Module 5: J2EE and JDBC(database access)

1.Java Database Connectivity (JDBC)

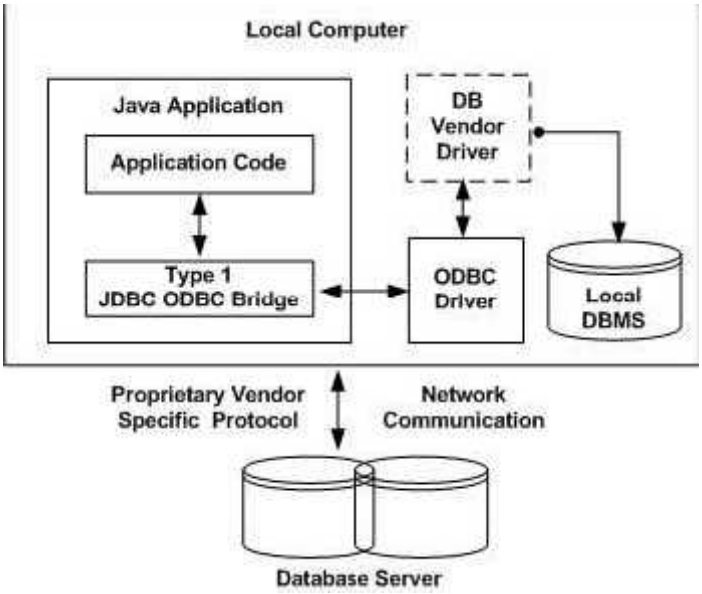
- Java Database Connectivity (JDBC) is an implementation of the Java programming language that dictates how databases communicate with each other.
- Through a standardized application programming interface (API), connectivity from database management systems (DBMS) to a wide range of SQL databases is accomplished.
- By deploying database drivers laced with JDBC technology, it is possible to connect to any database -- even in a heterogeneous environment -- and access tables, tabular data, flat files and more.
- When using JDBC, Java programmers have the ability to request connections to a database, send queries to the database using SQL statements, and receive results for advanced processing.

2.JDBC Drivers

- To connect with individual databases, JDBC requires drivers for each database.
- The various driver types are described in the following sections:
    - Type I: JDBC-ODBC Bridge
      Type II: Native API/JAVA
    - Type III: Pure Java driver for database middleware(JDBC protocol)
    - Type Four Driver : Direct-to-database pure Java driver (JAVA protocol)

Type 1: JDBC-ODBC Bridge Driver

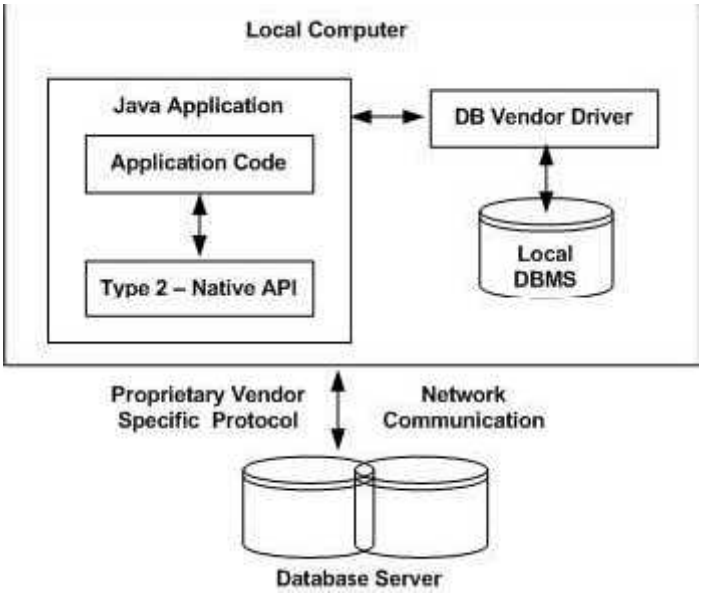
- In a Type 1 driver, a JDBC bridge is used to access ODBC drivers installed on each client machine. Using ODBC, requires configuring on your system a Data Source Name (DSN) that represents the target database.
- When Java first came out, this was a useful driver because most databases only supported ODBC access but now this type of driver is recommended only for experimental use or when no other alternative is available.



- A JDBC/ODBC bridge provides JDBC API access through one or more ODBC drivers. Some ODBC native code and in many cases native database client code must be loaded on each client machine that uses this type of driver.
- **The advantage** for using this type of driver is that it allows access to almost any database since the database ODBC drivers are readily available.
- Disadvantages for using this type of driver include the following:
    - Performance is degraded since the JDBC call goes through the bridge to the ODBC driver then to the native database connectivity interface. The results are then sent back through the reverse process
    - Limited Java feature set
    - May not be suitable for a large-scale application

Type Two Driver:Native API/JAVA protocal

- In a Type 2 driver, JDBC API calls are converted into native C/C++ API calls, which are unique to the database. These drivers are 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.

Advantages for using this type of driver include the following:

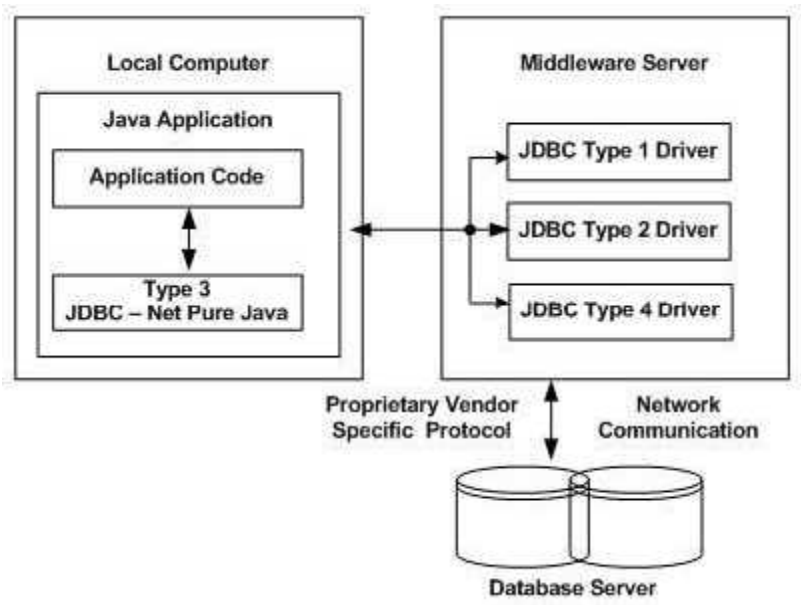
- Allows access to almost any database since the databases ODBC drivers are readily available
- Offers significantly better performance than the JDBC/ODBC Bridge
- Limited Java feature set

Disadvantages for using this type of driver include the following:

- Applicable Client library must be installed
- Type 2 driver shows lower performance than type 3 or 4

Type 3: JDBC-Net pure Java(JDBC PROTOCAL)

- In a Type 3 driver, a three-tier approach is used to access databases. The JDBC clients use standard network sockets to communicate with a 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.



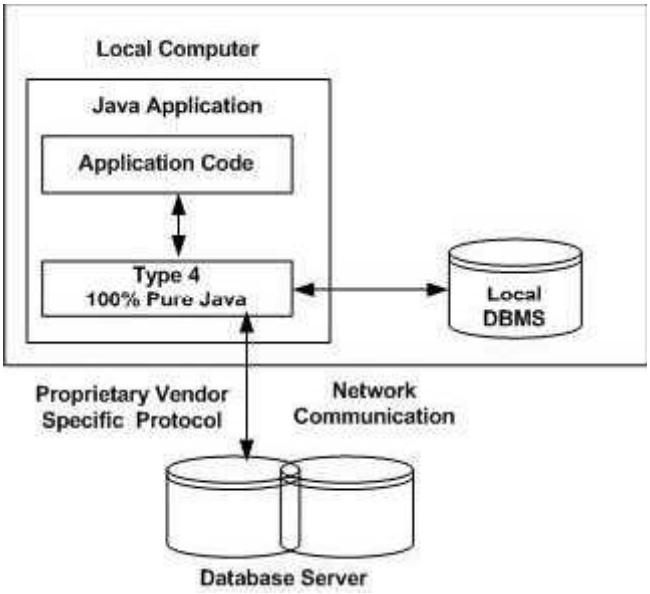
Advantages for using this type of driver include the following:

- Allows access to almost any database since the databases ODBC drivers are readily available
- Offers significantly better performance than the JDBC/ODBC Bridge and Type 2 Drivers
- Advanced Java feature set
- Scalable
- Caching
- Advanced system administration
- Does not require applicable database client libraries

The disadvantage for using this type of driver is that it requires a separate JDBC middleware server to translate specific native-connectivity interface.

Type 4: Pure Java protocal

- In a Type 4 driver, a pure Java-based driver communicates directly with the 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.



Advantages for using this type of driver include the following:

- Allows access to almost any database since the databases ODBC drivers are readily available
- Offers significantly better performance than the JDBC/ODBC Bridge and Type 2 Drivers
- Scalable
- Caching
- Advanced system administration
- Superior performance
- Advance Java feature set
- Does not require applicable database client libraries

The disadvantage for using this type of driver is that each database will require a driver

3.A brief overview of the JDBC process:

This process is divided into five steps:

- Loading the jdbc drivers
- Connecting to dbms
- Creating and executing statements
- Processing data returned by dbms
- Terminating the connection with the dbms

Loading the JDBC drivers:

- The jdbc must be loaded before the j2ee components can connect to the dbms.
- The Class.forName() method is used to load the jdbc driver and passing it the name of driver as an arguments to the method.
- **code snippet is shown below:**

```
try
{
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
}
catch(Exception e)
{
S.o.p(e);
}
```

Connect to the dbms:

- Once driver is loaded, the j2ee components must connect to the dbms using the static method getConnection().
- Where getConnection() methods belong to class called as DriverManager.
- getConnection() method passed the URL as argument of database and username ,password if necessary to database.Where URL is the string object that contains the driver name and databse name that is being accessed by the j2ee components.
- DriverManager.getConnection() methods returns a connection interface that is used throughout the process to reference the database.
- Code snippet is shown below:

```
try
{
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection c=DriverManager.getConnection("JDBC:ODBC:CSB);
}
catch(Exception e)
{
S.o.p(e);     }
```

Create and execute sql statements:

- After jdbc driver is loaded and connection is successfully made with the databse.
- Now database is managed by the dbms is to send a sql query to the dbms for processing.
- The createStatement() method is used to create the statement object. The createStatement() method is belongs to connection interface.
- The return value of createStatement() method is the Statement interface.
- The statement object is used to execute queryand return a resultset interface objectthat conatains the response from the dbms.
- The different methods are used to execute the query are as follows:
    - executeQuery(String)
    - ExecuteUpadate(String)
    - Execute(string)

Code snippet :

```
try
{
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection c=DriverManager.getConnection("JDBC:ODBC:CSB");
Statement s=c.createStatement();
ResultSet r=s.executeQuery("Select *from emp");
}
catch(Exception e)
{
S.o.p(e);
}
```

Process data returned by the dbms:

- ResultSet object is assigned to receive the data from the DBMS after the query processed.
- ResultSet object conatins the method used to intract with the data that is returned by DBMS to the j2ee components.
- Next() method is used to process the data from the DBMS.it is pointing to the first row of table. Next() method is always used in iterative process.

- getString() methods of ResultSet object is used to copy the value of specified columns in the current row of the ResultSet to a string object.
- The getString() methods is passed the name of the column or column index in the ResultSet whose content need to be copied.

Code snippet:

```
try
{
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection c=DriverManager.getConnection("JDBC:ODBC:CSB");
Statement s=c.createStatement();
ResultSet r=s.executeQuery("Select *from emp");
while(r.next())
{
String name=r.getString(1);
System.out.println("name="+name);
}
catch(Exception e)
{
S.o.p(e);
}
```

Terminating the connection to the DBMS:

The connection to the dbms is terminated by the close() method of the connection interface once the j2ee component is finished accessing the dbms.

```
c.close();
```

program to retrieve the data from the database:

```
import java.sql.*;
class A
{
A()
{
try
{
Class.forName(“sun.jdbc.odbc.JdbcOdbcDriver”);
Connection c=DriverManager.getConnection(“JDBC:ODBC:CSB”);
Statement s=c.createStatement();
ResultSet r=s.executeQuery(“Select *from emp”);
while(r.next())
{
String name=r.getString(1);
String usn=r.getString(2);
System.out.println(“name=”+name);
System.out.println(“USN=”+usn);
}
c.close();
}
catch(Exception e)
{
S.o.p(e);
}
}
public stataic void main(String ar[])
{
A a1=new A();
}
}
```

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
```

5.Database connection:

- Connection can be established using the **DriverManager.getConnection()** method.
- The data source that the jdbc components will connect to is defined using the url format. The url consist of three parts.
    - JDBC-Which indicates that the jdbc protocol is to be used to read the url
    - <subprotocal>-which indicates the jdbc driver name
    - <subname>- which indicates the name of the database.
- the three overloaded DriverManager.getConnection() methods −
    - getConnection(String url)
    - getConnection(String url, Properties prop)
    - getConnection(String url, String user, String password)

Using Only a Database URL

DriverManager.getConnection( ) method requires only a database URL −

getConnection(String url)

```
try
{
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection c=DriverManager.getConnection("JDBC:ODBC:CSB");
}
catch(Exception e)
{
Sysetm.out.println(e);
}
```

Using a Database URL with a username and password

- The most commonly used form of getConnection() requires you to pass a database URL, a *username*, and a *password*:
- Now you have to call getConnection() method with appropriate username and password to get a **Connection** object as follows −
    - getConnection(String url, String user, String password)

```
try
{
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection c=DriverManager.getConnection("JDBC:ODBC:CSB","CSB","Tiger");
}
catch(Exception e)
{
Sysetm.out.println(e);
}
```

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 is used to pass driver properties to the driver during a call to the getConnection() method.

```
try
{
Properties p = new Properties( );
FileInputStream f=new FileInputStream("p1.txt");
p.load(f);
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection c=DriverManager.getConnection("JDBC:ODBC:CSB", p);
}
catch(Exception e)
{
Sysetm.out.println(e);
}
```

NOTE:just for reference-Following table lists down the 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

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

5.The Statement Objects

- Once connection to the databse is opened,the j2ee component creates and sends a query to access data contained in database.
- There are three ways statement object are used:
    - Statement object
    - preparedStatemnt object
    - callableStatement object

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 s=c.createStatement();

  Once you've created a Statement object, you can then use it to execute an 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 number 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.

Program:

```
import java.sql.*;
class A
{
A()
{
try
{
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection c=DriverManager.getConnection("JDBC:ODBC:CSB");
Statement s=c.createStatement();
ResultSet r=s.executeQuery("Select *from emp");
while(r.next())
{
String name=r.getString(1);
String usn=r.getString(2);
System.out.println("name="+name);
System.out.println("USN="+usn);
}
c.close();
}
catch(Exception e)
{
S.o.p(e);
}
```

```
}
Public stataic void main(String ar[])
{
A a1=new A();
}
}
```

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.

  PreparedStatement p=new PreparedStatement("select name from emp where usn=?");

- 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.
  - **setXXX(int,string);**
- First parameter represent the column index and second parameter represent the values that replace the ? mark in the query.
- Next different execut methods of the preparedStatement object are called.

```
import java.sql.*;
class A
{
A()
{
try
{
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection c=DriverManager.getConnection("JDBC:ODBC:CSB");
```

PreparedStatement p=c.PreaparedStatement("select name from emp where usn=?");

p.setSting(2, "12cs001");

ResultSet r=p.executeQuery();

```
while(r.next())
{
String name=r.getString(1);
String usn=r.getString(2);
System.out.println("name="+name);
System.out.println("USN="+usn);
}
c.close();
}
catch(Exception e)
{
S.o.p(e);
}
}
public static void main(String ar[])
{
A a1=new A();
}
}
```

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.
- Three types of parameters exist: IN, OUT, and INOUT. The PreparedStatement object only uses the IN parameter. The CallableStatement object can use all the 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 −
- 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 that 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.

Program:

```
import java.sql.*;
class A
{
A()
{
try
{
Class.forName(“sun.jdbc.odbc.JdbcOdbcDriver”);
Connection c=DriverManager.getConnection(“JDBC:ODBC:CSB”);
CallableStatement p=c.CallableSatement(“Call lastOrderNumber(?)”);
p.registerOutParameter(1,TYPES.VARCHAR);
p.executeQuery();
```

```
String name=p.getString(1);
System.out.println("name="+name);
}
c.close();
}
catch(Exception e)
{
S.o.p(e);
}
}
public stataic void main(String ar[])
{
A a1=new A();
}
}
```

6. ResultSet

A `ResultSet` consists of records. Each records contains a set of columns.

A `ResultSet` can be of a certain type. The type determines some characteristics and abilities of the `ResultSet`.

Scrollable ResultSet:

At the time of writing there are three `ResultSet` types:

1. ResultSet.TYPE_FORWARD_ONLY
2. ResultSet.TYPE_SCROLL_INSENSITIVE
3. ResultSet.TYPE_SCROLL_SENSITIVE

The default type is `TYPE_FORWARD_ONLY`

- `TYPE_FORWARD_ONLY` means that the `ResultSet` can only be navigated forward. That is, you can only move from row 1, to row 2, to row 3 etc. You cannot move backwards in the `ResultSet`.
- `TYPE_SCROLL_INSENSITIVE` means that the `ResultSet` can be navigated (scrolled) both forward and backwards. You can also jump to a position relative to the current position,

or jump to an absolute position. The `ResultSet` is insensitive to changes in the underlying data source while the `ResultSet` is open. That is, if a record in the `ResultSet` is changed in the database by another thread or process, it will not be reflected in already opened `ResulsSet`'s of this type.

- `TYPE_SCROLL_SENSITIVE` means that the `ResultSet` can be navigated (scrolled) both forward and backwards. You can also jump to a position relative to the current position, or jump to an absolute position. The `ResultSet` is sensitive to changes in the underlying data source while the `ResultSet` is open. That is, if a record in the `ResultSet` is changed in the database by another thread or process, it will be reflected in already opened `ResulsSet`'s of this type.

Method	Description
`absolute()`	Moves the `ResultSet` to point at an absolute position. The position is a row number passed as parameter to the `absolute()` method.
`afterLast()`	Moves the `ResultSet` to point after the last row in the `ResultSet`.
`beforeFirst()`	Moves the `ResultSet` to point before the first row in the `ResultSet`.
`first()`	Moves the `ResultSet` to point at the first row in the `ResultSet`.
`last()`	Moves the `ResultSet` to point at the last row in the `ResultSet`.
`next()`	Moves the `ResultSet` to point at the next row in the `ResultSet`.
`previous()`	Moves the `ResultSet` to point at the previous row in the `ResultSet`.
`relative()`	Moves the `ResultSet` to point to a position relative to its current position. The relative position is passed as a parameter to the relative method, and can be both positive and negative.
	Moves the `ResultSet`

PROGRAM:

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

```
class A
{
A()
{
try
{
Class.forName(“sun.jdbc.odbc.JdbcOdbcDriver”);
Connection c=DriverManager.getConnection(“JDBC:ODBC:CSB”);
Statement s=c.createStatement(ResultSet.TPYE_SCROLL_SENSITIVE);
ResultSet r=s.executeQuery(“Select *from emp”);
While(r.next())
{
String name=r.getString(1);
String usn=r.getString(2);
System.out.println(“name=”+name);
System.out.println(“USN=”+usn);
}
r.first();
System.out.println(r.getString(1));
r.last();
System.out.println(r.getString(1));
r.previous();
System.out.println(r.getString(1));
r.absolute(2);
System.out.println(r.getString(1));
r.relative(2);
System.out.println(r.getString(1));
r.relative(-2);
System.out.println(r.getString(1));
c.close();
}
```

```
catch(Exception e)
{
S.o.p(e);
}
}
public static void main(String ar[])
{
A a1=new A();
}
}
```

Updatable ResultSet :

- The `ResultSet` concurrency determines whether the `ResultSet` can be updated, or only read.
- A `ResultSet` can have one of two concurrency levels:
    1. ResultSet.CONCUR_READ_ONLY
    2. ResultSet.CONCUR_UPDATABLE
- CONCUR_READ_ONLY means that the `ResultSet` can only be read.
- CONCUR_UPDATABLE means that the `ResultSet` can be both read and updated.
- If a `ResultSet` is updatable, you can update the columns of each row in the `ResultSet`. You do so using the many `updateXXX()` methods.
- `updateRow()` is called that the database is updated with the values of the row

```
import java.sql.*;
class A
{
A()
{
try
{
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection c=DriverManager.getConnection("JDBC:ODBC:CSB");
```

```
Statement s=c.createStatement(ResultSet.CONCUR_UPDATABLE);
ResultSet r=s.executeQuery("Select *from emp where usn=2");
r.update(1, "Avinash");
r.updateRow();
while(r.next())
{
String name=r.getString(1);
String usn=r.getString(2);
System.out.println("name="+name);
System.out.println("USN="+usn);
}
c.close();
}
catch(Exception e)
{
S.o.p(e);
}
}
public stataic void main(String ar[])
{
A a1=new A();
}
}
```

Inserting Rows into a ResultSet

If the `ResultSet` is updatable it is also possible to insert rows into it. You do so by:

1. update row column values using updateXX(string,string);
2. call ResultSet.insertRow()

```
import java.sql.*;
class A
{
A()
```

```
{
try
{
Class.forName(“sun.jdbc.odbc.JdbcOdbcDriver”);
Connection c=DriverManager.getConnection(“JDBC:ODBC:CSB”);
Statement s=c.createStatement(ResultSet.CONCUR_UPDATABLE);
ResultSet r=s.executeQuery(“Select *from emp ”);
r.update(1, ”Avinash”);
r.insertRow();
while(r.next())
{
String name=r.getString(1);
String usn=r.getString(2);
System.out.println(“name=”+name);
System.out.println(“USN=”+usn);
}
c.close();
}
catch(Exception e)
{
S.o.p(e);
}
}
public stataic void main(String ar[])
{
A a1=new A();
}
}
```

Deleteing row from a ResultSet:

- Deleterow() method is nused to delete the row from the databse.
- DeleteRow() method pass as an integer argument ,which specify the row to be deleted.

ResultSet.deleteRow(int);

Program:

```
import java.sql.*;
class A
{
A()
{
Try
{
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection c=DriverManager.getConnection("JDBC:ODBC:CSB");
Statement s=c.createStatement(ResultSet.CONCUR_UPDATABLE);
ResultSet r=s.executeQuery("Select *from emp ");
r.deleteRow(0);
while(r.next())
{
String name=r.getString(1);
String usn=r.getString(2);
System.out.println("name="+name);
System.out.println("USN="+usn);
}
c.close();
}
catch(Exception e)
{
S.o.p(e);
}
}
```

```
public static void main(String ar[])
{
A a1=new A();
}
}
```

7.Transactions

- A transaction is a set of actions to be carried out as a single, atomic action. Either all of the actions are carried out, or none of them are.
- Transaction is successfully completed only if each task is comleted successfully. If one of task is fail, the entire transaction is fail.
- If one of sql is failed, the sql statement that is executed successfully upto the point in the transaction must be rollback.
- Different methods of Transaction processing are:
    - **setAutoCommit(boolean)-**setAutoCommit() pass the parameter as false intial once all the transcation is completed. As soon as it invokes the commit() ,the setAutoCommit() method is set as true.
    - **setSavePoint(String);-**set the save point to the sql statement .
    - **releaseSavePoint(String);-**it realse the save point assing to the sql statement if and only if all sql statement are executed successfully.
    - **commit();-**once all sql statement are executed successfully,rollback is not possible.
    - **rollback();-**if one of the sql statement is failed,then rollback() method is invoked and control goes back to the fail sql statement for further execution.

Program:

```
import java.sql.*;
class A
{
A()
{
try
{
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

```
Connection c=DriverManager.getConnection("JDBC:ODBC:CSB");
Statement s=c.createStatement();
c.setAutoCommit(false);
c.setSavePoint("csb");
ResultSet r=s.executeQuery("Select *from emp where usn=2");
        r=s.executeQuery("Select *from emp");
c.releaseSavePoint("csb");
c.commit();
}
c.close();
}
catch(Exception e)
{
S.o.p(e);
c.rollback();
}
}
public static void main(String ar[])
{
A a1=new A();
}
}
```

8.Metadata:

Metadata is data about data. J2ee component can access metadata by using

- DatabaseMetaData interface.
- ResultSetMetaData interface

DatabaseMetaData interface:

- The DatabaseMetaData interface is used to retrieve information about database,table,columns and index amoung other information about dbms.

- J2ee component retrives metadata about the database by calling getMetaData() method of the connection interface object. The getMetaData() method return a DatabaseMetaData object that contain information of database and components.
- Most commonly used DatabaseMetaData interface methods as follows:
    - **getDataBaseProductName()-** returns the product name of the database.
    - **getUserName()-**returns the username of database
    - **getURL()-** returns the URL of the database
    - **getSchemas()-**returns the all schemas of the database which are available
    - **getPrimaryKeys()-**returns the primary key available in the database
    - **getTables()-**returns the table name in the database

program:

```
import java.sql.*;
class A
{
A()
{
try
{
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection c=DriverManager.getConnection("JDBC:ODBC:CSB");
Statement s=c.createStatement();
ResultSet  r=s.executeQuery("Select *from emp");
DatabaseMetaData d=c.getMetaData();
System.out.println(d.getUserNAme());
System.out.println(d.getTables());
System.out.println(d.getURL());
}
c.close();
}
catch(Exception e)
{
```

```
S.o.p(e);
}
}
public static void main(String ar[])
{
A a1=new A();
}
}
```

ResultSetMetaData interface:

- ResultSetMetaData interface is used to retrieve the information by calling the getMetaData() method of ResultSet interface.
- Different methods in the ResultSetMetaData inetface are as follows:
    - **getColunmCount()-**returns the number of column available in the table
    - **getColunmName(int)-**returns the name of column specified by the column number
    - **getColunmTye(int**)-returns the type of column specified by the column number

program:

```
import java.sql.*;
class A
{
A()
{
try
{
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection c=DriverManager.getConnection("JDBC:ODBC:CSB");
Statement s=c.createStatement();
ResultSet  r=s.executeQuery("Select *from emp");
ResultSetMetaData d=r.getMetaData();
System.out.println(d.getColunmName(1));
```

```
System.out.println(d.getColunmCount());
System.out.println(d.getColunmType(1));
}
c.close();
}
catch(Exception e)
{
S.o.p(e);
}
}
public static void main(String ar[])
{
A a1=new A();
}
}
```

Data Types:

The JDBC driver converts the Java data type to the appropriate JDBC type, before sending it to the database. It uses a default mapping for most data types. The following table summarizes the default JDBC data type that the Java data type is converted to, when you call the setXXX() method.

SQL	JDBC/Java
VARCHAR	String
CHAR	String
LONGVARCHAR	String
BIT	boolean
NUMERIC	java.math.BigDecimal
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
FLOAT	float
DOUBLE	double

VARBINARY	byte[ ]
BINARY	byte[ ]
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp
CLOB	java.sql.Clob
BLOB	java.sql.Blob

Exception :

SQLException Methods

An SQLException can occur both in the driver and the database. When such an exception occurs, an object of type SQLException will be passed to the catch clause.

The passed SQLException object has the following methods available for retrieving additional information about the exception −

Method	Description
getErrorCode( )	Gets the error number associated with the exception.
getMessage( )	Gets the JDBC driver's error message for an error, handled by the driver or gets the Oracle error number and message for a database error.
getSQLState( )	Gets the XOPEN SQLstate string. For a JDBC driver error, no useful information is returned from this method. For a database error, the five-digit XOPEN SQLstate code is returned. This method can return null.
getNextException( )	Gets the next Exception object in the exception chain.
printStackTrace( )	Prints the current exception, or throwable, and it's backtrace to a standard error stream.
printStackTrace(PrintStream s)	Prints this throwable and its backtrace to the print stream you specify.
printStackTrace(PrintWriter w)	Prints this throwable and it's backtrace to the print writer you specify.