**Java Database Programming**

**Data Concepts**

**Text :**

Meaningless information. It will be maintained by the text file. Notepad is used to create the text file.

Note pad will not maintain any dictionary.

Eg:

Text files.

**Data :**

It is a meaningful information. It will be maintained by word document file. Word pad is used to create the create the word document file. Word pad will maintain the dictionary.

Eg:

MS-Word document File.

**3 V's of Data**

Volume, Velocity, Variety of data.

**Volume**:

How much data, Bytes, KB, MB, GB ( RDBMS Can ) , TB, PB, EB

1 Byte byte = 8 bits

Kilobyte 1 KB = 1024 bytes

Megabyte 1 MB = 1024 KB

Gigabyte 1 GB = 1024 MB

Terabyte 1 TB = 1024 GB

Petabyte 1 PB = 1024 TB

Exabyte 1 EB = 1024 PB

Bigdata --> terabytes to Zettabytes.

**Velocity**:

How fast data is processed, Batch and Streaming Data.

**Variety**:

The various types of data, Structured data, Semi-Structured data and Un-Structured data

The data in it, will be of three types.

**Unstructured data** :

It is a data, which is not organized properly.

Eg:

Word, PDF, Text, Media Logs.

**Structured data :**

Data is organised in semantic chunks (entities)

Similar entities are grouped together (relations or classes)

Entities in the same group have the same descriptions (attributes)

Descriptions for all entities in a group (schema)

Have the same defined format

Have a predefined length are all present and follow the same order

Eg:

Relational data.

**Semi Structured data :**

It is a data between structured and unstructured.

Eg:

XML data

It is, XML but not HTML

**Data storage :**

There are two types of storage

1. Temporary storage
2. Permanent storage

**Temporary storage:**

It is a data, which will be lost, once Application is shutdown ( stopped). It is maintained inside the ram memory.

Eg:

String and StringBuffer

Collection Objects ( java.util package )

String and StringBuffer are used to store maintain unstructured small amount of data.

Collection Objects are used to store and maintain huge amount of structure , unstructed and semi structured data.

**Permanent storage**

It is a Backend for the application, where storing and maintaining data permanently .

Eg:

Files and Database Servers

**Files:**

Traditional file system and HDFS ( Hadoop Distributed File System ),are used to store the data.

|  |  |
| --- | --- |
| **Traditional file system**  Windows and Linux file system | **Hadoop Distributed File System**  **( HDFS )** |
| Tradition files system is capable of storing less amount of data in Giga Bytes only. | HDFS is capable of storing huge amount of data upto Zeta Bytes only. |
| Size of each block is only 4kb | Size of each block is only 128 MB |
| A file can be stored only on single system. | File splitting into multiple blocks and stored on multiple systems in a network clusture. Called as distributed data storge. |
| Used to maintain less amount of data | Used to maintain huge amount of data |
| Wnodws file system is NTFS and Linux file system is Ext4 | HDFS is a distributed file system. |

**Database Concepts**

**Database :**

It is a structured data. It will be maintained in the form of records. Database will be maintained in the form of files.

Eg: Excel files,

**DBMS ( Data Base Management System ) :**

It is collection of Data and Commands to update the data.

Data will be maintained in the files. This data is to be updated to meet the client ( eg : company, bank, railways ) requirements, Some Mechanism is required to update the data. Such mechanism is called as database management system. Some commands will be maintained by the DBMS to update the data.

Eg :

INSERT, DELETE, UPDATE, SELECT

In the DBMS, data will be maintained in the form of files.

Eg:

MS FOX-PRO, MS-ACCESS

MS FOX-PRO is standalone database. It can not be shared by multiple clients and it fails to maintain the huge amount of data base.

**Disadvantages of using files:**

1. No security for the files. There may be a chance of deleting files accidentally. Once file has been deleted, loss of entire data.

2) A file cannot be shared by the multiple users at a time through Network.

3) Data redundancy: It is duplication of data.

**Data Normalization cannot be achieved with files:**

4) Relationship cannot be established between Master Record and Detailed record.

5) There is no order of storing data such as ascending / descending order

6) Constraints cannot be established on the data.

7) Automatic generations of Id not possible using files

**RDBMS ( Relational Data Base Management System ) :**

With RDBMS we can overcome the disadvantages of DBMS packages.

Mathematician E.F CODD has formed **TWELVE** rules to overcome the all disadvantages with the DBMS. Any data base which satisfies at least **SIX** rules out of **TWELVE** rules formed by the mathematician CODD is called as RDBMS.

Eg:

All Database servers are RDBMS

ORACLE , MS-SQL , MY-SQL, INFORMICS, DB2, SYBASE, postgres

**Comparison of different Database servers**

**ORACLE:**

It was developed by the Oracle Corporation.

It is a mostly used data base in the real time applications.

It was developed for all Operating Systems and Machines in the market

We can achieve 95 % of RDBMS with this data base.

O/S : WINDOWS, LINUX, UNIX, SOLARIS, IBM-AIX, HP-UX,

Machines : Intel Machines and Spark Machines.

**MS-SQL Server :**

It was developed by the Micro Soft.

It is a mostly used with Micro Soft front end products.

It was developed only for Window Operating Systems and Intel Machines.

We can not achieve complete RDBMS with this data base.

Dis-Adv 🡪There is no security for data base server as it is running on Windows OS.

**MYSQL :**

It was developed my Apache software foundation.

Later it was adopted by the Sun Soft.

It was developed for all Operating Systems and Machines in the market

It is in-built with SOLARIS Operating system.

We can not achieve complete RDBMS with this data base.

It fails to support too many users to share the data.

Recently it was taken over by Oracle Corporation.

**DB2 :**

It was developed by the IBM

It is a mostly used data base with IBM mainframe applications.

We can achieve 99 % of RDBMS with this data base.

This is called universal database.

O/S : WINDOWS, LINUX, UNIX, SOLARIS, IBM-AIX, HP-UX,

Machines : Intel Machines

**SYBASE :**

It was developed by the Sybase Corporation.

It was developed for all Operating Systems and Machines in the market

We cannot achieve complete RDBMS with this data base.

**INFORMICS :**

It was developed by Informics Software company.

It was developed only for windows Operating Systems and Intel Machines.

It is mostly used with C++ Applications.

We cannot achieve complete RDBMS with this data base.

Postgres:

It was developed by the Postgres Inc.

It was developed for all Operating Systems and Machines in the market.

It can be used with any front applications.

**Advantages of RDBMS**

**Primary Advantages :**

1. Data Security can be achieved
2. We can overcome Data redundancy
3. Data can be sharable
4. Data Normalization can be achieved

**Other Advantages:**

1) We can achieve the CODD’s Rules

2) Data representation is in the form of rows and columns.

3) Clients Operating system can be any one ( mostly clients are Windows O/S)

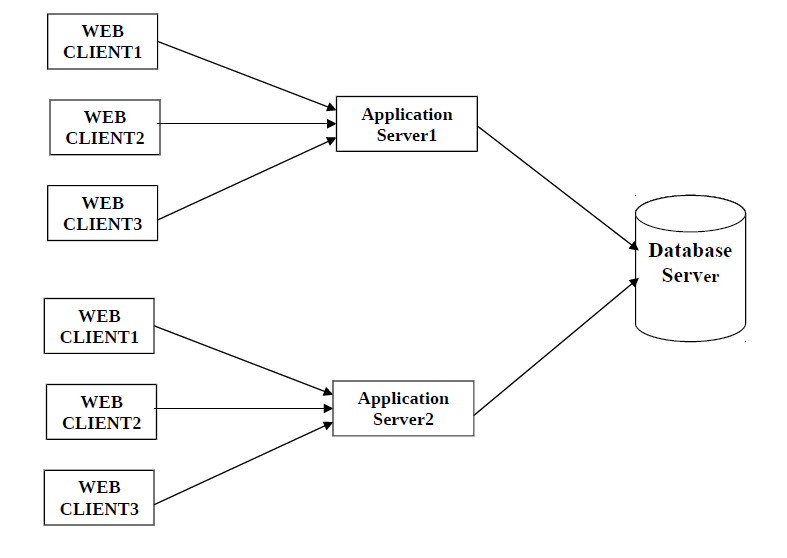
4) Data can stored in any format such Char, Number, Binary, Text

5) It provides support for ACID properties

6) Huge amount of data can be maintained

7) Data integrity can be achieved

## Databases on the Web



**Applications On the server :**

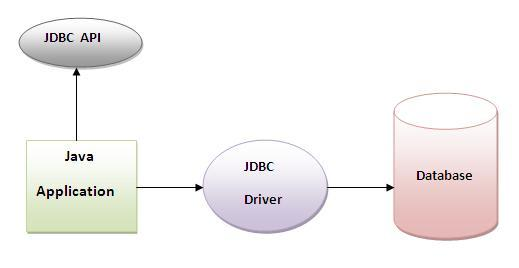
Every application will have its own drivers to connect to data base.

The common Applications are Java, **.**Net, ETL Tools, ERP Applications.

**Java Database Connectivity ( JDBC )**

Jdbc is a bridge between Java Application and Database.

Java JDBC is a java API to connect and execute query with the database. JDBC API uses jdbc drivers to connect with the database.

Why use JDBC

Before JDBC, ODBC API was the database API to connect and execute 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).

What is API

API (Application programming interface) is a document that contains 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:**

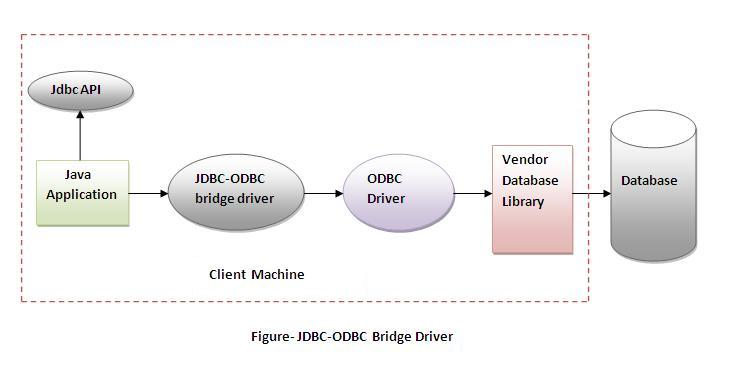
1) JDBC-ODBC bridge driver

2) Native-API driver

3) Network Protocol driver

4) Thin 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.

**Advantages:**

easy to use.

can be easily connected to any database.

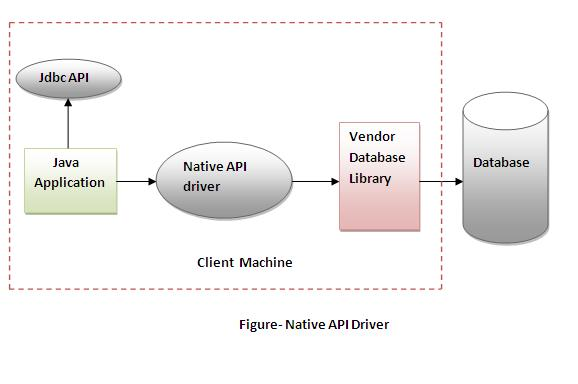
**Dis-Advantages:**

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.



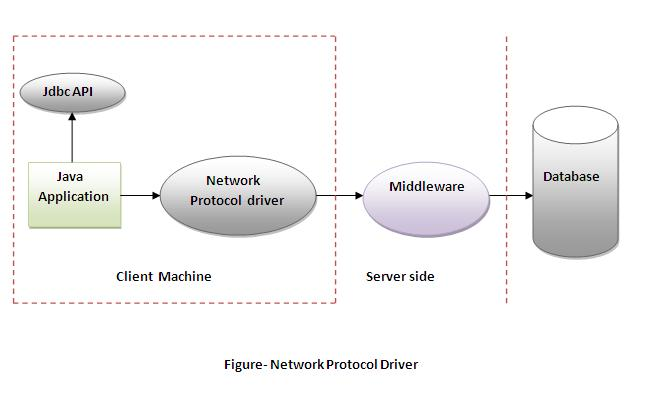
**Advantage:**

performance upgraded than JDBC-ODBC bridge driver.

**Dis-Advantage:**

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**

**Advantage:**

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

**Dis-Advantages:**

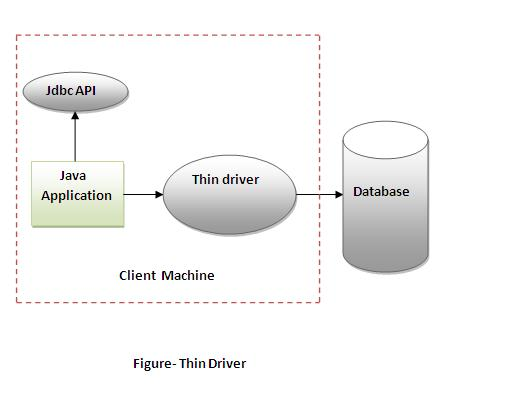
Network support is required on client machine

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.



**Advantage:**

Better performance than all other drivers.

No software is required at client side or server side.

**Dis-Advantage:**

Drivers depends on the Database.

**Steps in writing JDBC Progtamming :**

There are 7 steps to connect any java application with the database in java using JDBC.

They are as follows:

1) Register the driver class /(or) Load the driver class

2) Create connection to database

3) Creating statement

4) Prepare SQL query

5) Execute Query and get the ResultSet

6) Process ResultSet

7) Closing connection

**Example to Connect Java Application with Oracle database**

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

**package com.nrit.mnrao.test;**

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.sql.Statement;

import java.util.Date;

public class Test {

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

Connection con = null;

try

{

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

con = DriverManager.getConnection( "jdbc:oracle:thin:@localhost:1521:orcl",

"scott",

"tiger");

Statement st = con.createStatement();

String sqlQuery="select \* from emp";

ResultSet rs = st.executeQuery(sqlQuery);

while(rs.next())

{

System.out.println(

rs.getInt(1)+"\t"+

rs.getString(2)+"\t"+

rs.getString(3)+"\t"+

rs.getInt(4)+"\t"+

rs.getDate(5)+"\t"+

rs.getDouble(6)+"\t"+

rs.getDouble(7)+"\t"+

rs.getInt(8)

); }

con.close();

}

catch(ClassNotFoundException e)

{

e.printStackTrace();

}

catch(SQLException e)

{

e.printStackTrace();

}

finally

{

if(con!=null)

{

con.close();

}

}

}

}

**1) Register the driver class ( or ) load 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.

Driver class contains TCP/IP programming to connect to Database.

**Syntax of forName() method**

public static void forName(String className) throws ClassNotFoundException

Class.forName("oracle.jdbc.driver.OracleDriver"); --> Oracle.

Class.forName("com.mysql.jdbc.Driver"); --> Mysql.

**Database URL Formulation**

RDBMS JDBC driver name URL Format

MySQL com.mysql.jdbc.Driver jdbc:mysql://hostname:port/ databaseName

ORACLE oracle.jdbc.driver.OracleDriver

jdbc:oracle:thin:@hostname:portNumber:databaseName

DB2 COM.ibm.db2.jdbc.net.DB2Driver

jdbc:db2:hostname:portNumber/databaseName

Sybase com.sybase.jdbc.SybDriver

jdbc:sybase:Tds:hostname:portNumber/ databaseName

**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"); --> Oracle.

Connection con=DriverManager.getConnection(

"jdbc:mysql://localhost:3306/mydb","root","admin123"); --> Mysql.

URL including user name and password.

String URL = "jdbc:oracle:thin:username/password@localhost:1521:EMP";

Connection conn = DriverManager.getConnection(URL);

**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();**

**Types of statements**

These are of three types

1. Simple statement
2. Prepared Statement
3. Callable Statement

**4) Prepare SQL query.**

String sqlQuery = “SELECT \* FROM EMP”;

**5) 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(sqlQuery); ---> SELECT

int n = stmt.executeUpdate(sqlQuery); ---> INSERT, UPDATE and DELETE

int n = stmt.executeUpdate(); --> PreparedStatement

boolean b = stmt.execute(sqlQuery); --> CREATE

**6) Processing ResultSet:**

while(rs.next())

{

System.out.println (rs.getInt(1)+" "+rs.getString(2));

}

Database Data type and Java getter methods.

|  |  |
| --- | --- |
| SQL Data Types | Java getter() |
| Number | getInt() / getLong()/ getShort() |
| Character | getString() |
| Date | getDate() |
| Real number | getFloat()/ getDouble |
| Blob | getBlob() /getBinaryStream() |
| Clob | getCLob() / getCharacterStream() |
| Timestamp | getTimestamp() |
| BIT | getBoolean() |

**7) Close the connection object**

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

**Syntax of close() method**

public void close()throws SQLException

Example to close connection

con.close();

Example to connect to the mysql database

Mysql:

Step1:

prepare Database table data:

**connect to mysql server:**

$mysql -u root -p

Enter password:\*\*\*\*\*\*\*

mysql>show databases;

mysq> create database nrit;

mysql> show databases;

mysql> use nrit;

mysql> select database() from dual;

nrit

mysql > create table emp(empid int, name varchar(20), dept varchar(20), salary double(10,2));

mysql> show tables;

mysql> insert into emp values(1001,'venkat','dev',5000);

mysql> insert into emp values(1002,'pavan','test',6000);

mysql> insert into emp values(1003,'charan','admin',7000);

mysql> select \* from emp;

+-------+--------+-------+---------+

| empid | name | dept | salalry |

+-------+--------+-------+---------+

| 1001 | venkat | dev | 5000.00 |

| 1002 | pavan | test | 6000.00 |

| 1003 | charan | admin | 7000.00 |

+-------+--------+-------+---------+

Step2;

Develop JDBC application as below:

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

Down load “mysql-connector-java-5.1.18-bin.jar” and add to eclipse project ( add external jars )

package com.nrit.mnrao.test;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.sql.Statement;

public class MysqlFetchData {

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

Connection con = null;

try

{

Class.forName("com.mysql.jdbc.Driver");

con = DriverManager.getConnection(

"jdbc:mysql://localhost:3306/nrit",

"root",

"hadoop"

);

Statement st = con.createStatement();

String sqlQuery = "select \* from emp";

ResultSet rs = st.executeQuery(sqlQuery);

while(rs.next())

{

System.out.println(

rs.getInt(1)+"\t"+

rs.getString(2)+"\t"+

rs.getString(3)+"\t"+

rs.getDouble(4)

);

}

con.close();

}

catch (ClassNotFoundException e)

{

e.printStackTrace();

}

catch (SQLException e)

{

e.printStackTrace();

}

finally

{

if(con!=null)

{

con.close();

}

}

}

}

ResultSet always depends on SELECT query not on table.

Eg:

String sqlQuery = "select name, empid, salary, dept from emp";

ResultSet rs = st.executeQuery(sqlQuery);

while(rs.next())

{

System.out.println(

rs.getString(1)+"\t"+

rs.getInt(2)+"\t"+

rs.getDouble(3)+"\t"+

rs.getString(4)

);

}

**Program read all database names from MYSQL server.**

package com.nrit.mnrao.test;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.sql.Statement;

public class MysqlFetchData {

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

Connection con = null;

try

{

Class.forName("com.mysql.jdbc.Driver");

con = DriverManager.getConnection(

"jdbc:mysql://localhost:3306/",

"root",

"hadoop"

);

Statement st = con.createStatement();

String sqlQuery = "show databases";

ResultSet rs = st.executeQuery(sqlQuery);

while(rs.next())

{

System.out.println(rs.getString(1));

}

con.close();

}

catch (ClassNotFoundException e)

{

e.printStackTrace();

}

catch (SQLException e)

{

e.printStackTrace();

}

finally

{

if(con!=null)

{

con.close();

}

}

}

}

**program read all tables from MYSQL databse server.**

package com.nrit.mnrao.test;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.sql.Statement;

public class MysqlFetchData {

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

Connection con = null;

try

{

Class.forName("com.mysql.jdbc.Driver");

con = DriverManager.getConnection(

"jdbc:mysql://localhost:3306/empdb",

"root",

"hadoop"

);

Statement st = con.createStatement();

String sqlQuery = "show tables";

ResultSet rs = st.executeQuery(sqlQuery);

while(rs.next())

{

System.out.println(rs.getString(1));

}

con.close();

}

catch (ClassNotFoundException e)

{

e.printStackTrace();

}

catch (SQLException e)

{

e.printStackTrace();

}

finally

{

if(con!=null)

{

con.close();

}

}

}

}

**executeQuery():**

This is used to execute any SQL query,which is fetching data from database.

**Program to get current database name :**

package com.nrit.mnrao.test;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.sql.Statement;

public class MysqlFetchData {

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

Connection con = null;

try

{

Class.forName("com.mysql.jdbc.Driver");

con = DriverManager.getConnection(

"jdbc:mysql://localhost:3306/empdb",

"root",

"hadoop"

);

Statement st = con.createStatement();

String sqlQuery = "select database() from dual";

ResultSet rs = st.executeQuery(sqlQuery);

while(rs.next())

{

System.out.println(rs.getString(1));

}

con.close();

}

catch (ClassNotFoundException e)

{

e.printStackTrace();

}

catch (SQLException e)

{

e.printStackTrace();

}

finally

{

if(con!=null)

{

con.close();

}

}

}

}

List of setter() and getter() methods in JDBC for different data types of SQL



**Connection interface:**

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

By default, connection commits the changes after executing queries.

Commonly used methods of Connection interface:

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

2) public Statement createStatement(int resultSetType,int resultSetConcurrency):

Creates a Statement object that will generate ResultSet objects with the given type and concurrency.

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

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

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

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

**Statement interface**

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 insert, update, delete etc.

3) public boolean execute(String sql): is used to execute queries that may return multiple results, create and drop

4) public int[] executeBatch(): is used to execute batch of commands.

**Example of Statement interface**

**Program to insert records into data base table.**

package com.nrit.mnrao.test;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.SQLException;

import java.sql.Statement;

public class MysqlFetchData {

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

Connection con = null;

try

{

Class.forName("com.mysql.jdbc.Driver");

con = DriverManager.getConnection(

"jdbc:mysql://localhost:3306/nrit",

"root",

"hadoop"

);

Statement st = con.createStatement();

String sqlQuery = "INSERT INTO emp VALUES(1005,'nrit5','admin',5000)";

int count = st.executeUpdate(sqlQuery);

if(count>0)

{

System.out.println("Successfully inserted");

}

else

{

System.out.println("failed to insert");

}

con.close();

}

catch (ClassNotFoundException e)

{

e.printStackTrace();

}

catch (SQLException e)

{

e.printStackTrace();

}

finally

{

if(con!=null)

{

con.close();

}

}

}

}

**Program to update records into data base table.**

package com.nrit.mnrao.test;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.SQLException;

import java.sql.Statement;

public class MysqlFetchData {

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

Connection con = null;

try

{

Class.forName("com.mysql.jdbc.Driver");

con = DriverManager.getConnection(

"jdbc:mysql://localhost:3306/nrit",

"root",

"hadoop"

);

Statement st = con.createStatement();

String sqlQuery = "update emp set salary=salary+1000 where dept='dev'";

int count = st.executeUpdate(sqlQuery);

if(count>=0)

{

System.out.println(count+ " records updated");

}

con.close();

}

catch (ClassNotFoundException e)

{

e.printStackTrace();

}

catch (SQLException e)

{

e.printStackTrace();

}

finally

{

if(con!=null)

{

con.close();

}

}

}

}

**Program to delete reocrds;**

package com.nrit.mnrao.test;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.SQLException;

import java.sql.Statement;

public class MysqlFetchData {

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

Connection con = null;

try

{

Class.forName("com.mysql.jdbc.Driver");

con = DriverManager.getConnection(

"jdbc:mysql://localhost:3306/nrit",

"root",

"hadoop"

);

Statement st = con.createStatement();

String sqlQuery = "delete from emp where dept='dev'";

int count = st.executeUpdate(sqlQuery);

if(count>=0)

{

System.out.println(count+ " records deleted");

}

con.close();

}

catch (ClassNotFoundException e)

{

e.printStackTrace();

}

catch (SQLException e)

{

e.printStackTrace();

}

finally

{

if(con!=null)

{

con.close();

}

}

}

}

Connecting to database using externally configuration parameters/ externally configuration properties file

Prepare the properties file as below

“C:\project\DatabaseProperties.prop” and pass as command line parameter

dbProduct=oracle

dbDriver=oracle.jdbc.driver.OracleDriver

dbHost=localhost

dbPort=1521

dbUid=scott

dbPasswd=tiger

#oracle

dbSid=orcl

#mysql

dbName=nrit

package com.nrit.mnrao.test;

import java.io.FileNotFoundException;

import java.io.FileReader;

import java.io.IOException;

import java.io.PrintWriter;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.sql.Statement;

import java.util.Properties;

public class Test {

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

Connection con = null;

Properties dataBaseProperties = new Properties();

try

{

FileReader freader = new FileReader(args[0]);

dataBaseProperties.load(freader);

String dbProduct=dataBaseProperties.getProperty("dbProduct");

if(dbProduct==null || dbProduct.isEmpty())

{

System.out.println("missing dbProduct key / value " );

System.exit(0);

}

String dbDriver = dataBaseProperties.getProperty("dbDriver");

if(dbDriver==null ||dbDriver.isEmpty())

{

System.out.println("missing dbDriver key / value " );

System.exit(0);

}

String dbHost = dataBaseProperties.getProperty("dbHost");

if(dbHost==null ||dbHost.isEmpty())

{

System.out.println("missing dbHost key / value " );

System.exit(0);

}

String dbPort=dataBaseProperties.getProperty("dbPort");

if(dbPort==null ||dbPort.isEmpty())

{

System.out.println("missing dbPort key / value " );

System.exit(0);

}

String dbUid = dataBaseProperties.getProperty("dbUid");

if(dbUid==null ||dbUid.isEmpty())

{

System.out.println("missing dbUid key / value " );

System.exit(0);

}

String dbPasswd=dataBaseProperties.getProperty("dbPasswd");

if(dbPasswd==null ||dbPasswd.isEmpty())

{

System.out.println("missing dbPasswd key / value " );

System.exit(0);

}

String dbSid = dataBaseProperties.getProperty("dbSid");

if(dbSid==null ||dbSid.isEmpty())

{

System.out.println("missing dbSid key / value " );

System.exit(0);

}

String dbName= dataBaseProperties.getProperty("dbName");

if(dbName==null ||dbName.isEmpty())

{

System.out.println("missing dbName key / value " );

System.exit(0);

}

Class.forName(dbDriver);

String dbUrl=null;

if(dbProduct.equalsIgnoreCase("ORACLE"))

{

dbUrl="jdbc:oracle:thin:@"+dbHost+":"+dbPort+":"+dbSid;

}

else if(dbProduct.equalsIgnoreCase("MYSQL"))

{

dbUrl="jdbc:mysql://"+dbHost+":"+dbPort+"/"+dbName;

}

con = DriverManager.getConnection(dbUrl,dbUid,dbPasswd);

Statement st = con.createStatement();

String sqlQuery="select \* from emp";

ResultSet rs = st.executeQuery(sqlQuery);

while(rs.next())

{

System.out.println(

rs.getInt(1)+"\t"+

rs.getString(2)+"\t"+

rs.getString(3)+"\t"+

rs.getInt(4)+"\t"+

rs.getDate(5)+"\t"+

rs.getDouble(6)+"\t"+

rs.getDouble(7)+"\t"+

rs.getInt(8)

);

}

rs.close();

con.close();

}

catch(ClassNotFoundException e)

{

e.printStackTrace();

}

catch(IOException e)

{

e.printStackTrace();

}

finally

{

if(con!=null)

{

con.close();

}

}

}

}

**ResultSet interface:**

Scrollable ResultSet --> cursor can be moved forward and backward.

The object of Scrollable ResultSet is to maintains a cursor pointing to a particular row of data. Initially, cursor points to before the first row.

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

ResultSet.TYPE\_FORWARD\_ONLY.

But we can make this object to move forward and backward direction by passing either TYPE\_SCROLL\_INSENSITIVE or TYPE\_SCROLL\_SENSITIVE in createStatement(int,int) method as well as we can make this object as updatable by:

TYPE\_SCROLL\_INSENSITIVE

The constant indicating the type for a ResultSet object that is scrollable but generally not sensitive to changes to the data that underlies the ResultSet.

TYPE\_SCROLL\_SENSITIVE

The constant indicating the type for a ResultSet object that is scrollable and generally sensitive to changes to the data that underlies the ResultSet.

Statement stmt = con.createStatement(); 🡪 here resultset is forward only and insensitive

Statement stmt = con.createStatement(ResultSet.TYPE\_SCROLL\_SENSITIVE,

ResultSet.CONCUR\_UPDATABLE);

Sensitivness is with regards to the underlying data (a database):

Suppose you have EMP table in a database. You create insensitive statement:

Statement stmt = connection.createStatement(ResultSet.TYPE\_SCROLL\_INSENSITIVE,

ResultSet.CONCUR\_READ\_ONLY);

and at time 10:00 you issue a query

SELECT \* FROM EMP;

now you leave the result set open and you scroll through it using next(), previous() and absolute(int)) methods.

At 10:02 somebody updates data in EMP table.

At 10:03 you are still scrolling the result set but because you have INSENSITIVE result set, you see old data.

Now comes the difference. If you had created the statement with SENSITIVE then you would see all the changes that were being done at 10:02.

Commonly used methods of ResultSet interface

1) public boolean next(): is used to move the cursor to the one row next from the current position.

2) public boolean previous(): is used to move the cursor to the one row previous from the current position.

3) public boolean first(): is used to move the cursor to the first row in result set object.

4) public boolean last(): is used to move the cursor to the last row in result set object.

5) public boolean absolute(int row): is used to move the cursor to the specified row number in the ResultSet object.

6) public boolean relative(int row): is used to move the cursor to the relative row number in the ResultSet object, it may be positive ( forward) or negative (backward).

7) public int getInt(int columnIndex): is used to return the data of specified column index of the current row as int.

8) public int getInt(String columnName): is used to return the data of specified column name of the current row as int.

9) public String getString(int columnIndex): is used to return the data of specified column index of the current row as String.

10) public String getString(String columnName): is used to return the data of specified column name of the current row as String.

**Example of Scrollable ResultSet**

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

package com.nrit.mnrao.test;

import java.io.FileReader;

import java.io.IOException;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.sql.Statement;

import java.util.Date;

import java.util.Properties;

public class Test {

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

Connection con = null;

try

{

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

con = DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:orcl",

"scott",

"tiger");

Statement st =

con.createStatement(ResultSet.TYPE\_SCROLL\_INSENSITIVE,ResultSet.CONCUR\_UPDATABLE );

String sqlQuery="select \* from emp";

ResultSet rs = st.executeQuery(sqlQuery);

if(rs.first())

{

System.out.println("fisrt record id : "+rs.getInt(1));

}

if(rs.absolute(5))

{

System.out.println("fifth record id : "+rs.getInt(1));

}

if(rs.relative(-2))

{

System.out.println("third record id : "+rs.getInt(1));

}

if(rs.relative(4))

{

System.out.println("seventh record id : "+rs.getInt(1));

}

if(rs.last())

{

System.out.println("last record id : "+rs.getInt(1));

}

con.close();

}

catch(ClassNotFoundException e)

{

e.printStackTrace();

}

finally

{

if(con!=null)

{

con.close();

}

}

}

}

**SQL Prepared Statement**

Pre-compiled statement at the database server. For the First time of request query compiles and stored at database server.

For next time of request, just calling pre-compiled query by passing values. That improves the performance of database server.

Writing query for Prepared Statement

INSERT INTO emp VAUES ( ?, ?, ?, ?, ?........);

The above query is called as parameterized query.

**Java PreparedStatement :**

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

Eg:

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:

public PreparedStatement prepareStatement(String query)throws SQLException{}

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

**Methods of PreparedStatement interface**

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.

All the above setter methods depends on data table column type.

public int executeUpdate()

executes the query. It is used for create, drop, insert, update, delete etc.

public ResultSet executeQuery()

executes the select query. It returns an instance of ResultSet.

Example of PreparedStatement interface that inserts the record

First of all create table as given below:

create table emp1( id number(10), name varchar2(50), salary number(10,2), dept varchar2(20) );

Order of setter methds using with prepared statements depends on sql query not on the table columns.

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

package com.nrit.mnrao.test;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.PreparedStatement;

import java.sql.SQLException;

public class Test {

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

Connection con = null;

PreparedStatement pst =null;

try

{

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

con = DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:orcl",

"scott",

"tiger");

String sqlQuery="INSERT INTO emp1 VALUES(?,?,?,?)";

pst = con.prepareStatement(sqlQuery);

pst.setInt(1, 1001);

pst.setString(2, "nrit1");

pst.setFloat(3,5000);

pst.setString(4,"dev");

if(pst.executeUpdate()!=0)

{

System.out.println("successfully inserted");

}

else

{

System.out.println("Failed to insert");

}

con.close();

}

catch(ClassNotFoundException e)

{

e.printStackTrace();

}

catch(SQLException e)

{

e.printStackTrace();

}

finally

{

if(con!=null)

{

con.close();

}

}

}

}

**Example of PreparedStatement interface that updates the record**

package com.nrit.mnrao.test;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.PreparedStatement;

import java.sql.SQLException;

public class Test {

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

Connection con = null;

PreparedStatement pst =null;

try

{

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

con = DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:orcl",

"scott",

"tiger");

String sqlQuery="update emp1 set salary=salary+? where dept=?";

pst = con.prepareStatement(sqlQuery);

pst.setFloat(1,2000);

pst.setString(2, "testing");

if(pst.executeUpdate()!=0)

{

System.out.println("successfully updated");

}

else

{

System.out.println("Failed to insert");

}

con.close();

}

catch(ClassNotFoundException e)

{

e.printStackTrace();

}

catch(SQLException e)

{

e.printStackTrace();

}

finally

{

if(con!=null)

{

con.close();

}

}

}

}

**Example of PreparedStatement interface for select query**

package com.nrit.mnrao.test;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.PreparedStatement;

import java.sql.ResultSet;

import java.sql.SQLException;

public class Test {

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

Connection con = null;

PreparedStatement pst =null;

try

{

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

con = DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:orcl",

"scott",

"tiger");

String sqlQuery="select \* from emp1 where dept=?";

pst = con.prepareStatement(sqlQuery);

pst.setString(1, "dev");

ResultSet rs = pst.executeQuery();

while(rs.next())

{

System.out.println(rs.getInt(1)+"\t"+

rs.getString(2)+"\t"+

rs.getDouble(3)+"\t"+

rs.getString(4));

}

con.close();

}

catch(ClassNotFoundException e)

{

e.printStackTrace();

}

catch(SQLException e)

{

e.printStackTrace();

}

finally

{

if(con!=null)

{

con.close();

}

}

}

}

**Example of PreparedStatement interface that deletes the record**

package com.nrit.mnrao.test;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.PreparedStatement;

import java.sql.ResultSet;

import java.sql.SQLException;

public class Test {

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

Connection con = null;

PreparedStatement pst =null;

try

{

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

con = DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:orcl",

"scott",

"tiger");

String sqlQuery="delete from emp1 where dept=? and salary>=?";

pst = con.prepareStatement(sqlQuery);

pst.setString(1, "dev");

pst.setDouble(2, 7000);

int count = pst.executeUpdate();

System.out.println(count+" records deleted");

con.close();

}

catch(ClassNotFoundException e)

{

e.printStackTrace();

}

catch(SQLException e)

{

e.printStackTrace();

}

finally

{

if(con!=null)

{

con.close();

}

}

}

}

**Example of PreparedStatement to insert records until user press n**

import java.io.BufferedReader;

import java.io.InputStreamReader;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.PreparedStatement;

public class RSTest {

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 emp 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());

System.out.println ("enter dept:");

String dept = br.readLine();

ps.setInt(1, id);

ps.setString(2, name);

ps.setFloat(3, salary);

ps.setString(4, dept)

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();

}

}

**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**

1) public int getColumnCount()throws SQLException

it returns the total number of columns in the ResultSet object.

2) public String getColumnName(int index)throws SQLException

it returns the column name of the specified column index.

3) public String getColumnTypeName(int index)throws SQLException

it returns the column type name for the specified index.

4)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 :

package com.nrit.mnrao.test;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.PreparedStatement;

import java.sql.ResultSet;

import java.sql.ResultSetMetaData;

import java.sql.SQLException;

public class Test {

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

Connection con = null;

PreparedStatement pst =null;

try

{

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

con = DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:orcl",

"scott",

"tiger");

String sqlQuery="select \* from emp where 1=2";

pst = con.prepareStatement(sqlQuery);

ResultSet rs = pst.executeQuery();

ResultSetMetaData metaData = rs.getMetaData();

for(int i=1 ; i<=metaData.getColumnCount(); i++)

{

System.out.println(metaData.getColumnName(i)+"\t"+

metaData.getColumnTypeName(i)+"\t"+

metaData.getColumnType(i));

}

con.close();

}

catch(ClassNotFoundException e)

{

e.printStackTrace();

}

catch(SQLException e)

{

e.printStackTrace();

}

finally

{

if(con!=null)

{

con.close();

}

}

}

}

**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.Connection;

import java.sql.DatabaseMetaData;

import java.sql.DriverManager;

public class DbmdTest

{

public static void main(String [] args)

{

Connection con = null;

try

{

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

con = DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:orcl",

"scott",

"tiger");

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);

}

}

}

**Example of DatabaseMetaData interface that prints total number of tables including system tables.**

package com.nrit.mnrao.test;

import java.sql.Connection;

import java.sql.DatabaseMetaData;

import java.sql.DriverManager;

import java.sql.PreparedStatement;

import java.sql.ResultSet;

import java.sql.SQLException;

public class Test {

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

Connection con = null;

PreparedStatement pst =null;

try

{

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

con = DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:orcl",

"scott",

"tiger");

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(ClassNotFoundException e)

{

e.printStackTrace();

}

catch(SQLException e)

{

e.printStackTrace();

}

finally

{

if(con!=null)

{

con.close();

}

}

}

}

**User tables.**

package com.nrit.mnrao.test;

import java.sql.Connection;

import java.sql.DatabaseMetaData;

import java.sql.DriverManager;

import java.sql.PreparedStatement;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.sql.Statement;

public class Test {

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

Connection con = null;

PreparedStatement pst =null;

try

{

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

con = DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:orcl",

"scott",

"tiger");

Statement st = con.createStatement();

//String query = "SELECT table\_name from user\_tables";

String query ="SELECT object\_name from user\_objects where Object\_type='TABLE'";

ResultSet rs = st.executeQuery(query);

while (rs.next())

{

System.out.println (rs.getString(1));

}

con.close();

}

catch(ClassNotFoundException e)

{

e.printStackTrace();

}

catch(SQLException e)

{

e.printStackTrace();

}

finally

{

if(con!=null)

{

con.close();

}

}

}

}

**Example of DatabaseMetaData interface that prints total number of views :**

import java.sql.Connection;

import java.sql.DatabaseMetaData;

import java.sql.DriverManager;

import java.sql.ResultSet;

public class DbmdTest {

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);

}

}

}

**User Defined views**

package com.nrit.mnrao.test;

import java.sql.Connection;

import java.sql.DatabaseMetaData;

import java.sql.DriverManager;

import java.sql.PreparedStatement;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.sql.Statement;

public class Test {

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

Connection con = null;

PreparedStatement pst =null;

try

{

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

con = DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:orcl",

"scott",

"tiger");

Statement st = con.createStatement();

//String query = "SELECT table\_name from user\_tables";

String query ="SELECT object\_name from user\_objects where Object\_type='VIEW'";

ResultSet rs = st.executeQuery(query);

while (rs.next())

{

System.out.println (rs.getString(1));

}

con.close();

}

catch(ClassNotFoundException e)

{

e.printStackTrace();

}

catch(SQLException e)

{

e.printStackTrace();

}

finally

{

if(con!=null)

{

con.close();

}

}

}

}

**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 IMGTEST

( NAME VARCHAR2(40),

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.

package com.nrit.mnrao.test;

import java.io.FileInputStream;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.PreparedStatement;

import java.sql.SQLException;

public class InsertImageTest {

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

Connection con = null;

PreparedStatement pst =null;

try

{

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

con = DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:orcl",

"scott",

"tiger");

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

pst.setString(1, "abc");

FileInputStream fin = new FileInputStream(args[0]);

pst.setBinaryStream(2, fin, fin.available());

int i = pst.executeUpdate();

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

con.close();

}

catch(ClassNotFoundException e)

{

e.printStackTrace();

}

catch(SQLException e)

{

e.printStackTrace();

}

catch(Exception e)

{

e.printStackTrace();

}

finally

{

if(con!=null)

{

con.close();

}

}

}

}

**File path pass as comman line parameter.**

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 IMGTEST

( NAME VARCHAR2(40),

PHOTO BLOB

) /

package com.nrit.mnrao.test;

import java.io.FileOutputStream;

import java.sql.Blob;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.PreparedStatement;

import java.sql.ResultSet;

import java.sql.SQLException;

public class RetrieveImageTest {

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

Connection con = null;

PreparedStatement pst =null;

try

{

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

con = DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:orcl",

"scott",

"tiger");

pst = con.prepareStatement("select \* from IMGTEST");

ResultSet rs = pst.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(args[0]);

fout.write(barr);

fout.close();

} // end of if

System.out.println ("ok");

con.close();

}

catch(ClassNotFoundException e)

{

e.printStackTrace();

}

catch(SQLException e)

{

e.printStackTrace();

}

catch(Exception e)

{

e.printStackTrace();

}

finally

{

if(con!=null)

{

con.close();

}

}

}

}

**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 FILETEST

( ID NUMBER,

NAME CLOB

)

/

package com.nrit.mnrao.test;

import java.io.File;

import java.io.FileReader;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.PreparedStatement;

import java.sql.SQLException;

public class InsertFileTest {

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

Connection con = null;

PreparedStatement pst =null;

try

{

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

con = DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:orcl",

"scott",

"tiger");

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

File f = new File(args[0]);

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(ClassNotFoundException e)

{

e.printStackTrace();

}

catch(SQLException e)

{

e.printStackTrace();

}

catch(Exception e)

{

e.printStackTrace();

}

finally

{

if(con!=null)

{

con.close();

}

}

}

}

**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 FILETEST

( ID NUMBER,

NAME CLOB

)

/

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

package com.nrit.mnrao.test;

import java.io.FileWriter;

import java.io.Reader;

import java.sql.Clob;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.PreparedStatement;

import java.sql.ResultSet;

import java.sql.SQLException;

public class RetrieveFileTest {

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

Connection con = null;

try

{

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

con = DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:orcl",

"scott",

"tiger");

PreparedStatement ps = con.prepareStatement("select \* from FILETEST");

ResultSet rs = ps.executeQuery();

rs.next();// now on 1st row

Clob c = rs.getClob(2);

Reader r = c.getCharacterStream();

FileWriter fw = new FileWriter(args[0]);

int i;

while ((i = r.read()) != -1)

fw.write((char) i);

fw.flush();

fw.close();

con.close();

System.out.println("OK");

}

catch(ClassNotFoundException e)

{

e.printStackTrace();

}

catch(SQLException e)

{

e.printStackTrace();

}

catch(Exception e)

{

e.printStackTrace();

}

finally

{

if(con!=null)

{

con.close();

}

}

}

}

**CallableStatement Interface**

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

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

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

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. |
| A procedure, can make a call to function | Function, can not call to procedure. |
| Procedure supports input, output and in-out 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 test(

eid IN NUMBER,

ename IN VARCHAR2,

salary number,

dept VARCHAR2 )

is

begin

insert into emp1 values(eid, ename, salary,dept);

end;

package com.nrit.mnrao.test;

import java.sql.CallableStatement;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.SQLException;

public class ProcedureCall {

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

Connection con = null;

try

{

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

con = DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:orcl",

"scott",

"tiger");

CallableStatement cst = con.prepareCall("{call test(?,?,?,?)}");

cst.setInt(1, 2002);

cst.setString(2, "abc1");

cst.setFloat(3, 5000);

cst.setString(4, "admin1");

if( !cst.execute() )

{

System.out.println("successfully completed");

}

else

{

System.out.println("failed");

}

}

catch(Exception e)

{

e.printStackTrace();

}

finally

{

if(con!=null)

{

con.close();

}

}

}

}

**Procedure with in, out parameters**

create or replace procedure sample( x IN NUMBER, y IN NUMBER, result OUT NUMBER )

is

temp NUMBER;

begin

temp:=x+y;

result:= temp;

end;

**java program :**

default parameters are IN params. Hence these are not required to register.

OUT parameters are to be registered.

package com.nrit.mnrao.test;

import java.sql.CallableStatement;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.SQLException;

import java.sql.Types;

public class ProcedureCall {

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

Connection con = null;

try

{

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

con = DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:orcl",

"scott",

"tiger");

CallableStatement cst = con.prepareCall("{call sample(?,?,?)}");

cst.setInt(1, 10);

cst.setInt(2, 20);

cst.registerOutParameter(3, Types.INTEGER);

cst.execute();

int res = cst.getInt(3);

System.out.println(res);

}

catch(Exception e)

{

e.printStackTrace();

}

finally

{

if(con!=null)

{

con.close();

}

}

}

}

**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 sumTest (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.

package com.nrit.mnrao.test;

import java.sql.CallableStatement;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.SQLException;

import java.sql.Types;

public class FunctionCall {

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

Connection con = null;

try

{

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

con = DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:orcl",

"scott",

"tiger");

CallableStatement cst = con.prepareCall("{ ?=call sumTest(?,?)}");

cst.registerOutParameter(1,Types.INTEGER);

cst.setInt(2, 10);

cst.setInt(3, 20);

cst.execute();

int res = cst.getInt(1);

System.out.println(res);

}

catch(Exception e)

{

e.printStackTrace();

}

finally

{

if(con!=null)

{

con.close();

}

}

}

}

**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.

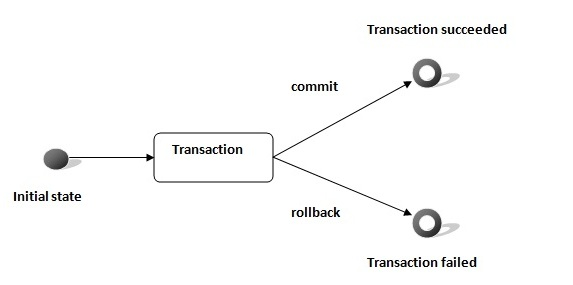
Consistency ensures bringing the database from one consistent state to another consistent state.

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.

1) void setAutoCommit(boolean status)

It is true bydefault means each transaction is committed bydefault.

2) void commit()

commits the transaction.

3) void rollback()

cancels the transaction.

create table emp1( id number(10), name varchar2(50), salary number(10,2), dept varchar2(20) );

Simple example of transaction management in jdbc using Statement

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

package com.nrit.mnrao.test;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.SQLException;

import java.sql.Statement;

public class JDBCTransactionManagement {

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

Connection con = null;

try

{

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

con = DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:orcl",

"scott",

"tiger");

con.setAutoCommit(false);

Statement stmt = con.createStatement();

stmt.executeUpdate("INSERT INTO emp1 values(1001,'abc1',5000,'dev')");

stmt.executeUpdate("INSERT INTO emp1 values(1002,'abc2',6000,'admin')");

stmt.executeUpdate("INSERT INTO emp1 values(1003,'abc3',4000,'dev')");

stmt.executeUpdate("INSERT INTO emp1 values(1004,'abc4',7000,'admin')");

con.commit();

con.close();

System.out.println("Succesfully inserted");

}

catch(Exception e)

{

e.printStackTrace();

con.rollback();

}

finally

{

if(con!=null)

{

con.close();

}

}

}

}

If you see the table emp, you will see that 2 records have been added.

Example of transaction management in jdbc using PreparedStatement

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

import java.io.BufferedReader;

import java.io.InputStreamReader;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.PreparedStatement;

public class TMTest {

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 EMP 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);

}

}

}

**Batch Processing in JDBC**

Instead of executing a single query, we can execute a batch (group) of queries. It makes the performance fast.

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:

1) void addBatch(String query) --> It adds query into batch.

2) int[] executeBatch() --> It executes the batch of queries.

**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

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.Statement;

public class InsertRecordsTest {

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 emp1 values(1001,'abc1',5000,'dev')");

stmt.addBatch("INSERT INTO emp1 values(1002,'abc1',5000,'dev')");

stmt.executeBatch();// executing the batch

con.commit();

con.close();

}

}

**Example of batch processing using PreparedStatement**

import java.io.BufferedReader;

import java.io.InputStreamReader;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.PreparedStatement;

public class BPTest {

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 EMP 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();

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

con.close();

} catch (Exception e) {

System.out.println (e);

}

}

}

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.

**Using Savepoints**

The new JDBC 3.0 Savepoint interface gives you the additional transactional control. Most modern DBMS, support savepoints within their environments such as Oracle's PL/SQL.

When you set a savepoint you define a logical rollback point within a transaction. If an error occurs past a savepoint, you can use the rollback method to undo either all the changes or only the changes made after the savepoint.

The Connection object has two new methods that help you manage savepoints −

setSavepoint(String savepointName): Defines a new savepoint. It also returns a Savepoint object.

releaseSavepoint(Savepoint savepointName): Deletes a savepoint. Notice that it requires a Savepoint object as a parameter. This object is usually a savepoint generated by the setSavepoint() method.

There is one rollback (String savepointName) method, which rolls back work to the specified savepoint.

**The following example illustrates the use of a Savepoint object −**

package com.nrit.mnrao.test;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.SQLException;

import java.sql.Savepoint;

import java.sql.Statement;

public class SavePointTest {

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

Connection con = null;

Savepoint savepoint1=null;

Savepoint savepoint2=null;

try

{

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

con = DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:orcl",

"scott",

"tiger");

con.setAutoCommit(false);

Statement stmt = con.createStatement();

// set a Savepoint

savepoint1 = con.setSavepoint("sp1");

String SQL = "INSERT INTO emp1 values(1005,'abc1',5000,'dev')";

stmt.executeUpdate(SQL);

// Submit a malformed SQL statement that breaks

SQL = "INSERT INTO emp1 values(1006,'abc1',5000,'dev')";

stmt.executeUpdate(SQL);

// If there is no error, commit the changes.

System.out.println("Succesfully inserted sp1");

savepoint2 = con.setSavepoint("sp2");

SQL = "INSERT INTO emp1 values(1010,'abc1',5000,'dev')";

stmt.executeUpdate(SQL);

// Submit a malformed SQL statement that breaks

SQL = "INSERT INTO emp1 values('xyz','abc1',5000,'dev')";

stmt.executeUpdate(SQL);

con.commit();

con.close();

System.out.println("Succesfully inserted sp2");

}

catch(Exception e)

{

e.printStackTrace();

con.rollback(savepoint2);

System.out.println("rollback sp2");

}

finally

{

if(con!=null)

{

con.close();

}

}

}

}

**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.

It is the wrapper of ResultSet. It holds tabular data like ResultSet but it is easy and flexible to use.

The implementation classes of RowSet interface are as follows:

JdbcRowSet

CachedRowSet

WebRowSet

JoinRowSet

FilteredRowSet

Let's see how to create and execute RowSet.

JdbcRowSet rowSet = RowSetProvider.newFactory().createJdbcRowSet();

rowSet.setUrl("jdbc:oracle:thin:@localhost:1521:xe");

rowSet.setUsername("system");

rowSet.setPassword("oracle");

rowSet.setCommand("select \* from EMP");

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:

It is easy and flexible to use

It is Scrollable and Updatable bydefault

**Simple example of JdbcRowSet**

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

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 RowSetTest {

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 EMP");

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));

}

}

}

**Interview Questions:**

1**) What is JDBC?**

JDBC is a Java API that is used to connect and execute query to the database. JDBC API uses jdbc drivers to connects to the database.

**2) What is JDBC Driver?**

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

JDBC-ODBC bridge driver

Native-API driver (partially java driver)

Network Protocol driver (fully java driver)

Thin driver (fully java driver)

**3) What are the steps to connect to the database in java?**

Registering the driver class

Creating connection

Creating statement

Executing queries

Closing connection

**4) What are the JDBC API components?**

The java.sql package contains interfaces and classes for JDBC API.

Interfaces:

Connection

Statement

PreparedStatement

ResultSet

ResultSetMetaData

DatabaseMetaData

CallableStatement etc.

Classes:

DriverManager

Blob

Clob

Types

SQLException etc.

**5) What are the JDBC statements?**

There are 3 JDBC statements.

Statement

PreparedStatement

CallableStatement

**6) What is the difference between Statement and PreparedStatement interface?**

In case of Statement, query is complied each time whereas in case of PreparedStatement, query is complied only once. So performance of PreparedStatement is better than Statement.

**7) How can we execute stored procedures and functions?**

By using Callable statement interface, we can execute procedures and functions.

**8) What is the role of JDBC DriverManager class?**

The DriverManager class manages the registered drivers. It can be used to register and unregister drivers. It provides factory method that returns the instance of Connection.

**9) What does the JDBC Connection interface?**

The Connection interface maintains a session with the database. It can be used for transaction management. It provides factory methods that returns the instance of Statement, PreparedStatement, CallableStatement and DatabaseMetaData.

**10) What does the JDBC ResultSet interface?**

The ResultSet object represents a row of a table. It can be used to change the cursor pointer and get the information from the database.

**11) What does the JDBC ResultSetMetaData interface?**

The ResultSetMetaData interface returns the information of table such as total number of columns, column name, column type etc.

**12) What does the JDBC DatabaseMetaData interface?**

The DatabaseMetaData interface returns the information of the database such as username, driver name, driver version, number of tables, number of views etc.

**13) Which interface is responsible for transaction management in JDBC?**

The Connection interface provides methods for transaction management such as commit(), rollback() etc.

**14) What is batch processing and how to perform batch processing in JDBC?**

By using batch processing technique in JDBC, we can execute multiple queries. It makes the performance fast.

**15) How can we store and retrieve images from the database?**

By using PreparedStatement interface, we can store and retrieve image