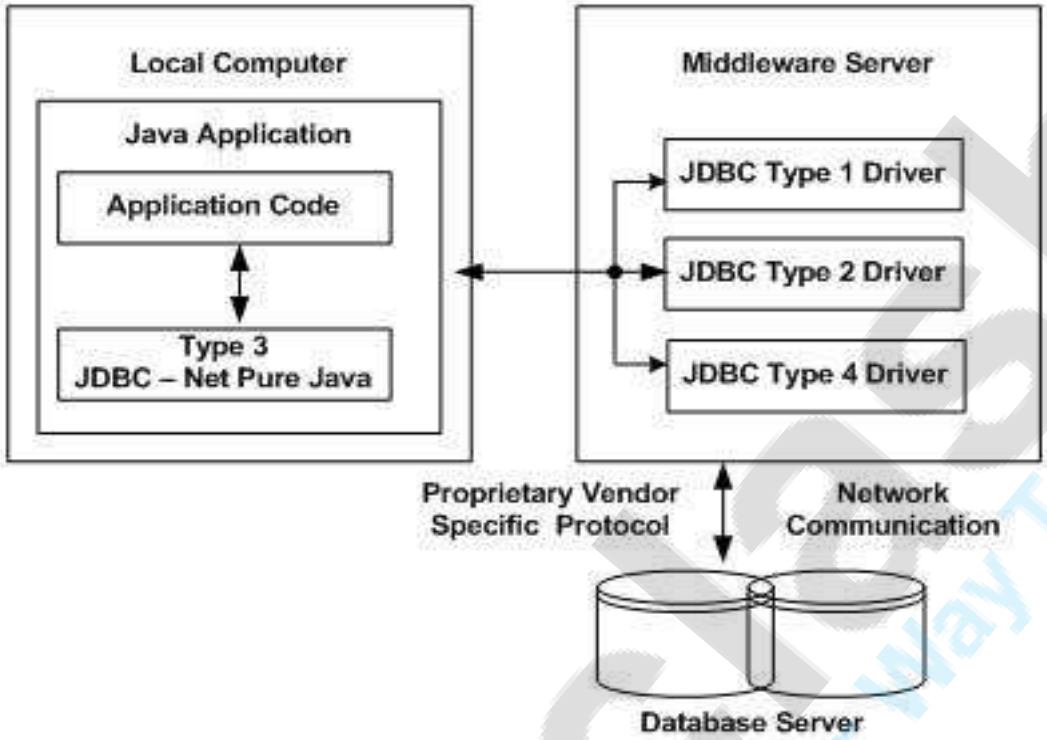

Fig. Type 3: JDBC-Net pure Java

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.

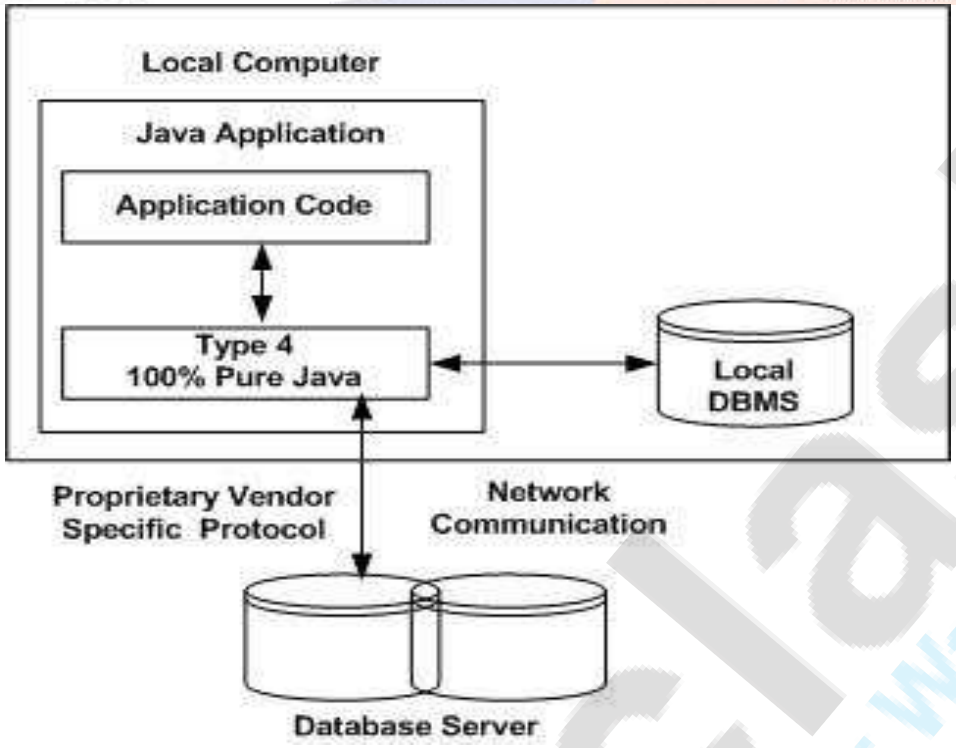

Fig. Type 4: 100% pure Java

8.3.3 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.

After you've installed the appropriate driver, it's time to establish a database connection using JDBC.

The programming involved to establish a JDBC connection is fairly simple. Here are these simple four steps:

- **Import JDBC Packages:** Add **import** statements to your Java program to import required classes in your Java code.
- **Register JDBC Driver:** This step causes the JVM to load the desired driver implementation into memory so it can fulfill your JDBC requests.
- **Database URL Formulation:** This is to create a properly formatted address that points to the database to which you wish to connect.
- **Create Connection Object:** Finally, code a call to the *DriverManager* object's *getConnection( )* method to establish actual database connection.

8.3.4 Import JDBC Packages:

The **Import** statements tell the Java compiler where to find the classes you reference in your code and are placed at the very beginning of your source code.

To use the standard JDBC package, which allows you to select, insert, update, and delete data in SQL tables, add the following *imports* to your source code:

```
import java.sql.* ;  // for standard JDBC programs
import java.math.* ; // for BigDecimal and BigInteger support
```

8.3.5 Register JDBC Driver:

You must register the your driver in your program before you use it. Registering the driver is the process by which the Oracle driver's class file is loaded into memory so it can be utilized as an implementation of the JDBC interfaces.

You need to do this registration only once in your program. You can register a driver in one of two ways.

Approach (I) - Class.forName():

The most common approach to register a driver is to use Java's `Class.forName()` method to dynamically load the driver's class file into memory, which automatically registers it. This method is preferable

because it allows you to make the driver registration configurable and portable.

The following example uses `Class.forName( )` to register the Oracle driver:

```
try {

Class.forName("oracle.jdbc.driver.OracleDriver");
}
catch(ClassNotFoundException ex) {
System.out.println("Error: unable to load driver class!");
System.exit(1);
}
```

You can use `getInstance()` method to work around noncompliant JVMs, but then you'll have to code for two extra Exceptions as follows:

```
try {
Class.forName("oracle.jdbc.driver.OracleDriver").newInstanc
e();
}
catch(ClassNotFoundException ex) {
      System.out.println("Error: unable to load driver
class!");
      System.exit(1);
catch(IllegalAccessException ex) {
      System.out.println("Error: access problem while
loading!");
      System.exit(2);
catch(InstantiationException ex) {
      System.out.println("Error: unable to instantiate
driver!");
      System.exit(3);
}
```

Approach (II) - DriverManager.registerDriver():

The second approach you can use to register a driver is to use the static `DriverManager.registerDriver()` method.

You should use the `registerDriver()` method if you are using a non-JDK compliant JVM, such as the one provided by Microsoft.

The following example uses `registerDriver()` to register the Oracle driver:

```
try {
Driver myDriver = new oracle.jdbc.driver.OracleDriver();
DriverManager.registerDriver( myDriver );
}
catch(ClassNotFoundException ex) {
System.out.println("Error: unable to load driver class!");
System.exit(1);
}
```

8.3.6 Database URL Formulation:

After you've loaded the driver, you can establish a connection using the `DriverManager.getConnection()` method. For easy reference, let me list the three overloaded `DriverManager.getConnection()` methods:

- `getConnection(String url)`
- `getConnection(String url, Properties prop)`
- `getConnection(String url, String user, String password)`

Here each form requires a database **URL.** A database URL is an address that points to your database.

Formulating a database URL is where most of the problems associated with establishing a connection occur.

Following table lists down popular JDBC driver names and database URL.

RDBMS	JDBC driver name	URL format
MySQL	com.mysql.jdbc.Driver	**jdbc:mysql:**//hostname/ databaseName
ORACLE	oracle.jdbc.driver.OracleDriver	**jdbc:oracle:thin:@**hostname:port Number:databaseName
DB2	COM.ibm.db2.jdbc.net.DB2Driver	**jdbc:db2:**hostname:port Number/databaseName
Sybase	com.sybase.jdbc.SybDriver	**jdbc:sybase:Tds:**hostname: port Number/databaseName

All the highlighted part in URL format is static and you need to change only remaining part as per your database setup.

8.3.7 Create Connection Object:

Using a database URL with a username and password:

I listed down three forms of `DriverManager.getConnection()` method to create a

connection object. The most commonly used form of `getConnection()` requires you to pass a database URL, a *username*, and a *password*:

Assuming you are using Oracle's **thin** driver, you'll specify a `host:port:databaseName` value for the database portion of the URL.

If you have a host at TCP/IP address 192.0.0.1 with a host name of amrood, and your Oracle listener is configured to listen on port 1521, and your database name is EMP, then complete database URL would then be:

```
jdbc:oracle:thin:@amrood:1521:EMP
```

Now you have to call `getConnection()` method with appropriate username and password to get a **Connection** object as follows:

```
String URL = "jdbc:oracle:thin:@amrood:1521:EMP";
String USER = "username";
String PASS = "password"
Connection conn = DriverManager.getConnection(URL, USER, PASS);
```

Using only a database URL:

A second form of the `DriverManager.getConnection( )` method requires only a database URL:

```
DriverManager.getConnection(String url);
```

However, in this case, the database URL includes the username and password and has the following general form:

```
jdbc:oracle:driver:username/password@database
```

So the above connection can be created as follows:

```
String URL =
"jdbc:oracle:thin:username/password@amrood:1521:EMP";
Connection conn = DriverManager.getConnection(URL);
```

Using a database URL and a Properties object:

A third form of the `DriverManager.getConnection( )` method requires a database URL and a Properties object:

```
DriverManager.getConnection(String url, Properties info);
```

A Properties object holds a set of keyword-value pairs. It's used to pass driver properties to the driver during a call to the `getConnection()` method.

To make the same connection made by the previous examples, use the following code:

```
import java.util.*;
String URL = "jdbc:oracle:thin:@amrood:1521:EMP";
Properties info = new Properties( );
info.put( "user", "username" );
info.put( "password", "password" );

Connection conn = DriverManager.getConnection(URL, info);
```

Closing JDBC connections:

At the end of your JDBC program, it is required explicitly close all the connections to the database to end each database session. However, if you forget, Java's garbage collector will close the connection when it cleans up stale objects.

Relying on garbage collection, especially in database programming, is very poor programming practice. You should make a habit of always closing the connection with the `close()` method associated with connection object.

To ensure that a connection is closed, you could provide a finally block in your code. A *finally* block always executes, regardless if an exception occurs or not.

To close above opened connection you should call `close()` method as follows:

```
conn.close();
```

Explicitly closing a connection conserves DBMS resources, which will make your database administrator happy.

8.4 Executing SQL statement And Query Execution

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.

They also define methods that help bridge data type differences between Java and SQL data types used in a 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) {
. . .
}
finally {
 . . .
}
```

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 Obeject:

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 = conn.createStatement( );
. . .
}
catch (SQLException e) {
 . . .
}
finally {
stmt.close();
}
```

The PreparedStatement Objects:

The `PreparedStatement` interface extends the Statement interface which gives you added functionality with a couple of advantages over a generic Statement object.

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) {
 . . .
}
finally {
 . . .
}
```

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.

Closing PreparedStatement Obeject:

Just as you close a Statement object, for the same reason you should also close the `PreparedStatement` object.

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

```
PreparedStatement pstmt = null;
try {
String SQL = "Update Employees SET age = ? WHERE id = ?";
pstmt = conn.prepareStatement(SQL);
 . . .
}
catch (SQLException e) {
. . .
}
finally {
pstmt.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.

Creating 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.

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) {
. . .
}
```

```
finally {
  . . .
}
```

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.

Closing CallableStatement Obeject:

Just as you close other Statement object, for the same reason you should also close the `CallableStatement` object.

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

```
CallableStatement cstmt = null;
try {
   String SQL = "{call getEmpName (?, ?)}";
   cstmt = conn.prepareCall (SQL);
   . . .
}
catch (SQLException e) {
   . . .
}
finally {
   cstmt.close();
}
```

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

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

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

- **Navigational methods:** used to move the cursor around.
- **Get methods:** used to view the data in the columns of the current row being pointed 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.**

Example-

```
/*******************************************************************/
  //This example was coded and tested with JDK
  //Access was used as database, over the JDBC-ODBC bridge which
  //comes with the JDK.
  //To run this example, you need a database with the following
  //properties:
  //=> a table called "Cust"
  //=> a system DSN called "Database"
/*******************************************************************/

import java.sql.* ;

class JDBCQuery
```

```
{
 public static void main( String args[] )
 {
  try
     {
      // Load the database driver
         Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver" ) ;

      // Get a connection to the database
         Connection conn =
         DriverManager.getConnection( "jdbc:odbc:Database" ) ;

      // Print all warnings
         for( SQLWarning warn = conn.getWarnings();
        warn != null;
         warn = warn.getNextWarning() )
          {
         System.out.println( "SQL Warning:" ) ;
         System.out.println( "State  : " + warn.getSQLState()  ) ;
         System.out.println( "Message: " + warn.getMessage()   ) ;
         System.out.println( "Error  : " + warn.getErrorCode() ) ;
          }

      // Get a statement from the connection
         Statement stmt = conn.createStatement() ;

      // Execute the query
         ResultSet rs = stmt.executeQuery( "SELECT * FROM Cust" )
;

      // Loop through the result set
         while( rs.next() )
         System.out.println( rs.getString(1) ) ;

      // Close the result set, statement and the connection
         rs.close() ;
         stmt.close() ;
         conn.close() ;
      }
      catch( SQLException se )
      {
      System.out.println( "SQL Exception:" ) ;

      // Loop through the SQL Exceptions
         while( se != null )
          {
           System.out.println( "State  : " + se.getSQLState()  ) ;
           System.out.println( "Message: " + se.getMessage()   ) ;
```

```
                System.out.println( "Error  : " + se.getErrorCode() ) ;

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

8.5 Scrollable and updatable result sets

Result Set types in JDBC

In JDBC ResultSets are classified into following ways.

1.on the basis of ResultSet cursor movement.

2.on the basis of ResultSet concurrency.

on the basis of ResultSet cursor movement there are two types of ResultSets

i)forward only ResultSet:

This ResultSet will allow users only to iterate the data in forward direction.To refer this ResultSet object `ResultSet` interface has provided the following constant.

```
Public static final int TYPE_FORWARD_ONLY
```

ii)scrollable ResultSets:

These `ResultSet` objects will allow the users to interact the data in both forward and backward directions.

`Scrollable ResultSets` can be divided into following two types.

a)ScrollSensitive ResultSet:

It is a `ResultSet` object, it will allow the later database modifications.

To represent this ResultSet object we have to use the following constant from ResultSet interface.

```
Public static final TYPE_SCROLL_SENSITIVE
```

b)ScrollInSensitive ResultSet:

These are `scrollable ResultSetss` objects, which will allow the later database modifications after creation.

To represent this `ResultSet` object we have to use the following constant from `ResultSet` interface.

```
Public static final TYPE_SCROLL_INSENSITIVE
```

on the basis of ResultSet concurrency there are two types of ResultSets

i)Read only ResultSet:

This ResultSet will allow the users only to read the data.To represent this `ResultSet` object we have to use following constant from ResultSet interface.

```
Public static final int CONCUR_READ_ONLY
```

ii)updatable ResultSet:

This `ResultSet` object will allow user to perform updation on it's content.

To represent this `ResultSet` object we have to use following constant from `ResultSet` interface.

```
Public static final int CONCUR_UPDATABLE.
```

If you want to specify particular ResultSet type in JDBC applications then we have to specify the type of `ResultSet` object at the time of creation of statement object.

To achieve this we have to use the following method

```
public statement createStatement(int
forwardonly/ScrollSensitive/ScrollInsensitive,int
readonly/updatable)throws SQLException
```

Scrollable and Updatable Result Sets Method Details

In JDBC applications by using `ScrollSensitive ResultSet` object we are able to get the Database later modifications in ResultSet object automatically.

In case of `SrollSensitive ResultSet` object after performing updations at Database to reflect that updations into the `ResultSet` object record we have to refresh each and every row in ResultSet object.

To refresh present report a scroll sensitive `ResultSet` object we have to use the following method.

```
Public void refreshRow() throws SQLException
```

In case of `scrollable ResultSet` object to move `ResultSet` cursor before first record position we have to use following method

```
public void beforeFirst()
```

To move ResultSet cursor after last record position we have to use following method

```
public void afterLast()
```

To move ResultSet cursor to first record position we will use the following method

```
public boolean first()
```

To move ResultSet cursor to last record position we will use the following method

```
public boolean last()
```

To move ResultSet cursor to a particular record position we will use the following method

```
public boolean absolute(int rec_position)
```

To skip particular no.of records from the current position of the ResultSet we have to use the following method

```
public boolean relative(int no_of_records)
```

note:If the int value is positive then ResultSet cursor will move in forward direction, if it is negative then ResultSet cursor will move in backward direction.

To insert new row in updatable ResultSet object we have to use following method

```
public void moveToInsertRow()
```

To insertrecord data temporarily in a row we have to use the following method

```
public void updatexxx(int column_index,xxx value)
```

In order to make temporary insertion as permanent insertion in the ResultSet object and database we have to use following method.

```
public void insertRow()
```

Example program - 1 :

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;
import java.util.Properties;

public class JdbcApp14 {
public static void main(String args[])throws Exception
{
      Class.forName("com.mysql.jdbc.Driver");
      Properties p=new Properties();
      Connection
con=DriverManager.getConnection("jdbc:mysql://localhost:3306/test
","root","system");


Statement
st=con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,ResultSet.
CONCUR_UPDATABLE);
boolean b=st.execute("select * from emp1");
System.out.println(b);
ResultSet rs=st.getResultSet();
System.out.println("_________");
System.out.println("ENO   ENAME    ESAL");
while(rs.next())
{
System.out.println(rs.getString("eno")+"   "+rs.getString("ename"
```

```
)     +"   "+rs.getString("esal"));
      }
      System.out.println("application in pausing state");
      System.out.println("perform updations at database");
      System.in.read();
      rs.beforeFirst();
      System.out.println("data after updations");
      System.out.println("--------");
      System.out.println("ENO   ENAME    ESAL");
      System.out.println("--------");
      while(rs.next())
      {
            rs.refreshRow();
      System.out.println(rs.getString("eno")+"   "+rs.getString("ename"
)     +"   "+rs.getString("esal"));
      }
      con.close();
      }

      }
```

note:

In JDBC Applications ScrollSensitive ResultSet object is supported by type-1 driver provided by SUN Micro Systems, it could not be supported by type-4 driver provided by ORACLE

In JDBC Applications ScrollInsensitive ResultSet could not be supported by both type-1 driver driver provided by SUN Micro Systems and type-4 driver provided by ORACLE.

The next example demonstrates the advance features of the ResultSet.

Example program - 2 :

```
package com.visualbuilder;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.ResultSet;

import java.sql.Statement;

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

```
try {
        /** Loading the driver*/
        Class.forName("com.mysql.jdbc.Driver");

        /** Getting Connection*/
        Connection con =
        DriverManager.getConnection("jdbc:mysql://localhost:33
        06/test","root","root");
        /** Creating Statement*/

        Statement stmt =
        con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,Re
        sultSet.CONCUR_UPDATABLE);

        /** Getting ResultSet*/

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

        while(rs.next()) {
              System.out.print("Id is " + rs.getInt("id"));
              System.out.println(" Name is
              "+rs.getString("name"));
              }
        rs.absolute(1);
        rs.updateString(2,"Visual Builder");
        rs.updateRow();
        rs.beforeFirst();
        System.out.println("After Updation");
        while(rs.next()) {
              System.out.print("Id is " + rs.getInt("id"));
              System.out.println(" Name is
              "+rs.getString("name"));
              }

        /** Closing the Connection*/

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

Output:-

Id is 1 Name is Sun java

Id is 2 Name is Visual Builder

Id is 3 Name is Eclipse

Id is 4 Name is IBM

Id is 5 Name is Jakarta

After Updation

Id is 1 Name is Visual Builder

Id is 2 Name is Visual Builder

Id is 3 Name is Eclipse

Id is 4 Name is IBM

Id is 5 Name is Jakarta

8.6 Row sets

A `RowSet` object is a java bean component and extends the `ResultSet` interface Thus, it has a set of JavaBeans properties and follows the JavaBeans event model. A RowSet object's properties allow it to establish its own database connection and to execute its own query in order to fill itself with data

Types of Rowset

The RowSet is majorly classified into two types as per their properties:-

1. *Connected Rowset.*:- The connected rowset as the name suggests in connected to the database connection object like the resultset. The `JDBCRowSet` is the example of the connected RowSet.

2. *Disconnected RowSet:*- The disconnected RowSet only connects to the database whenever required and after finishing the interaction they close the database connection. So if the connection pool is minimally used in this case.

There are following ways to create the `JDBCRowSet`:-

1. **Passing the ResultSet**:- In this approach the data is populated in the object and then you can retrieve the data by using the getter methods as we did in case of the ResultSet.

Exmple:-

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

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

JdbcRowSet jdbcRs = new JdbcRowSetImpl(rs);
```

2. **Creating the default object**:- This approach is useful if you want to set the data sources dynamically. In this approach the *Database URL, username and password is explicitly set in the RowSetObject.*

Exmple:-

```
JdbcRowSet jdbcRs = new JdbcRowSetImpl();

jdbcRs.setUsername("user");

jdbcRs.setPassword("password");

jdbcRs.setUrl("jdbc:mySubprotocol:mySubname");

jdbcRs.setCommand("select * from EMP ");

jdbcRs.execute();
```

The following example will illustrate the basic operation using the `JDBCRowSet` interface.

```
import java.sql.SQLException;

import javax.sql.rowset.JdbcRowSet;

import com.sun.rowset.JdbcRowSetImpl;

public class JDBCRowSetExample {

     public static void main(String[] args) throws SQLException{

          JdbcRowSet jdbcRs = new JdbcRowSetImpl();

          jdbcRs.setUsername("scott");
```

```
        jdbcRs.setPassword("tiger");

        jdbcRs.setUrl("jdbc:odbc:MyDsn");

        jdbcRs.setCommand("select * from employee");

        jdbcRs.execute();

        while(jdbcRs.next()) {

              System.out.println(jdbcRs.getString("ename"));
               }
           }
      }
```

Ouput:-

Adam

Susan

Adran

Hardy

Tim

Michael

8.7 metadata

Definition:

Data about the data is called as metadata.In JDBC there are two types of metadata.

1.Database metadata

2.ResultSet metadata.

JDBC Meta Data is the collective information about the data structure and property of a column available in table. The meta data of any table tells you the name of the columns,datatype used in column and constraint used to enter the value of data into column of the table.

Understand with Example

In this program, the code describe you JDBC Meta Data Get tables that explain the column property and structure of table. The class `Jdbc MetaDataGettables` include the list of methods to get the meta data property of table as given follow -

Loading a driver by calling a `class.forname( )`,this accept driver class as argument.

DriverManager.getConnection ( ) -This method return you a connection object and built a connection between url and database. Once a connection is built, a front end can access, insert ,update and retrieve the data in the backend database.

con.createStatement ( ) -This is used to create a sql object. An object of connection class is used to send and create a sql query in the database backend.

executeQuery ( ) -The method retrieve a record set from a table in database. The retrieve record set is assigned to a result set.

getMetaData ( ) - The Result Set call `get Metadata( )`,which return you the property of the retrieve record set (length,field,column).Meta Data account for data element and its attribute.

getcolumncount ( ) -The method return you a integer data type and provides you the number of column in the Result set object.

Finally the println print the table name, field, size and data type.

In case there is an exception in the try block The subsequent catch block caught and handle the exception.

JdbcMetaDataGettables.java

```
import java.sql.*;
public class JdbcMetaDataGettables {
        static public final String driver =
"com.mysql.jdbc.Driver";
        static public final String connection =
        "jdbc:mysql://localhost:3306/test";
        static public final String user = "root";
        static public final String password = "root";
        public static void main(String args[]) {
        try {
        Class.forName(driver);
```

```
            Connection con =
            DriverManager.getConnection(connection, user,password);
            Statement st = con.createStatement();
            String sql = "select * from person";
            ResultSet rs = st.executeQuery(sql);
            ResultSetMetaData metaData = rs.getMetaData();
            int rowCount = metaData.getColumnCount();
            System.out.println("Table Name : " +
            metaData.getTableName(2));
            System.out.println("Field \tsize\tDataType");
                  for (int i = 0; i < rowCount; i++) {
                  System.out.print(metaData.getColumnName(i + 1) +
"\t");
                  System.out.print(metaData.getColumnDisplayS ize(i + 1)
                  +"\t");
                  System.out.println(metaData.getColumnTypeNa me(i +
1));
                  }
            } catch (Exception e) {
            System.out.println(e);
            }
      }
}
```

Output:-

Table Name:person

Field Size DataTypes

id	2	VARCHAR
cname	50	VARCHAR
dob	10	DATE

8.8 Transaction

Transaction Management in JDBC

Transaction represents **a single unit of work.**

The ACID properties describes the transaction management well. **ACID stands for Atomicity, Consistency, isolation and durability.**

Atomicity means either all successful or none.

Consistency ensures bringing the database from one consistent state to another consistent state.

Isolation ensures that transaction is isolated from other transaction.

Durability means once a transaction has been committed, it will remain so, even in the event of errors, power loss etc.

Advantage of Transaction Mangaement

fast performance It makes the performance fast because database is hit at the time of commit.

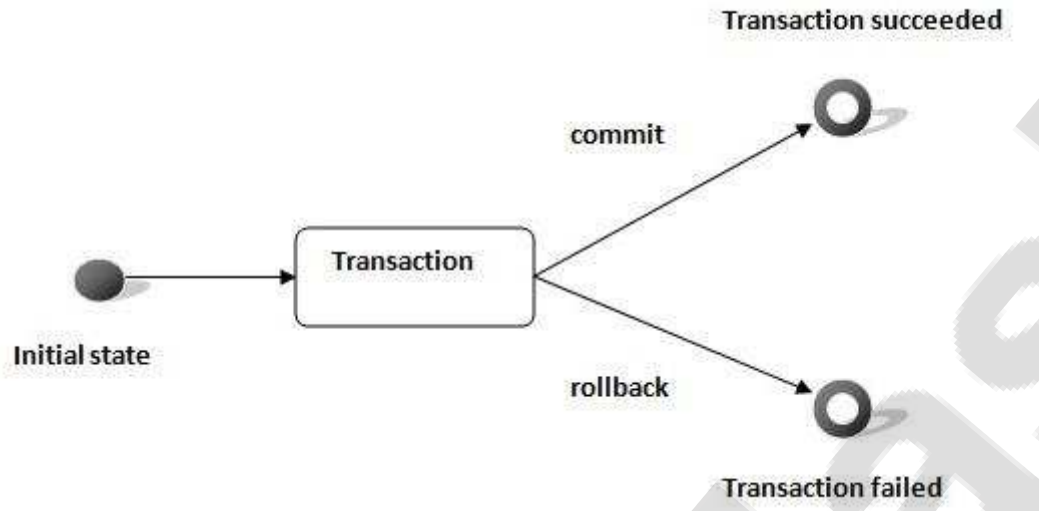

Fig. Transaction Management in JDBC

In JDBC, **Connection interface** provides methods to manage transaction.

Method	Description
void setAutoCommit(boolean status)	It is true bydefault means each transaction is committed bydefault.
void commit()	commits the transaction.
void rollback()	cancels the transaction.

Simple example of transaction management in jdbc using Statement

Let's see the simple example of transaction management using Statement.

```
import java.sql.*;

class FetchRecords{

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");

con.setAutoCommit(false);

Statement stmt=con.createStatement();

stmt.executeUpdate("insert into user420 values(190,'abhi',4000
   0)");

stmt.executeUpdate("insert into user420 values(191,'umesh',500
   00)");

con.commit();

con.close();

}}
```

If you see the table emp400, you will see that 2 records has been added.

Example of transaction management in jdbc using PreparedStatement

Let's see the simple example of transaction management using PreparedStatement.

```
import java.sql.*;

import java.io.*;

class TM{

public static void main(String args[]){

try{

Class.forName("oracle.jdbc.driver.OracleDriver");

Connection con=DriverManager.getConnection("jdbc:oracle:thi
n:@localhost:1521:xe","system","oracle");

con.setAutoCommit(false);

PreparedStatement ps=con.prepareStatement("insert into use4
20 values(?,?,?)");
```

```
BufferedReader br=new BufferedReader(new InputStreamReader(
System.in));

while(true){

System.out.println("enter id");

String s1=br.readLine();

int id=Integer.parseInt(s1);

System.out.println("enter name");

String name=br.readLine();

System.out.println("enter salary");

String s3=br.readLine();

int salary=Integer.parseInt(s3);

ps.setInt(1,id);

ps.setString(2,name);

ps.setInt(3,salary);

ps.executeUpdate();

System.out.println("commit/rollback");

String answer=br.readLine();

if(answer.equals("commit")){

con.commit();

}

if(answer.equals("rollback")){

con.rollback();

}

System.out.println("Want to add more records y/n");
```

```
String ans=br.readLine();

if(ans.equals("n")){

break;

}

}

con.commit();

System.out.println("record successfully saved");

con.close();//before closing connection commit() is called

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

}}
```

It will ask to add more records until you press n. If you press n, transaction is committed.