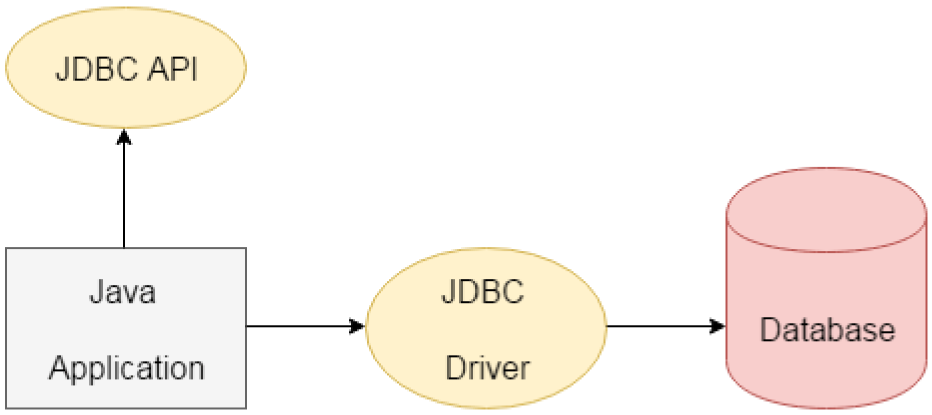
Java JDBC Tutorial

JDBC stands for Java Database Connectivity. JDBC is a Java API to connect and execute the query with the database. It is a part of JavaSE (Java Standard Edition). JDBC API uses JDBC drivers to connect with the database. There are four types of JDBC drivers:

- JDBC-ODBC Bridge Driver,
- Native Driver,
- Network Protocol Driver, and
- Thin Driver

We have discussed the above four drivers in the next chapter.

We can use JDBC API to access tabular data stored in any relational database. By the help of JDBC API, we can save, update, delete and fetch data from the database. It is like Open Database Connectivity (ODBC) provided by Microsoft.



The current version of JDBC is 4.3. It is the stable release since 21st September, 2017. It is based on the X/Open SQL Call Level Interface. The **java.sql** package contains classes and interfaces for JDBC API. A list of popular *interfaces* of JDBC API are given below:

- Driver interface
- Connection interface
- Statement interface
- PreparedStatement interface
- CallableStatement interface
- ResultSet interface
- ResultSetMetaData interface
- DatabaseMetaData interface
- RowSet interface

A list of popular *classes* of JDBC API are given below:

- DriverManager class
- Blob class
- Clob class
- Types class

Why Should We Use JDBC

Before JDBC, ODBC API was the database API to connect and execute the query with the database. But, ODBC API uses ODBC driver which is written in C language (i.e. platform dependent and unsecured). That is why Java has defined its own API (JDBC API) that uses JDBC drivers (written in Java language).

We can use JDBC API to handle database using Java program and can perform the following activities:

1. Connect to the database
2. Execute queries and update statements to the database
3. Retrieve the result received from the database.

What is API

API (Application programming interface) is a document that contains a description of all the features of a product or software. It represents classes and interfaces that software programs can follow to communicate with each other. An API can be created for applications, libraries, operating systems, etc.

JDBC Driver

JDBC Driver is a software component that enables java application to interact with the database. There are 4 types of JDBC drivers:

1. JDBC-ODBC bridge driver
2. Native-API driver (partially java driver)
3. Network Protocol driver (fully java driver)
4. Thin driver (fully java driver)

1) JDBC-ODBC bridge driver

The JDBC-ODBC bridge driver uses ODBC driver to connect to the database. The JDBC-ODBC bridge driver converts JDBC method calls into the ODBC function calls. This is now discouraged because of thin driver.

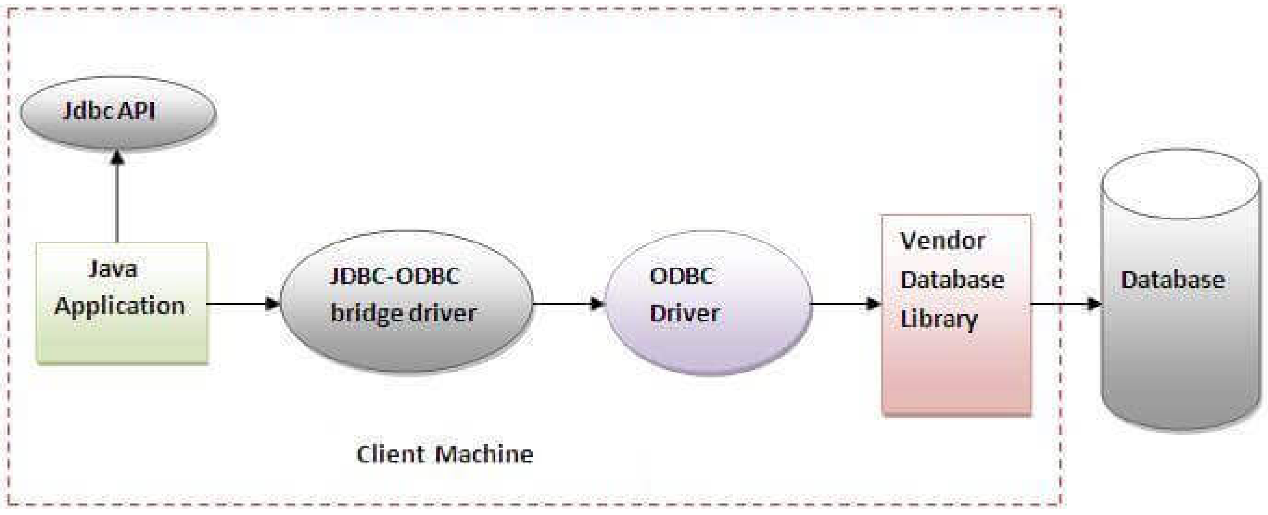


Figure- JDBC-ODBC Bridge Driver

In Java 8, the JDBC-ODBC Bridge has been removed.

Oracle does not support the JDBC-ODBC Bridge from Java 8. Oracle recommends that you use JDBC drivers provided by the vendor of your database instead of the JDBC-ODBC Bridge.

Advantages:

- easy to use.
- can be easily connected to any database.

Disadvantages:

- Performance degraded because JDBC method call is converted into the ODBC function calls.
- The ODBC driver needs to be installed on the client machine.

---

2) Native-API driver

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

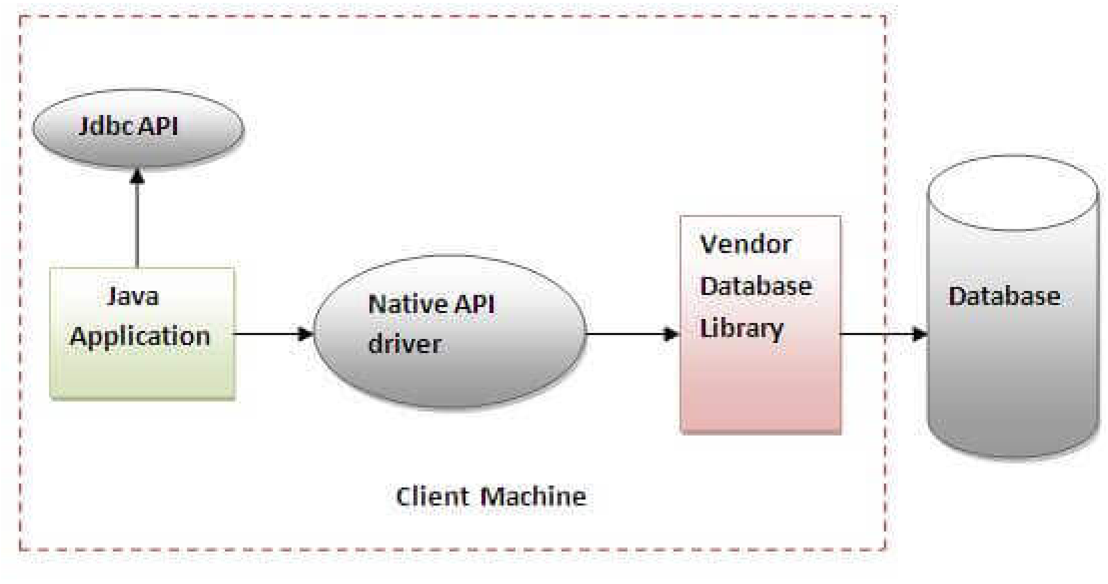


Figure- Native API Driver

Advantage:

- performance upgraded than JDBC-ODBC bridge driver.

Disadvantage:

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

---

3) Network Protocol driver

The Network Protocol driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol. It is fully written in java.

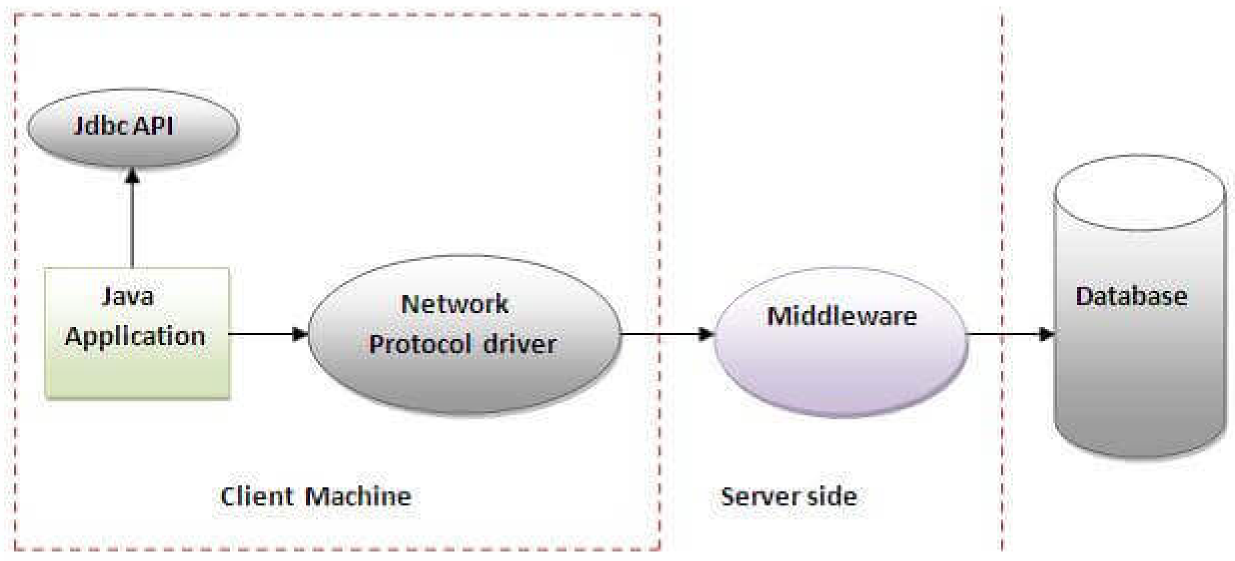


Figure- Network Protocol Driver

Advantage:

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

Disadvantages:

- Network support is required on client machine.
- Requires database-specific coding to be done in the middle tier.
- Maintenance of Network Protocol driver becomes costly because it requires database-specific coding to be done in the middle tier.

---

4) Thin driver

The thin driver converts JDBC calls directly into the vendor-specific database protocol. That is why it is known as thin driver. It is fully written in Java language.

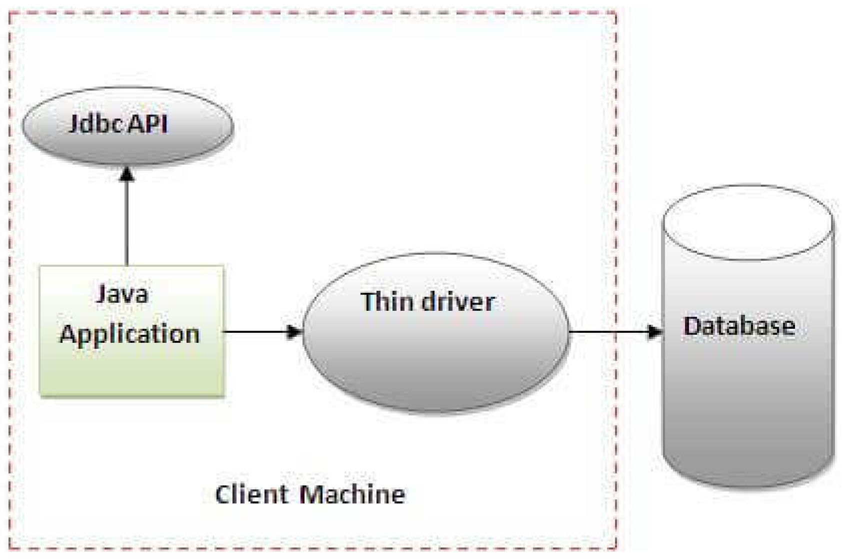


Figure- Thin Driver

Advantage:

- Better performance than all other drivers.
- No software is required at client side or server side.

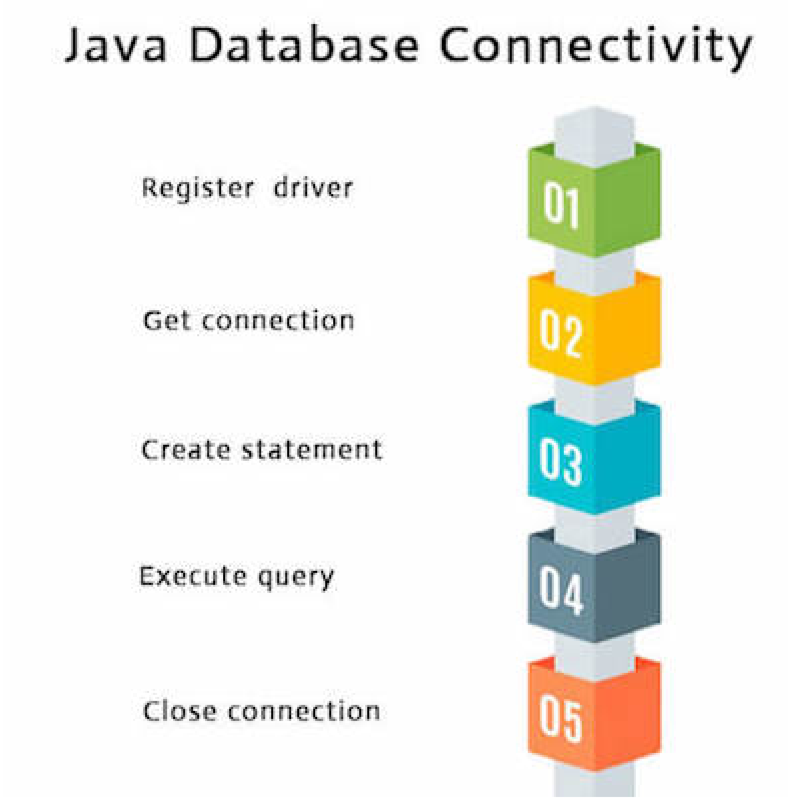
Disadvantage:

- Drivers depend on the Database.

Java Database Connectivity with 5 Steps

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

- Register the Driver class
- Create connection
- Create statement
- Execute queries
- Close connection



1) Register the driver class

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

Syntax of forName() method

```
public static void forName(String className)throws ClassNotFoundException
```

Note: Since JDBC 4.0, explicitly registering the driver is optional. We just need to put vender's Jar in the classpath, and then JDBC driver manager can detect and load the driver automatically.

Example to register the OracleDriver class

Here, Java program is loading oracle driver to esteblish database connection.

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

---

2) Create the connection object

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

Syntax of getConnection() method

```
1) public static Connection getConnection(String url)throws SQLException
2) public static Connection getConnection(String url,String name,String password)
throws SQLException
```

Example to establish connection with the Oracle database

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

---

3) Create the Statement object

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

Syntax of createStatement() method

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

Example to create the statement object

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

---

4) Execute the query

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

Syntax of executeQuery() method

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

Example to execute query

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

while(rs.next()){
```

```
4. System.out.println(rs.getInt(1)+" "+rs.getString(2));
5. }
```

---

5) Close the connection object

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

Syntax of close() method

```
1. public void close()throws SQLException
```

Example to close connection

```
1. con.close();
```

Note: Since Java 7, JDBC has ability to use try-with-resources statement to automatically close resources of type Connection, ResultSet, and Statement.

It avoids explicit connection closing step.

Java Database Connectivity with MySQL

To connect Java application with the MySQL database, we need to follow 5 following steps.

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

1. **Driver class:** The driver class for the mysql database is **com.mysql.jdbc.Driver**.
2. **Connection URL:** The connection URL for the mysql database is **jdbc:mysql://localhost:3306/sonoo** where jdbc is the API, mysql is the database, localhost is the server name on which mysql is running, we may also use IP address, 3306 is the port number and sonoo is the database name. We may use any database, in such case, we need to replace the sonoo with our database name.
3. **Username:** The default username for the mysql database is **root**.
4. **Password:** It is the password given by the user at the time of installing the mysql database. In this example, we are going to use root as the password.

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

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

Example to Connect Java Application with mysql database

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

```
import java.sql.*;
class MysqlCon{
public static void main(String args[]){
try{
Class.forName("com.mysql.jdbc.Driver");
Connection con=DriverManager.getConnection(
"jdbc:mysql://localhost:3306/sonoo","root","root");
//here sonoo is database name, root is username and password
Statement stmt=con.createStatement();
ResultSet rs=stmt.executeQuery("select * from emp");
while(rs.next())
System.out.println(rs.getInt(1)+"  "+rs.getString(2)+"  "+rs.getString(3));
con.close();
}catch(Exception e){ System.out.println(e);}
}
}
```

The above example will fetch all the records of emp table.

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

Two ways to load the jar file:

1. Paste the mysqlconnector.jar file in jre/lib/ext folder
2. Set classpath

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

Download the mysqlconnector.jar file. Go to jre/lib/ext folder and paste the jar file here.

2) Set classpath:

There are two ways to set the classpath:

- temporary
- permanent

How to set the temporary classpath

open command prompt and write:

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

How to set the permanent classpath

Go to environment variable then click on new tab. In variable name write **classpath** and in variable value paste the path to the mysqlconnector.jar file by appending mysqlconnector.jar;.; as C:\folder\mysql-connector-java-5.0.8-bin.jar;.;

Connectivity with Access without DSN

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

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

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

Example to Connect Java Application with access without DSN

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

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

   String url="jdbc:odbc:Driver={Microsoft Access Driver (*.mdb)};
                    DBQ=" + database + ";DriverID=22;READONLY=true";

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

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

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

}}
```

Example to Connect Java Application with access with DSN

Connectivity with type1 driver is not considered good. To connect java application with type1 driver, create DSN first, here we are assuming your dsn name is mydsn.

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

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

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

}}
```

DriverManager class

The DriverManager class is the component of JDBC API and also a member of the *java.sql* package. The DriverManager class acts as an interface between users and drivers. It keeps track of the drivers that are available and handles establishing a connection between a database and the appropriate driver. It contains all the appropriate methods to register and deregister the database driver class and to create a connection between a Java application and the database. The DriverManager class maintains a list of Driver classes that have registered themselves by calling the method DriverManager.registerDriver(). Note that before interacting with a Database, it is a mandatory process to register the driver; otherwise, an exception is thrown.

Methods of the DriverManager Class

Method	Description
1) public static synchronized void registerDriver(Driver driver):	is used to register the given driver with DriverManager. No action is performed by the method when the given driver is already registered.
2) public static synchronized void deregisterDriver(Driver driver):	is used to deregister the given driver (drop the driver from the list) with DriverManager. If the given driver has been removed from the list, then no action is performed by the method.
3) public static Connection getConnection(String url) throws SQLException:	is used to establish the connection with the specified url. The SQLException is thrown when the corresponding Driver class of the given database is not registered with the DriverManager.
4) public static Connection getConnection(String url,String userName,String password) throws SQLException:	is used to establish the connection with the specified url, username, and password. The SQLException is thrown when the corresponding Driver class of the given database is not registered with the DriverManager.
5) public static Driver getDriver(String url)	Those drivers that understand the mentioned URL (present in the parameter of the method) are returned by this method provided those drivers are mentioned in the list of registered drivers.
6) pubic static int getLoginTimeout()	The duration of time a driver is allowed to wait in order to establish a connection with the database is returned by this method.
7) pubic static void setLoginTimeout(int sec)	The method provides the time in seconds. sec mentioned in the parameter is the maximum time that a driver is allowed to wait in order to establish a connection with the database. If 0 is passed in the parameter of this method, the driver will have to wait infinitely while trying to establish the connection with the database.
8) public static Connection getConnection(String	A connection object is returned by this method after creating a connection to the database present at the mentioned URL, which is the first parameter of this method. The second parameter, which is "prop", fetches the authentication details of the database (username and

URL, Properties prop) throws SQLException	password.). Similar to the other variation of the getConnection() method, this method also throws the SQLException, when the corresponding Driver class of the given database is not registered with the DriverManager.

Connection interface

A Connection is a session between a Java application and a database. It helps to establish a connection with the database.

The Connection interface is a factory of Statement, PreparedStatement, and DatabaseMetaData, i.e., an object of Connection can be used to get the object of Statement and DatabaseMetaData. The Connection interface provide many methods for transaction management like commit(), rollback(), setAutoCommit(), setTransactionIsolation(), etc.

By default, connection commits the changes after executing queries.

Commonly used methods of Connection interface:

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

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

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

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

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

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

Connection Interface Fields

There are some common Connection interface constant fields that are present in the Connect interface. These fields specify the isolation level of a transaction.

TRANSACTION_NONE: No transaction is supported, and it is indicated by this constant.

TRANSACTION_READ_COMMITTED: It is a constant which shows that the dirty reads are not allowed. However, phantom reads and non-repeatable reads can occur.

TRANSACTION_READ_UNCOMMITTED: It is a constant which shows that dirty reads, non-repeatable reads, and phantom reads can occur.

TRANSACTION_REPEATABLE_READ: It is a constant which shows that the non-repeatable reads and dirty reads are not allowed. However, phantom reads and can occur.

TRANSACTION_SERIALIZABLE: It is a constant which shows that the non-repeatable reads, dirty reads as well as the phantom reads are not allowed.

Statement interface

The **Statement interface** provides methods to execute queries with the database. The statement interface is a factory of ResultSet i.e. it provides factory method to get the object of ResultSet.

Commonly used methods of Statement interface:

The important methods of Statement interface are as follows:

1) public ResultSet executeQuery(String sql): is used to execute SELECT query. It returns the object of ResultSet.
2) public int executeUpdate(String sql): is used to execute specified query, it may be create, drop, insert, update, delete etc.
3) public boolean execute(String sql): is used to execute queries that may return multiple results.
4) public int[] executeBatch(): is used to execute batch of commands.

Example of Statement interface

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

```
import java.sql.*;
class FetchRecord{
public static void main(String args[])throws Exception{
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
Statement stmt=con.createStatement();

//stmt.executeUpdate("insert into emp765 values(33,'Irfan',50000)");
//int result=stmt.executeUpdate("update emp765 set name='Vimal',salary=10000 where id=33");
int result=stmt.executeUpdate("delete from emp765 where id=33");
System.out.println(result+" records affected");
con.close();
}}
```

ResultSet interface

The object of ResultSet maintains a cursor pointing to a row of a table. Initially, cursor points to before the first row.

By default, ResultSet object can be moved forward only and it is not updatable.

But we can make this object to move forward and backward direction by passing either TYPE_SCROLL_INSENSITIV
TYPE_SCROLL_SENSITIVE in createStatement(int,int) method as well as we can make this object as updatable by:

```
Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                    ResultSet.CONCUR_UPDATABLE);
```

Commonly used methods of ResultSet interface

1) public boolean next():	is used to move the cursor to the one row next from the current position.
2) public boolean previous():	is used to move the cursor to the one row previous from the current position.
3) public boolean first():	is used to move the cursor to the first row in result set object.
4) public boolean last():	is used to move the cursor to the last row in result set object.
5) public boolean absolute(int row):	is used to move the cursor to the specified row number in the ResultSet obje
6) public boolean relative(int row):	is used to move the cursor to the relative row number in the ResultSet object may be positive or negative.
7) public int getInt(int columnIndex):	is used to return the data of specified column index of the current row as int.
8) public int getInt(String columnName):	is used to return the data of specified column name of the current row as int.
9) public String getString(int columnIndex):	is used to return the data of specified column index of the current row as Stri
10) public String getString(String columnName):	is used to return the data of specified column name of the current row as Stri

Example of Scrollable ResultSet

Let’s see the simple example of ResultSet interface to retrieve the data of 3rd row.

```
import java.sql.*;
class FetchRecord{
public static void main(String args[])throws Exception{

Class.forName("oracle.jdbc.driver.OracleDriver");
Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
Statement stmt=con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,ResultSet.CONCUR_UPDATABL
ResultSet rs=stmt.executeQuery("select * from emp765");

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

con.close();
}}
```

PreparedStatement interface

The PreparedStatement interface is a subinterface of Statement. It is used to execute parameterized query.

Let's see the example of parameterized query:

```
1. String sql="insert into emp values(?,?,?)";
```

As you can see, we are passing parameter (?) for the values. Its value will be set by calling the setter methods of PreparedStatement.

Why use PreparedStatement?

Improves performance: The performance of the application will be faster if you use PreparedStatement interface because query is compiled only once.

How to get the instance of PreparedStatement?

The prepareStatement() method of Connection interface is used to return the object of PreparedStatement. Syntax:

```
1. public PreparedStatement prepareStatement(String query)throws SQLException{}
```

Methods of PreparedStatement interface

The important methods of PreparedStatement interface are given below:

Method	Description
public void setInt(int paramIndex, int value)	sets the integer value to the given parameter index.
public void setString(int paramIndex, String value)	sets the String value to the given parameter index.
public void setFloat(int paramIndex, float value)	sets the float value to the given parameter index.
public void setDouble(int paramIndex, double value)	sets the double value to the given parameter index.
public int executeUpdate()	executes the query. It is used for create, drop, insert, update, delete etc.
public ResultSet executeQuery()	executes the select query. It returns an instance of ResultSet.

Example of PreparedStatement interface that inserts the record

First of all create table as given below:

```
1. create table emp(id number(10),name varchar2(50));
```

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

```
1. import java.sql.*;
2. class InsertPrepared{
3. public static void main(String args[]){
4. try{
5. Class.forName("oracle.jdbc.driver.OracleDriver");
6.
7. Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
PreparedStatement stmt=con.prepareStatement("insert into Emp values(?,?)");
stmt.setInt(1,101);//1 specifies the first parameter in the query
stmt.setString(2,"Ratan");

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

con.close();

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

}
}
```

Example of PreparedStatement interface that updates the record

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

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

Example of PreparedStatement interface that deletes the record

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

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

Example of PreparedStatement interface that retrieve the records of a table

```
PreparedStatement stmt=con.prepareStatement("select * from emp");
ResultSet rs=stmt.executeQuery();
while(rs.next()){
System.out.println(rs.getInt(1)+" "+rs.getString(2));
}
```

Example of PreparedStatement to insert records until user press n

```
import java.sql.*;
import java.io.*;
class RS{
public static void main(String args[])throws Exception{
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","oracle");

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

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

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

ps.setInt(1,id);
ps.setString(2,name);
ps.setFloat(3,salary);
int i=ps.executeUpdate();
System.out.println(i+" records affected");

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

con.close();
}}
```

Java ResultSetMetaData Interface

The metadata means data about data i.e. we can get further information from the data.

If you have to get metadata of a table like total number of column, column name, column type etc. , ResultSetMetaData interface is useful because it provides methods to get metadata from the ResultSet object.

Commonly used methods of ResultSetMetaData interface

Method	Description
public int getColumnCount()throws SQLException	it returns the total number of columns in the ResultSet object.
public String getColumnName(int index)throws SQLException	it returns the column name of the specified column index.
public String getColumnTypeName(int index)throws SQLException	it returns the column type name for the specified index.
public String getTableName(int index)throws SQLException	it returns the table name for the specified column index.

How to get the object of ResultSetMetaData:

The getMetaData() method of ResultSet interface returns the object of ResultSetMetaData. Syntax:

```
public ResultSetMetaData getMetaData()throws SQLException
```

Example of ResultSetMetaData interface :

```
import java.sql.*;
class Rsmd{
public static void main(String args[]){
try{
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection con=DriverManager.getConnection(
"jdbc:oracle:thin:@localhost:1521:xe","system","oracle");

PreparedStatement ps=con.prepareStatement("select * from emp");
ResultSet rs=ps.executeQuery();
ResultSetMetaData rsmd=rs.getMetaData();

System.out.println("Total columns: "+rsmd.getColumnCount());
System.out.println("Column Name of 1st column: "+rsmd.getColumnName(1));
System.out.println("Column Type Name of 1st column: "+rsmd.getColumnTypeName(1));

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

```
Output:Total columns: 2
       Column Name of 1st column: ID
       Column Type Name of 1st column: NUMBER
```

Java DatabaseMetaData interface

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

Commonly used methods of DatabaseMetaData interface

- **public String getDriverName()throws SQLException:** it returns the name of the JDBC driver.
- **public String getDriverVersion()throws SQLException:** it returns the version number of the JDBC driver.
- **public String getUserName()throws SQLException:** it returns the username of the database.
- **public String getDatabaseProductName()throws SQLException:** it returns the product name of the database.
- **public String getDatabaseProductVersion()throws SQLException:** it returns the product version of the database.
- **public ResultSet getTables(String catalog, String schemaPattern, String tableNamePattern, String[] types)throws SQLException:** it returns the description of the tables of the specified catalog. The table type can be TABLE, VIEW, ALIAS, SYSTEM TABLE, SYNONYM etc.

How to get the object of DatabaseMetaData:

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

```
public DatabaseMetaData getMetaData()throws SQLException
```

Simple Example of DatabaseMetaData interface :

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

Connection con=DriverManager.getConnection(
"jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
DatabaseMetaData dbmd=con.getMetaData();

System.out.println("Driver Name: "+dbmd.getDriverName());
System.out.println("Driver Version: "+dbmd.getDriverVersion());
System.out.println("UserName: "+dbmd.getUserName());
System.out.println("Database Product Name: "+dbmd.getDatabaseProductName());
System.out.println("Database Product Version: "+dbmd.getDatabaseProductVersion());

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

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

Example of DatabaseMetaData interface that prints total number of tables :

```
import java.sql.*;
class Dbmd2{
public static void main(String args[]){
try{
Class.forName("oracle.jdbc.driver.OracleDriver");

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

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

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

con.close();

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

}
}
```

Example of DatabaseMetaData interface that prints total number of views :

```
import java.sql.*;
class Dbmd3{
public static void main(String args[]){
try{
Class.forName("oracle.jdbc.driver.OracleDriver");

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

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

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

con.close();

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

}
}
```

Example to store image in Oracle database

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

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

Signature of setBinaryStream method

The syntax of setBinaryStream() method is given below:

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

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

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

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

Java Example to store image in the database

```
import java.sql.*;
import java.io.*;
public class InsertImage {
public static void main(String[] args) {
try{
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection con=DriverManager.getConnection(
"jdbc:oracle:thin:@localhost:1521:xe","system","oracle");

PreparedStatement ps=con.prepareStatement("insert into imgtable values(?,?)");
ps.setString(1,"sonoo");

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

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

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

Example to retrieve image from Oracle database

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

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

Signature of getBlob() method of PreparedStatement

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

Signature of getBytes() method of Blob interface

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

We are assuming that image is stored in the imgtable.

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

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

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

```
import java.sql.*;
import java.io.*;
public class RetrieveImage {
public static void main(String[] args) {
try{
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection con=DriverManager.getConnection(
"jdbc:oracle:thin:@localhost:1521:xe","system","oracle");

PreparedStatement ps=con.prepareStatement("select * from imgtable");
ResultSet rs=ps.executeQuery();
if(rs.next()){//now on 1st row

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

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

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

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

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

Example to store file in Oracle database:

The setCharacterStream() method of PreparedStatement is used to set character information into the parameterIndex.

Syntax:

1) public void setBinaryStream(int paramIndex,InputStream stream)throws SQLException

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

For storing file into the database, CLOB (Character Large Object) datatype is used in the table. For example:

```
CREATE TABLE  "FILETABLE"
   (    "ID" NUMBER,
    "NAME" CLOB
   )
/
```

Java Example to store file in database

```
import java.io.*;
import java.sql.*;

public class StoreFile {
public static void main(String[] args) {
try{
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection con=DriverManager.getConnection(
"jdbc:oracle:thin:@localhost:1521:xe","system","oracle");

PreparedStatement ps=con.prepareStatement(
"insert into filetable values(?,?)");

File f=new File("d:\\myfile.txt");
FileReader fr=new FileReader(f);

ps.setInt(1,101);
ps.setCharacterStream(2,fr,(int)f.length());
int i=ps.executeUpdate();
System.out.println(i+" records affected");

con.close();

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

Example to retrieve file from Oracle database:

The getClob() method of PreparedStatement is used to get file information from the database.

Syntax of getClob method

```
public Clob getClob(int columnIndex){}
```

Let's see the table structure of this example to retrieve the file.

```
CREATE TABLE  "FILETABLE"
   (    "ID" NUMBER,
    "NAME" CLOB
   )
/
```

The example to retrieve the file from the Oracle database is given below.

```
import java.io.*;
import java.sql.*;

public class RetrieveFile {
public static void main(String[] args) {
try{
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection con=DriverManager.getConnection(
"jdbc:oracle:thin:@localhost:1521:xe","system","oracle");

PreparedStatement ps=con.prepareStatement("select * from filetable");
ResultSet rs=ps.executeQuery();
rs.next();//now on 1st row

Clob c=rs.getClob(2);
Reader r=c.getCharacterStream();

FileWriter fw=new FileWriter("d:\\retrivefile.txt");

int i;
while((i=r.read())!=-1)
fw.write((char)i);

fw.close();
con.close();

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

Java CallableStatement Interface

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

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.

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.

How to get the instance of CallableStatement?

The prepareCall() method of Connection interface returns the instance of CallableStatement. Syntax is given below:

```
public CallableStatement prepareCall("{ call procedurename(?,?...?)}");
```

The example to get the instance of CallableStatement is given below:

```
CallableStatement stmt=con.prepareCall("{call myprocedure(?,?)}");
```

It calls the procedure myprocedure that receives 2 arguments.

Full example to call the stored procedure using JDBC

To call the stored procedure, you need to create it in the database. Here, we are assuming that stored procedure looks like this.

```
create or replace procedure "INSERTR"
(id IN NUMBER,
name IN VARCHAR2)
is
begin
```

```
insert into user420 values(id,name);
end;
/
```

The table structure is given below:

```
create table user420(id number(10), name varchar2(200));
```

In this example, we are going to call the stored procedure INSERTR that receives id and name as the parameter and inserts it into the table user420. Note that you need to create the user420 table as well to run this application.

```
import java.sql.*;
public class Proc {
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");

CallableStatement stmt=con.prepareCall("{call insertR(?,?)}");
stmt.setInt(1,1011);
stmt.setString(2,"Amit");
stmt.execute();

System.out.println("success");
}
}
```

Now check the table in the database, value is inserted in the user420 table.

Example to call the function using JDBC

In this example, we are calling the sum4 function that receives two input and returns the sum of the given number. Here, we have used the **registerOutParameter** method of CallableStatement interface, that registers the output parameter with its corresponding type. It provides information to the CallableStatement about the type of result being displayed.

The **Types** class defines many constants such as INTEGER, VARCHAR, FLOAT, DOUBLE, BLOB, CLOB etc.

Let's create the simple function in the database first.

```
create or replace function sum4
(n1 in number,n2 in number)
return number
is
temp number(8);
begin
temp :=n1+n2;
return temp;
end;
/
```

Now, let's write the simple program to call the function.

```
import java.sql.*;

public class FuncSum {
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");

CallableStatement stmt=con.prepareCall("{?= call sum4(?,?)}");
stmt.setInt(2,10);
stmt.setInt(3,43);
stmt.registerOutParameter(1,Types.INTEGER);
stmt.execute();

System.out.println(stmt.getInt(1));

}
}
```

```
Output: 53
```

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.

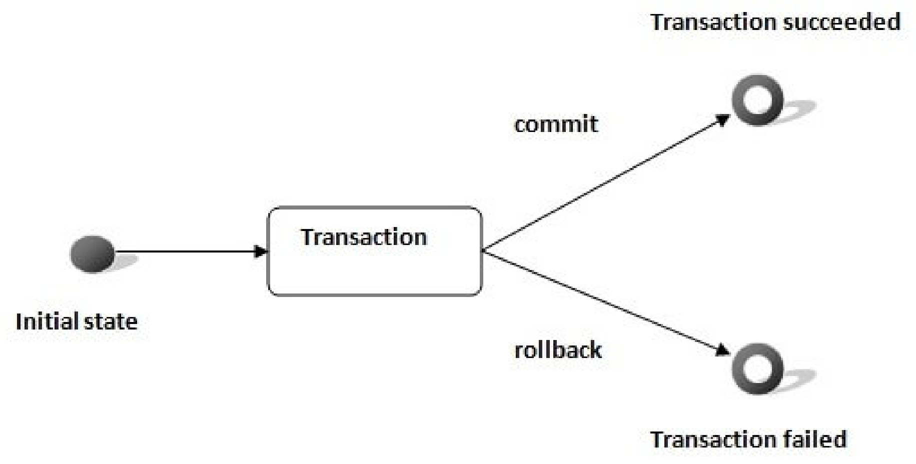
Consistency ensures bringing the database from one consistent state to another consistent state.

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.



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',40000)");
stmt.executeUpdate("insert into user420 values(191,'umesh',50000)");

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:thin:@localhost:1521:xe","system","oracle");
con.setAutoCommit(false);

PreparedStatement ps=con.prepareStatement("insert into user420 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.

Batch Processing in JDBC

Instead of executing a single query, we can execute a batch (group) of queries. It makes the performance fast. It is because when one sends multiple statements of SQL at once to the database, the communication overhead is reduced significantly, as one is not communicating with the database frequently, which in turn results to fast performance.

The java.sql.Statement and java.sql.PreparedStatement interfaces provide methods for batch processing.

Advantage of Batch Processing

Fast Performance

Methods of Statement interface

The required methods for batch processing are given below:

Method	Description
void addBatch(String query)	The addBatch(String query) method of the CallableStatement, PreparedStatement, and Statement is used to single statements to a batch.
int[] executeBatch()	The executeBatch() method begins the execution of all the grouped together statements. The method returns an integer array, and each of the element of the array represents the updated count for respective update statement.
boolean DatabaseMetaData.supportsBatchUpdates() throws SQLException	If the target database facilitates the batch update processing, then the method returns true.
void clearBatch()	The method removes all the statements that one has added using the addBatch() method.

Example of batch processing in JDBC

Let's see the simple example of batch processing in JDBC. It follows following steps:

- Load the driver class
- Create Connection
- Create Statement
- Add query in the batch
- Execute Batch
- Close Connection

FileName: FetchRecords.java

```
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.addBatch("insert into user420 values(190,'abhi',40000)");
stmt.addBatch("insert into user420 values(191,'umesh',50000)");

stmt.executeBatch();//executing the batch

con.commit();
con.close();
}}
```

If you see the table user420, two records have been added.

Example of batch processing using PreparedStatement

FileName: BP.java

```
import java.sql.*;
import java.io.*;
class BP{
public static void main(String args[]){
try{

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

PreparedStatement ps=con.prepareStatement("insert into user420 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.addBatch();
System.out.println("Want to add more records y/n");
String ans=br.readLine();
if(ans.equals("n")){
break;
}

}
ps.executeBatch();// for executing the batch

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

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

}}
```

Output:

```
enter id
101
enter name
Manoj Kumar
```

```
enter salary
10000
Want to add more records y/n
y
enter id
101
enter name
Harish Singh
enter salary
15000
Want to add more records y/n
y
enter id
103
enter name
Rohit Anuragi
enter salary
30000
Want to add more records y/n
y
enter id
104
enter name
Amrit Gautam
enter salary
40000
Want to add more records y/n
n
record successfully saved
```

It will add the queries into the batch until user press n. Finally, it executes the batch. Thus, all the added queries will be fired.

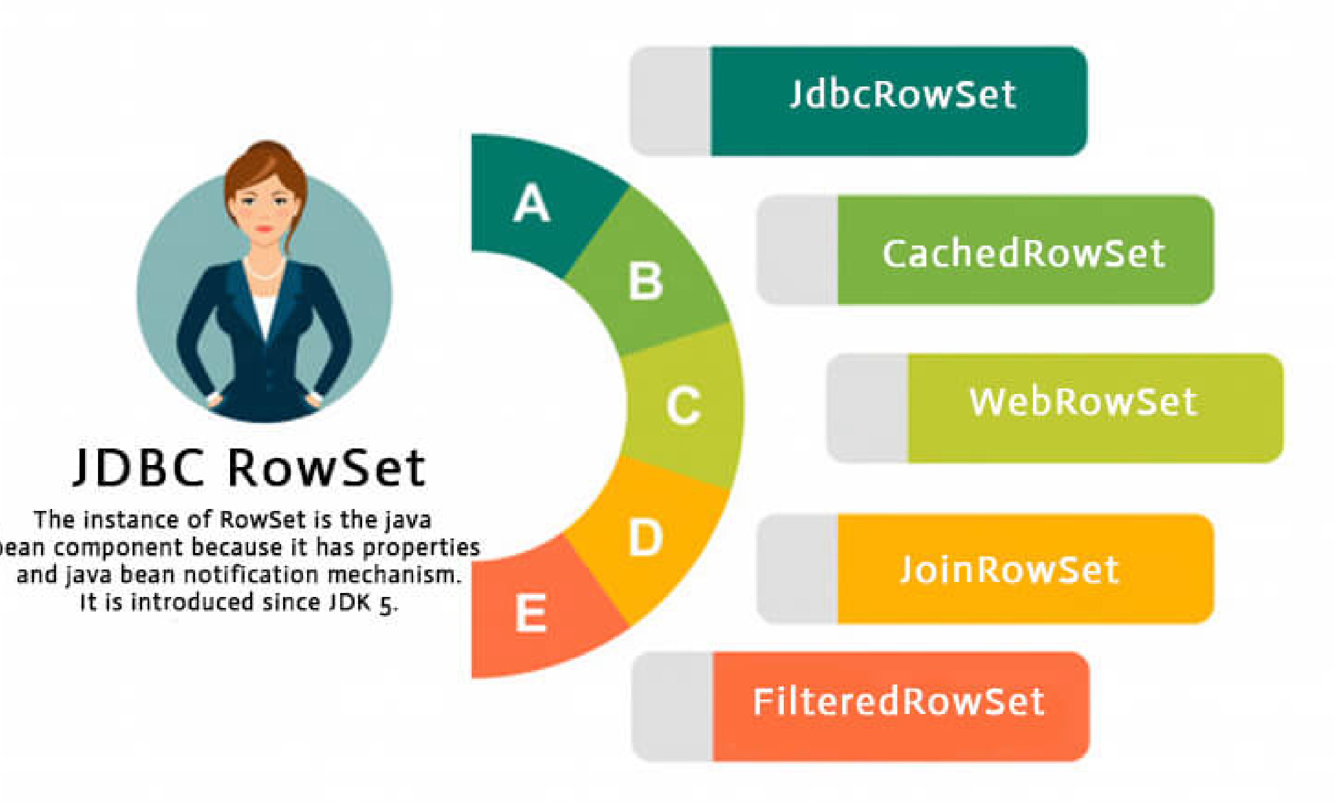
JDBC RowSet

An instance of **RowSet** is the Java bean component because it has properties and Java bean notification mechanism. It is the wrapper of ResultSet. A JDBC RowSet facilitates a mechanism to keep the data in tabular form. It happens to make the data more flexible as well as easier as compared to a ResultSet. The connection between the data source and the RowSet object is maintained throughout its life cycle. The RowSet supports development models that are component-based such as JavaBeans, with the standard set of properties and the mechanism of event notification.

It was in the JDBC 2.0, the support for the RowSet was introduced using the optional packages. But the implementations were standardized for RowSet in the JDBC RowSet Implementations Specification (JSR-114) by the Sun Microsystems that is being present in the JDK (Java Development Kit) 5.0.

The implementation classes of the RowSet interface are as follows:

- JdbcRowSet
- CachedRowSet
- WebRowSet
- JoinRowSet
- FilteredRowSet


JDBC RowSet
The instance of RowSet is the java bean component because it has properties and java bean notification mechanism. It is introduced since JDK 5.
A
B
C
D
E
JdbcRowSet
CachedRowSet
WebRowSet
JoinRowSet
FilteredRowSet

Let's see how to create and execute RowSet.

1. JdbcRowSet rowSet = RowSetProvider.newFactory().createJdbcRowSet();
2. rowSet.setUrl("jdbc:oracle:thin:@localhost:1521:xe");
3. rowSet.setUsername("system");

```
rowSet.setPassword("oracle");

rowSet.setCommand("select * from emp400");
rowSet.execute();
```

It is the new way to get the instance of JdbcRowSet since JDK 7.

Advantage of RowSet

The advantages of using RowSet are given below:

1. It is easy and flexible to use.
2. It is Scrollable and Updatable by default.

Example of JdbcRowSet

Let's see the simple example of JdbcRowSet without event handling code.

FileName: RowSetExample.java

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;
import javax.sql.RowSetEvent;
import javax.sql.RowSetListener;
import javax.sql.rowset.JdbcRowSet;
import javax.sql.rowset.RowSetProvider;

public class RowSetExample {
        public static void main(String[] args) throws Exception {
                Class.forName("oracle.jdbc.driver.OracleDriver");

    //Creating and Executing RowSet
        JdbcRowSet rowSet = RowSetProvider.newFactory().createJdbcRowSet();
        rowSet.setUrl("jdbc:oracle:thin:@localhost:1521:xe");
        rowSet.setUsername("system");
        rowSet.setPassword("oracle");

        rowSet.setCommand("select * from emp400");
        rowSet.execute();

    while (rowSet.next()) {
                        // Generating cursor Moved event
                        System.out.println("Id: " + rowSet.getString(1));
                        System.out.println("Name: " + rowSet.getString(2));
                        System.out.println("Salary: " + rowSet.getString(3));
                }

        }
}
```

The output is given below:

```
Id: 55
Name: Om Bhim
```

```
Salary: 70000
Id: 190
Name: abhi
Salary: 40000
Id: 191
Name: umesh
Salary: 50000
```

Example of JDBC RowSet with Event Handling

To perform event handling with JdbcRowSet, you need to add the instance of **RowSetListener** in the addRowSetListener method of JdbcRowSet.

The RowSetListener interface provides 3 method that must be implemented. They are as follows:

1. public void cursorMoved(RowSetEvent event);
2. public void rowChanged(RowSetEvent event);
3. public void rowSetChanged(RowSetEvent event);

Let's write the code to retrieve the data and perform some additional tasks while the cursor is moved, the cursor is changed, or the rowset is changed. The event handling operation can't be performed using ResultSet, so it is preferred now.

FileName: RowSetExample.java

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;
import javax.sql.RowSetEvent;
import javax.sql.RowSetListener;
import javax.sql.rowset.JdbcRowSet;
import javax.sql.rowset.RowSetProvider;

public class RowSetExample {
        public static void main(String[] args) throws Exception {
                Class.forName("oracle.jdbc.driver.OracleDriver");

    //Creating and Executing RowSet
    JdbcRowSet rowSet = RowSetProvider.newFactory().createJdbcRowSet();
    rowSet.setUrl("jdbc:oracle:thin:@localhost:1521:xe");
    rowSet.setUsername("system");
    rowSet.setPassword("oracle");

        rowSet.setCommand("select * from emp400");
        rowSet.execute();

    //Adding Listener and moving RowSet
    rowSet.addRowSetListener(new MyListener());

                while (rowSet.next()) {
                        // Generating cursor Moved event
                        System.out.println("Id: " + rowSet.getString(1));
                        System.out.println("Name: " + rowSet.getString(2));
                        System.out.println("Salary: " + rowSet.getString(3));
                }

        }
}

class MyListener implements RowSetListener {
      public void cursorMoved(RowSetEvent event) {
                System.out.println("Cursor Moved...");
      }
     public void rowChanged(RowSetEvent event) {
                System.out.println("Cursor Changed...");
     }
     public void rowSetChanged(RowSetEvent event) {
                System.out.println("RowSet changed...");
     }
}
```

The output is as follows:

```
Cursor Moved...
Id: 55
Name: Om Bhim
Salary: 70000
Cursor Moved...
Id: 190
Name: abhi
Salary: 40000
Cursor Moved...
Id: 191
Name: umesh
Salary: 50000
Cursor Moved...
```