CORE JAVA

By : Kalpesh Chauhan

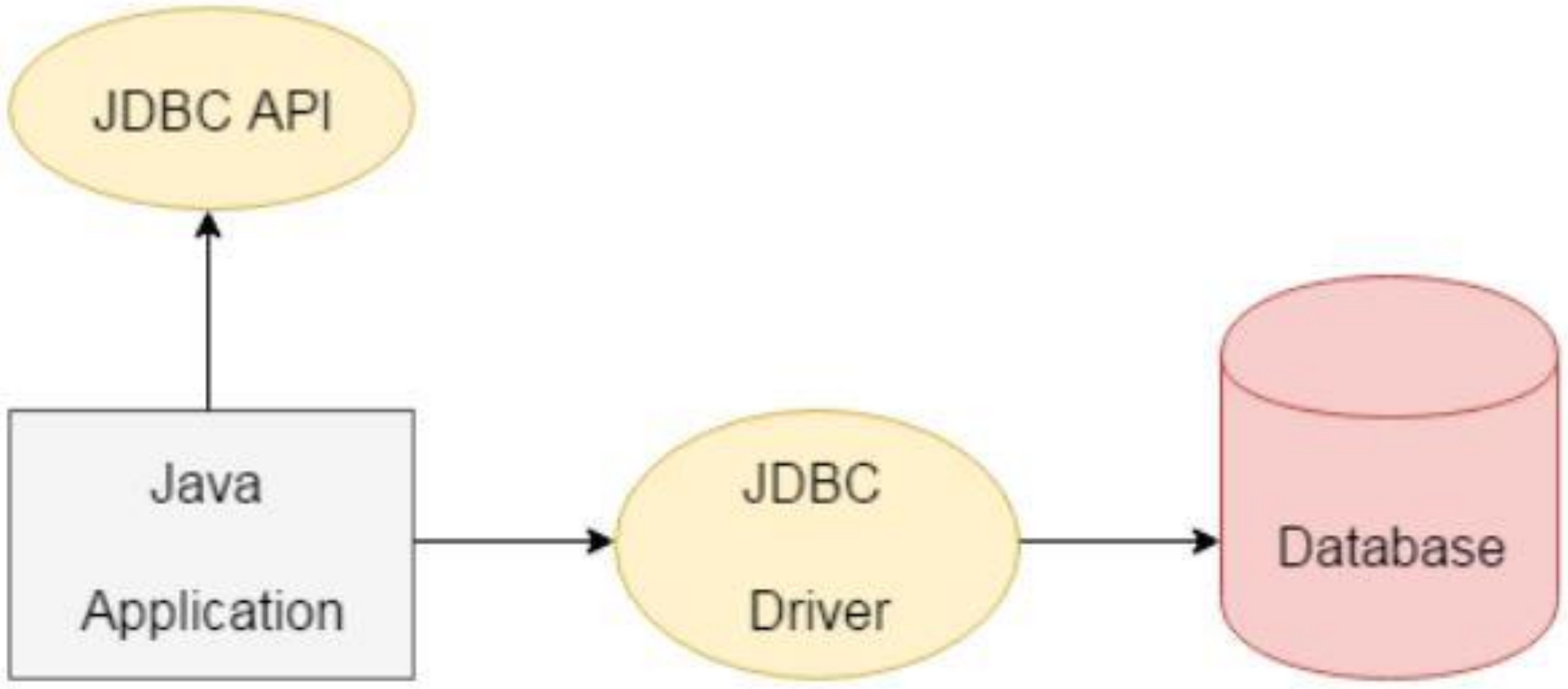
JDBC

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

1. JDBC-ODBC Bridge Driver,
2. Native Driver,
3. Network Protocol Driver, and
4. Thin Driver

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

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


JDBC API
Java Application
JDBC Driver
Database

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

Driver interface

Connection interface

Statement interface

PreparedStatement interface

CallableStatement interface

ResultSet interface

ResultSetMetaData interface

DatabaseMetaData interface

RowSet interface

WHY SHOULD WE USE JDBC

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

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

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

WHAT IS API

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

JDBC DRIVER

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

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

JDBC-ODBC BRIDGE DRIVER

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

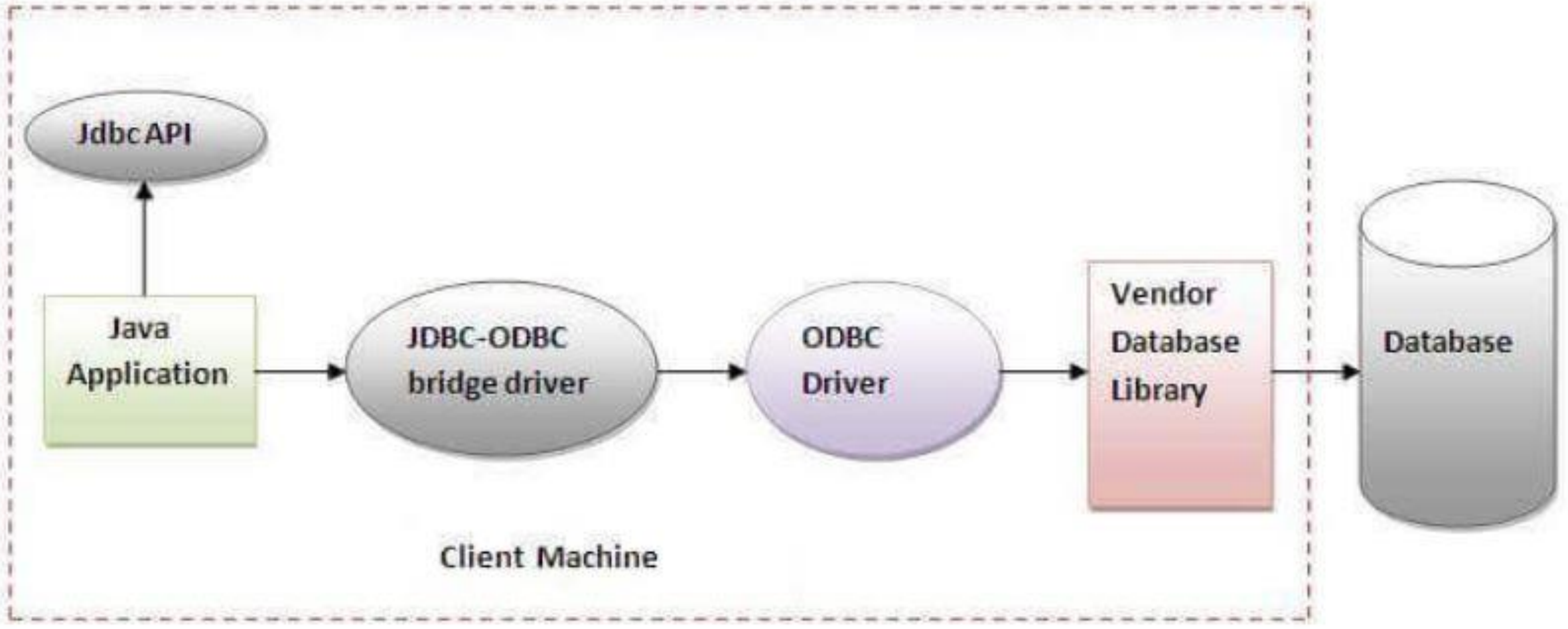


Figure- JDBC-ODBC Bridge Driver

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

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

Advantages:

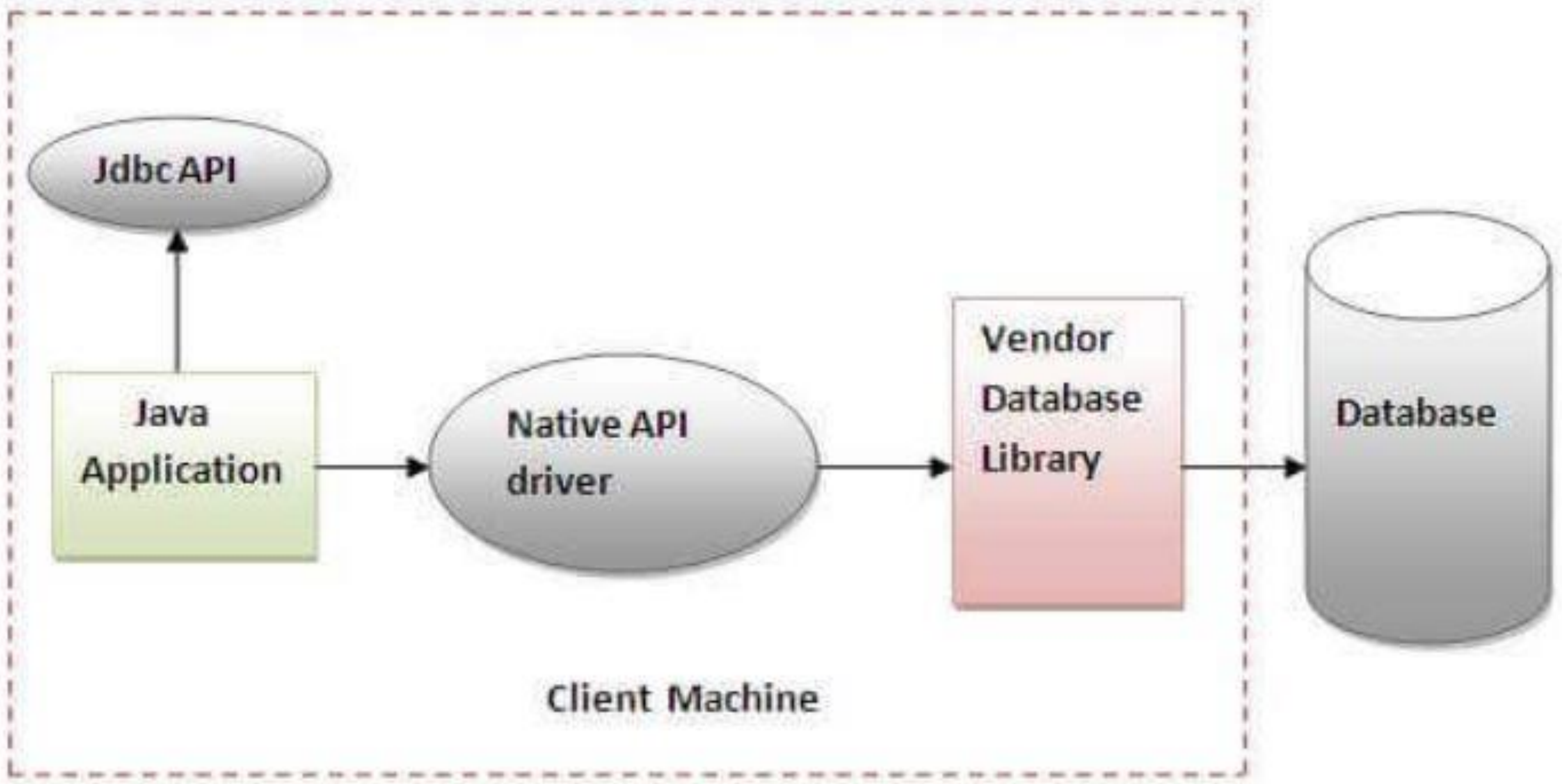
1. easy to use.
2. can be easily connected to any database.

Disadvantages:

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

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.


Jdbc API
Java Application
Native API driver
Vendor Database Library
Database
Client Machine

Advantage:

1. performance upgraded than JDBC-ODBC bridge driver.

Disadvantage:

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

NETWORK PROTOCOL DRIVER

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

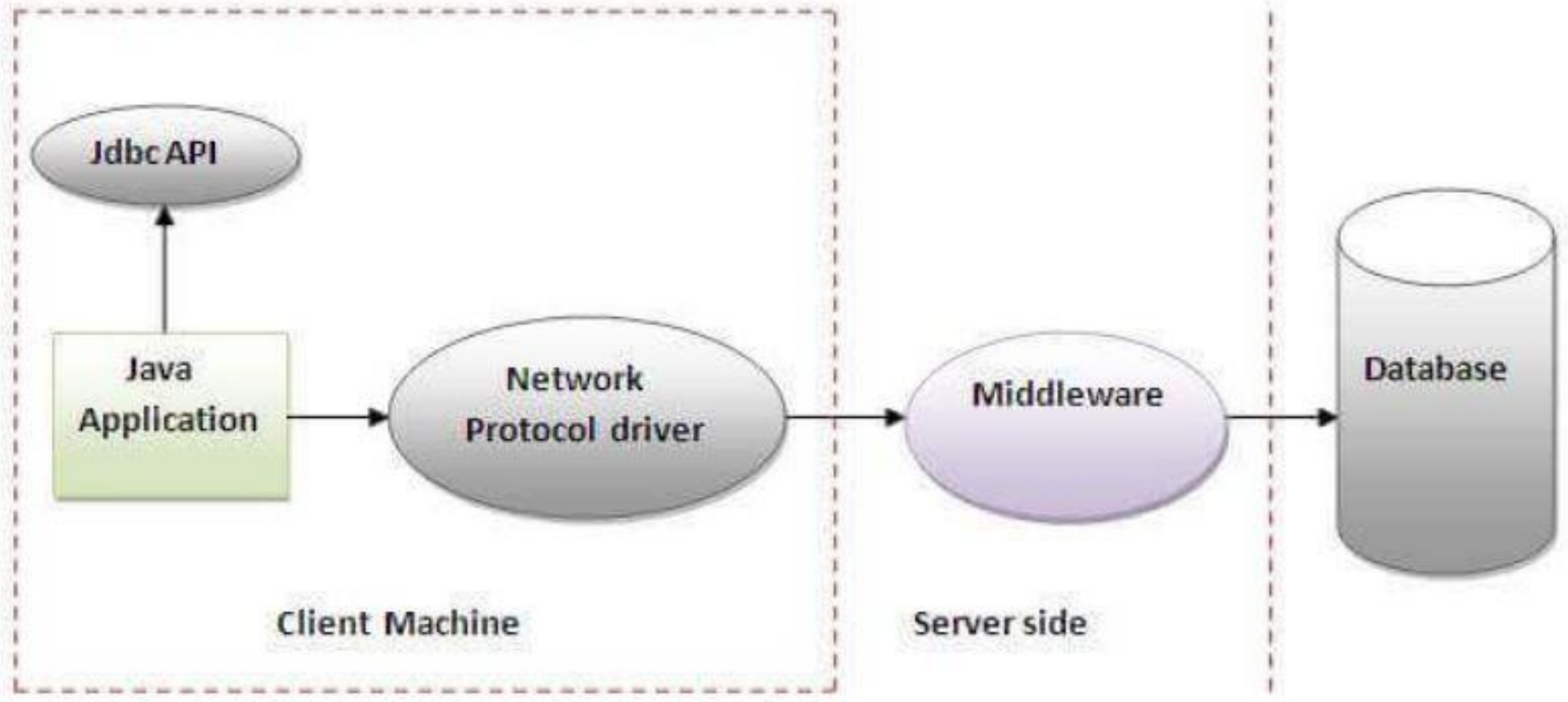


Figure- Network Protocol Driver

Advantage:

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

Disadvantages:

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

THIN DRIVER

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

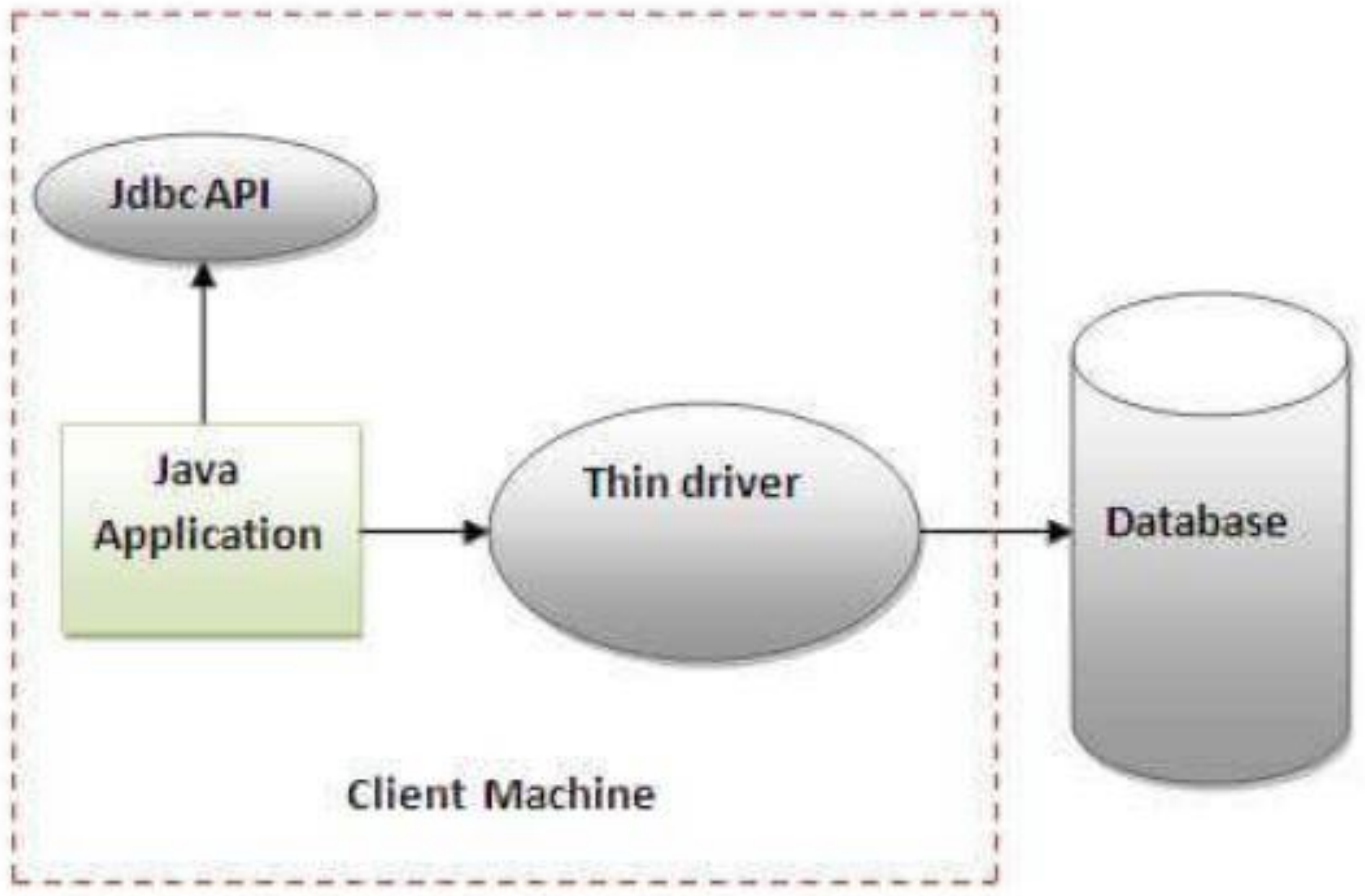


Figure- Thin Driver

Advantage:

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

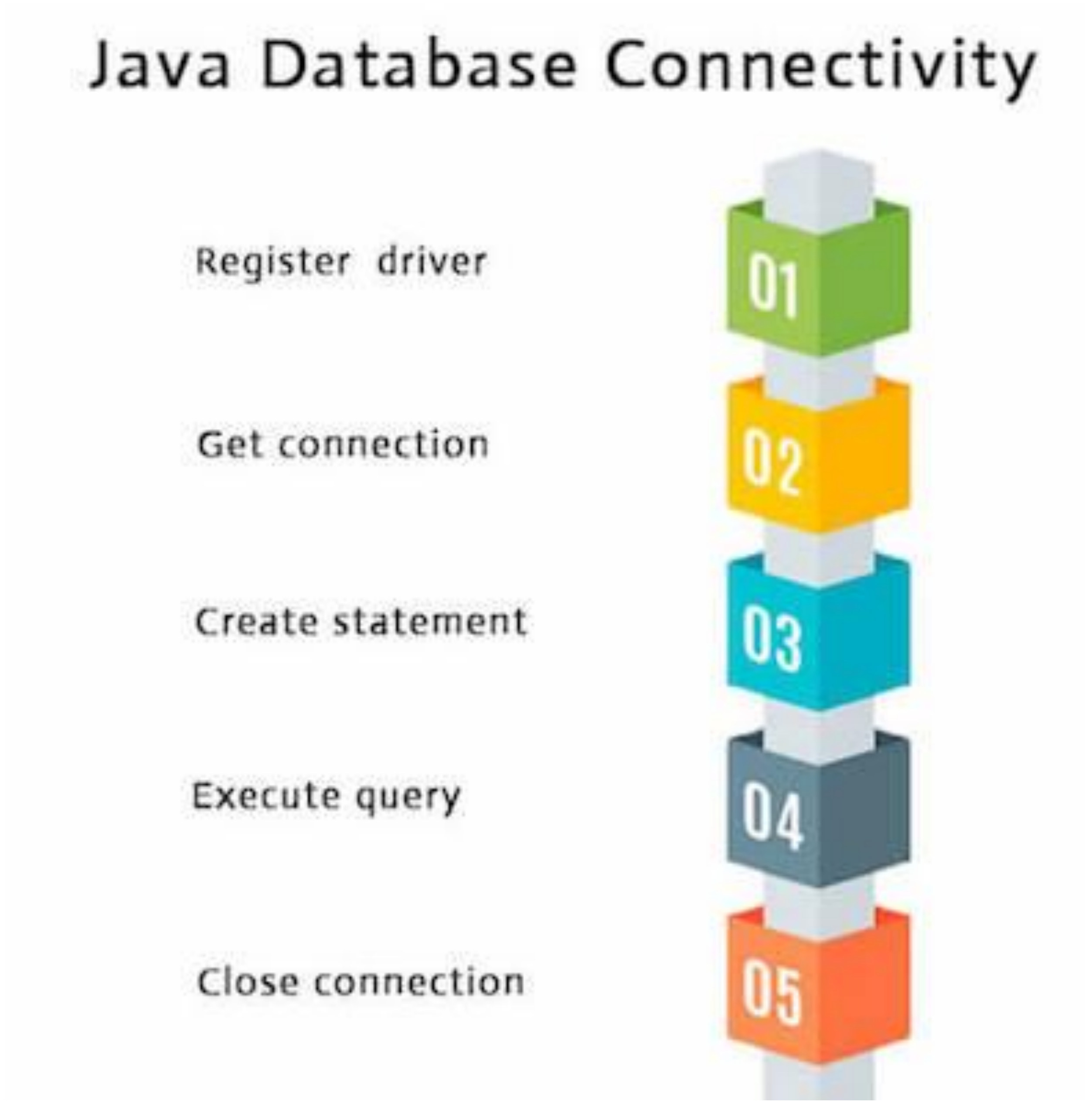
Disadvantage:

1. Drivers depend on the Database.

JAVA DATABASE CONNECTIVITY WITH 5 STEPS

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

1. Register the Driver class
2. Create connection
3. Create statement
4. Execute queries
5. Close connection


Java Database Connectivity
Register driver
01
Get connection
02
Create statement
03
Execute query
04
Close connection
05

REGISTER THE DRIVER CLASS

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

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

EXAMPLE TO REGISTER THE ORACLEDRIVER CLASS

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

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

CREATE THE CONNECTION OBJECT

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

public static Connection getConnection(String url,String name,String password)

throws SQLException

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.

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

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.

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

while(rs.next()){
System.out.println(rs.getInt(1)+" "+rs.getString(2));
}
```

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.

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

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

JAVA DATABASE CONNECTIVITY WITH MYSQL

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

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

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

EXAMPLE

```
import java.sql.*;
class MysqlCon{
public static void main(String args[]){
try{
Class.forName("com.mysql.jdbc.Driver");
Connection con=DriverManager.getConnection( "jdbc:mysql://localhost:3306/sonoo",
"root","root");
Statement stmt=con.createStatement();

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

while(rs.next())

System.out.println(rs.getInt(1)+"  "+rs.getString(2)+"  "+rs.getString(3));

con.close();

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

}

}
```

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

Paste the mysqlconnector.jar file in jre/lib/ext folder

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

DRIVERMANAGER CLASS

The DriverManager class acts as an interface between user and drivers. It keeps track of the drivers that are available and handles establishing a connection between a database and the appropriate driver. The DriverManager class maintains a list of Driver classes that have registered themselves by calling the method DriverManager.registerDriver().

METHODS

Method	Description
1) public static void registerDriver(Driver driver):	is used to register the given driver with DriverManager.
2) public static void deregisterDriver(Driver driver):	is used to deregister the given driver (drop the driver from the list) with DriverManager.
3) public static Connection getConnection(String url):	is used to establish the connection with the specified url.
4) public static Connection getConnection(String url,String userName,String password):	is used to establish the connection with the specified url, username and password.

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.

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.

The important methods of Statement interface are as follows:

1) public ResultSet executeQuery(String sql): is used to execute SELECT query. It returns the object of ResultSet.

2) public int executeUpdate(String sql): is used to execute specified query, it may be create, drop, insert, update, delete etc.

3) public boolean execute(String sql): is used to execute queries that may return multiple results.

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

RESULTSET INTERFACE

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

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:

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

Method	Use
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 or negative.
7) public int getInt(int columnIndex):	is used to return the data of specified column index of the current row as int.
8) public int getInt(String columnName):	is used to return the data of specified column name of the current row as int.
9) public String getString(int columnIndex):	is used to return the data of specified column index of the current row as String.
10) public String getString(String columnName):	is used to return the data of specified column name of the current row as String.

PREPAREDSTATEMENT INTERFACE

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

Let's see the example of parameterized query:

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.

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

EXAMPLE

```
import java.sql.*;
class InsertPrepared{
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 stmt=con.prepareStatement("insert into Emp values(?,?)");

stmt.setInt(1,101);//1 specifies the first parameter in the query

stmt.setString(2,"Ratan");

int i=stmt.executeUpdate();

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


con.close();
```

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


}
}
```

JAVA RESULTSETMETADATA INTERFACE

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

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

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

```
import java.sql.*;

class Rsmd{

public static void main(String args[]){

try{

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

Connection con=DriverManager.getConnection(

"jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
PreparedStatement ps=con.prepareStatement("select * from emp");
ResultSet rs=ps.executeQuery();
ResultSetMetaData rsmd=rs.getMetaData();

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

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

JAVA DATABASEMETADATA INTERFACE

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

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.

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

Connection con=DriverManager.getConnection(
"jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
DatabaseMetaData dbmd=con.getMetaData();
System.out.println("Driver Name: "+dbmd.getDriverName());
System.out.println("Driver Version: "+dbmd.getDriverVersion());
System.out.println("UserName: "+dbmd.getUserName());
System.out.println("Database Product Name: "+dbmd.getDatabaseProductName());
System.out.println("Database Product Version: "+dbmd.getDatabaseProductVersion());

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

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:

public void setBinaryStream(**int** paramIndex,InputStream stream)

throws SQLException

public void setBinaryStream(**int** paramIndex,InputStream stream,**long** length)

throws SQLException

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

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

EXAMPLE

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

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

ps.setString(1,"sonoo");

FileInputStream fin=new FileInputStream("d:\\g.jpg");

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

int i=ps.executeUpdate();

System.out.println(i+" records affected"
con.close();
}catch (Exception e) {e.printStackTrace();}
}
}
```

EXAMPLE TO RETRIEVE IMAGE FROM 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.

EXAMPLE

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

```
PreparedStatement ps=con.prepareStatement("select * from imgtable");
ResultSet rs=ps.executeQuery();
if(rs.next()){//now on 1st row
Blob b=rs.getBlob(2);//2 means 2nd column data
byte barr[]=b.getBytes(1,(int)b.length());//1 means first image
FileOutputStream fout=new FileOutputStream("d:\\sonoo.jpg");
fout.write(barr);
fout.close();
}//end of if
System.out.println("ok");
con.close();
}catch (Exception e) {e.printStackTrace();  }
}
}
```

TRANSACTION

TRANSACTION MANAGEMENT IN JDBC

Transaction represents **a single unit of work**.

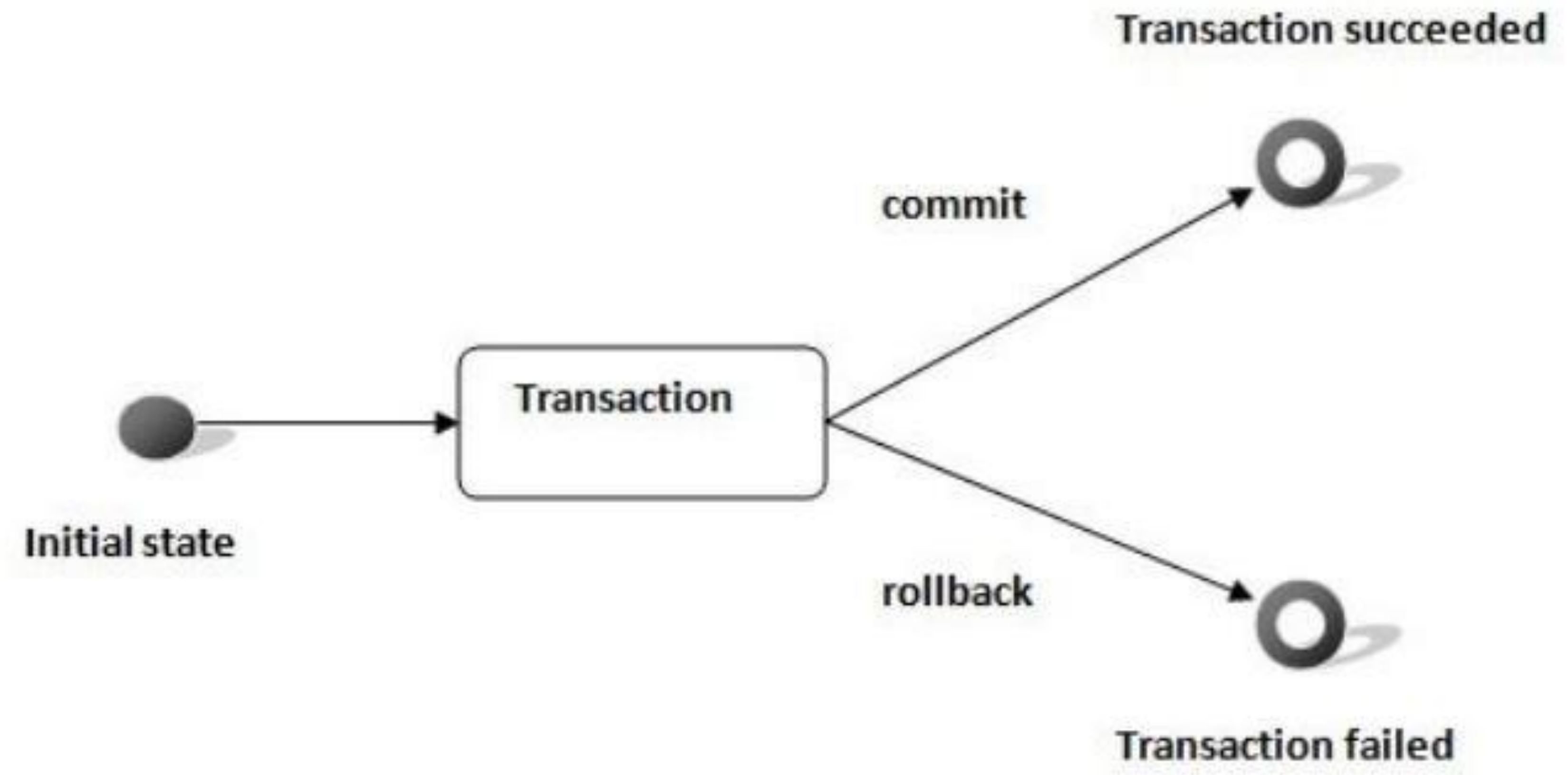
The ACID properties describes the transaction management well. ACID stands for Atomicity, Consistency, isolation and durability.

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.


Transaction succeeded
commit
Transaction
Initial state
rollback
Transaction failed

Method	Description
void setAutoCommit(boolean status)	It is true bydefault means each transaction is committed bydefault.
void commit()	commits the transaction.
void rollback()	cancels the transaction.

EXAMPLE

```
import java.sql.*;
class FetchRecords{
public static void main(String args[])throws Exception{
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe",
"system","oracle");
con.setAutoCommit(false);
Statement stmt=con.createStatement();
stmt.executeUpdate("insert into user420 values(190,'abhi',40000)");
stmt.executeUpdate("insert into user420 values(191,'umesh',50000)");

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

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.

Method	Description
void addBatch(String query)	It adds query into batch.
int[] executeBatch()	It executes the batch of queries.

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

1. Load the driver class
2. Create Connection
3. Create Statement
4. Add query in the batch
5. Execute Batch
6. Close Connection

```
import java.sql.*;

class FetchRecords{

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

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

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

con.setAutoCommit(false);
Statement stmt=con.createStatement();
stmt.addBatch("insert into user420 values(190,'abhi',40000)");
stmt.addBatch("insert into user420 values(191,'umesh',50000)");
stmt.executeBatch();//executing the batch
con.commit();
con.close();
}}
```

CONTINUE IN NEXT UNIT.....