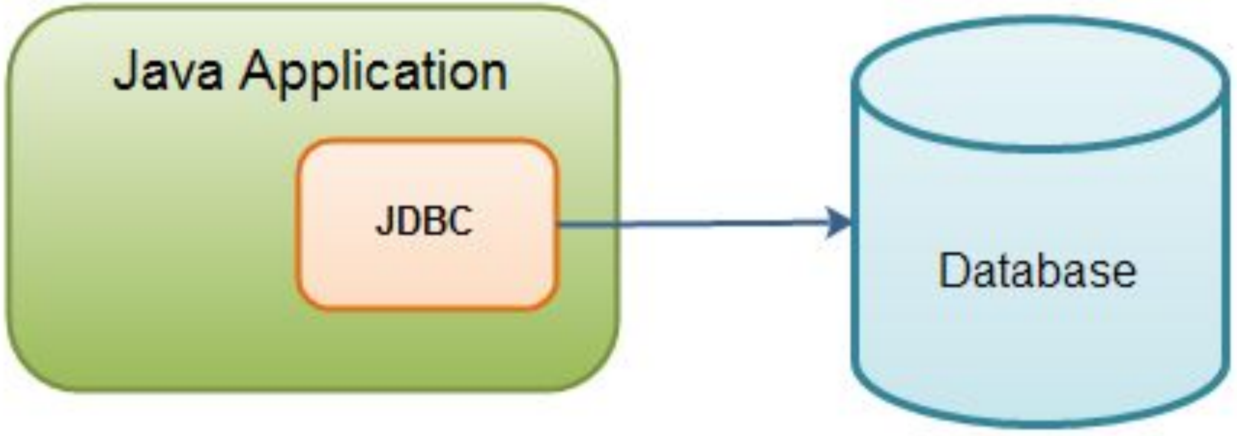
JDBC Programming

- What is Database connectivity?
- Java Database connectivity
- JDBC Driver Type
- JDBC APIs
- Brief Overview of JDBC process
- Statement Interface
- Scrollable ResultSet

What is Database Connectivity?

- Requirement of DBMS connectivity.
- Many commercial Database Management System available in market like Oracle, MS SQL Server, My SQL, DB2, Sybase etc…
- Application provides facility to communicate with Database.
- You can perform different operation like creating database and tables, insert, update and delete data in Database.

Java Database Connectivity



Java Database Connectivity

- Java Application cannot directly communicate with a database to submit data & retrieve the results of queries. This is because a database can interpret only SQL statements & not java language statements. So JDBC is a mechanism to translate java statements into SQL statements.
- JDBC is a Java API for executing SQL statements.

- JDBC is an application programming interface (API) for the programming language **Java**, that defines how a client may access a **database**.

- It is part of the **Java** Standard Edition platform, from Oracle Corporation.
- Sun Microsystems released JDBC as part of JDK 1.1 on February 19, 1997
- The latest version is JDBC 4.2.

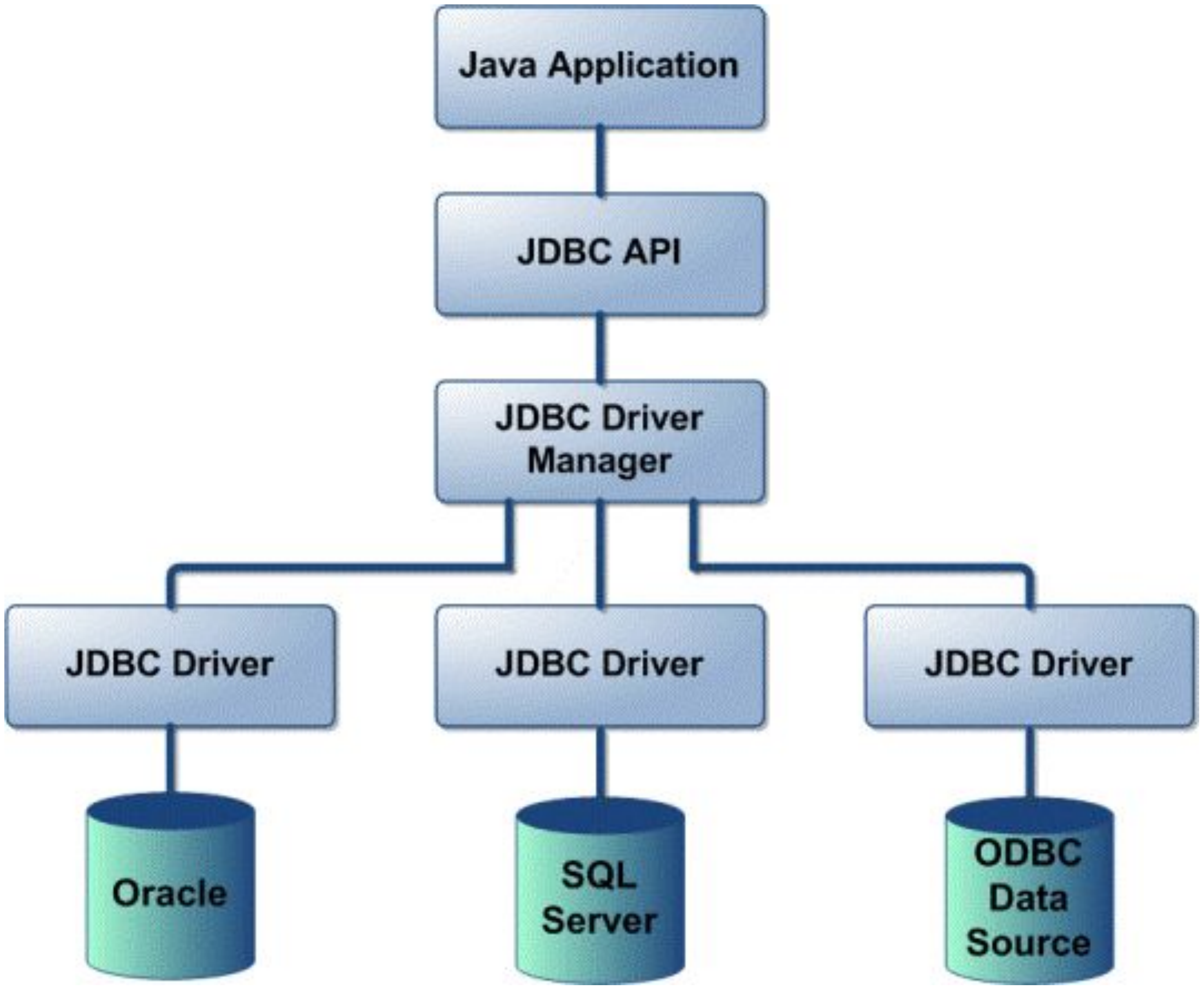
Java Database Connectivity

- What JDBC Do?
  - establish a connection with database
  - Send SQL statements
  - Process the results

JDBC and ODBC

- ODBC is for Microsoft and JDBC is for java applications.
- ODBC can't be directly used with Java because it uses a C interface.
- ODBC makes use of pointers which have been removed totally from java.

JDBC Architecture



JDBC Architecture

Components of JDBC

- **The JDBC API** – it provides various methods and interfaces for easy and effective communication with the databases.

- **The JDBC DriverManager** – it is a backbone of JDBC architecture. it connects a Java application to correct JDBC driver.

- **The JDBC test suite** – it evaluates the JDBC driver for its

compatibility with java EE.

- **The JDBC-ODBC bridge** – it connects database drivers to the database. This bridge translate JDBC method calls to ODBC function calls.

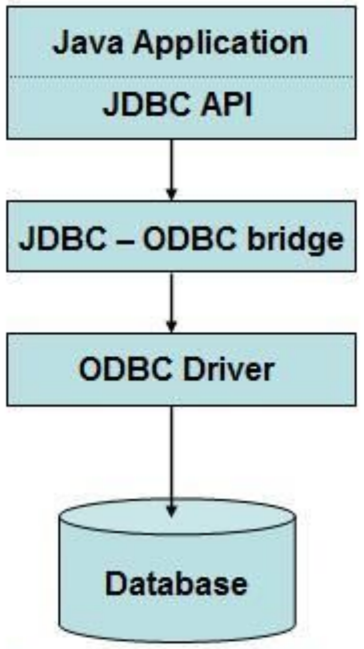
JDBC driver types

There are four kind of JDBC drivers available.

- Type 1 Driver – JDBC – ODBC bridge
- Type 2 Driver – Native API Partly Java Driver
- Type 3 Driver – Pure Java Driver for Database Middleware
- Type 4 Driver – Pure Java Driver

Type-1 Driver

- The Type – 1 driver acts as a bridge between JDBC and other database connectivity mechanisms, such as ODBC.
- An example of this type of driver is the Sun JDBC-ODBC driver



Type 1: JDBC-ODBC Bridge

Type-1 Driver

- JAVA App makes call to JDBC ODBC Bridge Driver
- JDBC ODBC Bridge Driver resolves JDBC Call and makes equivalent ODBC call to ODBC driver.
- ODBC Driver Completes the request and sends response to JDBC ODBC Bridge Driver.
- JDBC ODBC Bridge Driver converts the response to JDBC Standards and Displays the result into requesting JAVA App.
- Ex: Sun JDBC ODBC Bridge Driver

Type-1 Driver

Advantages:

- Represent single driver implementation to interact with different data source
- Allow you to communicate with all the databases supported by the ODBC driver

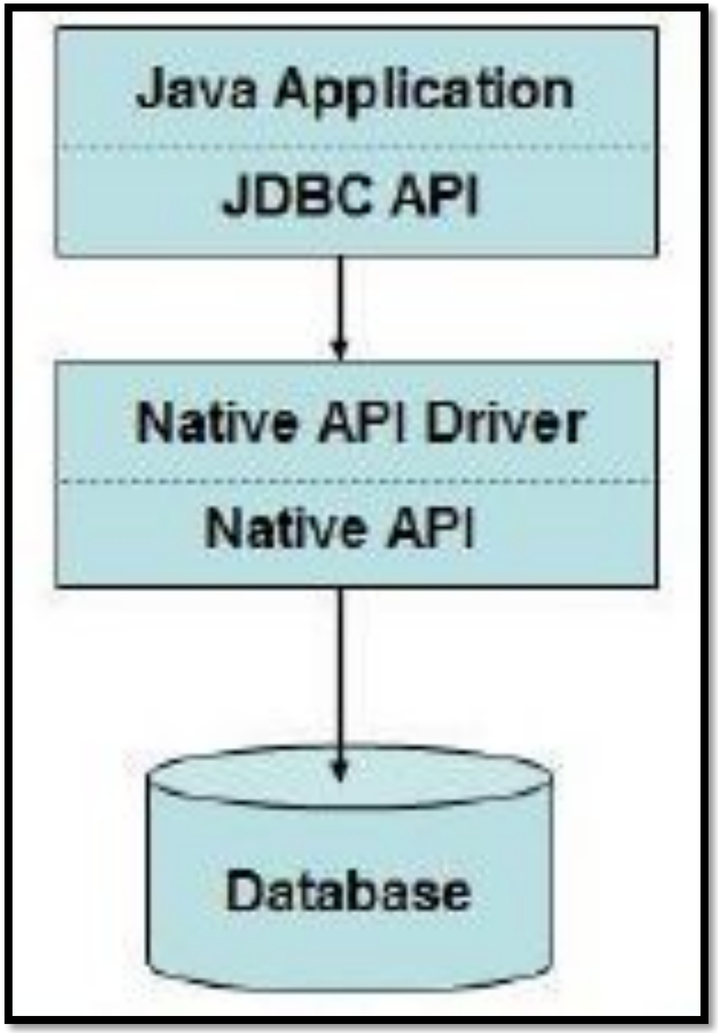
Disadvantages:

- Decreases the execution speed due to a large number of translations.
- It depends on ODBC driver.
- Requires the ODBC binary code or ODBC client library that must be installed on every client.

TYPE -2 DRIVER
Native API Partly Java Driver

- Follows 2- Tier Architecture
- JDBC calls are converted into database specific native calls in the Client Machine & Request Dispatched to database Specific native Libraries.
- Native Database Libraries sends requests to database server by using native protocol
- Ex: Driver for Sybase

TYPE -2 DRIVER



TYPE -2 DRIVER

Advantages :

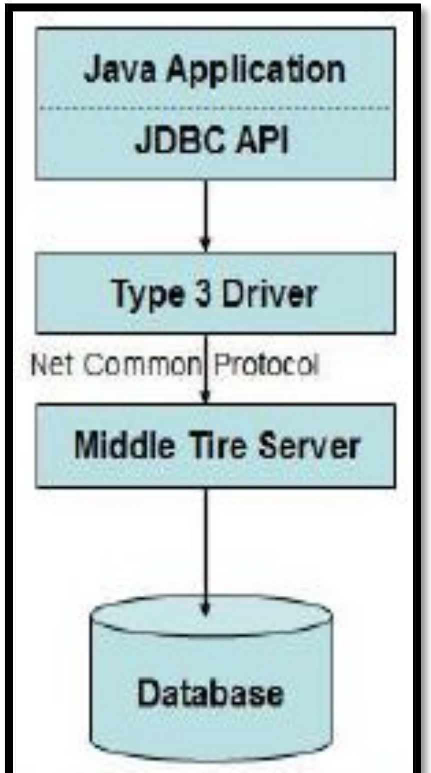
- Helps to access the data faster as compared to other types of drivers.

Disadvantages :

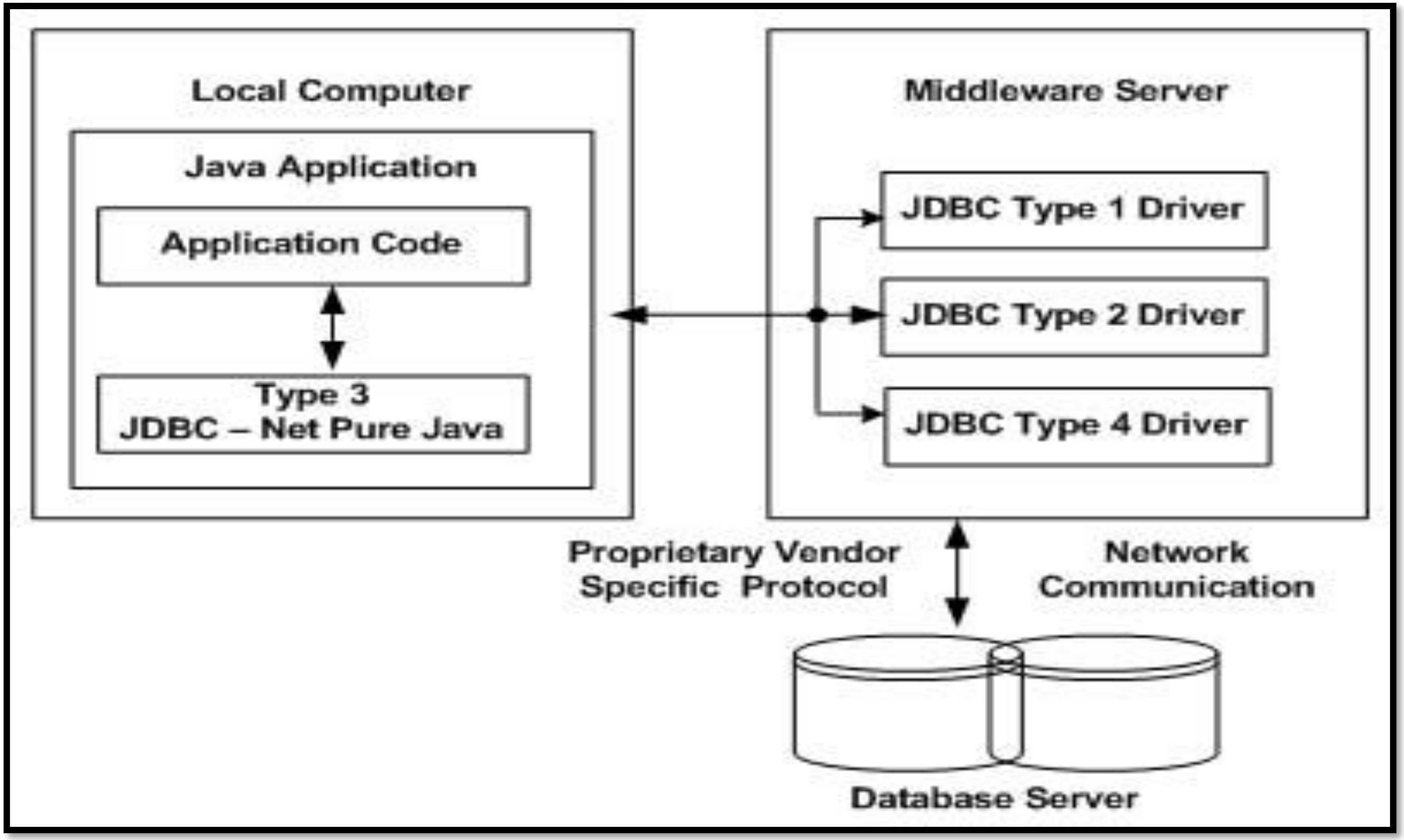
- Requires native libraries to be installed on client machines.
- Any bug in the Type-2 driver might crash the JVM.
- Increase the cost of the application in case it is run on different platforms.

TYPE-3 DRIVER
Pure Java Driver for Database Middleware

- Translates JDBC call to Direct server calls with the help of middleware server.
- Also known as Net Protocol Fully JAVA Driver



TYPE-3 DRIVER



TYPE-3 DRIVER

Advantages :

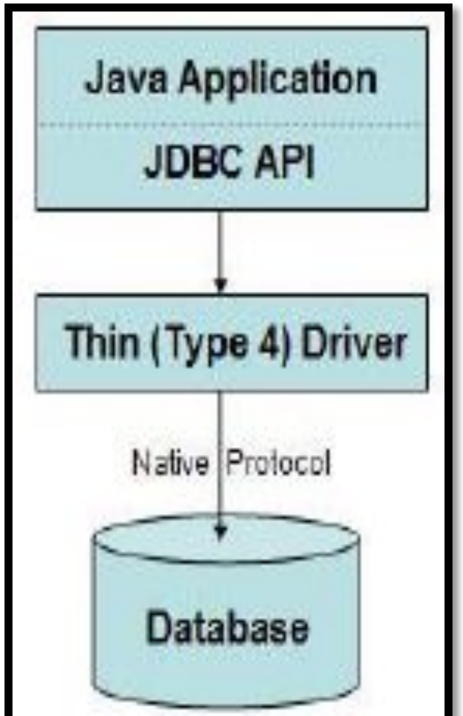
- Serves as all Java driver.
- Does not require any native library to be installed on the client machine.
- Ensures database independency.
- Does not provides database location information.
- Switch over from one database to another without changing the code.

Disadvantages:

- Perform the task slowly.
- Costlier.

TYPE- 4 DRIVER
Pure Java Driver

- Implements Database Protocol to Directly deal with the data base
- Do not require native libraries
- Translates JDBC class to direct database specific calls
- Also known as THIN Drivers.



TYPE- 4 DRIVER

Advantages:

- Pure Java driver.
- Does not require any native library to be installed on the client machine.
- Uses database server specific protocol.
- Does not require middleware server.

Disadvantages:

- It is DBMS vendor dependent

Which Driver should be used?

- If you are accessing one type of database, such as Oracle, Sybase, or IBM, the preferred driver type is 4.
- If your Java application is accessing multiple types of databases at the same time, type 3 is the preferred driver.
- Type 2 drivers are useful in situations where a type 3 or type 4 driver is not available yet for your database.
- The type 1 driver is not considered a deployment-level driver and is typically used for development and testing purposes only.

JDBC APIs

- JDBC APIs is a part of JDBC specification and provides a standard abstraction to use JDBC drivers……
- The JDBC API provides classes and interfaces that are used by Java Applications to communicate to databases.
- The JDBC driver communicates with a relational database for any requests made by a Java application by using the JDBC API.
- The JDBC drive not only processes the SQL commands, but also sends back the result of processing of these SQL commands.

JDBC APIs

The java.sql Package

- It also known as the JDBC core API.
- This package includes the interfaces and methods to perform JDBC core operations, such as creating and executing SQL queries.
- The java.sql package consists of the interfaces and classes that need to be implemented in an application to access a database.
  - Connection Management
  - Database access
  - Data Types
  - Exception and Warning.

JDBC APIs

The java.sql Package (cont…)

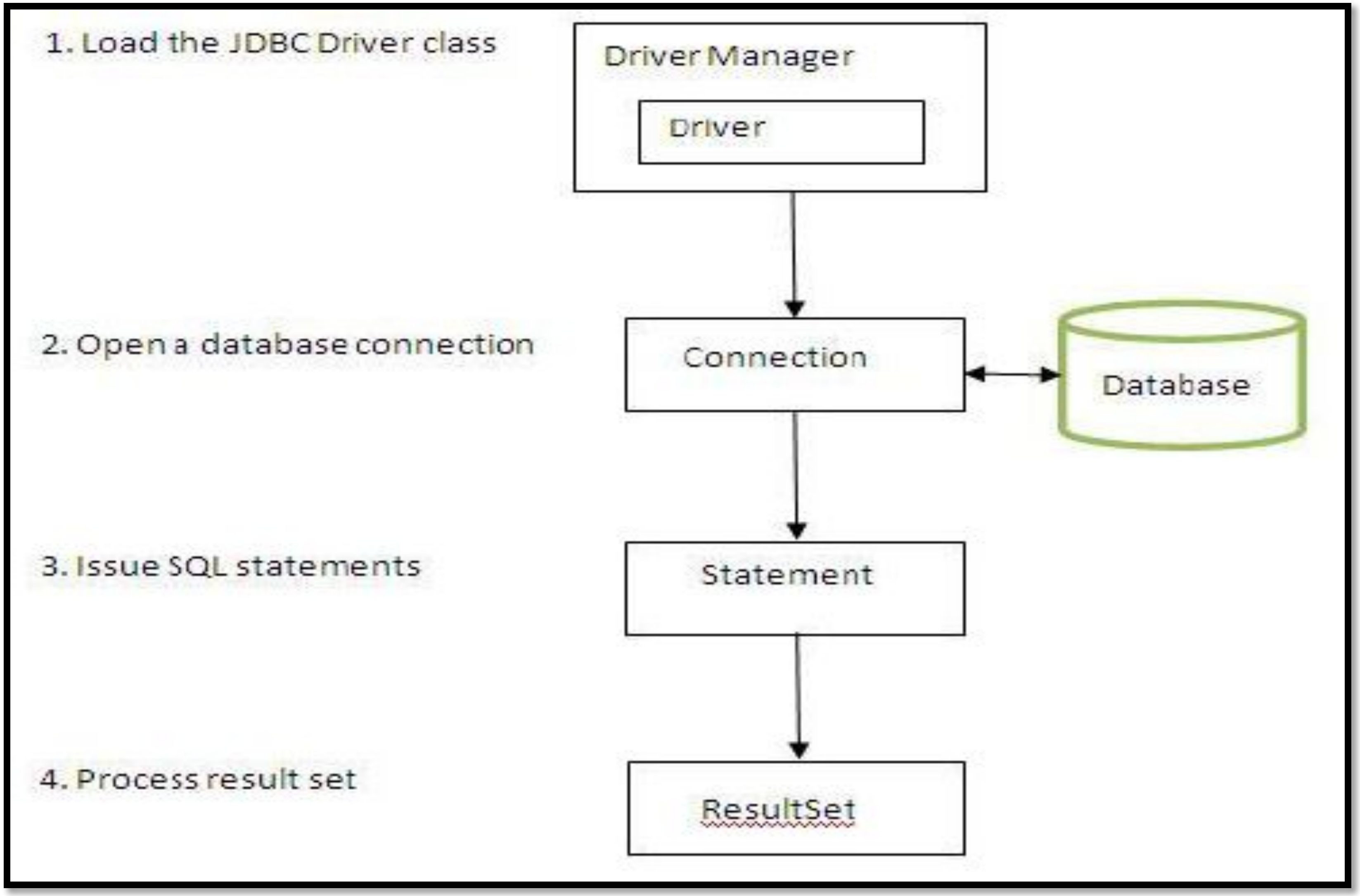
- Connection Management classes
  - Java.sql.Connection
  - Java.sql.Driver
  - Java.sql.DriverManager
  - Java.sql.DriverPropertyInfo
  - Java.sql.SQLPermission
- Database Access Interfaces
  - Java.sql.CallableStatement
  - Java.sql.PreparedStatement
  - Java.sql.ResultSet
  - Java.sql.Statement and java.sql.Savepoint

JDBC APIs

The JDBC process is divided into five routines.

1)Loading the JDBC driver
2)Connecting to the DBMS
3)Creating and executing a statement
4)Processing data returned by the DBMS
5)Terminating the connection with the DBMS

BASIC JDBC OPERATION



Overview of JDBC process

1) Loading the JDBC driver
   - The JDBC driver must be loaded before the J2EE component can connect to the DBMS.
   - Example for loading the JDBC driver
     - Class.forName(“sun.jdbc.odbc.JdbcOdbcDriver”);
     - Class.forName(“com.mysql.jdbc.Driver”);

Overview of JDBC process

2) Connecting to the DBMS

- DriverManager class provides different flavor of getConnection() methods to get the connection.
  - DriverManager.getConnection(String url);
  - DriverManager.getConnection(String url, String userName, String password);
  - DriverManager.getConnection(String url, Properties obj);

- Example of url String
  - “jdbc:odbc:dbName”
  - “jdbc:mysql://localhost:3306/dbName”

Overview of JDBC process

3) Creating and executing a statement

- The Statement object is used whenever a J2EE component needs to immediately execute a query without first having the query compiled.
- Statement object can be retrieve using Connection object.
- Example
  - connectionObj.createStatement();

Overview of JDBC process

3) Creating and executing a statement (cont…)

- Statement object has different methods to communicate with Database.
- Like
  - stmt.executeQuery(String query); this method use for simple select query.
  - stmt.executeUpdate(String query); this method use for INSERT, UPDATE & DELETE query.

Overview of JDBC process

4) Processing data returned by the DBMS

- executeQuery() method returns ResultSet object which contains virtual table with database record.
- We can interacte ResultSet object and can retrieve column information via column number or via column name.
- Example
  - rs.getString(1);
  - rs.getString(2);

  -----------------------------OR---------------------------

  - rs.getString(“FNAME”);
  - rs.getString(“LNAME”);

Overview of JDBC process

5) Terminating the connection with the DBMS

- The connection to the DBMS is terminated by using the close() method of the Connection object once the J2EE component is finished accessing the DBMS.
- You need to call the following method.
  - ConnectionObject.close();

Example of JDBC

```
import java.sql.*;

public class Main {

   public static void main(String[] args) {
     Connection conn = null;
     Statement stmt = null;
     Class.forName("com.mysql.jdbc.Driver");

       conn = DriverManager.getConnection ("jdbc:mysql://localhost:3306/employee,"root","");

      stmt = conn.createStatement();

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

Example of JDBC

```
while(rs.next()){
        int id  = rs.getInt(0);
       String nm = rs.getString(1);
       String dept = rs.getString(2);

        System.out.print("ID: " + id);
        System.out.print(" Name : " + nm);
       System.out.println(" Dept : " + dept);
     }

     rs.close();
     stmt.close();
     conn.close();
   } System.out.println("Goodbye!");
}
```

Common JDBC Components

- **DriverManager:** This class manages a list of database drivers. Matches connection requests from the java application with the proper database driver. The driver will be used to establish a database Connection.

- **Driver:** This interface handles the communications with the database server. You will interact directly with Driver objects very rarely. Instead, you use DriverManager objects, which manages objects of this type. It also abstracts the details associated with working with Driver objects.

Common JDBC Components

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

Statement Interface

- The Statement interface defines a standard abstraction to execute the SQL statements requested by a user and return the results by using the ResultSet Object.
- Once you've created a Statement object, you can then use it to execute an SQL statement with one of its three execute methods.
  - **execute**
  - **executeQuery**
  - **executeUpdate**

Statement Interface

- **boolean execute (String SQL)**: Returns a boolean value of true if a ResultSet object can be retrieved; otherwise, it returns false. Use this method to execute SQL DDL statements.

- **int executeUpdate (String SQL)**: Returns the number of rows affected by the execution of the SQL statement. Use this method to execute SQL statements for which you expect to get a number of rows affected - for example, an INSERT, UPDATE, or DELETE statement.

- **ResultSet executeQuery (String SQL)**: Returns a ResultSet object. Use this method when you expect to get a result set, as you would with a SELECT statement.

Statement Interface

- For Retrieving data :
  - rs = stmt.executeQuery("select * from emp");

- Inserting a Record
  - stmt.executeUpdate("insert into emp values (1,’john',’cse’)");

Statement Interface

- Updating aRecord
  - stmt.executeUpdate("update emp set dept = ‘account’ where emp_no = 2");
- Deleting Record
  - stmt.executeUpdate("delete from emp where emp_no = 1");
- Deleting Table
  - stmt.execute("drop table emp");

Statement Interface

- Before you can use a Statement object to execute a SQL statement, you need to create one using the Connection object's createStatement( ) method.
- There are three type of Statements interfaces are available.
  - Statement
  - PreparedStatement
  - CallableStatement

Statement Interface

- Execution of Statement object

1. The executeXXX() method is invoked on the Statement object by passing the SQL statement as parameter.
2. It submit the Query to the Database.
3. The DBMS compile the given SQL statement.
4. The DBMS execute the given SQL statement.
5. If the SQL statement is SELECT statement then the database caches the results of the SQL statement in the buffer.
6. The results are sent to the Statement object.
7. Finally, the response is sent to the Java application in the form of ResultSet.

PreparedStatement

- Execution of PreparedStatement object

1. The prepareStatement() method of the connection object is used to get the object of the PreparedStatement interface.
2. The Connection object submits the given SQL statement to the database.
3. The database compiles the given SQL statement.
4. An execution plan is prepared by the database to execute the SQL statement.
5. The database stores the execution plan with a unique ID and returns the identity to the Connection object.

PreparedStatement

- Execution of PreparedStatement object

- In normal Statement object SQL query will always compile when it is going to executed.
- If same query will compile more than one time than it will become overhead.
- SQL query can be precompiled and executed using PreparedStatement object.
- It also has method like executeQuery() and executeUpdate() methods.

PreparedStatement

- String query = “SELECT * FROM STUDENT WHERE BRANCH = ?”;
- PreparedStatement statement = conn.preparedStatement(query);

- statement.setString(1, “CE”);
- rs = statement.executeQuery();
- statement.setString(1, “IT”);
- rs = statement.executeQuery();

PreparedStatement

- String query = “SELECT * FROM STUDENT WHERE BRANCH = ? AND SEMESTER = ?”;
- PreparedStatement statement = conn.preparedStatement(query);

- statement.setString(1, “CE”);
- statement.setInt(2, 6);
- rs = statement.executeQuery();

PreparedStatement

Advantages

- Improves the performance of an application as compared to the Statement object that executes the same query multiple times.
- Provide a programmatic approach to set the values.

CallableStatement

- The CallableStatement interface extends the PreparedStatement interface and also provides support for both input as well as output parameters.
- It provides a standard abstraction for all the data sources to call stored procedures and functions, irrespective of the vendor of the data source.
- This interface is used to access, invoke and retrieve the results of SQL stored procedures, functions, and cursors.

CallableStatement

- Broad-level steps to use the CallableStatement

1. Create the CallableStatement Object
2. Setting the values of the parameters
3. Registering the OUT parameters type
4. Executing the procedure of function
5. Retrieving the parameter values.

CallableStatement

- A stored procedure encapsulates the values of the following types of parameters.
- **IN-** A parameter whose value is unknown when the SQL statement is created. You bind values to IN parameters with the setXXX() methods
- **OUT –** A parameter whose value is supplied by the SQL statement it returns. You retrieve values from theOUT parameters with the getXXX() methods.
- **IN OUT** – A parameter that provides both input and output values. You bind variables with the setXXX() methods and retrieve values with the getXXX() methods.

CallableStatement

- **How to create procedure in oracle**

```
Create or replace procedure insertData (rno number, name varchar2, address varchar2) is
Begin
insert into student values(rno, name, address);
End;
```

CallableStatement

- How to write down code…
- CallableStatement cs= con.prepareCall ("{call insertData (?,?,?)}");

- **// set IN parameters**

1. cs.setInt(1, 12);
2. cs.setInt(2, “Jay”);
3. cs.setString(3, “xyz....");
4. cs.execute();

CallableStatement

- **OUT parameter**

```
Create or replace procedure getBalance (acno
   number, bal OUT number) is
Begin
select bal into amt from bank where accno=acno;
End;
```

CallableStatement

- CallableStatement (cont…)
- OUT parameter (cont…)

```
CallableStatement cs= con.prepareCall("{call
getBalance(?,?)}");
    cs.setInt(1, s[0]));
    cs.execute();
    System.out.println("Balance : "+ cs.getDouble(2));

con.close();
```

Resultset Interface

- A ResultSet object maintains a cursor that points to the current row in the result set. The term "result set" refers to the row and column data contained in a ResultSet object.
- The methods of the ResultSet interface can be broken down into three categories −
  - **Navigational methods:** Used to move the cursor around.
  - **Get methods:** Used to view the data in the columns of the current row.
  - **Update methods:** Used to update the data in the columns of the current row.

Resultset Interface

- **createStatement(RSType, RSConcurrency);**

- Type of ResultSet
  – ResultSet.TYPE_FORWARD_ONLY
  – ResultSet.TYPE_SCROLL_INSENSITIVE
  – ResultSet.TYPE_SCROLL_SENSITIVE.

- Concurrency of ResultSet
  – ResultSet.CONCUR_READ_ONLY
  – ResultSet.CONCUR_UPDATABLE

Resultset Interface

- **createStatement(RSType, RSConcurrency);**

- Type of ResultSet
  - ResultSet.TYPE_FORWARD_ONLY
  - ResultSet.TYPE_SCROLL_INSENSITIVE
  - ResultSet.TYPE_SCROLL_SENSITIVE.

- Concurrency of ResultSet
  - ResultSet.CONCUR_READ_ONLY
  - ResultSet.CONCUR_UPDATABLE
- Example

```
Statement stmt = conn.createStatement(
ResultSet.TYPE_FORWARD_ONLY,
ResultSet.CONCUR_READ_ONLY);
```

Scrollable ResultSet

- Navigating a Result Set
- Methods
  - beforeFirst()
  - afterLast()
  - first()
  - last()
  - absolute(int row)
  - relative(int row)
  - previous()
  - next()
  - getRow()

Scrollable ResultSet

- Example
  - rs.absolute(2);
  - rs.absolute(-4);
  - rs.relative(-3);
  - rs.relative(2);
- Example
  - srs.absolute(4);
  - int rowNum = srs.getRow();
  - srs.relative(-3);
  - int rowNum = srs.getRow();
  - srs.relative(2);
  - int rowNum = srs.getRow();

Updatable Result Sets

- Java has ability to update rows in a result set using methods in the Java programming language rather than having to send an SQL command.
- create a ResultSet object that is updatable.
- supply the ResultSet constant CONCUR_UPDATABLE to the createStatement method.
- Ex.
- Statement stmt =
  con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
  ResultSet.CONCUR_UPDATABLE);

Updatable Result Sets

- stmt.executeUpdate("UPDATE CUSTOMERS SET BALANCE = 90000 WHERE CUST_ID = 20");
- **Another way**
- **rs.updateFloat(“BALANCE ”,90000);**
- The Resultet. updateXXX methods take two parameters:
  - the column to update
  - new value to put in that column.

- There is a different updateXXX method for updating each datatype like updateString , updateInt , and so on.

Updatable Result Sets

- Example :
- rs.updateFloat(“BALANCE ”,90000);
- rs.cancelRowUpdates();

Updatable Result Sets

- Inserting rows using programming

```
 rs.moveToInsertRow();
rs.updateInt(“emp_no", 4);
rs.updateInt(“emp_name", “john”);
rs.updateInt(“emp_dept", “it”);
rs.insertRow();
```

Updatable Result Sets

- deleting rows using programming
- **rs.deleteRow();**

Batch Processing

- Batch Processing allows you to group related SQL statements into a batch and submit them with one call to the database.
- The **addBatch()** method of Statement, PreparedStatement, and CallableStatement is used to add individual statements to the batch.
- The **executeBatch()** is used to start the execution of all the statements grouped together.
- The **executeBatch()** returns an array of integers, and each element of the array represents the update count for the respective update statement.
- Just as you can add statements to a batch for processing, you can remove them with the **clearBatch()** method. This method removes all the statements you added with the addBatch() method.

Example

```
Statement stmt = conn.createStatement();
String SQL = "INSERT INTO Emp VALUES(2,‘Sachin',  ‘account’)";
stmt.addBatch(SQL);

String SQL = "INSERT INTO Emp VALUES(3,'Raj', ‘hr’)";
stmt.addBatch(SQL);

String SQL = "UPDATE Emp  SET name = ‘Kumar’  WHERE emp_id =
    10";
stmt.addBatch(SQL);

int[] count = stmt.executeBatch();
```

Example

```
String SQL = "INSERT INTO Emp VALUES(?, ?, ?)”;
PreparedStatemen pstmt = conn.prepareStatement(SQL);
pstmt.setInt( 1, 40 );
pstmt.setString( 2, “smith" );
pstmt.setString( 3, “finance" );
pstmt.addBatch();

pstmt.setInt( 1, 401 );
pstmt.setString( 2, “Pawan" );
pstmt.setString( 3, “admin" );
pstmt.addBatch();
int[] count = stmt.executeBatch();
```

ResultSetMetaData

- The metadata means data about data i.e. we can get further information from the data.
- ResultSetMetaData interface is useful for providing methods to get metadata from the ResultSet object.
- This interface provides an object that can be used to get information about the types and properties of columns in data.

ResultSetMetaData

- Interface is
  - ResultSetMetaData
- Methods of resultset
  - **getColumnCount()**
  - **getColumnName(int index)**
  - **getColumnTypeName(int index)**
  - **getTableName()**
  - **isAutoIncrement(int index)**

- Example :

```
ResultSetMetaData rsmd=rs.getMetaData();

System.out.println(rsmd.getColumnCount());
System.out.println(rsmd.getColumnName(1));
System.out.println(rsmd.getColumnTypeName(1));
```

Database Metadata

- Interface :
  - DatabaseMetadata
- This interface provides information about the database to get information about capabilities and version of database drivers, attributes etc.
- DatabaseMetaData md = con.getMetaData();

Database Metadata

- Methods :
  - getDatabaseProductName()
  - getDatabaseProductVersion()
  - getDriverName()
  - getDriverVersion()
- Example
  - DatabaseMetaData dm = con.getMetaData();
  - System.out.print(dm.getDriverName());
  - System.out.print(dm. getDatabaseProductName());

Error handling and SQL exception

- Exception handling allows you to handle exceptional conditions such as program-defined errors in a controlled fashion.
- When an exception condition occurs, an exception is thrown. The term thrown means that current program execution stops, and the control is redirected to the nearest applicable catch clause. If no applicable catch clause exists, then the program's execution ends.
- JDBC Exception handling is very similar to the Java Exception handling but for JDBC, the most common exception you'll deal with is **java.sql.SQLException.**

Error handling and SQL exception

SQLException Methods

- The passed SQLException object has the following methods available for retrieving additional information about the exception :
  - **getErrorCode( ) -** Gets the error number associated with the exception.
  - **getMessage( )-** Gets the JDBC driver's error message for an error, handled by the driver or gets the Oracle error number and message for a database error.
  - **printStackTrace( )** - Prints the current exception, or throwable.

Error handling and SQL exception

Example

```
    try{
       }
    catch(SQLException se)
    {    se.printStackTrace();
    }
    catch(Exception e)
    {     e.printStackTrace();
    }
    Finally
    {
    }
```

JDBC data Types

- The JDBC driver converts the Java data type to the appropriate JDBC type, before sending it to the database. It uses a default mapping for most data types.
- For example, **a Java int is converted to an SQL INTEGER**. Default mappings were created to provide consistency between drivers.
- The following table summarizes the default JDBC data type that the Java data type is converted to, when you call the setXXX() method.
- ResultSet object provides corresponding getXXX() method for each data type to retrieve column value. Each method can be used with column name or by its ordinal position.

JDBC data Types

SQL	JDBC/Java	setXXX	getXXX
VARCHAR	java.lang.String	setString	getString
CHAR	java.lang.String	setString	getString
LONGVARCHAR	java.lang.String	setString	getString
BIT	Boolean	setBoolean	getBoolean
NUMERIC	java.math.BigDecimal	setBigDecimal	getBigDecimal
INTEGER	Int	setInt	getInt
FLOAT	Float	setFloat	getFloat
DOUBLE	Double	setDouble	getDouble
DATE	java.sql.Date	setDate	getDate
TIME	java.sql.Date	setDate	getDate
CLOB	java.sql.Clob	setClob	getClob
BLOB	java.sql.Blob	setBlob	getBlob

Transaction Management

- Transactions enable you to control if, and when, changes are applied to the database. It treats a single SQL statement or a group of SQL statements as one logical unit, and if any statement fails, the whole transaction fails.

- If your JDBC Connection is in *auto-commit* mode, which it is by default, then every SQL statement is committed to the database upon its completion.

- To enable manual- transaction support instead of the *auto-commit* mode that the JDBC driver uses by default, use the Connection object's **setAutoCommit()** method. If you pass a boolean false to setAutoCommit( ), you turn off auto-commit. You can pass a boolean true to turn it back on again.

Transaction Management

- to turn off auto-commit
  - **conn.setAutoCommit(false);**
- Once you are done with your changes and you want to commit the changes then call commit() method on connection object as follows
  - **conn.commit( );**
- Otherwise, to roll back updates to the database made using the Connection named conn, use the following code
  - **conn.rollback( );**

Transaction Management

```
Try
{
     conn.setAutoCommit(false);
     Statement stmt = conn.createStatement();
     String SQL = "INSERT INTO Employees VALUES (106, 20, 'Rita', 'Tez')";
     stmt.executeUpdate(SQL);
     String SQL = "INSERTED IN Employees VALUES (107, 22, 'Sita', 'Singh')";
     stmt.executeUpdate(SQL);
    conn.commit();
 }
catch(SQLException se)
{
conn.rollback();
}
```

Transaction Management

- Using Savepoints
- The Connection object has two new methods that help you manage savepoints
  1. **setSavepoint(String savepointName):** Defines a new savepoint. It also returns a Savepoint object.
  2. **releaseSavepoint(Savepoint savepointName):** Deletes a savepoint
  3. There is one **rollback (String savepointName)** method, which rolls back work to the specified savepoint.

example

```
try{
conn.setAutoCommit(false);
Statement stmt = conn.createStatement();

Savepoint savepoint1  = conn.setSavepoint("Savepoint1");
 String SQL = "INSERT INTO Employees VALUES (106, ‘Rita', ‘it')";

stmt.executeUpdate(SQL);
conn.commit();
}
catch(SQLException se)
{
conn.rollback(savepoint1);
}
```

```
package Edureka;
import java.sql.*;
import java.sql.DriverManager;
public class Example {
// JDBC driver name and database URL
static final String JDBC_DRIVER = "com.mysql.jdbc.Driver";
static final String DB_URL = "jdbc:mysql://localhost/emp";
//  Database credentials
static final String USER = "root";
static final String PASS = "";
public static void main(String[] args) {
Connection conn = null;
Statement stmt = null;
try{
//STEP 2: Register JDBC driver
Class.forName("com.mysql.cj.jdbc.Driver");
//STEP 3: Open a connection
System.out.println("Connecting to database...");
conn = DriverManager.getConnection(DB_URL,"root","");
//STEP 4: Execute a query
System.out.println("Creating statement...");
stmt = conn.createStatement();
String sql;
sql = "SELECT id, first, last, age FROM Employees";
ResultSet rs = stmt.executeQuery(sql);
//STEP 5: Extract data from result set
while(rs.next()){
//Retrieve by column name
int id  = rs.getInt("id");
int age = rs.getInt("age");
String first = rs.getString("first");
String last = rs.getString("last");
//Display values
System.out.print("ID: " + id);
System.out.print(", Age: " + age);
System.out.print(", First: " + first);
System.out.println(", Last: " + last);
}
//STEP 6: Clean-up environment
rs.close();
stmt.close();
conn.close();
}catch(SQLException se){
//Handle errors for JDBC
se.printStackTrace();
}catch(Exception e){
//Handle errors for Class.forName
e.printStackTrace();
}finally{
//finally block used to close resources
try{
if(stmt!=null)
stmt.close()
}catch(SQLException se2){
}// nothing can be done
try{
if(conn!=null)
conn.close();
}catch(SQLException se){
se.printStackTrace();
}//end finally try
}//end try
System.out.println("Goodbye!");
}//end main
} // end Example
```

THANK YOU !