Tech Mahindra

Java DataBase Connectivity (JDBC)

TLS

Rewards and Recognition

2012 ASTD BEST AWARD WINNER

Training TOP 125 Winner - 2013

Tata Institute of Social Sciences CLO Awards India Best Virtual Learning Award

Objective

At the end of the session, participants will be able to:

- Understand JDBC Architecture
- Perform CRUD operations using JDBC
- Execute different queries form front end to backend
- Manage Transaction using JDBC

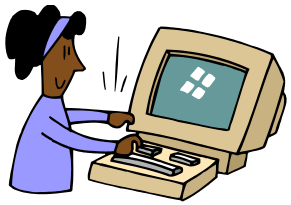
Agenda

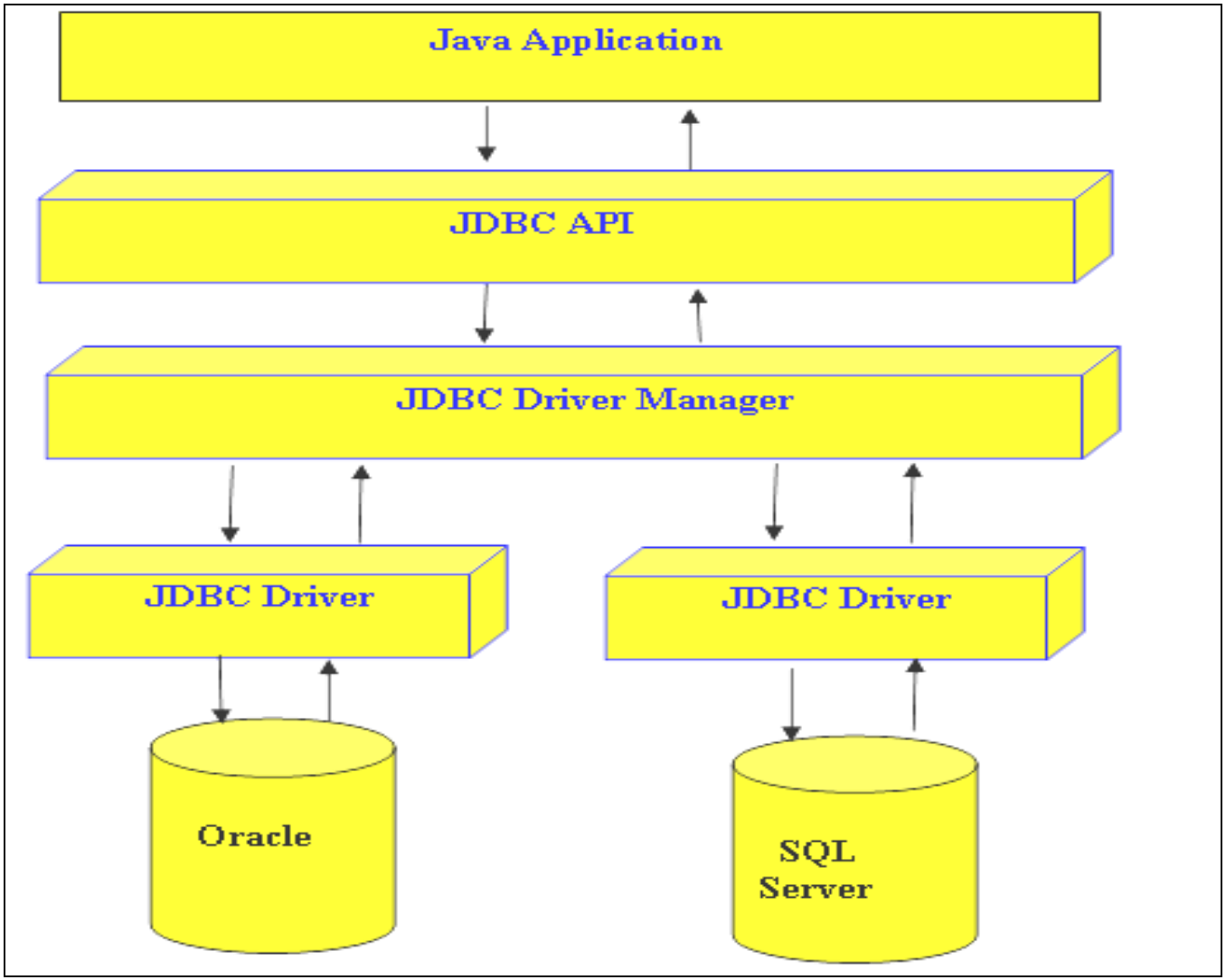
- JDBC Architecture
- Loading JDBC Driver
- Connecting to Database
- Manipulate data in the Database
- Transaction Management

What is JDBC?

- An API specification developed by Sun Microsystems
- Defines a uniform interface for accessing various relational databases
- The JDBC API uses a Driver Manager and database-specific drivers to provide connectivity to heterogeneous databases
- Driver Manager can support multiple concurrent drivers connected to multiple heterogeneous databases
- A JDBC driver translates standard JDBC calls into a network or database protocol or into a database library API call that facilitates communication with the database

JDBC Architecture





Types of JDBC Drivers

Four distinct types of JDBC drivers:

- JDBC-ODBC Bridge Driver

  OR Type1

- Native API Java Driver

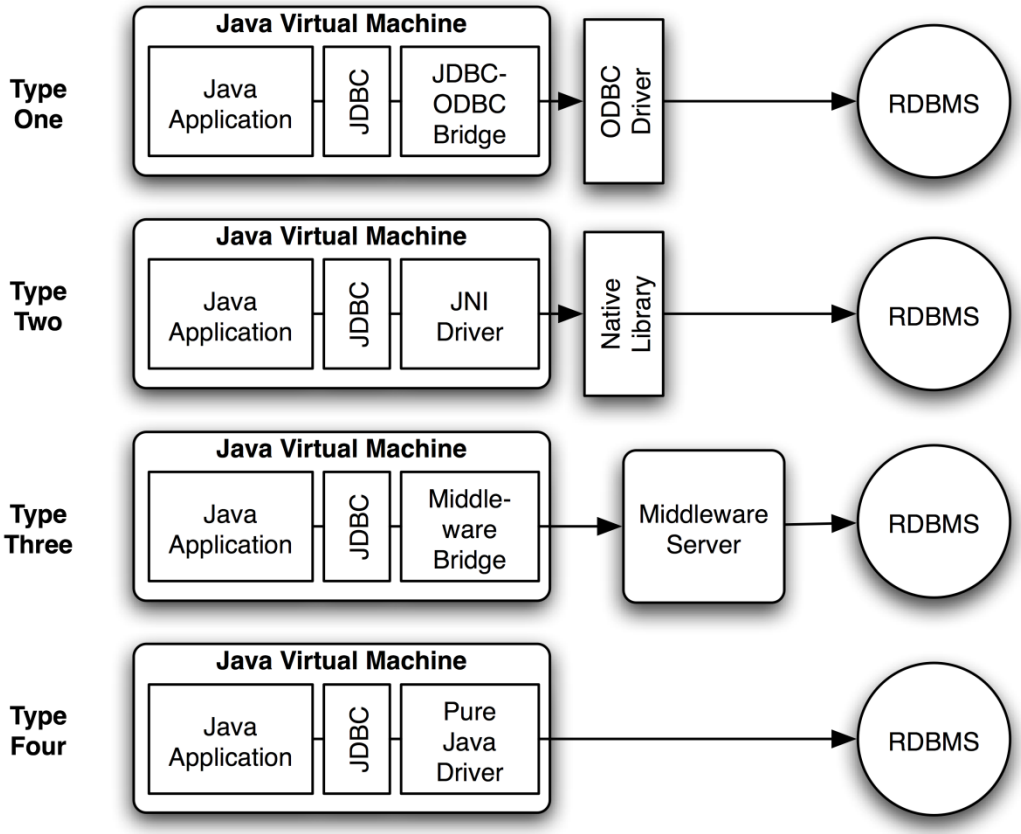
  OR Type2

- Java to Network Protocol

  OR Driver - Type3

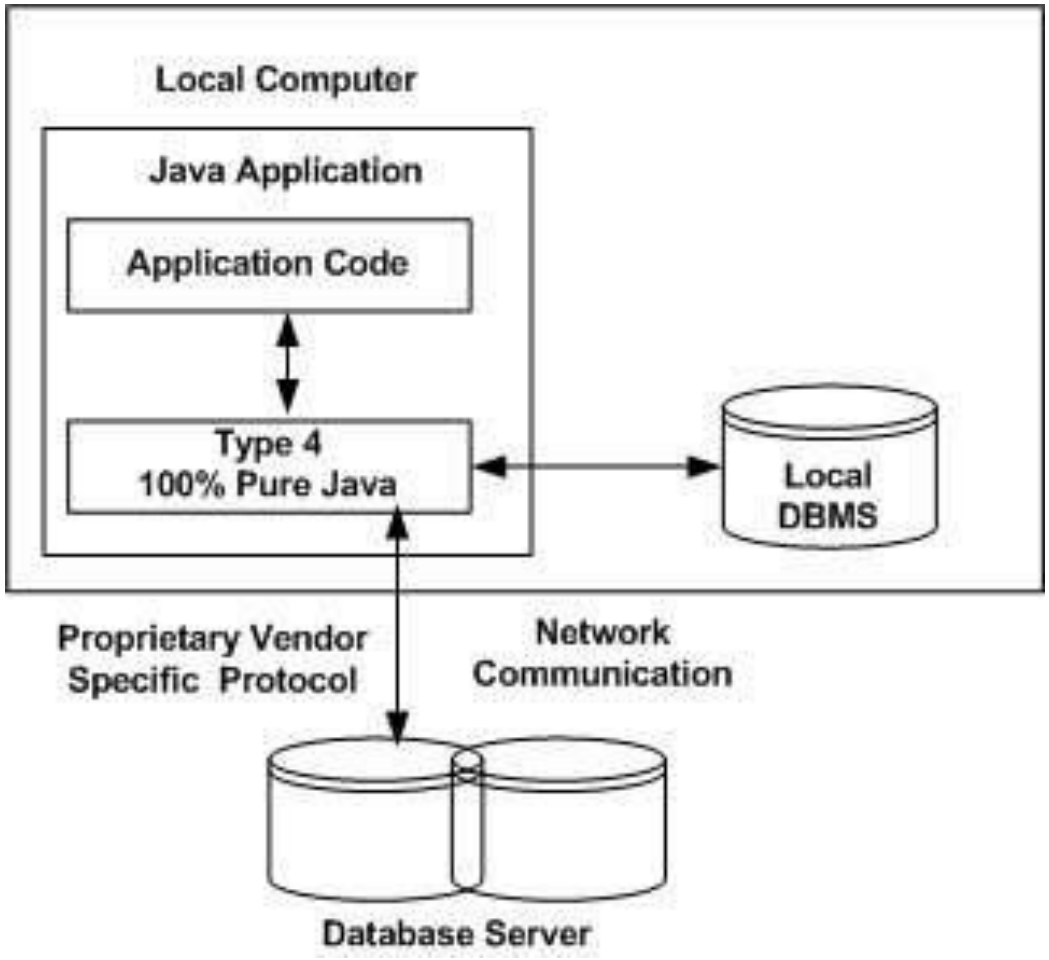
- Pure Java Driver

  OR Type4



[**Note** : Type 4 driver is frequently used in applications because of its features.]

Type 4: Java to Database Protocol Driver



Type 4: Java to Database Protocol Driver (Contd…)

- Pure Java drivers that implement a proprietary database protocol to communicate directly with the database
- Like Type 3 drivers, they do not require native database libraries and can be deployed over the Internet without client installation
- Communicate directly with the database engine rather than through middleware or a native library
- Usually the fastest JDBC drivers available
- Directly converts java statements to SQL statements
- **Drawback** :They are database / Vendor specific

Steps for writing JDBC code

1. Load and register the driver class
2. Establish the connection
3. Create a Statement
4. Execute the query on DB
5. Getting result in Java
6. Close the connection

1. Loading and registering the Driver

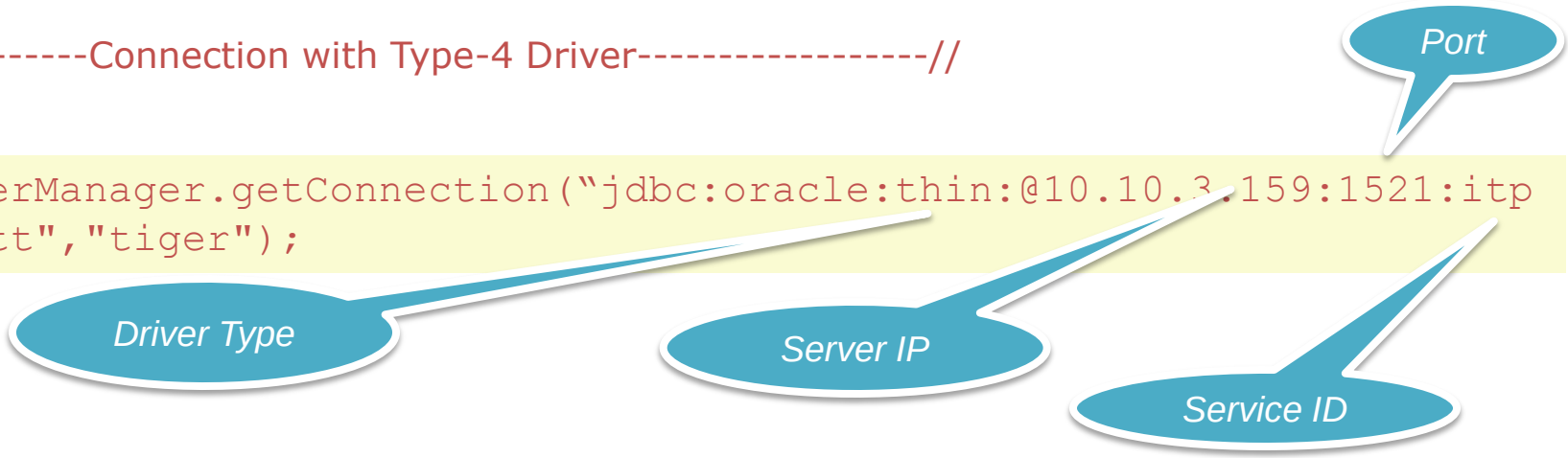
- Can be done in two ways:
  - Using *DriverManager*: Uses bulit in method registerDriver() to register the driver.
  - Using *Class.forName():*
    - A fully qualified name of the driver class should be passed as a parameter
    - The following call will load the Sun JDBC-ODBC driver:

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

2. Connecting to Database

- The *getConnection()* method is :
  - A static method of *DriverManager* class
  - Provides many overloaded versions
  - Returns a *Connection* object

```
//----------Connection with Type-4 Driver-----------------//
con=DriverManager.getConnection("jdbc:oracle:thin:@10.10.3.159:1521:itpdb","scott","tiger");
```



Connection API Interface

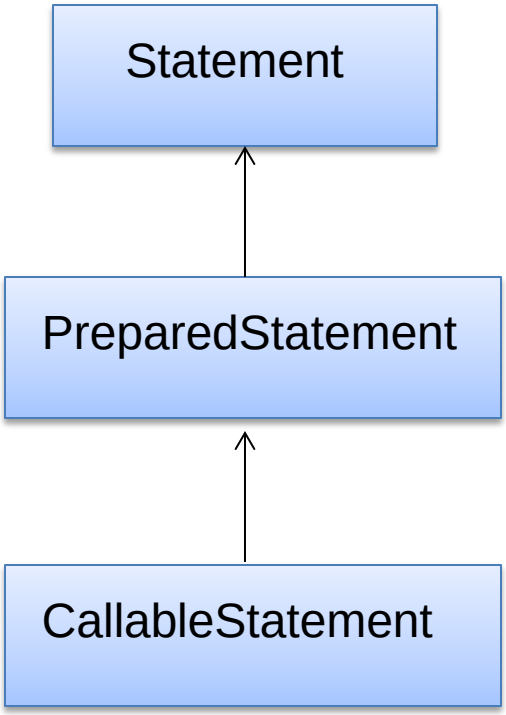
- When a Connection is opened, this represents a single instance of a particular database session

- As long as the connection remains open, SQL queries may be executed and results obtained
- Connection can be obtained from DriverManager or from DataSource(which can also return connection object from connection pool / JNDI).

- Has Methods which return objects of various types of Statements.
  - *createStatement()*
  - *preparedStatement()*
  - *prepareCall()*

3. Creating a statement using Statement API Interface

- There are three types of statements. Based upon requirement any one can be used in application



Statement API Interface

- Used pass a SQL String to the database for execution and to retrieve result from database

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

- The *createStatement()* method of Connection interface returns a Statement object

PreparedStatement API Interface

- The *prepareStatement()* method of Connection interface returns a PreparedStatement object

- Useful for frequently executed SQL statements and for accepting user inputs frequently at runtime.

- An SQL statement is pre-compiled and gets executed more efficiently than a plain statement

- For example: the statement "INSERT INTO EMP VALUES (?, ?)" can be used to add multiple records, with a different values each time
  - '?' acts as a placeholder

- Records are in *emp* table as follows:

```
pspmt=con.prepareStatement("insert into emp values(?,?)");
pspmt.setInt(1,26788);
pspmt.setString(2,"Ajay");
```

The PreparedStatement Interface (Contd…)

- The *PreparedStatement* Interface:
  - methods *executeQuery()* and *executeUpdate* for statement execution
  - *setXXX()* methods for assigning values for the placeholders
    - XXX stands for the data type of the value of the placeholder
    - For example: *setString(), setInt(), setFloat(), setDouble()* etc.
- To delete a record, the following code snippet is used:

```
pstmt = con.prepareStatement("Delete FROM emp WHERE empno=?”)
pstmt.setString(1,eno);
int i = pstmt.executeUpdate();
```

The PreparedStatement Interface (Contd…)

- To update records, the following code snippet can be used:

```
PreparedStatement stmt = con.prepareStatement("update Emp SET
Ename = ? WHERE Empno = ?");
st.setString(1,"Amit");
st.setString(2,"50000");
int i = stmt.executeUpdate();
```

- **Best Practices** : Use PreparedStatement for executing DML and SELECT queries as they are more efficient and can execute dynamic queries .

CallableStatement API Interface

- A *CallableStatement* object is created by calling the *prepareCall()* method on a Connection object
- Useful for calling and executing stored procedure (stored in backend) from Java (front end)
- The object provides a way to call stored procedures in a standard way for all DBMS
- *CallableStatement* inherits *Statement* methods, which deal with SQL statements in general, and it also inherits *PreparedStatement* methods, which deal with IN parameters
- All methods defined in the *CallableStatement* deal with OUT parameters or the output aspect of INOUT parameters

CallableStatement Interface (Contd…)

- Syntax for a stored procedure without parameters is:

```
{call procedure_name}
```

- Syntax for procedure call with two parameters:

```
{call procedure_name[(?, ?)] }
```

The CallableStatement Interface (Contd…)

// Creating CallableStatement

```
CallableStatement cs;
cs=con.prepareCall("{call SHOW_SUPPLIERS}");
ResultSet rs = cs.executeQuery();
```

// Stored Procedures with parameters

```
int age = 39;
String poetName = "dylan thomas";
CallableStatement proc =
con.prepareCall("{call set_age(?, ?)}");
proc.setString(1, poetName);
proc.setInt(2, age);
cs.execute();
```

Give this a Try…

1. Which Driver uses JDBC-ODBC Bridge to translate JDBC statements calls to the ODBC calls?

2. Which driver converts JDBC calls into the vendor-specific database protocol so that client applications can communicate directly with the database server?

3. Which interface do you think provides a way to call stored procedures in a standard way for all DBMS?

4. Which class should be used to efficiently execute SQL statement multiple times?

4.Executing the Query

- Provides methods to execute SQL statements:
  - executeQuery()
    - Used for SQL statements such as simple **SELECT**, which return a single result set
  - executeUpdate()
    - Used for other SQL statements like **INSERT**, **UPDATE** & **DELETE**
  - execute()
    - Returns true if the first object that the query returns is a ResultSet object
    - Use this method if the query could return one or more ResultSet objects. Retrieve the ResultSet objects returned from the query by repeatedly calling Statement.getResultSet().

5. Receiving output in java using *ResultSet* Interface

- The result of an SQL statement execution can be:
  - A *ResultSet* object
    - The object returned by *executeQuery()* method
    - Contains records retrieved by the SELECT statement
  - An integer
    - Returned in case of INSERT, UPDATE, DELETE statements
    - Indicates the number of rows affected due to the statement

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

The ResultSet Interface (Contd…)

- A *ResultSet* object represents the output table of data resulted from a SELECT query statement with following features:
- The data in a *ResultSet* object is organized in rows & columns
- *Each ResultSet object maintains a cursor (pointer) to identify the current data row*
- The cursor of a newly created *ResultSet* object is positioned before the first row

The ResultSet Interface (Contd…)

- Movement of the cursor depends on the scrollability of the ResultSet

  - Non-scrollable ResultSet:
    - Object type is *ResultSet.TYPE_FORWARD_ONLY*, the default type
    - Supports only forward move

  - Scrollable ResultSet:
    - Object type is *ResultSet.TYPE_SCROLL_INSENSITIVE* (scrollable but not sensitive to changes made by others) or
    - *ResultSet.TYPE_SCROLL_SENSITIVE* (scrollable and sensitive to changes made by others)

The ResultSet Interface (Contd…)

- *next()* method:
  - Moves the record pointer to the next record in the result set
  - Returns a boolean value true / false, depending on whether there are records in the result set

- *getXXX()* methods:
  - XXX stands for data type of the value being retrieved
  - Retrieve the values of individual column in the result set
  - Two overloaded versions of each *getXXX()* method are provided:

    1. Accepting an integer argument indicating the column position
    2. Accepting a string argument indicating name of the column

***Note** : The recommended approach is to use column names int getXXX() cause if you use index and the column gets interchanged in DB, the data access logic has to be changed to reflect new indexes.*

The ResultSet Interface (Contd…)

```
 Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT empno, ename ,sal
FROM emp");
while (rs.next() )
{
            //assuming there are 3 columns in the table
System.out.println( rs.getString(1));
System.out.println(rs.getString(2));
System.out.println(rs.getString(3));

}
rs.close();    //---- First ----//
stmt.close(); //---- Second ----//
con.close();   //---- Last ----//

 System.out.println(" You are done");
```

Don't forget to close the Resultset, Statement & Connection

Maintain the sequence as shown

6. Closing the connection

- Once the data access is done, the resultSet, Statement and connection should be closed in reverse order to ensure corresponding user process and memory allocated is released.
- This can be done as follows:

```
try
{
  // JDBC Code
}
catch(Exception e)
{
  // Exception handling code
}
finally
{
  rs.close();
  stmt.close();
  con.close();
}
```

Closing the connection Cont..

- From JDK 1.7 onwards we can have AutoClosable functionality for resources in try – catch block. This is called as try with resource facility in JDK 1.7

```
try (Statement stmt = con.createStatement())
{
        ResultSet rs = stmt.executeQuery(query);
        while (rs.next())
         {
          String coffeeName = rs.getString("COF_NAME");
          int supplierID = rs.getInt("SUP_ID");
          float price = rs.getFloat("PRICE");
          int sales = rs.getInt("SALES");
          int total = rs.getInt("TOTAL");
          System.out.println(coffeeName + ", " + supplierID + ", " + price + ", " + sales + ", " + total);
 }
 } catch (SQLException e)
   {
      // Exception handling code
   }
```

- Only the objects that implements java.lang.AutoClosable interface can have this functionality.

Closing the connection Cont..

- **Best Practices** : Always close connections (or any resource) after query execution (or use). Failure in closing connection will result in database server running out of connection and thus resulting in connection failure and application crash.
- **So always close connection in finally block or use try with resources feature (JDK 1.7 Onwards) to follow best practices.**

Brain Teaser..

- ResultSet connects to database and fetches record step by step. Till entire records are fetched, the connection is maintained or is required.

- Lets consider a scenario that we are in middle of salary processing for employees and what if the connection is broken in middle of records ? Will it allow us to continue there after ? What can we do to avoid this?

Solution - JDBC RowSet

- It is the wrapper of ResultSet. It holds tabular data like ResultSet but it is easy and flexible to use. Introduced from JDK 1.5
- **Advantages :**
  - Easy and flexible to use
  - By default Scrollable and Updatable
- A Rowset does not hold on to the database connection which was used to fetch the results of a query. Hence RowSet is Serializable. This makes a RowSet more usable in situations where we need to transfer query result over the wire.
- RowSet is essentially a Java Bean. It has all the great advantages of any typical Java Bean component. One of the most distinguishing advantages of RowSet is the ability to emit events based on the actions on the RowSet. The common events which can be used with RowSet are Update Event, Delete Event and an Insert Event.

JDBC RowSet Cont..

- **Types of RowSet :**
  - Connected
  - Disconnected

- **Categories according to implementations**
  - *JdbcRowSet*
  - *CachedRowSet*
  - *WebRowSet*
  - *FilteredRowSet*
  - *JoinRowSet*.

RowSet Implementation Eg:

```
package com.techm;

import java.sql.SQLException;
import javax.sql.rowset.JdbcRowSet;
import javax.sql.rowset.RowSetProvider;

public class RowSetDemo{

public static void main(String[] args) {
 //Get rowSet
  JdbcRowSet rowSet = RowSetProvider.newFactory().createJdbcRowSet();
try {
   //loading driver
      Class.forName("oracle.jdbc.driver.OracleDriver");

  //configure rowset
        rowSet.setUrl("jdbc:oracle:thin:@localhost:1521:xe");
        rowSet.setUsername("system");
        rowSet.setPassword("oracle");
//execute the query
        rowSet.setCommand("select * from emp");
        rowSet.execute();

while(rowSet.next())
{
 System.out.println("Emp No="+rowSet.getInt("EmpNo")+" name="+rowSet.getString("EmpName"));
}
```

RowSet Implementation (Cont..)

```
  }  //closing of try
catch (ClassNotFoundException e)
   {
          e.printStackTrace();
          System.out.println("Error in loading Driver");
   }catch (SQLException e)
    {
           e.printStackTrace();
           System.out.println("Error in SQL execution");
     }
    finally
     {
          try {
                  rowSet.close();
               } catch (SQLException e) {
                    e.printStackTrace();
                    System.out.println("Error in closing rowset");}
     }
  }  //closing of main

}  //closing of class
```

RowSet Vs ResultSet ,which should be used?

- There is no rule to use RowSet only. It depends upon requirement and feasibility of the code.
- A **ResultSet** is a essentially a connected view of the query result. The Database connection which was used for fetching the ResultSet will be held. This makes a ResultSet non-serializable.
- **RowSet** keeps all the data from the query result in-memory. This is very in-efficient with queries that return huge data.
- As the data from the **ResultSet** is preloaded into the RowSet, you cannot re query a particular column as a different datum. For example, lets say we want to query a Clob type as a String. In case of a RowSet, we need to change the returned Clob type to string inside the Java Code. When using a ResultSet, we can re query the same column using getString(...) method to retrieve the result as a String
- In **summary**, RowSet and ResultSet can compliment each other nicely and each one has their own advantages and disadvantages. The key is to use the right API in the right use-case.

Slide 36

SS4 has to be in correct case
Sanjana Sane, 9/8/2016

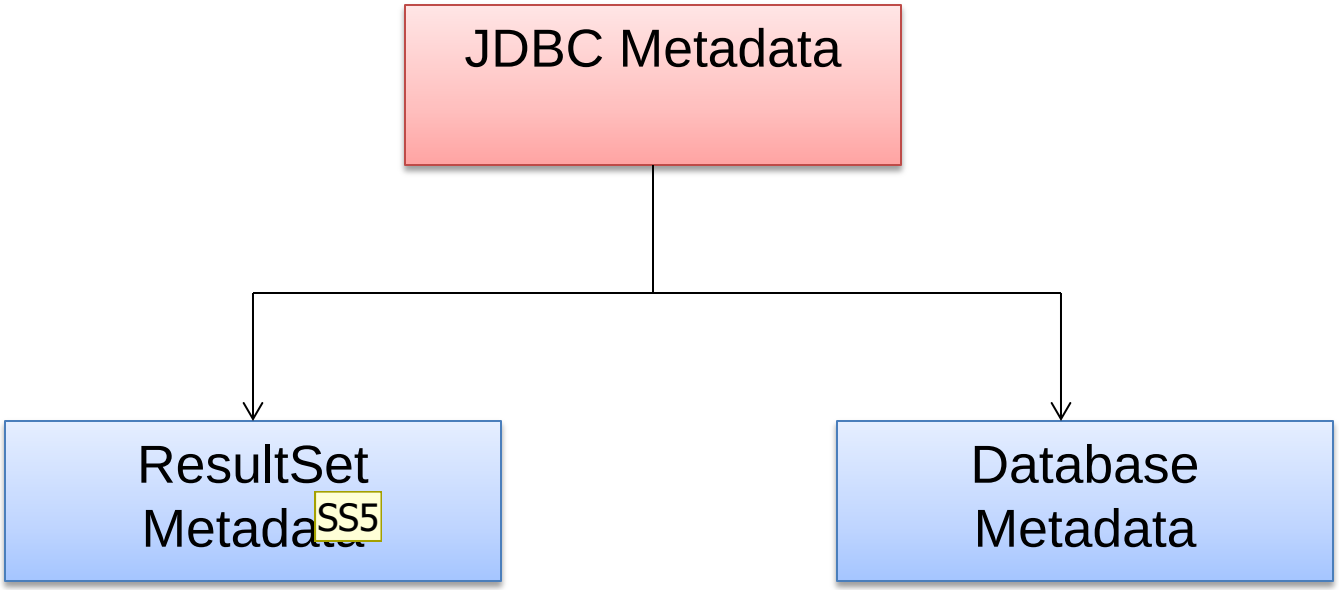
Brain Teaser..

- There is a requirement in projects where application need to work with table / columns existing in a database. But UI access to database / database client IDE is not be available. You only have credentials and does not have any details about tables / columns to work upon.

- How can we know about database table / columns from Java?

Solution - Metadata

- Metadata is the key to unlock this problem.
- JBDC gives us metadata which holds information about tables, columns in a database.
- It can be obtained by using one of the following ways



Slide 38

SS5 has to be written in correct case
Sanjana Sane, 9/8/2016

The ResultSetMetadata Interface

- *getMetaData()* of the ResultSet interface returns a *ResultSetMetaData* object containing details about the columns in a result set

- Some of the methods of *ResultSetMetaData* interface are:

  - *getColumnName()*
    - Returns the name of a column by taking an integer argument indicating the position of the column within the result set

  - *getColumnType()*
    - Returns the data type of a column

  - *getColumnCount()*
    - Returns the number of columns included in the result set

The ResultSetMetadata Interface (Contd…)

```
rsmat=rs.getMetaData();

int cols=rsmat.getColumnCount()

while(rs.next()
 {
       for(int i=1;i<=cols;i++)
       {
          System.out.print(rs.getString(i));
       }
 }
```

The DatabaseMetaData Interface

- Through the java.sql.DatabaseMetaData interface you can obtain meta data about the database you have connected to. For instance, you can see what tables are defined in the database, and what columns each table has, whether given features are supported etc.
- To get the instance of the interface getMetadata() can be used.

```
DatabaseMetaData dbMetaData = con.getMetaData();
```

- Once the instance is obtained, we can use various methods available in interface to get the information about db. Few of the methods are:
  - *getDatabaseProductName()*
  - *getDatabaseProductVersion()*
  - *getTables(..)*
  - *getColumns(..)*

The DatabaseMetadata Interface Cont..

- Demo to print no of tables in DB.

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


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

Brain Teaser..

- In web applications where multiple users access the same code / connection object at a time, how many connection objects should be created ?

  Single or multiple connection objects for multiple users?

- In case of single object – will it be sufficient to hold large information of all the active user at the same time?
- In case of multiple – can server store all these objects in memory at the same time? Will it be easy to create these objects in memory and to close them frequently at runtime when load is high?

Solution - Connection Pooling

- Java application creates a connection object every time to connect to a database
- **Problem** :
  - Process is resource heavy
  - Only one user can use connection object at a time. Results in blockage if multiple users wish to access same connection object to connect with database.
- **Solution** :
  - Multiple connection objects with same characteristics can be created and held in a pool for use as and when required.

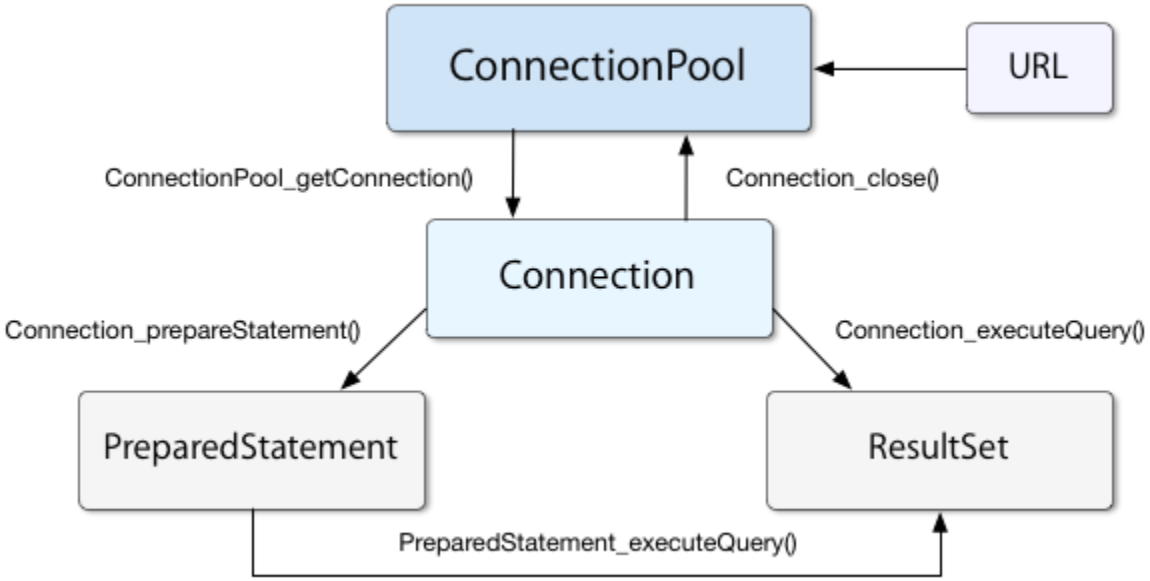
How it works ?

- In software engineering, a **connection pool** is a cache of database connections maintained so that the connections can be reused when future requests to the database are required.

- Connection pools are used to enhance the performance of executing commands on a database. Opening and maintaining a database connection for each user, especially requests made to a dynamic database-driven website application, is costly and wastes resources.

- In connection pooling, after a connection is created, it is placed in the pool and it is used again so that a new connection does not have to be established. If all the connections are being used, a new connection is made and is added to the pool. Connection pooling also cuts down on the amount of time a user must wait to establish a connection to the database.

How it works ? Cont..



Introduction to Transaction

- An indivisible unit of work comprised of several operations
- Either all, or none of the units need to be performed in order to preserve data integrity
- A complete task which is a combination of the smaller tasks
- For a major task to be completed, smaller tasks need to be successfully completed
- If any one task fails then all the previous tasks are reverted back to the original state

Transaction Properties

- Atomicity:
  - Implies indivisibility
  - Any indivisible operation (one which will either complete in totally, or not at all) is said to be atomic
- Consistency:
  - A transaction must transition persistent data from one consistent state to another
  - In the event of a failure occurs during processing, data must be restored to the state it was in prior to the transaction

Transaction Properties (Contd…)

- Isolation:
  - Transactions should not affect each other
  - A transaction in progress, not yet committed or rolled back, must be isolated from other transactions
- Durability:
  - Once a transaction commits successfully, the state changes committed by that transaction must be durable & persistent, despite any failures that occur afterwards

Transaction Management

- A connection commits all the changes once it is done with a query by default
- This can be stopped by calling *setAutoCommit(false)* on the connection object
- Now for every transaction, user will have to explicitly call the *commit()* method, else the changes wont be reflected in the database
- The *rollback()* method gets called for withdrawing all the changes done by queries in the database

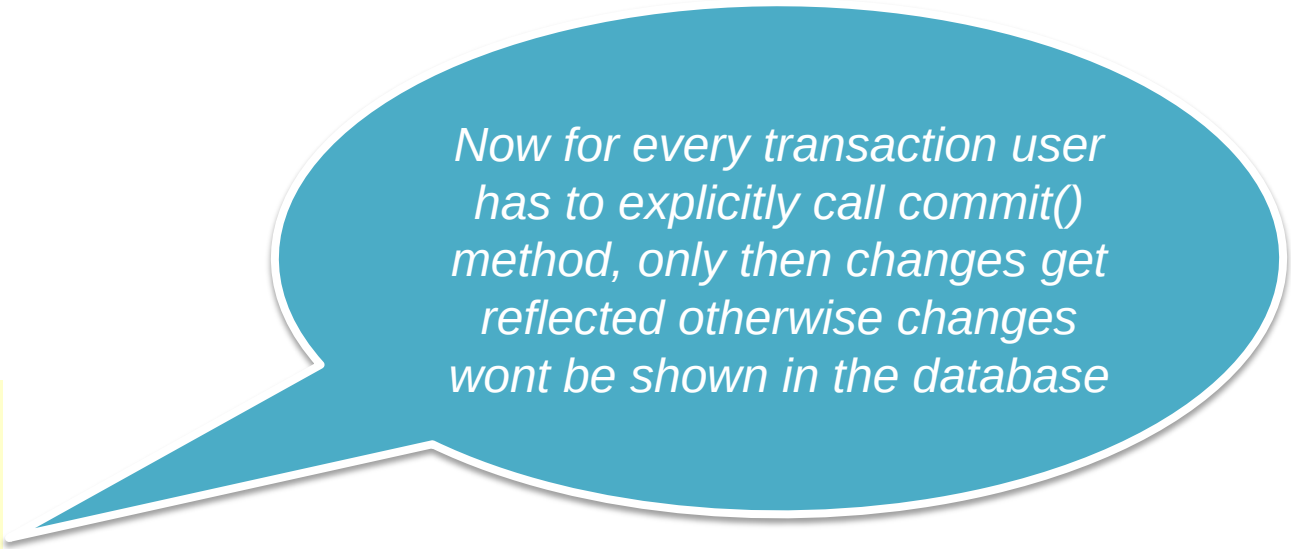
Transaction Management (Contd…)

By default the Connection commits all the changes once it is done with a single query. This can be stopped by calling setAutoCommit(false) on Connection object.

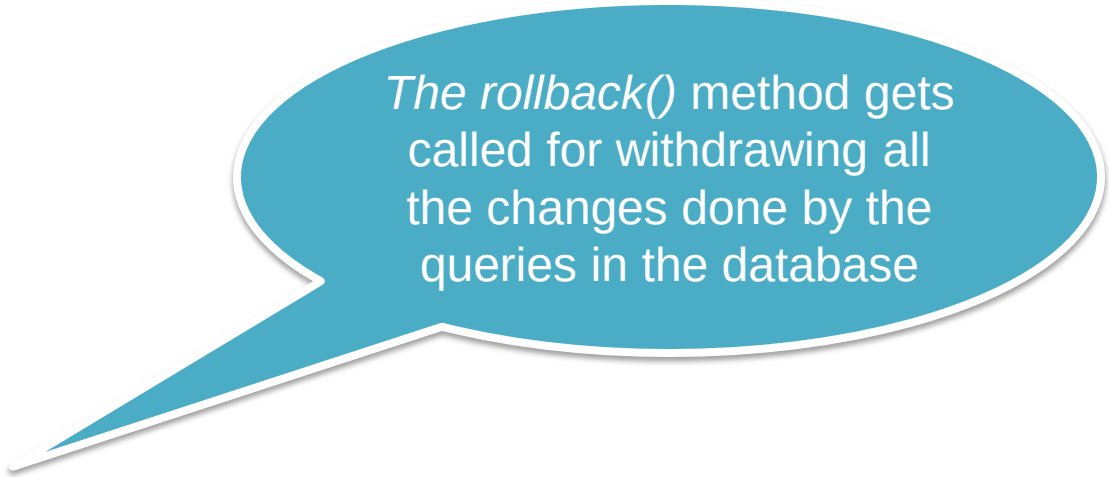
```
con.setAutoCommit(false);
System.out.println("Connection stablished");
stmt=con.createStatement();
```

Transaction Management (Contd…)

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



Transaction Management (Contd…)




```
con.commit();
con.rollback();
```

Summary

In this session, we have covered:

- JDBC Architecture
- Loading the drivers
- Connecting to Database
- Types of statements
- ResultSet and RowSet
- JDBC Metada
- Connection pooling
- Transaction Management

Thank You

Disclaimer

Tech Mahindra Limited, herein referred to as TechM provide a wide array of presentations and reports, with the contributions of various professionals. These presentations and reports are for informational purposes and private circulation only and do not constitute an offer to buy or sell any securities mentioned therein. They do not purport to be a complete description of the markets conditions or developments referred to in the material. While utmost care has been taken in preparing the above, we claim no responsibility for their accuracy. We shall not be liable for any direct or indirect losses arising from the use thereof and the viewers are requested to use the information contained herein at their own risk. These presentations and reports should not be reproduced, re-circulated, published in any media, website or otherwise, in any form or manner, in part or as a whole, without the express consent in writing of TechM or its subsidiaries. Any unauthorized use, disclosure or public dissemination of information contained herein is prohibited. Unless specifically noted, TechM is not responsible for the content of these presentations and/or the opinions of the presenters. Individual situations and local practices and standards may vary, so viewers and others utilizing information contained within a presentation are free to adopt differing standards and approaches as they see fit. You may not repackage or sell the presentation. Products and names mentioned in materials or presentations are the property of their respective owners and the mention of them does not constitute an endorsement by TechM. Information contained in a presentation hosted or promoted by TechM is provided “as is” without warranty of any kind, either expressed or implied, including any warranty of merchantability or fitness for a particular purpose. TechM assumes no liability or responsibility for the contents of a presentation or the opinions expressed by the presenters. All expressions of opinion are subject to change without notice.